Collision Detection

Using Continued Fractions

Botsingsdetectie met behulp van kettingbreuken

by

Aron Schouten

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 7, 2022 at 3:30 PM.

Student number: 4999169
Project duration: February 28, 2022 – July 7, 2022
Thesis committee: Dr. P. M. Visser (supervisor), TU Delft, Mathematical Physics
Prof. dr. D. C. Gijswijt, TU Delft, Discrete Mathematics & Optimization

TU Delft

Preface

You are about to read the thesis 'Collision Detection Using Continued Fractions'. This thesis was written as part of my graduation from my bachelor Applied Mathematics at the Delft University of Technology. From February to July 2022, I was engaged in conducting research and writing the thesis. The thesis is written for peers; fellow third-year students of Applied Mathematics. It may also be of interest to those who are good at mathematics and want to learn more about fast collision detection algorithms.

The project was carried out under the supervision of Paul Visser, who wrote an article about collision detection using continued fractions. The research consisted of examining his article. The article had to be supplemented with other sources to provide a good basic knowledge and background information. By doing research into these basics and background information, I was able to answer my research questions. When I could not figure something out on my own, Paul Visser, my supervisor, was available and willing to answer my questions. He was always able to help me with extensive notes about his article, continued fractions and lattice basis reduction, which where very useful. I am very grateful to him for this. Thank you Paul Visser.

Finally, my parents deserve a special thank you: your good advice and kind words have served me well, as always.

Enjoy reading.

Aron Schouten
Wateringen, June 2022

Abstract

Context. In astronomy, collision detection is the computational problem of finding when planets, asteroids or satellites collide. Computer simulations make it possible to show very precisely how bodies move through space, which makes it possible to detect collisions. However, these simulations require a lot of computing power if there are many bodies, which is of course undesirable.

Aims. The aim of this report is to provide more insight into a collision detection method that does not use the computationally expensive simulations, but rather fast mathematical techniques.

Methods. It can be shown that the collision detection problem between two planets is equivalent to finding the integer point (k, l) between two parallel lines, closest to the origin. The solution uses the continued fraction of the ratio of the orbital periods of the planets, as the slopes of the lines are this ratio. The convergents of the continued fraction form basis vectors with which the point (k, l) is searched for. This is called lattice basis reduction.

Results. Where brute force always takes k steps to find the integer point (k, l), the method with continued fractions and lattice basis reduction often finds the point in about $\sqrt{k}$ steps. The worst case however is still k steps, the same as brute force. This only happens if there are no basis transformations, so luckily k is then often small.

Conclusions. When there are many bodies between which collisions need to be detected, lattice basis reduction provides a fast method. For the basis vectors, the convergents of the continued fraction of the ratio of the orbital periods of the planets must be used. If the solution cannot be reached exactly, (k, l) can also be estimated.

List of Symbols

Symbol explanation:

s_i	radius of planet i
$\mathbf{v}_i$	velocity of planet i
$\mathbf{r}_i$	point on the orbit of planet i which is closest to the orbit of another planet
t_i	first time that planet i passes $\mathbf{r}_i$
T_i	orbital period of planet i
k, l	rounds to collision
a_n	n-th digit of continued fraction
x_n	n-th convergent of x
k_{n+2}	denominator of the n-th convergent
l_{n+2}	numerator of the n-th convergent
$\mathbf{b}_n = \begin{pmatrix} k_n \\ l_n \end{pmatrix}$	n-th basis vector

A continued fraction:

$$x_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots\ \cfrac{1}{a_n}}}} = \frac{l_{n+2}}{k_{n+2}}$$

Contents

1 Introduction

In astronomy, collision detection is the computational problem of finding when planets, asteroids or satellites collide. It plays a major role when, for example, doing research into the origin of the universe or when simulating the rings of Saturn. Computer simulations make it possible to show very precisely how bodies of planets, asteroids or satellites move through space, which makes it possible to detect collisions. However, when there are many bodies, these simulations take a lot of computing power and thus time. For example, when considering an N-body problem, each planet can collide with any other planet. This means that the number of particle-particle interactions that needs to be computed is of the order of N^2 (Wikipedia contributors, 2022).

Since the orbits in which planets move are known (so-called Kepler orbits), it is possible to detect collisions without using the computationally expensive simulations. Consider for example two planets between which a collision is possible, i.e

$$|\mathbf{r}_2 - \mathbf{r}_1| < s_1 + s_2,$$

where $\mathbf{r}_1$ represents the point on the orbit of planet 1 which is closest to the orbit of planet 2, $\mathbf{r}_2$ the point on the orbit of planet 2 which is closest to the orbit of planet 1, and s_1 and s_2 the radii of planets 1 and 2 respectively (see Figure 1.1).

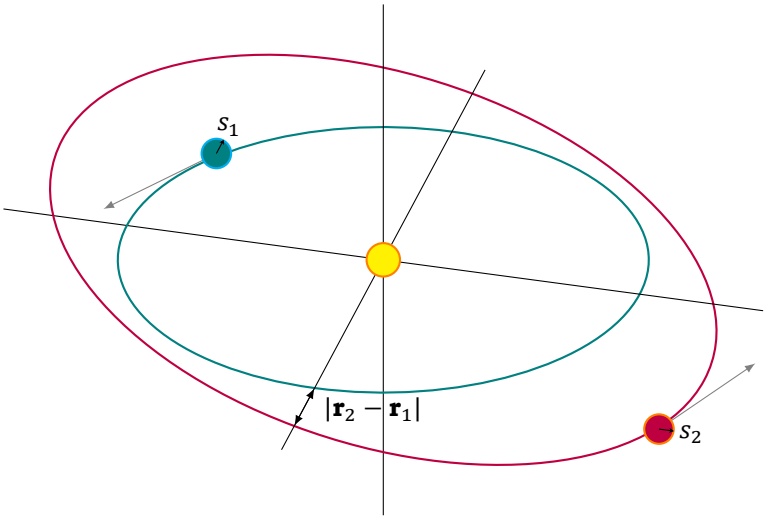

Figure 1.1: Two planets and their orbit. The planets have radii s_1 and s_2 and the (smallest) distance between the two orbits of the planets is $|\mathbf{r}_2 - \mathbf{r}_1|$. The planets can only collide if the sum of the radii of the planets is larger than the distance between the two orbits, that is $|\mathbf{r}_2 - \mathbf{r}_1| < s_1 + s_2$.

If the collision time between these planets is to be calculated, one has to answer the question: When are the planets 1 and 2 on $\mathbf{r}_1$ and $\mathbf{r}_2$ simultaneously? Define t_1 as the first time that planet 1 passes $\mathbf{r}_1$. Note that after 1, 2, 3, etc. full cycles around the orbit, the planet is back in the collision point and hence the time that planet 1 is in point $\mathbf{r}_1$ is:

$$t_1 + k \cdot T_1, \quad k \in \mathbb{N} \cup \{0\}.$$

Here T_1 is the orbital period of planet 1, the time it takes for the body to make one full cycle in its orbit. Similarly, let t_2 the first time that planet 2 passes $\mathbf{r}_2$ and T_2 the orbital period of planet 2. Then one has that the time that planet 2 is in point $\mathbf{r}_2$ is:

$$t_2 + l \cdot T_2, \quad l \in \mathbb{N} \cup \{0\}.$$

In order to collide, planets 1 and 2 must be on $\mathbf{r}_1$ and $\mathbf{r}_2$ at the same time, i.e:

$$t_1 + k \cdot T_1 = t_2 + l \cdot T_2. \tag{1.1}$$

Define $p = T_1/(t_2 - t_1)$ and $q = T_2/(t_2 - t_1)$. Then equation (1.1) can be rewritten:

$$\begin{aligned} t_1 + k \cdot T_1 &= t_2 + l \cdot T_2, \\ k \cdot T_1 - l \cdot T_2 &= t_2 - t_1, \\ k \cdot \frac{T_1}{t_2 - t_1} - l \cdot \frac{T_2}{t_2 - t_1} &= 1, \\ k \cdot p - l \cdot q &= 1. \end{aligned}$$

The planets have a certain radius s_i and speed $\mathbf{v}_i$, and hence it is possible that they collide just before or after they are in the collision point $\mathbf{r}_i$. A small delta is therefore introduced (Visser, 2022):

$$\delta = \frac{\sqrt{(s_1 + s_2)^2 - |\mathbf{r}_2 - \mathbf{r}_1|^2} \cdot |\mathbf{v}_2 - \mathbf{v}_1|}{|t_2 - t_1| \cdot |\mathbf{v}_1 \times \mathbf{v}_2|}.$$

The problem of finding the time that two planets collide can now be solved by just finding the non-negative integers k and l such that

$$1 - \delta < k \cdot p - l \cdot q < 1 + \delta \tag{1.2}$$

holds. This is of course an integer linear programming problem, which can be solved using an ILP solver. The problem is however such a specific case, that much faster algorithms can be devised. Note that if k represents a point on an x-axis and l a point on the y-axis, the inequalities of (1.2) yield two straight lines with the same slope,

$$y = x \cdot \frac{p}{q} - \frac{1 + \delta}{q} \quad \text{and} \quad y = x \cdot \frac{p}{q} - \frac{1 - \delta}{q}, \tag{1.3}$$

between which the integer point (k, l) must lie. If, without loss of generality, it is assumed that $T_1 > T_2$, one has that $p/q > 1$, and the plot in Figure 1.2 arises.

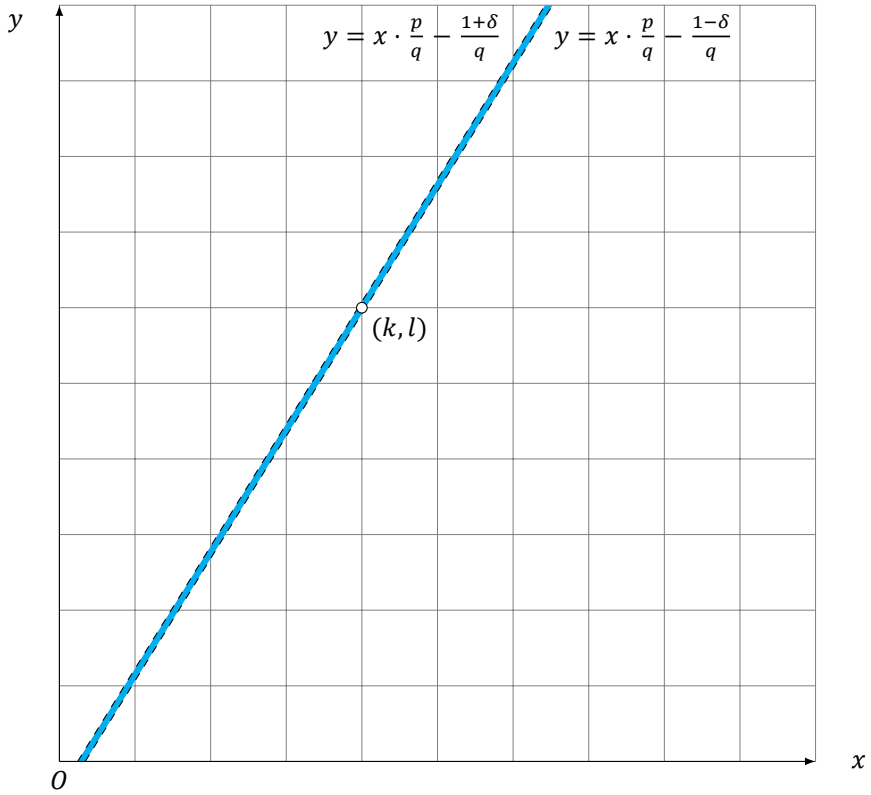

Figure 1.2: The problem of finding when two planets collide is the same as finding the integer point (k, l) closest to the origin that lies in a band. The band, which are the points that lie between equations (1.3), has been made blue and an integer point (k, l) which lies in it is drawn.

The complicated problem of the colliding planets has now been transformed into this 'simple' mathematical problem. However, the question that still remains is: How do we quickly find this (k, l)? Is there always a (k, l), or is it possible that two planets do not collide? In that case, can some useful things about the problem still be said? The aim of this report is to find answers to these questions. Continued fractions can be used to solve this problem. Therefore, the basics of continued fractions are explained in Chapter 2. It is explained why continued fractions are useful (Section 2.1), how one can find a continued fraction of a number (Section 2.2) and how the continued fractions can be represented using so called convergents (Section 2.3). Once the most important definitions and theorems on continued fractions are known, they are applied to the problem in Chapter 3. It is explained how the convergents of the continued fractions are used as basis vectors for a lattice basis reduction algorithm (Section 3.1). It is then shown that algorithm can be used to find the integer point (Section 3.2). Then, the algorithm is tested with Matlab and an improved algorithm is given (Section 3.3). After that, the run-times for different algorithms are given (Section 3.4). In Chapter 4 it is discussed when there is no solution; both analytically; there is no integer point in the band (Section 4.1) or numerically; the solution cannot be reached due to imprecise observational data or a lack of machine precision (Section 4.2). This last section covers the statistics of a possible solution. Finally, there is a discussion in Chapter 5, and a conclusion in Chapter 6.

2

Continued Fractions

In this chapter the basic definitions, algorithms and theorems of continued fractions are covered. Also the definition and some theorems of convergents, which are needed for the problem of the colliding planets, are introduced. But before introducing continued fractions, first the question “Why are continued fraction even useful?” is answered.

2.1. Why continued fractions are useful

The slope of the band in Figure 1.2 that arises due to equations (1.3) is

$$\frac{p}{q} = \frac{\left(\frac{T_1}{t_2 - t_1}\right)}{\left(\frac{T_2}{t_2 - t_1}\right)} = \frac{T_1}{T_2},$$

Since (k, l) must lie in the band, one has that

$$\frac{l}{k} \approx \frac{p}{q} = \frac{T_1}{T_2}$$

The k and l are integers, so l/k is rational, but T_1/T_2 however is irrational, as the orbital periods of planets are often irrational (Laskar, 2007). This is where continued fractions come into play: They are a perfect tool when approximating irrational numbers with rational numbers (Rockett and Szüsz, 1998).

