



YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

MASTER THESIS

**SIEVE ARRAY SCANNING TECHNIQUES FOR
NUMBER FIELD SIEVE ALGORITHM**

OZAN MURAT

THESIS ADVISOR: ASST.PROF. HÜSEYİN HIŞİL

COMPUTER ENGINEERING

PRESENTATION DATE: 14.08.2018

BORNOVA / İZMİR
SEPTEMBER 2018

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Jury Members:

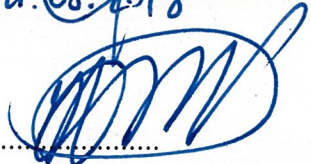
Dr. Öğr. Üy. Hüseyin HIŞIL
YAŞAR University


Prof. Dr. Urfat NURİYEV
EGE University

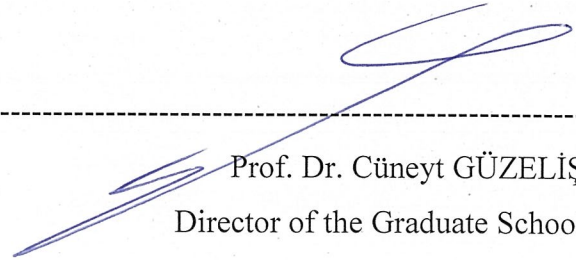
Dr. Öğr. Üy. Serap ŞAHİN
İZMİR INSTITUTE OF TECHNOLOGY
University

Signature:


14.08.2018


14.08.2018


14.08.2018


Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

SIEVE ARRAY SCANNING TECHNIQUES FOR NUMBER FIELD SIEVE ALGORITHM

Murat, Ozan

Msc, Computer Engineering

Advisor: Asst.Prof. Hüseyin HIŞIL, Ph.D.

September 2018

In this thesis, we investigate the lattice sieving step of the celebrated General Number Field Sieve (GNFS) algorithm. In particular, we focus on the fast determination of smooth entries in the sieved lattice and their conversion to exponent vectors. There are several ways to accomplish this step. We provide our experiments and discuss the impact of our modifications.

Key Words: Quadratic Sieve Algorithm, Number Field Sieve Algorithm, Integer factorization, Sieving, Line sieve, Lattice sieve.

ÖZ

SAYI CİSMİ ELEK ALGORİTMASI İÇİN ELEK DİZİSİ TARAMA TEKNİKLERİ

Murat, Ozan

Yüksek Lisans, Bilgisayar Mühendisliği

Danışman: Yrd.Doç.Dr. Hüseyin HIŞIL

Eylül 2018

Bu tezde, Genel Sayı Cismi Eleği (GNFS) algoritmasının kafes eleği adımı incelenmiştir. Özellikle, elekten geçirilmiş bir kafeste yer alan düzgün (smooth) girdilerin ve bu girdilerin üstel vektör şeklinde ifadesinin hızlı şekilde hesaplanmasına odaklanılmıştır. Bu hesaplamaların yapılması için birçok alternatif yol bulunmaktadır. Bu konuda deneyler sunulmuş ve sonuçlarının etkileri tartışılmıştır.

Anahtar Kelimeler: Kuadratik Elek Algoritması, Sayı Cismi Eleği Algoritması, Çarpanlara Ayırma, Elek, Çizgi Eleği, Kafes Eleği.

ACKNOWLEDGEMENTS

First I would like to express my gratitude to my supervisor Dr. Hüseyin Hışıl for his encouragement, patience and guidance during the study.

I would also like to thank my dear friend F. Bedirhan Yıldız for his support and friendship.

I would also like to thank my colleagues in Yaşar University Department of Computer Engineering for their motivation.

Finally, I would like to express my special thanks to my dear family for their endless love, patience, encouragement and support.

Ozan Murat

İzmir, 2018

TEXT OF OATH

I declare and honestly confirm that my study, titled “SIEVE ARRAY SCANNING TECHNIQUES FOR NUMBER FIELD SIEVE ALGORITHM” and presented as a Master’s Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Ozan Murat

Signature



.....

September 10, 2018

TABLE OF CONTENTS

FRONT MATTER	i
ABSTRACT	v
ÖZ	vii
ACKNOWLEDGEMENTS	ix
TEXT OF OATH	xi
LIST OF TABLES	xiv
LIST OF FIGURES	xv
LIST OF CODES	xvii
1 THESIS STATEMENT	1
1.1 MOTIVATION	2
1.2 THESIS GOALS AND CONTRIBUTIONS	2
1.3 OUTLINE OF THESIS	3
2 INTRODUCTION	5
2.1 INTEGER FACTORIZATION	5
2.2 BASIC CONSTRUCTION	7
2.3 SMOOTHNESS AND FACTOR BASE	8
2.4 POMERANCE'S SIEVE AND MONTGOMERY'S POLYNOMIALS	8
2.5 POLLARD'S ANALOGY FOR USING ALGEBRAIC INTEGERS	9
2.6 DISTRIBUTED RELATION COLLECTION	10
2.7 DISTRIBUTED/PARALLEL LINEAR ALGEBRA	10

3	THE NFS ALGORITHM	13
3.1	A QUICK REVIEW OF ALGEBRAIC NUMBER THEORY . . .	13
3.2	OVERVIEW OF NFS	14
3.3	THE POLYNOMIAL f AND AUXILIARY PARAMATERS . . .	16
3.4	FACTOR BASES	17
3.5	COLLECTING RELATIONS WITH SIEVING	19
3.6	NFS IN ACTION: A TOY EXAMPLE	26
4	ALTERNATIVE SCANNING METHODS	35
5	IMPLEMENTING NFS WITH ALTERNATIVE SCANNING	43
5.1	STRUCTURED DATA FOR HOUSE-KEEPING	44
5.2	PARAMATER SELECTION	45
5.3	POLYNOMIAL SELECTION	45
5.4	FACTOR BASE GENERATION	46
5.5	ALGEBRAIC FACTOR BASE	46
5.6	RATIONAL FACTOR BASE	47
5.7	SIEVING WITH: LINE SIEVING AND LATTICE SIEVING BY ROWS	47
5.8	SCANNING SIEVE ARRAY	49
5.9	EXPONENT VECTOR GENERATION AND STORING	50
5.10	LINEAR ALGEBRA WITH THE HELP OF MAGMA	50
6	CONCLUSION	53
	REFERENCES	54

LIST OF TABLES

4.1	Timing results for 100-bit number	37
4.2	Timing results for 150-bit number	38
4.3	Timing results for 200-bit number	39

LIST OF FIGURES

4.1	Test result for 100-bit inputs	37
4.2	Test result for 150-bit inputs	38
4.3	Test result for 200-Bit Input	39

LIST OF CODES

4.1	../elekmini/src/nfs/compute_nfs.c	35
5.1	../elekmini/src/_elekmini_nfs.h	44
5.2	../elekmini/src/_elekmini_nfs.h	44
5.3	project/magma_scripts.txt	45
5.4	../elekmini/src/nfs/init_poly.c	45
5.5	project/magma_scripts.txt	46
5.6	../elekmini/src/nfs/init_fb.c	46
5.7	../elekmini/src/nfs/init_fb.c	47
5.8	../elekmini/src/nfs/compute_nfs.c	48
5.9	../elekmini/src/nfs/init_fb.c	48
5.10	../elekmini/src/nfs/compute_nfs.c	48
5.11	../elekmini/src/nfs/compute_nfs.c	49
5.12	../elekmini/src/nfs/compute_nfs.c	49
5.13	../elekmini/src/nfs/compute_relations.c	50
6.1	../elekmini/src/_elekmini_nfs.h	59
6.2	../elekmini/src/nfs/init_data.c	60
6.3	../elekmini/src/nfs/init_fb.c	62
6.4	../elekmini/src/nfs/init_poly.c	64
6.5	../elekmini/src/nfs/compute_nfs.c	66

CHAPTER 1

THESIS STATEMENT

Integers have played a crucial role in the history of mathematics. Several algebraic constructions were built upon the abstraction of integers; e.g. as groups, rings, modules, fields, etc. Therefore, a deeper understanding of integers has always fascinated several number theorists, algebraists, mathematicians, computer scientists, and engineers. One of the most primitive properties of integers is its unique factorization property. The fundamental theorem of number theory states that every integer can be factored into its prime factors in a unique way (up to reordering of the factors). Finding these prime factors drew increasing attention by mathematicians. Integer factorization started to become a more algorithmic research area after the advent of fast factorization methods. Cryptanalysis has also had an indispensable force on the improvements on integer factorization literature.

Number Field Sieve (NFS) is by far the most sophisticated integer factorization algorithm which has several main steps where each main step has its own minor steps. There are several variations of each step allowing implementers a variety of ways to code NFS algorithm according to their expectations and limitations of the underlying hardware. There is also vast literature on each one of these steps.

This thesis project grew out of an interest in learning the fastest integer factorization algorithm for large integers: the NFS algorithm. It was soon understood that a distributed software that runs an optimized NFS requires a long-lasting team effort and is clearly beyond the scope of this work. On the other hand, understanding the steps of the algorithm, implementation of single-core versions of those steps with a high-level mathematical tool, and finally localizing

on the optimization of a specific step of NFS with the target of improving was decided a plausible attempt. We built our motivation in this direction and stated our research question in the following section.

1.1 MOTIVATION

This master thesis aims to answer the following research questions.

Can we design and implement a fast scanning stage for NFS algorithm? How does different variations compare in performance with each other? What is the optimum strategy for detecting smooth integers?

In order to answer these research questions, we started by building a stand alone C implementation of NFS algorithm to cover all steps up to the linear algebra step. Our implementation gave us a solid play ground to test different variations of implementation and measure the speed-ups that are gained with modifications. Then, we applied comprehensive tests on the alternative methods that have potential to answer the target research questions.

1.2 THESIS GOALS AND CONTRIBUTIONS

This thesis sets the following goals in order to answer the research questions of Section 1.1.

- Learning integer factorization algorithms of exponential and sub-exponential time complexities. Implementing these algorithms in Magma language.
- Implementing comprehensive versions of QS and NFS in C language with assembly support at time critical parts.
- Reverse engineering the libraries: Cado-NFS (Team, 2017), GGNFS (Monico, 2005), msieve (Papadopoulos,).

- Applying the modifications that will answer the research questions in our home built software. Make comprehensive experiments and derive verifiable conclusions.

The thesis makes the following contributions:

- A basic C/Magma based QS and NFS implementations are built and tested.
- The lattice sieve method is implemented using the sieve by row technique.
- Six different scanning phase scenarios were implemented, tested, benchmarked, and compared.

1.3 OUTLINE OF THESIS

The thesis is organized as follows. Chapter 2 presents a background information on integer factorization. Chapter 2 is not meant to be a comprehensive treatment but rather depicts a story line with an emphasis on the development of NFS algorithm. Chapter 3 provides a detailed explanation of NFS algorithm mostly excluding the linear algebra phase. The chapter puts more weight on the discussion of the scanning step that comes right after the sieving step. The readers can find a up-to-date literature review on NFS in Section 3.2 of Chapter 3. Chapter 4 addresses the research questions provided in Section 1.1. Chapter 5 provides implementation details, speed experiments, and evaluations. Chapter 6 derives conclusions out of this thesis and states possible future research directions in the light of gained experience.

CHAPTER 2

INTRODUCTION

This chapter provides an overview of developments in integer factorization that finally lead to the Number Field Sieve (NFS) algorithm. The selected topics do not cover all algorithms but rather the most notable milestones. Nevertheless, we provide pointers to literature for interested readers.

This chapter is organized as follows. Section 2.1 introduces the notion of integer factorization, celebrated algorithms in this area of research, and the industry has driven motivations behind studying the integer factorization problem. Section 2.2 explains the basic idea behind Quadratic Sieve (QS) and NFS algorithms, with historic pointers. Section 2.3 states the notion of smoothness and factor bases whose construction is a preliminary step of both QS and NFS algorithms. Sections 2.4 and 2.5 explains the birth of QS and NFS algorithms. Section 2.6 briefly points to the fully parallel nature of relation collection phase. Section 2.7 summarizes flag point results in the linear algebra phase including fast nullspace computation algorithms on sparse-matrices, and their parallelization over computer clusters or distributed networks. Sections 2.6 and 2.7 are not in the scope of this thesis. They are only included for completeness of the discussion.

2.1 INTEGER FACTORIZATION

Composite integers are integers which contain more than one prime factors. The fundamental theorem of number theory ensures that every composite integer is a product of positive powers of primes. Furthermore, the prime powers appearing in this product are unique up to reordering.

Integer factorization is the act of determining each prime factor of a composite integer. This can be performed in polynomial-time if all of the prime factors are known to be very small in advance, by trial division algorithm. Such composite integers are called smooth integers. On the other hand, the story is not the same for non-smooth composite integers. For instance, no polynomial-time factorization algorithm has been known to date for a composite integer which is the product of two *suitably* selected large primes. Such a factorization is widely assumed to be intractable.

Over years, integer factorization has become a challenging area for researchers. For this reason, several algorithms have been developed for factoring small-, medium-, large-sized¹ integers. Chronologically, Quadratic Sieve (QS), Number Field Sieve (NFS) and Elliptic Curve Factorization method (ECM) are the most known algorithms for factorizing integers. QS was introduced by Carl Pomerance in 1982. Today, QS is known to be the fastest algorithm for factoring integers of medium-size. ECM was developed by Lenstra in 1985. ECM is typically used as a special purpose algorithm for factoring smooth integers. NFS was announced by Pollard in 1988. Among these algorithms, NFS is the fastest algorithm known to date for factorizing composite large numbers. The original version of NFS was used for factoring integers of a special form. Several improvements have been made on NFS over time and now NFS comes in two flavors: Special Number Field Sieve (SNFS) and General Number Field Sieve (GNFS). GNFS is the fastest known generic purpose algorithm and it is mostly used for factoring large integers that are a product of two large prime integers.

RSA is a famous cryptographic algorithm and the conjectured security of RSA is based on the hardness of the factorization problem. RSA algorithm was published by Ron Rivest, Adi Shamir and Leonard Adleman in 1977. After the introduction of RSA, integer factorization has become a challenging and popular

¹Roughly speaking, integers smaller than 2^{85} are small-sized; integers between 2^{85} and 2^{350} are medium-sized, and integers greater than 2^{350} are large-sized integers. These bounds heavily depend on the available computational technology, algorithmic improvements, implementations. Therefore, the bounds tend to change over time.

research area. Researchers have been working on improvements of factorization algorithms both mathematically and computationally. RSA uses a pair of keys (private key and public key). The private key must be kept a secret to decrypt the encrypted message. Without the knowledge of the secret key, attacking RSA cryptosystem is performed by factoring the underlying modulus n . The latest factorization record is known as RSA-768 (Kleinjung et al., 2010) which factored a 768 bit RSA modulus. Therefore, users typically prefer 1024, 2048 or higher bit moduli for securing their data.

2.2 BASIC CONSTRUCTION

Both QS and NFS, and their predecessors rely on a very basic principle in number theory due to Fermat. Fermat's method expresses an integer n as a difference of two squares, $x^2 - y^2$, with the pair (x, y) different from $((n+1)/2, (n-1)/2)$. Such a pair gives a nontrivial factorization $n = (x - y)(x + y)$ with $x \geq \sqrt{n}$. Finding such an x and y exhaustively leads to an exponential time algorithm. However, there are possible improvements which made subexponential complexities to be achieved. Kraitchik described an empiric result in 1920s to speed up the procedure by collecting suitable x_i producing the congruence

$$y^2 \equiv \prod (x_i^2 - n) \equiv \prod x_i^2 \equiv \left(\prod x_i \right)^2 \pmod{n}.$$

Dixon showed how to construct such a collection of x_i systematically by incorporating a predefined factor base, negative x_i values, and linear algebra. We do not give full details here since these techniques are very well-documented in the literature. The main emphasis that should be made here is that the most sophisticated methods of the modern era of factorization have their roots from historic results.

2.3 SMOOTHNESS AND FACTOR BASE

Kraitchik/Dixon method exploits the smoothness notion of integers. An analogous idea was later carried over to the construction of the NFS algorithm. Therefore, it is necessary to define this simple yet useful term, clearly.

Definition 2.3.1 *A positive integer is called B -smooth if all of its prime factors is smaller than or equal to B .*

We will extend this definition later when dealing with the NFS algorithm. When deciding whether a given number is B -smooth using a computer program, it is useful to have all primes smaller than equal to B precomputed, sorted, and stored in an array. Such an array of primes is called a factor base.

2.4 POMERANCE'S SIEVE AND MONTGOMERY'S POLYNOMIALS

Collecting relations for incremental values of x_i is cumbersome since each value $x_i^2 - n$ is to be factored and kept if smooth. Pomerance showed a method to quickly determine possibly smooth $x_i^2 - n$ by an ancient method named *sieving* whose working principle dates back to Eratosthenes.

Pomerance's original suggestion of using the sieving technique benefits from the single polynomial $x_i^2 - n$. However, the values of $x_i^2 - n$ increase quadratically as x_i moves away from linearly from \sqrt{n} . Montgomery brought a very useful solution to this problem by showing how to use other quadratic polynomials to collect relations. We skip details of Montgomery's improvement and its successors; and refer to (Pomerance, 1985), (D. Silverman, 1987) for a complete treatment. The collection of all improvements on Fermat difference of squares method to Pollard's 1988 work –see below– is today accepted in the nomenclature of Quadratic Sieve. QS is the fastest algorithm for factoring integers in range 100 to 350 bits.

