Michaela Klopf, BSc

Congruent number elliptic curves of high rank

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Mathematical Computer Science

submitted to

Graz University of Technology

Supervisor
O.Univ.-Prof. Dr.phil. Robert Tichy

Institute of Analysis and Computational Number Theory (Math A)

Graz, November 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

The aim of this thesis is to find congruent number elliptic curves of high rank using an algorithm implemented in SageMath. First we give an introduction to elliptic curves and we consider the congruent number problem, which has a very long history in mathematics. This problem asks if a given number n is the area of a right-angled triangle with rational sides. Afterwards we discuss the relation between congruent numbers and elliptic curves. Next we give a general idea of finding elliptic curves of relatively high rank and we present our approach for finding high rank congruent number elliptic curves, which is based on this general idea. Furthermore we state rank records of elliptic curves in general and congruent number elliptic curves found so far. We also present our results which are congruent number elliptic curves of rank six. Further down the line we describe the functions we implemented to find these curves. Then we concentrate on applications of elliptic curves of high rank in cryptography such as cryptosystems that are based on the discrete logarithm problem. Finally a simplified version of our source code for finding such congruent number elliptic curves is given in the appendix.

Contents

Introduction

In this thesis we try to find congruent number elliptic curves of relatively high rank by implementing an algorithm for finding such curves in SageMath. In the case of congruent number elliptic curves, relatively high rank means already ranks that are greater or equal to six, in contrast to general elliptic curves where the highest known rank at the moment is 28 and was found by Noam D. Elkies in 2006. For further details see Section 3.1. A congruent number elliptic curve is given by the following equation

$$E_n : y^2 = x^3 - n^2 x,$$

where $n \geq 0$ is a squarefree congruent number and we can see such a curve in Figure 0.1. All pictures in this work were drawn using Asymptote by modifying the source code of the

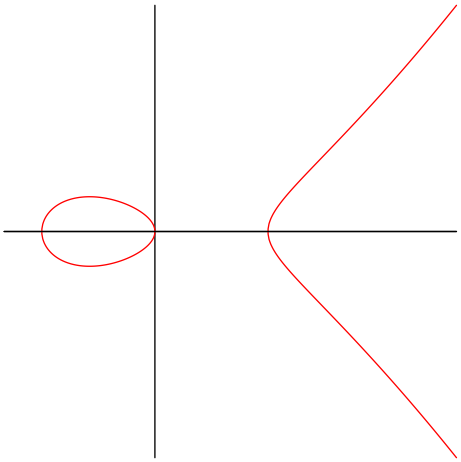

Figure 0.1.: The elliptic curve $E_n : y^2 = x^3 - 3^2 x$.

example 'elliptic.asy' which can be found at [3].
We call a squarefree integer congruent, if it is the area of a right-angled triangle with rational sides. As you can see in Section 3.4 we were able to find several congruent number elliptic curves with rank six.
Now we give a brief overview of each chapter in this work. In Chapter 1 we start with some basic knowledge about elliptic curves defined over a field K given in general Weierstrass form

$$E : y^2 - a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6,$$

which can be transformed to short Weierstrass form if the $\operatorname{char} K \neq 2, 3$

$$E : y^2 = x^3 - ax - b,$$

where $a, b \in K$. The characteristic of a field K is

$$\operatorname{char} K = \begin{cases} \min\{n \in N \mid n \cdot 1 = 0\} & \text{if such an } n \text{ exists,} \\ 0 & \text{otherwise} \end{cases}.$$

Hence, fields as the field of rational numbers $\mathbb{Q}$ or the field of real numbers $\mathbb{R}$ have characteristic 0. Whereas finite fields $\mathbb{F}_q = \mathbb{F}_{p^n}$ have characteristic p. Then we continue with singular curves, the projective space and the fact that the points on an elliptic curve form a group. The point at infinity $\mathcal{O}$ acts as an identity element for the point addition on elliptic curves. We also consider elliptic curves over the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$. Finally we present the reduction of an elliptic curve which is needed in our implementations see therefore Chapter 4.

Afterwards in Chapter 2 we consider the very old congruent number problem and we give a brief history on it starting with Diophantus in ancient times through to the 20-th century. We shall also present some problems which are equivalent to the congruent number problem and some classes of congruent numbers. Then we give the relation between congruent numbers and congruent number elliptic curves. This relation can be seen in the following theorem.

Theorem. *The positive integer n is a congruent number if and only if $E_n : y^2 = x^3 - n^2x$ defined over $\mathbb{Q}$ has rank $r > 0$.*

This theorem is later given as Theorem 2.10 in Chapter 2. Finally we present some facts about congruent number elliptic curves.

In Chapter 3 we search for congruent number elliptic curves of high rank. We first present rank records of general elliptic curves and also congruent number elliptic curves found so far. Then we give a general idea for finding elliptic curves with relatively high rank and afterwards our approach, due to Andrej Dujella [15], which is used in our implementation. Finally we present the congruent number elliptic curves of rank six found by our algorithm.

For more details about our implementation we discuss our program for finding high rank congruent number elliptic curves in Chapter 4. We additionally present some problems we came across during the implementation phase and some hints which where given by Andrej Dujella.

Since the 1980s, when elliptic curves were introduced by Neal Koblitz and Victor Miller for cryptographic reasons, elliptic curves became more and more popular. In cryptography mainly elliptic curves over finite fields are used. In Chapter 6 we see that we can easily transform cryptosystems such as the Diffie-Hellman key exchange from finite fields to

elliptic curves and also other cryptosystems which security is based on the discrete logarithm problem. Since the discrete logarithm in elliptic curves is much harder to solve than in finite fields we obtain a much more higher security even for smaller keys. We further present the Massey-Omura and ElGamal cryptosystems which can be used for exchanging secret messages over insecure channels. In contrast to the previously mentioned cryptosystems, we can use the elliptic curve digital signature algorithm for signing messages. There are also other applications of elliptic curves as they were used by Andrew Weil to prove Fermat's last theorem, which states that the equation

$$x^n + y^n = z^n$$

with $n > 2$ has no integer solutions. Today elliptic curve cryptography has found many applications in the area around the world wide web.
Finally we state the source code of our algorithm in the appendix.
For further details see [1, 13, 14, 15, 21, 25, 35].

1. Elliptic curves

In this chapter we will present some basic facts and definitions about elliptic curves which are based on the lecture notes [14] and the books [4, 10, 16, 19, 20, 21, 32, 33, 35] and [37].

Notation 1.1. *For a field K we denote by $\overline{K}$ its algebraic closure.*

Definition 1.2 (Elliptic curve)**.** *Let K be any field. An elliptic curve E over K is defined by an equation of the form*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1.1}$$

where the coefficients $a_1, a_2, a_3, a_4, a_6 \in K$ and the discriminant

$$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \neq 0,$$

for

$$\begin{aligned} b_2 &= a_1^2 + 4a_2, \\ b_4 &= 2a_4 + a_1a_3, \\ b_6 &= a_3^2 + 4a_6, \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{aligned}$$

Therefore the elliptic curve E consists of all points $(x, y) \in \overline{K} \times \overline{K}$ satisfying the above equation together with the point at infinity $\mathcal{O}$. We say that the above equation is in Weierstrass form.

Definition 1.3. *Since the discriminant $\Delta \neq 0$ we can further define the j-invariant of the curve E as*

$$j(E) = (b_2^2 - 24b_4)^3/\Delta.$$

Later we will see that we can give a classification of isomorphic elliptic curves with the help of the j-invariant.

Remark 1.4. *The field K could be for example the field of rational numbers $\mathbb{Q}$, the field of real numbers $\mathbb{R}$, the field of complex numbers $\mathbb{C}$ or any finite field $\mathbb{F}_q$ where $q = p^r$ with p prime and $r \geq 1$.*

Remark 1.5. *If E is defined over K, then it is also defined over any extension field $L \supseteq K$.*

Suppose we would like to limit the coordinates x, y to elements of the field K.

Definition 1.6. *We call a point $P = (x, y) \in E$ with $x, y \in K$ a K-rational point and the group*

$$E(K) = \{\mathcal{O}\} \cup \{(x, y) \in K \times K \mid y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\}$$

is called the group of K-rational points.

We can also search for points over larger fields.

Definition 1.7. *Let K and L be arbitrary fields with $L \supseteq K$ and let E be an elliptic curve defined over K. Then we can also consider E over L which is defined as follows:*

$$E(L) = \{\mathcal{O}\} \cup \{(x, y) \in L \times L \mid y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\}.$$

The elements of $E(L)$ are called L-rational points.

In the definition of an elliptic curve we have demanded that the discriminant has to be nonzero, but it is possible that we obtain a singular curve i.e., a curve with $\Delta = 0$, if we reduce an elliptic curve modulo a prime p. In the next section we consider such curves.

1.1. Singular curves

This section is based on [19, 21, 32, 33] and [37].

Definition 1.8. *A point $P = (x, y) \in \overline{K} \times \overline{K}$ on an elliptic curve E over K is called singular if both partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$ vanish at P. We call a point $P = (x, y)$ non-singular or smooth if at least one partial derivative at P unequals zero.*

Definition 1.9. *An elliptic curve is called singular if it has a singular point and it is called non-singular, if all points $P \in \overline{K} \times \overline{K}$ on the curve are non-singular.*

Remark 1.10. *The point at infinity $\mathcal{O}$ is not singular. For more details about this statement see Remark 1.38.*

Remark 1.11. *We also know, if there is a singular point on E, then this is the only one, cf. Proposition* 3.10 *in [19].*

Now let us suppose that we are looking for singular points on an given elliptic curve E and let us assume that $P = (x_0, y_0)$ is a singular point on $E : f(x, y) = 0$. It is sufficient to consider points $P \in \overline{K} \times \overline{K}$ in the affine plane since the point at infinity is non-singular. Therefore both partial derivatives must vanish at $P_0 = (x_0, y_0)$ and hence the following equations must be satisfied

$$f(x_0, y_0) = {y_0}^2 + a_1x_0y_0 + a_3y_0 - {x_0}^3 - a_2{x_0}^2 - a_4x_0 - a_6 = 0, \tag{1.2}$$

$$\frac{\partial f}{\partial x}(x_0, y_0) = a_1y_0 - 3{x_0}^2 - 2a_2x_0 - a_4 = 0, \tag{1.3}$$

$$\frac{\partial f}{\partial y}(x_0, y_0) = 2y_0 + a_1x_0 + a_3 = 0, \tag{1.4}$$

for a singular point.
In the following we present some pictures of singular curves. As already mentioned, the source code of the example "elliptic.asy", cf. [3], were modified by us.
As you can see in Figure 1.1, a singular curve can have a cusp. This happens if the right hand side of the elliptic curve equation has a triple root as it is the case for $E : y^2 = x^3$. Here the singularity is the point $P = (0, 0)$. In the two other cases the right hand sides of the equations have double roots. The example in Figure 1.2 shows the singular curve $E : y^2 = x^3 + x^2$ which has a node at the singular point $P = (0, 0)$. This point has two distinct tangent directions. Finally we present a singular curve with an isolated point $P = (0, 0)$ as it is illustrated in Figure 1.3 for the curve $E : y^2 = x^3 - x^2$.

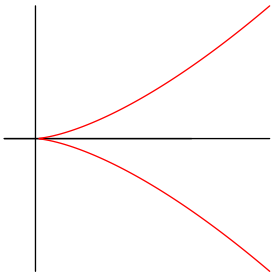

Figure 1.1.: A singular curve $E : y^2 = x^3$ with a cusp at $P = (0, 0)$.

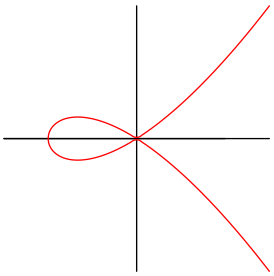

Figure 1.2.: A singular curve $E : y^2 = x^3 + x^2$ with a node at $P = (0, 0)$.

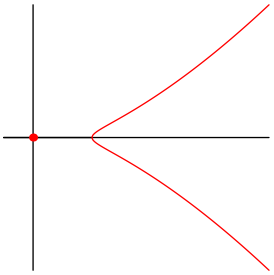

Figure 1.3.: A singular curve $E : y^2 = x^3 - x^2$ with an isolated point at $P = (0, 0)$.

It is possible to classify singular curves with the help of the following proposition.

Proposition 1.12. *Let K be a field of characteristic* $\operatorname{char} K \neq 2$. *For an elliptic curve E over K given in Weierstrass form the following statements are equivalent:*

(i) The elliptic curve E is non-singular.

(ii) The discriminant Δ of E is nonzero.

(iii) *The right hand side of the equation $y^2 = x^3 + ax^2 + bx + c$ has no multiple roots i.e. the equation $x^3 + ax^2 + bx + c$ has three distinct roots.*

Remark 1.13. *Since the field K has characteristic* $\operatorname{char} K \neq 2$ *we can transform the given long Weierstrass equation into $y^2 = x^3 + ax^2 + bx + c$ by an admissible change of variables. For further details see Section 1.2 and Section* 1.3 *in [33].*

Proof. $(i) \Longleftrightarrow (ii)$ See proof of Theorem 3.2 in [19].
$(i) \Longleftrightarrow (iii)$ See proof of Proposition 3.5 in [19]. □

Remark 1.14. *For a generalization of the equivalence $(i) \Longleftrightarrow (ii)$ (without the restriction:* $\operatorname{char} K \neq 2$*) see Appendix A in [32].*

In the next section we consider elliptic curves defined over fields K which can be transformed to shorter forms such as the short Weierstrass form.

1.2. Weierstrass equations

If we have a field K of special characteristic, then equation (1.1) in long Weierstrass form can be simplified by admissible changes of variables. As we already saw in the previous section for the case $\operatorname{char} K \neq 2$. For further details see [10, 16, 32] and [35].
In case of a field K with $\operatorname{char} K = 2$ we can transform the equation in long Weierstrass form (1.1) depending on a_1 to

$$y^2 + cy = x^3 + ax + b \quad (\text{if } a_1 = 0), \tag{1.5a}$$

with

$$\begin{aligned} \Delta &= c^4 \\ j(E) &= 0, \end{aligned}$$

or to

$$y^2 + xy = x^3 + ax^2 + b \quad (\text{if } a_1 \neq 0), \tag{1.5b}$$

with

$$\begin{aligned} \Delta &= b \\ j(E) &= 1/b, \end{aligned}$$

where $a, b, c \in K$.
If $\operatorname{char} K = 3$, then equation (1.1) can be written as

$$y^2 = x^3 + ax^2 + b \quad (\text{if } a_1^2 \neq -a_2), \tag{1.6a}$$

with

$$\Delta = -a^3 b,$$
$$j(E) = -a^3/b,$$

or as

$$y^2 = x^3 + ax + b \quad (\text{if } a_1^2 = -a_2), \tag{1.6b}$$

with

$$\Delta = -a^3,$$
$$j(E) = -0,$$

depending on a_1 and a_2 and $a, b \in K$.
For the case $\operatorname{char} K \neq 2, 3$, we can transform equation (1.1) to short Weierstrass form

$$y^2 = x^3 + ax + b. \tag{1.7}$$

The discriminant of the curve is given by

$$\Delta = -16(4a^3 + 27b^2),$$

and the curve E has j-invariant

$$j(E) = 1728a^3/4\Delta.$$

As already mentioned, we can use the j-invariant to determine if two elliptic curves are isomorphic. Since it does not depend on a particular chosen equation, but it is an invariant of the whole isomorphism class of an elliptic curve.

Proposition 1.15. *Let E and E' be two elliptic curves defined over K. Then $E \simeq E'$ over $\overline{K}$ if and only if the j-invariants of the two curves $j(E)$ and $j(E')$ coincide.*

Proof. See Proposition 1.4 in [32]. □

Remark 1.16. *There are two special cases for the j-invariant of an elliptic curve E. Namely $j(E) = 0$, when the elliptic curve is of the form $E : y^2 = x^3 + b$. The second case is a j-invariant $j(E) = 1728$, which implies that the corresponding E has the form $E : y^2 = x^3 + ax$.*

Remark 1.17. *Another interesting fact is, that over a non-algebraically closed field K it is possible that two elliptic curves, with the same j-invariant, cannot be transformed into each other by rational functions with coefficients in K.*

This is the case for the following curves.

Example 1.18. *The curves $E : y^2 = x^3 - 4x$ and $E' : y^2 = x^3 - 25x$ defined over the rational numbers $\mathbb{Q}$ cannot be transformed into the other by rational functions although the j-invariants $j(E) = j(E') = 1728$ coincide. The reason for that will be given in Example 2.12.*

Before we continue with the group law we present the definition of twists of elliptic curves.

Definition 1.19. *Let E and E' be two elliptic curves defined over K. If these two curves have the same j-invariant, then we say that they are twists of each other.*

1.3. Group law

Next we would like to show that the points of an elliptic curve form an abelian group where the point at infinity $\mathcal{O}$ is the identity element. This group is important for elliptic curve cryptography. For more information see [14, 21, 32, 33, 35, 37] and [38].
First of all we will define the negative of a given point and the addition of two points on an elliptic curve. Assume we have an elliptic curve E in short Weierstrass form over a field K with $\operatorname{char} K \neq 2, 3$.

Definition 1.20 (Negative of a point)**.** *The negative of a given point $P = (x, y)$ is the other point of intersection between a vertical line through P and the elliptic curve E. Suppose we have an elliptic curve in short Weierstrass form, then the negative of a point P is just the reflection along the x-axis i.e. $-P = (x, -y)$. The negative of the point at infinity $P = \mathcal{O}$ is the point at infinity itself. So $-\mathcal{O} = \mathcal{O}$.*

From the geometrical point of view the point addition can be seen as the following way of proceeding:

Definition 1.21 (Addition of two distinct points)**.** *For two points P and Q we draw a line through these two points. We label the third point of intersection between this line and the elliptic curve E as R. Since the elliptic curve is given by a cubic equation it intersects the line at exactly three points. However these points may not be pairwise distinct as it is the case if the line is tangent at a point. Afterwards we reflect this point R across the x-axis and get our result $P + Q = -R$.*

Definition 1.22 (Doubling a point)**.** *Suppose we want to compute $2P$. In that case we use a tangent line at P and consider the second point of intersection R between this line and the elliptic curve E. Again we have to reflect this point across the x-axis to obtain our required point $2P = -R$.*

The following pictures illustrate these constructions.