2.2. The basics of continued fractions

Let’s start by introducing some definitions and theorems of continued fractions (Khinchin, 1949).

2.2.1. Definitions

Definition 2.1 (Continued fraction)**.** A **continued fraction** of $x \in \mathbb{R}$ is a fraction whose denominator is a quantity $a_n \in \mathbb{Z}$ plus a fraction with numerator $b_n \in \mathbb{Z}$, which latter fraction has a similar denominator, and so on:

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\ddots \cfrac{b_n}{a_n}}}}$$

Either $n \in \{0, 1, \dots, N\}$ for finite N, and the continued fraction is called **finite**, or $n \in \mathbb{N}$, and the continued fraction may be **infinite**.

The integers a_n are called the **coefficients** or **digits** of the continued fraction.

Definition 2.2 (Simple continued fraction)**.** An (in)finite continued fraction of $x \in \mathbb{R}$ is called **simple**, or said to be in **canonical form** if

$$b_n = 1 \text{ and } a_n > 0 \quad \forall n \geq 1,$$

i.e. if it is of the form

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \cfrac{1}{a_n}}}},$$

where $a_0 \in \mathbb{N} \cup \{0\}$ and $a_n \in \mathbb{N}$ if $n \geq 1$. Simple continued fractions can be denoted using the **canonical representation**:

$$[a_0; a_1, a_2, \ldots a_n]$$

From now on, only simple continued fractions will be discussed. To find the canonical representation of a continued fraction, the Euclidean algorithm can be used (Wikipedia contributors, 2022). This algorithm is therefore explained in the next subsection.

2.2.2. The Euclidean algorithm

If the canonical representation is to be found for a (non-negative) rational number p/q, where $p, q \in \mathbb{N}$, one could use the Euclidean algorithm (Algorithm 2.1). It is assumed that $p \geq q$.

Algorithm 2.1 (The Euclidean Algorithm (Gijswijt, 2019))**.** *Let $p, q \in \mathbb{N}$. Construct two sequences of integers q_n and a_n as follows. First let $q_0 = p$ and $q_1 = q$. Then let q_{n+2} be the remainder when dividing q_n by q_{n+1}, and a_n be the greatest integer less than or equal to the quotient $\frac{q_n}{q_{n+1}}$, i.e. $a_n = \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor$. That is,*

$$q_{n+2} = q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1}.$$

The algorithm stops when $q_n = 0$.

The integers a_n that occur in the canonical representation of p/q are exactly the successive quotients a_n computed by the Euclidean algorithm. For example, take the rational number $10/7$. Then $q_0 = 10, q_1 = 7$ and one finds:

$$q_2 = 10 - \left\lfloor \frac{10}{7} \right\rfloor \cdot 7 = 10 - 1 \cdot 7 = 3 \qquad (a_0 = 1)$$

$$q_3 = 7 - \left\lfloor \frac{7}{3} \right\rfloor \cdot 3 = 7 - 2 \cdot 3 = 1 \qquad (a_1 = 2)$$

$$q_4 = 3 - \left\lfloor \frac{3}{1} \right\rfloor \cdot 1 = 3 - 3 \cdot 1 = 0 \qquad (a_2 = 3)$$

And hence

$$\frac{10}{7} = 1 + \frac{3}{7} = 1 + \frac{1}{\left(\frac{7}{3}\right)} = 1 + \frac{1}{2 + \frac{1}{3}}.$$

The intermediate steps, which are 1 and $3/2$, approximate $10/7$ a bit better at each step. More about these so-called convergents in section 2.3.

Note that the input of Algorithm 2.1 is a rational number p/q, meaning that this algorithm can not be used to find the canonical representation of an irrational number. Luckily, the algorithm can be modified to work for all (non-negative) real numbers. Therefore, it is assumed that no longer $q_n \in \mathbb{N}$, but $q_n \in \mathbb{R}$ (see Algorithm 2.2).

Algorithm 2.2 (The Euclidean Algorithm for real numbers)**.** *Let $x \in \mathbb{R}, x > 0$. Construct two sequences $q_n \in \mathbb{R}$ and $a_n \in \mathbb{Z}$ as follows. First let $q_0 = x$ and $q_1 = 1$. Then let q_{n+2} be the remainder when dividing q_n by q_{n+1}, and a_n be the greatest integer less than or equal to the quotient $\frac{q_n}{q_{n+1}}$, i.e. $a_n = \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor$. That is,*

$$q_{n+2} = q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1}.$$

Note that here it is not stated that the algorithm stops when $q_n = 0$. This is because there are two cases: If $x \in \mathbb{Q}$, that is x is rational, then the simple continued fraction of x is finite, and the algorithm terminates. But if $x \in \mathbb{R} \setminus \mathbb{Q}$, that is x is irrational, then the simple continued fraction of x is infinite, and the algorithm never terminates. The sequence q_n however does converge to 0. These statements are proven in the next subsection.

2.2.3. Theorems

Theorem 2.1. *When $x = p/q$, where $p, q \in \mathbb{Z}$, the Euclidean algorithm (Algorithm 2.1) will terminate. In other words, one has that*

$$\exists N \in \mathbb{N} : q_N = 0.$$

Proof. Let q_n be the sequence as defined in the algorithm. Note that since $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \leq \frac{q_n}{q_{n+1}}$,

$$\begin{aligned} q_{n+2} &= q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1} \\ &\geq q_n - \frac{q_n}{q_{n+1}} \cdot q_{n+1} \\ &= q_n - q_n \\ &= 0. \end{aligned}$$

Also since $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor$ is the **greatest** integer less then or equal to $\frac{q_n}{q_{n+1}}$, it holds that $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor > \frac{q_n}{q_{n+1}} - 1$, so

$$\begin{aligned} q_{n+2} &= q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1} \\ &< q_n - \left(\frac{q_n}{q_{n+1}} - 1 \right) \cdot q_{n+1} \\ &= q_n - \frac{q_n}{q_{n+1}} \cdot q_{n+1} + q_{n+1} \\ &= q_{n+1}. \end{aligned}$$

I.e. q_n is a strictly decreasing sequence. Since q_n is a strictly decreasing sequence **of integer numbers**, bounded below by 0, one must have that q_n must be zero at one point, i.e.

$$\exists N \in \mathbb{N} : q_N = 0.$$

□

Theorem 2.2. *When $x \in \mathbb{R} \setminus \mathbb{Q}$ (x is irrational), the Euclidean algorithm for real numbers (Algorithm 2.2) will never terminate. However, one has that*

$$q_n \to 0, \quad \text{as } n \to \infty.$$

Proof. This proof follows the same method as the proof of theorem 2.1, but the result is slightly less strong, as q_n is no longer a sequence of integers, but irrational numbers. Let q_n be the sequence as

defined in the algorithm. Note that since $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \leq \frac{q_n}{q_{n+1}}$,

$$\begin{aligned} q_{n+2} &= q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1} \\ &\geq q_n - \frac{q_n}{q_{n+1}} \cdot q_{n+1} \\ &= q_n - q_n \\ &= 0. \end{aligned}$$

Also since $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor$ is the greatest integer less then or equal to $\frac{q_n}{q_{n+1}}$, it holds that $\left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor > \frac{q_n}{q_{n+1}} - 1$, so

$$\begin{aligned} q_{n+2} &= q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1} \\ &< q_n - \left(\frac{q_n}{q_{n+1}} - 1 \right) \cdot q_{n+1} \\ &= q_n - \frac{q_n}{q_{n+1}} \cdot q_{n+1} + q_{n+1} \\ &= q_{n+1}. \end{aligned}$$

I.e. q_n is a strictly decreasing sequence. Since q_n is a strictly decreasing sequence of irrational numbers, bounded below by 0, one must have that q_n converges. Now,

$$\begin{aligned} q_{n+2} &= q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+1} \\ q_{n+2} &< q_n - \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor \cdot q_{n+2} && \big\downarrow\ q_{n+2} < q_{n+1} \\ q_{n+2}\left(1 + \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor\right) &< q_n && \big\downarrow\ \textit{Note: } 1 + \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor > 0 \\ q_{n+2} &< \frac{q_n}{1 + \left\lfloor \frac{q_n}{q_{n+1}} \right\rfloor} && \big\downarrow\ q_{n+1} < q_n \\ q_{n+2} &< \frac{q_n}{1 + \left\lfloor \frac{q_{n+1}}{q_{n+1}} \right\rfloor} \\ q_{n+2} &< \frac{q_n}{1+1} = \frac{1}{2} q_n. \end{aligned}$$

Meaning that after two iterations q_n will at least be halved. Since also $p_n \geq 0$, it automatically follows that

$$q_n \to 0, \quad \text{as } n \to \infty$$

□

An interesting result of theorem 2.1 that automatically follows is that a simple continued fraction of a number is finite if and only if the number is rational, as the algorithm terminates. Similarly due to theorem 2.2 one has that a simple continued fraction of a number is infinite if and only if the number is irrational:

Corollary 2.2.1. *Let* $x \in \mathbb{R}$.

The simple continued fraction of x *is finite* $\Leftrightarrow x \in \mathbb{Q}$.
The simple continued fraction of x *is infinite* $\Leftrightarrow x \in \mathbb{R} \setminus \mathbb{Q}$.

2.3. Convergents

In this section, convergents and their representation are explained. (Bosma and Kraaikamp, 2012)

Definition 2.3 (Convergents)**.** Let $x \in \mathbb{R}$, with continued fraction $[a_0; a_1, a_2, \dots]$. Then

$$x_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots\ \cfrac{1}{a_n}}}, \quad n \in \mathbb{N} \cup \{0\}.$$

is called the n-th **convergent** of x.

For example, when $x = [a_0; a_1, a_2, a_3, \dots]$, the first convergents are

$$x_0 = a_0 = \frac{a_0}{1},$$
$$x_1 = a_0 + \frac{1}{a_1} = \frac{a_1 a_0 + 1}{a_1},$$
$$x_2 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2}} = \frac{a_2(a_1 a_0 + 1) + a_0}{a_2 a_1 + 1},$$
$$x_3 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3}}} = \frac{a_3(a_2(a_1 a_0 + 1) + a_0) + (a_1 a_0 + 1)}{a_3(a_2 a_1 + 1) + 1}.$$

One can see that the numerator of a convergent is obtained by multiplying the digit by the numerator of the previous convergent, and adding the numerator of the convergent before the previous convergent, so

$$l_{n+2} = a_n \cdot l_{n+1} + l_n. \tag{2.1}$$

The same holds for the denominator

$$k_{n+2} = a_n \cdot k_{n+1} + k_n. \tag{2.2}$$

The first values are $k_0 = 1$, $k_1 = 0$, $l_0 = 0$, and $l_1 = 1$. This, and equations (2.2) and (2.1) are now proven.

Theorem 2.3. *Let $x \in \mathbb{R}$. The n-th convergent of x is defined by the recursive formula*

$$x_n = \frac{l_{n+2}}{k_{n+2}} = \frac{a_n \cdot l_{n+1} + l_n}{a_n \cdot k_{n+1} + k_n}, \quad \textit{where } \begin{pmatrix} k_0 \\ l_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \textit{ and } \begin{pmatrix} k_1 \\ l_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.3}$$

Proof. This theorem is a consequence of the Extended Euclidean algorithm. The algorithm states (Wikipedia contributors, 2021) that q_n is of the form:

$$q_{2n} = k_{2n} \cdot p - l_{2n} \cdot q$$

$$q_{2n+1} = -k_{2n+1} \cdot p + l_{2n+1} \cdot q$$

Using theorem 2.2, one has that

$$\begin{aligned} q_n &= \mp k_n \cdot p \pm l_n \cdot q \\ \pm\frac{q_n}{q \cdot k_n} &= -\frac{p}{q} + \frac{l_n}{k_n} \\ \frac{p}{q} \pm \frac{q_n}{q \cdot k_n} &= \frac{l_n}{k_n} \\ \lim_{n\to\infty}\left(\frac{p}{q} \pm \frac{q_n}{q \cdot k_n}\right) &= \lim_{n\to\infty}\frac{l_n}{k_n} \\ \frac{p}{q} \pm 0 &= \lim_{n\to\infty}\frac{l_n}{k_n} \\ \lim_{n\to\infty}\frac{l_n}{k_n} &= \frac{p}{q}. \end{aligned}$$

I.e. the k_n and l_n from the Extended Euclidean algorithm are the denominator and numerator of the n-th convergent. This means that if the recursive statement can be proven for the algorithm, it is then automatically also true for the numerator and denominator of the convergents.

Suppose one is at an odd step of the algorithm, say the $(2n+1)$-th. The standard Euclidean algorithm (Algorithm 2.1) then yields that:

$$\begin{aligned} q_{2n+2} &= q_{2n} - \left\lfloor \frac{q_{2n}}{q_{2n+1}} \right\rfloor \cdot q_{2n+1} \\ &= q_{2n} - a_{2n} \cdot q_{2n+1} \\ &= (k_{2n} \cdot p - l_{2n} \cdot q) - a_{2n} \cdot (-k_{2n+1} \cdot p + l_{2n+1} \cdot q) \\ &= (a_{2n} \cdot k_{2n+1} + k_{2n}) \cdot p - (a_{2n} \cdot l_{2n+1} + l_{2n}) \cdot q \end{aligned}$$

For the even steps of the algorithm, this result can be derived in the same way. So indeed $k_{n+2} = a_n \cdot k_{n+1} + k_n$ and $l_{n+2} = a_n \cdot l_{n+1} + l_n$.