2.5 POLLARD'S ANALOGY FOR USING ALGEBRAIC INTEGERS

NFS was developed by Pollard in 1988 to factor numbers of the special form $x^3 + k$. This idea first evolved in the special number field sieve (SNFS) that can only work on numbers of specific form $2^k + s$. Today, if the term of NFS used without any qualification, it refers to the general number field sieve (GNFS). We also mean GNFS when using the acronym NFS. GNFS is a generalization of the SNFS algorithm. It can factor any number apart from an integer N that is a power of a prime. It is the fastest known algorithm for factoring integers that are over 350 bits. NFS is shown to be practical for integers up to 768 bits. With improvements on NFS implementations, both bounds are extending over years leading a wider use of NFS.

NFS can be viewed as an improvement QS, which allows an integration of rings with the notion of smoothness besides \mathbb{Z} and $\mathbb{Z}/n\mathbb{Z}$ into the algorithm. The idea is to use the notion of smoothness in some suitably selected ring such as a ring of algebraic integers. Additionally, if there is a mapping between the ring and $\mathbb{Z}/n\mathbb{Z}$ then one can produce congruence of squares as in QS. We delay further details to Chapter 3. NFS factors composite n in heuristic expected time

$$\exp((c + o(1))(\log n)^{1/3}(\log \log n)^{2/3})$$

as $n \rightarrow \infty$, see (Lenstra et al., 1990). Here $c = (64/9)^{1/3} \approx 1.9$. The algorithm tries to find two perfect squares; one from \mathbb{Z} and the other from $\mathbb{Z}[\alpha]$ where α is an algebraic integer satisfying a polynomial with integer coefficients. One can then find a nontrivial factor of n using congruence of squares method with the help of a map from $\mathbb{Z}[\alpha]$ to \mathbb{Z} . The algorithm has 3 main processes: parameter and polynomial select, sieving and linear algebra.

2.6 DISTRIBUTED RELATION COLLECTION

Both in QS and NFS, the relation collection can be easily distributed over a TCP/IP network. Clients are typically responsible for finding relations individually and reporting to a central server. Since clients do not need to communicate with each other but only with the server, it is fairly easy to design an infrastructure with socket programming. In addition, multi-threaded clients can dedicate as many cores as needed without requiring additional software. Cado-NFS readily provides a comprehensively distributed relation collection layer. The main challenge here is to find enough number of clients that can dedicate their computational power. These issues are not within the scope of this work. Therefore, we skip further discussion here.

2.7 DISTRIBUTED/PARALLEL LINEAR ALGEBRA

The goal of the linear algebra step is to determine a suitable subset of the collected relations. This subset leads to the desired perfect squares that will eventually lead to the factorization by Fermat's difference of squares method. The relations are typically stored in a very sparse matrix over \mathbf{F}_2 . This matrix can contain around 10^{16} entries in a real-life challenge, where only $\approx 1/10^6$ of the entries are 1. Storing such a matrix requires special data structures that exploit the sparsity and fit all data in about 100GB of memory. We refer to (Kleinjung et al., 2010) for details.

More specifically, the linear algebra step computes the nullspace of the matrix. This is typically done with specially designed algorithms that can work efficiently with sparse matrices:

1. Block-Lanczos algorithm (Montgomery, 1995) for QS. Block-Lanczos

algorithm can be run in a parallel high-speed mesh network of processors. The matrix is smaller in comparison to NFS in the feasible region of the QS algorithm.

2. Block-Wiedemann algorithm (Coppersmith, 1994) for NFS, see cf. (Kleinjung et al., 2010). Block-Wiedemann algorithm is suitable for distributed networks. The matrix can be much larger than in QS. A major milestone in reducing the complexity of the algorithm was announced by Thomé in (Thomé, 2001).

The naive methods such as Gaussian elimination does not suit well with such computation since Gauss elimination will destroy the sparsity of the matrix and thus leading to an unfeasible memory requirement.

As in Section 2.5, distributed or parallel linear algebra algorithms are not in the scope of this work. Therefore, further discussion is omitted.

CHAPTER 3

THE NFS ALGORITHM

NFS is based on a concept introduced by John Pollard (Pollard, 1993a) in 1988, which is used to factor numbers of special form $x^3 + k$ where k is a small integer. Pollard's new technique made the use of an algebraic number field in factoring integers for the first time. Lenstra *et. al* (Lenstra et al., 1990) used this concept to build the Special Number Field Sieve (SNFS) algorithm which is applicable to integers of special form $r^e \mp s$ with $r, e, s \in \mathbb{Z}$ and $e > 0$. In 1990, SNFS (Lenstra et al., 1993) was used to factor the ninth Fermat number

$$F_9 = 2^{512} + 1 = p_9 \cdot p_{49} \cdot p_{99},$$

where p_9 , p_{49} , and p_{99} have 9, 49, and 99 decimal digits. The SNFS algorithm was later modified in order to factor arbitrary integers under the name General Number Field Sieve (GNFS) algorithm in (Buhler et al., 1993).

3.1 A QUICK REVIEW OF ALGEBRAIC NUMBER THEORY

Algebraic Number Theory has a vast literature which can easily shadow the main purpose of this thesis. Therefore, we give definitions that are relevant to efficient implementation of the NFS algorithm. An algebraic number is the zero of a polynomial with integer coefficients (Wagstaff, 2013). The algebraic number is called algebraic integer if the polynomial is monic. $\mathbb{Q}(\alpha)$ is the smallest algebraic number field containing the algebraic number α . $\mathbb{Z}[\alpha]$ represents the set of all algebraic integers in $\mathbb{Q}(\alpha)$. The algorithm uses a ring homomorphism

f which is defined from the set algebraic integers $\mathbb{Z}[\alpha]$ to \mathbb{Z}_n . In summary we have, $\mathbb{Z} \subseteq \mathbb{Z}[\alpha] \subseteq \mathbb{Q}(\alpha)$. $\mathbb{Z}[\alpha]$ is a Dedekind domain, and therefore, has unique factorization of ideals into prime ideals. These prime ideals are the building blocks of NFS algorithm, which are treated as smooth elements. We refer to (Cohen, 1993) and (Milne, 2009) for further reading. A homomorphism f from a ring R_1 to a ring R_2 is a function $f: R_1 \rightarrow R_2$ such that

$$f(1_{R_1}) = 1_{R_2},$$

$$f(r + s) = f(r) + f(s),$$

$$f(rs) = f(r)f(s)$$

for all $r, e \in R_1$ (Stewart and Tall, 2001). The algorithm uses ring homomorphism f from the algebraic integer $\mathbb{Z}[\alpha]$ to \mathbb{Z}_n . These concepts will be used in the thesis.

3.2 OVERVIEW OF NFS

We are now at a level to explain a high-level overview of NFS algorithm that can find a nontrivial factor of an integer n . NFS starts by selecting a polynomial

$$f(x) = c_0x_0 + c_1x_1 + \dots + c_dx^d \tag{3.1}$$

with $c_i \in \mathbb{Z}$. The polynomial f is chosen to satisfy two properties simultaneously:

- f is irreducible over \mathbb{Z} ,
- $f(m) \equiv 0 \pmod{n}$ for some $m \in \mathbb{Z}$.

Let α be a complex root of f . Then $\mathbb{Q}(\alpha)$ defines a number field that contains the ring $\mathbb{Z}[\alpha]$. Furthermore, there exists a natural ring homomorphism

$$\phi : \mathbb{Z}[\alpha] \rightarrow \mathbb{Z}/n\mathbb{Z}$$

that sends α to m . NFS tries to construct the set S of several pairs (a, b) with $a, b \in \mathbb{Z}$ that gives

$$\beta^2 = \prod_{(a,b) \in S} (a + b\alpha) \quad \text{and} \quad \prod_{(a,b) \in S} (a + bm) = y^2$$

simultaneously and therefore, leads to the congruence

$$\begin{aligned} x^2 &\equiv \phi(\beta) \cdot \phi(\beta) \equiv \phi(\beta^2) \equiv \phi\left(\prod_{(a,b) \in S} (a + b\alpha)\right) \\ &\equiv \prod_{(a,b) \in S} \phi(a + b\alpha) \equiv \prod_{(a,b) \in S} (a + bm) \equiv y^2 \pmod{n} \end{aligned} \tag{3.2}$$

for some $x \in \mathbb{Z}$. Such a congruence immediately yields the desired difference of squares $n = x^2 - y^2 = (x + y)(x - y)$. The factors $(x + y)$ and $(x - y)$ are proper factors with probability $\approx 1/2$.

In the above discussion, S is constructed from pairs (a, b) each of which is

- \mathcal{B}_1 -smooth for $a + b\alpha \in \mathbb{Z}[\alpha]$,
- \mathcal{B}_2 -smooth for $a + bm \in \mathbb{Z}$

simultaneously for some pair of smoothness bounds \mathcal{B}_1 and \mathcal{B}_2 on \mathbb{Z} and $\mathbb{Z}[\alpha]$, respectively.

NFS implementations splits the algorithm into the following steps

1. Selecting the polynomial along with auxiliary parameters
2. Forming the factor bases
3. Collecting relations with a suitable sieving method
4. Select relations which yield the perfect squares
5. Compute the square root
6. Find a proper factor of n

We give more details on each one of these steps. The main focus of this thesis is on the third item.

3.3 THE POLYNOMIAL f AND AUXILIARY PARAMETERS

The first step of GNFS is the polynomial selection. As in mention above, the reason the GNFS had outpassed the quadratic sieve in running time is using a ring other than $\mathbb{Z}/n\mathbb{Z}$, that is created by complex root α and irreducible polynomial. The most time-consuming step is the sieve step of the algorithm finding enough relation that to find non-trivial dependencies in the linear algebra step that will construct a difference of squares. If one can make the values $a + b\alpha$ and $a + bm$ more likely to be smooth then, the spend time in sieving step will be reduced. So, the selection of polynomial can affect running time of the sieving step (norm of the algebraic integer is defined by $N(a + b\alpha) = (-b^d)f(-a/b)$).

The polynomial selection methods can be divided into two groups. These are linear search and non-linear search methods. In linear methods, one can try to find two polynomial with one linear and other the higher degree, while non-linear method tries to find two non-linear polynomials. There are several linear search methods base- m method, Murphy's improvement of latter or Kleinjung's method. In this thesis, we discuss only method base- m .

The polynomial selection step aims to find a pair of polynomials, f and g which share a common root modulo n . The polynomial f is called *the algebraic polynomial* and g is called *the rational polynomial*. The linear polynomial is the g and higher degree polynomial is the f . The initial step to generate polynomial f is to select the degree of polynomial d . A simple method is to pick $d \approx ((3 \cdot \log n) / \log(\log(n)))$. In practice is usually between 3 and 10. However, its optimal value does slowly tend to infinity with n (Stevenhagen, 2008). After deciding the degree of the f , we can generate the polynomial f using the base- m

method. First m is selected to be close to $n^{1/d}$.

Definition 3.3.1 *The base- m representation of n is as follows:*

$$n = \sum_{i=0}^d c_i m^i$$

with $c_i < m/2$. We define the polynomials as follows;

$$f(x) = \sum_{i=0}^d c_i x^i, \text{ and } g(x) = x - m. \quad (3.3)$$

In this way, we have monic and linear polynomial $g(x)$ and nonlinear and monic polynomial $f(x)$. If the polynomial $f(x)$ is not reducible, it can be used to directly factor n , if that fails, one of the factors can be used instead of $f(x)$. For any given n to be factored, there may be a number of choices for f and integer m with $f(m) \equiv 0 \pmod{n}$. As mention above, the running time of finding relations heavily depends on a good choice of f . In real life implementations, it has been observed that a polynomial f can be accepted to be better than one other if more pairs (a, b) with $a + bm$ and $a + b\alpha$ are found to be smooth with respect to the same input parameters.

There are sophisticated techniques to determine a good polynomial. On the other hand, this is not in the scope of this thesis. Therefore, we skip further discussion here and refer readers to (Kleijnung, 2006), (B. A., 2010).

3.4 FACTOR BASES

The factor bases in the algorithm are used to test the smoothness of integers or algebraic integers. The algorithm has two distinct factor base that store primes up to bound. The algebraic factor base store the primes for testing smoothness of $a + b\alpha$ and rational factor base store primes for testing smoothness of $a + bm$. The factor bases are essential for finding smoothness relations for constructing

the matrix that store exponent vectors for every (a, b) are smooth over both algebraic and rational factor base.

Algebraic factor base. The algebraic factor base (AFB) consists of prime ideals of \mathcal{O} , especially, first-degree prime ideals are one with a small norm. The first step to construct AFB is to select bound \mathcal{B}_1 . Following this, one can construct pairs (r_i, p_i) that represent first degree prime ideals up to \mathcal{B}_1 .

Definition 3.4.1 *A first degree prime ideal \mathfrak{p} of a Dedekind domain D is a prime ideal of D such that $N(\mathfrak{p}) = p$ for some prime p .*

Theorem 1 (Briggs, 1998) *Let $f(x)$ be monic, irreducible polynomial with integer coefficients and $\alpha \in \mathbb{C}$ a root of $f(x)$. The set of pairs (r, p) where p is a prime integer and $r \in \mathbb{Z}/p\mathbb{Z}$ with $f(r) \equiv 0 \pmod{p}$ is in bijective correspondence with the set of all first degree prime ideals of $\mathbb{Z}[\alpha]$.*

Notice that a prime can occur up to d times in algebraic factor base because f may have d distinct root modulo p . The ideal $a + b\alpha$ is smooth over AFB if its factors completely over the primes of the AFB. Factoring over AFB is similar to process of factoring over RFB. However, we do not test if the ideal $a + b\alpha$ is smooth, but rather we test the norm of an ideal $N(a + b\alpha) = (-b)^d f(-a/b)$. If a and b coprime, $N(a + b\alpha)$ has factors p and $a \equiv -rb \pmod{p}$ for a unique root r of f modulo p then the prime ideal corresponding to the pair (r, p) divides ideal $a + b\alpha$.

Rational factor base. The rational side polynomial $g(x)$ with $d = 1$ so the algebraic number field $Q(\alpha_2)$ is simply Q hence, the first degree prime ideals in Q are the regular primes. The rational factor base consists of integer primes up to some bound \mathcal{B}_2 . Elements of rational factor base can also be expressed in pairs as in algebraic factor base. In this context, we compute and store pairs $(m \bmod p_j, p_j)$ up to \mathcal{B}_2 .

Quadratic characters. One problem with the norm of the algebraic number is that if the norm is square that does not imply that an algebraic integer is also square. To overcome this obstruct Adleman (Adleman, 1991) suggested the use of quadratic characters. When set of (a, b) pairs have been found, a further test is needed to determine whether or not the product of the corresponding elements $a + b\alpha \in \mathbb{Z}[\alpha]$ is a perfect square in $\mathbb{Z}[\alpha]$. This problem is solved by using ‘quadratic characters.’ An integer cannot be a square if it is not a square module some given prime. Using this fact the quadratic characters ensures that an algebraic number is also square. The quadratic character base is a small set of first-degree prime ideals of $\mathbb{Z}[\alpha]$ that do not exist in the algebraic factor base. The norm of an algebraic integer $a + b\alpha$ doesn’t guarantee that the algebraic integer is square in $\mathbb{Z}[\alpha]$. To solve this problem we add a few more entries to the exponent vector. Choose several prime q is greater than the largest prime p in algebraic factor base. For each of these q ’s find solution r to $f(r) \equiv (\text{mod } n)$ with $f'(r) \not\equiv 0 \pmod{q}$. For each prime q and each (a, b) pair, calculate the Legendre symbol $((a + br)/q)$ insert its value in the exponent vector and extend the linear algebra to ensure that

$$\prod_{a,b \in S} \left(\frac{a + br}{q} \right) = 1$$

for every q . The result set of S will very likely to be produced squares in the algebraic side.

3.5 COLLECTING RELATIONS WITH SIEVING

The aim of the sieving part is to find relations which correspond to pairs of integer pair (a, b) with $\gcd(a, b) = 1$ such that $f(a, b)$ and $g(a, b)$ are both smooth. The first step is to choose smoothness bounds \mathcal{B}_1 and \mathcal{B}_2 . $f(a, b)$ is considered

smooth if no prime exceeding \mathcal{B}_1 divides it (similar for $g(a, b)$). $f(a, b)$ or $g(a, b)$ values that divisible by prime p form a regular pattern in \mathbb{Z}^2 . There are two sieve methods for finding smooth (a, b) pairs. These are line sieve and lattice sieve. Lattice sieve has two different implementation: sieve by row and sieve by vectors. In linear algebra step, to find the factor of n , we need at least $\#\text{RFB} + \#\text{AFB} + \#\text{QCB} + 2$ smooth pairs.

Line sieve. The line sieve method has the same idea with sieving in the QS method. The difference is NFS has two separate sieving lines for rational and algebraic sides and two free variables a and b .

To see how sieving method can be used, let p be fixed prime in the rational factor base and fix some value for b with $b \geq 0$ to start sieving. After that any $a \in \mathbb{Z}$ p divides $(a + bm)$ if and only if $a + bm \equiv 0 \pmod{p}$. This refers that $a \equiv -bm \pmod{p}$ and thus a is necessary to be of the form $a \equiv -bm + kp$ for some $k \in \mathbb{Z}$. As a result, this gives possible (a, b) pairs that have $a + bm$ divisible by p for fixed integer b and prime p .

Sieving over the rational factor base begins with a ‘sieve array’ of computer memory with a single position allocated for each $-u < a < u$. In order to track that pairs are smooth, set up an array with one sieve value for every a -values then initialize every element in the array to 0. For each prime p from RFB, one computes the finite set of values $a = -bm + kp$ for $k \in \mathbb{Z}$ with $-u < a < u$. For each such a in the set that divisible by prime p , $\log_2 p$ will be added to the position corresponding to a in the sieve array. After this has been performed all primes in RFB, the sieve array is scanned for values which are $\log(a + bm)$ or close to $\log(a + bm)$ will be pairs (a, b) that may be smooth over RFB.