1.3.1. Explicit formulas for the point addition

Next we will present some arithmetic formulas for point addition because they will simplify computing the sum.
Suppose we have two distinct points

$$P = (x_1, y_1) \neq \mathcal{O} \text{ and}$$
$$Q = (x_2, y_2) \neq \mathcal{O} \text{ with } x_1 \neq x_2,$$

and we want to compute the sum

$$P + Q = (x_3, y_3).$$

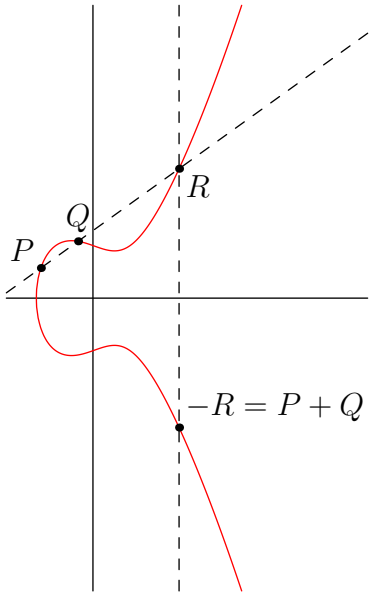

Figure 1.4.: Addition: $P + Q$

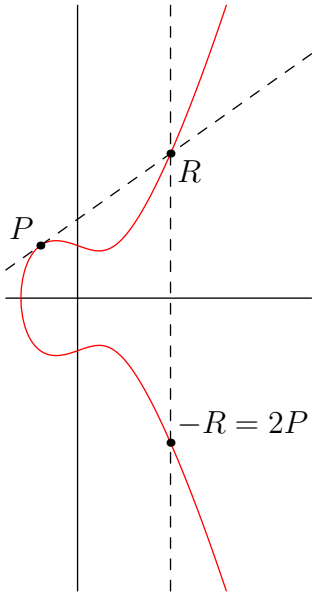

Figure 1.5.: Doubling: $2P$

In other words we try to compute from these two given points another point on the curve. As described above we have to draw a line ℓ through P and Q. This line has slope

$$\alpha = (y_2 - y_1)/(x_2 - x_1) \tag{1.8}$$

and hence its equation is of the form $y = \alpha(x - x_1) + y_1$. We know that for another point of intersection the following must hold:

$$\begin{aligned} y^2 &= (\alpha(x - x_1) + y_1)^2 = x^3 + ax + b, \\ &= (\alpha x)^2 - 2\alpha^2 x x_1 + (-\alpha x_1)^2 - 2\alpha x_1 y_1 + (y_1)^2 + 2\alpha x y_1 \\ &= (\alpha^2)x^2 - (2\alpha^2 x_1 - 2\alpha y_1)x - (2\alpha x_1 y_1 - \alpha^2 x_1^2 - y_1^2) \end{aligned}$$

and by rearranging we obtain the following equation

$$0 = x^3 - (\alpha^2)x^2 + (2\alpha^2 x_1 - 2\alpha y_1)x + (2\alpha x_1 y_1 - \alpha^2 x_1^2 - y_1^2). \tag{1.9}$$

Since we know that our equation in short Weierstrass form has three distinct roots we get

$$\begin{aligned} &x^3 - (\alpha^2)x^2 + (2\alpha^2 x_1 - 2\alpha y_1)x + (2\alpha x_1 y_1 - \alpha^2 x_1^2 - y_1^2) \\ &= (x - x_1)(x - x_2)(x - x_3) \\ &= x^3 - (x_1 + x_2 + x_3)x^2 + (x_1 x_2 + x_2 x_3 + x_1 x_3)x - (x_1 x_2 x_3). \end{aligned}$$

Now we can easily compute the coordinate $x_3 = \alpha^2 - x_1 - x_2$ because two intersection points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are already known. For y_3 we just have to put x_3 in

the equation of ℓ and so we have the point R. To get the sum we have to reflect this point along the x-axis i.e. negate the y-coordinate.
Therefore

$$\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2; \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \end{aligned} \tag{1.10}$$

If P and Q have the same x-coordinate but different y-coordinates, then the line ℓ intersecting P and Q is a vertical line and the third intersection point is the point at infinity $\mathcal{O}$. Since reflecting $\mathcal{O}$ also gives $\mathcal{O}$ we have $P + Q = \mathcal{O}$.
Suppose that $P = (x_1, y_1) \neq \mathcal{O}$ but $Q = \mathcal{O}$. Then the line through P and Q is again a vertical line that intersects the elliptic curve E in another point P' and reflecting it gives P again. So $P + \mathcal{O} = P$. Furthermore we know that $\mathcal{O} + \mathcal{O} = \mathcal{O}$.
If $P = Q = (x_1, y_1)$ we are in the case of doubling a point P. Here we cannot use our previous formula for α, but we can compute the slope of ℓ by using implicit differentiation of $y^2 = f(x)$ and if we proceed as above we get the following coordinates for $2P$:

$$\begin{aligned} x_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1; \\ y_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1. \end{aligned} \tag{1.11}$$

Here we have to assume that $y_1 \neq 0$ because otherwise the tangent line ℓ at P is a vertical line, where $R = \mathcal{O}$ is the only other point intersection between ℓ and the curve. So here we have $2P = \mathcal{O}$.
The next examples show applications of the above formulas and they are taken from [21].

Example 1.23 (Point addition)**.** *Let $P = (-3, 9)$ and $Q = (-2, 8)$ be two distinct points on the elliptic curve*

$$E : y^2 = x^3 - 36x.$$

We use the formulas (1.10) *with $x_1 = -3$, $y_1 = 9$, $x_2 = -2$, $y_2 = 8$ and get $x_3 = 6$ and $y_3 = 0$. See Figure 1.6.*

Example 1.24 (Point doubling)**.** *Let $P = (-3, 9)$ be a point on the elliptic curve*

$$E : y^2 = x^3 - 36x.$$

For $2P$ we substitute $x_1 = -3$, $y_1 = 9$, $a = -36$ in the formulas (1.11) *which gives $x_3 = \frac{25}{4}$ and $y_3 = \frac{-35}{8}$. See Figure 1.7.*

Theorem 1.25 (Group law)**.** *The set of points on an elliptic curve form an abelian group together with the point at infinity, since the point addition $P + Q$ as defined above satisfies the following properties:*

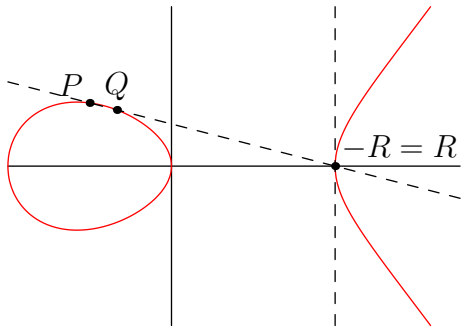

Figure 1.6.: Addition of $P = (-3, 9)$ and $Q = (-2, 8)$

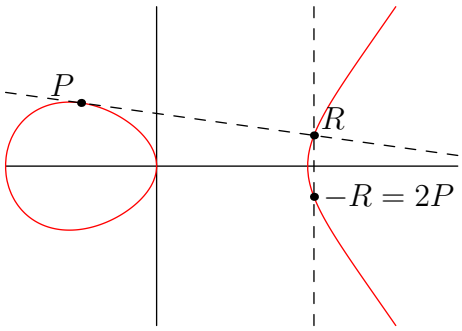

Figure 1.7.: Doubling of the point $P = (-3, 9)$

1. *$P + \mathcal{O} = \mathcal{O} + P = P$ for all points $P \in E$. (identity element)*
2. *For every $P \in E$ there exists a negative element $-P$ such that $P - P = \mathcal{O}$. (inverse element)*
3. *Let $P, Q, R \in E$. Then the following holds $(P+Q)+R = P+(Q+R)$. (associativity)*
4. *$P + Q = Q + P$ for all points $P, Q \in E$. (commutativity)*

Proof. See Section 2.2 in [35]. □

Remark 1.26. *If a line ℓ intersects the points P, Q and R then*

$$(P + Q) + R = \mathcal{O}.$$

Next we define the order of a point on an elliptic curve.

Definition 1.27. *The order N of a point P on an elliptic curve E is the smallest positive integer such that $NP = \mathcal{O}$. If $N < \infty$ we say the point has order N and we call the point a N-torsion point or simply just torsion point. These points are called points of finite order. We denote the set of all points of order N by*

$$E[N] = \{P \in E(\overline{K}) | NP = \mathcal{O}\},$$

and the set of finite points is called torsion subgroup, we write $E(\mathbb{Q})_{tors}$. Thus the following holds:

$$E(\mathbb{Q})_{tors} = \bigcup_{N=1}^{\infty} E[N].$$

Naturally such a finite N does not need to exist.

Remark 1.28. *Let E be an elliptic curve in short Weierstrass form. Then a point $P = (x, y) \neq \mathcal{O} \in E$ has order two if and only if $y = 0$. Since*

$$\begin{aligned} 2P = \mathcal{O} &\iff \\ P = -P &\iff \\ (x, y) = (x, -y). & \end{aligned}$$

The coordinates x are the (complex) roots of the right side of the elliptic curve equation.

The following example shows how the order of a point can be computed. It is taken from [21].

Example 1.29. *What is the order of the point $P = (2, 3)$ on the curve $y^2 = x^3 + 1$? By the formulas given in (1.11) we obtain $2P = (0, 1)$ and $4P = 2(2P) = (0, -1)$. Hence $2P = -4P$ and so $6P = \mathcal{O}$. Thus the order of P is $2, 3$ or 6, but $2P = (0, 1) \neq \mathcal{O}$. We now suppose that P has order 3, but then $4P = P$ must hold which is not true. Therefore P has order 6.*

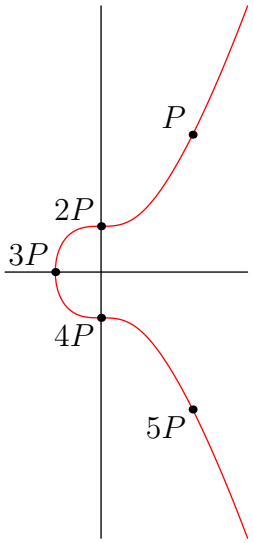

Figure 1.8.: A point of order six.

Remark 1.30. *Lagrange's theorem, see Theorem B.1 in [35], states that the order of an element g in a finite group G divides the group order. For a proof of this statement see Section 44 in [38].*

In the next section we will consider the projective plane and we will introduce the point at infinity in a different way.

1.4. Projective space

At first we start with some definitions as the projective space or a projective point. For more details see [21, 33] and [35].
Since the projective space is given by a set of equivalence classes we also have to define an equivalence relation.

Definition 1.31. *Two triples (X, Y, Z) and (X', Y', Z') are said to be equivalent if there is a nonzero scalar $\lambda \in K$ such that*

$$(X, Y, Z) = (\lambda X', \lambda Y', \lambda Z')$$

and we write

$$(X, Y, Z) \sim (X', Y', Z').$$

Definition 1.32. *The projective space over a field K is given by the set of equivalence classes of triples (X, Y, Z) with $X, Y, Z \in K$ and at least one component nonzero.*

Definition 1.33. *We call an equivalence class a projective point and it is denoted $(X : Y : Z)$ since it only depends on the ratios of X to Y to Z. On the other hand we call a point (X, Y, Z) a representative of the equivalence class $(X : Y : Z)$.*

Remark 1.34. *Both triples (X, Y, Z) and (X', Y', Z') are elements of the projective point $(X : Y : Z)$.*

Remark 1.35. *We call a projective point $(X : Y : Z)$ with $Z \neq 0$ finite and the projective points with $Z = 0$ form the line at infinity. The point at infinity will be defined in Definition 1.37.*

Suppose we have an elliptic curve in Weierstrass form in the affine plane. Then there exists a corresponding homogenous Weierstrass form in the projective space and it can be obtained by substituting x by X/Z and y by Y/Z and afterwards multiplying by a suitable power of Z to eliminate any denominators.

Example 1.36. *Let*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{1.12}$$

be an equation in Weierstrass form of an elliptic curve. Then we get the following projective equation

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{1.13}$$

by the above procedure.

As in the affine version the property that this curve should be non-singular means that the partial derivatives

$$\begin{aligned}\partial F/\partial X &= a_1YZ - 3X^2 - 2a_2XZ - a_4Z^2\\ \partial F/\partial Y &= 2YZ + a_1XZ + a_3Z^2\\ \partial F/\partial Z &= Y^2 + a_1XY + 2a_3YZ - a_2X^2 - 2a_4XZ - 3a_6Z^2\end{aligned}$$

of $F(X,Y,Z) = 0$ should not vanish at a point P simultaneously. This projective form is satisfied by projective points of the form $(X : Y : Z)$ with

$$\begin{aligned}X &= x \cdot Z\\ Y &= y \cdot Z \text{ and}\\ Z &= 1,\end{aligned}$$

if and only if (x, y) satisfies the affine equation. Moreover by projective points of the form $(X : Y : 0)$ which form the line at infinity.
So far we have seen that the point at infinity is the third point of intersection between every vertical line and a given elliptic curve E and besides that it serves as identity element in the group of points on an elliptic curve. Now we want to give a further definition with the help of the projective plane.

Definition 1.37. *We use the fact that the point at infinity $\mathcal{O}$ is the only point of intersection between the line at infinity and an elliptic curve E. So if we put $Z = 0$ in Equation (1.13), as it is necessary for the line at infinity, we obtain $0 = X^3$ and hence also $X = 0$. Since we need one component to be nonzero, $Y = y$ has to be nonzero. Rescaling by y gives then $(0 : 1 : 0)$. So the point at infinity $\mathcal{O}$ corresponds to the projective point $(0 : 1 : 0)$. Furthermore the 'top' and the 'bottom' of the y-axis are the same because*

$$(0, 1, 0) \sim (0, -1, 0).$$

Next we present the reason why the point at infinity $\mathcal{O}$ is non-singular.

Remark 1.38. *The point at infinity $\mathcal{O}$ is not singular since the partial derivative*

$$\partial F/\partial Y(\mathcal{O}) = 1$$

of the projective equation $F(X,Y,Z)$ never equals zero.

Straightaway we shall consider our given elliptic curve E over special fields for example the field of rational numbers $\mathbb{Q}$, the field of real numbers $\mathbb{R}$ or finite fields $\mathbb{F}_q$. For the rest of this chapter we consider elliptic curves in short Weierstrass form.
The set of points with rational coordinates form a subgroup of the set of points whose coordinates are real numbers. Because the sum and difference of rational (real) numbers are also rational (real) numbers. So the following relations hold:

$$\{\mathcal{O}\} \subset E(\mathbb{Q}) \subset E(\mathbb{R}).$$

We first consider elliptic curves over the rational numbers.

1.5. Elliptic curves over the rationals

For the present section let E be an elliptic curve over $\mathbb{Q}$ i.e. the constants a and b in the short Weierstrass form are elements in $\mathbb{Q}$. We look for points $(x, y) \in E$ with $x, y \in \mathbb{Q}$ satisfying the elliptic curve equation. For elliptic curves over the rationals we can give further details about the group of rational points $E(\mathbb{Q})$ by the Mordell-Weil Theorem and we get further information of the torsion subgroup $E(\mathbb{Q})_{tors}$ of $E(\mathbb{Q})$ by Mazur's Theorem. For more information see [5, 18, 20, 21, 23, 26, 32, 33] and [35].
We start with the definition of Q-rational points.

Definition 1.39. *Points (x, y) with $x, y \in \mathbb{Q}$ which are satisfying a given equation of an elliptic curve E are called $\mathbb{Q}$-rational points or even shorter rational points on an elliptic curve E.*

Next we shall present an important theorem called Mordell's theorem.

Theorem 1.40 (Mordell)**.** *Let E be an elliptic curve over $\mathbb{Q}$. Then the group of rational points $E(\mathbb{Q})$ is finitely generated and abelian.*

Proof. See chapter 8 in [35]. □

Mordell's theorem has been proved by Louis Mordell in 1922 and in 1928 André Weil extended this statement to elliptic curves over algebraic number fields in his thesis.

Theorem 1.41 (Mordell-Weil)**.** *Let K be a number field and let E be an elliptic curve defined over K. Then the Mordell-Weil group $E(K)$ is finitely generated.*

Proof. For a sketch of the proof see Section 6 in [18]. □

Mordell's theorem says that the group of rational points $E(\mathbb{Q})$ is finitely generated and abelian. Thus by the structure theorem of finitely generated abelian groups, there is a decomposition of the form

$$E(\mathbb{Q}) \simeq E(\mathbb{Q})_{tors} \oplus \mathbb{Z}^r,$$

where $E(\mathbb{Q})_{tors}$ is a finite abelian group and $r \geq 0$ is a natural number. So the group $E(\mathbb{Q})$ is given by a finite torsion subgroup (points of finite order) $E(\mathbb{Q})_{tors}$ plus a subgroup $\mathbb{Z}^r$ generated by a finite number of points of infinite order. The finite torsion subgroup $E(\mathbb{Q})_{tors}$ can be easily computed by Theorem 1.44, the Nagell-Lutz Theorem.

Definition 1.42 (Mordell-Weil rank)**.** *The number $r \geq 0$ above is called Mordell-Weil rank of $E(\mathbb{Q})$.*

The rank r is nonzero if and only if the elliptic curve E has infinitely many $\mathbb{Q}$-rational points. Hence the rank $r = 0$ if and only if the group $E(\mathbb{Q})$ is finite. For most cases the computation of the rank r is very difficult.
Before we state the Nagell-Lutz theorem we have a look at Mazur's theorem which gives a characterization of torsion subgroups.

Theorem 1.43 (Mazur)**.** *Let E be an elliptic curve over $\mathbb{Q}$. Then the subgroup of finite points is one of the following groups*

$$E(\mathbb{Q})_{tors} = \mathbb{Z}/n\mathbb{Z} \quad \text{for } n = 1, 2, 3, \ldots, 10, 12$$
$$\text{or}$$
$$E(\mathbb{Q})_{tors} = \mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \quad \text{for } n = 2, 4, 6 \text{ or } 8.$$

Proof. For a proof of this theorem see [23]. □

Next we will try to find the torsion points of an elliptic curve E.

Theorem 1.44 (Nagell-Lutz)**.** *Let $E : y^2 = x^3 + ax + b$ with $a, b \in \mathbb{Z}$ be an elliptic curve in short Weierstrass form. If $P = (x, y) \in E(\mathbb{Q})$ has finite order, then $x, y \in \mathbb{Z}$.*
In the case that $y = 0$, P has order 2 and otherwise

$$y^2 | D = 4a^3 + 27b^2.$$

Proof. See Chapter 8 in [35]. □

With the help of the Nagell-Lutz theorem we know that the points of finite order have integer coordinates and we can give a list of possible torsion points. For each point P in this list we try to find the order N. We know that it is sufficient to consider $N \leq 13$ by Mazur's theorem. Then either $NP = \mathcal{O}$ and so P is of order N, or NP has no integer coordinates and therefore P is no torsion point.
The following examples show how to compute the torsion subgroup $E(\mathbb{Q})_{tors}$ of given elliptic curves. These examples are taken from the list of exercises in Chapter II in [33].

Example 1.45. *Let $E : y^2 = x^3 + 4x$ be an elliptic curve over $\mathbb{Q}$. Since $a = 4$ and $b = 0$ we get the discriminant*

$$\Delta = -16(4a^3 + 27b^2) = -4096$$

and

$$D = 4a^3 + 27b^2 = 256 = (16)^2.$$

Let $P = (x, y)$ be a point in $E(\mathbb{Q})_{tors}$ and assume $y = 0$. Since the equation $0 = x^3 + 4x$ has no other rational solution than $x = 0$ we found the point $(0, 0)$. For the case $y \neq 0$ we obtain that $y^2 | D = (16)^2$ from the Nagell-Lutz theorem. Therefore we have the following possibilities

$$y = \pm 1, \pm 2, \pm 4, \pm 8, \pm 16,$$

but only $x^3 + 4x = x(x^2 + 4) = (\pm 4)^2 = 16$ has a rational solution, namely $x = 2$. So the only possible torsion points are

$$E(\mathbb{Q})_{tors} = \{\mathcal{O}, (0, 0), (2, -4), (2, 4)\}.$$

Now we have to compute the orders of the given points. We already know that $\mathcal{O}$ *has order* 1 *and that the point* $(0,0)$ *has order* 2*, since* $2P = \mathcal{O} \iff P = -P$*. For the two remaining points we compute its orders and obtain that* $N = 4$*, since*

$$4(2,-4) = \mathcal{O},$$

and

$$4(2,+4) = \mathcal{O}.$$

So these two points have order 4 *and hence*

$$E(\mathbb{Q})_{tors} \simeq \mathbb{Z}_4.$$

You can see the elements of the torsion subgroup without the point at infinity marked by dots in Figure 1.9.