Furthermore, note that the initial values of the Extended Euclidean algorithm require that $k_0 = 1$, $k_1 = 0$, $l_0 = 0$, and $l_1 = 1$. □

It can be shown that the even-order convergents x_{2n} monotonously increase to x and the odd-order convergents x_{2n+1} monotonously decrease to x. To show this, the following lemma comes in handy.

Lemma 2.4. *For all* $n \geq 0$,

$$l_n \cdot k_{n+1} - k_n \cdot l_{n+1} = (-1)^{n+1}.$$

Proof. If the denominator of equation (2.3) is multiplied by l_{n+1} and the numerator of equation (2.3) multiplied by k_{n+1} is then subtracted, one finds:

$$\begin{aligned} l_{n+1} \cdot k_{n+2} &= l_{n+1} \cdot (a_n \cdot k_{n+1} + k_n) \\ k_{n+1} \cdot l_{n+2} &= k_{n+1} \cdot (a_n \cdot l_{n+1} + l_n) \quad - \\ \hline l_{n+1} \cdot k_{n+2} - k_{n+1} \cdot l_{n+2} &= l_{n+1} \cdot k_n - k_{n+1} \cdot l_n \end{aligned}$$

or

$$l_{n+1} \cdot k_{n+2} - k_{n+1} \cdot l_{n+2} = -(l_n \cdot k_{n+1} - k_n \cdot l_{n+1}).$$

Furthermore, since

$$l_0 \cdot k_1 - k_0 \cdot l_1 = 0 \cdot 0 - 1 \cdot 1 = -1,$$

it follows that indeed

$$l_n \cdot k_{n+1} - k_n \cdot l_{n+1} = (-1)^{n+1}.$$

□

Theorem 2.5. *The convergents x_n form a monotone increasing sequence converging to x if n is even, and a monotone decreasing sequence converging to x if n is odd, that is:*

$$x_0 \leq \ldots \leq x_{2n} \leq x_{2n+2} \leq \ldots \leq x \leq \ldots \leq x_{2n+3} \leq x_{2n+1} \leq \ldots \leq x_1.$$

Proof. If the denominator of equation (2.3) is multiplied by l_n and the numerator of equation (2.3) multiplied by k_n is then subtracted, one finds:

$$\begin{aligned} l_n \cdot k_{n+2} &= l_n \cdot (a_n \cdot k_{n+1} + k_n) \\ k_n \cdot l_{n+2} &= k_n \cdot (a_n \cdot l_{n+1} + l_n) \quad - \\ \hline l_n \cdot k_{n+2} - k_n \cdot l_{n+2} &= a_n \cdot k_{n+1} \cdot l_n - a_n \cdot l_{n+1} \cdot k_n \end{aligned}$$

or

$$l_n \cdot k_{n+2} - k_n \cdot l_{n+2} = a_n \cdot (l_n \cdot k_{n+1} - k_n \cdot l_{n+1}).$$

Using lemma 2.4, one has

$$l_n \cdot k_{n+2} - k_n \cdot l_{n+2} = a_n \cdot (-1)^{n+1},$$

which can be rewritten to

$$\frac{l_{n+2}}{k_{n+2}} = \frac{l_n}{k_n} + \frac{a_n \cdot (-1)^n}{k_{n+2} \cdot k_n}.$$

Since a_n and k_n are both non-negative sequences, the result follows. □

3

Finding the integer point

Now that the key theorems and definitions about continued fractions are known, it is time to apply them to the problem of the colliding planets. Since the convergents of the continued fraction p/q approach the slope of the band, they can be used as basis vectors to find the point (k, l) quickly. This is called lattice basis reduction. The algorithm for lattice basis reduction in $\mathbb{Z}^2$ is closely related to the Euclidean algorithm and thus continued fractions (Galbraith, 2012).

3.1. Lattice basis reduction

Theorem 2.3 yields a method for finding (k, l) such that

$$\frac{l}{k} \approx \frac{p}{q}.$$

In matricial description, equation (2.3) becomes:

$$\begin{pmatrix} k_{n+1} & k_{n+2} \\ l_{n+1} & l_{n+2} \end{pmatrix} = \begin{pmatrix} k_n & k_{n+1} \\ l_n & l_{n+1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & a_n \end{pmatrix}. \tag{3.1}$$

Since the first matrix is of the form $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and the transformation matrix $\begin{pmatrix} 0 & 1 \\ 1 & a_n \end{pmatrix}$ is unimodular, the matrices $\begin{pmatrix} k_{n+1} & k_{n+2} \\ l_{n+1} & l_{n+2} \end{pmatrix}$ from equation (3.1) will form a sequence of lattice basis for $\mathbb{Z}^2$ (Berthé, 2019). For simplicity, define the bases $\{\mathbf{b}_n, \mathbf{b}_{n+1}\}$ for $\mathbb{Z}^2$ as:

$$\mathbf{b}_n = \begin{pmatrix} k_n \\ l_n \end{pmatrix}.$$

The slopes of the basis vectors are then the convergents of p/q. Therefore, by theorem 2.5, the slopes of the even basis vectors increase and the slopes of the odd basis vectors decrease to p/q. This can also be seen in Figure 3.1.

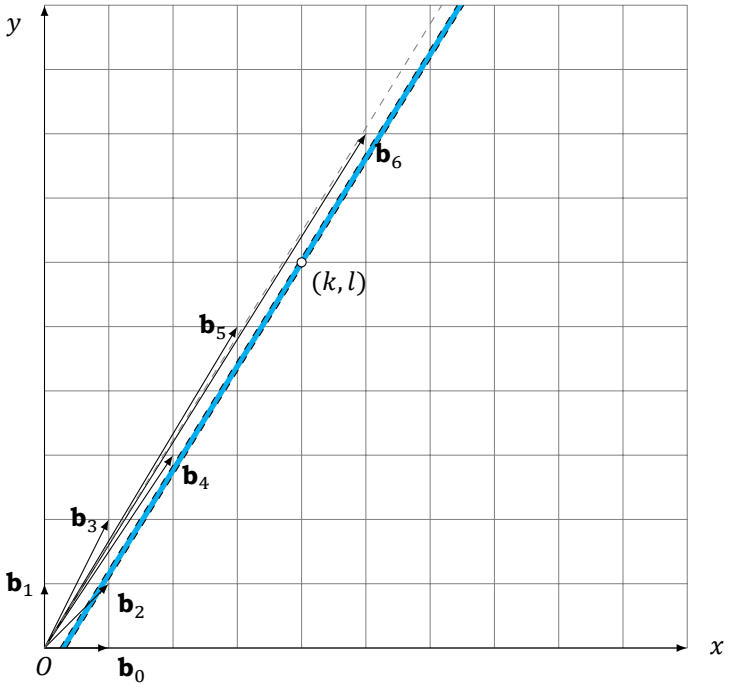

Figure 3.1: The blue band in which an integer point (k, l) must be found. Lattice basis reduction is used to find (k, l). For the basis vectors the convergents of the continued fraction p/q are used. The first 6 lattice basis vectors and a dashed line with slope p/q to which the vectors converge are drawn.

3.2. Lattice basis reduction to find the integer point

The algorithm for finding the integer point (k, l) using lattice basis reduction actually consist of two parts; Checking smaller parallelograms for integer points and switching to the next basis vectors.

3.2.1. Checking the parallelogram for integer points

To check the band for integer points, it is divided into smaller parallelograms. The considered parallelogram is the part of the band from $y = 0$ to the y-value where the bottom of the band intersects the span of the vector $\mathbf{b}_{n+2}$. This point of intersection always exists as the slope of the even basis vectors are always less than the slope of the band. Examples of such parallelograms are shown in red in Figure 3.2(a), and in green and yellow in Figure 3.2(d).

Checking the parallelogram for integer points can of course be done in multiple ways. The method that Visser (2022) uses checks for every y-value if the x-value that is closest to the band, lies in the band: Call it y-search. For every integer y between 0 and the top of the parallelogram, calculate the x-value of the left line, round it up, and check if it lies in the band, i.e. check if it is lower than the right line. Figure 3.2(a) and 3.2(d) show the checked integer points in dark blue.

3.2.2. Switching to the next basis vectors

If an integer point lies in the parallelogram, a solution is found. If not, the next parallelogram needs to be checked. This is done in new basis vectors $\{\mathbf{b}_{n+2}, \mathbf{b}_{n+3}\}$ (see Figure 3.2(b) and 3.2(c)). Since the odd basis vectors decrease, and the even basis vectors increase to the slope of the blue band, the considered area gets smaller and smaller. This can be clearly seen in Figure 3.2(b); After the basis transformation the light gray area is no longer considered. When transforming to the next two basis vectors two things happen. Firstly, the width of the band increases. Furthermore, the points on the grid are drawn closer to the origin. These things can be seen when comparing Figures 3.2(a) and 3.2(c). Both of these things contribute to finding the integer point (k, l) faster.

Note that the parallelogram after a basis transformation is drawn in green and yellow (see Figure 3.2(d)). The yellow part has already been looked at when checking the red parallelogram. Unfortunately the successive parallelograms overlap and therefore some pieces of the band are checked for integer points twice.

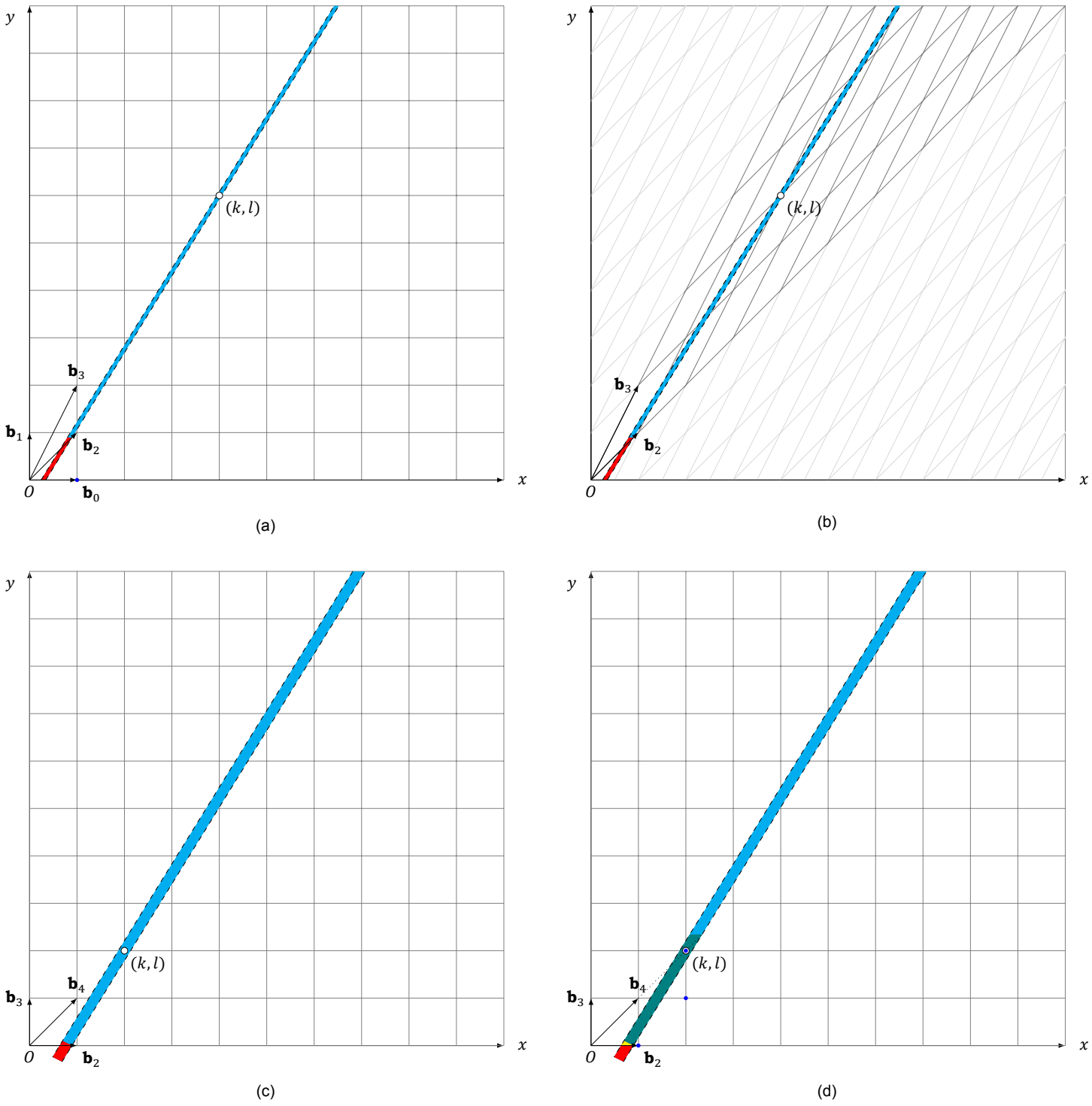

Figure 3.2: Four steps to find an integer point (k, l) using lattice basis reduction. First the parallelogram of the band between $y = 0$ and the y-value where the bottom of the band intersects span$\{\mathbf{b}_{n+2}\}$ is checked for integer points. This parallelogram is shown in red. The point $(1, 0)$ is checked, but does not lie in the parallelogram. Since no integer point is found, the grid is drawn with the next two basis vectors $\{\mathbf{b}_{n+2}, \mathbf{b}_{n+3}\}$ and again the parallelogram of the band between $y = 0$ and the y-value where the bottom of the band intersects the span of the next basis vector is considered. This parallelogram is shown in green and yellow. The yellow part has already been looked at when checking the red parallelogram. The points that are checked for this parallelogram are $(1, 0)$, $(2, 1)$ and $(2, 2)$. Since the latter is in the parallelogram, the algorithm stops here: (k, l) has been found!