Sieving over the algebraic factor base progresses in exactly the same manner as mention for rational factor base. However, because of the representation of first degree prime ideals of $\mathbb{Z}[\alpha]$ as pair of integer (r, p) that divides ideal $a + b\alpha$ if and only if divides $N(a + b\alpha)$. This occurs if and only if $a \equiv -br \pmod{p}$.

So, (a, b) pairs with ideal $a + b\alpha$ is divisible by (r, p) must have a of the form $a = -br + kp$ for $k \in \mathbb{Z}$.

This procedure is then continued for the next value of b until sufficient pairs (a, b) have been found with $a + bm$ and $N(a, b)$ smooth to allow for the linear algebra step. Note that, if $b \equiv 0 \pmod{p}$ then there does not exist an (a, b) and $N(a + b\alpha) \equiv 0 \pmod{p}$, because the condition of a and b being coprime. If $b \equiv 0 \pmod{p}$, then $a \equiv 0 \pmod{p}$ thus, a and b have common factor that contradicting the condition of $\gcd(a, b) = 1$.

Lattice sieve. Lattice sieve was proposed by Pollard (Pollard, 1993b) for find smooth pairs (a, b) . Lattice sieve is a variation of ‘special q ’ method for QS of Davis and Holdridge (Davis and Holdridge, 1984) . Lattice sieving with proper parameter selection lets us consider only a fraction of all (a, b) pairs. Lattice sieve requires noticeably more memory than the Line Sieve method. However, in practice lattice sieving is more efficient than line sieving (for large integers). Lattice sieve starts with partitioning of the factor base into two parts. We define

S: the small primes: $p \leq B_0$,

M: the medium primes: $B_0 < p \leq B_1$

where the ratio between both bound should be between 0.1 and 0.5 as mention in (Pollard, 1993b). The algorithm also uses the large primes:

L: the large primes: $B_1 < p \leq B_2$

where B_2 is much larger than B_1 .

After that, we construct the set Q that store special q ’s. Prime pairs (q, s) are selected from M to construct Q . For any $v = (q, s) \in Q$ let L_v be the lattice with basis $(q, 0)$ and $(s, 1)$ (Golliver et al., 1994) in the (a, b) plane. L_v corresponds to the pairs (a, b) for that q divides $N(a - b\alpha)$ and $r = a/b \pmod{q}$ for $v \in Q$. To

carry out lattice sieving for special q pair, we calculate reduced basis to find two shortest vector $V_0 = (a_0, b_0)$ and $V_1 = (a_1, b_1)$ which generate the lattice. Then the a typical point of the lattice is: $c \cdot V_0 + e \cdot V_1$.

$$(a, b) = (c \cdot a_0 + e \cdot a_1, c \cdot b_0 + e \cdot b_1) \quad (3.4)$$

For accept point (a, b) as a smooth a and b need to be coprime. So, it is necessary to that c and e be coprime. However, it can happen that c and b coprime and $a \equiv b \equiv 0 \pmod{q}$. In that case, $q \mid a$ and $q \mid b$ satisfy our conditions.

The sieving region \mathcal{A} in the c, e -plane is chosen relatively small, usually (depending on size of the n) $-C \leq c < C$ and $0 \leq e < E$. Belonging $(c, e) \in \mathcal{A}$ represents the point in the lattice. Note that, negative values of b may produce by (3.4). If that occurs signs of a and b is changed. There is a two variation of lattice sieving: sieving by rows and sieving by vectors.

Lattice sieving by row similar to line sieving on the q -lattice. Let r be the solution $f(x) \pmod{q}$ for $q \in Q$. We initialize a sieve array S with size of $2 \cdot C \cdot E$ and initialize all values in sieve array to 0. To finding first location in the lattice that the factor base prime hits, one can solve for c in

$$\frac{ca_0 + ea_1}{cb_0 + eb_1} \equiv r \pmod{p} \quad (3.5)$$

with fixed e and prime pair $(r, p) \in \text{AFB}$. This process is applied to all primes in AFB that $p < q$. For the rational side, we sieve the number $a - bm$ with all the primes in RFB. The sieving by row is good for small primes, but bad for the larger primes one since no integers are to be sieved in many rows.

Sieving by vector consider the points on a p -sublattice of the q -sublattice. For

prime (p, r) from AFB, the basis for sublattice $L_{q,p}$ in the (c, e) plane is given by

$$U_1 = (p, 0) \text{ and } U_2 = \left(\frac{a_2 - r \cdot b_2}{r \cdot b_1 - a_1} \bmod p, 1 \right). \quad (3.6)$$

Then, we generate two short vectors which generate the lattice

$$V_1 = (c_1, e_1) \text{ and } V_2 = (c_2, e_2).$$

A typical point of lattice is: $d \cdot V_1 + f \cdot V_2$ i.e

$$(c, e) = (d \cdot c_1 + f \cdot c_2, d \cdot e_1 + f \cdot e_2).$$

As shown above, linear combinations of V_1 and V_2 give points (a, b) such that $pq \mid N(a, b)$. In the lattice sieve, the number of integers is sieved by lattice sieve is much less than the linear sieve by a factor:

$$W = \sum_{q \in M} \frac{1}{q} \approx \frac{\log(1/k)}{B_1}$$

and still collect most of the relations found by other method.

Franke and Kleinjung (Franke and Kleinjung, 2005) give the details of an efficiently compute the indices in the sieve. One can also use bucket sieving (Aoki and Ueda, 2004) to reduce cache misses for larger p 's.

Scanning the sieve array. In this step, the algorithm scanned sieve arrays for possible candidates for smooth (a, b) pairs. The candidates are decided by if the sieve array value of candidate pair (a, b) is larger than some predefined threshold. After that, these pair (a, b) is tested if it is smooth over factor base. The smoothness test can be done via ECM or trial division. Trial division is a process to determine if a non-zero integer k is B -smooth. One can say k is B -smooth if p divides k for all primes $p < B$, while replacing k with k/p . At the end of the process if $k = 1$ then k is B -smooth and accepted as a relation.

After trial division sometimes k may have value of $1 < k < B^2$ then one can apply large prime variation to k . Large prime factor should be larger than B but less than B^2 .

The candidate smooth pairs are determined by if sieve array values of greater than a threshold. Then, we construct $a + bm$ and $N(a + b\alpha)$ for smoothness test. Then, $a + bm$ and $N(a + b\alpha)$ can be tested for smoothness with using trial division function and factor bases. If both algebraic and rational side is passed the smoothness test we construct exponent vector. The first entry of vector refers the sign of the $a + bm$. After sign entry, #RFB entry represents exponents of primes that divide $a + bm$. #AFB entry is represent exponents of the primes that divide $N(a + b\alpha)$. At the end of the vector, we append #QCB entry that represents quadratic character values for $a + b\alpha$.

Filtering. The relations that are gathered by the sieving process may contain many duplicates (a, b) pairs. For find solution in linear algebra process, the duplicates will be removed. Duplicate relations are deleted first. Because those relations may lead to a trivial solution of the matrix that only trivial factorization. In practice, duplicate relations are inevitable because of the lattice sieving. In lattice sieve, many special q -primes will generate identical relations. Line sieving will not produce duplicates in theory. However, from interrupted and re-started sieving process or overlapping of values of b may produce duplicate relations. After duplicate relations removal, the number of relations will be decreased while the number of the factor base primes that occurs in relations stays the same. So, duplicate relations makes hard to predict the number of useful relations found by lattice sieving. The lattice sieving over an estimated range of special q values: in the small sampling data, the ratio of duplicates to unique relations will be very small, but in the complete data set, it is often as large as 30%, see (Kruppa, 2010).

Definition 3.5.1 *The excess as defined as the difference between the number of*

rows (relations) and the number of columns (ideals). The excess must be positive to find the solution in the linear algebra step.

The next phase of filtering is the deletion of relations containing singleton ideals. The aim is the finding a subset of relations that in the product of these relations every prime ideal occurs to an even power. So, relations containing a prime ideal in an odd power that does not occur another relation cannot be part of any solution and can be extracted from the matrix. So, by doing this, one creates a new singleton, and singleton removal can be repeated until no singleton relation remains. Each singleton relation deletion contains at least one prime ideal that occurs nowhere else in the other relations. So, the number of relation decrease by 1 the number of remaining prime ideals also decrease by 1 hence, the excess does not decrease in this step. At the end of the singleton deletion, if the excess is positive, then matrix could be built and solve.

Linear algebra. The aim of the linear algebra step is, founded a set of relations produced by sieving step, to obtain a subset S such that satisfies

$$\beta^2 = \prod_{(a,b) \in S} (a + b\alpha) \quad \text{and} \quad \prod_{(a,b) \in S} (a + bm) = y^2$$

To find such a subset S , sieving process must collect enough relations. Factorization of $a + bm$ in \mathbb{Z} and ideal $a + b\alpha$ into primes, modulo 2 of the exponent of the each primes construct the row vectors over \mathbb{F}_2 . These row vectors form a matrix that contains a row for each relations and a column for each prime which occurs in the relations. This produces a very sparse and large matrix, because of each relations has only a small number of primes. With, using linear algebra method as mention in 2.7 and find a dependency among the binary vectors corresponding (a, b) pairs.

Computing the square root. In this chapter, we describe the finding the square root part of the algorithm. There are several approaches to calculating

the square root of algebraic side integer. This problem has been solved since the beginning of the research for the GNFS. The most used algorithm date back to 1993, one of the included algorithms in “*The development of the Number Field Sieve*” (K. Lenstra and W. Lenstra, 1993) by Couveignes (Couveignes, 1993). Couveignes square root method base on Chinese Remainder Theorem (CRT), and is only applicable under the condition that the number field degree d be odd. the Another algorithm is proposed by Montgomery (Montgomery, 1994). The algorithm based on lattice reduction for calculating square roots in algebraic number fields. It uses little memory and it is fast but, the implementation of the algorithm is more complex than others. For the rational side, finding square root is a simple task, since we know the prime factorization of the product.

When the square root is calculated, the algorithm outputs the factor of the number n . However, sometimes β that constructed by the set of S founded in the linear algebra step does not yield a square or that the difference of squares yields a trivial factor (n or 1). For this reason, we seek more relations than the number of columns of the matrix so, we will have more than one solution for the linear equation.

Finding a proper factor of n . All process that are above to generate congruence of difference $x^2 \equiv y^2 \pmod{n}$ to find proper factor of n . Any other representation of n as $x^2 - y^2$ gives a nontrivial factorization $n = (x - y)(x + y)$ such that $x \geq \sqrt{n}$. If n is composite and not a prime power then there is at least $1/2$ chance that $\gcd_{x-y,n}$ and $\gcd_{x+y,n}$ are non-trivial factors of n .

3.6 NFS IN ACTION: A TOY EXAMPLE

It is helpful to see an example that exemplifies the different stage of the algorithm. This goal will be achieved by factorizing the integer 53743 using GNFS. Although numbers of such small integers would never be factored with using GNFS, this small integer makes a useful example.

The polynomial. We start with the choice of d , m , and polynomial $f(x)$. The first parameter to decide before factoring $n = 53743$ is the degree d of the polynomial $f(x)$ which the rest of the algorithm depends on it. A trivial method to pick a degree $d \approx (3 \log(n) / \log(\log(n)))^{1/3}$ for this particular n . Next, m should be chosen $m \approx n^{1/d}$, which in our case m should be around $53743^{1/3} \approx 37$. The base- m method and the value $m = 37$ is used in this example. The base- m expansion of n :

$$53743 = 37^3 + 2 \cdot 37^2 + 9 \cdot 37 + 19 \quad (3.7)$$

produces the polynomial $f(x) = x^3 + 2 \cdot x^2 + 9 \cdot x + 19$ for that $f(m) \equiv 0 \pmod{n}$ because $f(x)$ was constructed with $f(37) = 53743$.

Factor bases. Algebraic factor base consists of first degree prime ideals of $\mathbb{Z}[\alpha]$. These primes represented as pair (r, p) where r is the root of $f(x) = x^3 + 2 \cdot x^2 + 9 \cdot x + 19$ and p is a prime. The size of the algebraic factor base depends on bound \mathcal{B}_1 and degree d of the $f(x)$. In this case \mathcal{B}_1 is 107. To compute the algebraic factor base, one can find roots of $f(x)$ modulo $p \leq \mathcal{B}_1$. For prime $p = 53$ the function $f(x)$ may have d distinct roots r . In this case our algebraic factor base primes for p are $(2, 53)$, $(24, 53)$, and $(25, 53)$. We compute pairs (r, p) for every prime $p \leq \mathcal{B}_1$.

$$\begin{aligned} \text{AFB} = \left[\right. & (3, 7), (10, 11), (3, 13), (8, 17), (10, 17), (14, 17), \\ & (6, 19), (11, 19), (15, 23), (1, 31), (28, 37), (24, 41), \\ & (8, 43), (2, 53), (24, 53), (25, 53), (57, 67), (56, 79), \\ & (62, 83), (11, 89), (36, 101), (68, 101), (96, 101), \\ & \left. (32, 103), (82, 103), (90, 103) \right]. \quad (3.8) \end{aligned}$$

The rational factor base consists of primes integers up to a particular bound

\mathcal{B}_2 that generally determined by heuristically. In this example, we set bound \mathcal{B}_2 to 31. The rational factor base can be stored as pairs $(m \pmod{p}, p)$. We can compute m modulo for every prime p up to bound \mathcal{B}_2 .

$$\text{RFB} = \left[(1, 2), (1, 3), (2, 5), (2, 7), (4, 11), (11, 13), \right. \\ \left. (3, 17), (18, 19), (14, 23), (8, 29), (6, 31) \right]. \quad (3.9)$$

The quadratic character base consists of the primes that are larger than the algebraic factor base bound 103. Quadratic character base is used for increasing the probability of algebraic side integer to be square at the end of the algorithm. For $n = 53743$ quadratic character base is listed below.

$$\text{QCB} = \left[(92, 109), (76, 127), (74, 139), (75, 139), (127, 139) \right].$$

Sieve. For n , the sieving interval is chosen that $-1000 < a < 1000$ and positive integer b that starting at 1 and advance until more than enough (a, b) pairs are found that $a + bm$ and $a + b\alpha$ are smooth. To guarantee a linear dependence among the exponent vectors associated with pairs, the algorithm need to find more than 42 pairs. Two sieve arrays in memory will be created. One for $a + bm$ and the other $N(a + b\alpha)$ each with size 2000 entries for all a values for fixed b . Sieving smooth values for $a + bm$ proceed like as in 3.5. For prime $p = 7$ and $b = 3$, the values of a that $a + bm$ is divisible by p are of the form $a = -3m + 7k$ for $k \in \mathbb{Z}$. After that, $\log p$ have added to these a values. This process is applied every element in 3.9. A similar process is followed with pairs (r, p) in 3.8 for sieve array $N(a + b\alpha)$. When sieving process is finished, each sieve array is scanned for positions that the values are close to $\log(a + bm)$ and $\log(N(a + b\alpha))$. For next b , whole process is repeated.

Scanning the sieve array. In this process, we look at the sieve values that exceed the defined threshold for both sides. Then using (a, b) values of sieve entry we construct $a + bm$ and $a + b\alpha$ to test the smoothness. If the value is smooth we add this (a, b) to the matrix as a relation.

A matrix exponent vectors. Rows of the matrix correspond to an (a, b) pairs found in sieving process. For this example, column size of the matrix is 43, because of 26 first degree prime ideals in the algebraic factor base, 11 primes in rational factor base, 5 first degree prime ideal in the quadratic character base and one bit for the sign of $a + bm$. To demonstrate how the matrix formed, the row entry $(-5, 4)$ found in sieving will be examined. For $(-5, 4)$ rational side integer $a + bm = -5 + 4 \cdot 37 = 143$. The first bit of the row vector set 0 because, $a + bm$ is positive integer. First 11 entries in the row vector are prepared from factorization of $a + bm = 143$ over RFB:

$$143 = 2^0 \cdot 3^0 \cdot 5^0 \cdot 7^0 \cdot 11^1 \cdot 13^1 \cdot 17^0 \cdot 19^0 \cdot 23^0 \cdot 29^0 \cdot 31^0$$

which all RFB primes used to show factorization of $a + bm = 143$. Then, the exponent vector for pair $(-5, 4)$ is

$$(0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)$$

over \mathbf{F}_2 .