Example 1.46. *Let* $E : y^2 = x^3 + 1$ *be an elliptic curve over the rational numbers. Since* $a = 0$ *and* $b = 1$ *we have the discriminant*

$$\Delta = -16(4a^3 + 27b^2) = -432 \text{ and}$$
$$D = 4a^3 + 27b^2 = 27.$$

Let $P = (x,y) \in E(\mathbb{Q})$ *be a point of finite order. Since there is no rational solution for* $0 = x^3 + 1$ *we get that there is no rational point with* $y = 0$*. So we assume* $y \neq 0$ *and by using the Nagell-Lutz theorem we get that* $y^2 | 27 = 3^3$*. So the only possibilities left are*

$$y = \pm 1, \pm 3.$$

By solving the equation $y^2 = x^3 + 1$ *we get the following possible torsion points:*

$$\{\mathcal{O}, (0,1), (0,-1), (2,3), (2,-3)\}.$$

Computing the orders of the given points gives

$$3(0,1) = 3(0,-1) = \mathcal{O},$$
$$6(0,1) = 6(0,-1) = \mathcal{O}.$$

So the torsion subgroup

$$E(\mathbb{Q})_{tors} \simeq \mathbb{Z}_6.$$

The dots in Figure 1.10 mark the points of finite order without the point at infinity.

At the end of this section we present the definition of an quadratic twist of an elliptic curve which we will need later.

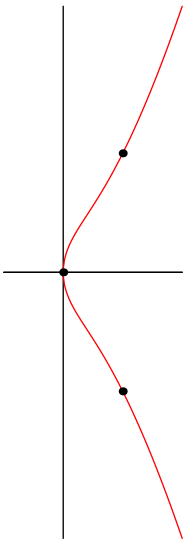

Figure 1.9.: $E : y^2 = x^3 + 4x$ and $E(\mathbb{Q})_{tors} = \mathbb{Z}_4$.

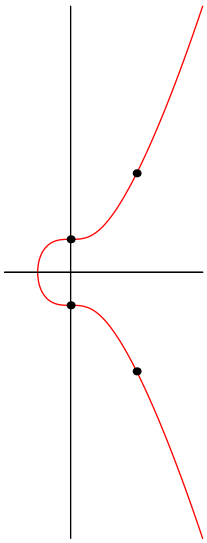

Figure 1.10.: $E : y^2 = x^3 + 1$ and $E(\mathbb{Q})_{tors} = \mathbb{Z}_6$.

Definition 1.47. *Let E be an elliptic curve*

$$E : y^2 = x^3 + ax + b$$

in short Weierstrass form defined over the rationals $\mathbb{Q}$ and with coefficients $a, b \in \mathbb{Z}$. For $d \neq 0$ a squarefree integer, we define the d-th quadratic twist of E as

$$E^d : dy^2 = x^3 + ax + b,$$

which can be transformed to

$$E^d : y'^2 = x'^3 + ad^2x' + bd^3,$$

multiplying the above equation by d^3 and setting $y' := d^2y$, $x' := dx$.

In the next section we consider elliptic curves over the real numbers.

1.6. Elliptic curves over the reals

For most fields K it is impossible to draw useful pictures of elliptic curves over it. However, in the field of real numbers $\mathbb{R}$ we can see an elliptic curve as an ordinary curve in the plane together with $\mathcal{O}$. For further details see [21] and [33].

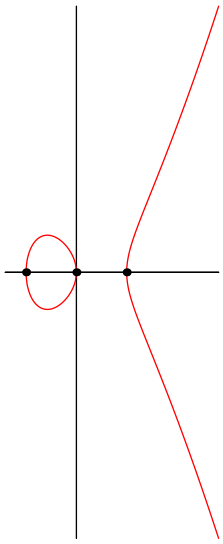

Figure 1.11.: $E : y^2 = x^3 - 2x$ with $\Delta = 512$.

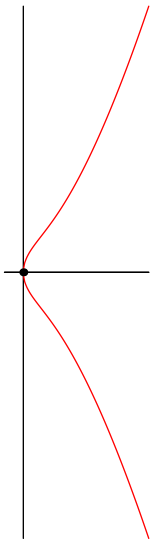

Figure 1.12.: $E : y^2 = x^3 + 2x$ with $\Delta = -512$.

There are two basic types of elliptic curves defined over $\mathbb{R}$: those with three real roots as you can see in Figure 1.11 and those with one real root depicted in Figure 1.12.
Since we are just considering non-singular curves we know that they cannot have multiple roots.
In the next section we consider elliptic curves defined over finite fields and we give a formula for the number of points on an elliptic curve E defined over $\mathbb{F}_q$. We already know that $\mathbb{F}_q \supset \mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ and $\mathbb{F}_q$ is a vector space over $\mathbb{F}_p$ with dimension r if $q = p^r$. Hence, the finite field $\mathbb{F}_q$ has $q = p^r$ elements and the characteristic $\operatorname{char} \mathbb{F}_q = p$. For every $q = p^r$ there is a unique field with q elements and $\mathbb{F}_q \simeq (\mathbb{Z}/p\mathbb{Z})[x]/(f(x))$ where f is an irreducible polynomial of degree r over $\mathbb{Z}/p\mathbb{Z}$. The multiplicative group of the field $\mathbb{F}_q$ is denoted by $\mathbb{F}_q^\times$ and it is cyclic, i.e., there is an element $g \in \mathbb{F}_q^\times$ such that the powers of g run through all elements of $\mathbb{F}_q^\times$.

1.7. Elliptic curves over finite fields

In this section let K be any finite field $\mathbb{F}_q$ and let E be an elliptic curve defined over $\mathbb{F}_q$. These elliptic curves are important for cryptographic reasons. As in the general case we can simplify the given elliptic curve equation to (1.5) if the characteristic $\operatorname{char} \mathbb{F}_q = 2$ and if $\operatorname{char} \mathbb{F}_q = 3$ then the given equation can be simplified and written as in (1.6). See [10, 21, 32, 33] and [35] for further details.
We can easily see that the group of points $E(\mathbb{F}_q)$ is finite since there are only finitely many

points $P = (x, y)$ with $x, y \in \mathbb{F}_q$. In particular an elliptic curve cannot have more than $2q + 1$ points (x, y) with $x, y \in \mathbb{F}_q$. Namely $2q$ pairs (x, y) where for each x there are at most 2 possible choices for y satisfying the given elliptic curve equation plus the point at infinity.
For an easier understanding we have a look at the following example.

Example 1.48. *Let us consider the elliptic curve $E : y^2 = x^3 + x + 1$ over $\mathbb{F}_3$. Hence there are three possibilities for $x \in \mathbb{F}_3$ and $y \in \mathbb{F}_3$:*

x	x^2	x^3	$x^3 + x + 1$
0	0	0	1
1	1	1	0
2	1	2	2

y	y^2
0	0
1	1
2	1

So $E(\mathbb{F}_3) = \{\mathcal{O}, (0, 1), (0, 2), (1, 0)\}$ are the points of E over $\mathbb{F}_3$ and $\#E(\mathbb{F}_3) = 4$.

Now we will give bounds on the order of the group $E(\mathbb{F}_q)$ respectively the cardinality of the elliptic curve E by Hasse's Theorem.

Theorem 1.49 (Hasse's Theorem)**.** *Let E be an elliptic curve over $\mathbb{F}_q$. Then the following estimate holds*

$$|q + 1 - \#E(\mathbb{F}_q)| \leq 2\sqrt{q}.$$

Proof. For a proof see chapter V in [32]. □

We can give further details about the group of $\mathbb{F}_q-$rational points by the following theorem.

Theorem 1.50. *Let E be an elliptic curve defined over $\mathbb{F}_q$. Then either*

$$E(\mathbb{F}_q) \simeq \mathbb{Z}_n,$$

or

$$E(\mathbb{F}_q) \simeq \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2},$$

where $n, n_1, n_2 \geq 1$ and $n_1 | n_2$.

Proof. See Section 4.1 in [35]. □

We present some examples of this theorem in the following.

Example 1.51. *Consider again the elliptic curve $E : y^2 = x^3 + x + 1$ as in Example 1.48. Since the $\mathbb{F}_3-$rational points have the following orders*

$$2(1, 0) = \mathcal{O},$$
$$4(0, 1) = \mathcal{O},$$
$$4(0, 2) = \mathcal{O},$$

we get that

$$E(\mathbb{F}_3) \simeq \mathbb{Z}_4.$$

Example 1.52. *Now let $E : y^2 = x^3 + 2$ be an elliptic curve over $\mathbb{F}_7$. Then the group of $\mathbb{F}_7$-rational points is given by*

$$E(\mathbb{F}_7) = \{\mathcal{O}, (0,3), (0,4), (3,1), (3,6), (5,1), (5,6), (6,1), (6,6)\}.$$

Since all of these points except $\mathcal{O}$ have order 3 we get that

$$E(\mathbb{F}_7) \simeq \mathbb{Z}_3 \oplus \mathbb{Z}_3.$$

Now we state the Legendre symbol which will be used later.

Definition 1.53. *Let $x \in \mathbb{F}_p$ and let p be an odd prime. Then the Legendre symbol is defined as*

$$\left(\frac{x}{p}\right) = \begin{cases} 0, & \text{if } x \equiv 0 \mod p, \\ +1, & \text{if } x \equiv t^2 \mod p \text{ has a solution } t \not\equiv 0 \mod p, \\ -1, & \text{if } x \equiv t^2 \mod p \text{ has no solution } t. \end{cases}$$

The second case means nothing else than t is a quadratic residue modulo p and the third case says that t is a quadratic nonresidue modulo p.
Further we can even count the points in the group $E(\mathbb{F}_q)$ by the following theorem.

Theorem 1.54. *Let E be an elliptic curve over $\mathbb{F}_q$ in short Weierstrass form, then*

$$\#E(\mathbb{F}_q) = q + 1 + \sum_{x \in \mathbb{F}_q} \left(\frac{x^3 + ax + b}{\mathbb{F}_q}\right), \tag{1.14}$$

where

$$\left(\frac{x}{\mathbb{F}_q}\right) = \begin{cases} 0, & \text{if } x = 0, \\ +1, & \text{if } x = t^2 \text{ has a solution } t \in \mathbb{F}_q^\times, \\ -1, & \text{if } x = t^2 \text{ has no solution } t \in \mathbb{F}_q^\times, \end{cases}$$

for $x \in \mathbb{F}_q$ and q odd. Here $\left(\frac{x}{\mathbb{F}_q}\right)$ describes a more general Legendre symbol defined in any finite field $\mathbb{F}_q$ with q an odd prime power.

Proof. See Section 4.3 in [35]. □

In the next example we can see an application of this theorem.

Example 1.55. *Let $E : y^2 = x^3 + x + 1$ be an elliptic curve in short Weierstrass form defined over $\mathbb{F}_3$ as in Example 1.48. Then the only possible nonzero square is* $1 \mod 3$. *Hence the formula in the previous theorem gives*

$$\begin{aligned}\#E(\mathbb{F}_3) &= 3 + 1 + \sum_{x=0}^{2} \left(\frac{x^3 + x + 1}{3}\right) \\ &= 4 + \left(\frac{1}{3}\right) + \left(\frac{0}{3}\right) + \left(\frac{2}{3}\right) \\ &= 4 + 1 + 0 - 1 = 4.\end{aligned}$$

This matches the order in Example 1.48.

In the following we are going to define the reduction of an elliptic curve modulo a prime.

1.8. Reduction of an elliptic curve

In this section we introduce the reduction of an elliptic curve. For a more detailed information see [5, 15, 18, 19, 33] and [35].

Definition 1.56. *Let $E : y^2 = x^3 + ax + b$ with $a, b \in \mathbb{Z}$ be a non-singular elliptic curve over $\mathbb{Q}$ in short Weierstrass form. Then for any prime p we define the reduction of the elliptic curve E modulo p as follows:*

$$\tilde{E}(\mathbb{F}_p) : y^2 = x^3 + \tilde{a}x + \tilde{b}, \tag{1.15}$$

where $\tilde{a} \equiv a \mod p$ and $\tilde{b} \equiv b \mod p$. In other words we consider the curve E over the finite field $\mathbb{F}_p$ and the reduction $\tilde{E}(\mathbb{F}_p)$ has discriminant $\tilde{\Delta} = -16(4\tilde{a}^3 + 27\tilde{b}^2) = \Delta \mod p$.

Even if E over $\mathbb{Q}$ is a non-singular elliptic curve, this does not imply that the reduced elliptic curve $\tilde{E}(\mathbb{F}_p)$ is non-singular too. This is only the case if $p \nmid \Delta$ as you can see in the following definition.

Definition 1.57. *We say an elliptic curve E has good reduction at p if and only if the reduced elliptic curve $\tilde{E}(\mathbb{F}_p)$ is non-singular. Which is equivalent to the fact that the reduced discriminant $\tilde{\Delta} \neq 0$.*
We say E has bad reduction at p, if $\tilde{E}(\mathbb{F}_p)$ is singular.

Remark 1.58. *If E is an elliptic curve defined over $\mathbb{Q}$ with good reduction at p, then the reduction function $r_p : E(\mathbb{Q}) \longrightarrow \tilde{E}(\mathbb{F}_p)$ is a group homomorphism.*

Proof. For a proof of this remark see Section 5 in [18]. □

Next we try to find points $\tilde{P} = (\tilde{x}, \tilde{y})$ on $\tilde{E}(\mathbb{F}_p)$, therefore we try to reduce points $P = (x, y)$ modulo p. This method works just fine if the coordinates $x, y \in \mathbb{Z}$. However, for points

with rational coordinates we have to guarantee that the denominators are not divisible by p for a successful application of this method.
So the set of points on the reduced elliptic curve $\tilde{E}(\mathbb{F}_p)$ is given by

$$\tilde{E}(\mathbb{F}_p) = \{\mathcal{O}\} \cup \{P = (\tilde{x}, \tilde{y}) \in \mathbb{F}_p \times \mathbb{F}_p : \tilde{y}^2 \equiv \tilde{x}^3 + \tilde{a}\tilde{x} \mod p\}, \tag{1.16}$$

and the number of points on the curve $\tilde{E}(\mathbb{F}_p)$ is

$$\#\tilde{E}(\mathbb{F}_p) = 1 + \#\{P = (\tilde{x}, \tilde{y}) \in \mathbb{F}_p \times \mathbb{F}_p : \tilde{y}^2 \equiv \tilde{x}^3 + \tilde{a}\tilde{x} \mod p\}. \tag{1.17}$$

Now we shall present a very old problem in mathematics, the congruent number problem.

2. Congruent number problem

In this chapter we give a brief history on the congruent number problem and some details about its relation to elliptic curves of the form $E_n : y^2 = x^3 - n^2x$. For further information see references [2, 5, 6, 7, 8, 11, 12, 15, 17, 18, 19, 20, 29, 31, 34, 35] and [36]. But first of all we state the congruent number problem and the definition of a congruent number.

Definition 2.1. *A squarefree integer $n \geq 0$ is a congruent number if it is the area of a right-angled triangle with rational sides.*

Example 2.2. *For example the number 6 is a congruent number since it is the area $A = \frac{XY}{2} = \frac{12}{2} = 6$ of a right-angled triangle with the sides $X = 3$, $Y = 4$ and $Z = 5$, where X and Y are the catheti and Z is the hypotenuse.*

We can have a look at Figure 2.1 to get a better idea of the definition.

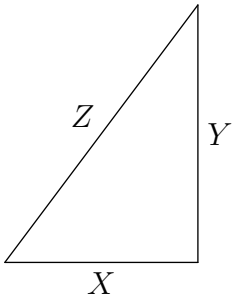

Figure 2.1.: A right-angled triangle with area n.

Remark 2.3. *Without loss of generality we can assume, that congruent numbers are positive squarefree integers. Suppose we would allow rational numbers to be congruent and let $0 \neq r \in \mathbb{Q}$ be a congruent number. Then there is a right-angled triangle with area r and rational sides X, Y and Z. Then we can always find some $s \in \mathbb{Q}$ such that s^2r is a squarefree integer and it is the area of a right-angled triangle with rational sides sX, sY and sZ.*
Now let n and n' be two integers and $s \in \mathbb{N}$. The number $n' = s^2n$ is congruent, if and only if n is congruent. Because if we suppose that sX, sY, sZ, are the rational sides of a right-angled triangle with area n', then X, Y, Z are the rational sides of a right-angled triangle with area n and vice versa. Therefore we check only squarefree integers for being congruent.

Next we give the congruent number problem.

Definition 2.4 (Congruent number problem)**.** *The congruent number problem is the problem deciding whether a given integer n is a congruent number.*

Suppose we would like to express the above problem in terms of equations. Let X, Y and Z, describe the sides of a right-angled triangle and let Z be the hypotenuse. Then the following equations describe the congruent number problem:

$$X^2 + Y^2 = Z^2, \tag{2.1}$$

$$n = \frac{XY}{2}. \tag{2.2}$$

A given positive integer n is a congruent number if and only if this system of equations has a solution (X, Y, Z) with X, Y, $Z \in \mathbb{Q}$.

2.1. History of the congruent number problem

Now we will continue with some historical details about this problem. For further details see [2, 7, 8, 12, 17, 20] and [34]. The congruent number problem is one of the oldest unsolved problems in mathematics and many authors have worked to achieve progress in this area. Although this problem is very easy to state, cf. Definition 2.4, finding an answer for a given number n is a challenging task. We start with Diophantus who was already searching for right-angled triangles, such that their areas are equal to given numbers n, in ancient times. However, also the Arabs were familiar with the congruent number problem, but they considered the following form:

Definition 2.5 (Congruent number problem - Arabs)**.** *Given an integer n. Is there any rational number x such that $x^2 + n$ and $x^2 - n$ are squares of rational numbers?*

This definition describes the congruent number problem as finding an arithmetic progression of three rational squares with a common difference n. This equivalent form had also been known to the Greeks. Furthermore Dickson mentioned in [12] that a manuscript which had been written before 972 AD already contained the congruent number problem. However, the Arabs probably had not known about the work of Diophantus. Instead it is assumed that the Arabs were introduced to this problem by the Hindus, who had already been familiar with the work of Diophantus. The following congruent numbers

$$5, 6, 14, 15, 21, 30, 34, 65, 70, 110, 154, 190, 210, 221, 231, 246, 290, 390, 429, 546, \ldots$$

together with ten even larger numbers had already been computed by Arab mathematicians.
Many years later in 1225 Fibonacci took a close look at the congruent number problem and he found a right-angled triangle with area 5 and conjectured that 1 is not a congruent number but he was not able to prove this statement. Only Fermat could give a proof on

that in 1659 more than four centuries later. This proof provides furthermore a solution of Fermat's last theorem for exponent 4, cf. [7]. Namely that there are no integer solutions other than the trivial ones for the equation

$$x^4 + y^4 = z^4.$$

Whereas the general statement

$$x^n + y^n = z^n \text{ with } n \geq 3,$$

has only been proved by Andrew Wiles in 1995.
Now let us return to congruent numbers. Beside Fermat, who additionally showed that 2 and 3 are non-congruent numbers, also Euler was considering congruent numbers and he was the first who found a right-angled triangle with area 7 in the 18th century.
Even in the 20th century many mathematicians worked on this problem and Géradin listed another 62 squarefree congruent numbers which are less than 1000 in 1915. In the early seventies Alter, Curtz and Kubota presented the following conjecture in [2]:

Conjecture 2.6. *Let n be a squarefree integer congruent to*

$$5,\ 6 \text{ or } 7 \mod 8,$$

then n is a congruent number.

