Integer Factorization

Bachelor Thesis

Martin M. Lauridsen

Supervisors: Lars Ramkilde Knudsen, Department of Mathematics, DTU
Søren Steffen Thomsen, Department of Mathematics, DTU

Submission Date: June 25, 2010
Abstract

Cryptographic algorithms used in real world applications, such as the RSA public-key encryption scheme, rely on the difficulty of the integer factorization problem. We discuss two algorithms for factoring integers. These are Pollard’s Rho method, which is a randomized algorithm, and the Quadratic Sieve, which is based on the random equal squares method by Dixon. We present implementations of Pollard’s Rho method and four algorithm variants in the Quadratic Sieve family, one of which is a distributed implementation of the Multiple Polynomial variant. These implementations are tested on composites of different sizes. Our distributed implementation of the Multiple Polynomial Quadratic Sieve factors 60-digit composites in about 20 minutes, using 16 factoring clients, giving us approximately 9 times speedup compared to the same algorithm, which is not parallelized in any way. For composites up to 20 digits, our implementation of Pollard’s Rho is the fastest.
Preface

This thesis has been submitted in partial fulfilment of the requirements to complete the Bachelor of Science (B.Sc.) degree at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark, DTU.

The work has been completed at the Department of Mathematics (MAT) at DTU, under supervision of Professor Lars Ramkilde Knudsen, MAT, and Assistant Professor Søren Steffen Thomsen, MAT.

February 21, 2013

Martin M. Lauridsen
Notation

Here, we define notation that will be used throughout the thesis.

- Vectors will be written in the form \( v = (v_1, v_2, \ldots, v_n) \).
- For a matrix \( A \), \( A^T \) means the transpose of \( A \).
- For a set \( S \), \( |S| \) means the cardinality of \( S \), i.e. the number of elements in \( S \).
- \( \gcd(x, y) \) is the greatest common divisor of \( x \) and \( y \).
- \( a \mid b \) means "a divides b", i.e. \( b = ac \) for some \( c \).
- Vectors and matrices will be given by the context. We use normal math font for these. An exception is the zero vector, which will be written in bold: \( \mathbf{0} \).
- \( \lfloor a \rfloor \) is the floor function. It returns the largest integer less than or equal to \( a \).
- \( \lceil a \rceil \) is the rounding function.
- We use \( \log \) for the base 2 logarithm and \( \ln \) for the natural logarithm.
# Contents

Abstract .................................................. i  
Preface .................................................... iii  
Notation ................................................... v  
Contents .................................................. vi  
List of Algorithms ........................................ ix  
List of Figures ........................................... x  
List of Tables ............................................. xi  

1 Introduction .............................................. 1  

2 Mathematical background ................................ 3  
  2.1 The birthday problem .................................. 3  
  2.2 Infinite periodic sequences ......................... 6  
  2.3 Floyd's cycle-finding algorithm ................... 8  
  2.4 Linear dependencies and matrix kernel ........... 10  
  2.5 Hensel's lifting lemma ............................... 13  

3 Pollard's Rho method ..................................... 15  
  3.1 Choice of pseudo-random function ................. 16  
  3.2 Finding cycles ...................................... 17  
  3.3 Complexity .......................................... 18
CONTENTS

3.4 Summary ................................................. 19

4 Dixon’s Random Squares ................................. 21

5 Quadratic Sieve ............................................ 25
  5.1 Motivation ............................................. 25
  5.2 The algorithm ....................................... 26
  5.3 Sieving ............................................... 31
  5.4 Parameter tuning .................................... 34
  5.5 Complexity .......................................... 39
  5.6 Summary ............................................. 41

6 The Multiple Polynomial Quadratic Sieve ............ 43
  6.1 Motivation ............................................ 44
  6.2 Multiple polynomials ............................... 44
  6.3 Choice of coefficients .............................. 45
  6.4 Choice of sieve length parameter \( M \) ............. 48
  6.5 Computing polynomial roots ....................... 48
  6.6 Choice of sieving candidate bound ............... 50
  6.7 Parallelization ...................................... 50
  6.8 Summary ............................................. 51

7 Implementation .......................................... 53
  7.1 Technicalities ....................................... 53
  7.2 Pollard’s Rho method .............................. 54
  7.3 The Quadratic Sieve ................................ 55
  7.4 Sieving implementation ............................ 58
  7.5 Multiple polynomials .............................. 59
  7.6 Distributed MPQS .................................. 60
  7.7 Optimizations ....................................... 65

8 Computational results .................................. 67
  8.1 Pollard’s Rho ........................................ 68
  8.2 Quadratic Sieve and variations .................. 69

9 Conclusion ............................................... 73

A Pseudo code for factoring by trial division .......... 75

B Proof that \((\frac{n}{z}) = 1 \Rightarrow x^2 \equiv n \pmod{z}\) has two solutions .... 77
CONTENTS

C  Maple code for generating composites 79
D  Pollard’s Rho method source code 81
E  Quadratic Sieve without sieving source code 83
F  Quadratic Sieve with sieving source code 85
G  Multiple Polynomial Quadratic Sieve source code 87
H  Distributed Multiple Polynomial Quadratic Sieve source code 89

Bibliography 91
List of Algorithms

1. Pollard’s Rho factoring method. 19
2. Quadratic Sieve algorithm. 29
3. Factoring by trial division. 75
List of Figures

2.1 Infinite sequence graph indicating $\rho$-shape. .......................... 7
2.2 Example showing Floyd’s cycle-finding algorithm. ...................... 9
2.3 Example of infinite periodic sequence. .................................. 9

5.1 Evaluation of $\Psi(x, B)/x$ for $x \in [1, 100000]$ and $B \in [2, 2000]$. . . 26
5.2 Comparison of factorizations by trial divisions needed for 100 com-
postes. ................................................................. 33
5.3 Smooth $Q(x)$ with sieve values and candidate bounds for $n = 1000076001443$, $M = 100000$ and $|S| = 88$ (smoothness $B = 1000$). 35
5.4 Factor base sizes for the Landquist bound and $S_E(n)$. ............. 38

6.1 Plot of $q(x)$ with $n = 1000076001443$, $M = 100000$, $a = 14 \approx 
\sqrt{2n}/M$ and $b = 7$. .............................................. 45

7.1 Flow diagram showing the QS algorithm. ............................... 57
7.2 Distribution of polynomials to client programs. ...................... 61
7.3 Distributed sieving and sending of smooth relations. ............... 62

8.1 Steps required for Pollard’s Rho method for various composites $n$. 68
8.2 Factorization time for multiple polynomial variations. ............. 72
List of Tables

5.1  Precision of the Prime Number Theorem for varying upper bounds $x$. 37

6.1  MPQS parameters proposed by Silverman[Sil87] (K means 1000). 48

8.1  Average factorization time for Pollard’s Rho method. . . . . . . . 68
8.2  Average factorization time in seconds for Quadratic Sieve family
     algorithms, using the Landquist bound. . . . . . . . . . . . . . . . 70
8.3  Average factorization time in seconds for Quadratic Sieve family
     algorithms, using the Contini bound. . . . . . . . . . . . . . . . . . 71
It has been well known for a long time, that any whole number larger than 1, can be written as the unique product of prime numbers. Euclid was close to proving this in his series of books *Euclid’s Elements* ca. 300 BC, and the first full proof was given in *Disquisitiones Arithmeticae* by Carl Friedrich Gauss in 1798. Given some composite number, the problem of finding its unique prime factorization is known as the integer factorization problem.

The integer factorization problem is generally considered hard, when the composite is very large. However, the composite can be of a special form, e.g. $2^{10000}$, that makes it easy to factor. Several cryptographic algorithms, e.g. RSA public-key encryption and the RSA digital signature scheme, use very large composite numbers for their public keys, and the prime factors for secret keys. Hence, the security of these algorithms rely on the integer factorization problem being hard. Thus, researching integer factorization is very important, because if we found a way to quickly factor even the hardest composites, the schemes would no longer be safe, and we would have to abandon these methods.

In general, the integer factorization problem is finding the unique prime factorization of some composite integer. For this thesis, we will focus on composite integers that are the product of two distinct, odd primes, of about the same size. Throughout the thesis, we will denote the composite number $n$, and the prime factors of about the same size, $p$ and $q$, such that

$$n = p \cdot q.$$  \hspace{1cm} (1.1)

During the past 40 years, the availability of computers has had a major growth. Fast computers are purchasable to anyone at a reasonable price. Mathematicians view computers as a tool, which may aid them in arriving at results much faster than what has been previously possible.
CHAPTER 1. INTRODUCTION

The great accessibility of computers has changed the way mathematicians look at a problem. For instance, when discussing an algorithm, they talk about computational complexity, parallelization and memory consumption. In other words, the way mathematicians work has been affected by the introduction of computers as a tool in their research.

In the past three decades, several interesting algorithms for integer factorization have been proposed. The increasing power of computers has made it feasible to factor large composites. The most recent record for a composite of the form (1.1), is the factorization of RSA-768\(^1\), a 768 bit number with 232 digits in December 2009. This was done using the General Number Field Sieve, an algorithm beyond the scope of this thesis.

Many factoring algorithms are fast for composite numbers, that are not of the type stated in (1.1), i.e. have either small prime factors, or are of some other special form. These algorithms utilize some kind of structure of the composite in attempting to factor it. These types of algorithms are called special-purpose algorithms. In this thesis we shall look at one special-purpose algorithm, Pollard’s Rho method. Algorithms that do not rely on the composite being of some kind of special structure, are called general-purpose algorithms. We shall cover the Quadratic Sieve and Multiple Polynomial Quadratic Sieve algorithms, which are general-purpose algorithms.

\(^1\text{RSA-768 = 123018668453011775513049495838496272077285356959533479219}\)
\[73224521517264005072636575187452021997864693899564749427740638405925192\]
\[557326303453731548268507917026122124291346167042921431160222124047927473\]
\[7794080665351419597459856902143413.\]
Mathematical background

In this chapter, we present theory on a range of mathematical topics, which will find use in description of the algorithms in later chapters. We start off by describing The birthday problem.

2.1 The birthday problem

This problem, also known as the Birthday 'paradox', is in fact not a paradox. It got its name, because it shows a result which is surprising to many. The problem can be formulated in two ways:

- How many people must be in a room, for the chance of two people having the same birthday being 50%?
- How many people must be in a room, for the expected number of pairs of people having the same birthday, is 1?

The surprising answer to the first question is, that among just 23 people, the chance that two people share birthdays is greater than 50%.

For our analysis, we will focus on the last formulation above. We will need to do some assumptions. First, we assume that there are 365 days in a year, that is, we ignore leap years which have one extra day. Also, we assume that all days of the year are equally likely to be someone’s birthday. Statistically, this is not true in reality. We also assume no presence of twins, triplets, etc. sharing birthday, to keep the sample unbiased. Lastly, we assume that a random selection of people’s birthdays is independent.

In the following, we let $m$ be the sample space size. For the birthday paradox, $m = 365$. The number of people in the room will be denoted $k$. 
CHAPTER 2. MATHEMATICAL BACKGROUND

Let \( b_i \in \{1, 2, \ldots, m\} \) be the birthday of some person \( i \). Since selection of birthdays is independent, the probability that two people, \( i \) and \( j \), both are born on some day \( t \) is

\[
Pr\{b_i = t \land b_j = t\} = Pr\{b_i = t\} \cdot Pr\{b_j = t\} = \left(\frac{1}{m}\right)^2 = \frac{1}{m^2}.
\]

(2.1)

Thus, the probability of person \( i \) and \( j \) sharing birthday on any day in \( \{1, 2, \ldots, m\} \) is

\[
Pr\{b_i = b_j\} = \sum_{t=1}^{m} Pr\{b_i = t \land b_j = t\} = \sum_{t=1}^{m} \frac{1}{m^2} = m \cdot \frac{1}{m^2} = \frac{1}{m}.
\]

(2.2)

This makes good sense intuitively, since, when some day \( b_i \) is chosen, the chance of \( b_j \) being chosen the same as \( b_i \) is \( \frac{1}{m} \).

From probability theory, we will need the notion of expected value of a discrete random variable \( X \), denoted \( E[X] \). It is defined as

\[
E[X] = \sum_x (x \cdot Pr\{X = x\}) , \text{ see [CLRS01].}
\]

(2.3)

The expectation obeys the following property, called linearity of expectations. If \( X \) and \( Y \) are two random variables, then

\[
E[X + Y] = E[X] + E[Y] , \text{ see [CLRS01].}
\]

(2.4)

To determine how many people must be in a room, for exactly one pair having the same birthday, we will need the definition of indicator variables.

**Definition 1.** We define an indicator variable to be a variable which is 1 for some event, and 0 for another. More specifically, for the purpose of showing the birthday paradox, we let

\[
X_{ij} = \begin{cases} 
1 & \text{, when person } i \text{ and } j \text{ share birthday} \\
0 & \text{, otherwise}
\end{cases}
\]
2.1. THE BIRTHDAY PROBLEM

Using Definition 1 and our knowledge from (2.2), (2.3) and (2.4), we can write

\[ E[X_{ij}] = 1 \cdot \frac{1}{m} + 0 \cdot \frac{m - 1}{m} = \frac{1}{m} \quad \text{(2.5)} \]

If we let \( X \) be a random variable which counts the number of \((i, j)\)-pairs that have the same birthday, where \(1 \leq i, j \leq m\) and \(i \neq j\), we get

\[ X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}, \quad \text{(2.6)} \]

since the expression

\[ \sum_{i=1}^{k} \sum_{j=i+1}^{k} (i, j) \]

iterates through exactly all possible \((i, j)\)-pairs.

What we actually want to know is, what \(k\) and \(m\) need to be, for \(X\) to have an expected value of 1. Thus, taking the expectation \(E\) on both sides, we get

\[ E[X] = E \left[ \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} \right]. \quad \text{(2.7)} \]

By applying the linearity of expectations from (2.4), we get

\[ E \left[ \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} \right] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{1}{m}. \quad \text{(2.8)} \]

As mentioned, the two summations iterate as many times as all possible \((i, j)\)-pairs. This is the same value as the binomial coefficient that we know from elementary probability theory, defined as the number of ways we can pick \(b\) elements out of \(a\) possible:

\[ \binom{a}{b} = \frac{a!}{b!(a-b)!}, \quad \text{see [CLRS01].} \quad \text{(2.9)} \]
Thus, the two summations iterate \( \binom{k}{2} \) times. Hence, our expression becomes

\[
E[X] = \binom{k}{2} \frac{1}{m} \\
= \frac{k!}{2!(k-2)!} \cdot \frac{1}{m} \\
= \frac{k!}{(k-2)!} \cdot \frac{1}{2!} \cdot \frac{1}{m} \\
= k(k-1) \cdot \frac{1}{2} \cdot \frac{1}{m} \\
= \frac{k(k-1)}{2m}.
\] (2.10)

Thus, when \( k(k-1) = 2m \), the expected number of \((i,j)\)-pairs of people with the same birthday equals one. Solving for \( k \), we get

\[
k(k-1) = 2m \\
\iff k^2 - k + \left(\frac{1}{2}\right)^2 = 2m + \left(\frac{1}{2}\right)^2 \\
\iff \left(k - \frac{1}{2}\right)^2 = 2m + \frac{1}{4} \\
\iff \left(k - \frac{1}{2}\right)^2 = \frac{8m + 1}{4} \\
\iff \left(k - \frac{1}{2}\right) = \sqrt{\frac{8m + 1}{4}} \\
\iff k - \frac{1}{2} = \sqrt{\frac{8m + 1}{4}} \\
\iff k = \sqrt{\frac{8m + 1}{4}} + \frac{1}{2}.
\] (2.11)

Thus, if we set \( m = 365 \) and \( k = 28 \), we see by (2.10), that the expected number of pairs having the same birthday is \( \frac{28 \cdot 27}{2 \cdot 365} \approx 1.0356 \). Generally, if we have a randomly distributed sample space of size \( m \), we expect to choose \( k \approx \sqrt{\frac{8m+1}{2}} \) values before we get a value we had before.

### 2.2 Infinite periodic sequences

Let \( S = \{0, 1, 2, \dots, m-1\} \), and let \( f : S \to S \) be a random mapping, which maps an element from \( S \) to an element in \( S \). We note that the cardinality of
2.2. INFINITE PERIODIC SEQUENCES

$S$ is $m$, i.e. $|S| = m$. Let $x_0 \in S$. We define $\langle x \rangle = x_0, x_1, \ldots$ as the infinite sequence, where $x_i = f(x_{i-1})$ for $i \geq 1$.

Since $S$ is a finite set, the sequence $\langle x \rangle$ will eventually start repeating itself, i.e. we will get $x_i = x_j$ for $i \neq j$. Thus, we say that $\langle x \rangle$ is an infinite periodic sequence. When such values $i$ and $j$ are found, a cycle or collision is said to have been found. Several algorithms for finding cycles exist, for example due to Brent\cite{Bre80}, Sedgewick et al.\cite{SSY82} and Nivasch\cite{Niv04}. One algorithm, due to Floyd\cite{Flo67}, shall be described in Section 2.3.

If one draws the functional graph for an infinite periodic sequence such as $\langle x \rangle$, the shape of this graph will resemble the Greek letter $\rho$ as shown in Figure 2.1. This is what gave the name to the algorithm Pollard’s Rho which we will describe in Chapter 3.