Second step is to compute the norm of an algebraic number. The norm of algebraic integer can be compute with $N(a + b\alpha) = (-b)^d f(-a/b)$. In this case, it can be computed with $d = 3$ and $f(x) = x^3 + 2x^2 + 9x + 19$ like

$$\begin{aligned} N(a + b\alpha) &= (-b)^3 \cdot \left(\frac{-a^3}{b^3} + 2\frac{a^2}{b^2} - 9\frac{a}{b} + 19 \right) \\ &= a^3 - 2a^2b + 9ab^2 - 19b^3. \end{aligned}$$

Thus, the norm of the $-5 + 4\alpha$ is $N(-5 + 4\alpha) = (-5)^3 - 2(-5)^2 \cdot 4 + 9 \cdot (-5) \cdot$

$4^2 - 19 \cdot 4^3 = -2261$ and $-2261 = -1 \cdot 7 \cdot 17 \cdot 19$ gives the factorization of the norm over the primes p that shown in Table 3.8. The next 26 bits in the row vector for $(-5, 4)$:

$$(1, 0, 0, 0, 0, 1, 1, 0).$$

The last step is to compute a quadratic character vector for $-5+4\alpha$ according to 3.4. For every pair (s, q) in quadratic character base, the Legendre symbol $\left(\frac{-5+4s}{q}\right)$ will be calculated. For example, pair $(76, 127)$ from the quadratic character base an example yields

$$\left(\frac{-5 + 4 \cdot 79}{127}\right) = -1.$$

In this case, 1 is stored the vector coordinate which corresponds $(76, 127)$. This process is applied the remaining (s, q) pairs of the quadratic character base which construct final 5 bits in the row vector for $(-5, 4)$.

$$(0, 1, 0, 1, 0)$$

The final form of the 42-bit row vector (exponent vector) for $(-5, 4)$ is

$$(0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0).$$

The whole process is applied to rest of the (a, b) pairs which are found in sieving process to construct the 43×42 binary matrix B .

$$\begin{aligned}
(a, b) \text{ pairs} = & \left[(-1, 1), (-2, 1), (-3, 1), (-6, 1), (-8, 1), \right. \\
& (-10, 1), (-11, 1), (-24, 1), (-25, 1), (-32, 1), \\
& (-65, 1), (-68, 1), (-82, 1), (-94, 1), (-367, 1), \\
& (1, 1), (3, 1), (5, 1), (7, 1), (8, 1), \\
& (9, 1), (13, 1), (21, 1), (23, 1), (77, 1), \\
& (78, 1), (-41, 2), (-207, 2), (-373, 2), (1, 2), \\
& (3, 2), (7, 2), (-7, 3), (1, 3), (4, 3), \\
& (5, 3), (10, 3), (43, 3), (-1, 4), (-5, 4), \\
& \left. (-43, 4), (13, 4), (27, 4) \right]
\end{aligned}$$

Linear algebra. The GF(2) matrix that we constructed for $n = 53743$ with the size of 43x42 shown in below

Finding a proper factor of n . In the last part of the algorithm we have $y \equiv 21111 \pmod{53743}$ and $x \equiv 29064 \pmod{53743}$. The next step is to prepare congruence of squares

$$29064^2 \equiv 21111^2 \pmod{53743}$$

to find non-trivial factor of 53743. Then, $\gcd(29064 - 21111, n)$ or $\gcd(29064 + 21111, n)$ return factor of 53743. The factors of 53743 are 223 and 241.

CHAPTER 4

ALTERNATIVE SCANNING METHODS

In the scanning part, we try to find (a, b) pair that yield as a relation to constructing our matrix. Scanning step has two phase first scanning sieve arrays and find candidate (a, b) that will be smooth and second test these (a, b) pairs smoothness with the trial division functions. In the first step, we look the sieve array values that correspond (a, b) pairs, if the value exceeds for defined threshold value then, (a, b) is accepted as candidate. In the second step, we check if the (a, b) is smooth over factor bases. In our implementation, we have two sides algebraic and rational. So, when we found the (a, b) is a candidate, we construct $a - bm$ (rational side) and norm of the algebraic side to test smoothness for both sides. Smoothness testing process will be performed functions `trialDivA` (for algebraic side) and `trialDivZ` (for rational side). The implementation of scanning part in C language shown below:

```
1   for (i = 0; i < (d->ar * 2); i++) {
2   #if THRESHOLD_MODE == 0
3       if ((d->arrSieveR[i] > d->thrR) && (d->arrSieveA[i] > d->thrA)) {
4
5   #elif THRESHOLD_MODE == 1
6       if ((d->arrSieveR[i] > d->thrR)) {
7   #elif THRESHOLD_MODE == 2
8       if ((d->arrSieveA[i] > d->thrA)) {
9   #endif
10      if (i < d->ar) {
11          a = i * (-1);
12      } else {
13          a = i - d->ar;
14      }
15      /* GCD(a,b) = 1 check */
16      gcd = binary_gcd(a, b);
17      if (gcd == 1) {
18  #if TRIAL_DIV_MODE == 0
19          mpz_norm1(d, res0, a, b, T2); /* a-b*alpha */
20          check = trialDivisionA(d, fb, arrTrialA, res0, b, a, fp);
21  #elif TRIAL_DIV_MODE == 1
22          mpz_mul_ui(T3, d->m, b);
23          mpz_set_si(res1, a);
24          mpz_sub(res1, res1, T3); /* a-b*m */
25          check = trialDivisionZ(d, fb, arrTrialR, res1, a);
26  #endif
27          if (check == 1) {
28  #if TRIAL_DIV_MODE == 0
29              /* a-b*m */
30              mpz_mul_ui(T3, d->m, b);
31              mpz_set_si(res1, a);
32              mpz_sub(res1, res1, T3);
33              check = trialDivisionZ(d, fb, arrTrialR, res1, a);
34  #elif TRIAL_DIV_MODE == 1
35              mpz_norm1(d, res0, a, b, T2); /* a-b*alpha */
36              check = trialDivisionA(d, fb, arrTrialA, res0, b, a);
37  #endif
38
39          if (check == 1) {
40              rows += nfs_relation_processing(d, fb, fulls, partials, arrTrialR,
              arrTrialA, res0, res1, T0, T1, sign, a, b, rows, hashtable_p,
```

```

41         hashtable_r, &half_to_full_rel);
42     }
43 } else {
44     //printf("GCD a and b not equal 1\n");
45 }
46 }
47 }

```

Code 4.1: ../elekmini/src/nfs/compute_nfs.c

We have 2 parameters and 2 functions that use in the scanning process. Parameters are `d->thrR` and `d->thrA`. The functions are trial division functions mention above. Different results can be obtained by using different combinations of these parameters and functions. In our own NFS implementation, we used these combinations to calculate the time spent in the scanning for n with different sizes. When these tests are performed, 6 different combinations are defined. We will use the `THR` keyword for the parameters and the `TrialDiv` keyword for the order of trial division functions. The combinations are listed below:

- **Both / TrialDivA** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed both rational and algebraic side threshold. Then, first check the smoothness of the algebraic side.
- **Both / TrialDivZ** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed both rational and algebraic side threshold. Then, first check the smoothness of the rational side.
- **Algebraic / TrialDivZ** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed algebraic side threshold. Then, first check the smoothness of the rational side.
- **Algebraic / TrialDivA** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed algebraic side threshold. Then, first check the smoothness of the algebraic side.
- **Rational / TrialDivA** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed rational side threshold. Then, first check the smoothness of the algebraic side.

THR / TrialDiv	Time (msec)
Both / TrialDivA	0.1421
Both / TrialDivZ	0.1431
Algebraic / TrialDivZ	0.3214
Algebraic / TrialDivA	1.8486
Rational / TrialDivA	17.4626
Rational / TrialDivZ	19.1303

Table 4.1: Timing results for 100-bit number

- **Rational / TrialDivZ** : While scanning sieve array we accept (a, b) if sieve value that represents (a, b) exceed rational side threshold. Then, first check the smoothness of the rational side.

100 Bits. The first test was done on

$$n = 737774618560715804003035572653.$$

Parameter for n are RFB bound 20000, AFB bound 30000, threshold for rational sieve array 30, threshold for algebraic sieve array 50, sieving range is $-50000 < a < 50000$, and degree of polynomial 3. In the following table, the time spent in scanning for 1 relation is given according to different combinations.

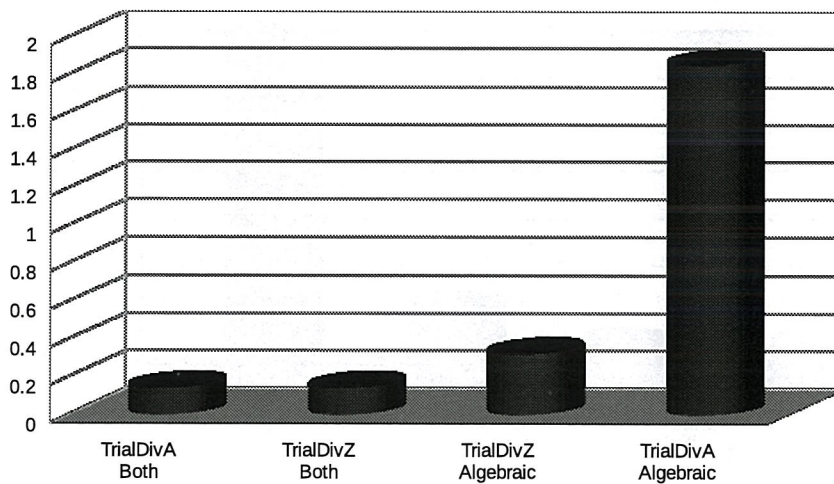


Figure 4.1: Test result for 100-bit inputs

150 Bits. The second test was done on

$$n = 2257727241354194125292213943385759534140088451.$$

Parameters for n are RFB bound 80000, AFB bound 100000, threshold for rational sieve array 35, threshold for algebraic sieve array 55, sieving range is $-200000 < a < 200000$, and degree of polynomial 4. In the following table, the time spent in scanning for 1 relation is given according to different combinations.

THR / TrialDiv	Time (msec)
Both / TrialDivA	0.8408
Both / TrialDivZ	0.8788
Algebraic / TrialDivZ	1.7272
Algebraic / TrialDivA	7.4516
Rational / TrialDivA	251.9948
Rational / TrialDivZ	292.6532

Table 4.2: Timing results for 150-bit number

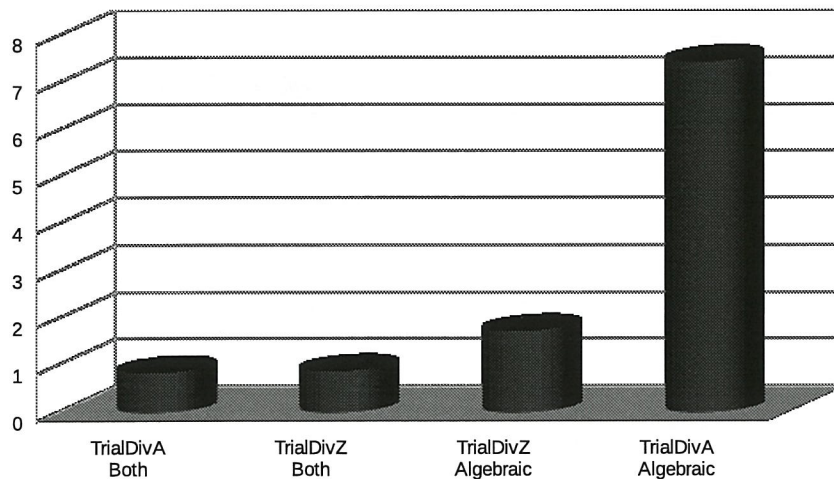


Figure 4.2: Test result for 150-bit inputs

200 Bits. The third test was done on

$$n = 90377629292003121684002147101760858109247336549001090677693.$$

Parameters for n are RFB bound 160000, AFB bound 200000, threshold for rational sieve array 40, threshold for algebraic sieve array 60, sieving range is $-350000 < a < 350000$, and degree of polynomial 4. In the following table, the time spent in scanning for 1 relation is given according to different combinations.

THR / TrialDiv	Time (msec)
Both / TrialDivA	4.612
Both / TrialDivZ	4.937
Algebraic / TrialDivZ	14.055
Algebraic / TrialDivA	107.358
Rational / TrialDivA	2605.028
Rational / TrialDivZ	2693.768

Table 4.3: Timing results for 200-bit number

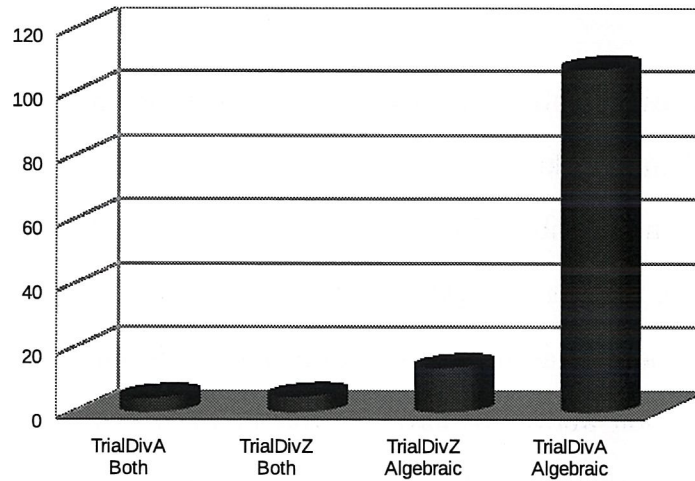


Figure 4.3: Test result for 200-Bit Input

Rational / TrialDivA and Rational/TrialDivZ. According to the results of the tests performed, only the rational scan is considered to have the worst

performance. As the size of the numbers increases, the time difference between other combinations is greatly increased. The reason is that the increase in the size of the numbers n causes the sieve arrays to increase in size. In each line of sieving, there will be far more smooth $a - bm$ than smooth $N(a, b)$. So, we have too many candidate (a, b) pairs are tested in trial division functions, and (a, b) are not algebraic smooth most of the time thus, we spend unnecessary time in trial division function.

Algebraic / TrialDivA and Algebraic / TrialDivZ. It has better performance than the checking rational side threshold. As mention above, we have far too many smooth rational (a, b) than the algebraic side. So, checking algebraic side threshold decrease the number of the (a, b) to test in the trial division step. If we came to order of trial division function, we know that the number exceeds the threshold of the algebraic side. This means that the $N(a, b)$ is highly likely to be smooth thus, to be accepted (a, b) as a relation depending on the rational side smoothness. For this reason, if we check the first smoothness of the rational side, we can avoid doing extra divisions and norm calculation for the algebraic side.

Both/TrialDivA and Both/TrialDivZ. As a result of our testing, checking both side threshold and checking first the smoothness of the algebraic side is more efficient than the other combinations. Because, checking both side thresholds help us to reduce the number of (a, b) pairs that send to trial division functions. So, we spend less time in trial division functions and avoid unnecessary division operation. As mention above we have far too many $a - bm$ smooth in the sieving range. So, in this case, the rational side is most likely to be smooth then, accepting the (a, b) pairs as a relation mostly depend on the smoothness of $N(a, b)$. Firstly checking whether the algebraic side is smooth is working as a kind of early abort mechanism to save us from the unnecessary division done in trialDivZ.

In conclusion, if one wants to implement NFS algorithm efficiently, the best option for scanning part is to accept (a, b) pairs that the corresponding sieve values in both sieve arrays exceed the threshold bound. This way one can reduce the number that is tested in trial division functions. This also reduces the time spent in the scanning process. The norm of an algebraic integer $a - b\alpha$ (in the algebraic side) is generated by using the function $N(a, b)$. Consequently, there will be less number of smooth values per line in the sieving phase in comparison to the rational side using the integer $a - bm$.

CHAPTER 5

IMPLEMENTING NFS WITH ALTERNATIVE SCANNING

This chapter contains low-level implementation details of the NFS algorithm. Our first task is to implement the steps of NFS excluding the linear algebra step. The implemented steps include;

1. Structured data for house-keeping,
2. Parameter selection,
3. Polynomial selection,
4. Factor base generation,
5. Sieving with: Line sieving and lattice sieving by rows,
6. **Scanning sieve array**,
7. Exponent vector generation and storing,
8. Linear algebra with the help of Magma,

We then modify the scanning step (in bold face) with respect to methods discussed in Chapter 4.

Our implementation is written in C language. On the other hand, we call Magma routines from C on-the-fly fashion at some non-time-critic moments in order to simplify the code development and focus more on the relevant parts.

We now explain the main parts of our software. The complete source code can be found in Appendix-6.

5.1 STRUCTURED DATA FOR HOUSE-KEEPING

We store algorithm related parameters in structs. We have two main structs for parameters.

NFS-data stores parameters related to factor base bound for algebraic and rational, coefficients and degree of polynomial, the number n that we want to factor, unique root m that $f(m) \equiv (\text{mod } n)$, sieve arrays for both line and lattice sieve, and sieve ranges for lattice and line sieving.

FB-nfs stores factor bases that generated with respect to factor bases bound that define in NFS-data, quadratic characters and special- q primes that generate for lattice sieving.

```
1 typedef struct {
2   mpz_t n;
3   mpz_t m;
4   ui B_r;
5   ui B_a;
6   ui b;
7   ui ar;
8   ui rangeC;
9   ui rangeE;
10  uc d;
11  ui thrR;
12  ui thrA;
13  ui thrLatticeA;
14  ui thrLatticeR;
15  ui lpb0;
16  ui lpb1;
17  ui smallSieveSize;
18  double *arrSieveR;
19  double *arrSieveA;
20  double *arrLatticeSieveA;
21  double *arrLatticeSieveR;
22  char *poly;
23  mpz_t *polyCoefficient;
24 } DATA_nfs_t[1], *DATA_nfs;
```

Code 5.1: ../elekmini/src/_elekmini_nfs.h

```
1 typedef struct {
2   ui asize;
3   ui rsize;
4   ui qsize;
5   ui sqsize;
6   ui *pr;
7   ui *pa;
8   ui *pq;
9   ui *psq;
10  ui *groots;
11  ui *roots;
12  ui *rroots;
13  ui *sqroots;
14  ui *sqi;
15  us *lr;
16  us *la;
17  LV_nfs_t *rbsv;
18 } FB_nfs_t[1], *FB_nfs;
```

Code 5.2: ../elekmini/src/_elekmini_nfs.h

5.2 PARAMATER SELECTION

Some parameters need to be specified in order for the algorithm to start working. These parameters is defined in the `init_data.c`. In this class, for specific n we define factor base bounds, threshold bounds, sieve array size, the degree of the algebraic side polynomial, and large prime bounds for both sides.