This conjecture was proved by Stephens in 1975 under the assumption of the Birch and Swinnerton-Dyer conjecture. For further details see Section 2.5 and [34].
Now we show some problems which are equivalent to the congruent number problem.

2.2. Problems equivalent to the congruent number problem

In the following we will give some problems which are equivalent to the congruent number problem. The information of this section is based on [5, 7, 8, 15, 17, 20, 29, 31].
We already know that the congruent number problem is equivalent to the fact that (X, Y, Z) is a Pythagorean triple, i.e. $X^2 + Y^2 = Z^2$ where the congruent number corresponds to the area of the triangle with sides X, Y and Z.

Proposition 2.7. *Let X, Y and Z be rational numbers and let $u, v \in \mathbb{N}$. Then the following two statements are equivalent:*

1. *(X, Y, Z) is a primitive Pythagorean triple, i.e. X, Y, Z satisfy the equation $X^2 + Y^2 = Z^2$ and $gcd\,(X, Y, Z) = 1$.*

2. *There are u, v with $v > u$, $gcd\,(u, v) = 1$ and $u + v = 1 \mod 2$, which define a right-angled triangle with rational sides X, Y, Z and area $n = uv(v^2 - u^2)$. So n is a congruent number.*

By the following transformations

$$X = 2uv,\ Y = v^2 - u^2 \text{ and } Z = v^2 + u^2.$$

Proof. See Chapter 2 in [5]. □

Hence we have seen that the congruent number problem is equivalent to the problem of finding a primitive Pythagorean triple. This method gives us further the opportunity to produce congruent numbers, as you can see in the following proposition.

Proposition 2.8. *Let u, v be two positive relatively prime integers with $v > u$ and let u and v be of opposite parity, i.e. $u + v = 1 \mod 2$. Then the squarefree part of*

$$uv(v-u)(v+u)$$

is a congruent number.

Proof. For a proof of this statement see [29]. □

This type of congruent numbers is used in our method of finding high rank congruent number elliptic curves.
Here are some examples of congruent numbers less than 200 of the above type, namely

$$14, 15, 21, 34, 39, 41, 46, 55, 65, 69, 85, 102, 111, 119, 138, 141, 145, 154, 161, 165, 194.$$

Another equivalent problem to the congruent number problem is the problem of finding an arithmetic progression of rational squares with common difference n as you can see in the following proposition.

Proposition 2.9. *Let $n > 0$ be a squarefree congruent natural number and let X, Y, Z and x with $X < Y < Z$ denote rational numbers. Then the following statements are equivalent:*

1. *The number n is a congruent number i.e. n is the area of a right-angled triangle with sides X, Y and Z where Z is the hypotenuse.*
2. *There is a rational number x such that $x, x+n$ and $x-n$ are squares of rationals i.e. we have an arithmetic progression of three rational squares with common difference n.*

By the following transformations

$$X, Y, Z \to x = (Z/2)^2$$
$$x \to X = \sqrt{x+n} - \sqrt{x-n},\ \ Y = \sqrt{x+n} + \sqrt{x-n},\ \ Z = 2\sqrt{x}.$$

Proof. See Chapter 1 in [20]. □

There are also more straight forward classes of congruent numbers as we state in the following section.

2.3. Classes of congruent numbers

We will present some examples for congruent numbers less than 200. For further information see [2, 7] and [31]. As seen above Conjecture 2.6 tells us that primes p which satisfy

$$p \equiv 5,\ 6,\ 7 \mod 8,$$

are congruent numbers under the assumption of the Birch and Swinnerton-Dyer Conjecture. For example the following congruent numbers are congruent $5 \mod 8$,

$$5,\ 13,\ 29,\ 37,\ 53,\ 61,\ 101,\ 109,\ 149,\ 157,\ 173,\ 181,\ 197.$$

Furthermore these congruent numbers

$$7,\ 23,\ 31,\ 47,\ 71,\ 79,\ 127,\ 151,\ 167,\ 191,\ 199,$$

are congruent $7 \mod 8$. Both classes were introduced by Stevens in 1975.
There are also other classes of congruent numbers as you can see in the following.
If $p \equiv 3 \mod 8$, then

$$n = 2p,$$

is a congruent number. Therefore the following positive integers are congruent numbers

$$6,\ 22,\ 38,\ 86,\ 118,\ 134,\ 166.$$

This type was discovered by Heegner in 1952 and by Birch in 1968.
Let $p \equiv 3 \mod 8$ and $q \equiv 5 \mod 8$, then

$$n = 2pq,$$

is a congruent number. Examples for this type of congruent numbers are

$$30,\ 78,\ 110,\ 174,\ 190.$$

Let $p \equiv 5 \mod 8$ and $q \equiv 7 \mod 8$, then

$$n = 2pq,$$

is a congruent number. Here 70 and 182 are examples for this type of congruent numbers.

2.4. From congruent numbers to elliptic curves

In this section we present a relation between congruent numbers and congruent number elliptic curves E_n. We can start with a Pythagorean triple and transform it into an elliptic curve equation as you can see in [5, 7, 17, 18] and [20].

As we already know, a squarefree natural number n is a congruent number if and only if there are some rationals X, Y and Z such that the equations (2.1) and (2.2) are satisfied simultaneously.
We use this system of equations as our starting point and try to get to an equation of an elliptic curve.
First of all we multiply Equation (2.2) by 4. Then we obtain

$$(X+Y)^2 = X^2 + Y^2 + 2XY = Z^2 + 4n, \tag{2.3}$$

by adding the previous result to Equation (2.1). Furthermore we get

$$(X-Y)^2 = X^2 + Y^2 - 2XY = Z^2 - 4n, \tag{2.4}$$

by subtracting the multiple of (2.2) from (2.1). Now we divide these equations by 4 which gives

$$\left(\frac{X+Y}{2}\right)^2 = \left(\frac{Z}{2}\right)^2 + n, \tag{2.5}$$

and

$$\left(\frac{X-Y}{2}\right)^2 = \left(\frac{Z}{2}\right)^2 - n. \tag{2.6}$$

Multiplication of the above equations yields to

$$\left(\frac{X^2-Y^2}{4}\right)^2 = \left(\frac{Z}{2}\right)^4 - n^2. \tag{2.7}$$

Now substituting $v := (X^2 - Y^2/4)$ and $u := (Z/2)$ gives

$$(v)^2 = (u)^4 - n^2. \tag{2.8}$$

By multiplication with u^2 we obtain

$$(uv)^2 = (u)^6 - n^2u^2. \tag{2.9}$$

Now replacing $x := u^2$ and $y := (uv)$ delivers our required equation of an elliptic curve

$$E_n : y^2 = x^3 - n^2x. \tag{2.10}$$

We call such an elliptic curve a congruent number elliptic curve or shorter CN-elliptic curve.
So this shows the relation between congruent numbers and elliptic curves.

Theorem 2.10. *The positive integer n is a congruent number if and only if $E_n(\mathbb{Q})$ has a rank $r > 0$, i.e., the elliptic curve E_n has infinitely many $\mathbb{Q}$-rational points.*

Proof. See Proposition 18 in [20]. □

2.5. Congruent number elliptic curves

In this section we will give some basic facts about congruent number elliptic curves

$$E_n : y^2 = x^3 - n^2 x.$$

In Figure 2.2 such a curve is illustrated. As we have already seen in Theorem 2.10 in the previous section there is a equivalence between congruent numbers and congruent number elliptic curves. For further information see the references [5, 6, 7, 11, 15, 17, 20, 35, 36].

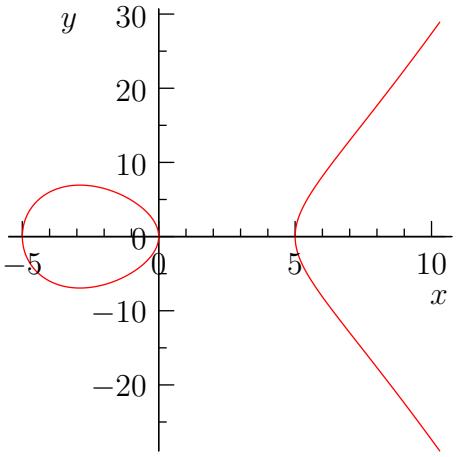

Figure 2.2.: The elliptic curve $E_n : y^2 = x^3 - 5^2 x$.

As in the general case we can transform the above affine equation into an equation in the projective plane

$$Y^2 Z = X^3 - n^2 X Z^2.$$

Before we describe the torsion subgroup $E_n(\mathbb{Q})_{tors}$ we define the quadratic twist and the reduction of a congruent number elliptic curve modulo p.

Definition 2.11. *Let E_n be a congruent number elliptic curve. Then we define the d-th quadratic twist of E_n as*

$$E_n{}^d : dy^2 = x^3 - n^2 x.$$

We already presented the following example, but now we are able to give a reason for the statement in Example 1.18.

Example 2.12. *The curves $E : y^2 = x^3 - 4x$ and $E' : y^2 = x^3 - 25x$ defined over the rational numbers $\mathbb{Q}$ cannot be transformed into the other although they have the same j-invariant $j(E) = j(E') = 1728$. The reason for that is that E' has infinitely many rational points e.g. all integer multiples of the point $(-4, 6)$, whereas E has only the following four rational points $\mathcal{O}, (2, 0), (-2, 0)$ and $(0, 0)$. Therefore we cannot transform neither E nor E' into the other by functions defined over $\mathbb{Q}$, but it is possible in $\mathbb{Q}(\sqrt{10})$ as you can see in Section 2.6 in [35].*

Furthermore we need the definition of the reduction of an congruent number elliptic curve.

Definition 2.13. *Let p be a prime number with $p \nmid \Delta$ and let E_n be a congruent number elliptic curve over $\mathbb{Q}$. Then we denote the reduction of E_n modulo p by*

$$\tilde{E}_n : y^2 = x^3 - \tilde{n}^2 x.$$

The set of points on $\tilde{E}_n$ is then given by

$$\tilde{E}_n(\mathbb{F}_p) = \{\mathcal{O}\} \cup \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 \equiv x^3 - n^2 x \mod p\}.$$

Now we present an example of this method.

Example 2.14. *Let $E_5 : y^2 = x^3 - 5^2 x$ be an elliptic curve and suppose we consider it over the finite field $\mathbb{F}_3$. Then*

$$\begin{aligned} \tilde{E}_5(\mathbb{F}_3) &= \{(x, y) \in \mathbb{F}_3 \times \mathbb{F}_3 : y^2 \equiv x^3 - 5^2 x \mod 3\} \\ &= \{\mathcal{O}, (0,0), (1,0), (2,0)\}, \end{aligned}$$

and hence $\tilde{E}_5(\mathbb{F}_3) \simeq \mathbb{Z}_2 \oplus \mathbb{Z}_2$.

Straightaway we describe the points of finite order which are the elements of the torsion subgroup $E_n(\mathbb{Q})_{tors}$.

Proposition 2.15. *The torsion subgroup $E_n(\mathbb{Q})_{tors}$ of E_n contains only the following four rational points of finite order*

$$E_n(\mathbb{Q})_{tors} = \{\mathcal{O}, (0,0), (-n, 0), (n, 0)\}.$$

Hence $E_n(\mathbb{Q})_{tors} \simeq \mathbb{Z}_2 \oplus \mathbb{Z}_2$.

Proof. See Section $I.9$ in [20]. □

Example 2.16. *Let us consider the elliptic curve $E_n : y^2 = x^3 - 5^2 x$. Then the points of finite order are*

$$E_5(\mathbb{Q})_{tors} = \{\mathcal{O}, (0,0), (-5, 0), (+5, 0)\}.$$

Before we state the famous Birch and Swinnerton-Dyer conjecture we first define the trace of Frobenius a_p and the L-function of a CN-elliptic curve. The same definition holds true for any other elliptic curve but we only need the case of a congruent number elliptic curve. We define a_p in the following way.

Definition 2.17. *Let $\tilde{E}(\mathbb{F}_p)$ be the reduction of E modulo p. Where E has good reduction at p. Then we define the trace of Frobenius as*

$$a_p = p + 1 - \#\tilde{E}(\mathbb{F}_p).$$

Definition 2.18. *Let E_n be a CN-elliptic curve defined over $\mathbb{Q}$. Then we define the Hasse-Weil L-function of E_n as*

$$L(s, E_n) = \prod_{p \nmid 2n} (1 - a_p\, p^{-s} + p^{1-2s})^{-1},$$

where s is a complex number with a sufficiently large real part.

The Birch and Swinnerton-Dyer conjecture is one of the famous Millenium Prize Problems, which had been announced in 2000 by the Clay Mathematics Institute of Cambrige, Massachusetts. Anyone who is able to solve one of these problems, receives a price which is worth one million dollars. This list of still unsolved problems else contains the Riemann Hypothesis and the P versus NP problem. The aim of this list is to support mathematicians to solve deep problems. The problems on this list had already been known before this announcement, by contrast to the 23 problems proposed by Hilbert in 1900. This information is taken from [5] and [6].
Now we present the Weak Birch and Swinnerton-Dyer conjecture which is enough for our purposes.

Conjecture 2.19 (Weak Birch and Swinnerton-Dyer conjecture (BSD))**.** *Let E_n be a congruent number elliptic curve defined over $\mathbb{Q}$. Then E_n has rank > 0 if and only if $L(1, E_n) = 0$.*

Remark 2.20. *The Birch and Swinnerton-Dyer conjecture is much more general than stated here but this version is sufficient for our purposes.*

Proposition 2.21. *Suppose the weak Birch and Swinnerton-Dyer Conjecture holds true, then n is a congruent number if and only if $L(1, E_n) = 0$.*

Proof. See the proof of Proposition 8.2 in [5]. □

Another big improvement in the twentieth century concerning a classification of congruent numbers is due to Jerrold B. Tunnell. See [7] and [17] for further details.

Theorem 2.22 (Tunnell)**.** *Let n be a positive number. If we define the following sets*

$$\begin{aligned}
A_n &= \{(x, y, z) \in \mathbb{Z}^3 \mid n = x^2 + 2y^2 + 8z^2\},\\
B_n &= \{(x, y, z) \in \mathbb{Z}^3 \mid n = x^2 + 2y^2 + 32z^2\},\\
C_n &= \{(x, y, z) \in \mathbb{Z}^3 \mid n/2 = x^2 + 4y^2 + 8z^2\},\\
D_n &= \{(x, y, z) \in \mathbb{Z}^3 \mid n/2 = x^2 + 4y^2 + 32z^2\},
\end{aligned}$$

then the following statements hold:

(i) Suppose n is odd. If n is a congruent number then $\#A_n = 2\#B_n$.

(ii) Suppose n is even. Then the property that n is a congruent number implies $\#C_n = 2\#D_n$.

Remark 2.23. *If the weak Birch and Swinnerton-Dyer conjecture is true, then*

$$\#A_n = 2\#B_n$$

implies that an odd number n is congruent. In the case where n is even the fulfilled equation

$$\#C_n = 2\#D_n$$

implies that n is congruent.

So if the weak Birch and Swinnerton-Dyer conjecture is true, we can reduce the congruent number problem to determining the cardinality of finite sets, where the sets depend on the parity of the given number.

3. Finding elliptic curves of high rank

The goal of this thesis is finding congruent number elliptic curves

$$E_n : y^2 = x^3 - n^2 x$$

of high rank. Before we give a general approach of finding CN-elliptic curves of high rank we present high ranks of elliptic curves found so far. This chapter is based on [1, 13, 14, 15, 18, 25] and [35].

3.1. Rank records

Computing the rank of a given elliptic curve is a very tough job but the following table contains high ranks of elliptic curves found so far. The largest currently known rank of an elliptic curve over the rational numbers is 28 and was found by Noam D. Elkies in 2006. Notice, that the given ranks in Table 3.1 present only lower bounds for suspected ranks. By contrast, the record for the largest exactly known rank of an elliptic curve over $\mathbb{Q}$ is 19 and this curve was also given by Elkies in 2009. The information of this section is based on [1, 13, 14] and [25].
Since we are searching for high ranks of CN-elliptic curves, we present in Table 3.2 the largest ranks of CN-elliptic curves found so far.
Column 'Number n' states the first found integer n such that the corresponding elliptic curve $E_n : y^2 = x^3 - n^2 x$ had the rank given in column 'Rank'.
Now we continue with a general approach of finding high rank elliptic curves.

3.2. General idea for finding elliptic curves with relatively high rank

This section is based on [13] and [14].

1. **Construction:** We first produce a family of elliptic curves defined over $\mathbb{Q}$, which we believe (or know) to contain curves of high rank. A possible method for finding a family of elliptic curves which contains relatively high rank curves is using elliptic curves which are induced by Diophantine triples.

Rank $\geq$	Year	Authors
3	1938	Billing
4	1945	Wiman
6	1974	Pommerance
7	1975	Pommerance
8	1977	Grunewald - Zimmert
9	1977	Brumer-Kramer
12	1982	Mestre
14	1986	Mestre
15	1991	Mestre
17	1992	Nagao
19	1992	Fermigier
20	1993	Nagao
21	1994	Nagao-Kouya
22	1996	Fermigier
23	1998	Martin - McMillen
24	2000	Martin - McMillen
28	2006	Elkies

Table 3.1.: Rank records of elliptic curves

Rank	Year	Author	Number n
5	2000	Rogers	4 132 814 070
6	2000	Rogers	61 471 349 610
7	2004	Rogers	797 507 543 735

Table 3.2.: Rank records of CN-elliptic curve

2. **Sifting:** Then we compute some data which gives us information about the rank (upper, lower bounds) for each curve in the above family. Based on this information we choose good candidates for the next step. It is more likely that elliptic curves with relatively high Mestre-Nagao sums have larger ranks. Elliptic curves with relatively high 2-Selmer-ranks $s(n)$ are also assumed to be good candidates for high rank curves.

3. **Computing:** For each curve in this small list of best candidates we try to compute the rank (or at least a lower bound for the rank).

Almost all methods for phases 1 and 2 were introduced by Jean-Francois Mestre.

3.3. Our approach

We are searching for congruent number elliptic curves $E_n : y^2 = x^3 - n^2x$ of high rank and therefore we are working with positive squarefree numbers n corresponding to E_n.