**Definition 2 (Tail length).** The tail length, denoted $\lambda$, of an infinite periodic sequence $\langle x \rangle = x_0, x_1, \ldots$, is the number of edges in the path from the starting vertex to the first vertex, which is in a cycle, in the functional graph of the sequence. If $x_i$ is the first value to be repeated in the sequence, the tail length is $i$.

**Definition 3 (Cycle length).** The cycle length, denoted $\mu$, of an infinite periodic sequence $\langle x \rangle = x_0, x_1, \ldots$, is the number of edges in the cycle of the
functional graph of the sequence. If \( x_i = x_j \) and \( i \neq j \), the cycle length divides \( |j - i| \).

**Example 1.** The graph of Figure 2.1 has tail length \( i \) and cycle length 13.

Using our analysis of the birthday paradox of Section 2.1, we might give a heuristic about the expected length of an infinite periodic sequence, up to the point where it starts repeating itself. Since there are \( m \) elements in \( S \), we suspect that we will have a length of about \( \sqrt{\frac{\sqrt{m^2 + 1}}{2}} + \frac{1}{2} \), asymptotically \( \Theta(\sqrt{m}) \). Relating this to Figure 2.1, we would expect \( \Theta(\sqrt{m}) \) edges in the functional graph for the sequence.

### 2.3 Floyd’s cycle-finding algorithm

This algorithm, attributed to Floyd [Flo67], is an algorithm for finding cycles in infinite periodic sequences as those described in Section 2.2. The algorithm is also known as the *tortoise and hare algorithm* due to its nature.

One of the main benefits of Floyd’s algorithm is, that it uses a constant amount of memory locations. It uses two pointers that point to different indices of a sequence at any given time. At each iteration, no knowledge is needed about the sequence values at the previous indices, which is why it uses constant memory. This is a very appealing feature of the algorithm.

The algorithm starts out with the *tortoise* pointer, pointing at \( x_0 \) and the *hare* pointer pointing at \( x_1 \). In each iteration, the tortoise moves one index forward, and the hare moves two indices forward. Thus, if the tortoise is pointing to \( x_i \), the hare will be pointing to \( x_{2i} \). The algorithm will keep doing this, until \( x_r = x_{2r} \) for some \( r \). When this happens, a cycle has been found.

**Example 2.** Using the infinite periodic sequence defined by \( f(x) = x^2 + 1 \) (mod 75) with a start value \( x_0 = 3 \), we get \( \langle x \rangle = 3, 10, 26, 2, 5, 26, 2, 5, \ldots \). Figure 2.2 illustrates how Floyd’s cycle-finding algorithm proceeds on this example. At iteration 1, the tortoise pointer starts at \( x_0 = 3 \) and the hare pointer starts at \( x_1 = 10 \). At iteration 2, the tortoise points to \( x_1 = 10 \) and the hare to \( x_3 = 2 \). At iteration 3, the tortoise points to \( x_2 = 26 \) and the hare to \( x_5 = 26 \), and the cycle is detected, since \( x_2 = x_5 \), which are the tortoise’s and hare’s current positions, respectively.

Let us try to analyze when the tortoise and hare first get to positions having the same value. That is, we wish to find the smallest \( m \), s.t. \( x_m = x_{2m} \). Since the tortoise visits each \( x_i, i \geq 0 \), until a cycle is found, the smallest \( m \) must be found and the cycle will be detected here.
2.3. FLOYD’S CYCLE-FINDING ALGORITHM

The most important realization in the algorithm is, that the tortoise and hare will be at \( x_i = x_{2i} \) whenever \( i \) is a multiple of the cycle length \( \mu \), as long as \( i \geq \lambda \).

This follows from the fact, that whenever the difference in indices of their positions is a multiple of \( \mu \), they must necessarily be standing at the same value. Note that this only holds, as long as the tortoise has reached at least the first cycle.

Looking at Figure 2.3, the first cycle is marked a shaded background. Here the cycle length is \( \mu = 5 \). Had the tortoise been standing on the first value of 13, i.e. \( x_4 = 13 \), the hare would also be on value 13 as long as the difference between them is a multiple of 5, i.e. \( x_9, x_{14}, \) etc. Since the difference between them is always \( i \), the above must hold.

Naturally, the tortoise and hare can not actually find the cycle until the tortoise reaches it, thus the tortoise must at least be at position \( x_\lambda \). This leads us to the conclusion that the smallest possible \( m \) must obey

\[
\lambda \leq m
\]

The first time \( m \) is a multiple of the cycle length \( \mu \) is when \( m = \mu \). Unfortunately, we can not be sure that the tortoise has reached the beginning of the cycle yet. Specifically, this is the case when \( \lambda > \mu \). Had the tail length \( \lambda \) always been less than or equal to \( \mu \), they would always meet when \( m = \mu \).

This means, that the tortoise and hare will meet the first time a multiple
of $\mu$ is greater than or equal to $\lambda$. Thus, we find that
\[ m = \mu \left( 1 + \left\lfloor \frac{\lambda}{\mu} \right\rfloor \right). \] (2.12)

The last $x_i$ to occur in a sequence, before the sequence starts cycling, is the element $x_{\lambda+\mu-1}$. In the case where $\lambda \equiv 0 \pmod{\mu}$, the first multiple $m$ of $\mu$ which is greater than $\lambda$ occurs when $m = \lambda + \mu - 1$. Had $\lambda$ had some none-zero residue modulo $\mu$, the first multiple $m$ of $\mu$ which is greater than $\lambda$, would have occurred earlier in the cycle. Thus, we conclude that
\[ \lambda \leq m < \lambda + \mu. \]

### 2.4 Linear dependencies and matrix kernel

In this section we discuss dependencies between linear equations and relate these to matrix kernels. Algorithms we describe in later chapters will rely heavily on this.

**Definition 4.** For a set of vectors $v_1, v_2, \ldots, v_m$, we say these vectors are linearly dependent if and only if there exists a set of coefficients $(a_1, a_2, \ldots, a_m) \neq (0, 0, \ldots, 0)$, such that
\[ a_1v_1 + a_2v_2 + \cdots + a_mv_m = 0. \]

If the only solution is $(a_1, a_2, \ldots, a_m) = (0, 0, \ldots, 0)$, we say that $v_1, v_2, \ldots, v_m$ are linearly independent.

A consequence of Definition 4 is, that a set of vectors $v_1, v_2, \ldots, v_m$, are linearly dependent if and only if one of the $v_i$ can be written as a linear combination of the others.

**Definition 5 (Matrix rank).** For some matrix $A$, the rank of $A$ denoted $\text{rank}(A)$, is the maximum number of linearly independent vectors in either the rows or the columns of $A$. If $A$ is an $m \times k$ matrix, then $\text{rank}(A) \leq \min(m, k)$.

From Definition 5, we may conclude, that if $A$ has $k$ columns, we can have at most $k$ linearly independent row vectors in $A$. If we collect $k + 1$ vectors in the rows of $A$, we are guaranteed that they are linearly dependent. This is shown by Example 3.
Example 3. A $4 \times 3$ matrix $A$ is given below. The first three row vectors of $A$ are linearly independent, but all the row vectors together are linearly dependent.

\[
A = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 1 & 0
\end{pmatrix}
\]

Let us consider some $m \times k$ matrix $A$ as the left hand side of some system of linear equations. We might set all these equations equal 0. Say, for example, we have equations of the form

\[
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k = 0 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2k}x_k = 0 \\
\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mk}x_k = 0.
\]  

(2.13)

Putting the coefficients $a_{ij}$ into the appropriate places in $A$, we could write (2.13) as

\[
Ax = 0.
\]  

(2.14)

Definition 6 (Kernel). Using our matrix $A$ as defined above, the kernel denoted $\ker(A)$, also called the null space of $A$, is the set of vector solutions of the form $(x_1, x_2, \ldots, x_k)$ to (2.14). Note that for any matrix $A$, $\ker(A) \neq \emptyset$, since $(0, 0, \ldots, 0)$ is always a solution.

From Definition 6, we note that if we let $w_1, w_2, \ldots, w_k$ be the column vectors of $A$, and $(x_1, x_2, \ldots, x_k)$ is some vector in $\ker(A)$, we find that

\[
x_1w_1 + x_2w_2 + \cdots x_kw_k = 0.
\]

Thus, by solving $\ker(A)$, we find a linear combination of the column vectors of $A$ that gives the zero vector. Note, that if instead wanted to find a linear combination of the row vectors $v_1, v_2, \ldots, v_m$ of $A$, such that the result is the zero vector, we could simply transpose $A$ before computing $\ker(A)$, as shown by the following.
If we transpose $A$ and multiply by $x = (x_1, x_2, \ldots, x_m)$, the equation becomes $A^T x = 0$, and the system of linear equations of (2.13) becomes

\[
\begin{align*}
a_{11}x_1 + a_{21}x_2 + \cdots + a_{m1}x_m &= 0 \\
a_{12}x_1 + a_{22}x_2 + \cdots + a_{m2}x_m &= 0 \\
&\vdots \\
a_{1k}x_1 + a_{2k}x_2 + \cdots + a_{mk}x_m &= 0.
\end{align*}
\tag{2.15}
\]

Thus, if we solve $\ker(A^T)$, we will find solution vectors $x = (x_1, x_2, \ldots, x_m)$ such that using the row vectors $v_1, v_2, \ldots, v_m$ of $A$, we find a linear combination such that

\[x_1v_1 + x_2v_2 + \cdots x_mv_m = 0.\]

The vectors in $\ker(A)$ have the properties

\[
\begin{align*}
x \in \ker(A) &\Rightarrow cx \in \ker(A) & \text{for some constant } c \\
x, y \in \ker(A) &\Rightarrow (x + y) \in \ker(A)
\end{align*}
\tag{2.16, 2.17}
\]

**Definition 7.** Let $V$ be a possibly infinite set of vectors, and let $\{a_1, a_2, \ldots, a_k\}$ be a set of linearly independent vectors. If any vector $x = (x_1, x_2, \ldots, x_k) \in V$ can be written as a linear combination of the vectors in $\{a_1, a_2, \ldots, a_k\}$, we say that $\text{span}\{a_1, a_2, \ldots, a_k\} = V$. Furthermore, we call $\{a_1, a_2, \ldots, a_k\}$ a basis for $V$.

**Definition 8.** The nullity of a matrix $A$, denoted $\text{nullity}(A)$, is the number of vectors defining the basis of $\ker(A)$.

A result from linear algebra, known as the Rank-Nullity theorem states that for an $m \times k$ matrix $A$,

\[k = \text{rank}(A) + \text{nullity}(A). \tag{2.18}\]

Using our definition of rank from Definition 5, we note that if $A$ is an $m \times k$ matrix, with $k > m$, then $\text{rank}(A) \leq m$. Furthermore, from (2.18), we see that $\text{nullity}(A) \geq k - m$.

Note, that if the matrix $A$ is over $\mathbb{Z}_2$, which will be the case in algorithms we describe in later chapters. We will show this by the following informal argument.

Note, that multiplying a vector in the basis of $\ker(A)$ by a scalar, as of (2.16), will either give the vector itself, in the case where the scalar is odd, or
give the zero vector in the case where the scalar is even. Thus, the possible results are finite.

The second property (2.17) however, is more interesting. Assume the basis of \( \ker(A) \) is \( \{x_1, x_2, \ldots, x_j\} \), then we can write a linear combination of these as

\[
c_1 x_1 + c_2 x_2 + \cdots + c_j x_j, \quad c_i \in \mathbb{Z}_2.
\]

(2.19)

The reason we restrict \( c_i \in \mathbb{Z}_2 \) is by the argument above, that going beyond \( \mathbb{Z}_2 \), will not give any new vectors. Thus, (2.19) has a finite number of results.

In other words, we take a linear combination where, for each vector, we decide if we will use it or not. Naturally, the resulting vector will also be over \( \mathbb{Z}_2 \), and will be in \( \ker(A) \). This means, if we have \( j \) vectors in the basis of \( \ker(A) \), then there are \( 2^j \) different vectors in \( \ker(A) \), including the zero vector.

### 2.5 Hensel’s lifting lemma

Hensel’s lifting lemma is a result by Kurt Hensel, which can be used iteratively for finding modular square roots, when the modulus is a prime power of some prime \( z \). Suppose \( f(x) \) is a polynomial with integer coefficients, and \( z \) is a prime. The lemma assumes \( f(x) \equiv 0 \pmod{z} \) has a solution.

Let \( r \) be a root of \( f(x) \pmod{z^{k-1}} \), i.e. \( f(r) \equiv 0 \pmod{z} \), for some \( k \geq 2 \). If \( f'(r) \not\equiv 0 \pmod{z} \), then \( \exists t \in \mathbb{Z}_z \), satisfying

\[
f(r + tz^{k-1}) \equiv 0 \pmod{z^k},
\]

with \( t \) given by

\[
t f'(r) \equiv -(f(r)/z^{k-1}) \pmod{z}.
\]

We will make use of this lemma in the description of the Multiple Polynomial Quadratic Sieve of Chapter 6, to compute modular square roots, where the modulus is a prime power. For more information on Hensel’s lemma, see [Mcg01].
Chapter 3

Pollard’s Rho method

The Pollard’s Rho method for integer factorization, is an algorithm in the family of special-purpose factorization algorithms. The method was first published by John M. Pollard in 1975 [Pol75]. The method is a heuristic, which means that neither the success of the method, nor its algorithmic complexity, which we will analyze later, is guaranteed.

The purpose of the method is to find collisions in a random infinite periodic sequence related to the structure of the composite $n$ which is to be factored. Specifically, the algorithm will try to find a collision in $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$, i.e. two elements $x, x' \in \mathbb{Z}_n$, such that $x \not\equiv x' \left(\text{mod } n\right)$ and $x \equiv x' \left(\text{mod } p\right)$. If two such elements are found, the following reduction shows that we obtain a non-trivial factor of $n$. Assume $x \not\equiv x' \left(\text{mod } n\right)$:

\[
\begin{align*}
x \equiv x' \pmod{p} \\
\iff x = x' + kp, \quad k \geq 1 \\
\iff x' = x - kp.
\end{align*}
\]

(3.1)

Thus,

\[
gcd(x - x', n) = gcd(x - (x - kp), n) \\
= gcd(kp, n).
\]

(3.2)

Hence we have two elements $x$ and $x'$ such that $p \leq gcd(x - x', n) < n$. The remaining question is how to find such elements in an elegant way. Pollard’s Rho uses a pseudo-random mapping $f : \mathbb{Z}_n \to \mathbb{Z}_n$ to generate a sequence in which we must find a cycle. We continue by describing the choice of the pseudo-random function $f$ to be used.
CHAPTER 3. POLLARD’S RHO METHOD

3.1 Choice of pseudo-random function

The function \( f \) to be used in Pollard’s Rho method should mimic a true random function. However, we want our function to be easily implemented and quickly evaluated, so we settle for a pseudo-random function. A good choice of function is also one, which minimizes the cycle length. The algorithm terminates whenever a cycle is found, thus we would like this to happen as soon as possible. The function is evaluated three times per iteration, so it is important that it computes quickly.

Consider for example the function \( h(x) = ax + c \pmod{n} \). If \( \gcd(a, n) = 1 \), then \( h(x) \) is a bijection, as we shall see. First, we start by showing that a function \( g(x) = ax \) is a bijection.

**Theorem 1.** Let \( g(x) = ax \pmod{n} \), where \( \gcd(a, n) = 1 \). Then \( g : \mathbb{Z}_n \to \mathbb{Z}_n \) is a bijection.

**Proof.** Pick \( c, d \in \mathbb{Z}_n \), s.t.

\[
c \not\equiv d \pmod{n}. \tag{3.3}
\]

Assume that \( g(c) = g(d) \), then

\[
g(c) = g(d) \\
\iff ac \equiv ad \pmod{n} \\
\iff ac - ad \equiv 0 \pmod{n} \\
\iff a(c - d) \equiv 0 \pmod{n} \tag{3.4}
\]

Given \( \gcd(a, n) = 1 \), we know that \( a \) has a modular inverse \( b \in \mathbb{Z}_n \). Assume there is some \( e \not\equiv 0 \pmod{n} \), s.t. \( ae \equiv 0 \pmod{n} \). Then we have that \( (ba)e \equiv 1 \cdot e \equiv e \pmod{n} \), but also \( bae \equiv b \cdot 0 \equiv 0 \pmod{n} \). Thus, \( e \equiv 0 \pmod{n} \), and we have a contradiction. This means, the only solution for \( ae \equiv 0 \pmod{n} \) is \( e \equiv 0 \pmod{n} \). Using this in (3.4), we get

\[
a(c - d) \equiv 0 \pmod{n} \\
\Rightarrow c - d \equiv 0 \pmod{n} \\
\iff c \equiv d \pmod{n},
\]

which is a contradiction to (3.3). Thus, two distinct elements \( c \) and \( d \) in \( \mathbb{Z}_n \) will map to two distinct elements \( g(c) \) and \( g(d) \) in \( \mathbb{Z}_n \), when \( \gcd(a, n) = 1 \). In other words, \( g \) is a permutation of the finite set of elements \( \mathbb{Z}_n \), and necessarily bijective. \( \square \)
Consider a permutation $P$ of $\mathbb{Z}_n$. Clearly, doing modular addition with a constant $c$, on the elements of $P$, gives us another permutation $P'$ on $\mathbb{Z}_n$. Thus, the function $h(x)$ as described above, is a bijection. For Pollard’s Rho method, we do not wish to use a function which is a bijection, since in general, they require too many steps before they start cycling.