5.3 POLYNOMIAL SELECTION

We use magma scripts to generate our polynomial f (algebraic side). f generated by using base- m expansion. We execute the magma script in C language then, the output will be parsed and stored in struct NFS-data. The parameters which we get from this script are coefficients of f and m that satisfies $f(m) \equiv 0 \pmod{n}$. We do not need to store rational side polynomial because we can represent them as $a - bm$.

```
1  _<x>:=PolynomialRing(Integers());
2  f:=0;
3  k:=1;
4  repeat
5    k:= NextPrime(k);
6    ks:=IntegerToSequence(n,m);
7    for i:=1 to #ks do
8      if ks[i] lt m then
9        f:= f + x^(i-1)*ks[i];
10       end if;
11     end for;
12     if Degree(f) eq d then
13       break;
14     end if;
15     m:= m - (k*k);
16     until (Degree(f) eq d);
17     ks:= IntegerToSequence(n,m);
18     for i:=1 to #ks do
19       if ks[i] gt Floor(m/2) then
20         ks[i]:= ks[i] - m;
21         ks[i+1]+:=1;
22       end if;
23     end for;
24     f:=0;
25     for i:=1 to #ks do
26       f:= f + x^(i-1)*ks[i];
27     end for;
28 end function;
```

Code 5.3: project/magma_scripts.txt

```
1  gmp_sprintf(MAGMA_BUFFER, MAGMA_SCRIPT1, d->n, d->d);
2  char *polStr = magma_exec(MAGMA_BUFFER);
3  char *token;
4  int i = 0;
5  d->polyCoefficient = malloc((d->d + 1) * sizeof(mpz_t));
6  for (i = 0; i <= d->d; i++) {
7    token = strtok(polStr, ",");
8    mpz_init_set_str(d->polyCoefficient[i], token, 10);
9  }
10 token = strtok(polStr, ":");
11 mpz_init_set_str(d->m, token, 10);
12 token = strtok(polStr, ",");
13 d->poly = token;
```

Code 5.4: ../elekmini/src/nfs/init_poly.c

5.4 FACTOR BASE GENERATION

In GNFS, we need two factor base for smoothness testing. These are algebraic and rational factor bases. To generate these factor bases, we need to define some bounds for them. The bounds for algebraic and rational factor base are define in `init_data.c` and stored in a struct. `d->B_r` and `d->B_a` are rational and algebraic bounds for factor bases.

5.5 ALGEBRAIC FACTOR BASE

We use magma script to generate our algebraic factor base. We execute the magma script in C then, the output will be parsed and stored in struct `FB-nfs`.

```

1 InitAlgebraicFactorBase2:= function(poly,
2   bound)
3   fbpa:=[];
4   p:=2;
5   while p lt bound do
6     PR<x>:=PolynomialRing(GF(p));
7     pol:=PR!poly;
8     if (HasRoot(pol)) eq true then
9       roots:=Roots(pol);
10      for i:=1 to #roots do
11        tuple:=<0,p>;
12        tuple[1]:=roots[i][1];
13        Append(~fbpa,tuple);
14      end for;
15    end if;
16    p:=NextPrime(p);
17  end while;
18  return fbpa;
end function;

```

Code
project/magma_scripts.txt

```

1 fb->pa = (ui *) malloc(sizeof(ui) * (d->B_a) / 2);
2 fb->roots = (ui *) malloc(sizeof(ui) * (d->B_a) / 2);
3
4 gmp_sprintf(MAGMA_BUFFER, MAGMA_SCRIPT, d->poly, d->B_a);
5 char *fbStr = magma_exec(MAGMA_BUFFER);
6 char *token;
7 while ((token = strsep(&fbStr, ":")) {
8   if ((j % 2) == 0) {
9     fb->roots[i] = atoi(token);
10  } else if ((j % 2) == 1) {
11    fb->pa[i] = atoi(token);
12    i++;
13  }
14  j++;
15 }
16 fb->pa = (ui *) realloc(fb->pa, sizeof(ui) * i);
17 fb->roots = (ui *) realloc(fb->roots, sizeof(ui) * i);
18 fb->asize = i;

```

Code 5.6: ../elekmini/src/nfs/init_fb.c

The smothness test (a, b) of the algebraic side of the pairs found in the sieving process will be performed with the primes found on the algebraic factor base that generated above. As in mention in thesis algebraic factor base will be stored as a pairs (r, p) . The executed magma script returns string that represent primes pairs. We parse this string in C and store p 's in array `fb->ap` and r 's in the `fb->roots`. The memory allocation of `fb->ap` and `fb->roots` is performed in parser function. The magma script output is like `2:3:4:7:5:11:10:11:0:13:14:19:16:23:...`

5.6 RATIONAL FACTOR BASE

We generate rational factor base in C language. To generate a rational factor base we used GMP library. The rational factor base is used for the smoothness testing of the rational side in the algorithm. We generate our rational factor base with respect to bound $d \rightarrow B_r$. To generate our rational factor base we use GMP function `mpz_nextprime`. We manually add first prime 2 into the array `fb->pr`. Then, generate and store other primes until the bound $d \rightarrow B_r$ with `mpz_nextprime`. We also store $(m \bmod p)$ in the array `fb->rroots` to use in rational side sieve in line sieve. So, we store our rational factor base as a pair like algebraic factor base.

```
1 while (mpz_cmp_ui(p, d->B_r) != 1) {
2   fb->pr[i] = (ui) mpz_get_ui(p);
3   mpz_mod(temp, d->m, p);
4   fb->rroots[i] = mpz_get_ui(temp);
5   i++;
6   mpz_nextprime(p, p);
7 }
8 fb->pr = (ui *) realloc(fb->pr, sizeof(ui) * i);
9 fb->rroots = (ui *) realloc(fb->rroots, sizeof(ui) * i);
10 fb->rsize = i;
```

Code 5.7: ../elekmini/src/nfs/init_fb.c

5.7 SIEVING WITH: LINE SIEVING AND LATTICE SIEVING BY ROWS

We have two different implementations of the sieving process in c lattice and line sieving. The implementation of lattice sieve is more complicated than line sieving. Sieving is the core process of this algorithm and most of the process time is spent in this stage.

For line sieving in our implementation, we fixed value of b then sieve values of a within range $-d \rightarrow ar < a < d \rightarrow ar$. We represent this sieving region with two array, negative and positive values of a . The same sieving process will follow for both array. In rational side sieve, we calculate smallest value of a for p that satisfies $a \equiv mb \pmod{p}$. The same procedure is applied to algebraic side with little change in calculating smallest value of a for p .

```

1  for (i = 0; i < fb->rsize; i++) {
2      p = fb->pr[i];
3      pLog = log2((double) p);
4      posStartPoint = (b * fb->rroots[i]) % p;
5      negStartPoint = posStartPoint - p;
6      negStartPoint *= -1;
7
8      for (j = negStartPoint; j < d->ar; j = j + p) {
9          d->arrSieveR[j] = d->arrSieveR[j] + pLog;
10     }
11     for (j = posStartPoint + (d->ar); j < (d->ar * 2); j = j + p) {
12         d->arrSieveR[j] = d->arrSieveR[j] + pLog;
13     }
14 }
15
16 /* Algebraic Side Line Sieve */
17 for (i = 0; i < fb->asize; i++) {
18     p = fb->pa[i];
19     pLog = log2(p);
20     mpz_set_ui(T0, fb->roots[i]);
21     mpz_mul_ui(T0, T0, b);
22     mpz_mod_ui(T0, T0, p);
23     posStartPoint = mpz_get_ui(T0);
24     negStartPoint = posStartPoint - p;
25     negStartPoint *= -1;
26     for (j = negStartPoint; j < d->ar; j = j + p) {
27         d->arrSieveA[j] = d->arrSieveA[j] + pLog;
28     }
29     for (j = posStartPoint + (d->ar); j < (d->ar * 2); j = j + p) {
30         d->arrSieveA[j] = d->arrSieveA[j] + pLog;
31     }
32 }

```

Code 5.8: ../elekmini/src/nfs/compute_nfs.c

For lattice sieve by row in our implementation, we first need to generate special- q primes that will be used in sieving. We select our special- q primes from computed algebraic factor base. We choose primes from algebraic factor base with greater than given bound.

```

1  for (i = 0; i < fb->asize; i++) {
2      if (fb->pa[i] > q_bound) {
3          fb->psq[j] = fb->pa[i];
4          fb->sqroots[j] = fb->roots[i];
5          fb->sqi[j] = i;
6          reduceBasis(fb->rbsv[j], fb->psq[j], fb->sqroots[j]);
7          j++;
8      }
9  }

```

Code 5.9: ../elekmini/src/nfs/init_fb.c

For lattice sieving by rows in our implementation, as mention above we need to find reduce basis vector of lattice L_q . So, we precalculated reduced basis with function `reduceBasis` for every special- q and store in the `fb->rbsv`. Then, lattice sieve followed as below:

```

1  for (k = 0; k < fb->sqi[sqIndex]; k++) {
2      p = fb->pa[k];
3      r = fb->roots[k];
4      pLog = log2((double) p);
5      j=1;
6      c_1 = calculateStartPointForLatticeSieve(fb->rbsv[sqIndex], j, p, r);
7      c_d = 0;
8      for (j = 1; j < d->rangeE; j++) {
9          /* Negative part sieve (-C,0) */
10         i0 = (c_1 + c_d) % p;
11         for (i = (p - i0); i < d->rangeC; i += p) {

```

```

12|     d->arrLatticeSieveA[((j - 1) * d->rangeC) + i] += pLog;
13|     }
14|     /* Positive part sieve (0, C) */
15|     for (i = (d->rangeC + i0); i < (2 * d->rangeC); i += p) {
16|         d->arrLatticeSieveA[((j - 1) * d->rangeC) + i] += pLog;
17|     }
18|     c_d = i0;
19| }
20| }
21|
22| for (k = 0; k < fb->rsize; k++) {
23|     p = fb->pr[k];
24|     r = fb->rroots[k];
25|     pLog = log2((double) p);
26|     c_1 = calculateStartPointForLatticeSieve(fb->rbsv[sqIndex], j, p, r);
27|     c_d = 0;
28|     for (j = 1; j < d->rangeE; j++) {
29|         i0 = (c_1 + c_d) % p;
30|         /* Negative part sieve (-C, 0) */
31|         for (i = (p - i0); i <= d->rangeC; i += p) {
32|             d->arrLatticeSieveR[((j - 1) * d->rangeC) + i] += pLog;
33|         }
34|         /* Positive part sieve (0, C) */
35|         for (i = (d->rangeC + i0); i <= (2 * d->rangeC); i += p) {
36|             d->arrLatticeSieveR[((j - 1) * d->rangeC) + i] += pLog;
37|         }
38|         c_d = i0;
39|     }
40| }

```

Code 5.10: ../elekmini/src/nfs/compute_nfs.c

5.8 SCANNING SIEVE ARRAY

When sieving process has finished, we scan sieve array with the values on both side are greater or equal to threshold. If we use line sieve we have values a, b then we construct $a - bm$ and $N(a - b\alpha)$ to test for smoothness. The smoothness test will be performed function `trialDivZ` (rational side) and `trialDivA` (algebraic side). If we use lattice sieving, we have values c, e . We calculate a, b with given c, e and follow the same process mentioned in 3.5.

```

1  for (j = 0; j < fb->rsize; j++) {
2  p = fb->pr[j];
3  while (mpz_divisible_ui_p(res, p) > 0) {
4  mpz_divexact_ui(res, res, p);
5  arr[j]++;
6  }
7  }

```

Code
../elekmini/src/nfs/compute_nfs.c

```

1  for (i = 0; i < fb->asize; i++) {
2  p = fb->pa[i];
3  r = fb->rroots[i];
4  mpz_set_ui(A3, b);
5  mpz_mul_ui(A2, A3, r);
6  mpz_mod_ui(A2, A2, p);
7  mpz_mod_ui(A3, A1, p);
8  if (mpz_cmp(A3, A2) == 0) {
9  while ((mpz_divisible_ui_p(res, p)) != 0) {
10  mpz_divexact_ui(res, res, p);
11  arr[i]++;
12  }
13  }
14  }

```

Code 5.12: ../elekmini/src/nfs/compute_nfs.c

5.9 EXPONENT VECTOR GENERATION AND STORING

After both $a - bm$ and $N(a - b\alpha)$ pass the smoothness test we accept (a, b) pair as relations. In `trialDivA` and `trialDivZ` function construct the exponent vector for given number. Factorization vector for rational side stored in `arrTrialR` and algebraic side store in `arrTrialA`. Then, we merge `arrTrialA`, `arrTrialZ` as relations. Then, we calculate legendre symbol of all quadratic characters with using GMP function `mpz_legendre_symbol` and added to end of the merged vector.

```

1   for (j = 0; j < fb->rsize; j++) {
2       fulls->vector[j + rows * cols] = arrTrialR[j];
3   }
4   for (j = 0; j < fb->asize; j++) {
5       fulls->vector[j + rows * cols + fb->rsize] = arrTrialA[j];
6   }
7   for (j = 0; j < fb->qsize; j++) {
8       mpz_set_si(T0, a);
9       mpz_sub_ui(T0, T0, b * fb->qroots[j]);
10      mpz_set_ui(T1, fb->pq[j]);
11      if (mpz_legendre(T0, T1) == 1) {
12          fulls->vector[j + rows * cols + fb->asize + fb->rsize] = 0;
13      } else {
14          fulls->vector[j + rows * cols + fb->asize + fb->rsize] = 1;
15      }
16  }
17
18  fulls->a[rows] = a;
19  fulls->b[rows] = b;
20  sign0 = mpz_get_si(res1);
21  if (sign0 < 0) {
22      fulls->sign[rows] = 1;
23  } else {
24      fulls->sign[rows] = 0;
25  }

```

Code 5.13: ../elekmini/src/nfs/compute_relations.c

5.10 LINEAR ALGEBRA WITH THE HELP OF MAGMA

In this step, we generate `txt` file that written in MAGMA language to solve the matrix that construct in the process above. The linear algebra file is generated by function `linear_algebra_nfs`. When enough relationships are gathered for the linear algebraic step, we call the function to generate the file. The file stores

factor bases , (a, b) pairs, matrix that store exponent vectors for corresponding (a, b) pair, polynomial f , n and m . If we execute the magma code written to the file we can obtain the factors of the n .

CHAPTER 6

CONCLUSION

Number Field Sieve (NFS) is a general purpose integer factorization algorithm. NFS currently has the lowest asymptotic complexity and it has a crucial role in research on attacking the widely used industry standard RSA cryptosystem. NFS is composed of several steps each of which might be amenable to further improvements and thus becoming a more serious threat against RSA in the future.

In particular, this thesis focused on the optimization of the scanning stage that comes right after the sieving stage. In this context, we investigated existing techniques in literature and implementation choices in freely available software. We reproduced these techniques in from-the-scratch C & Magma implementations and then incorporated our modifications. In this direction, we tried six different variations in order to get the best timings which are depicted in Chapter 4. We came to a conclusion that the best practice is to

- to compare against the preset thresholds of both the algebraic and the rational sides together.
- send the values from the algebraic side to smooth factorization test, which passes the threshold test.
- send the values from the rational side to smooth factorization test, which passes the threshold test and the smoothness test of item 2.

Our experiment results are consist with previous attempts which are only empirically available in open source libraries CADO-NFS (Team, 2017), GGNFS (Monico, 2005), and msieve (Papadopoulos,). This thesis work is by no means a complete NFS implementation. There are several other parts to be implemented and optimized such as lattice sieve by vector, lattice sieve by vector

Kleijung's improvement (Kleijung et al., 2010), better polynomial selection, parallelization of sieving process, bucket sieving, and cache optimization. These further topics are left as future work.