Before we state our approach in detail we have to define Monsky's formula for $s(n)$ and Mestre-Nagao's sum. See Section 3 in [15].

Definition 3.1 (Monsky's formula)**.** *Suppose n is a positive squarefree integer, where $p_1, \ldots, p_t$ describe the odd prime factors of n. Then we define the four $t \times t$-matrices A, D_{-2}, D_{-1} and D_2 as follows*

$$\begin{aligned} A &= (a_{ij}) \\ D_{-2} = D_{-1} &= D_2 = (d_{ii}), \end{aligned}$$

where A is a square matrix and D_{-2}, D_{-1} and D_2 are diagonal matrices. If $i \neq j$ we define

$$a_{ij} = \begin{cases} 0, & \text{if } (\frac{p_j}{p_i}) = 1, \\ 1, & \text{if } (\frac{p_j}{p_i}) = -1, \end{cases}$$

where $(\frac{p_j}{p_i})$ describes the Legendre symbol as in Definition 1.53.
For $i = j$ we define

$$a_{ii} = \sum_{j:j\neq i} a_{ij}.$$

This means, that the diagonal element is the sum of the other elements in this column.
For the diagonal matrices D_ℓ with $\ell \in \{-2, -1, 2\}$ we have the following condition

$$d_{ii} = \begin{cases} 0, & \text{if } (\frac{\ell}{p_i}) = 1; \\ 1, & \text{if } (\frac{\ell}{p_i}) = -1. \end{cases}$$

Furthermore we need the two $2t \times 2t$-matrices M_{odd} and M_{even} depending on n

$$M_{odd} = \begin{bmatrix} A + D_2 & D_2 \\ D_2 & A + D_{-2} \end{bmatrix} \quad \text{and} \quad M_{even} = \begin{bmatrix} D_2 & A + D_2 \\ A^T + D_2 & D_{-1} \end{bmatrix}.$$

Now we can compute $s(n)$ as

$$s(n) = \begin{cases} 2t - \operatorname{rank}_{\mathbb{F}_2}(M_{odd}), & \text{if } n \text{ is odd}; \\ 2t - \operatorname{rank}_{\mathbb{F}_2}(M_{even}), & \text{if } n \text{ is even}. \end{cases}$$

A further definition we need in our approach of finding congruent number elliptic curves of high rank is the Mestre-Nagao sum. For further information have a look at Section 4 in [15].

Definition 3.2. *Mestre-Nagao's sum is defined as*

$$\begin{aligned} S(N,E) &= S(N,E_n) = S(N,n) \\ &= \sum_{p\in P_N} \left(1 - \frac{p+1}{\#E(\mathbb{F}_p)}\right) \log p \\ &= \sum_{p\in P_N} \frac{\#E(\mathbb{F}_p) - p - 1}{\#E(\mathbb{F}_p)} \log p \\ &= \sum_{p\in P_N} \frac{-a_p + 2}{p + 1 - a_p} \log p, \end{aligned}$$

where P_N describes the set of all primes which are less than N.

Now we come to our approach.

1. **Construction:** First of all we choose a parameter s which is used to ensure that all elliptic curves in our family have $s(n) \geq s$. We produce a set T of elliptic curves E_n defined over $\mathbb{Q}$, by searching for squarefree congruent numbers n with $s(n) \geq s$.

2. **Sifting:** Then we try to sieve through our family of elliptic curves to find the best candidates. This is done by using the set

   $$\mathcal{M}_s = \{(N_i, M_i) : 0 < N_1 < \cdots < N_k,\ \ 0 < M_i,\ \ 1 \leq i \leq k\},$$

   where k is a positive integer, as bound. Then we eliminate all possible candidates which have a Mestre-Nagao-Sum

   $$S(N_i, n) < M_i,$$

   for all $1 \leq i \leq k$. The set T_s^j, which is the last set that contains any candidates, is used in the next step.

3. **Computing:** Now for each curve E_n in this small list of best candidates we try to compute the rank with the help of Cremona's MWRANK function.

3.4. Results

The following tables give high rank CN-elliptic curves $E_n : y^2 = x^3 - n^2x$ found by implementing the previously given algorithm during my master's thesis. Furthermore these tables contain u and v such that $n = \text{sqrfr\_prt}(uv(v-u)(v+u))$, the rank of E_n and the Mestre-Nagao sum Ms of each curve.

n	u	v	rank	Ms
6 611 719 866	2976	7633	6	39.55
61 471 349 610	134	779	6	36.84
94 823 967 361	74752	82249	6	37.82
129 448 648 329	269	25900	6	40.18
179 483 163 699	9717	9764	6	32.19
208 645 752 554	1751	4018	6	36.49
213 691 672 290	245	3502	6	39.56
227 011 077 345	1936	6305	6	35.66
248 767 798 521	13328	13369	6	33.34
344 731 563 386	10406	13275	6	37.80
531 670 544 130	3424	5739	6	33.60
797 804 045 274	2450	7633	6	42.78
898 811 499 201	12716	21627	6	33.62
1 351 528 542 210	8232	80645	6	34.90
1 440 993 982 946	28543	40064	6	32.62
1 544 991 154 746	3991	9538	6	35.62
1 663 586 838 899	3708	12869	6	35.63
2 280 190 889 130	1682	10537	6	49.09
2 993 601 315 705	9152	55447	6	35.98
4 707 197 976 210	15337	28920	6	35.80
5 190 465 353 874	809	33638	6	31.83
6 128 804 829 210	7442	15793	6	32.26
8 231 905 771 386	3827	9416	6	30.43
9 033 322 597 530	762	2365	6	32.46
16 051 126 378 931	39917	42500	6	33.70
17 434 310 103 210	9112	18487	6	31.36
18 361 479 032 130	27590	29887	6	33.62

Table 3.3.: CN-elliptic curves of high rank part 1

n	u	v	rank	Ms
20 873 924 653 090	827	22338	6	31.26
21 193 369 120 770	71680	76219	6	36.92
23 148 215 699 289	25488	29449	6	35.15
46 485 304 142 530	6274	23005	6	40.27
87 390 970 562 434	23713	60896	6	31.68
90 181 020 280 890	3970	6777	6	40.34
121 110 989 796 834	86	32775	6	41.90
165 130 972 136 130	4914	7901	6	36.56
170 078 314 006 986	33374	36423	6	34.56
197 385 243 713 034	44914	66833	6	30.34
205 873 902 867 745	31744	85455	6	34.70
257 306 357 070 354	216	35003	6	39.04
339 507 119 347 242	791	22066	6	39.09
405 941 588 462 586	19097	22846	6	31.92
420 824 792 637 249	9100	82739	6	35.09
444 724 421 083 665	14912	18105	6	30.06
455 089 600 428 474	22	27451	6	37.48
674 928 351 312 369	45424	53103	6	39.65
701 641 549 951 530	37195	37994	6	35.66
846 249 312 638 730	3131	6970	6	36.74
896 740 623 261 329	7029	17612	6	45.32
1 056 710 141 801 930	769	23134	6	38.52
1 071 795 744 409 866	13664	33511	6	36.99
1 799 308 052 046 681	18204	30943	6	31.66
1 902 736 244 939 034	50279	89954	6	40.41
4 132 282 640 911 035	51701	69904	6	32.23
4 194 267 377 608 770	7537	24838	6	35.67
5 262 441 841 603 947	14300	35309	6	42.20
9 294 013 431 797 010	3066	14689	6	31.06
27 401 430 048 260 114	17656	40943	6	34.82
75 136 867 709 572 130	25470	36971	6	40.08

Table 3.4.: CN-elliptic curves of high rank part 2

4. Implementation

In this chapter we will present some details about our implementation. During the implementation of this algorithm, we were confronted with several problems which we will describe in the following.
At the very beginning we had a big problem concerning the time consumption for creating congruent numbers. To recognize where this enormous time-related overhead comes from we used cProfile, which gives information how often a function is called and how long it is executed. See [28] for further information. With the help of this profiler we were able to detect that the time consumption is mainly produced by the following line, which adds a single element to a given set T :

```
T = T.union(Set([n]));.
```

By testing several possibilities for improvement we observed, that it is the best way to use a list instead of a set for adding elements. Before we return the computed list, we transform it into a set. So we need much less time and we also have the advantage that a congruent number in the set T can not occur more than once. This fact is important to minimize the effort in the next steps.
After solving this problem another time problem occurred. We already knew the time intensive parts from cProfile. The next idea was to parallelize these parts to further reduce the amount of time. Independent computations which are done in a for-loop can easily be parallelized. This is the reason why we were able to parallelize the functions $precompute\_squarefree\_parts\_from\_to(\ldots)$, $get\_congruent\_numbers(\ldots)$, $get\_set\_Ts(\ldots)$, $parallel\_step2(\ldots)$ and $compute\_rank(\ldots)$.
Therefore we consider parallel computing in SageMath which will be used in the precomputation as well as in the computation phase. So we have a look at the usage in SageMath by considering an easy example. For more information see [30].
Let us suppose, that the following function, which sums all integers between a and b, needs too much time.

```
def sum(a,b):
  sum = 0;
  for i in xrange(a,b+1):
    sum = sum + i;
  return sum;
```

For example this function needs for the function call

```
sum(1,10^8)
 5000000050000000
```

about 9.9 seconds. So we want to use parallelization to improve the needed amount of time. Therefore we have to write '@parallel' before the definition of the function.

```
@parallel
def sum_parallel(a,b):
  sum = 0;
  for i in xrange(a,b+1):
    sum = sum + i;
  return sum;
```

Now, if we would like to call the function in parallel, we have to modify the input. This means, we have to divide the input into parts of nearly the same size. Because otherwise parallelization would not cause such a big improvement. Suppose we would like to sum again the numbers from 1 to 10^8 and we would like to use four CPU cores. Then we have to divide the input into four parts

```
input=[(1,25000000),(25000001,50000000),(50000001,75000000),(75000001,10^8)].
```

The input argument of a '@parallel'-labeled function, is a list containing the input arguments for each single function call.
Then we can call the function in parallel and obtain the following results.

```
sorted(list(sum_parallel(input)))
 (((25000001,  50000000), {}), 937500012500000)  -> 2.41 s
 (((50000001,  75000000), {}), 1562500012500000) -> 2.41 s
 (((       1,  25000000), {}), 312500012500000)  -> 2.41 s
 (((75000001, 100000000), {}), 2187500012500000) -> 2.41 s.
```

To get the same result, namely 5000000050000000, we have to sum the last entries in each row. So it is important not to forget to combine the output of each parallel computation to avoid any data loss. However, here we have a total time of 2.41 seconds which is four times faster as above.
Now let us continue with more detailed descriptions and problems regarding the precomputations and computations.

4.1. Precomputations

Before we give a description of the computations we start with some precomputations which are needed. A very helpful hint given by Andrej Dujella, was to do the computations needed for the squarefree part of numbers at the beginning and save them in a global list. We do the same for a_p such that these computations has to be done only once.

By the fundamental theorem of arithmetic we know that every integer $n > 1$ can be written uniquely as the product of prime numbers

$$n = \prod_{i=1}^{k} p_i^{a_i},$$

where p_i are the prime factors of n and a_i are the corresponding exponents.
Now we can define the squarefree part of an integer easily.

Definition 4.1. *The squarefree part of a positive integer n is*

$$\text{sqrfr\_prt}(n) := \prod_{i=1}^{k} p_i^{a_i \mod 2},$$

i.e., the squarefree part of an integer is the product of primes which have an odd exponent in the prime factorization.

Proposition 4.2. *Every positive integer u can be written uniquely in the form*

$$u = \text{sqrfr\_prt}(u)(u')^2,$$

where $\text{sqrfr\_prt}(u)$ *is the squarefree part of u and $(u')^2$ is the largest perfect square dividing n.*

Proof. We consider the prime factorization of $u = \prod_{i=1}^{k} p_i^{a_i}$. Then we define

$$b_i = \begin{cases} a_i & \text{if } a_i \text{ is even} \\ a_i - 1 & \text{if } a_i \text{ is odd,} \end{cases}$$

and

$$c_i = a_i - b_i$$

for all $i \in \{1, \ldots, k\}$.
So every b_i is even and every c_i is either 1 or 0. We set

$$b = \prod_{i=1}^{k} p_i^{b_i}$$
$$c = \prod_{i=1}^{k} p_i^{c_i}.$$

Hence we have

$$bc = \prod_{i=1}^{k} p_i^{b_i + c_i} = \prod_{i=1}^{k} p_i^{a_i} = u.$$

Since in the prime factorization of b all exponents are even, we know, that b is a perfect square. Now we have to show that c is squarefree. Therefore we assume that there is an $a > 1$ with $a^2 \mid c$ and let p be a prime dividing a. Then $p^2 \mid a^2$ and hence $p^2 \mid c$, but then the prime p in the prime factorization of c must have an exponent greater or equal to 2, which is impossible, since all exponents in the prime factorization of c are at most 1. So c is squarefree. □

Proposition 4.3. *The squarefree part* $\text{sqrfr\_prt}(n)$ *is a multiplicative function i.e., if* $n = u \cdot v$ *and* u *and* v *are coprime then*

$$\text{sqrfr\_prt}(n) = \text{sqrfr\_prt}(u) \cdot \text{sqrfr\_prt}(v).$$

Proof. Since u and v are coprime we know that $gcd\,(u, v) = 1$ and let $u'' = \text{sqrfr\_prt}(u)$ and $v'' = \text{sqrfr\_prt}(v)$ be the squarefree parts of u and v. By the above proposition we know that we can write

$$u = u'' \cdot (u')^2,$$

and

$$v = v'' \cdot (v')^2,$$

where $(u')^2$ and $(v')^2$ are perfect squares. Then

$$\text{sqrfr\_prt}(u)\,\text{sqrfr\_prt}(v) = u''v'',$$

with $gcd\,(u'', v'') = 1$. Furthermore

$$uv = u''v''(u'v')^2,$$

with $u''v''$ squarefree. Hence $\text{sqrfr\_prt}(n) = \text{sqrfr\_prt}(uv) = \text{sqrfr\_prt}(u)\,\text{sqrfr\_prt}(v)$.

□

Corollary 4.4. *Let* u *and* v *satisfy the conditions of Proposition 2.8 and let*

$$n = uv(v - u)(v + u).$$

Then $u, v, (v - u)$ *and* $(v + u)$ *are relatively prime and hence*

$$\text{sqrfr\_prt}(n) = \text{sqrfr\_prt}(u) \cdot \text{sqrfr\_prt}(v) \cdot \text{sqrfr\_prt}(v - u) \cdot \text{sqrfr\_prt}(v + u).$$

As Watkins stated in [36].

The advantage that Corollary 4.4 provides is that it is sufficient to compute the squarefree pars of u and v where $1 \leq u, v \leq 10^6$ and save them in a global list, since $n = \text{sqrfr\_prt}(uv(v - u)(v + u))$. Then we can easily compute the squarefree part of n, by multiplying the squarefree parts of the terms u, v, $(v - u)$, $(v + u)$. We use this fact in the function *precompute_ap*(...).
In the following we will describe the functions of our program.

Function 1: *divide_input*($l\_bound, u\_bound, ncpus$)

input : l_bound ... a lower bound for dividing the input,
u_bound ... an upper bound for dividing the input,
$ncpus$ ... the number of CPUs.

output: An input needed for the parallel function
$precompute\_squarefree\_parts\_from\_to(\ldots)$.

This function divides the range of numbers into $ncpus$ equal parts for computing the squarefree parts by the parallel function
$precompute\_squarefree\_parts\_from\_to(\ldots)$.

Function 2: @*parallel* $precompute\_squarefree\_parts\_from\_to(l\_bound, u\_bound)$

input : l_bound ... a lower bound for computing the squarefree parts,
u_bound ... an upper bound for computing the squarefree parts.

output: The squarefree parts of the numbers in the given range.

This function computes and returns the squarefree parts of the numbers in the given range $[l\_bound, u\_bound]$.

Function 3: $precompute\_squarefree\_parts(l\_bound, u\_bound, ncpus)$

input : l_bound ... a lower bound for computing the squarefree parts,
u_bound ... an upper bound for computing the squarefree parts,
$ncpus$ ... the number of CPUs.

output: A list of squarefree parts of the numbers given in $[l\_bound, u\_bound]$.

This function divides the given input by calling *divide_input*(...) for the parallel function $precompute\_squarefree\_parts\_from\_to(\ldots)$. This function computes the squarefree parts and returns the results in a list afterwards.

Function 4: *precompute_ap*()

output: The a_ps for p in PN for the CN-elliptic curve E_1.

Returns the a_p for all primes p in the global set PN for the elliptic curve E_1.

Function 5: *precompute*(*bound_primes*, *ncpus*)

input : *bound_primes* ... a bound for the prime numbers used in computing a_p,
ncpus ... the number of CPUs.

This function computes the a_p for the elliptic curve E_1 for all primes which are less than *bound_primes* and it computes the squarefree parts of all numbers which are less than 10^6.

The results of the function *precompute*(...) are stored in global variables such that we can access this data at any time.

4.2. Computations

4.2.1. Construction

Our function $step1(\ldots)$ corresponds to the construction step in the idea of finding elliptic curves with relatively high rank in Chapter 3. Here we try to produce a family of elliptic curves defined over $\mathbb{Q}$ which is assumed to contain high rank curves. In our case we suppose that CN-elliptic curves $E_n : y^2 = x^3 - n^2x$, with n a squarefree congruent number and $s(n) \geq s = 6$ have large ranks. Hence the following functions are needed for this step.

Function 6: @*parallel get_congruent_numbers*$(u\_low, u\_upp, v\_low, v\_upp)$

---

input : $u\_low \ldots$ a lower bound for u,
$u\_upp \ldots$ an upper bound for u,
$v\_low \ldots$ a lower bound for v,
$v\_upp \ldots$ an upper bound for v.

output: A list of congruent numbers T and a list *uv_map_temp* of the corresponding values for u and v of the numbers in list T.

This function computes all congruent numbers of the form $n = \text{sqrfr\_prt}(uv(v-u)(v+u))$ where $u \in [u\_low, u\_upp]$ and $v \in [v\_low, v\_upp]$. See Proposition 2.8 for the reason why n is a congruent number and see Corollary 4.4 for the reason why the squarefree part is multiplicative, if $u < v$, u and v are of opposite parity and u and v are coprime. This algorithm returns a list T of congruent numbers and a list *uv_map_temp* which contains the corresponding u and v values, which will be needed for the output.

Function 7: *divide_input_get_sqfree_numbers*$(u\_low, u\_upp, v\_low, v\_upp, ncpus)$

---

input : $u\_low \ldots$ a lower bound for u,
$u\_upp \ldots$ an upper bound for u,
$v\_low \ldots$ a lower bound for v,
$v\_upp \ldots$ an upper bound for v,
$ncpus \ldots$ number of used CPUs.

output: A list *input_new* of input for the parallel function *get_congruent_numbers*$(\ldots)$.

This function tries to divide the input for the parallel function *get_congruent_numbers*$(\ldots)$ into *npus* parts of nearly the same size to optimize the parallelization step.