3.3. The algorithm in practice

When Matlab is used (see Appendix A.1) to check the algorithm in practice, Figure 3.3 arises. The figure contains a random run of the algorithm with Vissers y-search. In the code, a maximum orbital period is set:

$$\mathrm{tmax} = 1.$$

This t_{max} is the orbital period of the first planet. For the second planet, the t_{max} times a random number in the interval $(0, 1)$ is used:

$$\mathrm{T1} = \mathrm{tmax}, \quad \text{and} \quad \mathrm{T2} = \mathrm{tmax}{*}\mathrm{rand}.$$

With these T_1 and T_2 random t_1 and t_2 are chosen, and using these the p and q are made:

$$\mathrm{p} = \mathrm{T1}/\mathrm{abs}(\mathrm{t1}{-}\mathrm{t2}), \quad \text{and} \quad \mathrm{q} = \mathrm{T2}/\mathrm{abs}(\mathrm{t1}{-}\mathrm{t2})).$$

The δ is set to be 10^{-i}, where i is an integer in the interval $[1, 10]$:

$$\mathrm{delta} = 10\hat{}(-\mathrm{randi}([1{,}10])).$$

This run needed two basis transformations. In the first three plots the band (blue), the next two basis vectors (dashed, black) and the checked points (dark-blue) are drawn. It can now also be clearly seen that for each y-value one x-value is checked. The last plot contains a zoomed-in version of the blue band with the found integer point $(k, l) = (458, 8564)$ in it.

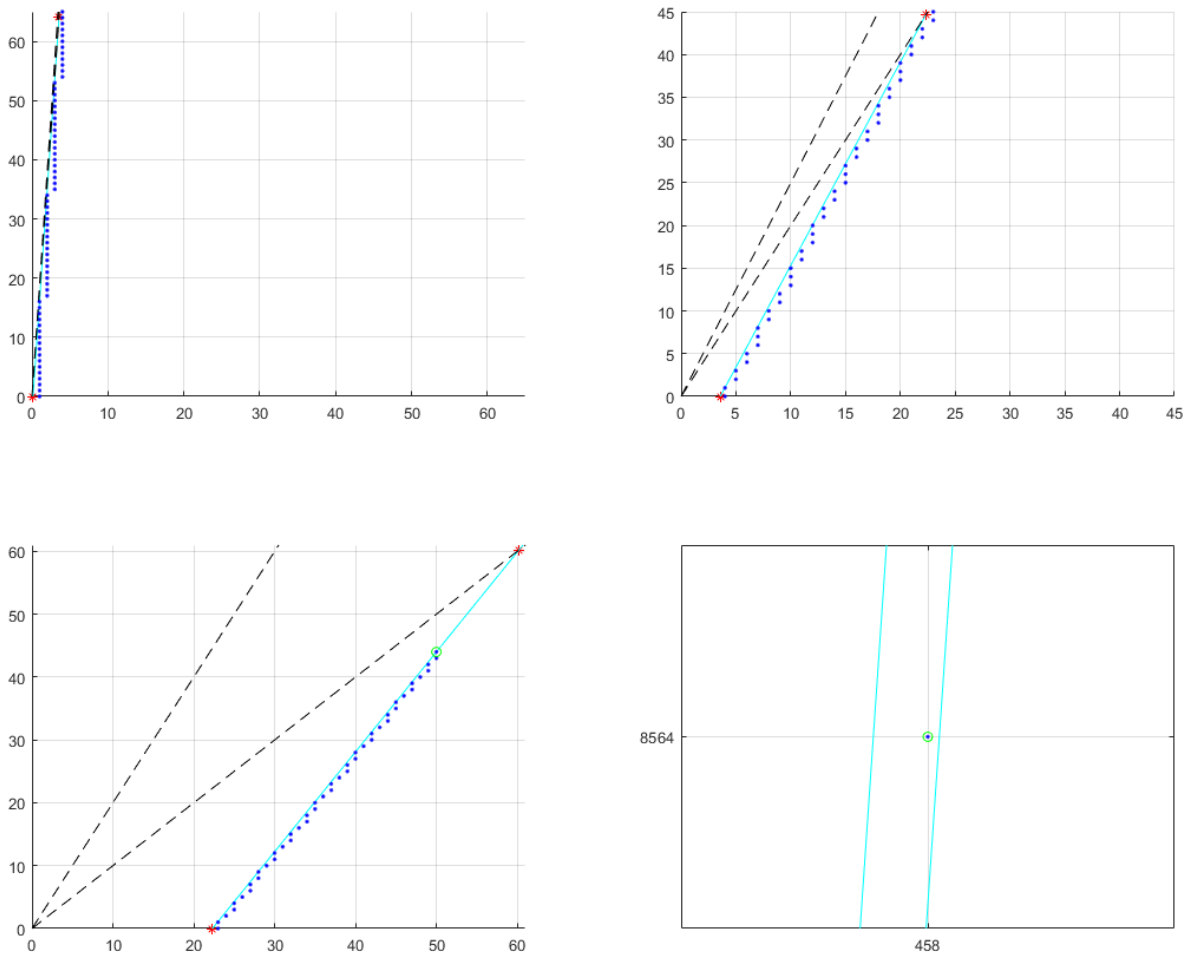

Figure 3.3: An example of a run with the algorithm which needed two basis transformations. In the first three plots the blue band is drawn together with the next two basis vectors (dashed, black). The dark blue dots represent the checked points. The last plot contains a zoomed-in version of the blue band with the found integer point $(k, l) = (458, 8564)$ in it.

3.3.1. Improving the algorithm

Since it is assumed that $T_1 > T_2$, one has that $p/q > 1$. I.e. the slope of the band is greater than the slope of $y = x$. Therefore, the method that is used by Visser can be improved if every x-value of the parallelogram is checked, instead of checking every y-value (Call it x-search). For every integer x between the left and right of the parallelogram, calculate the y-value of the lowest line, round it up, and check if it lies in the band, i.e. check if it is lower than the highest line. Figure 3.4 shows the plot that arises when a random run with the improved algorithm is performed (see Appendix A.2 for the code). It can be seen that now all x-values are checked instead of all y-values.

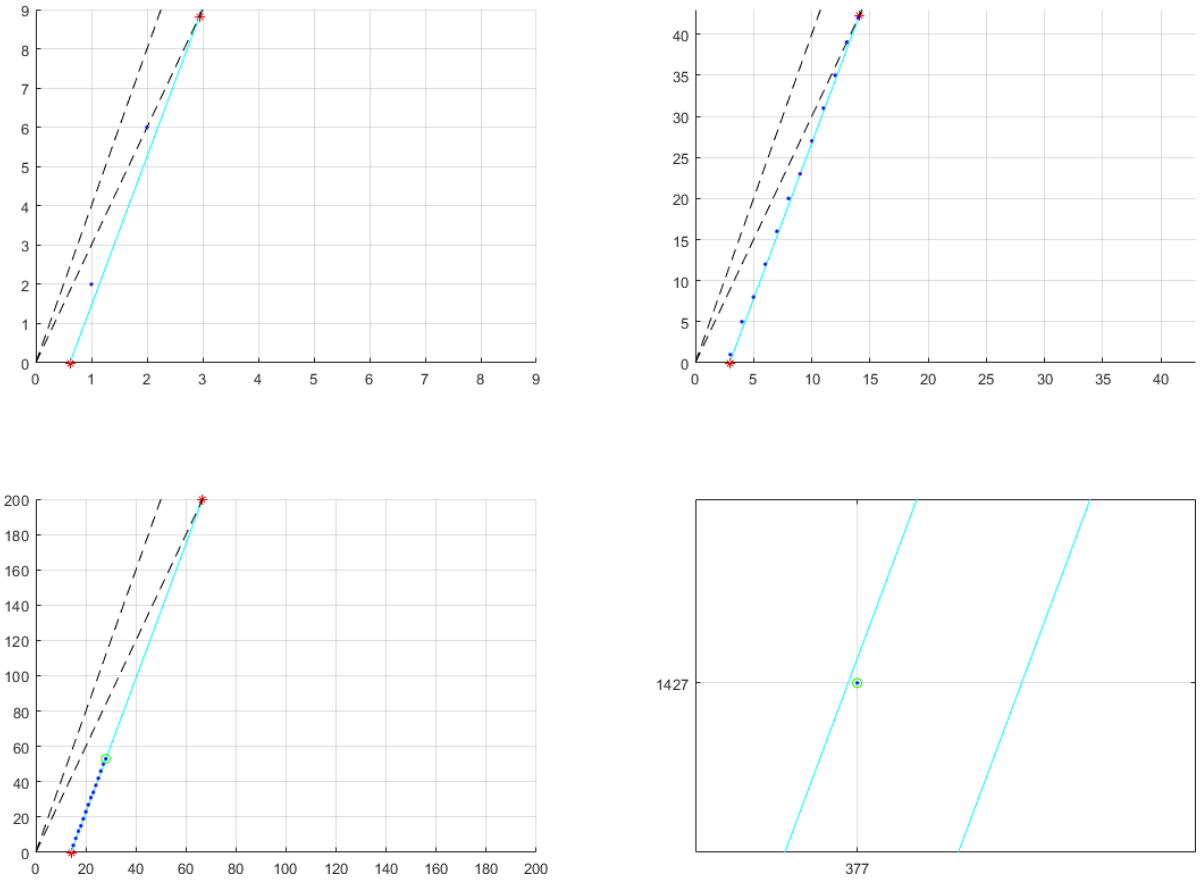

Figure 3.4: An example of a run with the improved algorithm which needed two basis transformations. In the first three plots the blue band is drawn together with the next two basis vectors (dashed, black). The dark blue dots represent the checked points. It can be seen that all x-values are checked instead of all y-values. The last plot contains a zoomed-in version of the blue band with the found integer point $(k, l) = (377, 1427)$ in it.

3.4. The algorithms efficiency

Because the aim was to give insight in a fast algorithm, in contrast to the computationally expensive simulations, it is of course important to see how fast this algorithm really is. This section discusses the number of steps and basis transformations required by the algorithm. Also more insight is given into which steps are skipped compared to a brute force method.

3.4.1. Run-time analysis

The number of steps when using the method of Visser (y-search) is at most l (because in the worst case all y-values have to be checked). The improved method (x-search) has a worst case number of steps of k (because in the worst case all x-values have to be checked). This only occurs if no basis transformations are done, so luckily k is then often small. Since $k < l$, the improved method is always faster than the original method. The two methods are tested in Matlab with random p, q and δ a million (10^6) times (see Appendix A.3). If the number of steps is saved for the million runs, a frequency

histogram can be made (see Figure 3.5). It can be clearly seen that in practice the number of steps of the improved method is often much lower than the original method. To see which number of steps appears the most, the plot on the right in Figure 3.5 can be used. Many runs with a low number of steps are preferred, i.e. the top of the graph should be as far to the left as possible.

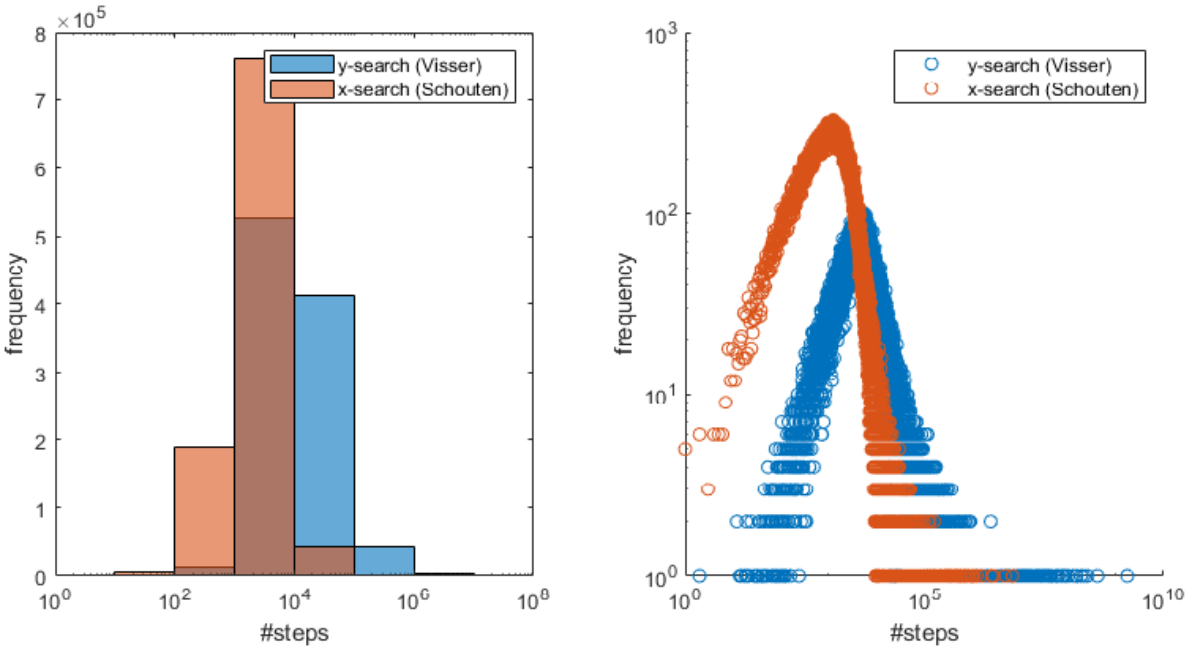

Figure 3.5: The method of Visser and the improved method (Schouten) are tested 10^6 times using Matlab. In the left plot can be seen that the original method takes more steps than the improved method. To see which number of steps appears the most, the plot on the right can be used. Many runs with a low number of steps are preferred, i.e. the top of the graph should be as far to the left as possible.