For this algorithm we use a function $f : \mathbb{Z}_n \to \mathbb{Z}_n$, defined by the polynomial $f(x) = x^2 + c \pmod{n}$. It generates a pseudo-random sequence $(x) = x_0, f(x_0), f(f(x_0)), \ldots$, where $x_0 \in \mathbb{Z}_n$.

It turns out there are values for $c$ that are unfortunate choices. If we choose $c = -2$, we have $f(x) = x^2 - 2$. If $x$ is of the form $y^k + y^{-k}$, it turns out we get values of $f(x)$, that are not very random. This is shown by (3.5).

\[
\begin{align*}
 f(y + y^{-1}) &= (y + y^{-1})^2 - 2 = y^2 + y^{-2} + 2yy^{-1} - 2 = y^2 + y^{-2} \\
 f(y^2 + y^{-2}) &= (y^2 + y^{-2})^2 - 2 = y^4 + y^{-4} + 2y^4y^{-4} - 2 = y^4 + y^{-4} \\
 &\vdots \\
 f(y^k + y^{-k}) &= (y^k + y^{-k})^2 - 2 = y^{2k} + y^{-2k} + 2y^{2k}y^{-2k} - 2 = y^{2k} + y^{-2k}.
\end{align*}
\]

(3.5)

Furthermore, using e.g. $c = 0$ will give only the quadratic residues for the specified modulus, which is not random enough[Luc05]. In [Luc05], Brandon Luders makes a comparison between the expected number of steps before cycling, based on the random heuristic and the actual observed number of steps before cycling for some random seeds in $\mathbb{Z}_p$. Here, the random heuristic means, that the expected number of steps before cycling is in the order of $\sqrt{p}$. This research indicates, that using $f(x) = x^2 + c$, is a good approximation for $p < 50,000$ and for $p > 300,000$. It seems that in between these values, some unexplained deviations occur, but since we generally focus on composites of two primes somewhat larger than 300,000, this is not a problem.

### 3.2 Finding cycles

Pollard’s Rho method uses Floyd’s cycle-finding algorithm of Section 2.3 to find the values $x$ and $x'$. The algorithm is started with an external pseudo-random function $f$ and an element $x_0 \in \mathbb{Z}_n$ as input. Two variables $x$ and $x'$ are kept. $x$ is initialized to $x_0$ and $x'$ to $f(x_0)$. At each step of the algorithm, the values $x = f(x)$ and $x' = f(f(x'))$ are computed, thus generating the sequence $\langle x \rangle$.

While the sequence $\langle x \rangle$ is being computed, the algorithm implicitly computes the sequence $\langle x_p \rangle = \langle x \rangle \pmod{p}$ too, even though $p$ is actually not
known. This is seen from (3.6), showing that $\langle x_p \rangle$ follows the same recurrence as $\langle x \rangle$, only modulo $p$. Note that the sequence $\langle x_p \rangle$ is just the sequence $\langle x \rangle$ reduced modulo $p$ element wise.

\[
\begin{align*}
    x_{p,i+1} & \equiv x_{i+1} \pmod{p} \\
    & \equiv f(x_i) \pmod{p} \\
    & \equiv (x_i^2 + c \pmod{n}) \pmod{p} \\
    & \equiv x_i^2 + c \pmod{p}, \text{ since } p \text{ divides } n \\
    & \equiv (x_i^2 \pmod{p}) + c \pmod{p} \\
    & \equiv (x_{p,i})^2 + c \pmod{p} \\
    & \equiv f(x_{p,i}) \pmod{p}.
\end{align*}
\] (3.6)

The purpose of the Pollard’s Rho method is not to find cycles in $\langle x \rangle$, but rather in $\langle x_p \rangle$. This is due to the fact, that when two elements $x, x' \in \langle x \rangle$ have been found, s.t. $\gcd(x - x', n) > 1$, a cycle has been found in $\langle x_p \rangle$, namely $x \equiv x' \pmod{p}$. As described above, this results in a non-trivial factor of $n$. In this way, computing the $\gcd(x - x', n)$ at each iteration, works as a "window" into the unseen world modulo $p$. Thus, without actually knowing $p$ beforehand, we can detect cycles in $\langle x_p \rangle$.

The algorithm terminates with failure if $\gcd(x - x', n) = n$. In this case, the cycle in the sequence $\pmod{n}$ and the cycle in the sequence $\pmod{p}$ were found in the same iteration step. This means that $x \equiv x' \pmod{n} \land x \equiv x' \pmod{p}$. In this case, the gcd naturally returns the trivial divisor $n$.

The pseudo code for the Pollard’s Rho factoring method is seen in Algorithm 1. The algorithm takes as input a pseudo-random function $f$ and an element $x_0 \in \mathbb{Z}_n$ and outputs either a non-trivial factor $p$ of $n$ or Failure.

3.3 Complexity

Let $p$ be the smallest prime factor of $n$. If we assume that $n$ is not the product of the same two primes, i.e. $n \neq p^2$, we have that $p < \sqrt{n}$. The infinite periodic sequence we use, defined by the function $f(x) = x^2 + 1 \pmod{p}$, will start cycling after about $\sqrt{p}$ steps. When a cycle is found in this sequence, we find a non-trivial factor of $n$, given that we do not find a cycle in the sequence modulo $n$ in the same step.

Thus, the expected complexity of the algorithm is $O(\sqrt[n]{n}) = O(n^{1/4})$. For the type of composites we work with, the product of two primes about the same
3.4. SUMMARY

Algorithm 1 Pollard’s Rho factoring method.

1. $x \leftarrow x_1$
2. $x' \leftarrow f(x)$
3. $d \leftarrow \gcd(x - x', n)$
4. while $d = 1$ do
5. \hspace{1em} $x \leftarrow f(x)$
6. \hspace{1em} $x' \leftarrow f(f(x'))$
7. \hspace{1em} $d \leftarrow \gcd(x - x', n)$
8. if $d = n$ then
9. \hspace{1em} return Failure
10. else
11. \hspace{1em} return $d$

size, the running time of the algorithm is likely to be the worst case possible. The analysis given here is, however a heuristic, meaning it an estimate of the expected complexity of the algorithm, rather than a rigorous proof.

3.4 Summary

In this chapter we have presented an algorithm for factoring integers in the order of $\sqrt{p}$ steps, where $p$ is the smallest factor of $n$. The algorithm uses a pseudo-random function to define an infinite period sequence modulo our composite $n$. However, using $\gcd$ operations, we may actually implicitly compute the sequence modulo the prime factors $p$ and $q$ without knowing them. It uses the algorithm by Floyd we described in Section 2.3 to detect when a cycle occurs, and when it does, attempts to factor $n$.

The algorithm is in the family of special-purpose algorithms, in that the number of steps required depends on the smallest factor $p$, and not on the composite number $n$. Thus, the algorithm will run much faster if $n$ has a small prime factor. The type of composites we work with in this thesis, namely those of the form (1.1), are the worst case scenario for Pollard’s Rho method, where the number of steps required will be in the order of $n^{1/4}$. 
Dixon’s Random Squares

This chapter will introduce a method known as Dixon’s Random Squares algorithm. The idea behind the method is, that if one can find numbers $x$ and $y$ in $\mathbb{Z}_n$ such that $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$, then $n$ divides $x^2 - y^2 = (x + y)(x - y)$, but $n$ does not divide $(x + y)$ nor $(x - y)$. Suppose $\gcd(x - y, n) = 1$, i.e. it is a trivial factor of $n$, then $(x + y)$ must be a multiple of $n$, which gives a contradiction. Thus, $\gcd(x - y, n)$ must be a non-trivial factor of $n$.

In the algorithm by Dixon, the numbers $x$ and $y$ are attempted found at random, utilizing a strategy which proves to be a strong technique which is also used in the Quadratic Sieve algorithm, which we shall describe later in Chapter 5.

**Definition 9 (Factor base).** Let $P$ be a set of small\(^1\) prime numbers. We define a factor base to be $\{-1\} \cup P$, sorted in increasing order. Throughout this thesis, we shall use $S$ to denote factor bases.

When we use a factor base in our algorithms, we shall use it to try and factor several numbers smaller than the composite $n$. We will need to store the factorization of these numbers in vector format. Since some of these numbers will be negative, we include $-1$ in the factor base, so we may factor negative numbers as well.

**Definition 10 (Smoothness).** Let $S = \{-1, z_1, z_2, \ldots, z_t\}$ be a factor base with $z_t$ being the largest prime. If a number $x \in \mathbb{Z}$ can be written on the form $(-1)^{e_0} \cdot z_1^{e_1} \cdot z_2^{e_2} \cdots z_t^{e_t}$ where $e_i \in \{0\} \cup \mathbb{N}$, we say $x$ is $z_t$-smooth. We also

\(^1\)Here, we mean small in the sense, that the prime number is small compared to the composite $n$. 

21
say, that \( x \) is smooth over \( S \), or simply that \( x \) is smooth, when the factor base is implicit.

Dixon’s algorithm will attempt to find numbers \( a_i \) and \( b_i \) such that

\[
a^2_i \equiv b_i \pmod{n},
\]

(4.1)

and \( b_i \) is smooth over \( S \), i.e.

\[
b_i = (-1)^{c_{i0}} \prod_{j=1}^{t} z_{ij}^{e_{ij}}, \quad e_{ij} \in \{0\} \cup \mathbb{N}.
\]

Obviously, the first requirement is easily satisfied. One simply chooses an \( a_i \) at random, squares the value and reduces \( \pmod{n} \). To test if \( b_i \) is smooth over \( S \), one can use a trial division algorithm. Pseudo code for one such algorithm is provided in Appendix A.

The next step of the algorithm is to find a set of the \( b_i \)'s, such that the product of these form a square. To do this, we want to select \( b_i \) values, such that for the product of the \( b_i \) values, the power of each prime \( z \) in the factor base, is even. Upon doing the trial division of the \( b_i \)'s, we may store the parity of the exponents for all the primes in the factor base (including \(-1\)) for the factorization of each \( b_i \). We will call this vector \( v_i \). Example 4 shows how this vector is constructed.

**Example 4.** Using the factor base \( S = \{-1, 2, 3, 5, 7\} \), the unique factorization of \( b_1 = 238140 \) is \( 2^2 \cdot 3^5 \cdot 5 \cdot 7^2 \), so \( 238140 \) is 7-smooth. Looking at the exponents, we get the vector \( v_1 = (0, 0, 1, 1, 0) \).

Now, the problem of finding a set of \( b_i \)'s whose product form a perfect square, has been reduced to finding a set of \( v_i \)'s whose sum, when reduced \( \pmod{2} \) becomes the zero vector. It is very suitable, to represent the collection of \( v_i \)'s in a matrix \( A \), one in each row. This way, \( A \) will have as many columns as there are elements in the factor base, i.e. \( |S| \).

Due to Section 2.4, we know if we can collect \(|S| + 1\) relations of the form \( a_i^2 \equiv b_i \pmod{n} \), where \( b_i \) is smooth over \( S \), the associated vectors must be linearly dependent. This allows us to find a subset of those, such that their sum (reduced modulo 2) is the zero vector. Please note, that finding such a subset is the same as solving \( \text{ker}(A^T) \).

Let \( T \) be the indices of the vectors in mentioned subset. That is,

\[
\sum_{r \in T} v_r \equiv 0 \pmod{2}
\]
When such a subset $T$ is found, we have $b_i$ values such that $\prod_{r \in T} b_r$ forms a perfect square. Clearly, for $r \in T$, all $a_r^2$ are perfect squares, so $\prod_{r \in T} a_r^2$ is also a perfect square.

Thus, if we set

$$x = \prod_{r \in T} a_r, \quad \text{and}$$

$$y = \prod_{r \in T} b_r,$$

we have a relation of the form

$$x^2 \equiv y^2 \pmod{n}.$$

If also $x \not\equiv \pm y \pmod{n}$, we have found a non-trivial factor of $n$, as described in the beginning. Should we be so unlucky, that $x \equiv \pm y \pmod{n}$, we go back and see if we can find a different subset $T$ with the same property, i.e. another vector in $\ker(A)$. Should this not be possible, we will have to randomly generate more $(a_i, b_i)$ pairs. In practice, we could collect more than $|S| + 1$ relations and, by the Rank-Nullity theorem of Section 2.4, increase the number of vectors in the kernel, giving us more combinations to try.
In this chapter, we will be describing a factorization method, called the Quadratic Sieve. It was invented in 1981 by Carl Pomerance [Pom96], based on ideas by Kraitchik and Dixon [Lan01]. The method exists in form of various algorithms. The original algorithm shall be denoted QS (Quadratic Sieve). Chapter 6 describes an improved version of the original algorithm called MPQS (Multiple Polynomial Quadratic Sieve). More versions still exist, one is the SIQS (Self Initializing Quadratic Sieve). For more information on SIQS, please see [Con97].

5.1 Motivation

In Chapter 4 we described Dixon’s Random Squares algorithm, which chooses values at random to find squares that factor over some factor base. One may use these relations to form a perfect square relation that hopefully will provide a non-trivial factor of the composite $n$.

The motivation for the Quadratic Sieve, is that one can do better than choosing values at random to find these relations. Let us consider an interval $[2, x]$ where $x$ is some integer. How many numbers in the interval $[2, x]$ can be factored using only primes less than or equal to some prime $B < x$, i.e. they are $B$-smooth? Let us consider some value $y \in [2, x]$. Intuitively, it makes good sense, that the larger $y$ is, the less likely it is to be $B$-smooth. Thus, the distribution of $B$-smooth numbers in $[2, x]$ is probably more dense in the beginning of the interval. This theory is supported by the work of Karl Dickman [Dic30] and Nicolaas G. de Bruijn [dB51][dB66]. They introduce a
function to estimate the proportion of $B$-smooth numbers in the interval $[1, x]$,

$$\frac{\Psi(x, B)}{x} \approx u^{-u},$$

where $u = \frac{\ln x}{\ln B}$, and $\Psi(x, B)$ is the number of $B$-smooth numbers in $[0, x]$. Thus, the expression above gives an estimate of the probability of some number $x$ being $B$-smooth. Figure 5.1 shows a plot of this function. Clearly $B$-smooth values are much more probable when $x$ is low. This function is a heuristic, which has also been estimated by Pomerance\cite{Pom05} and Marian Kechlibar\cite{Kec05}.

Looking back at (4.1) of Chapter 4, we still want to find relations $a_i^2 \equiv b_i \pmod{n}$ s.t. $b_i$ is $B$-smooth. Since it is the $b_i$ values we hope to be smooth over the factor base, using the argument above, we now want to choose $a_i$ s.t. $b_i$ becomes small, rather than choosing $a_i$ at random. This way, the probability of $b_i$ being $B$-smooth becomes larger. The choice of the smoothness bound $B$ will be discussed in Section 5.4.

### 5.2 The algorithm

As always, let $n$ be the composite integer which we wish to factor. We let $m = \lceil \sqrt{n} \rceil$, that is, $m$ is the largest integer less than or equal to $\sqrt{n}$. We define

$$Q(x) = (x + m)^2 - n = x^2 + m^2 + 2mx - n \approx x^2 + 2mx.$$  \hspace{1cm} (5.2)

We let $a_i = (x + m)$ and $b_i = (x + m)^2 - n$, and hope that $b_i$ is smooth over the factor base. Note, that according to (5.2), $b_i \approx x^2 + 2mx$ is small compared to
5.2. THE ALGORITHM

the composite \( n \). When chosen in a structured manner like this, the \( b_i \) values should be smooth over the factor base with higher probability, than if they were chosen at random like they are in the Dixon’s Random Squares algorithm of Chapter 4.

**Definition 11** (Quadratic residue). *When the relation* \( x^2 \equiv u \pmod{z} \) *has a solution for* \( x \), *u is said to be a quadratic residue* \( \pmod{z} \). *If no solution for* \( x \) *exists, u is said to be a quadratic non-residue* \( \pmod{z} \). *If* \( x = 0 \) *is the only solution, u is neither a quadratic residue, nor a quadratic non-residue. Throughout this thesis, we shall often state that u is a residue, when the modulus is given implicit, and we mean that u is a quadratic residue.*

**Definition 12** (Legendre symbol). *We define the Legendre symbol* \( \left( \frac{u}{z} \right) \) *for* \( u \in \mathbb{Z} \) *and a prime* \( z \) *to be* \( 1 \) *if* \( u \) *is a quadratic residue* \( \pmod{z} \), *and* \( -1 \) *if* \( u \) *is a quadratic non-residue* \( \pmod{z} \). *If* \( u \equiv 0 \pmod{z} \) *we define* \( \left( \frac{u}{z} \right) \) *as* \( 0 \). *That is,*

\[
\left( \frac{u}{z} \right) = \begin{cases} 
1 & : u \text{ is a quadratic residue } \pmod{z} \\
-1 & : u \text{ is a quadratic non-residue } \pmod{z} \\
0 & : u \equiv 0 \pmod{z}
\end{cases}
\]

Please note, that if for some prime \( z, z \mid b_i \), meaning \( z \) divides \( b_i \), then

\[
z \mid b_i \\
\iff z \mid (x + m)^2 - n \\
\iff zk = (x + m)^2 - n \\
\iff (x + m)^2 - zk = n \\
\iff (x + m)^2 \equiv n \pmod{z}.
\] (5.3)

Hence \( \left( \frac{n}{z} \right) = 1 \). This means we need only include primes in our factor base, for which \( n \) is a quadratic residue.