In *choose_set_T_of_congruent_numbers*$(\ldots)$ we use the improvement of using a list T instead of a set T. Because the union of new elements with the already given set is very time consuming. So calling $Set(T)$ only once provides an enormous time reduction.

Function 8: *choose_set_T_of_congruent_numbers*$(u\_low, u\_upp, v\_low, v\_upp, ncpus)$

input : u_low ... a lower bound for u,
u_upp ... an upper bound for u,
v_low ... a lower bound for v,
v_upp ... an upper bound for v,
$ncpus$ ... number of used CPUs.

output: A set T of congruent numbers

This function calls $divide\_input\_get\_sqfree\_numbers(\ldots)$ to get the input, needed for the parallel function $get\_congruent\_numbers(\ldots)$ and assembles the single outputs of each parallel function call to a common list T afterwards. Before returning we transform the list into a set to eliminate any repeated congruent numbers.

Function 9: *divide_set_T*$(T, ncpus, s)$

input : T ... a set of squarefree congruent numbers,
$ncpus$ ... number of used CPUs,
s ... a bound needed in the function $get\_set\_Ts(\ldots)$.

output: A list *input_new* of input for the parallel function $get\_set\_Ts(\ldots)$.

This function tries to divide the input for the parallel function $get\_set\_Ts(\ldots)$ into $npus$ parts of nearly the same size to optimize the parallelization step.

Function 10: @*parallel* $get\_set\_Ts(from\_number, to\_number, T, s)$

input : $from\_number$ ... first congruent number for computing $s(n)$ in set T,
to_number ... last congruent number for computing $s(n)$ in set T,
T ... a set of congruent numbers,
s ... a lower bound for $s(n)$.

output: A set Ts of congruent numbers with $s(n) \geq s$.

This function computes for each congruent number n in set T, the value of $s(n)$ by Monsky's formula cf. Definition 3.1. If $s(n) >= s$ we put the number n into the set Ts, which will be returned afterwards.

Function 11: $step1(u\_low, u\_upp, v\_low, v\_upp, ncpus)$

input : u_low ... a lower bound for u,
u_upp ... an upper bound for u,
v_low ... a lower bound for v,
v_upp ... an upper bound for v,
$ncpus$ ... number of used CPUs.

output: A set Ts of congruent numbers such that each $n \in Ts$ has $s(n)$ greater or equal to a defined bound.

This function computes for u and v in a given range $u \in [u\_low, u\_upp]$ respectively $v \in [v\_low, v\_upp]$ congruent numbers of the type $n = \text{sqrfr\_prt}(uv(v-u)(v+u))$ by calling the function $choose\_set\_T\_of\_congruent\_numbers(\ldots)$ and stores the output in a set T. Afterwards it removes all numbers $n \in T$ with $s(n) < s = 6$ and stores the other numbers in the set Ts. Since the function $get\_set\_Ts(\ldots)$ is a parallel function we have to assemble the outputs of each parallel execution to one common set Ts, which is returned at the end of this function.

4.2.2. Sifting

Now we consider step two, which is the sifting step. Therefore we need the following functions. This subsection is based on [9].

Function 12: *get_S*$(n, bound\_primes)$

input : n ... a congruent number,
bound_primes ... a prime bound for computing Mestre-Nagao's sum.
output: Mestre-Nagao's sum for the congruent number n.

This function computes Mestre-Nagao's sum for the congruent number n for all prime numbers in the global set P_N with an index less than *bound_primes*. This means, we use all prime numbers up to the *bound_primes*-th prime number. For further information see Definition 3.2.

The function *get_S*(...) computes the Mestre-Nagao sum

$$S(n, N) = \sum_{p \in P_N} \left(1 - \frac{p+1}{\#E(\mathbb{F}_p)}\right) \log p,$$

where P_N describes the set of all prime numbers which are less than N. This computation was speeded up incredibly by the following hint of Andrej Dujella. He advised me to use the fact that the number of points $\#E_n$ on the elliptic curve $E_n : y^2 = x^3 - n^2x$ in Mestre-Nagao's sum equals

$$\#E_n(\mathbb{F}_p) = p + 1 - a_p,$$

where the trace of Frobenius a_p, is already computed during the precomputations. Hence we could remove the call

```
E_count = E.count_points(1);
```

and use

```
E_count = p + 1 - a_p[PN.index(p)];
```

instead. In the above code line

```
PN.index(p)
```

describes the p-th prime number. The new version is much faster since we need the number of points on an elliptic curve in each summand of the Mestre-Nagao sum.
Another hint makes use of the following proposition where we use Definition 2.11.

Remark 4.5. *Let $E_1 : y^2 = x^3 - x$ be the congruent number elliptic curve with $n = 1$. Then*

$$E_1{}^d : y^2 = x^3 - d^2x,$$

is the d-th quadratic twist of E_1.

Proposition 4.6. *Let E be an elliptic curve defined over the finite field $\mathbb{F}_p$ with $p > 2$ a prime and $d \in \mathbb{F}_p^\times$. Furthermore let E^d be the d-th quadratic twist of E. Then*

$$a_p(E^d) = a_p(E) \left(\frac{d}{p}\right),$$

where $a_p(E) = p + 1 - \#E(\mathbb{F}_p)$ and $a_p(E^d) = p + 1 - \#E^d(\mathbb{F}_p)$.

Proof. For a proof of this proposition see Section 7.3 in [9]. □

So it suffices to compute a_p for the congruent number elliptic curve $E_1 : x^3 - x$, since

$$a_p(E_n) = a_p(E_1) \left(\frac{n}{p}\right),$$

where E_n is the n-th quadratic twist of E_1 and $(\frac{n}{p})$ is the Legendre symbol.

With the help of these two facts we could reduce the time consumption of the function *get_S*(...) incredibly. Now we continue with the description of the other functions in this step.

Function 13: *divide_numbers*(*set_of_numbers*, *bound_primes*, *bound*, *ncpus*)

---

input : *set_of_numbers* ... a set of squarefree congruent numbers,
bound_primes ... a list of lists which contains prime numbers up to a given bound,
bound ... a lower bound for the Mestre-Nagao sum,
ncpus ... number of used CPUs.

output: A list of inputs for the parallel function *parallel_step*2(...).

This function tries to divide the input for the parallel function *parallel_step*2(...) into *npus* parts of nearly the same size to optimize the parallelization step. Each element in the returned list consists of a *number_from*, a *number_to*, the previously computed list of lists of prime numbers *bound_primes*, a lower bound for the Mestre-Nagao sum *bound* and the whole list of possible congruent numbers *list_of_numbers*.

Function 14: @*parallel parallel_step*2($nr\_from, nr\_to, bound\_primes, bound, list\_of\_nrs$)

input : nr_from ... the congruent number with the smallest index in our list,
nr_to ... the congruent number with the largest index in our list,
bound_primes ... a list containing lists of prime numbers for Mestre-Nagao's sum,
bound ... a lower bound for Mestre-Nagao's sum,
$list\_of\_nrs$ ... list of all congruent numbers returned by *step*1(...)

output: A list of congruent numbers which have Mestre-Nagao sums $S \geq bound$.

This function computes Mestre-Nagao's sum for a list of congruent numbers $n \in [nr\_from, nr\_to]$ in parallel and removes those candidates which have a Mestre-Nagao sum $< bound$.

Function 15: *compute_prime_bounds*(Ms)

input : Ms ... a list of elements $[N_i, M_i]$.

output: A list *bounds* where each element is a list of primes.

Returns a list where each element *bounds*[i] is a list of primes that are less than N_i.

Function 16: *step*2($Ts, npus$)

input : Ts ... a list of congruent numbers,
ncpus ... number of used CPUs.

output: An array Ts_array of best candidates for CN-elliptic curves of high rank.

First of all this function computes a list of lists of prime numbers for bounds given in Ms. Then it calls the function *divide_numbers*(...) to obtain the needed input format for the parallel function *parallel_step*2(...). This function sieves through the given congruent numbers and returns the elements which are suspected to be good congruent numbers i.e., those n with CN-elliptic curves E_n, that are believed to have relatively high ranks. We think that a curve E_n has relatively high rank, if the Mestre-Nagao sum of n is greater than a lower bound $M_i \in Ms = [[N_i, M_i]]$.

4.2.3. Computing

Now we try to compute the rank for the good candidates.

Function 17: $divide\_candidates\_for\_computing\_rank(Ts\_array, ncpus)$

input : Ts_array ... a list of good squarefree congruent numbers,
$ncpus$ ... number of used CPUs.
output: A list of inputs for the parallel function $compute\_rank(\ldots)$.

This function tries to divide the given list of good congruent numbers for the parallel function $compute\_rank(\ldots)$ into $npus$ parts of nearly the same size to optimize parallelization.

In the function $compute\_rank\_for\_one\_curve(\ldots)$ we use the decorator class $fork$ which allows us to define a timeout such that this function is terminated after $timeout = 1800$ seconds. See [30] for further information. We sometimes had the problem that computing the rank took more than several hours but no rank was returned. On the other hand for most of the candidates computation finishes before this defined timeout. Another problem was that a 'Division by zero'-exception has been raised from time to time, therefore we implemented the exception handling and print any details about a raised exception now.

Function 18: $@fork(timeout = 1800)\ compute\_rank\_for\_one\_curve(n)$

input : n ... a good congruent number.
output: The rank of E_n if computing was successfull.

This function tries to compute the rank of the CN-elliptic curve $E_n : y^2 = x^3 - n^2x$ by Cremona's MWRANK function in 1800 seconds. If the function is not able to do it, computing is aborted to reduce time for difficult curves. If any exception arises, we print the details of this exception.

Computing the rank is another very time consuming part in our implementation. Here, we use again parallelization to reduce the time consumption. If any problem such as a timeout or an exception occurs in the function $compute\_rank(\ldots)$, we save the corresponding n in a list of numbers where no rank has been computed.

Function 19: @*parallel compute_rank*(Ts_array)

input : Ts_array ... a list of good congruent numbers.

output: The number of candidates *counter_candidates* for which we have tried to compute the rank of E_n, an indicator variable $curve\_found$ if a computed rank was ≥ 6, a list of numbers $numbers\_found$ which corresponds to CN-elliptic curves of rank ≥ 6, and a list of numbers for those the rank computation was not successfull *no_rank_curves* and the number of such curves *counter_no_rank_computed*.

This function tries to compute the rank of E_n for each element n in the list Ts_array by calling the function $compute\_rank\_for\_one\_curve(\ldots)$.

Function 20: $step3(Ts\_array, npus)$

input : Ts_array ... a list of good candidates computed by $step2(\ldots)$, $ncpus$ ... number of used CPUs.

output: An array of congruent numbers with an elliptic curve E_n of rank ≥ 6 and a list of congruent numbers where we were not able to compute the rank of E_n.

First of all this function finds the last list element of Ts_array which contains any elements. These elements n are used to compute the rank for the corresponding CN-elliptic curves E_n with the help of the parallel function $compute\_rank(\ldots)$. Again we have to create correct inputs for this function by $divide\_candidates\_for\_computing\_rank(\ldots)$. Finally this function returns the numbers of high rank elliptic curves and curves where computing the rank was not possible.

4.2.4. Main

Now we have to combine the functions given so far, therefore we have the following functions which provides furthermore the possibility to split the given input for faster results.

Function 21: $find\_numbers((u\_low, u\_upp, v\_low, v\_upp, ncpus)$

---

input : u_low ... a lower bound for u,
u_upp ... an upper bound for u,
v_low ... a lower bound for v,
v_upp ... an upper bound for v,
$ncpus$ ... number of used CPUs.

output: An array of congruent numbers with an elliptic curve E_n of rank ≥ 6 and a list of congruent numbers where no rank was computed for E_n.

This function combines all functions defined so far. First of all it calls $step1(\ldots)$ to get a list Ts of squarefree congruent numbers. Afterwards if Ts is not empty it calls $step2(\ldots)$ to get a list Ts_array of good candidates for high rank CN-elliptic curves and finally it calls the function $step3(\ldots)$ for trying to compute the rank of E_n where $n \in Ts\_array$.

Function 22: $get\_split\_input(u\_low, u\_upp, v\_low, v\_upp)$

---

input : u_low ... a lower bound for u,
u_upp ... an upper bound for u,
v_low ... a lower bound for v,
v_upp ... an upper bound for v.

output: A split input for the function $find\_numbers(\ldots)$.

This function splits the input into smaller parts such that the computation of each part does not take so much time. Especially for the case $u\_upp = 10^5$ this is important for a better usability. In this range we get an enormous amount of candidates u and v for computing congruent numbers. So this trick enables us to see that the program is still running. In any other cases we use a much larger ranges. With the help of this function we can call $find\_numbers(\ldots)$ for much smaller parts implicitly.

Function 23: $main(u\_low, u\_upp, v\_low, v\_upp, ncpus, split)$

input : u_low ... a lower bound for u,
u_upp ... an upper bound for u,
v_low ... a lower bound for v,
v_upp ... an upper bound for v,
$ncpus$ ... number of used CPUs,
$split$ ... an indicator if the given input should be split into smaller pieces.

This function calls the function $get\_split\_input(\ldots)$ for the given input if needed i.e., if $split == 1$, and calls afterwards the function $find\_numbers(\ldots)$ for each part of the split input and combines afterwards the returned results. If the given input should not be split then this function just calls $find\_numbers(\ldots)$ with the whole input. In both cases this function prints all results on the screen and writes each $print(\ldots)$ statement also into a produced file.

5. Applications of elliptic curves

In this chapter we compare ordinary cryptosystems to their corresponding elliptic curve versions. The security of these ordinary cryptosystems is based on the discrete logarithm problem (DLP) in finite fields. That is the reason why we can easily transfer those cryptosystems to elliptic curves. However, in practice sub-exponential methods for solving DLP in $\mathbb{F}_q^\times$ such as the baby-step giant-step algorithm, the Pohlig-Hellman algorithm, the Pollard's rho algorithm or the number field sieve are known. These algorithms are often inspired by integer factorization algorithms. The security of these analogous elliptic curve versions is based on the discrete logarithm problem on elliptic curves (ECDLP). Since that is much harder to solve, the necessary key sizes in elliptic curve cryptography are considerably smaller. Nevertheless we obtain a comparable security. This chapter is based on [16, 21, 22, 24] and [27].

5.1. Diffie-Hellman key exchange

We first consider the Diffie-Hellman key exchange in finite fields and on elliptic curves. It is used if two parties want to agree upon a shared secret key, which can be used in symmetric encryption schemes.

5.1.1. Diffie-Hellman

The Diffie-Hellman key exchange was developed by Martin Hellman, Whitfield Diffie and Ralph Merkle and was published in 1976. Let us suppose that Alice and Bob want to agree upon a key on an insecure channel. The key should be a random element of $\mathbb{F}_q^\times$ which can be used in a symmetric cipher. The Diffie-Hellman key exchange is based on the discrete logarithm problem. As long as the Diffie-Hellman assumption holds, a third party is not able to compute the key g^{ab} with g^a and g^b.

Definition 5.1 (Discrete logarithm problem (DLP))**.** *Let $\mathbb{Z}_p^\times$ be a finite cyclic group of order $p-1$. Let $\alpha \in \mathbb{Z}_p^\times$ be a primitive element and $\beta \in \mathbb{Z}_p^\times$ any element. We search for an $x \in \mathbb{Z}_p^\times$ with $1 \leq x \leq p-1$, such that*

$$\alpha^x \equiv \beta \bmod p.$$

Hence $x = \log_\alpha \beta \bmod p$.

Definition 5.2 (Diffie-Hellman problem)**.** *Given a finite cyclic group G of order n, a primitive element $\alpha \in G$ and two elements $A = \alpha^a$ and $B = \alpha^b$ in G. Then the Diffie-Hellman problem is the problem searching the element α^{ab}.*

The Diffie-Hellman problem could be solved by the discrete logarithm problem.

Definition 5.3 (Diffie-Hellman assumption)**.** *It is computationally infeasible to compute g^{ab}, if we only know g^a and g^b. The Diffie-Hellman Assumption holds as long as the discrete logarithm cannot be computed efficiently.*

The key exchange works in the following way.

Protocol:

1. First Alice and Bob agree on a public finite field $\mathbb{F}_q$ and an element $g \in \mathbb{F}_q$ such that g is a generator of the multiplicative group $\mathbb{F}_q^\times$.
2. Alice chooses a random integer $a \in \{1 \dots q-1\}$ which she keeps secret. Furthermore she computes $A = g^a \in \mathbb{F}_q$ and sends A to Bob.
3. Bob chooses a random integer $b \in \{1 \dots q-1\}$ which he keeps secret. Afterwards he computes $B = g^b \in \mathbb{F}_q$ and sends B to Alice.
4. Each of them computes their shared secret key K. Alice uses the information of Bob $K = B^a$ and Bob uses the information of Alice $K = A^b$ in $\mathbb{F}_q$.

In the following we consider the elliptic curve Diffie-Hellman key exchange.

5.1.2. Elliptic curve Diffie-Hellman

Here Alice and Bob want to agree on a shared secret key again, which could be used for a symmetric encryption scheme. As long as the Diffie-Hellman assumption (Definition 5.3) holds, a third party is not able to compute the key abB knowing aB and bB.

Definition 5.4 (Elliptic curve discrete logarithm problem (ECDLP))**.** *Let E be an elliptic curve over $\mathbb{F}_q$ and let B and P be points on E. We search for an integer $x \in \mathbb{Z}$, such that $xB = P$, but such an integer x does not need to exist. This problem is called discrete logarithm problem on E (to the base B).*

Next we give the protocol of the elliptic curve Diffie-Hellman key exchange (ECDH).

Protocol:

1. First Alice and Bob agree on a public finite field $\mathbb{F}_q$, an elliptic curve E over $\mathbb{F}_q$ and a point $B \in E$ to serve as their base.
2. To create a shared key Alice chooses a random integer a and keeps it secret.
3. Then Alice computes $aB \in E$ and makes it public.
4. Bob does the same: he chooses a random integer b and makes $bB \in E$ public.

5. The shared secret key is $P = abB \in E$. Both users can compute this key and it can be used for en- and decryption.

For example Alice knows bB, which is public and her secret key a. However a third party only knows aB and bB. Without solving the discrete logarithm problem it seems to be impossible to compute abB.
B plays the role of a generator g in the finite field version of the Diffie-Hellman system. We do not require, that B is a generator of the group of points on E. In fact this group does not need to be cyclic. Even if this group is cyclic we do not want to check if B is a generator of it, but the subgroup generated by B should be large. We prefer subgroups of the same order as E. We assume that B is a fixed public point on E of huge order (either N or a large divisor of N).
Suppose we want to use the Diffie-Hellman key exchange for message transmitting, then we get the following cryptosystem.

5.2. Massey-Omura cryptosystem

5.2.1. Massey-Omura on finite fields

This method was developed by James Massey and Jim Omura in 1983. With the help of this protocol users have the possibility to exchange messages secretely over an insecure channel. In this protocol neither a public key nor a shared secret key is necessary.
Suppose Alice wants to send the message P_m to Bob. Then the following procedure results.

Protocol:

1. First both parties agree on a public finite field $\mathbb{F}_q$.

2. Alice chooses secretly a random integer $e_A \in \{1, \ldots, q-1\}$ such that

$$gcd\,(e_A, q-1) = 1$$

   and she computes the inverse d_A of e_A with the help of the Euclidean algorithm $d_A = e_A^{-1} \bmod q-1$. Hence $d_A e_A \equiv 1 \bmod q-1$.