The worst case run-time of the improved algorithm is k, but how often is the algorithm really worst case? If the continued fraction algorithms (both x-search and y-search) are compared to a simple brute force method that checks all the points (again, both x-search and y-search), are they often (much) faster? Figure 3.6 provides insight into these questions. Four methods (brute force x- and y-search and the continued fraction method x- and y-search) have been tested a million times with random p, q and δ (see Appendix A.4). As expected, the brute force; x-search always lies on the line $y = x$, i.e. Number of steps $= k$, as it just check all the points until k; the first point inside the band. The brute force; y-search is always above the line $y = x$, so a little slower. This is because the number of steps is now l, and $k < l$. The points for the continued fraction methods are difficult to see, because they are often much lower than the brute force methods. This is why the same plot has been made, but with the logarithm of the number of steps and k, to bring the points a little closer together. In practice the number of steps for both methods is often much lower than the worst case k for the x-search and l for the y-search.

Another topic that may be of interest is the number of basis transformations. Furthermore, if the number of transformations is known, one can also determine how many steps are needed on average per basis; the number of steps divided by the number of basis transformations plus one. (The $+1$ is necessary as the first steps start without performing a basis transformation.) The improved algorithm is run again a thousand times with different p, q and δ (see Appendix A.5). For each run, the number of steps and the number of bases are tracked, and with these the average number of steps per basis is calculated. The three are plotted against k in a log-log plot in Figure 3.7. Also three lines have been drawn; $y = x$, $y = \sqrt{x}$ and $y = \log(x)$. It can (again) be seen that the worst case is k, because there are a few points on the line $y = x$, but never above it. Furthermore, it looks like the number of steps seems to follow the line $y = \sqrt{x}$. This would imply that the algorithm is (unfortunately) not exponentially fast, but $\sqrt{k}$ fast. The number of bases however do seem exponential, since they to follow the line $y = \log(x)$. Note that since it is a log-log plot, the values of the number of bases plus the number of steps per basis equals the number of steps.

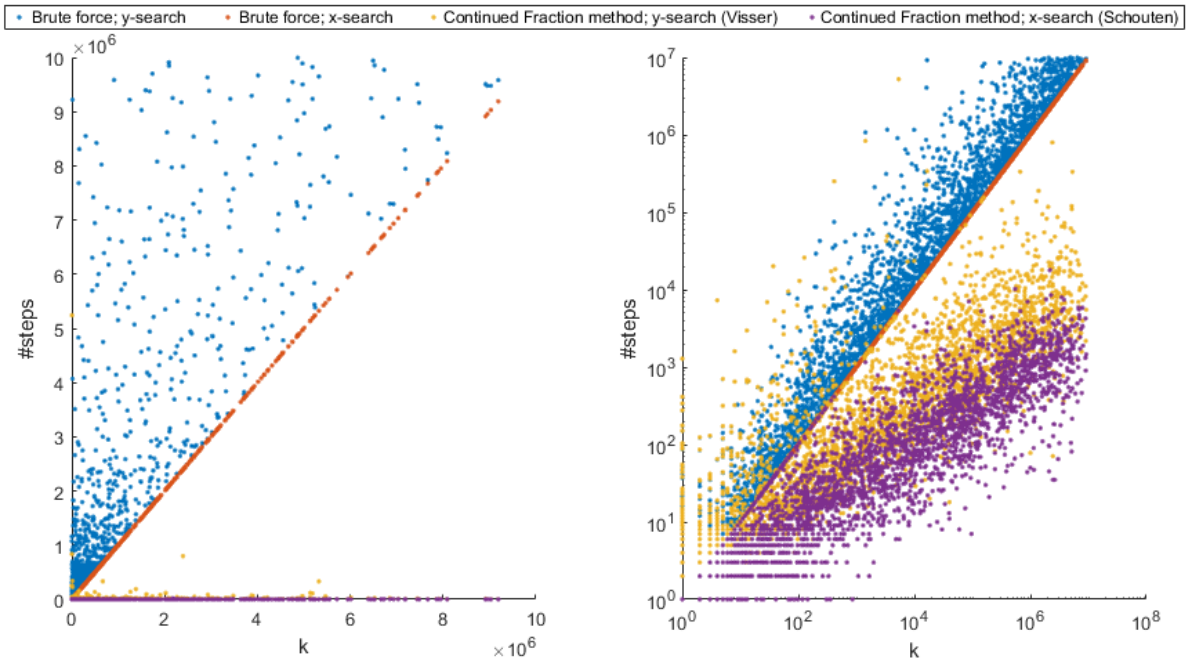

Figure 3.6: Four methods (brute force x- and y-search and the continued fraction method x- and y-search) have been tested a million times with random p, q and δ. Two plots are made; the number of steps are plotted against k to properly see the relationship between the number of steps and k, and the logarithm of the number of steps are plotted against the logarithm of k, to distribute the points a little bit better on the canvas. In practice the number of steps for both methods is often much lower than the worst case k for the x-search and l for the y-search.

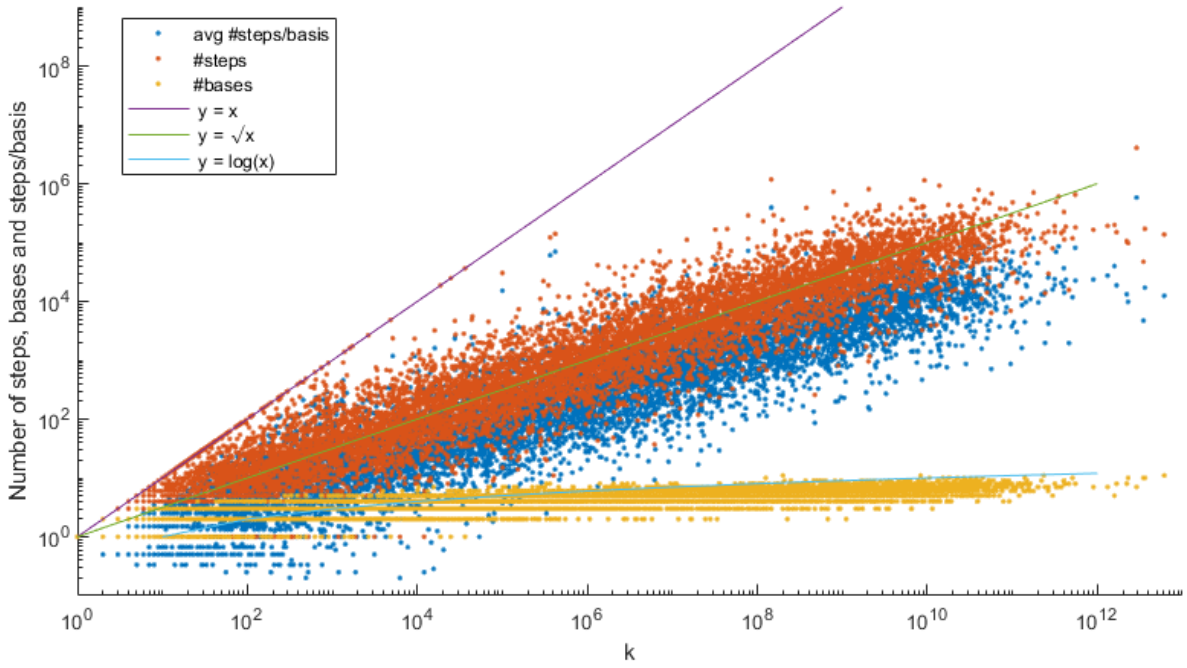

Figure 3.7: The improved algorithm is run a thousand times with random p, q and δ. A log-log plot is made with the number of steps, the number of bases and the average number of steps per basis. It can be seen that the number of steps seem to follow the line $y = \sqrt{x}$, and that the worst case is k; the points on the line $y = x$. The number of bases seem to follow the line $y = \log(x)$.

3.4.2. The skipped steps

It is now clear that the continued fraction method is indeed faster than a simple brute force method. A question that then may arise is; which steps are skipped when using lattice basis reduction? Figure 3.8 shows the steps made with brute force in rainbow colors. The points circled in red are the points that would also be checked with the continued fraction method. It can be seen that in the beginning every point is checked with both methods, but that over time more and more points are skipped by the continued fraction method. Figure 3.9 provides more insight into why these points are skipped. Exactly the same points are plotted again, but now in the transformed bases. It is now very clear why so many points can be skipped. After a basis transformation, a lot of points which one would check with brute force method (as they would lie close to the band) are now suddenly transformed far away from the band and can therefore be skipped. Moreover, it can be seen that after each basis transformation more and more points are 'transformed away' from the band meaning more and more points can now be skipped.

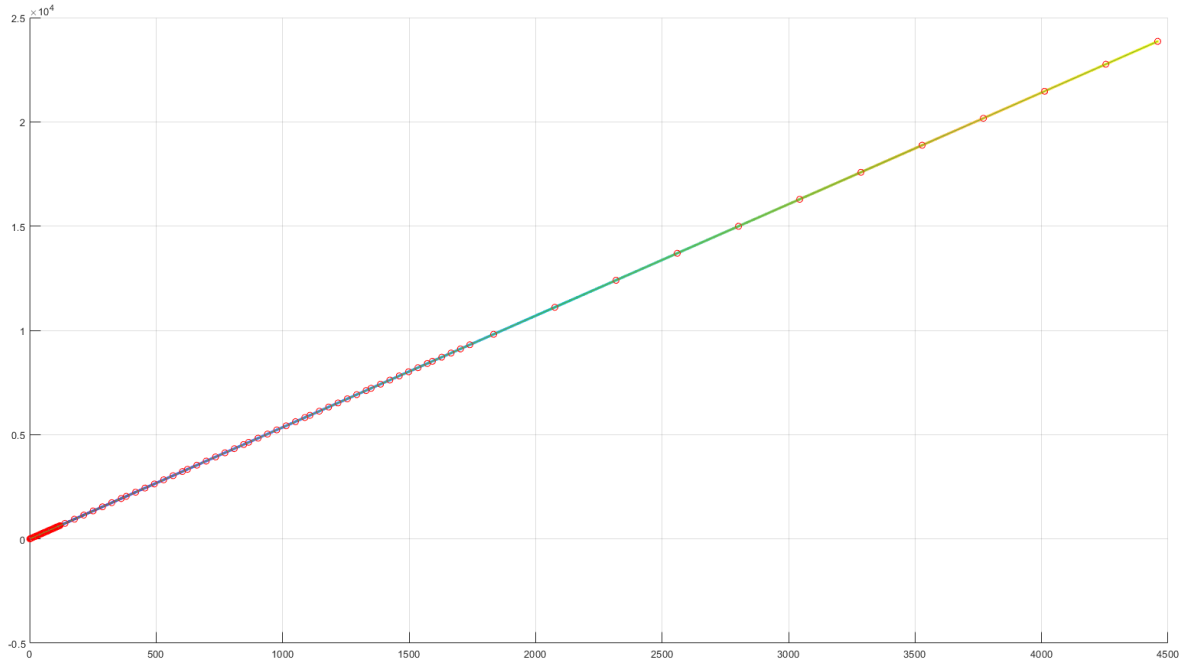

Figure 3.8: The points checked with the brute force method in rainbow colors and the points checked with the continued fraction method circled in red. In the beginning all points are checked by both methods, but over time the continued fraction method skips more and more points.

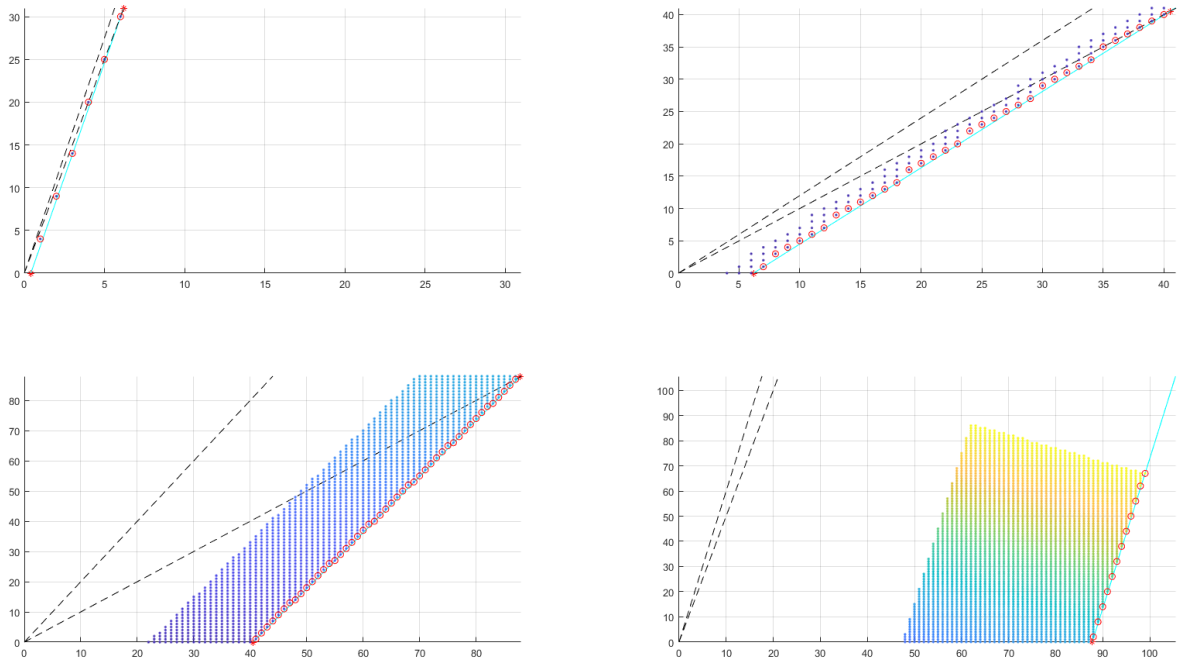

Figure 3.9: The points checked with the brute force method in rainbow colors and the points checked with the continued fraction method circled in red. However, now all points are plotted in the transformed bases. It is now clear why so many points can be skipped; after each basis transformation, points that were close to the band are suddenly transformed away from the band. In other words; a lot of points that are checked with the brute force method can be skipped with the continued fraction method.