For the original Quadratic Sieve, we simply generate \((a_i, b_i)\) pairs by computing \( a_i = (x + m) \), with \( x = 0, \pm 1, \pm 2, \ldots \). This way we start with the smallest values and increase them, until we have found enough relations for the matrix \( A \), which holds the parity of the prime exponents associated with the factorizations of the smooth \( b_i \)'s.

Just as for the Dixon’s Random Squares algorithm, which we presented in Chapter 4, we collect \(|S| + 1\) \((a_i, b_i)\) pairs using some factor base \( S \), where of course the \( b_i \) values are smooth over \( S \). Again, we save the parity of the prime exponents for the factorization of each \( b_i \) in a row vector \( v_i \) which we insert
into our matrix $A$ which will eventually be a $(|S| + 1) \times |S|$ matrix. Note also, that $A$ is defined over $\mathbb{Z}_2$.

Next, we solve $\ker(A^T)$. Assume $w$ is a vector of length $|S| + 1$ in $\ker(A^T)$. Again, we let $T$ denote a set of indices of vector $w$ containing a 1. That is

$$T = \{ x \mid \text{element } x \text{ of } w = 1 \}, \quad (5.4)$$

meaning $T$ is the set of indices of row vectors in $A$ for which the sum over $\mathbb{Z}_2$ is the zero vector. Having found $T$, we again compute $x$ and $y$ just like in (4.2) and (4.3). If we find $x$ and $y$ s.t. $x \not\equiv \pm y \pmod{n}$, then $\gcd(x - y, n)$ is a non-trivial factor of $n$. Should this not be the case, we try one of the other combinations in the kernel. Should we be so unfortunate, that none of these yield a non-trivial factor of $n$, we replace some of the rows of $A$ with new $(a_i, b_i)$ pairs and compute $\ker(A^T)$ again.

Pseudo code for the Quadratic Sieve is presented in Algorithm 2. At first glance, it might not be clear what is going on in line 21 through 23 of Algorithm 2, so we will explain that here. What we wish is to find the square root of the product of our $b_i$ values. Of course, we would like to avoid computing the modular square root (mod $n$), since this is generally hard for composites $n$ which are not prime powers. Let us assume we use different $b_i$ values with $i \in \{0, 1, \ldots, j\}$, and a factor base with $t + 1$ elements (including $-1$). We have

$$b_0 = z_0^{e_{00}} \cdot z_1^{e_{01}} \cdots z_t^{e_{0t}}$$
$$b_1 = z_0^{e_{10}} \cdot z_1^{e_{11}} \cdots z_t^{e_{1t}}$$
$$\vdots$$
$$b_j = z_0^{e_{j0}} \cdot z_1^{e_{j1}} \cdots z_t^{e_{jt}}.$$  

The product of those then becomes

$$b_0 \cdot b_1 \cdots b_j = \left( z_0^{e_{00}} \cdot z_1^{e_{01}} \cdots z_t^{e_{0t}} \right) \cdot \left( z_0^{e_{10}} \cdot z_1^{e_{11}} \cdots z_t^{e_{1t}} \right) \cdots \left( z_0^{e_{j0}} \cdot z_1^{e_{j1}} \cdots z_t^{e_{jt}} \right)$$
$$= \left( z_0^{e_{00} + e_{10} + \cdots + e_{j0}} \cdot z_1^{e_{01} + e_{11} + \cdots + e_{j1}} \cdots z_t^{e_{0t} + e_{1t} + \cdots + e_{jt}} \right). \quad (5.5)$$

Since we already made sure, by computing $\ker(A^T)$ over $\mathbb{Z}_2$, that the sum of each prime’s exponents of the used $b_i$ values are even, that the prime exponents of (5.5) are all even. This allows us to easily compute the square root of the
5.2. **THE ALGORITHM**

---

**Algorithm 2** Quadratic Sieve algorithm.

1. **INPUT:** A composite integer \( n \) to be factored
2. **OUTPUT:** A non-trivial factor of \( n \)
3. Compute a factor base \( S = \{z_1, z_2, \ldots, z_t\} \). Set \( z_1 = -1 \), and \( z_j \), \( j \geq 2 \) is the \((j - 1)^{th}\) prime satisfying \( \left( \frac{n}{z_j} \right) = 1 \).
4. \( m \leftarrow \lfloor \sqrt{n} \rfloor \)
5. Initialize a matrix \( A \) with dimension \((|S| + 1 \times |S|) \) with 0's
6. \( x \leftarrow 0 \)
7. \( i \leftarrow \) index of first empty row in \( A \)
8. **while** \( i < |S| + 1 \) **do**
9. \( b \leftarrow Q(x) = (x + m)^2 - n \)
10. **if** \( b \) is smooth over \( S \) **then**
11. \( a_i \leftarrow (x + m) \)
12. \( b_i \leftarrow b \)
13. \( v_i \leftarrow (e_{i1}, e_{i2}, \ldots, e_{it}) \pmod{2} \), where \( e_{ij} \) is the exponent for \( \) element \( j \) in the factor base in the unique factorization of \( b_i \)
14. Insert \( v_i \) into row \( i \) of \( A \)
15. \( i \leftarrow i + 1 \)
16. \( x \leftarrow \text{next}(x) \) \{next\( (x) \) is the next value in the sequence \( 0, \pm 1, \pm 2, \ldots \)\}
17. \( K \leftarrow \ker(A^T) \)
18. **for** every vector \( w \) in \( K \) **do**
19. Let \( T \) be the indices of non-zero values in \( w \)
20. \( x \leftarrow \prod_{i \in T} a_i \pmod{n} \)
21. **for** \( j = 1 \) to \(|S| \) **do**
22. \( l_j \leftarrow (\sum_{i \in T} e_{ij}) / 2 \)
23. \( y \leftarrow \prod_{j=1}^{|S|} z_j^{l_j} \pmod{n} \)
24. **if** \( x \not\equiv \pm y \pmod{n} \) **then**
25. **return** \( \gcd(x - y, n) \)
26. Remove some rows from \( A \) and return to line 7.

---

\[
\sqrt{b_0 \cdot b_1 \cdots b_j} = \sqrt{\left( z_0^{e_{00}+e_{10}+\ldots+e_{j0}} \cdot z_1^{e_{01}+e_{11}+\ldots+e_{j1}} \cdots z_t^{e_{0t}+e_{1t}+\ldots+e_{jt}} \right)} \\
= \left( z_0^{(e_{00}+e_{10}+\ldots+e_{j0})/2} \cdot z_1^{(e_{01}+e_{11}+\ldots+e_{j1})/2} \cdots z_t^{(e_{0t}+e_{1t}+\ldots+e_{jt})/2} \right). \tag{5.6}\]

Note that the exponents of (5.6) are exactly those computed in line 21 through 22 of Algorithm 2, allowing us to compute \( y \) without computing the square
root of the $b_i$ product. The following example will show the workings of the Quadratic Sieve.

**Example 5.** Let the composite $n = 164009 = 401 \cdot 409$. We choose a smoothness bound $B = 41$, meaning every prime in the factor base $S$ will be less than or equal to 41. We calculate our factor base $S$ to be $\{-1, 2, 5, 13, 19, 31, 37, 41\}$.

Next we compute $m = \lfloor \sqrt{164009} \rfloor = 404$. We now proceed to check $Q(x) = (x + m)^2 - n$ for smoothness over $S$, in the order $x = 0, \pm 1, \pm 2, \ldots$ until we have found $|S| + 1 = 9$ pairs of $(a_i, b_i)$. The results of this process are seen in the following table.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x$</th>
<th>$a_i$</th>
<th>$Q(x)$</th>
<th>$Q(x)$ factorization vector</th>
<th>$v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>405</td>
<td>16</td>
<td>$(0, 4, 0, 0, 0, 0, 0, 0)$</td>
<td>$(0, 0, 0, 0, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>403</td>
<td>-1600</td>
<td>$(1, 6, 2, 0, 0, 0, 0, 0)$</td>
<td>$(1, 0, 0, 0, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>402</td>
<td>-2405</td>
<td>$(1, 0, 1, 1, 0, 0, 0, 0)$</td>
<td>$(1, 0, 1, 0, 0, 0, 1, 0)$</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>407</td>
<td>1640</td>
<td>$(0, 3, 1, 0, 0, 0, 0, 0)$</td>
<td>$(0, 1, 1, 0, 0, 0, 0, 1)$</td>
</tr>
<tr>
<td>5</td>
<td>-7</td>
<td>397</td>
<td>-6400</td>
<td>$(1, 8, 2, 0, 0, 0, 0, 0)$</td>
<td>$(1, 0, 0, 0, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>412</td>
<td>5735</td>
<td>$(0, 0, 1, 0, 0, 0, 0, 1)$</td>
<td>$(0, 0, 0, 0, 0, 0, 1, 0)$</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>413</td>
<td>6560</td>
<td>$(0, 5, 1, 0, 0, 0, 0, 0)$</td>
<td>$(0, 1, 1, 0, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>417</td>
<td>9880</td>
<td>$(0, 3, 1, 1, 1, 0, 0, 0)$</td>
<td>$(0, 1, 1, 1, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>419</td>
<td>11552</td>
<td>$(0, 5, 0, 0, 2, 0, 0, 0)$</td>
<td>$(0, 1, 0, 0, 0, 0, 0, 0)$</td>
</tr>
</tbody>
</table>

If we collect all the $v_i$ vectors as being the rows in a matrix $A$, we can find a linear combination of $v_i$ vectors that becomes the zero vector over $\mathbb{Z}_2$, by solving $\ker(A^T)$. Doing so we get the kernel basis

$$\ker(A^T) = \text{span}\{(0, 0, 0, 1, 0, 0, 1, 0, 0), (0, 1, 0, 0, 1, 0, 0, 0, 0), (1, 0, 0, 0, 0, 0, 0, 0, 0)\}$$

As the basis of $\ker(A^T)$ contains three vectors, we have $2^3 - 1$ different linear combinations to get a perfect square out of the $b_i$ values. Using the notation of Algorithm 2, for the first vector $w = (0, 0, 0, 1, 0, 0, 1, 0, 0)$ in $\ker(A^T)$, we get our indices $T = \{4, 7\}$, hence $x = (a_4a_7 \mod n) = 4082$. 
We then compute the $l$ values $l_j = \left(\sum_{i \in \mathcal{T}} e_{ij}\right)/2$, $1 \leq j \leq |S|:$

\[
\begin{align*}
l_1 &= (0 + 0)/2 = 0 \\
l_2 &= (3 + 5)/2 = 4 \\
l_3 &= (1 + 1)/2 = 1 \\
l_4 &= (0 + 0)/2 = 0 \\
l_5 &= (0 + 0)/2 = 0 \\
l_6 &= (0 + 0)/2 = 0 \\
l_7 &= (0 + 0)/2 = 0 \\
l_8 &= (1 + 1)/2 = 1
\end{align*}
\]

We get $y = \prod_{i=1}^{|S|} z_i^{l_i} = 2^4 \cdot 5 \cdot 41 \mod n = 3280$. Since $x \not\equiv \pm y \pmod{n}$, we compute $\gcd(4082 - 3280, 164009) = 401$.

Please note that in this case, an even easier solution exists. In the table, $v_1$ is the zero vector by itself already. This means, we have found a single $(a_i, b_i)$ pair, where both are perfect squares modulo $n$. Using $a_1 = 405$ and $b_1 = 2^{4/2} = 4$, we see that $a_i \not\equiv \pm 4 \pmod{n}$. Thus, $\gcd(405 - 4, 164009) = 401$ is a factor of $n$.

Next, we will proceed by describing how we can improve the Quadratic Sieve algorithm.

5.3 Sieving

Even though the version of the Quadratic Sieve we described so far, is clearly an improvement compared to Dixon’s Random Squares of Chapter 4, there is a quite clever improvement we can use, to avoid checking each $Q(x)$ for smoothness by trial division. This improvement is based on an observation regarding the structure of the polynomial $Q(x)$ we have chosen for our $b_i$ values. In the following derivation (5.7), assume that $Q(r) \equiv 0 \pmod{z}$, and let $l \in \mathbb{Z}$.

\[
Q(r + lz) \equiv ((r + lz) + m)^2 - n \pmod{z} \\
\equiv (r + lz)^2 + m^2 + 2m(r + lz) - n \pmod{z} \\
\equiv r^2 + (lz)^2 + 2rlz + m^2 + 2mr + 2mlz - n \pmod{z} \\
\equiv (r^2 + m^2 + 2mr - n) + (lz)^2 + 2rlz + 2mlz \pmod{z} \\
\equiv Q(r) + (lz)^2 + 2lz(r + m) \pmod{z} \\
\equiv 0 \pmod{z} \quad , \text{assumes } z \mid Q(r). \quad (5.7)
\]
From (5.7) we can deduce, that if \( x \) is a root of \( Q(x) = (x + m)^2 - n \) (mod \( z \)), meaning \( Q(x) \equiv 0 \) (mod \( z \)), for some prime \( z \) in our factor base, then \( z \) also divides \( Q(x + lz) \) for any \( l \in \mathbb{Z} \). This is a very useful observation for our application. Seeing as we are looking for \( Q(x) \) values that are smooth over our factor base, we may use this technique to find many \( Q(x) \) quickly, for which some prime \( z \) in our factor base divides \( Q(x) \). This technique is called sieving, hence the Quadratic Sieve.

To sieve, we first need to find a solution to \( Q(x) \equiv 0 \) (mod \( z \)) for each prime \( z \) in our factor base:

\[
Q(x) \equiv 0 \pmod{z} \quad (5.8)
\]

\[
\Leftrightarrow (x + m)^2 - n \equiv 0 \pmod{z} \quad \Leftrightarrow (x + m)^2 \equiv n \pmod{z} \quad (5.9)
\]

From (5.9) we see, that finding a root becomes a problem of computing a modular square root modulo a prime number. Since for every prime \( z \) in our factor base, we have that \( \left( \frac{\mathbb{Q}}{\mathbb{Z}} \right) = 1 \), we know there are two solutions (see Appendix B). For this purpose an algorithm like Shanks-Tonelli's \([\text{Sti}05]\) can be used. Let \( t \) and \( -t \) be solutions to \( x^2 \equiv n \) (mod \( z \)), then from (5.9) we find two roots for (5.8), \( r_1 = (t - m) \) (mod \( z \)) and \( r_2 = (-t - m) \) (mod \( z \)).

If we look at the factorization of some \( Q(x) \), we can use (5.10) to explain a relation between the factorization of \( Q(x) \) and \( \log(|Q(x)|) \):

\[
Q(x) = z_1^{e_1} \cdot z_2^{e_2} \cdots z_t^{e_t}
\]

\[
\Leftrightarrow \log(|Q(x)|) = e_1 \log|z_1| + e_2 \log|z_2| + \cdots + e_t \log|z_t| \quad (5.10)
\]

Now, let \( L \) be an array indexed by \(-M \leq x \leq M\). We initialize \( L \) with 0 on each position. Let \( r_{1z} \) and \( r_{2z} \) be the solutions to \( Q(x) \equiv 0 \) (mod \( z \)) for each prime \( z \) in our factor base. We then go through \( L \) and add \( \log z \) to each \( L[r_{1z} + lz] \) and \( L[r_{2z} + lz] \), as long as \(-M \leq r_{1z} + lz \leq M \) and \(-M \leq r_{2z} + lz \leq M \).

The idea behind the sieving process is, that after performing the sieving process with each prime in the factor base, we would expect \( L[x] \) to be somewhat close to \( \log(|Q(x)|) \). This is only true, of course, if we did actually sieve with all the prime factors of \( Q(x) \). In other words, if \( L[x] \approx \log(|Q(x)|) \), there is a good chance that \( Q(x) \) is smooth over our factor base!

Please note, that the sieving process described here, does not take prime powers into consideration. To do so we would have to, for each prime in the factor base, compute the exponent for that prime in the factorization of \( Q(x) \) for that prime. Doing so, we would in effect end up performing trial division.
on $Q(x)$, which is what we wanted to avoid in the first place. Also note, that had we chosen to take prime powers into consideration, we would still not expect $L[x] = \log(|Q(x)|)$ for smooth $Q(x)$, due to rounding errors.

This taken into consideration, we may still expect for $Q(x)$ values which are smooth over the factor base, to have considerably higher values of $L[x]$. This way, we can do the sieving process and find a set of candidates that are more likely to be smooth over the factor base, than the others. For example we may choose our candidates as those $Q(x)$ with $L[x]$ being larger than some bound, for $-M \leq x \leq M$. As an alternative, one may keep a list of indices $x$ sorted by the value $L[x]$, while sieving. One may then try to factor the $Q(x)$ candidates one at a time, by trial division, to verify that they are indeed smooth over the factor base. Say we choose to only check $Q(x)$ values by trial division, for which $L[x]$ has accumulated some minimum bound, how should this bound be chosen? We will attend this question in Section 5.4.

Figure 5.2 Comparison of factorizations by trial divisions needed for 100 composites.

Factoring by trial division is a cumbersome process, thus it makes sense to try and minimize the use of it. That is what is accomplished by the sieving process. Figure 5.2 shows a relation between the number of factorizations with trial division performed, by two different implementations of the QS, one that uses sieving and one that does not. We will describe these implementations in Chapter 7 (see also Appendices E and F). The plot shows, for both implementations, the number of trial division factorizations done for 100 composites, that are the product of two primes of about the same size. Please note that the 2. axis is in a logarithmic scale. The composites $n$ were generated by the code in Appendix C. The smoothness bound used for the factor base was $B = 1000$ and the sieve array $L$ had a length of 200001. Next, we will describe how we
may tune the parameters of the Quadratic Sieve algorithm.