3. Bob chooses secretly a random integer $e_B \in \{1, \ldots, q-1\}$ such that

$$gcd\,(e_B, q-1) = 1$$

   and he computes $d_B = e_B^{-1} \bmod q-1$. Therefore $d_B e_B \equiv 1 \bmod q-1$.

4. Alice sends the element ${P_m}^{e_A}$ to Bob.

5. Bob is not able to compute P_m since he does neither know e_A nor d_A. That is why, Bob computes ${P_m}^{e_A e_B}$ and sends it back to Alice.

6. Now Alice decrypts her encryption by computing ${P_m}^{e_A e_B d_A}$ and afterwards she sends ${P_m}^{e_B}$ back to Bob.

7. Now Bob is able to read the message P_m by computing ${P_m}^{e_B d_B}$.

Here, it is important that a good signature scheme is used. Otherwise a third party C, which should not know the message, could impersonate Bob and send the message ${P_m}^{e_A e_C}$ to Alice. Alice would not even notice, that there is a third party. Therefore she would raise the message to d_A and hence C would be able to decrypt the message. That is the reason why the message should contain any kind of authentification of Bob. For example a signature which could only belong to Bob.
Furthermore it is important that neither user B nor user C would be able to compute e_A after decrypting some messages and hence knowing some pairs $(P_m, {P_m}^{e_A})$. The security of this cryptosystem is again based on the discrete logarithm problem. Suppose Bob would be able to solve the discrete logarithm problem in $\mathbb{F}_q^\times$ then he would be able to compute e_A with P_m and ${P_m}^{e_A}$. Hence he would also be able to compute $d_A = e_A^{-1} \bmod q-1$ easily and he would be able to decrypt every message from Alice sent to him or anyone else.
Instead of computing the power, also other operations can be used in this system and it has an analogous version on elliptic curves.

5.2.2. Elliptic curve Massey-Omura

As in the finite field version we can use this public key cryptosystem to transmit messages m. We assume that this message is embedded as point P_m on an elliptic curve $E \in \mathbb{F}_q$. (Where E is public and q is large.) We further assume that the number of points N on the curve E has been computed and N is public. We consider the single steps of this protocol in the case that Alice wants to send the message P_m to Bob.

Protocol:

1. Alice secretly chooses a random integer e_A between 1 and N, such that $gcd\,(e_A, N) = 1$. Furthermore she computes the inverse of e_A with the help of the euclidean algorithm $d_A = e_A^{-1} \bmod N$. Hence $d_A e_A \equiv 1 \bmod N$.

2. Bob also secretly chooses an random integer e_B between 1 and N, such that $gcd\,(e_B, N) = 1$. Then he computes $d_B = e_B^{-1} \bmod N$ using the Euclidean algorithm. Thus $d_B e_B \equiv 1 \bmod N$.

3. Alice sends the point $e_A P_m$ to Bob.

4. Bob cannot reconstruct the message P_m because he does neither know d_A nor e_A. Therefore he multiplies the message by e_B and sends $e_B e_A P_m$ back to Alice.

5. Now Alice removes her encryption of the message by multiplying $e_B e_A P_m$ by d_A. Since $N P_m = 0$ and $d_A e_A \equiv 1 \bmod N$ the point $e_B P_m$ results and Alice sends this point to Bob.

6. Bob can read the message by multiplying the point $e_B P_m$ by d_B.

Unfortunately, a third party which knows $e_A Pm,\ e_B e_A P_m,\ e_B P_m$ and is able to solve the discrete logarithm problem on E could determine e_B with the help of the first two points. Furthermore $d_B = e_B^{-1} \bmod N$ and $P_m = d_B(e_B P_m)$ could be computed afterwards.
In the next section we consider the ElGamal cryptosystem.

5.3. ElGamal cryptosystem

5.3.1. ElGamal on finite fields

This is another public key cryptosystem for transmitting messages and is again based on the idea of the Diffie-Hellman key exchange. We assume that we want to send plaintext messages as a numeric value $P_m \in \mathbb{F}_q$.
If Alice wants to send the message P_m to Bob then she proceeds as follows. The public key for encryption is the element $g^{a_B} \in \mathbb{F}_q$.

Protocol:

1. At first Alice and Bob agree on a public finite field $\mathbb{F}_q$ and an element $g \in \mathbb{F}_q^\times$. (Preferably g should be a generator.)

2. Alice randomly chooses an integer a_A in $0 < a_A < q-1$ and keeps it secret. The public key is the element $g^{a_A} \in \mathbb{F}_q$.

3. Bob randomly chooses an integer a_B in $0 < a_B < q-1$ and keeps it secret too. His public key is the element $g^{a_B} \in \mathbb{F}_q$.

4. Then Alice randomly chooses an integer k and sends the pair $(g^k, P_m g^{a_B k})$ to Bob. Alice can compute $g^{a_B k}$ without knowing a_B by raising g^{a_B} to the power k.

5. Bob knows a_B. Now Bob can reconstruct P_m from this pair by raising g^k to the power a_B and then dividing the second element by the previous result.

So to speak Alice sends a message to Bob which consists of a masked P_m (mask $= g^{a_B k}$) and an advice g^k. This advice helps Bob to remove the mask from P_m, but this advice is only helpful if a_B is known.

If somebody can solve the discrete logarithm problem in $\mathbb{F}_q$ then he can also break this cryptosystem by computing the secret key a_B from the public key g^{a_B}. However, it is suspected that it is not possible to compute $g^{a_B k}$ with g^{a_B} and g^k without solving the discrete logarithm problem.
Now we consider the elliptic curve version of this cryptosystem.

5.3.2. Elliptic curve ElGamal

This public key cryptosystem is also used to transmit secret messages. Suppose Alice wants to send the message P_m to Bob, then this system works as follows.

Protocol:

1. At first Alice and Bob agree on a public finite field $\mathbb{F}_q$, an elliptic curve E over the finite field $\mathbb{F}_q$ and a point $B \in E$. The number of points N is not necessarily needed in this protocol.

2. Alice randomly chooses an integer a_A and keeps it secret. Further she computes the point $a_A B$ and publishes it.

3. Bob randomly chooses an integer a_B and keeps it secret. He computes the point $a_B B$ and also publishes it.

4. Then Alice randomly chooses an integer k and sends the pair $(kB, P_m + k(a_B B))$ to Bob. (a_B is the private key of Bob)

5. If Bob would like to read the message he has to multiply the first point of the pair by his private key a_B and he has to subtract the result from the second point of the pair
$$P_m + k(a_B B) - a_B(kB) = P_m.$$

Again, Alice sends a message to Bob which consists of a masked P_m (mask $= ka_B B$) and an advice kB. This advice helps to remove the mask as long as the secret integer a_B is known. A third party which can solve the discrete logarithm problem on E can compute a_B with the help of the published information B and $a_B B$.

5.4. Digital Signature Algorithm

Finally we present a digital signature scheme which is a variant of the ElGamal scheme. The Digital Signature Algorithm was proposed by the U.S. National Institute of Standards and Technology (NIST) in 1991. Usually we do not sign a message m itself but a shorter 'digest' i.e., the hash value of the message $h(m)$ is used. Therefore this scheme requires a hash function h (e.g. SHA-2).

5.4.1. Digital Signature Algorithm

Suppose Bob wants to sign a message $m \in \mathbb{F}_q$. He can use the same public and private key pair $g^{a_B} = y$ and a_B as in ElGamal cipher. This subsection is based on [24].

I. DSA Key generation Users of this algorithm should decide on two very large prime numbers p and q such that q divides $p-1$, and on a generator g of the unique cyclic group of order q in $\mathbb{F}_p^\times$. This generator can be found by selecting an element $\alpha \in \mathbb{F}_p^\times$ and computing $g = \alpha^{(p-1)/q} \bmod p$. If $g \neq 1$ a generator was found. Otherwise try another element $\alpha \in \mathbb{F}_p^\times$. For generating public and private keys, each user should do the following:

1. Select a random integer $a \in \{1, \ldots, q-1\}$.
2. Compute $y = g^a \bmod p$.
3. Then (p, q, g, y) is the public key and a is the corresponding private key.

II. DSA Signature generation If Bob wants to send a signed message m to Alice he does the following:

1. Bob selects a random secret integer $k \in \{1, \ldots, q-1\}$.
2. He computes $r = (g^k \bmod p) \bmod q$. If $r = 0$ then goto 1.
3. He computes $k^{-1} \bmod q$.
4. Furthermore Bob computes $s = k^{-1}(h(m) + a_B r) \bmod q$. If $s = 0$ then goto 1.
5. Then Bob's signature for the message m is the pair (r, s).

III. DSA Signature verification To verify Bob's signature, Alice proceeds as follows:

1. Alice obtains Bob's public key (p, q, g, y).
2. She verifies that r and s are integers in $\{1, \ldots, q-1\}$. Otherwise she would reject the signature.
3. She computes $w = s^{-1} \bmod q$ and $h(m)$.
4. Further Alice computes $u_1 = h(m)w \bmod q$ and $u_2 = rw \bmod q$.
5. Then she computes $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$ and verifies the signature by checking $v = g^{u_1} y^{u_2} = g^{u_1}(g^{a_B})^{u_2} = g^{h(m)w} g^{a_B rw} = g^{w(h(m)+a_B r)} = g^{wsk} = g^{s^{-1}sk} = g^k = r$.
6. She accepts the signature only if $v = r$.

5.4.2. Elliptic Curve Digital Signature Algorithm

The elliptic curve version ECDSA has been adopted as an official ANSI standard in 1999.

I. ECDSA Key generation Let E be an elliptic curve over $\mathbb{F}_p$ (where p is prime) and let P be a point of prime order n in $E(\mathbb{F}_p)$. Then each user of this scheme has to do the following:

1. Select a random $a \in \{1, \ldots, n-1\}$.
2. Compute $Q = aP$.
3. Then Q is the public key and a is the private key.

II. ECDSA Signature generation If Bob wants to send a signed message m to Alice he does the following:

1. Bob selects a random $k \in \{1, \ldots, n-1\}$.
2. He computes $kP = (x_1, y_1)$ and $r = x_1 \bmod n$. If $r = 0$ then goto 1.
3. He computes $k^{-1} \bmod n$.
4. Furthermore Bob computes $s = k^{-1}(h(m) + a_A r) \bmod n$. If $s = 0$ then goto 1.
5. Then the signature of Bob is (r, s).

III. ECDSA Signature verification To verify Bob's signature, Alice proceeds as follows:

1. Alice obtains Bob's public key Q_B.
2. She verifies that r and s are integers in $\{1, \ldots, n-1\}$, otherwise she rejects the signature.
3. She computes $w = s^{-1} \bmod n$ and $h(m)$.
4. Alice further computes $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.
5. Then she computes $u_1 P + u_2 Q_B = (x_0, y_0)$ and $r = x_0 \bmod n$.
6. She accepts the signature only if $v = r$.

In this chapter we saw some applications of elliptic curves in cryptography. Notice, that elliptic curve cryptography has found many applications in the security sector today. Finally in the appendix you can find the source code of our implemented algorithm.

A. Source code

The ideas of Section 3.3 have been implemented in the following form. We used therefore the computer algebra system SageMath.

A.1. Precomputations

```
def divide_input(l_bound,u_bound,ncpus):
  part = ceil((u_bound-l_bound+1)/ncpus);
  input_new = []
  for i in xrange(ncpus-1):
    input_new.append((l_bound+i*part,(l_bound+(i+1)*part)-1));
  input_new.append((l_bound+(ncpus-1)*part,u_bound));
  return input_new;


@parallel
def precompute_squarefree_parts_from_to(l_bound, u_bound):
  parts = []
  for i in xrange (l_bound,u_bound+1):
    parts.append(squarefree_part(i));
  return parts;


def precompute_squarefree_parts(l_bound,u_bound,ncpus):
  input_sqfp = divide_input(l_bound,u_bound,ncpus);
  big_list = sorted(list(precompute_squarefree_parts_from_to(input_sqfp)));
  sqfree_parts = [];
  for i in xrange(ncpus):
    sqfree_parts = sqfree_parts + big_list[i][1];
  return sqfree_parts;


def precompute_ap():
  a_p = [];
  E = EllipticCurve([0,0,0,-1,0]);
  for p in PN:
```

```
    a_p.append(E.ap(p));
  return a_p;


def precompute(bound_primes,ncpus):
  w = walltime();
  global PN;
  PN = list(primes(1,bound_primes-1));
  global a_p;
  a_p = precompute_ap_up_to(bound_primes);
  global sqfree_parts;
  sqfree_parts = precompute_squarefree_parts(1,10^6,ncpus)
  print_time(...);
```

A.2. Computations

To reduce the number of code lines in the following we removed and simplified many print statements.

```
import socket
import gc

@parallel
def get_congruent_numbers(u_low,u_upp,v_low,v_upp):
  uv_map_temp = {}
  T = [];

  for u in xrange(u_low,u_upp+1):
    v_bound = v_low;
    if (u > v_low):
      v_bound = u+1;
    for v in xrange(v_bound,v_upp+1):
      if ((u < v) and (mod(u+v,2)==1) and (gcd(u,v)==1)):
        n =
sqfree_parts[u-1]*sqfree_parts[v-1]*sqfree_parts[(v-u)-1]
*sqfree_parts[(v+u)-1];
        T.append(n);
        uv_map_temp[n] = (u,v);

  return [T,uv_map_temp];