REFERENCES

- Adleman, L. M. (1991). Factoring numbers using singular integers. In Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing, STOC '91, pages 64–71, New York, NY, USA. ACM. 19
- Aoki, K. and Ueda, H. (2004). Sieving using bucket sort. In Lee, P. J., editor, Advances in Cryptology - ASIACRYPT 2004, pages 92–102, Berlin, Heidelberg. Springer Berlin Heidelberg. 23
- B. A., M. (2010). Polynomial Selection for the Number Field Sieve Integer Factorisation. PhD thesis, Australian National University. 17
- Briggs, M. E. (1998). An introduction to the general number field sieve. 18
- Buhler, J. P., Lenstra Jr, H. W., and Pomerance, C. (1993). Factoring integers with the number field sieve. In The development of the number field sieve, pages 50–94. Springer. 13
- Cohen, H. (1993). A Course in Computational Algebraic Number Theory. Springer-Verlag, Berlin, Heidelberg. 14
- Coppersmith, D. (1994). Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. Math. Comput., 62(205):333–350. 11
- Couveignes, J.-M. (1993). Computing a square root for the number field sieve. In Lenstra, A. K. and Lenstra, H. W., editors, The development of the number field sieve, pages 95–102, Berlin, Heidelberg. Springer Berlin Heidelberg. 26

- D. Silverman, R. (1987). The multiple polynomial quadratic sieve. Math. Comp. 48, 48:329–329. 8
- Davis, J. A. and Holdridge, D. B. (1984). Factorization Using the Quadratic Sieve Algorithm, pages 103–113. Springer US, Boston, MA. 21
- Franke, J. and Kleinjung, T. (2005). Continued fractions and lattice sieving. Special-Purpose Hardware for Attacking Cryptographic Systems–SHARCS, page 40. 23
- Golliver, R. A., Lenstra, A. K., and McCurley, K. S. (1994). Lattice sieving and trial division. In Adleman, L. M. and Huang, M.-D., editors, Algorithmic Number Theory, pages 18–27, Berlin, Heidelberg. Springer Berlin Heidelberg. 21
- K. Lenstra, A. and W. Lenstra, H. (1993). The development of the number field sieve /. 1554. 26
- Kleinjung, T. (2006). On polynomial selection for the general number field sieve. Mathematics of Computation, 75(256):2037–2047. 17
- Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., Te Riele, H., Timofeev, A., and Zimmermann, P. (2010). Factorization of a 768-bit RSA modulus. In Proceedings of the 30th Annual Conference on Advances in Cryptology, CRYPTO’10, pages 333–350, Berlin, Heidelberg. Springer-Verlag. 7, 10, 11, 54
- Kruppa, A. (2010). Speeding up Integer Multiplication and Factorization. PhD thesis, Université Henri Poincaré - Nancy I. 24
- Lenstra, A. K., Lenstra, H. W., Manasse, M. S., and Pollard, J. M. (1993). The factorization of the ninth Fermat number. Mathematics of Computation, 61(203):319–349. 13

- Lenstra, A. K., Lenstra, Jr., H. W., Manasse, M. S., and Pollard, J. M. (1990). The number field sieve. In Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing, STOC '90, pages 564–572, New York, NY, USA. ACM. 9, 13
- Milne, J. S. (2009). Algebraic number theory. www.jmilne.org/math. 14
- Monico, C. (2005). GGNFS, a number field sieve implementation. 2, 53
- Montgomery, P. L. (1994). Square roots of products of algebraic numbers. In Gauthschi, W., editor, Mathematics of Computation 1943-1993: A Half-Century of Computational Mathematics. 26
- Montgomery, P. L. (1995). A Block Lanczos Algorithm for Finding Dependencies over $GF(2)$, pages 106–120. Springer Berlin Heidelberg, Berlin, Heidelberg. 10
- Papadopoulos, J. MSIEVE, a library for factoring large integers. 2, 53
- Pollard, J. M. (1993a). Factoring with cubic integers. In Lenstra, A. K. and Lenstra, H. W., editors, The development of the number field sieve, pages 4–10, Berlin, Heidelberg. Springer Berlin Heidelberg. 13
- Pollard, J. M. (1993b). The lattice sieve. In Lenstra, A. K. and Lenstra, H. W., editors, The development of the number field sieve, pages 43–49, Berlin, Heidelberg. Springer Berlin Heidelberg. 21
- Pomerance, C. (1985). The quadratic sieve factoring algorithm. In Beth, T., Cot, N., and Ingemarsson, I., editors, Advances in Cryptology, pages 169–182, Berlin, Heidelberg. Springer Berlin Heidelberg. 8
- Stevenhagen, P. (2008). The number field sieve. 16
- Stewart, I. and Tall, D. (2001). Algebraic Number Theory and Fermat's Last Theorem. 14

Team, T. C.-N. D. (2017). CADO-NFS, an implementation of the number field sieve algorithm. Release 2.3.0. 2, 53

Thomé, E. (2001). Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, ISSAC '01, pages 323–331, New York, NY, USA. ACM. 11

Wagstaff, S. S. (2013). The Joy of Factoring. AMS, American Mathematical Society. 13

APPENDIX

```
1 typedef struct {
2     mpz_t n;
3     mpz_t m;
4     ui B_r;
5     ui B_a;
6     ui b;
7     ui ar;
8     ui rangeC;
9     ui rangeE;
10    uc d;
11    ui thrR;
12    ui thrA;
13    ui thrLatticeA;
14    ui thrLatticeR;
15    ui lpb0;
16    ui lpb1;
17    ui smallSieveSize;
18    double *arrSieveR;
19    double *arrSieveA;
20    double *arrLatticeSieveA;
21    double *arrLatticeSieveR;
22    char *poly;
23    mpz_t *polyCoefficient;
24 } DATA_nfs_t[1], *DATA_nfs;
25
26 typedef struct {
27     int v1[2]; /* Reduce basis vector 1 */
28     int v2[2]; /* Reduce basis vector 2 */
29 } LV_nfs_t[1], *LV_nfs;
30
31 typedef struct {
32     ui asize;
33     ui rsize;
34     ui qsize;
35     ui sqsize;
36     ui *pr;
37     ui *pa;
38     ui *pq;
39     ui *psq;
40     ui *qroots;
41     ui *roots;
42     ui *rroots;
43     ui *sqroots;
44     ui *sqi;
45     us *lr;
46     us *la;
47     LV_nfs_t *rbsv;
48 } FB_nfs_t[1], *FB_nfs;
49
50
51 typedef struct {
52     uc *type;
53     ul size;
54     sl *a;
55     ul *b;
56     sl *partial_a;
57     ul *partial_b;
58     uc *vector; /* exponent vector */
59     uc *sign; /* sign of the a-bm */
60 } REL_FULL_NFS_t[1], *REL_FULL_NFS;
61 void init_full_relations_nfs(REL_FULL_NFS fulls, FB_nfs fb, int vectorSize);
62 void kill_full_relations_nfs(REL_FULL_NFS fl1);
63
64 typedef struct {
65     sl a;
66     ui b;
67     uc s; /*Matrix of the sign */
68     uc *e; /*Matrix of the exponents */
69 } LARGE_PRIME_DATA_NFS_t[1], *LARGE_PRIME_DATA_NFS;
70
71 typedef struct {
72     //ui data_size;
73     ui data_count;
74     LARGE_PRIME_DATA_NFS_t *data;
75 } REL_PARTIAL_NFS_t[1], *REL_PARTIAL_NFS;
76 void init_relations_partial_nfs(REL_PARTIAL_NFS partials);
77 void kill_relations_partial_nfs(REL_PARTIAL_NFS partials);
78
```

```

79 #ifndef __HASH_H__
80 #define __HASH_H__ 1
81
82 typedef struct svz_hash_entry svz_hash_entry_t;
83 typedef struct svz_hash_bucket svz_hash_bucket_t;
84 typedef struct svz_hash svz_hash_t;
85
86 /* begin svzint */
87 /*
88 * This structure keeps information of a specific hash table.
89 * It's here (rather than in .c) for the benefit of cfg.c áhash_dupá.
90 */
91 struct svz_hash {
92     size_t buckets; /* number of buckets in the table */
93     size_t fill; /* number of filled buckets */
94     size_t keys; /* number of stored keys */
95     int (*equals)(const char *, const char *); /* key string equality callback */
96     unsigned long (*code)(const char *); /* hash code calculation callback */
97     size_t (*keylen)(const char *); /* how to get the hash key length */
98     void (*destroy)(void *); /* element destruction callback */
99     svz_hash_bucket_t *table; /* hash table */
100 };
101 /* end svzint */
102
103 typedef void (svz_hash_do_t)(void *, void *, void *);
104
105 __BEGIN_DECLS
106
107 /*
108 * Basic hash table functions.
109 */
110 svz_hash_t *svz_hash_create(size_t);
111 svz_hash_t *svz_hash_configure(svz_hash_t *hash, size_t (*keylen)(const char *), unsigned long (*code)(
    const char *), int (*equals)(const char *, const char *));
112 void svz_hash_destroy(svz_hash_t *);
113 void *svz_hash_delete(svz_hash_t *, const char *);
114 void *svz_hash_put(svz_hash_t *, const char *, void *);
115 void *svz_hash_get(const svz_hash_t *, const char *);
116 void svz_hash_foreach(svz_hash_do_t *, svz_hash_t *, void *);
117 size_t svz_hash_size(const svz_hash_t *);
118 char *svz_hash_contains(const svz_hash_t *, void *);
119 int svz_hash_exists(const svz_hash_t *, char *);
120
121 __END_DECLS
122
123 #endif /* not __HASH_H__ */
124
125
126
127 /* Initialize Process */
128 void nfs_init_data(DATA_nfs d, LV_nfs lv);
129 void nfs_init_poly(DATA_nfs d);
130 void nfs_init_factor_base_rational(FB_nfs fb, DATA_nfs d);
131 void nfs_init_factor_base_algebraic(FB_nfs fb, DATA_nfs d);
132 void nfs_init_quadratic_characterbase(FB_nfs fb, DATA_nfs d);
133 void nfs_init_special_q_characterbase(FB_nfs fb, DATA_nfs d, ui q_bound);
134
135
136 /* Sieving functions and Linear Algebra */
137 int compute_numberfield_sieve(REL_FULL_NFS fulls, REL_PARTIAL_NFS partials, FB_nfs fb, DATA_nfs d, POL p,
    LARGE_PRIME_DATA hd, FILE *fp);
138 int compute_numberfield_sieve_lattice(REL_FULL_NFS fulls, REL_PARTIAL_NFS partials, FB_nfs fb, DATA_nfs d,
    POL p, LARGE_PRIME_DATA hd, LV_nfs lv, FILE *fp);
139 int nfs_relation_processing(DATA_nfs_t d, FB_nfs_t fb, REL_FULL_NFS fulls, REL_PARTIAL_NFS partials, uc *
    arrTrialR, uc *arrTrialA, mpz_t res0, mpz_t res1, mpz_t T0, mpz_t T1, int sign, sl a, ul b, int rows,
    svz_hash_t *hashtable_p, svz_hash_t *hashtable_r, int *htfc);
140 void linear_algebra_nfs(DATA_nfs_t d, FB_nfs_t fb, REL_FULL_NFS_t r, int cnt);
141 //void mpz_norm(DATA_nfs d, mpz_t res, sl a, int b);
142
143 /* Util functions */
144 ui binary_gcd(sl a, int b);
145 void reduceBasis(LV_nfs lv, ui pr, int r);
146 void mpz_norm(DATA_nfs d, mpz_t res, sl a, int b, mpz_t temp);
147 int mpz_evalNorm(DATA_nfs d, mpz_t res, sl a, int b);
148 void mpz_norm1(DATA_nfs d, mpz_t res, sl a, int b, mpz_t temp);
149
150 /*unsigned long iLog(unsigned long b);
151 unsigned long bitcount(unsigned long b);*/

```

Code 6.1: ../elekmini/src/_elekmini.nfs.h

```

1 #include "_elekmini.h"
2
3 #define TEST_ELEK_100 /* 50, 100, 200, 250, 300 */
4 #define LINE_SIEVE /* LINE_SIEVE, LATTICE_SIEVE_BY_ROWS, LATTICE_SIEVE_BY_VECTORS */
5 // #define LATTICE_SIEVE_BY_ROWS
6 void nfs_init_data(DATA_nfs d, LV_nfs lv) {
7     mpz_init(d->n);
8     mpz_init(d->m);
9 }
10 #ifdef TEST_ELEK_50
11     mpz_set_str(d->n, "1053490610310707", 10);
12     d->E_r = 1000;
13     d->B_a = 1500;
14     d->thrR = 8;
15     d->thrA = 10;

```



```

16 | d->ar = 10000;
17 | d->rangeE = 60;
18 | d->rangeC = 100;
19 | d->lpb0 = 1;
20 | d->lpb1 = 1;
21 | d->d = 3; //Degree of the polynomial.
22 | #endif
23 |
24 | #ifdef TEST_ELEK_70
25 | //mpz_set_str(d->n, "2794708678894578235763", 10);
26 | mpz_set_str(d->n, "1180591622349498875779", 10);
27 | d->B_r = 3000;
28 | d->B_a = 5000;
29 | d->thrR = 12;
30 | d->thrA = 14;
31 | d->ar = 16000;
32 | d->rangeE = 80;
33 | d->rangeC = 200;
34 | d->lpb0 = 1;
35 | d->lpb1 = 17;
36 | d->d = 3; //Degree of the polynomial.
37 | #endif
38 |
39 | #ifdef TEST_ELEK_100
40 | mpz_set_str(d->n, "737774618560715804003035572653", 10);
41 | //mpz_set_str(d->n, "1267650600228312155139856129339", 10);
42 | d->B_r = 20000;
43 | d->B_a = 30000;
44 | d->thrR = 30;
45 | d->thrA = 45; //50
46 | d->thrLatticeA = 25;
47 | d->ar = 50000;
48 | d->rangeE = 40;
49 | d->rangeC = 1000;
50 | d->lpb0 = 1;
51 | d->lpb1 = 19;
52 | d->d = 3; //Degree of the polynomial.
53 | #endif
54 |
55 | #ifdef TEST_ELEK_150
56 | mpz_set_str(d->n, "2257727241354194125292213943385759534140088451", 10);
57 | d->B_r = 80000;
58 | d->B_a = 100000;
59 | d->thrR = 35;
60 | d->thrA = 55;
61 | d->ar = 300000;
62 | d->rangeE = 200;
63 | d->rangeC = 10000;
64 | d->lpb0 = 1;
65 | d->lpb1 = 23;
66 | d->d = 4; //Degree of the polynomial.
67 | #endif
68 |
69 | #ifdef TEST_ELEK_200
70 | mpz_set_str(d->n, "90377629292003121684002147101760858109247336549001090677693", 10);
71 | d->B_r = 160000;
72 | d->B_a = 200000;
73 | d->thrR = 48;
74 | d->thrA = 65;
75 | d->ar = 350000;
76 | d->rangeC = 10000;
77 | d->rangeE = 200;
78 | d->lpb0 = 1;
79 | d->lpb1 = 25;
80 | d->smallSieveSize = 1000;
81 | d->d = 4; //Degree of the polynomial.
82 | #endif
83 |
84 | #ifdef TEST_ELEK_300
85 | mpz_set_str(d->n,
86 | //mpz_set_str(d->n,
87 | "229819220467665407292310136245859203083400224623948937516985063841549618327517460879624657", 10);
88 | "1018517988167243043134222844204689080525734400215764335917353275482941296434725287633102751", 10);
89 | d->B_r = 600000;
90 | d->B_a = 1000000;
91 | d->thrR = 55;
92 | d->thrA = 75;
93 | d->ar = 3000000;
94 | d->rangeE = 10;
95 | d->rangeC = 100;
96 | d->lpb0 = 23;
97 | d->lpb1 = 23;
98 | d->smallSieveSize = 5000;
99 | d->d = 5; //Degree of the polynomial.
100 | #endif
101 | #ifdef LINE_SIEVE
102 | d->arrSieveR = (double *) malloc((double) sizeof(double) * (d->ar*2));
103 | d->arrSieveA = (double *) malloc((double) sizeof(double) * (d->ar*2));
104 |
105 | if (d->arrSieveR == NULL) {
106 |     printf("> Cannot allocate memory for the sieve array.\n");
107 |     exit(0);
108 | } else {
109 |     printf("> Allocated %.0lf kbytes for the sieve array.\n", ((double) ((double) sizeof(double) * (d->ar*2)
110 |     )) / 1024);

```

```

111 if (d->arrSieveA == NULL) {
112     printf(" > Cannot allocate memory for the sieve array.\n");
113     exit(0);
114 } else {
115     printf(" > Allocated %.01f kbytes for the sieve array.\n", ((double) ((double) sizeof(double) * (d->arrSieveA * 2))) / 1024);
116 }
117 #endif
118
119 #ifdef LATTICE_SIEVE_BY_ROWS
120 d->arrLatticeSieveA = (double *) malloc(sizeof(double) * ((d->rangeC * 2) * d->rangeE));
121 d->arrLatticeSieveR = (double *) malloc(sizeof(double) * ((d->rangeC * 2) * d->rangeE));
122
123 if (d->arrLatticeSieveA == NULL) {
124     printf(" > Cannot allocate memory for the sieve array.\n");
125     exit(0);
126 } else {
127     printf(" > Allocated %.01f kbytes for the sieve array.\n", ((double) ((double) sizeof(double) * (d->rangeC * 2) * d->rangeE)) / 1024);
128 }
129
130 if (d->arrLatticeSieveR == NULL) {
131     printf(" > Cannot allocate memory for the sieve array.\n");
132     exit(0);
133 } else {
134     printf(" > Allocated %.01f kbytes for the sieve array.\n", ((double) ((double) sizeof(double) * (d->rangeC * 2) * d->rangeE)) / 1024);
135 }
136 #endif
137 }
138
139 void nfs_kill_data(DATA_nfs d) {
140 #ifdef LINE_SIEVE
141     free(d->arrSieveA);
142     free(d->arrSieveR);
143 #endif
144
145 #ifdef LATTICE_SIEVE_BY_ROWS
146     free(d->arrLatticeSieveR);
147     free(d->arrLatticeSieveA);
148 #endif
149 }
150 }

```

Code 6.2: ../elekmini/src/nfs/init_data.c

```

1 #include "_elekmini.h"
2 #define VERBOSE
3
4 void nfs_init_factor_base_rational(FB_nfs fb, DATA_nfs d) {
5     int i;
6     mpz_t p, temp;
7     mpz_init(p);
8     mpz_init(temp);
9     fb->pr = (ui *) malloc(sizeof(ui) * (d->B_r) / 2); /* Allocate more than enough space. Trimmed after the
    loop.*/
10    fb->rroots = (ui *) malloc(sizeof(ui) * (d->B_r) / 2);
11    /* Add 2 manually. */
12    fb->pr[0] = 2;
13    mpz_mod_ui(temp, d->m, 2);
14    fb->rroots[0] = mpz_get_ui(temp);
15    i = 1;
16    mpz_set_ui(p, 3);
17    while (mpz_cmp_ui(p, d->B_r) != 1) {
18        fb->pr[i] = (ui) mpz_get_ui(p);
19        mpz_mod(temp, d->m, p);
20        fb->rroots[i] = mpz_get_ui(temp);
21        i++;
22        mpz_nextprime(p, p);
23    }
24    fb->pr = (ui *) realloc(fb->pr, sizeof(ui) * i);
25    fb->rroots = (ui *) realloc(fb->rroots, sizeof(ui) * i);
26    fb->rsize = i;
27    mpz_clear(temp);
28    mpz_clear(p);
29
30    #ifdef VERBOSE
31        printf(" >          i          p          (m mod p)\n");
32        printf(" > -----\n");
33        //for (i = 0; i < fb->rsize; i++) {
34        for (i = 0; i < 10; i++) {
35            printf(" > %9d %9d %9d \n", i, fb->pr[i], fb->rroots[i]);
36        }
37        printf(" >          ...          ...          ... \n");
38        for (i = fb->rsize - 5; i < fb->rsize; i++) {
39            printf(" > %9d %9d %9d \n", i, fb->pr[i], fb->rroots[i]);
40        }
41        fflush(stdout);
42    #endif
43    }
44
45    #define MAGMA_SCRIPT "\
46    InitAlgebraicFactorBase2:= function(poly,bound)  \n\
47    fbpa=[];                                         \n\
48    p:=2;                                           \n\
49    while p lt bound do                             \n\