## 5.4 Parameter tuning

There are a number of variables for the Quadratic Sieve algorithm, that we have yet to describe how to choose in an optimal manner. These variables are the factor base size $|S|$ and the bound on $L[x]$, for which we consider the corresponding $Q(x)$ a candidate for factorization by trial division.

### Choice of sieving candidate bound

Obviously, the choice of bound for what we consider a *sieving candidate*, i.e. a sieve value that has accumulated some specific minimum bound, will have an impact on the success of the algorithm.

If we choose the bound low, we will have a lot of candidates, and probably also more $Q(x)$ values that are smooth over the factor base among those candidates. However, this implies that we will probably also take a larger fraction of non-smooth $Q(x)$ values into consideration as candidates. Thus, we risk attempting to factor too many non-smooth $Q(x)$ values by trial division, than what good is.

So why not be on the safe side, and keep a rather high bound on what we consider being candidates? The downside to this approach is, we might end up excluding potentially many smooth $Q(x)$ values that would otherwise give us good relations. Note that even if we had set the bound very low, e.g. a bound of 0 meaning no requirement, for some factor base $S$, we could not be certain to find $|S| + 1$ values of $Q(x)$ that are smooth over $S$. This could be the case, if we choose our sieve interval $[-M, M]$ or smoothness bound $B$ (thus also $|S|$), too small. Assuming this is not the case, there is potential of discarding too many smooth $Q(x)$ values if we choose the sieve candidate bound too high. If this happens, we will have to either use a larger factor base, a larger sieve interval or possibly set up a lower sieve candidate bound.

What we really want, is a good bound value, that hopefully assures we find enough smooth $Q(x)$ values, without having to attempt factorization by trial division on too many non-smooth $Q(x)$ values. In [Con97], Scott Contini gives a suggestion for a bound,

$$SB(x) = \log[2x\sqrt{n}] \quad (5.11)$$

This value makes good sense, since from (5.2) we have that $Q(x) \approx x^2 + 2xm = x^2 + 2x\lfloor\sqrt{n}\rfloor$. Thus, we use the term $x^2$ as an error margin for rounding errors.
5.4. PARAMETER TUNING

in the sieving process, plus the fact that we do not sieve with prime powers. This bound is the one we will use in our implementations of algorithms in the Quadratic Sieve family.

Eric Landquist suggests a different value in [Lan01], based on one by Robert Silverman [Sil87]:

\[ SB_2(n) = \frac{\log[n]}{2} + \log[M] - T \log[z_t], \quad (5.12) \]

where \( T \) is a value close to 2 and \( z_t \) is the largest prime in the factor base. Other literature, e.g. [Ger83] and [MvOV01], give

\[ SB_3(x) = \log(|Q(x)|), \quad (5.13) \]

as a value. In practice, it does not matter too much which of the values above we use. Looking at (5.11) and (5.13), for example, and letting \( n = 10^{70} \), we may solve that we will need a sieve length of \( M = 2 \cdot 10^{35} \) for there to be any difference between the two bounds, which is not realistic in any way.

Figure 5.3 shows how the implemented sieve candidate bound of (5.11) for the Quadratic Sieve algorithm captures the smooth \( Q(x) \) values.

![Figure 5.3 Smooth Q(x) with sieve values and candidate bounds for n = 1000076001443, M = 100000 and |S| = 88 (smoothness B = 1000).](image)

Choice of factor base size

First off, note that whether we talk about the size of the factor base used, or the smoothness bound \( B \) used for the factor base, in the Quadratic Sieve algorithm, does not really matter. The two values go hand in hand: fixing some
number of primes in the factor base automatically determines a largest prime in the factor base $z_t$, such that all $Q(x)$ values of interest will be $z_t$-smooth. On the contrary, if we choose some smoothness bound $B$ on our factor base, there will be a fixed amount of factor base primes. Different literature on the Quadratic Sieve algorithm give different optimal choices here. Some papers give an optimal value for smoothness bound $B$\cite{Con97}\cite{Pom05}, while others give an optimal value for the factor base size $|S|$\cite{Lan01}.

The size of the factor base $|S|$ to be used in the Quadratic Sieve is very much an optimization problem. Clearly, it is a choice that will have an impact on the complexity, and ultimately on the time spent to factor a composite $n$. The choice is a tradeoff between two major metrics.

On one hand, if we choose a somewhat low value $|S|$, the matrix for which we will have to solve the kernel, will be relatively small. Naturally, this will make the problem of solving the kernel, and finding $x$ and $y$, faster. However for small values of $|S|$, the distribution of numbers in $\mathbb{Z}_n$ that are smooth over the factor base is sparse\cite{Pom05}. Thus, we may have to try a lot of $Q(x)$ values, before we find one that is smooth over $S$.

On the other hand, if we just go ahead and choose a high value of $|S|$, it will be easier to find values that are smooth over $S$. But choosing $|S|$ big, we will have to gather all that more relations, and we might end up with a matrix much larger than needed. As a consequence of this, it will not only take longer time to gather the relations, but the kernel will take longer to solve.

Obviously if two different factor base sizes lead to similar results with the other parameters fixed, it is always better to choose the smaller one, for saving computation time in the sieving phase and in the kernel computation phase. The problem of course is, that we have no way of knowing beforehand, if two different factor base sizes both will lead to positive results.

The following value for a smoothness bound $B$ is suggested by Contini\cite{Con97}, Pomerance\cite{Pom05} and Menezes et. al.\cite{MvOV01}:

$$B = e^{(0.5\sqrt{\ln n \ln \ln n})}.$$ \hfill (5.14)

For the rest of this thesis, we shall refer to this bound as the \textit{Contini bound}. Another value on the size of the factor base $|S|$ is suggested by Landquist\cite{Lan01}:

$$|S| = e^{\left(\frac{\sqrt{2}}{4}\sqrt{\ln n \ln \ln n}\right)}. \hfill (5.15)$$

This bound will from here on be denoted the \textit{Landquist bound}. We need to make a slight modification to (5.14) to be able to compare it to (5.15). First, we will need the \textit{Prime Number Theorem}. 


Definition 13 (Prime Number Theorem). Define the number of primes less than or equal to $x$ as $\pi(x)$. The Prime Number Theorem states that

$$\pi(x) \approx \frac{x}{\ln x}.$$ 

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\pi(x)$</th>
<th>$\pi(x) - \lfloor x/\ln x \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10$</td>
<td>$4$</td>
<td>$0$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$25$</td>
<td>$3$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$168$</td>
<td>$23$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$1229$</td>
<td>$143$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$9592$</td>
<td>$906$</td>
</tr>
</tbody>
</table>

Table 5.1 Precision of the Prime Number Theorem for varying upper bounds $x$.

Table 5.1 shows values for $\pi(x)$ for varying $x$, along with the precision of the Prime Number Theorem.

As we know, we only include the primes in the factor base, for which $n$ is a quadratic residues. We will need the following result.

Theorem 2. For some odd prime $z$, there are $(z - 1)/2$ quadratic residues in $\mathbb{Z}_p$.

Proof. The proof follows from the fact that any $x^2 \in \mathbb{Z}_z$ has two solutions. In other words there are two different $x, x' \in \mathbb{Z}_z$ s.t. $x^2 \equiv x'^2 \pmod{z}$. Hence, we have a 2-to-1 mapping, and half of $\mathbb{Z}_z$ must be quadratic residues, and the other half quadratic non-residues.

We may now estimate how many primes we can expect in a factor base of some given smoothness bound $B$. In particular, we wish to find out how many primes we may expect in the factor base using the Contini bound, allowing us to compare with the Landquist bound.

Assuming the quadratic residues in $\mathbb{Z}_z$ are randomly distributed, we may use Theorem 2, and claim that $n$ is a quadratic residue modulo $z$ with probability 0.5. Thus, we may expect $n$ to be a quadratic residue modulo half of the primes below some bound. We combine this with the Prime Number Theorem, to give an estimate $S_E(n)$ on the number of primes in the factor base, when
using the Contini bound:

\[ S_E(n) = \frac{B}{2\ln B} \]

\[ = e^{(0.5\sqrt{\ln \ln n})} \]

\[ = \frac{2\ln e^{(0.5\sqrt{\ln \ln n})}}{\sqrt{\ln \ln n}}. \]  

(5.16)

Figure 5.4a shows how the expected factor base size, when using the Contini bound, relates to the Landquist bound. The two functions cross at about \( n = 8 \cdot 10^{40} \). Thus, we can expect that using the Landquist bound for composites smaller than this value, should result in larger factor bases, than had we used the Contini bound. However, when \( n \) becomes sufficiently large, we expect that using the Contini bound will yield the largest factor base.

For our implementations of the Quadratic Sieve and its variants, we will implement a switch, that allows us to change between the Contini bound and Landquist bound, to determine our factor base size. Doing so will allow us to compare the time spent to factor \( n \) for both bounds.

Figure 5.4b which shows the actual factor base sizes, for some smaller composites \( n \), indicates that using a smoothness bound rather than a fixed factor base size, can result in very varying factor base sizes for two different \( n \) close to each other. This is a consequence of the fact that using the Contini bound relies on the structure of the composite \( n \) rather than its size. Thus, using it can give variations in the factor base size.
5.5 Complexity

The Quadratic Sieve is a rather complex algorithm, compared to e.g. Pollard’s Rho method of Chapter 3. There are a lot of different steps, thus the analysis of the complexity of the algorithm is necessarily going to be rough. The two major steps of the algorithm are:

1. Collect $|S| + 1$ residues $Q(x)$ that are smooth over $S$.

2. Find a subset of the smooth $Q(x)$ whose product form a perfect square. We do this by solving the kernel over $\mathbb{Z}_2$ of the matrix with the parity of the prime factorization exponents of the smooth $Q(x)$ in the rows.

For our analysis, we will be asking questions to which no correct answers have been proven. The best we can do is use approximations, thus our analysis will be a heuristic. One example of such a question is "how many primes are in the interval $[0, x]$?". A question to which we have already given an estimation of an answer with the Prime Number Theorem.

First, let us try to analyze how many operations we need for the first step. The first assumption we shall make, is that the absolute value of our residues $Q(x)$ have an upper bound of $n$. Recall that $Q(x) = (x + m)^2 - n \approx x^2 + 2x m$. Thus, as long as $x^2 + 2x \sqrt{n} < n$, our assumption will hold. Solving for $x$, we find that our requirement is about $x < 2\sqrt{n}$. This is reasonable, since if $x$ gets this big, factorization of $n$ by trial division would have been faster.

Recall, that we have given two bounds that relate to the size of the factor based use, namely the Contini smoothness bound $B$ and the Landquist bound on $|S|$. In this analysis, we will assume using some smoothness bound $B$ to determine the factor base primes. We need to collect $|S| + 1$ smooth residues, to attempt to factor $n$. However, using a smoothness bound $B$, the factor base size $|S|$ is not predetermined. This is mainly due to the restriction, that $n$ should be a quadratic residue modulo all factor base primes. For simplicity, we shall assume all primes below $B$ will be in the factor base. We can then estimate the factor base size with $\pi(B)$.

In Section 5.1, we defined a function which gives the proportion of $B$-smooth numbers in an interval $[0, x]$, approximated by

$$\frac{\Psi(x, B)}{x} \approx u^{-u},$$

where $u = \frac{\ln x}{\ln B}$. The function $\Psi(x, B)$ is the number of $B$-smooth numbers in $[0, x]$. Thus, we can use the expression $\Psi(x, B)/x$ as an approximation of
the probability of a random number in $[0, x]$ being $B$-smooth. Taking the reciprocal of this probability, we find that we need to check about $x/\Psi(x, B)$ numbers in $[0, x]$ to find just one that is $B$-smooth.

What we are interested in, is how many numbers in $[0, n]$ we need to check, to find $\pi(B)$, that are smooth over the factor base. Thus, we expect to check

$$\frac{n}{\pi(n, B)}$$

residues in order to find enough to attempt to factor $n$.

But how long does it take to check a single residue for $B$-smoothness? Recall, that in the sieving phase, we sieve on values $L[r_z + lz]$, where $l \geq 0$ is a whole number $Q(r_z) \equiv 0 \pmod{z}$. Thus, we sieve on every $z$ values, or in other words, we sieve on $1/z$ values in the interval, for each prime $z$ in the factor base. When looking for smooth residues in $[0, n]$, this takes

$$n \sum_{z \in S} \frac{1}{z}$$

steps. Using this argument, it will take about $\sum_{z \in S} \frac{1}{z}$ steps to recognize a single smooth residue. In [Pom05], Pomerance estimates, for all primes $z \leq t$, that

$$\sum_{z \leq t} \frac{1}{z} = \ln \ln t + C + O \left( \frac{1}{\ln t} \right)$$

(5.17)

for some constant $C$. For our analysis, we discard the $C + O \left( \frac{1}{\ln t} \right)$ term, as we are interested in an asymptotic complexity. Since we are using a smoothness bound $B$ to define our factor base $S$, we may use (5.17) to estimate the total number of steps required to collect $\pi(B)$ smooth residues in $[0, n]$:

$$\pi(B) \ln \ln B \frac{n}{\Psi(n, B)} = \frac{B}{\ln B} \ln \ln B \left( \frac{\ln n}{\ln B} \right) \left( \frac{\ln n}{\ln B} \right)$$

(5.18)

Using the Contini bound, $B = e^{0.5\sqrt{\ln n \ln \ln n}}$, it can be shown that (5.18) can be reduced to

$$O \left( e^{\sqrt{\ln n \ln \ln n}} \right)$$

(5.19)

which serves as an asymptotic upper bound on the complexity of the Quadratic Sieve.
Note, that the heuristic complexity of (5.19) is somewhat imprecise. For example, it does not take into consideration the step of computing the kernel of the matrix, containing the smooth relations. Also, it does not address the fact, that we may have to remove relations from the matrix, and find new smooth relations.

5.6 Summary

In this chapter, we have presented the Quadratic Sieve algorithm, which was invented by Carl Pomerance in 1981. The algorithm builds on the idea of the Dixon’s Random Squares algorithm of Chapter 4.

The algorithm introduces structure in how we attempt to find relations, for which the quadratic residue is smooth over some factor base. This is in contrast to the Dixon’s algorithm where we chose values at random. Specifically, we use a polynomial $Q(x)$ of a certain structure to keep the quadratic residues low, which makes it more probable to be smooth over the factor base.

The structure of our polynomial $Q(x)$ also allows us to use the concept of sieving. This way, we can find a lot of $Q(x)$ that are likely to be smooth over the factor base, without checking each one for factorization by trial division. After sieving, we will have discarded most of the non-smooth $Q(x)$, and have just a small fraction of sieve candidates left, which we check for factorization by trial division.

Finally, we have given hints on how to optimize the parameters of the algorithm, to make the algorithm run as fast as possible. Based on the choice of parameters, we have given a heuristic analysis of the complexity of the algorithm. The structure of the composite $n$ may cause the collection of smooth relations to take much longer than expected. Also, the solutions in the kernel may not lead to a non-trivial factor, and we might have to remove some relations and collect more. Thus, the complexity analysis is a heuristic, rather than a rigorous proof.
Chapter 6

The Multiple Polynomial Quadratic Sieve

In Chapter 5 we presented an algorithm which extended the idea of Dixon’s Random Squares algorithms of Chapter 4, and is more complex than Pollard’s Rho method of Chapter 3. The Quadratic Sieve presented in Chapter 5 can be summarized in the following steps:

1. Set the polynomial \( Q(x) = (x + m)^2 - n \) where \( m = \lfloor \sqrt{n} \rfloor \) and \( n \) is the composite number.

2. Compute a factor base \( S \) of \(-1\) and primes \( z \) for which \( \left( \frac{n}{z} \right) = 1 \).

3. Initialize a sieve array \( L \) indexed by \([-M, M]\) with zeros.

4. For each prime \( z \) in the factor base, solve \( Q(x) \equiv 0 \pmod{z} \). If \( r_1 \) and \( r_2 \) are solutions, then add \( \log z \) to \( L[r_i + l z] \) satisfying \(-M \leq r_i + l z \leq M\) for some integer \( l \) and \( i \in \{1, 2\} \).

5. Check the value \( Q(x) \) for those sieve array indices \( x \), that have accumulated some minimum value, for factorization by trial division over the factor base, until \(|S| + 1\) smooth \( Q(x) \) have been found.

6. Save the parity of the prime factorization exponents of the smooth \( Q(x) \) values found, as rows in a matrix \( A \).

7. Use solutions in \( \ker(A^T) \) to try and find a non-trivial factor of the composite.
In this chapter we will introduce a variant of the Quadratic Sieve, which is very similar to that presented in Chapter 5.

6.1 Motivation

In the original Quadratic Sieve algorithm, we only use one polynomial $Q(x)$. Naturally, this $Q(x)$ becomes numerically larger when $x$ increases, given the nature of the polynomial. In Section 5.1 we gave an argument which indicates, that as $Q(x)$ grows in size, the chance of it being smooth over the factor base decreases. In other words, the larger sieve array we use, the less likely are we to find smooth $Q(x)$ values for those $x$ that are in the beginning and the end of the array.