4

When no solution can be found

This chapter examines the second question posed in the introduction; Does (k, l) always exist? First, it is explained when there is no exact solution; when is there no integer point in the band and do the planets not collide? After that, statistics are used to explain what to do if, due to for example inaccurate observational data or a lack of machine precision, no exact solution can be found.

4.1. No integer point in the band

When $p/q \notin \mathbb{Q}$, there will always be an exact solution, assuming there is no uncertainty in the data. By corollary 2.2.1, the continued fraction of such p/q is infinite so the basis transformations could go on indefinitely. Since the band gets thicker with every transformation, it will eventually have a width of 1, and then there must be an integer point in the band. (Often "coincidentally" a solution is found (far) before the band has width 1, due to δ.)

When $p/q \in \mathbb{Q}$ however, it could be that no solution exists. Two examples are given. If p and q are both integers the problem is then finding integer k and l such that:

$$k \cdot p - l \cdot q = 1,$$

where $p, q \in \mathbb{Z}$ now as well. Such integer point (k, l) exists if and only if p and q are coprime (Gijswijt, 2019), which is of course not always the case. Hence if p and q are not coprime, and δ is small enough, no solution exists. The band has then exactly the right offset and slope to lie between all grid points. Another case where there is no solution is when p is a multiple of q. Then the orbital period of planet 1 is a multiple of the orbital period of planet 2, i.e:

$$T_1 = m \cdot T_2, \text{ where } m \in \mathbb{N}.$$

A collision can then only occur if the time it takes for the planets to arrive at the collision point is the same, possibly plus an arbitrary number of full cycles around the orbit, $n \cdot T_1$:

$$\begin{aligned} t_1 + k \cdot T_1 &= t_2 + l \cdot T_2 \\ t_1 + k \cdot T_1 &= t_2 + l \cdot m \cdot T_1 \\ t_1 &= t_2 + (l \cdot m - k) \cdot T_1 \\ t_1 &= t_2 + n \cdot T_1 \end{aligned}$$

Note that in the last step $n = l \cdot m - k \in \mathbb{Z}$, which is the arbitrary number of cycles around the orbit.

4.2. The statistics of the solution

It could also be the case that the correct solution cannot be found due to imprecise observational data or a lack of machine precision. It may then be interesting to estimate k instead of calculating it exactly. There are often inaccuracies in simulations anyway, so estimating variables could save computing time. The effect of small errors on p, q and δ is investigated to find out what a (probability distribution of a) solution might then look like.

The integer points are uniformly distributed over the plane. The assumption is now made that the integer points are statistically independent. (This is of course incorrect; when the of location one point is known, the location of all points are.) If so, the probability that there is an integer point (k, l) in the band with k in the interval $[x, x + \mathrm{d}x]$ is then the area of the parallelogram of the band (if the area $<< 1$). This area is $(2\delta/q)\,\mathrm{d}x$, as the width of the parallelogram is $\mathrm{d}x$, and the height is $2\delta/q$. Hence, the probability that there is no integer point (k, l) in the band with $k < x$ is the product of one minus the probability that there ís an integer point in the small parallelograms from zero to x, i.e

$$\mathbb{P}(k > x) = \lim_{\mathrm{d}x \to 0} \left(1 - \frac{2\delta}{q}\,\mathrm{d}x\right)^{x/\mathrm{d}x} = \mathrm{e}^{-\frac{2\delta}{q}x},$$

and hence

$$\mathbb{P}(k \leq x) = 1 - \mathrm{e}^{-\frac{2\delta}{q}x}. \quad (Visser, 2022)$$

The probability density function, which is then the derivative of this probability, will be:

$$p(k) = \frac{2\delta}{q}\,\mathrm{e}^{-\frac{2\delta}{q}k}. \tag{4.1}$$

Now, a fixed p, q and δ are chosen (randomly). Then the algorithm is run a thousand times, but at each run a small error is added to the p, q and δ (see Appendix A.6). For p for example:

```
p = p*(1 + 10^(-4)*(-1+2*rand));
```

When a normalized histogram of the frequency of k is made, Figure 4.1 is arises. The probability density function from equation (4.1) is added to the figure, and the expected value $2\delta/q$ is also plotted. As can be seen in the figure, equation (4.1) does not quite give the correct probability density function. Especially when the y-values are plotted as a logarithm (see Figure 4.2); the simulated data is, (in contrast to the probability density function of Visser) not even exponential, as it does not become a straight line.

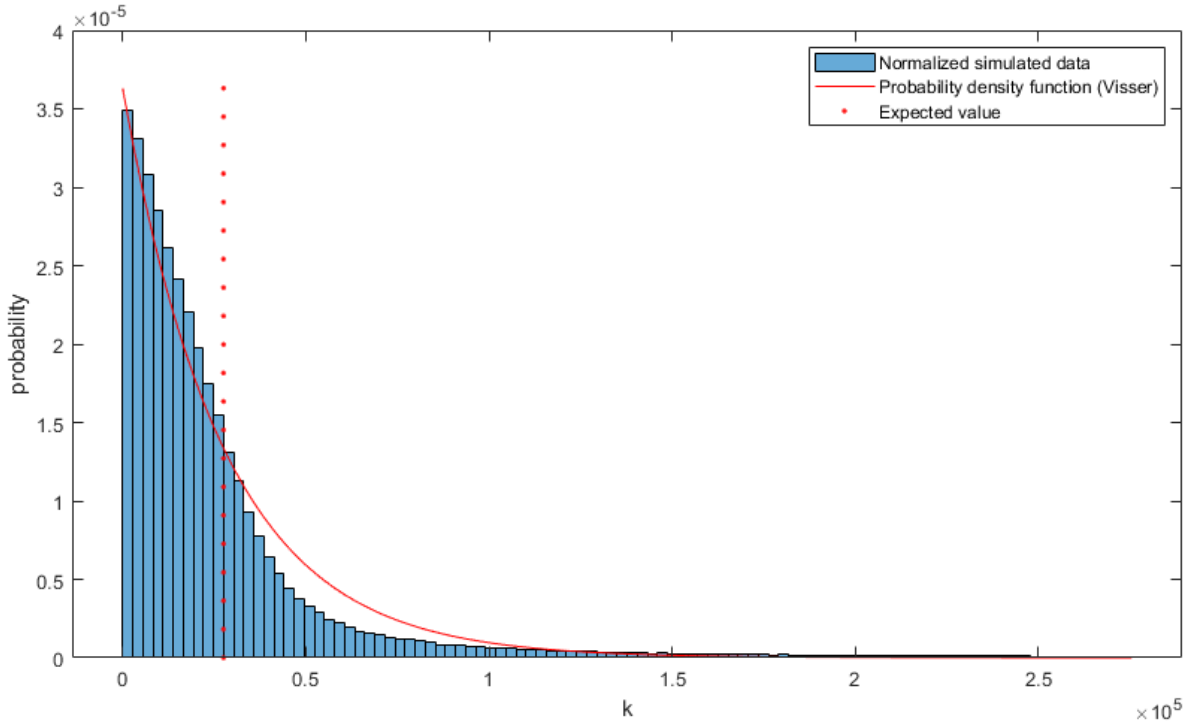

Figure 4.1: To validate the probability density function, the algorithm is run a thousand times with small errors added to a fixed p, q and δ. In this plot, $\delta = 10^{-5}$. The normalized frequency histogram is shown in blue, with the probability density function in red. As can be seen they are not quite similar; the probability density function is not exactly correct. The red dotted line shows the expected value of the probability density function.

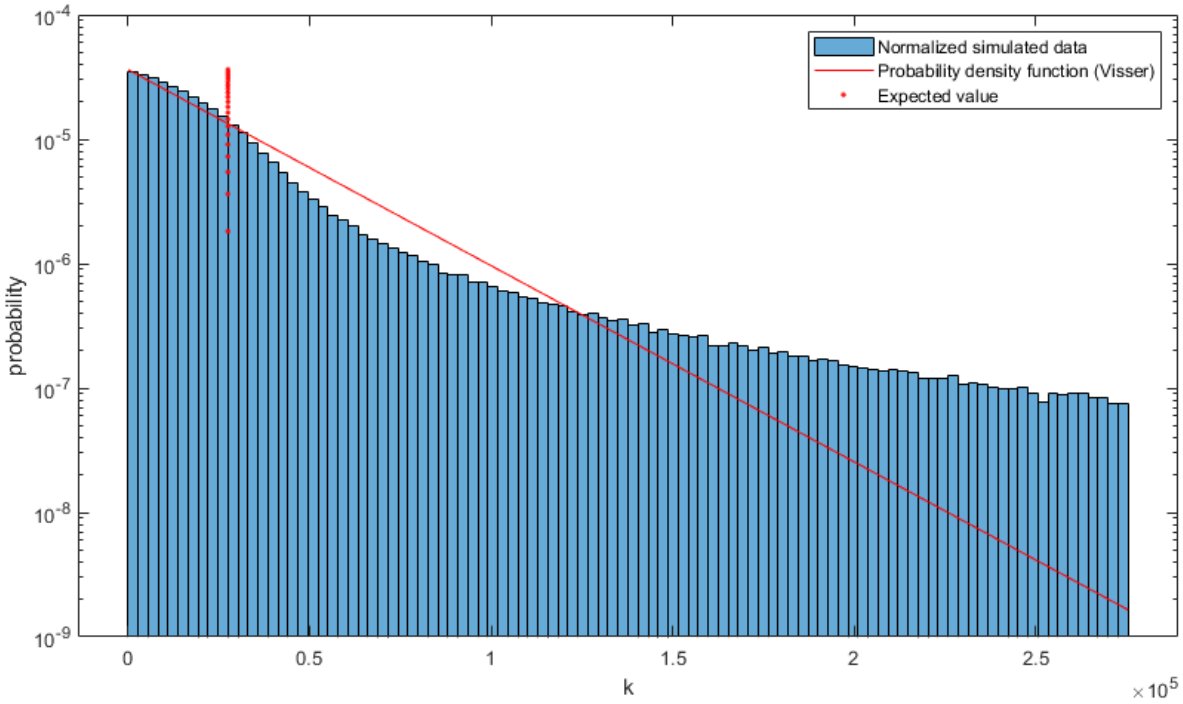

Figure 4.2: Again the normalized frequency histogram in blue, with the probability density function in red, with the same values as in Figure 4.1. Now however, the y-values have been plotted on a logarithmic scale. It can be clearly seen that the probability density function from equation (4.1) does not fit well. The simulated data does not even give a straight line, and is thus, unlike the expected probability density function, not exponential.

When the error decreases, the distribution will slowly shift to the true value of k; the value it would be if there were no error. This transition from a random solution to a deterministic solution can be seen in Figure 4.3. The relative error in this figure can be a maximum of $10^{-15.75}$ before it will affect the value of k that is found. If the error is larger, wrong values for k will be found. The maximum that the error may be is usually around δ^2. A large maximum error is desired. Not only because small measurement errors in for example the orbital periods of the planets have little influence, but also because numerical errors will be less influential.

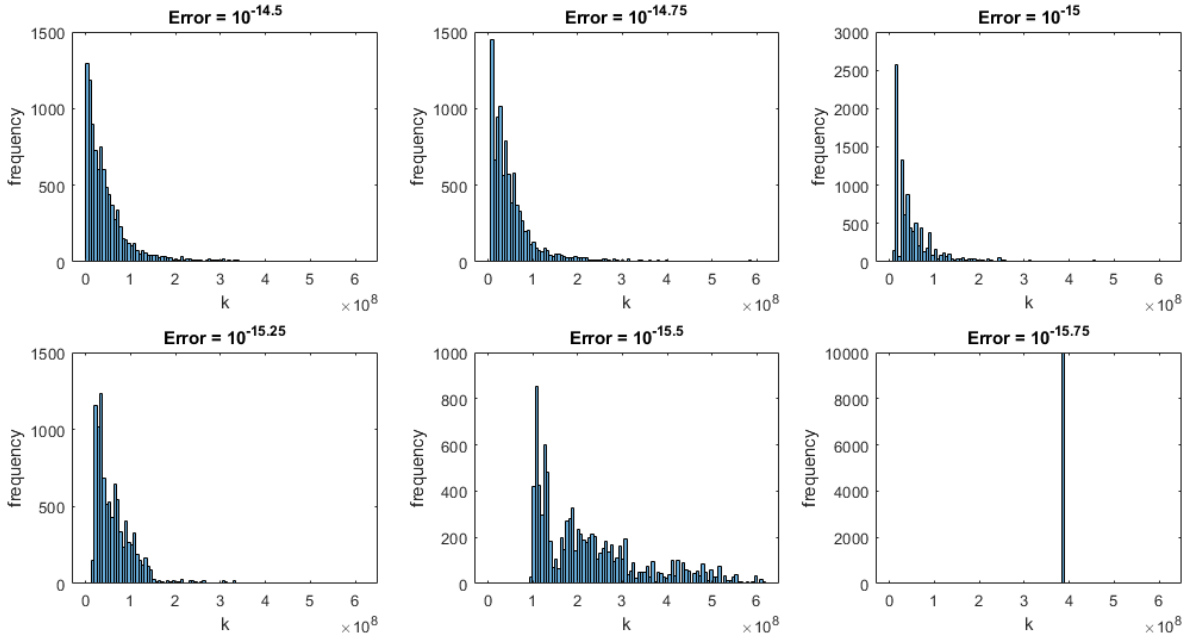

Figure 4.3: The error slowly decreases to the point where it no longer has any influence; the solution for k transitions from a random solution to a deterministic solution. When the relative error is $10^{-15.75}$ (or less) it no longer affects the k found; the exact value for k is then found on each run. The maximum for the relative error is around δ^2. In these plots $\delta = 10^{-8}$, so the transition to the deterministic solution was indeed expected around $\delta^2 = 10^{-16}$.