```

```

50     PR<x>:=PolynomialRing(GF(p));           \n\
51     pol:=PR!poly;                          \n\
52     if (HasRoot(pol)) eq true then         \n\
53         roots:=Roots(pol);                 \n\
54         for i:=1 to #roots do              \n\
55             tuple:=<0,p>;                 \n\
56             tuple[1]:=roots[i][1];         \n\
57             Append(~fbpa,tuple);          \n\
58         end for;                            \n\
59     end if;                                 \n\
60     p:=NextPrime(p);                       \n\
61     end while;                             \n\
62     return fbpa;                           \n\
63 end function;                              \n\
64 PR<x>:=PolynomialRing(Integers());         \n\
65 res:=InitAlgebraicFactorBase2(0*x+%s,%u); \n\
66 delim:=CodeToString(58); //The semi-colon character. \n\
67 str:=Substring(delim,1,0);                 \n\
68 for i:=1 to #res-1 do                      \n\
69     str:=str cat IntegerToString(res[i][1]) cat delim; \n\
70     str:=str cat IntegerToString(res[i][2]) cat delim; \n\
71 end for;                                   \n\
72 str:=str cat IntegerToString(res[#res][1]) cat delim; \n\
73 str:=str cat IntegerToString(res[#res][2]); \n\
74 str;                                       \n"
75
76 #define MAGMA_SCRIPTO "\
77 quadraticCharacterBase:=function(poly,bound,k) \n\
78     qcBase=[];                             \n\
79     p:=bound;                               \n\
80     while #qcBase lt k do                   \n\
81         p:=NextPrime(p);                   \n\
82         PR<x>:=PolynomialRing(GF(p));       \n\
83         pol:=PR!poly;                      \n\
84         if (HasRoot(pol)) eq true then     \n\
85             roots:= Roots(pol);           \n\
86             for j:=1 to #roots do          \n\
87                 pA:=[];                   \n\
88                 if #qcBase eq k then      \n\
89                     break;               \n\
90                 end if;                   \n\
91                 pA[1]:=Integers(!roots[j][1]); \n\
92                 pA[2]:=p;                 \n\
93                 Append(~qcBase,pA);       \n\
94             end for;                       \n\
95         end if;                             \n\
96     end while;                             \n\
97     return qcBase;                         \n\
98 end function;                              \n\
99 PR<x>:=PolynomialRing(Integers());         \n\
100 res:=quadraticCharacterBase(0*x+%s,%u,%u); \n\
101 delim:=CodeToString(58); //The semi-colon character. \n\
102 str:=Substring(delim,1,0);                 \n\
103 for i:=1 to #res-1 do                      \n\
104     str:=str cat IntegerToString(res[i][1]) cat delim; \n\
105     str:=str cat IntegerToString(res[i][2]) cat delim; \n\
106 end for;                                   \n\
107 str:=str cat IntegerToString(res[#res][1]) cat delim; \n\
108 str:=str cat IntegerToString(res[#res][2]); \n\
109 str;\n"
110
111 void nfs_init_factor_base_algebraic(FB_nfs fb, DATA_nfs d) {
112     int i = 0, j = 0;
113     fb->pa = (ui *) malloc(sizeof(ui) * (d->B_a) / 2);
114     fb->roots = (ui *) malloc(sizeof(ui) * (d->B_a) / 2);
115
116     gmp_sprintf(MAGMA_BUFFER, MAGMA_SCRIPT, d->poly, d->B_a);
117     char *fbStr = magma_exec(MAGMA_BUFFER);
118     char *token;
119     while ((token = strsep(&fbStr, ";")) {
120         if ((j % 2) == 0) {
121             fb->roots[i] = atoi(token);
122         } else if ((j % 2) == 1) {
123             fb->pa[i] = atoi(token);
124             i++;
125         }
126         j++;
127     }
128     fb->pa = (ui *) realloc(fb->pa, sizeof(ui) * i);
129     fb->roots = (ui *) realloc(fb->roots, sizeof(ui) * i);
130     fb->asize = i;
131
132 #ifdef VERBOSE
133     printf(" > i r p \n");
134     printf(" > -----\n");
135     for (i = 0; i < 10; i++) {
136         printf(" > %9d %9d %9d \n", i, fb->roots[i], fb->pa[i]);
137     }
138     printf(" > ... ..\n");
139     for (i = fb->asize - 5; i < fb->asize; i++) {
140         printf(" > %9d %9d %9d \n", i, fb->roots[i], fb->pa[i]);
141     }
142     //printf(" > %s", fbStr);
143     fflush(stdout);
144 #endif
145     free(fbStr);
146 }
147

```

```

148 void nfs_init_quadratic_characterbase(FB_nfs fb, DATA_nfs d) {
149     int j = 0, i = 0;
150     fb->qsize = 200;
151     gmp_sprintf(MAGMA_BUFFER, MAGMA_SCRIPT0, d->poly, 999999999, fb->qsize);
152     char *qcStr = magma_exec(MAGMA_BUFFER);
153     char *token;
154     fb->pq = (ui *) malloc(sizeof(ui) * (fb->qsize + 1));
155     fb->qroots = (ui *) malloc(sizeof(ui) * (fb->qsize + 1));
156
157     while ((token = strtok(&qcStr, ":")) {
158         if ((j % 2) == 0) {
159             fb->qroots[i] = atoi(token);
160         } else if ((j % 2) == 1) {
161             fb->pq[i] = atoi(token);
162             i++;
163         }
164         j++;
165     }
166
167     #ifdef VERBOSE
168     printf(" >          i          r          p \n");
169     printf(" > -----\n");
170     for (i = 0; i < 10; i++) {
171         printf(" > %9d %9d %9d \n", i, fb->qroots[i], fb->pq[i]);
172     }
173     printf(" >          ...          \n");
174     for (i = fb->qsize - 5; i < fb->qsize; i++) {
175         printf(" > %9d %9d %9d \n", i, fb->qroots[i], fb->pq[i]);
176     }
177     fflush(stdout);
178     #endif
179     free(qcStr);
180 }
181
182 void nfs_init_special_q_characterbase(FB_nfs fb, DATA_nfs d, ui q_bound) {
183     int i, j = 0;
184     fb->psq = (ui *) malloc(sizeof(ui) * (fb->asize));
185     fb->sqi = (ui *) malloc(sizeof(ui) * (fb->asize));
186     fb->sqroots = (ui *) malloc(sizeof(ui) * (fb->asize));
187     fb->rbsv = (LV_nfs_t *) malloc(sizeof(LV_nfs_t) * fb->asize);
188
189     for (i = 0; i < fb->asize; i++) {
190         if (fb->pa[i] > q_bound) {
191             fb->psq[j] = fb->pa[i];
192             fb->sqroots[j] = fb->roots[i];
193             fb->sqi[j] = i;
194             reduceBasis(fb->rbsv[j], fb->psq[j], fb->sqroots[j]);
195             j++;
196         }
197     }
198     fb->sqsize = j;
199     fb->psq = (ui *) realloc(fb->psq, sizeof(ui) * (fb->sqsize));
200     fb->sqi = (ui *) realloc(fb->sqi, sizeof(ui) * (fb->sqsize));
201     fb->sqroots = (ui *) realloc(fb->sqroots, sizeof(ui) * (fb->sqsize));
202     fb->rbsv = (LV_nfs_t *) realloc(fb->rbsv, sizeof(LV_nfs_t) * fb->sqsize);
203
204     #ifdef VERBOSE
205     printf(" >          i          r          p \n");
206     printf(" > -----\n");
207     for (i = 0; i < 10; i++) {
208         printf(" > %9d %9d %9d \n", fb->sqi[i], fb->sqroots[i], fb->psq[i]);
209     }
210     printf(" >          ...          \n");
211     for (i = fb->sqsize - 5; i < fb->sqsize; i++) {
212         printf(" > %9d %9d %9d \n", fb->sqi[i], fb->sqroots[i], fb->psq[i]);
213     }
214     fflush(stdout);
215     #endif
216
217 }

```

Code 6.3: ../elekmini/src/nfs/init_fb.c

```

1 #include "_elekmini.h"
2
3 /*
4 #define MAGMA_SCRIPT "\
5 InitPolynomial:=function(n,d) \n\
6     <x>:=PolynomialRing(Integers()); \n\
7     r:=Floor(n^(1/(d+1))); \n\
8     repeat \n\
9         r:=NextPrime(2*r); \n\
10        ks:=IntegerToSequence(n,r); \n\
11        until #ks ne d+1; \n\
12        m:=2*r; \n\
13    repeat \n\
14        m:=m div 2; \n\
15        m:=PreviousPrime(m); \n\
16        ks:=IntegerToSequence(n,m); \n\
17        f:=0; \n\
18        for i:=1 to #ks do \n\
19            f:= f + x^(i-1)*ks[i]; \n\
20        end for; \n\
21    until IsIrreducible(f) and (Degree(f) eq d) and IsMonic(f); \n\
22    assert Evaluate(f,m) eq n; \n\
23    delim:=CodeToString(58); \n\

```

```

24 | str:=Substring(delim,1,0); \n\
25 | cn:=Coefficients(f); \n\
26 | for i:=1 to #cn do \n\
27 | str:= str cat IntegerToString(cn[i]) cat delim; \n\
28 | end for; \n\
29 | str:= str cat IntegerToString(m) cat delim; \n\
30 | assert Evaluate(f,m) eq n; \n\
31 | return str,f; \n\
32 | end function; \n\
33 | InitPolynomial(%Zd,%u); \n"
34 | */
35 |
36 | #define MAGMA_SCRIPT "\
37 | InitPolynomial:=function(n,d) \n\
38 | m:=Floor(n^(1/d)); \n\
39 | _<x>:=PolynomialRing(Integers()); \n\
40 | k:=1; \n\
41 | repeat \n\
42 | k:=NextPrime(k); \n\
43 | ks:=IntegerToSequence(n,m); \n\
44 | f:=0; \n\
45 | for i:=1 to #ks do \n\
46 | if ks[i] le m then \n\
47 | f:=f + x^(i-1)*ks[i]; \n\
48 | end if; \n\
49 | end for; \n\
50 | if IsIrreducible(f) and (Degree(f) eq d) and IsMonic(f) then \n\
51 | break; \n\
52 | else \n\
53 | m:= m - k; \n\
54 | end if; \n\
55 | until IsIrreducible(f) and (Degree(f) eq d) and IsMonic(f); \n\
56 | //assert Evaluate(f,m) eq n; \n\
57 | delim:=CodeToString(58); \n\
58 | str:=Substring(delim,1,0); \n\
59 | cn:=Coefficients(f); \n\
60 | for i:=1 to #cn do \n\
61 | str:= str cat IntegerToString(cn[i]) cat delim; \n\
62 | end for; \n\
63 | str:= str cat IntegerToString(m) cat delim; \n\
64 | return str,f; \n\
65 | end function; \n\
66 | InitPolynomial(%Zd,%u); \n"
67 |
68 | #define MAGMA_SCRIPT1 "\
69 | InitPolynomial:=function(n,d) \n\
70 | m:= Floor(n^(1/(d))); \n\
71 | ks:=IntegerToSequence(n,m); \n\
72 | _<x>:=PolynomialRing(Integers()); \n\
73 | f:=0; \n\
74 | k:=1; \n\
75 | repeat \n\
76 | k:= NextPrime(k); \n\
77 | ks:=IntegerToSequence(n,m); \n\
78 | for i:=1 to #ks do \n\
79 | if ks[i] lt m then \n\
80 | f:= f + x^(i-1)*ks[i]; \n\
81 | end if; \n\
82 | end for; \n\
83 | if Degree(f) eq d then \n\
84 | break; \n\
85 | end if; \n\
86 | m:= m + (k*k); \n\
87 | until (Degree(f) eq d); \n\
88 | ks:= IntegerToSequence(n,m); \n\
89 | for i:=1 to #ks do \n\
90 | if ks[i] gt Floor(m/2) then \n\
91 | ks[i]:= ks[i] - m; \n\
92 | ks[i+1] += 1; \n\
93 | end if; \n\
94 | end for; \n\
95 | f:=0; \n\
96 | for i:=1 to #ks do \n\
97 | f:= f + x^(i-1)*ks[i]; \n\
98 | end for; \n\
99 | delim:=CodeToString(58); \n\
100 | str:=Substring(delim,1,0); \n\
101 | cn:=Coefficients(f); \n\
102 | for i:=1 to #cn do \n\
103 | str:= str cat IntegerToString(cn[i]) cat delim; \n\
104 | end for; \n\
105 | str:= str cat IntegerToString(m) cat delim; \n\
106 | return str,f; \n\
107 | end function; \n\
108 | InitPolynomial(%Zd,%u); \n"
109 |
110 | void nfs_init_poly(DATA_nfs d) {
111 | gmp_sprintf(MAGMA_BUFFER, MAGMA_SCRIPT1, d->n, d->d);
112 | char *polStr = magma_exec(MAGMA_BUFFER);
113 | char *token;
114 | int i = 0;
115 | d->polyCoefficient = malloc((d->d + 1) * sizeof(mpz_t));
116 | for (i = 0; i <= d->d; i++) {
117 | token = strtok(&polStr, "");
118 | mpz_init_set_str(d->polyCoefficient[i], token, 10);
119 | }
120 | token = strtok(&polStr, "");
121 | mpz_init_set_str(d->m, token, 10);

```



```

122 token = strsep(&polStr, ":");
123 d->poly = token;
124
125 free(polStr);
126
127 #ifdef VERBOSE
128 printf(" > %s\n", d->poly);
129 for (i = 0; i <= d->d; i++) {
130     gmp_printf("%Zd ", d->polyCoefficient[i]);
131 }
132 #endif
133 }
134
135 void nfs_kill_poly(DATA_nfs d) {
136     int i;
137     for (i = 0; i <= d->d; i++) {
138         mpz_clear(d->polyCoefficient[i]);
139     }
140     mpz_clear(d->m);
141 }

```

Code 6.4: ../elekmini/src/nfs/init_poly.c

```

1 #include "_elekmini.h"
2 #include "ecm.h"
3 #include <sys/time.h>
4 #include <time.h>
5 #include <stdio.h>
6 #define TRIAL_DIV_MUL
7 // #define TRIAL_DIV
8 // #define PRINT_SMOOTHS
9 // #define PRINT_RELATIONS
10
11 #define THRESHOLD_MODE 0
12 #define TRIAL_DIV_MODE 0
13 #define SUPPORT_PROCESSOR_x86_64 1
14
15 #ifdef SUPPORT_PROCESSOR_x86_64
16
17 #define km_mul_2(zH, zL, a, b) __asm__( \
18     "mulq %3;" \
19     : "=d"((zH)), "=a"((zL)) \
20     : "a"((a)), "m"((b)) \
21 )
22 #endif
23
24 int trialDivisionZ(DATA_nfs d, FB_nfs fb, uc *arr, mpz_t res, sl a) {
25     int j;
26     /* res calculation out of function */
27     ui p;
28     for (j = 0; j < fb->rsize; j++) {
29         arr[j] = 0;
30     }
31
32     for (j = 0; j < fb->rsize; j++) {
33         p = fb->pr[j];
34         while (mpz_divisible_ui_p(res, p) > 0) {
35             mpz_divexact_ui(res, res, p);
36             arr[j]++;
37         }
38     }
39
40     if (mpz_cmp_ui(res, 1) == 0 || mpz_cmp_si(res, -1) == 0 || mpz_sizeinbase(res, 2) < d->lpb0) {
41         return 1;
42     } else {
43         //printf("Rational side is not smooth\n");
44         return 0;
45     }
46 }
47
48 int trialDivisionA(DATA_nfs d, FB_nfs fb, uc *arr, mpz_t res, int b, sl a, FILE *fp) {
49     int i;
50     mpz_t A1, A2, A3;
51     ui p, r;
52     mpz_init_set_si(A1, a);
53     mpz_init_set_si(A2, 1);
54     mpz_init(A3);
55
56     for (i = 0; i < fb->asize; i++) {
57         arr[i] = 0;
58     }
59
60     for (i = 0; i < fb->asize; i++) {
61         p = fb->pa[i];
62         r = fb->roots[i];
63         mpz_set_ui(A3, b);
64         mpz_mul_ui(A2, A3, r);
65         mpz_mod_ui(A2, A2, p);
66         mpz_mod_ui(A3, A1, p);
67         if (mpz_cmp(A3, A2) == 0) {
68             while ((mpz_divisible_ui_p(res, p)) != 0) {
69                 mpz_divexact_ui(res, res, p);
70                 arr[i]++;
71             }
72         }
73     }