That is just the problem with the Quadratic Sieve described in Chapter 5. As we might not find all the smooth $Q(x)$ values we need for our matrix in the first sieving round, we might have to expand our sieving array again and again until we have collected enough values. But as the sieve array grows, the gain in smooth $Q(x)$ per increased array size decreases.

What we would really like is, to combine the sieving technique with small sieve arrays, since we know smooth values are more dense in the small sieve arrays. The problem is, there might not be enough smooth values in a small sieve array, to use in our matrix. The answer to our problem, as we shall see, is to use multiple polynomials.

6.2 Multiple polynomials

In the original Quadratic Sieve algorithm we use a polynomial,

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n.$$ 

Suppose instead we use polynomials of the form

$$q(x) = (ax + b)^2 - n$$

$$= a^2x^2 + 2abx + b^2 - n,$$

where $a$ and $b$ are integers. Again, we will be using a sieve array $L$, indexed with values in some interval $[-M, M]$. The graph of (6.1) is a parabola, which has vertical axis of symmetry in $(-2ab)/(2a^2) = -a/b$. If we choose $a$ and $b$ satisfying $0 \leq b \leq a$, this axis of symmetry will be between $x = -1$ and $x = 0$, which helps making the largest value of $q(x)$ as small as possible. The largest value of $q(x)$ for $x \in [-M, M]$ will be approximately $a^2M^2 - n$ for
the $x$ values in the beginning and end of the sieve array. The $2abx$ and $b^2$ terms will be negligible compared to the others. The smallest value of $q(x)$ will approximately $-n$, namely when $x = 0$.

Remember, that it is exactly the absolute value of $q(x)$ in the given sieving interval we would like to be smooth over our factor base. Thus, it makes good sense to try and minimize this value. Obviously, the choice of the coefficients $a$ and $b$ for our polynomial, will have an impact on this. We will attend this choice next.

6.3 Choice of coefficients

To minimize the value of our polynomial $q(x)$, we may choose our coefficient $a$, such that the minimal and maximal values have about the same absolute value. As stated above, the minimal value is fixed about $-n$, but we can solve for $a$ such that the maximal value becomes close to $n$:

\[
\begin{align*}
a^2M^2 - n &\approx n \\
\Leftrightarrow a^2 &\approx \frac{2n}{M^2} \\
\Leftrightarrow a &\approx \sqrt{\frac{2n}{M}}. 
\end{align*}
\] (6.2)

Figure 6.1 shows, that with our coefficient $a$ chosen as above we get the maximal and minimal value of $q(x)$ in our sieve interval to be approximately the same absolute size, thus optimizing the polynomial for sieving.

![Figure 6.1](image)

**Figure 6.1** Plot of $q(x)$ with $n = 1000076001443$, $M = 100000$, $a = 14 \approx \sqrt{2n/M}$ and $b = 7$. 
What about the coefficient \( b \)? The only hint we have given for \( b \) so far, is that it should satisfy \( 0 \leq b \leq a \). Suppose we choose \( b \), such that after determining \( a \), \( a \) divides \( b^2 - n \):

\[
b^2 - n \equiv 0 \pmod{a} \\
\Leftrightarrow b^2 - n = ac, \text{ for some } c \in \mathbb{Z}.
\] (6.3)

With this constraint, the polynomial \( q(x) \) in (6.1) becomes,

\[
q(x) = a^2x^2 + 2abx + b^2 - n \\
= a^2x^2 + 2abx + ac, \text{ inserting (6.3)} \\
= a(ax^2 + 2bx + c).
\] (6.4)

We have now shown how to choose the coefficients \( a \) and \( b \), and in doing so we ended up with \( q(x) \) being of the form (6.4). Please note, that with the constraints of (6.3), the coefficient \( c \) is given by \( c = (b^2 - n)/a \). From (6.4) we note, that if we let \( v = (ax + b) \) and \( u = a(ax^2 + 2bx + c) \), then we have a relation of the form \( v^2 \equiv u \pmod{n} \). Now if \( q(x) \) is smooth over the factor base, it proves an excellent relation for the matrix we will solve, to find a non-trivial factor of \( n \), just as described in the Quadratic Sieve of Chapter 5.

Assume we choose \( a \) to be a square, i.e. \( a = s^2 \) for some \( s \), then we may write (6.4) as

\[
q(x) = s^2(ax^2 + 2bx + c).
\] (6.5)

If the term \( ax^2 + 2bx + c \) is then smooth over our chosen factor base, we gain a relation given by (6.6), by using the fact that \( a \) is a square:

\[
\left((ax + b)s^{-1}\right)^2 = q(x)/a \\
= ax^2 + 2bx + c.
\] (6.6)

This allows us to lower the size of our polynomials, which will eventually be used in the sieving process, by a factor \( a \). This will greatly reduce the time spent on doing factorization by trial division on the sieve candidates. From here on, we will let \( Q(x) = q(x)/a \), so the relation we use becomes

\[
\left((ax + b)s^{-1}\right)^2 \equiv Q(x) \pmod{n}.
\] (6.7)
6.3. CHOICE OF COEFFICIENTS

From (6.3) we know, that we should choose $b$ s.t. $b^2 \equiv n \pmod{a}$. However, since $a$ is not a prime, we cannot use Shanks-Tonelli’s algorithm[Sti05] to compute a modular square root of $n$, like suggested in Chapter 5. However, if we choose $s$ to be a prime, s.t. $(\frac{2}{s}) = 1$, we can use Hensel’s lifting lemma of Section 2.5 to compute the modular square root of $n$, as shown below.

Let the polynomial for Hensel’s lifting lemma be $f(x) = x^2 - n$. We can use Shanks-Tonelli’s algorithm for computing a modular square root, $r$, of $n$ modulo $s$, rather than $s^2$. We assume $r \neq 0$, since this would mean $n \equiv 0 \pmod{s}$. In other words, we assume $s$ is not a non-trivial factor of $n$. Thus, we get

$$f(r) \equiv 0 \pmod{s}.$$  

Taking the derivative of $f(x)$, we get $f'(r) = 2r$, which is clearly not 0 modulo $s$. Thus, according to Hensel’s lifting lemma, we can find a number $t \in \mathbb{Z}_s$, s.t. $f(r + ts) \equiv 0 \pmod{s^2}$, with $t$ given by

$$tf'(r) \equiv -(f(r)/s) \pmod{s}.$$  

Thus, we get

$$tf'(r) \equiv -(f(r)/s) \pmod{s}$$

$$t2r \equiv -\left(\frac{r^2 - n}{s}\right) \pmod{s}$$

$$t \equiv -\left(\frac{r^2 - n}{s}\right)(2r)^{-1} \pmod{s}. \quad (6.8)$$

Having found $t$, a solution to $f(x) \equiv 0 \pmod{s^2}$ is $r + ts$. Remember, we wanted $b$ to satisfy $b^2 - n \equiv 0 \pmod{s^2}$, which is exactly the same as saying $b$ should be a solution to $f(x) \equiv 0 \pmod{s^2}$. Thus, by setting $b = r + ts$, it will satisfy (6.3).

For the Quadratic Sieve we did not discuss how to choose the sieve length parameter, $M$. This was because we were working only with one polynomial. If we did not find enough values of $Q(x)$ that were smooth over the factor base in the sieve array, we would simply have to expand the array size and try the new values. With the Multiple Polynomial Quadratic Sieve, however, we may simply choose another polynomial, and keep the sieve length constant. We do this exactly as described in this section, but we make sure we do not use the same $a$, $b$ and $c$ again.
### 6.4 Choice of sieve length parameter $M$

We will be sieving on an array of length $2M + 1$, just like with the Quadratic Sieve. Obviously, if we choose $M$ very large, we will have a lot of $Q(x)$ values to sieve on, but these residues will grow rapidly as $x$ grows, which is undesirable. Remember, the reason for using multiple polynomials is to keep the residues small.

On the other hand, if we choose $M$ very small, we will not get many $L[x]$ values to sieve on. One might say this does not matter, since we can always just generate a new polynomial and get new residues of approximately the same size. The problem here is, that it takes time to generate the coefficients for the new polynomial. Also, every time we switch to a new polynomial, we must figure out for each prime $z$ in the factor base, which values of $Q(x)$ are divisible by $z$, before we start sieving. We call this the *initialization stage*.

If we choose $M$ too small, we will be spending most of the time initializing with a new polynomial, which is quite wasteful. Silverman\[Sil87\] gives a table of good values of $M$ and factor base size $|S|$, for various sizes of $n$. Those values are given in Table 6.1.

Next, let us describe how to compute the roots for our new polynomials.

### 6.5 Computing polynomial roots

A consequence of using $Q(x)$ polynomials of a different structure than those of the Quadratic Sieve is, that we need to figure out in a different way, for each prime $z$ in the factor base, which $Q(x)$ is divisible by $z$.

Recall, that for the Quadratic Sieve algorithm, where we had $Q(x) = (x + m)^2 - n$, we computed the roots for each prime $z$ in the factor base, simply
as \((\pm t - m)\), where \(t\) is a modular square root of \(n\) modulo \(z\).

With the MPQS, our polynomial has the form \(Q(x) = ax^2 + 2bx + c\), referring to (6.7) for the relation. We wish to find out, for each prime \(z\) in the factor base, which \(x\) give a \(Q(x)\) which is divisible by \(z\). Again, we may use the same trick as we did with the Quadratic Sieve in (5.7). In (6.9), assume that \(r\) is a solution to \(Q(r) \equiv 0 \pmod{z}\), that is \(z\) divides \(Q(r)\).

\[
Q(r + lz) \equiv a(r + lz)^2 + 2b(r + lz) + c \pmod{z} \\
\equiv a(r^2 + (lz)^2 + 2rlz) + 2br + 2blz + c \pmod{z} \\
\equiv (ar^2 + 2br + c) + a((lz)^2 + 2rlz) + 2blz \pmod{z} \\
\equiv Q(r) + a((lz)^2 + 2rlz) + 2blz \pmod{z} \\
\equiv 0 \pmod{z}, \text{ assumes } z \mid Q(r). \quad (6.9)
\]

This shows again, that if we find a solution to \(Q(x) \equiv 0 \pmod{z}\), for each prime \(z\) in the factor base, we actually find a lot of \(Q(x)\) values in our sieving interval, which are divisible by \(z\).

To find the roots of \(Q(x)\) modulo \(z\), we may simply use the general formula for the roots of a quadratic formula \(ax^2 + bx + c\), which is given by \(r = (-b \pm \sqrt{b^2 - 4ac})/2a\). Thus, for our polynomial \(Q(x) = ax^2 + 2bx + c\), we will have roots

\[
\begin{align*}
    r &\equiv -2b \pm \sqrt{(2b)^2 - 4ac} \pmod{2a} \\
    &\equiv -2b \pm \sqrt{4b^2 - 4ac} \pmod{2a} \\
    &\equiv -2b \pm \sqrt{4b^2 - 4ac} \pmod{2a} \\
    &\equiv -2b \pm \sqrt{4b^2 - ac} \pmod{2a} \\
    &\equiv -b \pm \sqrt{b^2 - ac} \pmod{2a} \\
    &\equiv (-b \pm \sqrt{n}) a^{-1} \pmod{2a}, \text{ since } b^2 - ac = n \\
\end{align*}
\]

Note that the existence of a modular square root of \(n\) modulo \(z\), is guaranteed by the restriction on primes in the factor base, namely that \((\frac{n}{z}) = 1\), for each prime \(z\) in the factor base.

From (6.10), we see that we need to compute the inverse of \(a\), for each prime \(z\) in the factor base. Remember, that \(a = s^2\), for some prime \(s\), which satisfies \((\frac{n}{z}) = 1\). This means, that \(s\) has a chance of actually being one of
the primes in our factor base. Thus, we can not compute the roots of $Q(x)$ (mod $s$), since $a$ does not have a modular inverse in $\mathbb{Z}_s$, because $a \equiv 0 \pmod{s}$.

For the case where $s$ is in the factor base, we will check all the $Q(x)$ values in the sieve interval, and see if $s$ divides it. If it does, we add $\log s$ to the corresponding $L[x]$.

Next, let us attend which bound to use for defining sieving candidates, in the MPQS.

### 6.6 Choice of sieving candidate bound

For the Quadratic Sieve algorithm of Chapter 5, we described in Section 5.4 how to choose the bound on which $L[x]$, for which we would consider the associated $Q(x)$ value for factorization by trial division.

Recall that we chose our coefficients, s.t. $q(x) = a(ax^2 + 2bx + c)$ had a maximum absolute value of about $n$. However, we sieve on the polynomial $Q(x) = ax^2 + 2bx + c$, and we have chosen $a \approx \sqrt{2n/M}$. Thus, by dividing by $a$, we get that the expected maximum absolute value of the $Q(x)$ residues are approximately

\[
\frac{n}{a} = \frac{n}{\sqrt{2n/M}} = M \frac{n}{\sqrt{2n}} = M \sqrt{n/2}. \tag{6.11}
\]

Referring to Figure 6.1, we expect that the sieve values on indices $-M$, 0 and $M$ are approximately the logarithm of (6.11), i.e. $\log(M\sqrt{n/2})$. In [Con97], Scott Contini suggests using a sieve bound $\log(M\sqrt{n})$, minus a small error term. This is to make up for the fact that we do not sieve with prime powers, that we change polynomials thus increasing the size of the residues, and that this bound is for the maximum possible residues. The bound given by Contini is the one we will use in our implementation of the algorithm, which we shall describe in Chapter 7.

One very appealing consequence of using multiple polynomials is, that it allows for parallelization of a part of the algorithm, as we will discuss this next.

### 6.7 Parallelization

The data collection phase, where relations of the form $((ax+b)s^{-1})^2 \equiv Q(x) \pmod{n}$ are collected, relies on some polynomial $Q(x)$. But since we have now
6.8. **SUMMARY**

In this chapter, we have shown how we can alter the Quadratic Sieve algorithm of Chapter 5 into one which uses several different polynomials, rather than a single one. This means, we can make our sieving array much smaller, by effectively choosing a smaller \( M \). The consequence of this is, we will be working with only small residues, relative to those of the QS algorithm. If we do not find enough relations in the first sieving round, we switch to a new polynomial and sieve again, before the residues get too large. This was not possible with the normal QS algorithm.

We have shown how to choose the coefficients for these polynomials, and explained what constraints they should meet. Furthermore we have explained how to compute the roots of the new polynomials, allowing us to use the sieving method which made the QS algorithm superior to Dixon's Random Squares algorithm of Chapter 4. We have given values for the sieve array length based on the size of the composite \( n \), as suggested by Silverman\cite{Sil87}.

The MPQS algorithm we have introduced lends itself well to parallelization, in the data collection phase. This means that multiple machines can sieve with different polynomials at the same time, giving a noteworthy speedup of the algorithm.
In this chapter, we will discuss our implementations of the algorithms described in this thesis. Implementations have been done for the Pollard’s Rho method of Chapter 3, for the Quadratic Sieve of Chapter 5: one without sieving and one with sieving. Also, implementations have been done for the Multiple Polynomial Quadratic Sieve algorithm of Chapter 6: one which does not utilize parallelization, and one that does.

### 7.1 Technicalities

The algorithm implementations for this thesis, have all been done using the C++ programming language. C++ is an object oriented low-level programming language, which extends the C language. Memory manipulation is very similar to that of C, allowing code optimization on a deeper level than in higher-level programming languages, like Java.

The nature of the algorithms, and indeed their purpose as a whole, namely factoring integers that are the product of two large primes, puts some requirements on what we need to be able to do with the programming language. A 32-bit unsigned integer, for example, can hold a maximum value of $2^{32} - 1 = 4,294,967,295$, which is not nearly enough for our application. We would like to factor composites with a lot of more digits.

For this purpose, a C++ library called NTL\(^1\) by Victor Shoup is used in our implementations. NTL offers ready-to-use data structures for arbitrary length integers called ZZ objects. This means, we may use the data structures of NTL in our implementations as needed, to store e.g. the large composites

\(^1\)http://www.shoup.net/ntl/ (accessed 07-06-2010).
we want to factor, polynomial values, etc. NTL also offers data structures for vectors and matrices over arbitrary size integers as well as finite fields and algorithms for manipulating them. These are useful for certain parts of the discussed algorithms.

7.2 Pollard’s Rho method

Comparing Pollard’s Rho method of Chapter 3 to the Quadratic Sieve of Chapter 5 and Multiple Polynomial Quadratic Sieve of Chapter 6, it is easily realized, that Pollard’s Rho method is the simplest. Thus, it will also be easier to implement.

Our implementation requires one or two parameters. If two parameters are specified, the first must be a file containing one composite \( n \) per line. The second parameter is a file to which program output will be written, called the log file. If only one argument is specified, this will be used as log file, and the composite \( n \) is requested input by the user at runtime.

For the log file, an existing file can be used, in which case the output from several runs of the program can be accumulated in a single log file. Thus, we run the program as

\[
\text{pollards [input_file] log_file},
\]

where \([\text{input_file}]\) means an optional parameter.

The pseudo-random function of the algorithm, \( f(x) = x^2 + 1 \), is implemented as a single function. The NTL library offers functionality to compute \( x^2 \) by means of a \texttt{power} function. The NTL library readily offers an implementation of the gcd operator with the \texttt{GCD} function, which is an implementation of Euclid’s algorithm (see e.g. [MvOV01]), that we will use. For purposes of timing, NTL offers a function \texttt{GetTime()}, which returns a double-precision number of the seconds spent by the process. This function is used extensively throughout all our algorithm implementations, in order to compare algorithm results in Chapter 8.