5

Discussion

Most of the sources used in this thesis are mathematics books published a long time ago. Their reliability is almost undisputed. However, one may have also noticed that Wikipedia is referenced a number of times. The critical reader may not consider this a good source, as anyone could edit Wikipedia's pages. However, since the pages used only contain mathematical facts (and, for example, no opinions about war), it can be assumed that these sources are also reliable.

Secondly, the machine precision and the precision of Matlab may cause errors in the results. However, a lot of testing has been done, so the chances that the results contain errors are reduced this way. In addition, an entire section has been devoted to what to do if an exact answer can not be found due to lack of machine precision.

In section 4.2 a probability density function is given. As seen however, it was not an exact match to the simulated data. An explanation for this could be that the assumption that the integer points are statistically independent was too harsh. Another explanation could be that the random error was not 'random enough'.

Furthermore, a few of assumptions have been made in the physical derivation of the two lines. For example, it is assumed that there is no mutual gravity, meaning the planets do not attract each other if they are close to each other. As a result, there is no nodal and apsidal precession, and therefore the collision point does not change over time. For further research it may be interesting to see what is the best approach if the mutual gravity is taken into account. Is linearization necessary? Are the lines still straight, and parallel?

Another interesting topic for further research may be the following: Planets follow the inverse-square law for gravitation, but charged particles, for example electrons, adhere to the inverse-square law for electrostatics. This means that charged particles also move in Kepler orbits, meaning the algorithm could perhaps also be applied in this field. Is the algorithm indeed usable for this, and perhaps in other fields?

6

Conclusion

This research was aimed at providing insight into a fast collision detection algorithm. It could be shown that the problem of collision detection between two planets is equivalent to finding the integer point in a band between two parallel lines, closest to the origin. Therefore, in this research an answer was sought to the question: How to quickly find an integer point (k, l) between two parallel lines? In addition, an answer was also sought to the question: Is it possible that there is no integer in the band and if so, when? To answer these questions, the article by Visser (2022) has been examined well and supplemented with other sources to provide good background information.

It has been shown that lattice basis reduction is a good method for finding the integer point quickly. As basis vectors, the convergents of the continued fraction of the slope of the band can be used, as the convergents provide good rational approximations for the (often) irrational slope. In addition, the basis vectors created using the convergents always span whole $\mathbb{Z}^2$. Dividing the band into small parallelograms allowed each parallelogram to be checked with the optimal basis. After each basis transformation, the width of the band increased and the points of the band were transformed closer and closer to the origin. Both these things contributed to find the integer point faster. Furthermore, by checking all points along the x-axis, and not the y-axis, the algorithm could be improved even more.

Where brute force always took k steps to find the integer point (k, l), the method with continued fractions and lattice basis reduction often found the point in about $\sqrt{k}$ steps. Only if no basis transformations were needed, the point (k, l) was found in the worst case k number of steps.

It has been shown that there is always a solution when the slope of the band is irrational. However, when the slope was rational, it could be that no solution existed. It was found that when δ was small enough, and the orbital periods of the planets not coprime (assuming they were integer), the band had the right offset and slope to lie between all grid points, and then no solution existed. Furthermore, when the orbital periods of the planets where a multiple of each other (not necessarily integer), the planets could only collide if the arrival time at the collision point for both planets happened to be exactly the same. Moreover, it could also be the case that a solution could not be found due to imprecise observational data, or a lack of machine precision. In that case, the most probable solution could be found using probability theory. However, the probability density function found did not exactly match the simulated data.

Bibliography

[1] V. Berthé. Lattice reduction and continued fractions. https://ssl.informatics.uow.edu.au/MACAO/workshop_2019/Berthe.pdf, November 2019. Presentation slides.

[2] W. Bosma and C. Kraaikamp. *Continued Fractions*. Radboud Universiteit Nijmegen, 2012.

[3] S. D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.

[4] D. C. Gijswijt. *Algebra 1*. Delft University of Technology, 2019.

[5] A. Ya. Khinchin. *Continued Fractions*. Dover Publications, 1949.

[6] J. Laskar. Stability of the solar system. http://www.scholarpedia.org/article/Stability_of_the_solar_system, September 2007. Retrieved May 3, 2022.

[7] A. M. Rockett and P. Szüsz. *Continued Fractions*. World Scientific Publishing Company, 1998.

[8] P. M. Visser. Collision detection for N-body Kepler systems. *Astronomy & Astrophysics*, 2022.

[9] Wikipedia contributors. Extended Euclidean algorithm. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm, October 2021. Retrieved April 4, 2022.

[10] Wikipedia contributors. Continued fraction. https://en.wikipedia.org/wiki/Continued_fraction, February 2022. Retrieved March 18, 2022.

[11] Wikipedia contributors. N-body simulation. https://en.wikipedia.org/wiki/N-body_simulation, April 2022. Retrieved May 4, 2022.

A

Matlab code

A.1. Vissers algorithm (y-search)

The code for the plots of Vissers algorithm. The % in line 61 can be removed to make the checked dots appear one by one:

```
1  clear; close all;
2
3  tmax = 1;
4  T1 = tmax;
5  T2 = tmax*rand;
6  t1 = T1*rand;
7  t2 = T2*rand;
8  p = T1/abs(t1-t2);
9  q = T2/abs(t1-t2);
10 delta = 10^(-randi([1,10]));
11
12 q0 = p; q1 = q;
13 k0 = 1; k1 = 0;
14 l0 = 0; l1 = -1;
15
16 figure;
17 for n = 0:100
18     a0 = floor(q0/q1);
19     q2 = q0 - a0*q1;
20     if q2 == 0
21         k = "No solution yet";
22         l = "No solution yet";
23         break
24     end
25     k2 = k0 - a0*k1;
26     l2 = l0 - a0*l1;
27     a1 = floor(q1/q2);
28
29
30     x = 0: ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)): ...
           ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2));
31     y = x*(q0/q1) - 1/q1;
32     left = y - delta/q1;
33     right = y + delta/q1;
34
35     A = [k0, -k1;
36          l0, -l1];
37
38     b2 = A\[k2; l2];
39     b3 = -A\[k1 - a1*k2; l1 - a1*l2];
40
41     subplot(3,3,n+1)
42     hold on;
43     plot((1-delta)/q0,0,'r*')
44     plot((1+delta)/q0,0,'r*')
```

```
45      plot((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2),(1+delta)*(q0-q2)/(q1*q2),'r*')
46      plot((1+delta)/q2,(1+delta)*(q0-q2)/(q1*q2),'r*')
47
48      plot(x,left,'c')
49      plot(x,right,'c')
50      plot([0, b2(1)*left(2)], [0, b2(2)*left(2)],'k--')
51      plot([0, b3(1)*left(2)], [0, b3(2)*left(2)],'k--')
52
53      axis([0 max([ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)), ...
            ceil((1+delta)*(q0-q2)/(q1*q2))]) 0 ...
            max([ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)), ...
            ceil((1+delta)*(q0-q2)/(q1*q2))])])
54      grid on
55
56      for y = 0:ceil( (1+delta)*a0/q2 )
57          x = ceil( (q1*y + 1-delta)/q0);
58          %pause
59          plot(x,y,'b.')
60          if x*q0 - y*q1 < 1+delta
61              plot(x,y,'go')
62              k = x*k0 - y*k1;
63              l = x*l0 - y*l1;
64
65              if n<8
66                  subplot(3,3,9)
67                  x_ = 0:k+1:k+1;
68                  mid = x_*(p/q) - 1/q;
69
70                  left = mid - delta/q;
71                  right = mid + delta/q;
72
73                  plot(x_,left,'c')
74                  hold on
75                  plot(x_,right,'c')
76                  plot(k,l,'.b')
77                  plot(k,l,'og')
78
79                  axis([k-delta k+delta l-delta l+delta])
80                  set(gca,'xtick',[0:1:k+1])
81                  set(gca,'ytick',[0:1:l+1])
82                  grid on
83              end
84
85              return
86          end
87      end
88
89      q3 = q1 - a1*q2;
90      if q3 == 0
91          k = "No solution yet";
92          l = "No solution yet";
93          break
94      end
95
96      k3 = k1 - a1*k2;
97      l3 = l1 - a1*l2;
98
99      q0 = q2; q1 = q3;
100     k0 = k2; k1 = k3;
101     l0 = l2; l1 = l3;
102 end
```

A.2. The improved algorithm (x-search)

The code for the improved algorithm.

```
1 clear; close all;
```

```
2
3  tmax = 1;
4  T1 = tmax;
5  T2 = tmax*rand;
6  t1 = T1*rand;
7  t2 = T2*rand;
8  p = T1/abs(t1-t2);
9  q = T2/abs(t1-t2);
10 delta = 10^(-randi([1,10]));
11
12 q0 = p; q1 = q;
13 k0 = 1; k1 = 0;
14 l0 = 0; l1 = -1;
15
16 figure;
17 for n = 0:100
18     a0 = floor(q0/q1);
19     q2 = q0 - a0*q1;
20     if q2 == 0
21         k = "No solution yet";
22         l = "No solution yet";
23         break
24     end
25     k2 = k0 - a0*k1;
26     l2 = l0 - a0*l1;
27     a1 = floor(q1/q2);
28
29
30     x = 0:ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)): ...
           ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2));
31     y = x*(q0/q1) - 1/q1;
32     left = y - delta/q1;
33     right = y + delta/q1;
34
35     A = [k0, -k1;
36          l0, -l1];
37
38     b2 = A\[k2; l2];
39     b3 = -A\[k1 - a1*k2; l1 - a1*l2];
40
41     subplot(3,3,n+1)
42     hold on;
43     plot((1-delta)/q0,0,'r*')
44     plot((1+delta)/q0,0,'r*')
45     plot((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2),(1+delta)*(q0-q2)/(q1*q2),'r*')
46     plot((1+delta)/q2,(1+delta)*(q0-q2)/(q1*q2),'r*')
47
48     plot(x,left,'c')
49     plot(x,right,'c')
50     plot([0, b2(1)*left(2)], [0, b2(2)*left(2)],'k--')
51     plot([0, b3(1)*left(2)], [0, b3(2)*left(2)],'k--')
52
53     axis([0 max([ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)), ...
          ceil((1+delta)*(q0-q2)/(q1*q2))]) 0 ...
          max([ceil((1-delta)/q0+(1+delta)*(q0-q2)/(q0*q2)), ...
          ceil((1+delta)*(q0-q2)/(q1*q2))])])
54     grid on
55
56     for x = ceil((1-delta)/q0):floor((1+delta)/q2)
57         y = max(0,ceil(q0/q1*x-(1+delta)/q1));
58         %pause
59         plot(x,y,'b.')
60         if x*q0 - y*q1 > 1-delta
61             plot(x,y,'go')
62             k = x*k0 - y*k1;
63             l = x*l0 - y*l1;
64
65             if n<8
66                 subplot(3,3,9)
67                 x_ = 0:k+1:k+1;
68                 mid = x_*(p/q) - 1/q;
```

```
69
70                  left = mid - delta/q;
71                  right = mid + delta/q;
72
73                  plot(x_,left,'c')
74                  hold on
75                  plot(x_,right,'c')
76                  plot(k,l,'.b')
77                  plot(k,l,'og')
78
79                  axis([k-delta k+delta l-delta l+delta])
80                  set(gca,'xtick',[0:1:k+1])
81                  set(gca,'ytick',[0:1:l+1])
82                  grid on
83              end
84
85              return
86          end
87      end
88
89      q3 = q1 - a1*q2;
90      if q3 == 0
91          k = "No solution yet";
92          l = "No solution yet";
93          break
94      end
95
96      k3 = k1 - a1*k2;
97      l3 = l1 - a1*l2;
98
99      q0 = q2; q1 = q3;
100     k0 = k2; k1 = k3;
101     l0 = l2; l1 = l3;
102 end
```