```

```

74 |
75 | mpz_clear(A3);
76 | mpz_clear(A2);
77 | mpz_clear(A1);
78 |
79 | if (mpz_cmp_ui(res, 1) == 0 || mpz_cmp_si(res,-1) == 0 || mpz_sizeinbase(res, 2) < d->lbp1) {
80 |     return 1;
81 | } else {
82 |     /*fprintf(fp, "a:=");
83 |     mpz_out_str(fp, 10, res);
84 |     fprintf(fp, "; Log(2,a);\n");
85 |     fprintf(fp, "assert IsPrime(a);\n");*/
86 |     return 0;
87 | }
88 | }
89 |
90 | int trialDivAWithMul(DATA_nfs d, FB_nfs fb, uc *arr, mpz_t res, int b, sl a, float *compTime) {
91 |     int i;
92 |     struct timeval t_start[1], t_end[1], t_diff[1];
93 |     mpz_t A1, A2, A3, res_mul;
94 |     ui r;
95 |     mpz_init_set_si(A1, a);
96 |     mpz_init_set_ui(res_mul, 1);
97 |     mpz_init_set_si(A2, 1);
98 |     ul mul_small = 1, mul_small_pre, zh, p;
99 |     mpz_init(A3);
100 |     for (i = 0; i < fb->asize; i++) {
101 |         arr[i] = 0;
102 |     }
103 |
104 |     for (i = 0; i < d->smallSieveSize; i++) {
105 |         p = fb->pa[i];
106 |         r = fb->roots[i];
107 |         mpz_set_ui(A3, b);
108 |         mpz_mul_ui(A2, A3, r);
109 |         //mpz_mul_si(A2, A2, -1); /* a+b*alpha*/
110 |         mpz_mod_ui(A2, A2, p);
111 |         mpz_mod_ui(A3, A1, p);
112 |         if ( mpz_cmp(A3, A2) == 0) {
113 |             if (mpz_divisible_ui_p(res, fb->pa[i]) > 0) {
114 |                 mul_small_pre = mul_small;
115 |                 km_mul_2(zh, mul_small, mul_small, p);
116 |                 if (zh > 0) {
117 |                     mpz_divexact_ui(res, res, mul_small_pre);
118 |                     mul_small = 1;
119 |                 }
120 |                 arr[i]++;
121 |             }
122 |         }
123 |     }
124 |     mpz_divexact_ui(res, res, mul_small);
125 |
126 |     for (i = 0; i < fb->asize; i++) {
127 |         p = fb->pa[i];
128 |         r = fb->roots[i];
129 |         mpz_set_ui(A3, b);
130 |         mpz_mul_ui(A2, A3, r);
131 |         mpz_mod_ui(A2, A2, p);
132 |         mpz_mod_ui(A3, A1, p);
133 |         if ( mpz_cmp(A3, A2) == 0) {
134 |             while ((mpz_divisible_ui_p(res, p)) != 0) {
135 |                 mpz_divexact_ui(res, res, p);
136 |                 arr[i]++;
137 |             }
138 |         }
139 |     }
140 |     mpz_clear(A3);
141 |     mpz_clear(A2);
142 |     mpz_clear(A1);
143 |
144 |     if (mpz_cmp_ui(res, 1) == 0 || mpz_cmp_si(res,-1) == 0) {
145 |         return 1;
146 |     } else {
147 |         return 0;
148 |     }
149 | }
150 |
151 | int trialDivZWithMul(DATA_nfs d, FB_nfs fb, uc *arr, mpz_t res, sl a, float *compTime) {
152 |     mpz_t res_mul;
153 |     ul mul_small = 1, mul_small_pre = 1, zh = 0, p;
154 |     mpz_init_set_ui(res_mul, 1);
155 |     int j;
156 |
157 |     for (j = 0; j < fb->rsize; j++) {
158 |         arr[j] = 0;
159 |     }
160 |
161 |     for (j = 0; j < d->smallSieveSize; j++) {
162 |         p = fb->pr[j];
163 |         if (mpz_divisible_ui_p(res, fb->pr[j]) > 0) {
164 |             mul_small_pre = mul_small;
165 |             km_mul_2(zh, mul_small, mul_small, p);
166 |             if (zh > 0) {
167 |                 mpz_divexact_ui(res, res, mul_small_pre);
168 |                 mul_small = 1;
169 |             }
170 |             arr[j]++;
171 |         }

```



```

172 }
173 mpz_divexact_ui(res, res, mul_small);
174
175 for (j = 0; j < fb->rsize; j++) {
176   p = fb->pr[j];
177   while (mpz_divisible_ui_p(res, p) > 0) {
178     mpz_divexact_ui(res, res, p);
179     arr[j]++;
180   }
181 }
182
183 if (mpz_cmp_ui(res, 1) == 0 || mpz_cmp_si(res, -1) == 0) {
184   return 1;
185 } else {
186   return 0;
187 }
188 }
189
190 void sieveNFS(DATA_nfs d, FB_nfs fb, int b, int sign) {
191   int i, j, posStartPoint = 0, negStartPoint = 0;
192   mpz_t T0;
193   mpz_init(T0);
194   ui p, r;
195   double pLog;
196   /* initialize sieve array for further sieve */
197   for (i = 0; i < (d->ar * 2); i++) {
198     d->arrSieveR[i] = (double) 0;
199     d->arrSieveA[i] = (double) 0;
200   }
201
202   /* Rational Side Line Sieve */
203   for (i = 0; i < fb->rsize; i++) {
204     p = fb->pr[i];
205     pLog = log2((double) p);
206     posStartPoint = (b * fb->rroots[i]) % p;
207     negStartPoint = posStartPoint - p;
208     negStartPoint *= -1;
209
210     for (j = negStartPoint; j < d->ar; j = j + p) {
211       d->arrSieveR[j] = d->arrSieveR[j] + pLog;
212     }
213     for (j = posStartPoint + (d->ar); j < (d->ar * 2); j = j + p) {
214       d->arrSieveR[j] = d->arrSieveR[j] + pLog;
215     }
216   }
217
218   /* Algebraic Side Line Sieve */
219   for (i = 0; i < fb->asize; i++) {
220     p = fb->pa[i];
221     pLog = log2(p);
222     mpz_set_ui(T0, fb->roots[i]);
223     mpz_mul_ui(T0, T0, b);
224     mpz_mod_ui(T0, T0, p);
225     posStartPoint = mpz_get_ui(T0);
226     negStartPoint = posStartPoint - p;
227     negStartPoint *= -1;
228     for (j = negStartPoint; j < d->ar; j = j + p) {
229       d->arrSieveA[j] = d->arrSieveA[j] + pLog;
230     }
231     for (j = posStartPoint + (d->ar); j < (d->ar * 2); j = j + p) {
232       d->arrSieveA[j] = d->arrSieveA[j] + pLog;
233     }
234   }
235
236   mpz_clear(T0);
237
238 }
239
240 inline ui calculateStartPointForLatticeSieve(LV_nfs_t lv, int j, ui p, ui r) {
241   ul latticeSieveStartPoint, modInv;
242   int res;
243   mpz_t temp, temp0, temp1;
244   mpz_init(temp1);
245   res = lv->v1[0] - r * lv->v1[1];
246   mpz_init_set_si(temp, res);
247   mpz_init_set_ui(temp0, p);
248   mpz_invert(temp1, temp, temp0);
249   modInv = mpz_get_ui(temp1);
250   res = ((j * (r * lv->v2[1] - lv->v2[0])) * modInv);
251   mpz_set_si(temp, res);
252   mpz_mod_ui(temp, temp, p);
253   latticeSieveStartPoint = mpz_get_ui(temp);
254   mpz_clear(temp0);
255   mpz_clear(temp);
256   mpz_clear(temp1);
257   return latticeSieveStartPoint;
258 }
259
260 void latticeSieveByRowsNFS(DATA_nfs d, FB_nfs fb, int sqIndex) {
261   int i = 0, j = 0, k = 0;
262   ui p, r;
263   ui i0; /* Sieve start point for lattice */
264   ui c_1, c_d;
265   double pLog;
266   for (i = 0; i < (2 * d->rangeC * d->rangeE); i++) {
267     d->arrLatticeSieveA[i] = (double) 0;
268     d->arrLatticeSieveR[i] = (double) 0;
269   }

```

```

270
271 for (k = 0; k < fb->sqi[sqIndex]; k++) {
272     p = fb->pa[k];
273     r = fb->roots[k];
274     pLog = log2((double) p);
275     j=1;
276     c_1 = calculateStartPointForLatticeSieve(fb->rbsv[sqIndex], j, p, r);
277     c_d = 0;
278     for (j = 1; j < d->rangeE; j++) {
279         /* Negative part sieve (-C,0) */
280         i0 = (c_1 + c_d) % p;
281         for (i = (p - i0); i < d->rangeC; i += p) {
282             d->arrLatticeSieveA[((j - 1) * d->rangeC) + i] += pLog;
283         }
284         /* Positive part sieve (0, C) */
285         for (i = (d->rangeC + i0); i < (2 * d->rangeC); i += p) {
286             d->arrLatticeSieveA[((j - 1) * d->rangeC) + i] += pLog;
287         }
288         c_d = i0;
289     }
290 }
291
292 for (k = 0; k < fb->rsize; k++) {
293     p = fb->pr[k];
294     r = fb->rroots[k];
295     pLog = log2((double) p);
296     c_1 = calculateStartPointForLatticeSieve(fb->rbsv[sqIndex], j, p, r);
297     c_d = 0;
298     for (j = 1; j < d->rangeE; j++) {
299         i0 = (c_1 + c_d) % p;
300         /* Negative part sieve (-C,0) */
301         for (i = (p - i0); i <= d->rangeC; i += p) {
302             d->arrLatticeSieveR[((j - 1) * d->rangeC) + i] += pLog;
303         }
304         /* Positive part sieve (0, C) */
305         for (i = (d->rangeC + i0); i <= (2 * d->rangeC); i += p) {
306             d->arrLatticeSieveR[((j - 1) * d->rangeC) + i] += pLog;
307         }
308         c_d = i0;
309     }
310 }
311 }
312
313
314 /* NFS with Line Sieving */
315 int compute_numberfield_sieve(REL_FULL_NFS fulls, REL_PARTIAL_NFS partials, FB_nfs fb, DATA_nfs d, POL p,
    LARGE_PRIME_DATA hd, FILE *fp) {
316     int i, j, partial_data_count = 0, cols, rows = 0, check, extra = 10;
317     int b = 1;
318     //char temp = 0;
319     ui gcd;
320     sl a;
321     //float timeComp = 0;
322     int sign = 0;
323     int half_to_full_rel = 0;
324     cols = fb->asize + fb->rsize + fb->qsize;
325     svz_hash_t *hashtable_p = svz_hash_create(100000); //extra_prime hash
326     svz_hash_t *hashtable_r = svz_hash_create(100000);
327     mpz_t res0, res1, T0, T1, T2, T3;
328     mpz_init_set_ui(res0, 0);
329     mpz_init_set_ui(res1, 0);
330     mpz_init(T0);
331     mpz_init(T1);
332     mpz_init(T2);
333     mpz_init(T3);
334
335     printf("\n cols : %d \n", cols);
336     uc *arrTrialA = (uc *) malloc(sizeof(uc) * (fb->asize)); /*plus 1 because of sign information*/
337     uc *arrTrialR = (uc *) malloc(sizeof(uc) * (fb->rsize));
338
339     while (rows < cols + extra) {
340         sieveNFS(d, fb, b, 1);
341         for (i = 0; i < (d->ar * 2); i++) {
342             #if THRESHOLD_MODE == 0
343                 if ((d->arrSieveR[i] > d->thrR) && (d->arrSieveA[i] > d->thrA)) {
344
345                 #elif THRESHOLD_MODE == 1
346                     if ((d->arrSieveR[i] > d->thrR)) {
347                 #elif THRESHOLD_MODE == 2
348                     if ((d->arrSieveA[i] > d->thrA)) {
349                 #endif
350                     if (i < d->ar) {
351                         a = i * (-1);
352                     } else {
353                         a = i - d->ar;
354                     }
355                     /* GCD(a,b) = 1 check */
356                     gcd = binary_gcd(a, b);
357                     if (gcd == 1) {
358                 #if TRIAL_DIV_MODE == 0
359                         mpz_norm1(d, res0, a, b, T2); /* a-b*alpha */
360                         check = trialDivisionA(d, fb, arrTrialA, res0, b, a, fp);
361                 #elif TRIAL_DIV_MODE == 1
362                         mpz_mul_ui(T3, d->m, b);
363                         mpz_set_si(res1, a);
364                         mpz_sub(res1, res1, T3); /* a-b*m */
365                         check = trialDivisionZ(d, fb, arrTrialR, res1, a);
366                 #endif

```

```

367     if (check == 1) {
368 #if TRIAL_DIV_MODE == 0
369         /* a-b*m */
370         mpz_mul_ui(T3, d->m, b);
371         mpz_set_si(res1, a);
372         mpz_sub(res1, res1, T3);
373         check = trialDivisionZ(d, fb, arrTrialR, res1, a);
374 #elif TRIAL_DIV_MODE == 1
375         mpz_norm1(d, res0, a, b, T2); /* a-b*alpha */
376         check = trialDivisionA(d, fb, arrTrialA, res0, b, a);
377 #endif
378
379     if (check == 1) {
380         rows += nfs_relation_processing(d, fb, fulls, partials, arrTrialR, arrTrialA, res0, res1, T0,
381             T1, sign, a, b, rows, hashtable_p, hashtable_r, &half_to_full_rel);
382     }
383     } else {
384         //printf("GCD a and be not equal 1\n");
385     }
386 }
387 }
388
389 printf("\n ROW is : %d \n", rows);
390 //printf("half to full relations count is %d\n",half_to_full_rel);
391 b++;
392 }
393
394 linear_algebra_nfs(d, fb, fulls, rows);
395 free(arrTrialR);
396 free(arrTrialA);
397 mpz_clear(T0);
398 mpz_clear(T2);
399 mpz_clear(T1);
400 mpz_clear(T3);
401 mpz_clear(res1);
402 mpz_clear(res0);
403 return rows;
404 }
405
406 /* NFS Lattice Sieve By Row */
407 int compute_numberfield_sieve_lattice(REL_FULL_NFS fulls, REL_PARTIAL_NFS partials, FB_nfs fb, DATA_nfs d,
408     POL p, LARGE_PRIME_DATA hd, LV_nfs lv, FILE *fp) {
409     int j, i, cols, partial_data_count, rows = 0, check, extra = 200, sqRange = 0, c, e, b, sign = 0;
410     sl a;
411     int half_to_full_rel = 0;
412     cols = fb->asize + fb->rsize + fb->qsize;
413     mpz_t res0, res1, T0, T1, T2, T3;
414     mpz_init(res0);
415     mpz_init(res1);
416     mpz_init(T0);
417     mpz_init(T1);
418     mpz_init(T2);
419     mpz_init(T3);
420
421     uc *arrTrialA = (uc *) malloc(sizeof(uc) * (fb->asize));
422     uc *arrTrialR = (uc *) malloc(sizeof(uc) * (fb->rsize));
423
424     svz_hash_t *hashtable_p = svz_hash_create(100000); //extra_prime hash
425     svz_hash_t *hashtable_r = svz_hash_create(100000); //extra_prime hash
426     printf("sq-size is %d\n", fb->sqsize);
427     printf("\n cols : %d \n", cols);
428     while (rows < cols + extra) {
429         latticeSieveByRowsNFS(d, fb, sqRange);
430         for (i = 1; i <= d->rangeE; i++) {
431             for (j = 0; j < (d->rangeC * 2); j++) {
432                 if ((binary_gcd(i, j) == 1) && (d->arrLatticeSieveA[((i - 1) * d->rangeC) + (j)] > d->thrA) && (d->
433                     arrLatticeSieveR[((i - 1) * d->rangeC) + (j)] > d->thrR)) {
434                     if (j < d->rangeC) {
435                         c = -j;
436                     } else {
437                         c = j - d->rangeC;
438                     }
439                     e = i;
440                     a = fb->rbsv[sqRange]->v1[0] * c + fb->rbsv[sqRange]->v2[0] * e;
441                     b = fb->rbsv[sqRange]->v1[1] * c + fb->rbsv[sqRange]->v2[1] * e;
442                     if (b < 0) {
443                         b = b * (-1);
444                         a = a * (-1);
445                     }
446                     /* GCD(a,b) = 1 check */
447                     if (binary_gcd(a, b) == 1) {
448                         mpz_norm1(d, res0, a, b, T2);
449                         check = trialDivisionA(d, fb, arrTrialA, res0, b, a, fp);
450                         if (check == 1) {
451                             mpz_mul_ui(T3, d->m, b);
452                             mpz_set_si(res1, a);
453                             mpz_sub(res1, res1, T3);
454                             check = trialDivisionZ(d, fb, arrTrialR, res1, a);
455                             if (check == 1) {
456                                 rows += nfs_relation_processing(d, fb, fulls, partials, arrTrialR, arrTrialA, res0, res1, T0,
457                                     T1, sign, a, b, rows, hashtable_p, hashtable_r, &half_to_full_rel);
458                             }
459                         } else {
460                             //printf("GCD a and be not equal 1\n");
461                         }
462                     }
463                 }
464             }
465         }
466     }

```

```
461     }
462   }
463   printf("\n Found %d out of %d relations\n", rows, (cols + extra));
464   sqRange++;
465
466 }
467
468 linear_algebra_nfs(d, fb, fulls, rows);
469 free(arrTrialR);
470 free(arrTrialA);
471 mpz_clear(res0);
472 mpz_clear(res1);
473 mpz_clear(T0);
474 mpz_clear(T1);
475 mpz_clear(T2);
476 mpz_clear(T3);
477 return rows;
478 }
```

Code 6.5: ../elekmini/src/nfs/compute_nfs.c