As mentioned, the implementation of the Pollard’s Rho method is very straightforward. The entire program is written in a single C++ file. For the source code, see Appendix D.

We will move on directly to more interesting algorithms, implementation wise, namely the Quadratic Sieve and the Multiple Polynomial Quadratic Sieve. While Pollard’s Rho method uses constant memory, the QS is much more complicated to implement.
We will begin by describing how we implemented the most basic version of the Quadratic Sieve, namely the single polynomial version which does not use sieving. Gradually, we describe how more functionality is built on top, and end up with the distributed MPQS implementation which uses a client/server model to parallelize the data collection phase of the algorithm.

7.3 The Quadratic Sieve

First, we turn our attention to the factor base of the Quadratic Sieve algorithm. We know it should consist of primes $z$ below some bound $B$, for which $\left(\frac{n}{z}\right) = 1$, for our composite $n$. In Chapter 5 we gave, with the Contini bound in (5.14), an estimate on an optimal smoothness bound for the factor base as $B = e^{0.5 \sqrt{\ln n \ln \ln n}}$. For e.g. $10^{70}$, we get $B \approx 1,641,126$. Thus, it is unlikely we will ever need a prime in our factor base, which can not be contained in an unsigned 32-bit integer with a maximum value of 4,294,967,295.

With this reasoning, we can safely choose our factor base to be an array of unsigned 32-bit integers. C++ offers the vector type, which is like an array that resizes dynamically as need be, which we have chosen to use. In conjunction with PrimeSeq, a class for generating primes from NTL, we may keep generating primes $z$ in increasing size, put those that satisfy $\left(\frac{n}{z}\right) = 1$ into the factor base vector, and stop when we have reached the bound $B$. NTL offers also a function Jacobi(n,z), which computes the Legendre symbol $\left(\frac{n}{z}\right)$ when $z$ is prime.$^2$. This makes it easy for us to compute the factor base.

The next thing we need, is an implementation of the function $Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n$. For NTL, operators have been implemented, meaning the library can understand e.g. adding two arbitrary precision integers, using the plus symbol. However, using the regular operators, e.g. +, -, * and / leads to creation of temporary variables, which take up memory. To circumvent this, NTL offers procedural versions of most operators, so e.g. one may use add(x,a,b) to save a+b in variable x, instead. This has been used in our implementations whenever possible, for optimization. The function which computes $Q(x)$ has been overloaded, so one function returns the result as a ZZ object, and one takes a pointer to the variable as a parameter, in which to store the result.

The next data structure from NTL we make use of, is the vec ZZ class. It offers a structure similar to vector, but is optimized for arbitrary size integers of type ZZ. We use two instances of this data structure to store the values of

$^2$The Jacobi symbol reduces to the Legendre symbol which we defined in Definition 12, when the second argument is a prime.
(x + m) and Q(x), whenever we find a smooth Q(x). We add the values to both vectors at the same time, and since vec_ZZ preserves ordering, we are guaranteed the pairs do not get mixed up.

Recall the matrix we denoted A in our treatment of the Quadratic Sieve. This is the matrix that keeps the parity of the factorization prime exponents of smooth Q(x) values, one for each row. For this matrix, NTL offers a very suitable data structure called mat_GF2, which implements matrices over GF(2) = Z_2. The nice thing about using this structure is, that the class automatically makes sure that all operations we perform on the matrix results in matrices that are also over Z_2, so we do not have to handle this ourselves.

We can index the mat_GF2 matrices directly, using e.g. A[x][y] to get the element in row x, column y of A. If we just index A once, we will get a vec_GF2 element out, representing an entire row of A. We may then pass a pointer to this row to an implementation of a function test_smoothness. This function attempts to factor a given candidate Q(x) over the factor base by trial division. The test_smoothness function stores the parity of the found prime exponents in the given row of A. If the Q(x) is smooth over the factor base, the next row of A is used the next time, until all rows of A have been filled. Otherwise, the same row is used again for a different candidate Q(x), which are chosen in the order x = 0, ±1, ±2, ....

For the row of A passed to test_smoothness, a corresponding row of a mat_ZZ object E is also passed. We do this, because at some point we will need the full factorization of the smooth Q(x) again, not only the parity of the exponents. To avoid computing the factorization by trial division again, we store it for later use in the matrix E. Note that E saves the full exponent of the factorizations, while A only saved the parity. Thus, if we reduce E modulo 2, we will get A.

When enough smooth Q(x) have been found to fill up A and E, we need to solve a matrix kernel. Fortunately, NTL has a function kernel readily available for both mat_GF2 and mat_ZZ. Inspection of the source code, reveals that the kernel function uses elementary Gaussian reduction on the matrix. Naturally, we want to use the one that operates on mat_GF2. The kernel function of NTL does not, however, solve the kernel as we defined it in Section 2.4. Instead, the NTL implementation solves xA = 0. The only consequence of this is, that we do not need to transpose our A matrix before computing the kernel, which only makes things easier.

After computing the kernel, we may systematically try every combination of vectors in it, to gain a non-trivial factor of n, as described in Section 5.2. This involves computing the variables x and y, and finally using the GCD function of NTL to compute GCD(x-y,n). Just as with the implementation of Pollard’s
Rho method, the program has been made, such that one or two arguments must be specified, with an optional input file, that requires one composite \( n \) per line. The C++ source code for this implementation is found in Appendix E.

Figure 7.1 gives a rough sketch of the flow of the implemented QS algorithm. It shows how a composite \( n \) is input and different \( Q(x) \) values are checked for smoothness, until enough relations to compute the kernel of our matrix have been found. We try different solutions in the kernel to factor \( n \). Either we succeed or we must remove some relations from the matrix and look for more smooth \( Q(x) \) values. We will move on to describe the implementation of the sieving procedure.
CHAPTER 7. IMPLEMENTATION

7.4 Sieving implementation

As we recall from Chapter 5 on the Quadratic Sieve algorithm, the sieving procedure should allow us to gather a lot of smooth \(Q(x)\) values without attempting factorization by trial division on many non-smooth \(Q(x)\).

Regarding the sieving array \(L\), it should be clear that using an array of regular 32-bit integers should suffice. We do not need arbitrary size integers here. Recall that the upper bound on the sieve values are \(\log(Q(x))\), which is bounded above by \(\log n\), the number of bits used to represent \(n\).

For the QS with sieving, we do not use a fixed sieve array length as we do with the MPQS. If we do not find enough smooth \(Q(x)\) values with the initially chosen sieve array length \(2M + 1\), we will have to increase \(M\), thus increasing the sieve array length. However, for optimization purposes we require not having to keep expanding our sieve array. Instead, we would like to use the same memory that we used in previous sieving rounds. We can do this, because we do not need to sieve on indices that have already been sieved.

The risk of having to go back and perform sieving again, complicates things implementation wise, compared to the QS without sieving. Thus, we have chosen to implement the QS with sieving as a class. This simplifies the implementation a lot, since we may use class variables, rather than having to pass pointers around to the different functions.

For the sieving, we keep two sieve arrays of length \(M\) each: one for the negative indices and one for the positive indices. We also keep a variable which holds the start sieve index for the next sieve round. This will be 1 for the first round, \(M + 1\) for the next round, etc. Using two functions, we convert between indices that fit into the two sieve arrays of length \(M\), and the indices we input to \(Q(x)\). Doing this, we can keep sieving on \(2M\) values each round, but actually keeping the memory usage constant. Note, that by doing this, we skip sieving on \(Q(0)\), to make implementation easier.

A further optimization has been implemented. After each sieve round, we gather all sieve candidates in a queue. We use these sieve candidates to try and fill our matrix. After filling the matrix, there might still be sieve candidates left. In case the solutions in the kernel do not yield a factor of \(n\), we make sure to use the rest of these candidates before sieving again. This way, we do not throw away any sieve candidates from previous sieving rounds, before doing another round.

Remember, that when we introduce sieving, we also need to compute two roots, i.e. solutions to \((x + m)^2 - n \equiv 0 \pmod{z}\) for each prime \(z\) in the factor base. With the QS algorithm, this was quite easy: if \(t\) is a modular square root of \(n\), the two roots are \((\pm t - m)\). As we mentioned in Section 5.3, we
7.5. MULTIPLE POLYNOMIALS

...can use Shanks-Tonelli's [Sti05] algorithm to compute such a modular square root. However, NTL offers a function SqrRootMod which is readily available to compute such modular square roots. In the beginning we store the value of $t$ above for each prime $z$ in the factor base in a vector. This way, we avoid computing the roots again in case we have to go back and expand the sieve array. The implementation of QS that utilizes sieving is found in Appendix F.

We will move on to describe how we implement the use of multiple polynomials.

7.5 Multiple polynomials

For implementing the use of multiple polynomials, the changes to the program are actually not very radical. Recall that when we switch to a new polynomial, we keep the same factor base. In other words, the implementation of the factor base we have covered so far still suffices.

For the implementation of the MPQS, the demand of being able to switch polynomials, means we should re-use the variables for the polynomial coefficients $a$, $b$, and $c$ for optimization purposes. Thus, it is natural to implement the MPQS as a class on its own. This way, the polynomial variables, factor base, etc. can be implemented as class variables, thus reusing those that need to be altered, to minimize memory usage.

Polynomial computation

For computing the coefficient $a$, recall that we want $a \approx \sqrt{2nM}$, and that $a = s^2$ for some prime $s$ for which $n$ is a quadratic residue. We start by computing the value $\sqrt{\frac{2n}{M}}$. We then find the next prime greater than or equal to this value, s.t. $n$ is a quadratic residue modulo this prime. We call this prime $s$ and compute $a = s^2$. We find the primes using PrimeSeq from NTL and test them with the Jacobi function, just as when we compute the factor base. If we need to switch to a new polynomial at some point in the algorithm, we simply start checking for a new value of $s$ by looking for primes greater than $s$ satisfying the same property. This guarantees we do not re-use polynomials we already sieved with.

To compute the coefficient $b$, recall that $b$ is completely fixed by $n$ and $a$. Thus, after computing $a$, there is only one solution for $b$. We use Hensel’s lifting lemma as described in Section 6.3 to compute $b$. For the purpose of computing modular inverses, NTL offers the function InvMod$(x, a, n)$ which computes $x = a^{-1} \pmod{n}$.
Remaining is just to compute and store \( c = \frac{(b^2 - n)}{a} \). From here on, the change of polynomial from QS to MPQS is easy. We just change the function that computes the polynomial values. Specifically we change it to computing \( Q(x) = ax^2 + 2bx + c \). When we sieve and find a smooth \( Q(x) \), this is the value we save in a vector along with \((ax + b)s^{-1}\), thus giving us the relation \((ax + b)s^{-1})^2 \equiv Q(x) \pmod{n}\). Another thing we need to change, when switching to multiple polynomials, is the sieving procedure.

**Sieving with multiple polynomials**

Actually, the sieving procedure when using multiple polynomials, is simpler to implement than when using a single polynomial. When we were using a single polynomial, we risked having to go back and expand the sieving interval, if we did not find enough smooth \( Q(x) \) values in the first run. When using multiple polynomials we have set a fixed value of \( M \), thus we will have a constant sieving interval size.

Thus, we simply compute the roots as described in Section 6.5, and use these to sieve with the current polynomial. After sieving and checking all sieving candidates for smoothness by trial division, if we still have not collected enough smooth residues, we switch to a new polynomial as explained above, compute the roots again, reset the sieving array and sieve again.

Next, we will go through the implementation of the distributed MPQS algorithm.

### 7.6 Distributed MPQS

As we mentioned in Section 6.7, the MPQS algorithm lends itself well to parallelization, because the sieving procedure for each polynomial is independent from the others. We have done an implementation of a distributed MPQS, which is partitioned into two different programs: a server program and a client program.

The server program handles determination of the parameters \( B \) and \( M \) given some input composite \( n \), since these are constant in the MPQS. The client program can receive these data from the server program, in order to initialize the factor base and sieving array.

Upon request from a client program, the server program generates a polynomial of the MPQS algorithm, making sure the same polynomial is not generated twice. The server program will send this polynomial to the client program that requested it. The advantage of this is, that several clients...
may simultaneously prompt the server program for a polynomial. Generating
the polynomials is fast compared to the sieving process, so in this way $C$
client programs may work on $C$ different polynomials generated by the server
program, at the same time.

This idea is illustrated by Figure 7.2, which is a sequence diagram in UML\(^3\)
notation. The dotted vertical lines are called object life lines. They illustrate
the events happening at the associated objects. Figure 7.2 shows two client
programs, Client 1 and Client 2, and one server program Server. First,
the composite $n$ is input to the server program, after which it computes the
parameters $M$ and $B$. The client programs send the message HELLO to the

\(^{3}\)Please see e.g. [FS03] for more on the UML standard.
server program, to which the server program responds by sending the data $n$, $B$ and $M$. Afterwards, the client programs send a message \texttt{REQ. POLYNOMIAL} to the server program, thus making the server program generate the next polynomial and return the coefficients $a$, $b$ and $c$ to the requesting client.

![Figure 7.3 Distributed sieving and sending of smooth relations.](image)

After receiving a polynomial from the server program, the client program has all the data it needs to start the sieving procedure, and look for smooth polynomial values. After the process on the client side is over, it can send any smooth relations of the form (6.7): \[ ((ax + b)s^{-1})^2 \equiv Q(x) \pmod{n} \] that were found, back to the server program. The client will keep asking the server program for a new polynomial, use it for sieving and send back the smooth relations, until either the server program tells the client that no more relations are needed or the client program shuts down.

In continuation of Figure 7.2, the behavior of two client programs receiving
polynomials, using them to sieve, and send back the smooth relations to the server program, is illustrated in Figure 7.3. Please note, that for simplicity, each client program in Figure 7.3 only sieves with one polynomial. In practice, each client program would loop between asking for a polynomial and receiving it as illustrated in Figure 7.2, and using this polynomial for sieving and sending the smooth relations back to the server, as illustrated in Figure 7.3.

Figure 7.3 assumes that after receiving data from two different client programs once, the server program has enough relations to attempt to factor $n$. After receiving this data, the server program computes the kernel of the matrix with the smooth relations and attempts to use it for factoring $n$. In the case where the server program can not factor $n$ with the relations received, it removes some relations and waits for more new relations from the client programs, which is in compliance with the previously described implementations.

**Client/server model**

The implementation of the distributed MPQS algorithm uses a modified client/server model. The traditional client/server model consists of a server application providing a service for the client applications. The idea of the model is to partition workload between the server application and the client applications.

The model used in this implementation of the distributed MPQS algorithm, is a modified version of the traditional client/server model. The basic idea is as described above, where a central server application services requests from the client applications. However, rather than using the client/server model to implement a traditional service, the setup used here is more mutual, in the sense that the server program serves polynomials for the clients, while the clients also offer a service for the server program in the sieving phase. In our model, it is the server that is the program in focus, while the client programs are merely helpers in the goal of factoring the composite $n$.

For the implementation of this model, we need to use network communication. The communication method of choice for an application like this is sockets using the TCP (Transmission Control Protocol). Using TCP allows computers in a network to communicate under two important guarantees: *stability* and *ordered delivery*. Stability guarantees that messages that are sent over the network will eventually be delivered. Thus, detection of message loss is incorporated in the protocol, so our implementation can assume delivery of the messages we send, which is an important assumption. The guarantee of ordered delivery guarantees that messages we send will be delivered in the same order that we sent them. This is also an important guarantee in our
application, because otherwise a client program could e.g. receive a polynomial from the server to use, before receiving the initial data \( n \), \( M \) and \( B \).

For our implementation of the distributed MPQS algorithm, the C++ library Boost\(^4\) has been used. Boost has classes for many purposes, including ones for TCP sockets communication, under the common name Boost::Asio. These classes allow for communication between two machines in a network, via message sending and message receiving functions. Let us move on to show what the structure of the implementation looks like.

**Class structure**

While for two of the previously described implementations, we introduced the use of classes for improved organization of the data, the distributed MPQS is much larger and requires abstraction into several classes. In the implementation we clearly distinguish between classes belonging to the server program, and those belonging to the client program. Let us start by describing the server side program.

**Server program structure**

The server program has a class `mpqs_polynomial`, which handles generation of a new polynomial. Each time the server must generate a new polynomial for a client, it creates a new object of this type, passing the minimum bound on \( s = \sqrt{\alpha} \) as an argument. The object handles determination of the actual coefficients. Next, it manages one `mpqs` object, which is where the composite \( n \) is contained, the next bound on \( s \) is held, etc. It is also this class that handles inserting relations received from clients into a matrix and solving the kernel in the end.

For the socket communication, the server program uses one object of the type `mpqs_server`, which handles several connections to client programs. The `mpqs_server` object itself handles multiple `mpqs_session` objects. Each of these represent a single socket connection to one client program. Here, communication going in and out on the socket is handled. The received messages are passed to an object of the type `server_protocol`. The `server_protocol` parses the messages and acts accordingly. A class `network_message` has been implemented to keep control of the messages sent across the socket. This class is used by both the client program and the server program. An interface called `protocol_consts` defines the types of messages that can be delivered between the client and the server. Upon receipt of a message, checks are made using

\(^4\)http://www.boost.org/ (accessed 12-06-2010).
this interface to detect the type of message received. Finally, a main method in the file `mpqs_host` initializes the whole server program and starts listening for incoming connections. The server program is run as

```
```

We now turn our attention to the client program structure.