A.3. Frequencies of different algorithms

The code for plotting the frequencies:

```
1  clear; close all;
2
3  n = 10^6;
4  arr1 = zeros(n,1);
5  arr2 = zeros(n,1);
6
7
8  for i=1:n
9      %% Initializing
10     tmax = 1;
11     T1 = tmax;
12     T2 = tmax*rand;
13     if T1 < T2
14         T1 = tmax - T1;
15         T2 = tmax - T2;
16     end
17     t1 = T1*rand;
18     t2 = T2*rand;
19     p = T1/abs(t1-t2);
20     q = T2/abs(t1-t2);
21     delta = 1/10^7;
22
23     %% y-search (Visser)
24     q0 = p; q1 = q;
25     k0 = 1; k1 = 0;
26     l0 = 0; l1 = -1;
27
28     brk = 0;
```

```
29      steps = 0;
30
31      for transformations = 0:100
32          a0 = floor(q0/q1);
33          q2 = q0 - a0*q1;
34          if q2 == 0
35              k = "No solution yet";
36              l = "No solution yet";
37              break
38          end
39          k2 = k0 - a0*k1;
40          l2 = l0 - a0*l1;
41          a1 = floor(q1/q2);
42
43          for y=0:floor( (1+delta)*a0/q2 )
44              steps = steps + 1;
45              x = ceil( (q1*y + 1-delta)/q0);
46              if x*q0 - y*q1 < 1+delta
47                  k = x*k0 - y*k1;
48                  l = x*l0 - y*l1;
49                  brk = 1;
50                  arr1(i) = steps;
51                  break
52              end
53          end
54
55          if brk == 1
56              break
57          end
58
59          q3 = q1 - a1*q2;
60          if q3 == 0
61              k = "No solution yet";
62              l = "No solution yet";
63              break
64          end
65
66          k3 = k1 - a1*k2;
67          l3 = l1 - a1*l2;
68
69          q0 = q2; q1 = q3;
70          k0 = k2; k1 = k3;
71          l0 = l2; l1 = l3;
72      end
73
74      %% x-search (Schouten)
75      q0 = p; q1 = q;
76      k0 = 1; k1 = 0;
77      l0 = 0; l1 = -1;
78
79      brk = 0;
80      steps = 0;
81
82      for transformations = 0:100
83          a0 = floor(q0/q1);
84          q2 = q0 - a0*q1;
85          if q2 == 0
86              k = "No solution yet";
87              l = "No solution yet";
88              break
89          end
90          k2 = k0 - a0*k1;
91          l2 = l0 - a0*l1;
92          a1 = floor(q1/q2);
93
94          for x = ceil((1-delta)/q0):floor((1+delta)/q2)
95              steps = steps + 1;
96              y = max(0,ceil(q0/q1*x-(1+delta)/q1));
97              if x*q0 - y*q1 > 1-delta
98                  k = x*k0 - y*k1;
99                  l = x*l0 - y*l1;
```

```
                brk = 1;
                arr2(i) = steps;
                break
            end
        end

        if brk == 1
            break
        end

        q3 = q1 - a1*q2;
        if q3 == 0
            k = "No solution yet";
            l = "No solution yet";
            break
        end

        k3 = k1 - a1*k2;
        l3 = l1 - a1*l2;

        q0 = q2; q1 = q3;
        k0 = k2; k1 = k3;
        l0 = l2; l1 = l3;
    end

end

%% Plotting
subplot(1,2,1)
histogram(arr1,10.^(0:8))
hold on
histogram(arr2,10.^(0:8))
set(gca, "XScale", "log")
xlabel('#steps')
ylabel('frequency')
legend('y-search (Visser)', 'x-search (Schouten)')

subplot(1,2,2)
tbl = tabulate(categorical(arr1));
plot(:,1) = str2double(tbl(:,1));
plot(:,2) = cell2mat(tbl(:,2));
scatter(plot(:,1),(plot(:,2)))
hold on
tbl2 = tabulate(categorical(arr2));
plot2(:,1) = str2double(tbl2(:,1));
plot2(:,2) = cell2mat(tbl2(:,2));
scatter(plot2(:,1),(plot2(:,2)))
xlabel('#steps')
ylabel('frequency')
legend('y-search (Visser)', 'x-search (Schouten)')
set(gca, "XScale", "log")
set(gca, "YScale", "log")
```

A.4. Run-time of all algorithms

The code for plotting the run-time of the brute force methods and the continued fraction methods:

```
clear; close all;

n = 10^6;
steps = ones(n,6);

for i=1:n
    %% Initializing
    tmax = 1;
    T1 = tmax;
    T2 = tmax*rand;
```

```
11      t1 = T1*rand;
12      t2 = T2*rand;
13      p = T1/abs(t1-t2);
14      q = T2/abs(t1-t2);
15      delta = 10^(-randi([1,10]));
16
17      break_loop = 0;
18
19      %% Brute force; y-search
20      n = 0;
21      k_int = round(1/p);
22      l_int = 0;
23
24      while ¬(l_int>k_int*p/q-(1+delta)/q && l_int<k_int*p/q-(1-delta)/q)
25          n = n + 1;
26
27          if l_int > k_int*(p/q) - 1/q
28              k_int = k_int + 1;
29          end
30
31          l_int = l_int + 1;
32
33          if k_int>10^7 || l_int>10^7
34              break_loop = 1;
35              steps(i,1) = "Too many steps. Break to avoid long runtime.";
36              break
37          end
38
39      end
40
41      if break_loop == 1
42          continue
43      end
44
45      steps(i,2) = n;
46
47      %% Brute force; x-search
48      n = 0;
49      k_brute = 0;
50      l_brute = 0;
51
52      while ¬(l_brute < k_brute*p/q- (1-delta)/q)
53          n = n + 1;
54
55          k_brute = k_brute + 1;
56          l_brute = ceil(k_brute*p/q - (1+delta)/q);
57
58      end
59
60      steps(i,3) = n;
61
62      %% Continued Fraction method; y-search (Visser)
63      q0 = p; q1 = q;
64      k0 = 1; k1 = 0;
65      l0 = 0; l1 = -1;
66
67      brk = 0;
68      teller = 0;
69
70      for n = 0:100
71          a0 = floor(q0/q1);
72          q2 = q0 - a0*q1;
73          if q2 == 0
74              k_cont1 = "No solution yet";
75              l_cont1 = "No solution yet";
76              break
77          end
78          k2 = k0 - a0*k1;
79          l2 = l0 - a0*l1;
80          a1 = floor(q1/q2);
81
```

```
82          for y=0:floor( (1+delta)*a0/q2 )
83              x = ceil( (q1*y + 1-delta)/q0);
84              teller = teller + 1;
85              if x*q0 - y*q1 < 1+delta
86                  k_cont = x*k0 - y*k1;
87                  l_cont = x*l0 - y*l1;
88                  brk = 1;
89                  break
90              end
91          end
92
93          if brk == 1
94              break
95          end
96
97          q3 = q1 - a1*q2;
98          if q3 == 0
99              k_cont1 = "No solution yet";
100             l_cont1 = "No solution yet";
101             break
102         end
103
104         k3 = k1 - a1*k2;
105         l3 = l1 - a1*l2;
106
107         q0 = q2; q1 = q3;
108         k0 = k2; k1 = k3;
109         l0 = l2; l1 = l3;
110     end
111
112     steps(i,4) = teller;
113
114     %% Continued Fraction method; x-search (Schouten)
115     q0 = p; q1 = q;
116     k0 = 1; k1 = 0;
117     l0 = 0; l1 = -1;
118
119     brk = 0;
120
121     teller = 0;
122
123     for n = 0:100
124         a0 = floor(q0/q1);
125         q2 = q0 - a0*q1;
126         if q2 == 0
127             k_cont2 = "No solution yet";
128             l_cont2 = "No solution yet";
129             break
130         end
131         k2 = k0 - a0*k1;
132         l2 = l0 - a0*l1;
133         a1 = floor(q1/q2);
134
135         for x = ceil((1-delta)/q0):floor((1+delta)/q2)
136             teller = teller + 1;
137             y = max(0,ceil(q0/q1*x-(1+delta)/q1));
138             if x*q0 - y*q1 > 1-delta
139                 k_cont2 = x*k0 - y*k1;
140                 l_cont2 = x*l0 - y*l1;
141                 brk = 1;
142                 break
143             end
144         end
145
146         if brk == 1
147             break
148         end
149
150         q3 = q1 - a1*q2;
151         if q3 == 0
152             k_cont2 = "No solution yet";
```

```
153             l_cont2 = "No solution yet";
154             break
155         end
156
157         k3 = k1 - a1*k2;
158         l3 = l1 - a1*l2;
159
160         q0 = q2; q1 = q3;
161         k0 = k2; k1 = k3;
162         l0 = l2; l1 = l3;
163     end
164
165     steps(i,5) = teller;
166     steps(i,1) = k_cont2;
167     steps(i,6) = l_cont2;
168
169
170     %% Checking for differences
171     if k_int ~= k_brute || k_int ~= k_cont || k_int ~= k_cont2
172         steps(i,1) = "Machine precision too low. Break to avoid mistakes in plot.";
173         break
174     end
175     if l_int ~= l_brute || l_int ~= l_cont || l_int ~= l_cont2
176         steps(i,1) = "Machine precision too low. Break to avoid mistakes in plot.";
177         break
178     end
179
180 end
181
182 figure
183 subplot(1,2,1)
184 scatter((steps(:,1)),(steps(:,2)),'.')
185 hold on
186 scatter((steps(:,1)),(steps(:,3)),'.')
187 scatter((steps(:,1)),(steps(:,4)),'.')
188 scatter((steps(:,1)),(steps(:,5)),'.')
189 xlabel('k')
190 ylabel('#steps')
191
192 subplot(1,2,2)
193 scatter((steps(:,1)),(steps(:,2)),'.')
194 hold on
195 scatter((steps(:,1)),(steps(:,3)),'.')
196 scatter((steps(:,1)),(steps(:,4)),'.')
197 scatter((steps(:,1)),(steps(:,5)),'.')
198 legend("Brute force; y-search","Brute force; x-search","Continued Fraction method; ...
        y-search (Visser)","Continued Fraction method; x-search ...
        (Schouten)",'Orientation','horizontal')
199 xlabel('k')
200 ylabel('#steps')
201 set(gca, "XScale", "log")
202 set(gca, "YScale", "log")
```

A.5. Number of steps and basis transformations

The code for plotting the number of steps and the number of basis transformations for the improved algorithm:

```
1 clear; close all;
2
3 n = 10^4;
4 arr = ones(n,4);
5
6 for i=1:n
7     %% Initializing
8     tmax = 1;
9     T1 = tmax;
```

```
10      T2 = tmax*rand;
11      t1 = T1*rand;
12      t2 = T2*rand;
13      p = T1/abs(t1-t2);
14      q = T2/abs(t1-t2);
15      delta = 10^(-randi([1,10]));
16
17      %% Continued Fraction method; x-search (Schouten)
18      q0 = p; q1 = q;
19      k0 = 1; k1 = 0;
20      l0 = 0; l1 = -1;
21
22      brk = 0;
23
24      steps = 0;
25
26      for transformations = 0:100
27          a0 = floor(q0/q1);
28          q2 = q0 - a0*q1;
29          if q2 == 0
30              break
31          end
32          k2 = k0 - a0*k1;
33          l2 = l0 - a0*l1;
34          a1 = floor(q1/q2);
35
36          for x = ceil((1-delta)/q0):floor((1+delta)/q2)
37              steps = steps + 1;
38              y = max(0,ceil(q0/q1*x-(1+delta)/q1));
39              if x*q0 - y*q1 > 1-delta
40                  k = x*k0 - y*k1;
41                  l = x*l0 - y*l1;
42                  brk = 1;
43                  break
44              end
45          end
46
47          if brk == 1
48              break
49          end
50
51          q3 = q1 - a1*q2;
52          if q3 == 0
53              break
54          end
55
56          k3 = k1 - a1*k2;
57          l3 = l1 - a1*l2;
58
59          q0 = q2; q1 = q3;
60          k0 = k2; k1 = k3;
61          l0 = l2; l1 = l3;
62      end
63
64      arr(i,1) = k;
65      arr(i,2) = steps;
66      arr(i,3) = transformations+1;
67      arr(i,4) = steps/(transformations+1);
68
69
70  end
71
72  ka=10.^(0:12);
73  logka=log10(ka);
74  wortelka=ka.^0.5;
75
76
77  scatter((arr(:,1)),(arr(:,4)),'.')
78  hold on
79  scatter((arr(:,1)),(arr(:,2)),'.')
80  scatter((arr(:,1)),(arr(:,3)),'.')
```

```
plot((ka),(ka))
plot((ka),(wortelka))
plot((ka),(logka))
legend("avg #steps/basis","#steps","#bases"," y = x"," y = \surd{x}"," y = log(x)")
xlabel('k')
ylabel('Number of steps, bases and steps/basis')
set(gca, "XScale", "log")
set(gca, "YScale", "log")
axis([10^0 10^13 10^-1 10^9])
```

A.6. The probability density function

The code for plotting the simulated probability density and the probability density function given by Visser:

```
clear; close all;

n = 10^7;
arr = zeros(1,n);

tmax = 1;
T1 = tmax;
T2 = tmax*rand;
t1 = T1*rand;
t2 = T2*rand;
p_ = T1/abs(t1-t2);
q_ = T2/abs(t1-t2);
delta_ = 10^(-randi([1,10]));
avg=q_/2/delta_;

delta_sav = zeros(1,n);

for i=1:n
    %% Initializing
    p = p_*(1+10^(-4)*(-1+2*rand));
    q = q_*(1+10^(-4)*(-1+2*rand));
    delta = delta_*(1+10^(-4)*(-1+2*rand));

    %% x-search (Schouten)
    q0 = p; q1 = q;
    k0 = 1; k1 = 0;
    l0 = 0; l1 = -1;

    brk = 0;

    for transformations = 0:100
        a0 = floor(q0/q1);
        q2 = q0 - a0*q1;
        if q2 == 0
            k = "No solution yet";
            l = "No solution yet";
            break
        end
        k2 = k0 - a0*k1;
        l2 = l0 - a0*l1;
        a1 = floor(q1/q2);


        for x = ceil((1-delta)/q0):floor((1+delta)/q2)
            y = max(0,ceil(q0/q1*x-(1+delta)/q1));
            if x*q0 - y*q1 > 1-delta
                k = x*k0 - y*k1;
                l = x*l0 - y*l1;
                arr(i) = k;
                brk = 1;
                break
            end
```

```
53          end
54
55          if brk == 1
56              break
57          end
58
59          q3 = q1 - a1*q2;
60          if q3 == 0
61              k = "No solution yet";
62              l = "No solution yet";
63              break
64          end
65
66          k3 = k1 - a1*k2;
67          l3 = l1 - a1*l2;
68
69          q0 = q2; q1 = q3;
70          k0 = k2; k1 = k3;
71          l0 = l2; l1 = l3;
72      end
73
74  end
75
76  %% Plotting
77  histogram(arr,0:10*avg/100:10*avg,'Normalization','pdf')
78  hold on
79  k = 0:10*avg/100:10*avg;
80  P = exp(-k/avg)/avg;
81  P_disc = exp(-k/avg)*(1-exp(-1/avg));
82  plot(k,P,'r-')
83  plot(avg,0:0.05/avg:1/avg,'r.')
84  %set(gca, "YScale", "log")
85  xlabel('k')
86  ylabel('probability')
87  legend('Normalized simulated data','Probability density function (Visser)','Expected ...
        value')
```