```

```
24 def divide_input_get_sqfree_numbers(u_low,u_upp,v_low,v_upp,ncpus):
25   part = floor((u_upp-u_low+1)/ncpus);
26   if(u_upp > v_upp):
27     part = floor((v_upp-u_low+1)/ncpus);
28   input_new = []
29   if (part < 1):
30     input_new.append((u_low,u_upp,v_low,v_upp));
31   else:
32     rest = u_upp - u_low + 1 - (part * ncpus);
33     rest_curr = rest;
34     for i in xrange(ncpus-1):
35       if (rest_curr > 0):
36         input_new.append((u_low+i*part + i, (u_low+(i+1)*part)-1+(i+1),
37 v_low, v_upp));
38       elif (rest_curr <= 0 and i >= rest):
39         input_new.append((u_low+i*part + rest, (u_low+(i+1)*part)-1+rest,
40 v_low,
41 v_upp));
42       else:
43         input_new.append((u_low+i*part, (u_low+(i+1)*part)-1, v_low, v_upp));
44       rest_curr = rest_curr - 1;
45         if(u_upp > v_upp):
46           input_new.append((u_low+(ncpus-1)*(part+1), v_upp, v_low, v_upp));
47         else:
48           if (rest > 0):
49             input_new.append((u_low+(ncpus-1)*part+rest, u_upp, v_low, v_upp));
50           else:
51             input_new.append((u_low+(ncpus-1)*part, u_upp, v_low, v_upp));
52   return input_new;
53
54
55 def choose_set_T_of_congruent_numbers(u_low,u_upp,v_low,v_upp,ncpus):
56   divided_input = divide_input_get_sqfree_numbers(u_low, u_upp, v_low, v_upp,
57 ncpus);
58
59   T_list= sorted(list(get_congruent_numbers(divided_input)));
60   T = []
61
62   for i in xrange(len(T_list)):
63     if(len(T_list[i][1][0])==1):
64       print("  computed list T part has length 1");
65       if(T_list[i][1][0] == 'N'):
66         print("  computed list T equals 'N'");
```

```
67          print("  list[i][1][1]: "+str(T_list[i][1][1]));
68        else:
69          T = T + T_list[i][1][0];
70          uv_map.update(T_list[i][1][1]);
71      else:
72        T = T + T_list[i][1][0];
73        uv_map.update(T_list[i][1][1]);
74
75    T = Set(T).list();
76    return T;
77
78
79  def divide_set_T(T,ncpus,s):
80    list_of_numbers = list(T);
81    length_list = len(list_of_numbers);
82    input_new = []
83    if (length_list < ncpus):
84      input_new.append((list_of_numbers[0],list_of_numbers[length_list-1],T,s));
85        return input_new;
86    else:
87      part = ceil(length_list/ncpus);
88      for i in xrange(ncpus-1):
89        input_new.append((list_of_numbers[i*part],
90  list_of_numbers[((i+1)*part)-1], T, s));
91
92      input_new.append((list_of_numbers[(ncpus-1)*part],
93  list_of_numbers[length_list-1], T, s));
94
95    return input_new;
96
97
98  @parallel
99  def get_set_Ts(from_number,to_number,T,s):
100   Ts = [];
101
102   #compute s(n) by Monsky's formula
103   for counter in xrange(T.index(from_number),T.index(to_number)+1):
104     primes = T[counter].factor();
105     t = len(primes);
106     t_start = 0;
107     t_end =t;
108     if (primes[0][0] == 2):
109       t_start=1;
```

```
110       t = t-1;
111
112     #create matrices
113     D_1 = matrix(t,t);
114     D_2 = matrix(t,t);
115     D2 = matrix(t,t);
116     A = matrix(t,t);
117
118     for i_counter in xrange(t_start,t_end):
119       # modify indices
120       if (t_start <> 0):
121         i = i_counter-1;
122       if (t_start == 0):
123         i = i_counter;
124
125       # l = -1
126       if (legendre_symbol(-1,primes[i_counter][0]) == 1):
127         D_1[i,i] = 0;
128       if (legendre_symbol(-1,primes[i_counter][0]) == -1):
129         D_1[i,i] = 1;
130
131       # l = -2
132       if (legendre_symbol(-2,primes[i_counter][0]) == 1):
133         D_2[i,i] = 0;
134       if (legendre_symbol(-2,primes[i_counter][0]) == -1):
135         D_2[i,i] = 1;
136
137       # l = 2
138       if (legendre_symbol(2,primes[i_counter][0]) == 1):
139         D2[i,i] = 0;
140       if (legendre_symbol(2,primes[i_counter][0]) == -1):
141         D2[i,i] = 1;
142
143       # create A
144       for j_counter in xrange(t_start,t_end):
145         # modify indices
146         if (t_start <> 0):
147           j = j_counter-1;
148
149         if (t_start == 0):
150           j = j_counter;
151
152         if (i <> j):
```

```
153             if (legendre_symbol(primes[j_counter][0],primes[i_counter][0]) == 1):
154               A[i,j] = 0;
155             if (legendre_symbol(primes[j_counter][0],primes[i_counter][0]) == -1):
156               A[i,j] = 1;
157
158       for i in xrange(t):
159         for k in xrange(t):
160           if (i <> k):
161             A[i,i] = mod(A[i,i] + A[i,k],2);
162
163       #compute matrices Mo and Me
164       s_n = 0;
165
166       if t_start == 0:
167         Mo = block_matrix(GF(2),2, 2, [ A+D2, D2, D2, A+D_2 ])
168         s_n = 2*t -Mo.rank();
169       else:
170         Me = block_matrix(GF(2),2, 2, [ D2, A+D2, transpose(A)+D2, D_1 ])
171         s_n = 2*t-Me.rank();
172
173       if (s_n >= s):
174         Ts.append(T[counter]);
175     return Ts;
176
177
178   # Step 1
179   def step1(u_low,u_upp,v_low,v_upp,ncpus):
180     print_info(...);
181
182     s = 6;
183     T = choose_set_T_of_congruent_numbers(u_low,u_upp,v_low,v_upp,ncpus);
184
185     print("len T: "+str(len(T)));
186     print_time(...);
187
188     Ts_output = sorted(list(get_set_Ts(divide_set_T(T,ncpus,s))))
189     Ts = []
190
191     for i in xrange(len(Ts_output)):
192       if(len(Ts_output[i][1])==1):
193         print("  computed list Ts_output part has length 1");
194         if(Ts_output[i][1] == 'N'):
195           print("  computed list Ts_output equals 'N'");
```

```
196         else:
197           Ts = Ts + Ts_output[i][1];
198       else:
199         Ts = Ts + Ts_output[i][1];
200 
201     print("len Ts: "+str(len(Ts)));
202     print_time(...);
203 
204     print_time(...);
205     return Ts;
206 
207 
208 # Mestre-Nagao's sum
209 def get_S(n,bound_primes):
210     sum = 0;
211     for i in xrange(bound_primes):
212       p = PN[i]
213       disc = (-16)*(4*(-(n^2))^3);
214 
215       if (mod(disc,p) <> 0):
216         ap = a_p[PN.index(p)]*kronecker(n,p);
217         sum = sum + RR(((-ap+2)/(p+1-ap))*log(p))
218     return sum;
219 
220 
221 def divide_numbers(set_of_numbers, bound_primes, bound, ncpus):
222     list_of_numbers = list(set_of_numbers);
223     length_list = len(list_of_numbers);
224     input_new = []
225     if (length_list < ncpus):
226       input_new.append((list_of_numbers[0],
227 list_of_numbers[length_list-1], bound_primes , bound, list_of_numbers));
228       return input_new;
229     else:
230       part = floor(length_list/ncpus);
231       for i in xrange(ncpus-1):
232         input_new.append((list_of_numbers[i*part],list_of_numbers[((i+1)*part)-1],
233 bound_primes,bound,list_of_numbers));
234 
235       input_new.append((list_of_numbers[(ncpus-1)*part],
236 list_of_numbers[length_list-1] , bound_primes, bound, list_of_numbers));
237     return input_new;
238 
```

```
239
240 @parallel
241 def parallel_step2(nr\_from, nr\_to, bound\_primes, bound, list\_of\_nrs):
242   list_tmp = [];
243   mns_tmp = {};
244   uv_map_tmp = {};
245   for i in xrange(list\_of\_nrs.index(nr\_from), list\_of\_nrs.index(nr\_to)+1):
246     n = list\_of\_nrs[i];
247     S = get_S(n, bound_primes);
248     if  S >= bound:
249       list_tmp.append(n);
250       mns_tmp[n] = numerical_approx(S);
251       uv_map_tmp[n] = (uv_map[n][0],uv_map[n][1]);
252
253   return [list_tmp, mns_tmp, uv_map_tmp];
254
255
256 def compute_prime_bounds(Ms):
257   bounds = [];
258   for i in xrange(len(Ms)):
259     bounds.append(len(list(primes(1,Ms[i][0]-1))));
260   return bounds;
261
262
263 #Step 2
264 def step2(Ts,ncpus):
265   print_info(...);
266
267   Ms =
268 [[500,10],[1000,12],[5000,15],[10000,20],[15000,25],[20000,30],[30000,45]];
269
270   prime_bounds = compute_prime_bounds(Ms);
271   k = len(Ms);
272   Ts_array = [];
273   Ts_array.append(Ts);
274
275   for i in xrange(1,k+1):
276     list_tmp = [];
277     if len(Ts_array[i-1]) > 0:
278       list_tmp =
279 get_sieved_numbers(sorted(list(parallel_step2(divide_numbers(Ts_array[i-1],
280 prime_bounds[i-1], Ms[i-1][1], ncpus)))));
281     Ts_array.append(list_tmp);
```

```
282
283   print_sieving_info(...);
284   print_time(...);
285
286   return Ts_array;
287
288
289 def divide_candidates_for_computing_rank(Ts_array, ncpus):
290   length_list = len(Ts_array);
291   part = floor((length_list)/ncpus);
292   input = []
293   if (part < 1):
294     for i in xrange(length_list):
295       input.append(Ts_array[i:i+1]);
296     return input;
297   else:
298     rest = length_list - (part * ncpus);
299     rest_curr = rest;
300     for i in xrange(ncpus-1):
301       if (rest_curr > 0):
302         input.append(Ts_array[i*part+i:((i+1)*part)+(i+1)]);
303       elif (rest_curr <= 0 and i >= rest):
304         input.append(Ts_array[i*part+rest:((i+1)*part)+rest]);
305       else:
306         input.append(Ts_array[i*part:((i+1)*part)]);
307       rest_curr = rest_curr - 1;
308     if (rest > 0):
309       input.append(Ts_array[(ncpus-1)*part+rest:length_list]);
310     else:
311       input.append(Ts_array[(ncpus-1)*part:length_list]);
312
313     return input;
314
315
316 @fork(timeout=1800)
317 def compute_rank_for_one_curve(n):
318   try:
319     En = mwrank_EllipticCurve([0, 0, 0, -(n^2), 0])
320     rank = En.rank();
321     return rank;
322   except Exception as detail:
323     print_exception(...);
324
```

```
325
326
327 @parallel
328 def compute_rank(Ts_array):
329   cntr_cand = 0;
330   cntr_no_rank_crvs = 0;
331   curve_found = false;
332   numbers_found = [];
333   no_rank_crvs = [];
334   En = 0;
335
336   for n in Ts_array:
337     rank = 0;
338     rank = compute_rank_for_one_curve(n);
339
340     if (rank == 'NO DATA' or rank == None):
341       cntr_no_rank_crvs = cntr_no_rank_crvs + 1;
342       no_rank_crvs.append(get_no_rank_curve(n));
343     elif(rank == 'NO DATA (timed out)'):
344       print_exception(...);
345       cntr_no_rank_crvs = cntr_no_rank_crvs + 1;
346       no_rank_crvs.append(get_no_rank_curve(n));
347     elif(rank >= 6):
348       numbers_found.append([n,rank]);
349       curve_found = true;
350
351     cntr_cand = cntr_cand + 1;
352
353   print_time(...);
354   return [cntr_cand, curve_found, numbers_found, cntr_no_rank_crvs,
355 no_rank_crvs];
356
357
358 # Step 3
359 def step3(Ts_array, ncpus):
360   print_info(...);
361
362   numbers_found = [];
363   no_rank_crvs = [];
364   cntr_no_rank_crvs = 0;
365   k = len(Ts_array);
366   j = 0;
367
```

```
368   #find j
369   for i in xrange (1,k):
370     if (len(Ts_array[i]) <> 0):
371       j = i;
372   #check sets
373   setsOK = true;
374   if j == 0:
375     setsOK = false;
376
377   curve_found = false;
378
379   #compute MWRANK
380   cntr_cand = 0;
381   if (setsOK):
382     input = divide_candidates_for_computing_rank(Ts_array[j],ncpus);
383     output = sorted(list(compute_rank(input)));
384
385     [cntr_cand, curve_found, numbers_found, cntr_no_rank_crvs,
386 no_rank_crvs] = get_found_numbers(output);
387
388   print_summary(...);
389
390   return [numbers_found, cntr_cand, cntr_no_rank_crvs, no_rank_crvs];
391
392
393 def find_numbers(u_low,u_upp,v_low,v_upp,ncpus):
394   no_set_Ts_found = true;
395
396   Ts = [];
397   Ts_array = [];
398
399   Ts = step1(u_low, u_upp, v_low, v_upp, ncpus);
400
401   if (len(Ts) == 0):
402     print("No set Ts found.");
403     return [[],0,0,[]]
404   else:
405     no_set_Ts_found = false;
406     Ts_array = step2(Ts,ncpus);
407     gc.collect();
408
409     return step3(Ts_array,ncpus);
410
```

```
411
412 def get_split_input(u_low,u_upp,v_low,v_upp):
413   if (u_upp == 10^5):
414     new_range = 250;
415     number_of_parts = ceil((v_upp-v_low) / new_range);
416   else:
417     bound_of_numbers = 7000000;
418     range_of_numbers = (u_upp-u_low)*(v_upp-v_low);
419     number_of_parts = ceil(range_of_numbers / bound_of_numbers);
420     new_range = ceil((v_upp-v_low) / number_of_parts);
421
422   input = []
423   v_bound = 0;
424
425   for i in xrange(number_of_parts-1):
426     v_bound = v_low + i * new_range;
427     input.append((u_low,u_upp,v_bound,v_bound+new_range));
428   input.append((u_low,u_upp,v_low + (number_of_parts-1)*new_range, v_upp));
429
430   return input;
431
432
433 def main(u_low,u_upp,v_low,v_upp,ncpus,split):
434   if 'sqfree_parts' in globals():
435     openFile(u_low,u_upp,v_low,v_upp);
436     found = 0;
437     total_numbers_found = 0;
438     ctr_cand = 0;
439     bound_split_up = 20 * 10^6;
440
441     global uv_map;
442     uv_map = {}
443
444     global mestre_nagao_sums;
445     mestre_nagao_sums = {}
446
447     numbers_found = []
448
449     if (u_upp < u_low or v_upp < v_low):
450       print("Bounds are wrong. Please check given bounds.");
451     else:
452       bound = (u_upp-u_low)*(v_upp-v_low);
453       split_cond_small = (bound >= bound_split_up and u_upp == 10^4);
```

```
454         split_cond_large = (u_upp == 10^5);
455
456         if ((split_cond_small or split_cond_large) and split):
457           split_input  = get_split_input(u_low, u_upp, v_low, v_upp);
458           numbers_found_all = []
459           numbers_found_part = []
460           numbers_no_rank = []
461           ctr_no_rank = 0;
462
463           for inp in split_input:
464             print_bounds(...);
465
466             output = find_numbers(inp[0],inp[1],inp[2],inp[3],ncpus);
467             [numbers_fnd, new_ctr_cand, ctr_no_rnk_crvs, no_rnk_crvs] = output;
468
469             numbers_no_rank = numbers_no_rank + no_rnk_crvs;
470             ctr_no_rank = ctr_no_rank + ctr_no_rnk_crvs;
471
472             if (numbers_fnd <> None):
473               found = len(numbers_fnd);
474               numbers_found_part = createFoundNumbers(...);
475               numbers_found_all = numbers_found_all + numbers_found_part;
476
477               total_numbers_found = total_numbers_found + found;
478               ctr_cand = ctr_cand + new_ctr_cand;
479
480             if (new_ctr_cand > 0):
481               print_no_rank_computed_curves(...);
482               print_found_numbers(...);
483
484             print_summary_part(...);
485
486           print_summary(...);
487
488         else:
489           output = find_numbers(u_low, u_upp, v_low, v_upp, ncpus);
490           [numbers_fnd, ctr_cand, ctr_no_rnk_crvs, no_rnk_crvs] = output;
491
492           numbers_found_list = []
493
494           if (numbers_fnd <> None):
495             found = len(numbers_fnd);
496             numbers_found_list = createFoundNumbers(numbers_fnd, found);
```

```
497
498         if (ctr_cand > 0):
499           print_no_rank_computed_curves(...);
500
501           print_summary(...);
502   else:
503     print("Please perform 'precompute(bound_primes, number_of_cpus)'.")
```

Now we present a very easy example for an easier understanding of this quite long source code for finding CN-elliptic curves of high rank.

Example A.1. *A typical usage of the above source code is.*

```
sage: load('\local\home\klopf\scripts\precomputations.sage');
sage: precompute(30000,4);
sage: load('\local\home\klopf\scripts\findCongruentNumbers.sage');
sage: main(21,87,27450,32780,4,1)
```

Here you can see the corresponding output.

```
sage: main(21,87,27450,32780,4,1)
2015-10-07 13:19:51: Bounds:
            u: 21 - 87
            v: 27450 - 32780

2015-10-07 13:19:51: step1
--------------------------------------------------------------------------------
2015-10-07 13:19:53: len T: 144306
2015-10-07 13:21:31: len Ts: 976
2015-10-07 13:21:31: step 1: 99.58 sec,  1.66 min,  0.03 h

2015-10-07 13:21:31: step2
--------------------------------------------------------------------------------
Ms:[[500,10],[1000,12],[5000,15],[10000,20],[15000,25],[20000,30],[30000,45]]
2015-10-07 13:21:31: sieving:
2015-10-07 13:21:34:      i: 1 len(Ts_array[i]):        297
2015-10-07 13:21:37:      i: 2 len(Ts_array[i]):        192
2015-10-07 13:21:42:      i: 3 len(Ts_array[i]):        138
2015-10-07 13:21:51:      i: 4 len(Ts_array[i]):         71
2015-10-07 13:21:58:      i: 5 len(Ts_array[i]):         32
2015-10-07 13:22:03:      i: 6 len(Ts_array[i]):         10
2015-10-07 13:22:08:      i: 7 len(Ts_array[i]):          0
2015-10-07 13:22:08: step 2: 36.97 sec,  0.62 min,  0.01 h
```

```
2015-10-07 13:22:08: step3
--------------------------------------------------------------------------------
2015-10-07 13:22:38: computing rank: 30.26 sec,  0.50 min,  0.01 h
2015-10-07 13:22:50: computing rank: 42.68 sec,  0.71 min,  0.01 h
2015-10-07 13:22:53: computing rank: 44.86 sec,  0.75 min,  0.01 h
2015-10-07 13:23:00: computing rank: 52.43 sec,  0.87 min,  0.01 h
2015-10-07 13:23:00: step 3: 52.45 sec,  0.87 min,  0.01 h

2015-10-07 13:23:00: good candidates: 10
2015-10-07 13:23:00: numbers no rank computed: 0

2015-10-07 13:23:00: FOUND: 2
  1. n = 121 110 989 796 834, rank = 6, u = 86, v = 32 775, MS = 41.90
  2. n = 455 089 600 428 474, rank = 6, u = 22, v = 27 451, MS = 37.48

2015-10-07 13:23:00: total time: 189.12 sec,  3.15 min,  0.05 h
```

Bibliography

[1] J. Aguirre, A. Dujella, M. Jukic Bokun, and J. C. Peral. *Periodica Mathematica Hungarica*.

[2] R. Alter, T. B. Curtz, and K. K. Kubota. Remarks and results on congruent numbers. In *Proceedings of the Third Southeastern Conference on Combinatorics, Graph Theory and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1972)*, pages 27–35. Florida Atlantic Univ., Boca Raton, Fla., 1972.

[3] Asymptote. Example elliptic.asy. `http://asymptote.sourceforge.net/gallery/`.

[4] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 2000. Reprint of the 1999 original.

[5] J. Brown. Congruent numbers and elliptic curves. `http://www.math.caltech.edu/~jimlb/congruentnumberslong.pdf`, 2007.

[6] J. Carlson, A. Jaffe, and A. Wiles, editors. *The Millennium Prize Problems*. Clay Mathematics Institute, Cambridge, MA; American Mathematical Society, Providence, RI, 2006.

[7] V. Chandrasekar. The congruent number problem. *Resonance - Journal of Science Education*, 3(8):33–45, 1998.

[8] J. H. Coates. Congruent number problem. *Q. J. Pure Appl. Math.*, 1(1):14–27, 2005.

[9] H. Cohen. *Number theory. Volume I. , Tools and diophantine equations*. Graduate Texts in Mathematics. Springer, New York, 2007.

[10] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, editors. *Handbook of elliptic and hyperelliptic curve cryptography*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2006.

[11] J. E. Cremona. *Algorithms for Modular Elliptic Curves*. Cambridge University Press, second edition edition, 1997.

[12] L. E. Dickson. *History of the theory of numbers. Vol. II: Diophantine analysis*. Chelsea Publishing Co., New York, 1966.

[13] A. Dujella. High rank elliptic curves and related diophantine problems. Zahlentheoretisches Kolloquium, Graz.

[14] A. Dujella. Lecture notes on ak zahlentheorie. 2001.

[15] A. Dujella, A. S. Janfada, and S. Salami. A search for high rank congruent number elliptic curves. *J. Integer Seq.*, 12(5):11, 2009.

[16] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Professional Computing. Springer-Verlag, New York, 2004.

[17] W. B. Hart, G. Tornaría, and M. Watkins. Congruent number theta coefficients to 10^{12}. In *Algorithmic Number Theory*, volume 6197 of *Lecture Notes in Computer Science*, pages 186–200. Springer Berlin Heidelberg, 2010.

[18] D. Husemöller. *Elliptic curves*, volume 111 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 2004. With appendices by Otto Forster, Ruth Lawrence and Stefan Theisen.

[19] A. W. Knapp. *Elliptic curves*, volume 40 of *Mathematical Notes*. Princeton University Press, Princeton, NJ, 1992.

[20] N. Koblitz. *Introduction to elliptic curves and modular forms*, volume 97 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1993.

[21] N. Koblitz. *A course in number theory and cryptography*, volume 114 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1994.

[22] M. Madritsch. Mathematische grundlagen der kryptographie. Lecture notes.

[23] B. Mazur. Rational isogenies of prime degree (with an appendix by D. Goldfeld). *Invent. Math.*, 44(2):129–162, 1978.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, 1997. With a foreword by Ronald L. Rivest.

[25] F. Najman. Some rank records for elliptic curves with prescribed torsion over quadratic fields. *An. Ştiinţ. Univ. "Ovidius" Constanţa Ser. Mat.*, 22(1):215–219, 2014.

[26] K. Ono. Twists of elliptic curves. *Compositio Math.*, 106(3):349–360, 1997.

[27] E. Oswald. Introduction to elliptic curve cryptography. 2005.

[28] Python. The python profilers. `https://docs.python.org/2/library/profile.html`.

[29] S. Roberts. Note on a problem of fibonacci's. *Proc. London Math. Soc.*, (11):35–44, 1879.

[30] SageMath. Parallel computing. `http://doc.sagemath.org/html/en/reference/parallel/index.html`.

[31] P. Serf. Congruent numbers and elliptic curves. In A. Pethö, M. E. Pohst, H. C. Williams, and H. G. Zimmer, editors, *Proceedings of the Colloquium on Computational Number Theory held at Kossuth Lajos University, Debrecen (Hungary), September 4-9, 1989*, pages 227–238. Walter de Gruyter, 1991.

[32] J. H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1992. Corrected reprint of the 1986 original.

[33] J. H. Silverman and J. Tate. *Rational points on elliptic curves*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1992.

[34] N. M. Stephens. Congruence properties of congruent numbers. *Bull. London Math. Soc.*, 7:182–184, 1975.

[35] L. C. Washington. *Elliptic curves*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2003. Number theory and cryptography.

[36] M. Watkins, S. Donnelly, N. D. Elkies, T. Fisher, A. Granville, and N. F. Rogers. Ranks of quadratic twists of elliptic curve. *Publications mathématiques de Besançon*, pages 63–98, 2014.

[37] A. Werner. *Elliptische Kurven in der Kryptographie*. Springer-Verlag Berlin Heidelberg, 2002.

[38] T. A. Whitelaw. *Introduction to abstract algebra*. Blackie Academic & Professional, London, third edition, 1995.