**Client program structure**

The structure of the client program is very similar to that of the server program. For the algorithm part, it uses one object of type `mpqs_sieve`, which handles computation of factor base, sieving and checking candidates for smoothness.

On the communication side, the client program does not need to handle connections to multiple computers. Thus, we only use one object of the type `mpqs_client` to handle the communication with the server program, both incoming and outgoing. Upon receipt of a message from the server, this object passes the message to an object of the type `client_protocol`. Just as with the server program, this object parses the message received, and acts upon that message. Typically, this will consist of setting a new polynomial in the `mpqs_sieve` object and starting the sieving process. To start the initialization of the client’s communication to the server, a main method in the file `mpqs_helper` attempts to start a connection on the provided host name and port. The program is run as

```
client host_name port.
```

For the source code of the server and client programs, see Appendix H.

### 7.7 Optimizations

The implementations described in this chapter can be optimized in different ways. There is one optimization in particular, which is used by de facto implementations of the Quadratic Sieve algorithm and its variations.

Let us consider the matrix, where we collect the parity of the prime exponents for the smooth $Q(x)$ values. The matrix is over $\mathbb{Z}_2$, thus we need just a single bit of memory for each entry. For large $n$, it is not unreasonable that our matrix will need about $100,000^2$ entries. Thus, we will need about $10,000,000,000$ bits $\approx 1192$ MB memory for the matrix. Even though modern computers come with more than this amount of memory, it is still a lot to process.
Several methods exist, that can help us reduce the size of this matrix, and speed up the process of finding the kernel. It is a fair assumption that most of the smooth $Q(x)$ we find for our matrix can be factored with just a small part of the factor base. Even for those primes that are used in the factorization of $Q(x)$, the exponent might be positive. In these cases, a 0 will be stored in the matrix for that position. Thus, most of the entries in the matrix will be zeros. Such a matrix is called *sparse*.

The Block Wiedemann algorithm[Wie86] is one method of solving sparse linear equations over finite fields. In [Cop94], Don Coppersmith describes how to use the Block Wiedemann algorithm to solve sparse linear equations over $GF(2)$, which is what we need. Another method has been proposed by Peter Montgomery called the Block Lanczos algorithm[Mon95].

The discussion of these algorithms is beyond the scope of this thesis. However, we might mention an obvious observation regarding the kernel of the matrix with the smooth numbers. The observation is, that if any column $j$ in the matrix contains a 1 on row $i$ and the rest of the positions are zeros, there is no way row $i$ can ever be used in a linear combination to give the zero vector. This follows from the fact, that any linear combination using row $i$ would always yield a 1 on position $j$.

The methods described above can greatly reduce the amount of memory needed for the matrix, by implementing it using a sparse matrix data structure. This has been utilized in e.g. the MPQS implementation msieve\(^5\).

Chapter 8

Computational results

In this thesis we have presented different factorization algorithms. The first was Pollard’s Rho method which, according to our analysis, will find a factor of $n$ in $O(n^{1/4})$ steps, for composites of the type we are working with. The expected number of steps is, however, a heuristic rather than a rigorous proof. This is due to the fact, that we used expectations regarding pseudo-random functions in our analysis.

We have also presented four different algorithms in the Quadratic Sieve family, where the last three are extensions of the original Quadratic Sieve. The last and most advanced is a distributed Quadratic Sieve algorithm that uses multiple polynomials, and attempts to find a factor by dividing up the data collection phase in a network of computers.

In Chapter 7, we described the implementation of these algorithms. In this chapter, we will present the computational results found, using these implementations. We introduce short names for the algorithms as follows:

- QS: Original Quadratic Sieve without sieving.
- QSSieve: Quadratic Sieve with sieving.
- MPQS: Multiple Polynomial Quadratic Sieve.
- DMPQS: Distributed Multiple Polynomial Quadratic Sieve.

In order to test the five program properly, we have generated six input files. Each file contains 100 composites $n = p \cdot q$, where $p$ and $q$ are primes of about the same size. We have generated files for composites of 10, 20, \ldots, 60
digits. These input files were given as parameter to each program, letting the programs write the factors and time spent to the log file. The machine used for computing the results provided in this chapter has a Quad-Core AMD Opteron 2200 MHz processor and 8 GB of memory.

First, we turn our attention to Pollard’s Rho method.

### 8.1 Pollard’s Rho

As mentioned, we expect that for the type of composites in our input files, the PR algorithm will find a factor in about $O(n^{1/4})$ steps. Figure 8.1 shows, for a range of composites, the theoretically expected number of steps for the algorithm, as well as the actual needed number of steps. We see, that in general, the actual required number of steps is lower than the theoretical upper bound.

![Figure 8.1 Steps required for Pollard’s Rho method for various composites $n$.](image)

<table>
<thead>
<tr>
<th>Digits of $n$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0011</td>
</tr>
<tr>
<td>20</td>
<td>0.1897</td>
</tr>
<tr>
<td>30</td>
<td>68.8095</td>
</tr>
<tr>
<td>40</td>
<td>22415.2678</td>
</tr>
</tbody>
</table>

**Table 8.1** Average factorization time for Pollard’s Rho method.

Table 8.1 shows the time spent by Pollard’s Rho method to factor the composites of the first four input files. Remember, that the type of composites
factored give the method the worst-case expected running time, because there are just two factors of about size $\sqrt{n}$.

We see that factorization of 10- and 20-digit composites is very fast, i.e. less than a second on average. For 30-digit composites, the average time to find a factor is just above one minute. From Table 8.1, we see that a drastic increase in running time occurs when going from 30 to 40 digits. The average time spent to factor a 40-digit composite was about 6 hours. Note, that not all hundred 40-digit composites were factored, we had to stop the computation before it was finished. An attempt was made to factor a 50-digit composite, but the computation was stopped after almost 2 days with no result.

We move on to present the results of our computations with the remaining implementations.

## 8.2 Quadratic Sieve and variations

For the implementations in the family of Quadratic Sieve algorithms, we have built in a switch on one important parameter. This switch means, that we can change between using a bound of the factor base size $|S|$ and a smoothness bound $B$ for the factor base, as described in Section 5.4.

We began by attempting to factor the smallest input composites, i.e. those with 10 digits, using the Contini bound with the QS implementation. We quickly realized, that using this bound would result in QS running for an extremely long time. As speculated, this turned out to be a result of a too small factor base. For example, for the first 10-digit composite attempted in the input file used, $n = 1601760403$, QS would run for a unreasonably long time, much longer than Pollard’s Rho method, without finding a solution. This trend was supported by the QSSieve, MPQS and DMPQS implementations, confirming the small factor base as a culprit.

Using the Contini bound in this case, gave us just 7 primes in the factor base, excluding $-1$. However, when switching to the Landquist bound, solutions were found instantly, with a factor base size of 17, excluding $-1$. This experiment was repeated for $n$ with more digits for all implementations, and the same result seemed to emerge: the factor base provided by using the Contini bound seems to be too low in practice, for the small composites.

The factor base size is of course a tradeoff, since we can never know the optimal value before running the program several times for the same input. However, our results indicate, that it is a good idea in practice to aim high rather than low, when choosing the factor base size, because smooth $Q(x)$ will be much more common. Even though we have to collect more smooth
$Q(x)$, the benefit from having a larger factor base cancels out the time spent to collect the extra relations and solve the larger matrix, and then some. Note that all results for the DMPQS have been acquired, using one server and 16 clients, run on machines similar to those mentioned above.

<table>
<thead>
<tr>
<th>Digits of $n$</th>
<th>QS</th>
<th>QSSieve</th>
<th>MPQS</th>
<th>DMPQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.09</td>
<td>192.45</td>
<td>0.15</td>
<td>0.08</td>
</tr>
<tr>
<td>20</td>
<td>21.92</td>
<td>29.39</td>
<td>0.47</td>
<td>0.17</td>
</tr>
<tr>
<td>30</td>
<td>2276.30</td>
<td>2806.52</td>
<td>6.54</td>
<td>1.06</td>
</tr>
<tr>
<td>40</td>
<td>95314.30</td>
<td>29208.57</td>
<td>109.31</td>
<td>12.88</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>1382.43</td>
<td>150.16</td>
</tr>
<tr>
<td>60</td>
<td>-</td>
<td>-</td>
<td>15158.60</td>
<td>2636.68</td>
</tr>
</tbody>
</table>

**Table 8.2** Average factorization time in seconds for Quadratic Sieve family algorithms, using the Landquist bound.

Table 8.2 summarizes the results of factorization of the input composites, for the four Quadratic Sieve family algorithms using the Landquist bound. The factorization times presented are the average times spent in seconds. Note that for some algorithms, not all composites were factored, as is the case for e.g. QS and QSSieve. Taking for example the average time of QS for factoring 40-digit composites, the average time is about 26 hours. Of course this may be very inaccurate, given the low number of composites actually factored, but using this average we expect that factoring all one hundred 40-digit composites will take about 110 days.

An interesting observation regarding Table 8.2 is, that the computation time for QS is superior to that of QSSieve, for composites of 10 to 30 digits. Our implementation of the QSSieve is supposed to be faster than the QS. We suspect that the result seen here, is due to the way sieving is implemented in the QSSieve. Recall that for the QSSieve, we have to expand our sieve interval, if we do not find enough smooth $Q(x)$ in the first sieving round. As we described in Chapter 7, we implemented this by using two vectors of constant size, combined with two conversion methods to convert between ”real” indices in $[-M,M]$, and those that fit into the two arrays. We suspect that the extra computation associated with index conversion, each time we wish to add a value to $L[x]$ for some $x$, is to blame for this unexpectedly high computational time.

Table 8.3 shows the computational results for the implementations of Quadratic Sieve family algorithms, using the Contini bound. Comparing the results using the Contini bound, to those using the Landquist bound of
8.2. QUADRATIC SIEVE AND VARIATIONS

<table>
<thead>
<tr>
<th>Digits of $n$</th>
<th>QS</th>
<th>QSSieve</th>
<th>MPQS</th>
<th>DMPQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.18</td>
<td>0.07</td>
<td>341.02</td>
<td>1.09</td>
</tr>
<tr>
<td>20</td>
<td>204.02</td>
<td>1945.88</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>30</td>
<td>2874.87</td>
<td>3946.90</td>
<td>8.28</td>
<td>5.79</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>36727.20</td>
<td>101.61</td>
<td>54.19</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>1070.79</td>
<td>106.93</td>
</tr>
<tr>
<td>60</td>
<td>-</td>
<td>-</td>
<td>11421.83</td>
<td>1306.79</td>
</tr>
</tbody>
</table>

Table 8.3: Average factorization time in seconds for Quadratic Sieve family algorithms, using the Contini bound.

Table 8.2, we see that generally the primitive QS and QSSieve are much slower when using the Contini bound. The QSSieve in special is much slower for 20-digit composites, when using the Contini bound, with an average time of 1945 seconds as opposed to 29 seconds using the Landquist bound.

The average time of 0.07 sec. for the QSSieve found in Table 8.3 is not very reliable: after several hours of computation, we had factored just one single value. We tried several times with other 10-digit composites, but the program seemed to stall on these values. We suspect this is a result of too small factor bases for the 10-digit composites, when using the Contini bound. We reach this conclusion, because the trend does not occur when using the Landquist bound.

Our results of Table 8.3 show, that also the more advanced variations, MPQS and DMPQS, are slower when using the Contini bound, for the small composites. Again, we suspect this is because of the too small factor bases, for the 10- and 20-digit composites. The very high average time for the MPQS with the 10-digit composites, 341 seconds, is the result of one extreme outlier which took 9772 seconds to factor. This shows, that the particular structure of a composite, combined with a too small factor base, may result in extremely long running time.

An interesting observation however, is that for composites of 50 or more digits, the MPQS and DMPQS with the Contini bound, are faster on average than when using the Landquist bound. This trend seems to support the idea, that for up to about 40 digits, the Contini bound yields a too small factor base. However, for larger composites, the Contini bound gives a larger factor base than when using the Landquist bound, which is in compliance with Figure 5.4b. Again, this supports our theory, that aiming for a larger factor base is beneficial in practice. For our testing of the DMPQS, we used 16 client programs. Our results show that the DMPQS is a lot faster than the
CHAPTER 8. COMPUTATIONAL RESULTS

MPQS. For the 60-digit composites, we gained approximately 9 times speedup using 16 clients. However, this is not close to the theoretical maximum for 16. This is due to the fact, that a part of the total time is spent handling network communication, which is not the case for the MPQS implementation. Also, a noteworthy amount of time is spent solving the matrix kernel, which is not parallelized.

Figure 8.2 shows a comparison of the factorization time, between the two implemented multiple polynomial version of the Quadratic Sieve. Note, that the 2. axis is on a logarithmic scale. We clearly see that the DMPQS algorithm is much faster than the MPQS algorithm on average. The plot also shows a definite improvement in factorization, when switching from the Landquist bound to the Contini bound, when $n$ becomes sufficiently large. This is due to the larger factor base for the Contini bound, when $n$ is about 40 digits, as we argued in Section 5.4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{factorization_time.png}
\caption{Factorization time for multiple polynomial variations.}
\end{figure}
Conclusion

In this thesis, we have presented different algorithms for integer factorization. One is the randomized Pollard’s Rho method, which is a special-purpose algorithm, in that the running time depends on the size of the smallest factor. Thus, Pollard’s Rho method is well suited for composites which have a small prime factor. We have presented the Quadratic Sieve algorithm, and described various modifications to the original algorithm, that result in faster factorization in practice.

We presented implementations of the algorithms discussed. These implementations were written in C++, using the NTL library supporting arbitrary precision integers. The most complex implementation involved using a distributed Multiple Polynomial variation of the Quadratic Sieve. The implementation utilizes the fact, that the data collection phase of the algorithm lends itself well to parallelization. The workload is partitioned among any number of computers forming a network using sockets communication.

The implementations were tested using input composites with 10, 20, ..., 60 digits, in order to compare the running time of the algorithms. On the basis of the results, we conclude that the simplest algorithms, namely Pollard’s Rho method and the original Quadratic Sieve, are the best for composites up to 20 digits. For larger composites, our results show that the Quadratic Sieve algorithms using multiple polynomials, are far superior. Especially, the distributed Multiple Polynomial Quadratic Sieve performed well for the largest composites.

The implementations of the Quadratic Sieve family algorithms contain a switch, which allowed us to change between two factor base size parameters. Comparison of these bounds showed, that the larger factor base size using one
bound is superseded by the other for composites about $8 \cdot 10^{40}$.

Meanwhile, our computational results show, that for composites up until about $10^{50}$, using one bound gives better factorization times than when using the other. When our composites become larger than about $10^{50}$, switching to the other bound will find a factor faster. On the basis of these results, we conclude that for the family of Quadratic Sieve algorithms, choosing a too large factor base is better than choosing it too small. The greater availability of smooth residues, given the extra primes in the factor base, more than makes up for the extra number of relations required, as well as the larger matrix for which we must compute the kernel.

By parallelizing the data collection phase in our implementation of the distributed Multiple Polynomial Quadratic Sieve, we were able to gain approximately a 9 times speedup using 16 clients for the 60-digit composites. This is compared to the same algorithm which is not parallelized in any way. For the small composites up to 20 digits, however, none of the algorithms in the Quadratic Sieve family beat Pollard’s Rho method.
Algorithm 3 Factoring by trial division.

1. Input: Factor base $S$
2. if $value < 0$ then
3.   $value \leftarrow -value$
4.   $vector[0] \leftarrow 1$
5. else
6.   $vector[0] \leftarrow 0$
7. if $value \neq 0$ then
8.   for $i = 1$ to $|S|$ do
9.     $z \leftarrow S[i]$
10.    $e \leftarrow 0$
11.    while $value \equiv 0 \pmod{z}$ do
12.       $value \leftarrow \frac{value}{z}$
13.       $e \leftarrow e + 1$
14.    if $value = 1$ then
15.       return true
16. return false
Proof that \((\frac{n}{z}) = 1 \Rightarrow x^2 \equiv n \pmod{z}\) has two solutions

Assume \((\frac{n}{z}) = 1\) for a prime \(z\), that is \(n\) is a quadratic residue \((\pmod{z})\). Thus,

\[
\begin{align*}
  x^2 &\equiv n \pmod{z} \\
  \iff x^2 &\equiv y^2 \pmod{z} \\
  \iff x^2 - y^2 &\equiv 0 \pmod{z} \\
  \iff z | x^2 - y^2 \\
  \iff z | (x - y)(x + y).
\end{align*}
\]

Hence, there must be exactly two solutions to \(x^2 \equiv n \pmod{z}\). Furthermore, if \(t\) is one solution, then \(-t\) is the other.
Appendix C

Maple code for generating composites

```maple
p:=nextprime(1000000): q:=nextprime(p):
for i from 0 to 100 do
  p:=nextprime(q): q:=nextprime(p):
  n := p*q:
end do:
```

Listing C.1 Loop generating composites of two primes of approximately same size.
Appendix D

Pollard’s Rho method source code
Quadratic Sieve without sieving
source code
Appendix F

Quadratic Sieve with sieving source code
Appendix G

Multiple Polynomial Quadratic Sieve source code
Appendix H

Distributed Multiple Polynomial Quadratic Sieve source code
Bibliography


