Viet Pham Hoang

INTEGER FACTORIZATION WITH THE GENERAL NUMBER FIELD SIEVE

Rovaniemen
ammattikorkeakoulu
University of Applied Sciences

Rovaniemen ammattikorkeakoulun

julkaisusarja B 12

Rovaniemi University of Applied Sciences

School of Technology

Degree Programme in Information Technology

Thesis

INTEGER FACTORIZATION WITH THE GENERAL NUMBER FIELD SIEVE

Pham Hoang Viet

2008

Rovaniemi University of Applied Sciences
Publications
Jokiväylä 11 C
96300 Rovaniemi
Finland
tel. +358 20 798 4000
www.ramk.fi/julkaisutoiminta
julkaisut@ramk.fi

ISSN: 1239-7733
ISBN: 978-952-5153-77-4 (vol)
ISBN: 978-952-5153-78-1 (PDF)

Rovaniemi University of Applied Sciences Publication Serie B no. 12

Rovaniemi 2008
Tornion kirjapaino

Abstract

Since Fermat's work on integer factorization, the mathematical community has experienced substantial research and improvements following his method of decomposing integers. This mathematical aspect has nowadays been utilized in a number of applications, such as in testing the security level of several encryption methods like the RSA algorithm. As the latest achievement, the *General Number Fields Sieve* (GNFS) has recently been recognized as the fastest algorithm of this kind that is being used for factoring integers of size up to 800 bits in length.

Considering the topic of this thesis, it concentrates on explaining the most significant details necessary for understanding and implementing the GNFS. In particular, it exploits the fact that this algorithm is a *sieve-like* process which factors *general* form integers using a variety of results from the *number field* theory. In this aspect, the thesis firstly addresses the underlying principle as well as many other important concepts used for constructing the GNFS. Then, as the implementation note moves on, several computing techniques will be plugged in that support the transition of this algorithm from theory into practice. In addition, the study also presents an extended setup for the Condor *distributed computing* system that can be used for parallel executions of this factorization technique. Ultimately, an extended example given in the last chapter generally illustrates how the algorithm works in practice by showing the data produced at every step, starting from the input modulus and ending with the output factors.

Keywords: number field sieve, factorization, rsa, distributed computing

Contents

List of Tables

List of Figures

Foreword

Originally this study is a thesis work at the Rovaniemi University of Applied Sciences (RAMK).

The publication has many exceptional features. The author, Viet Pham Hoang, came from a special school for mathematically oriented students in Vietnam to Rovaniemi somewhat by accident, having seen the announcement of our Degree Programme in Information Technology in 2004.

In the thesis, Viet shows his mathematical talent in an outstanding way. He masters abstract algebra and number theory far beyond the level we require in our courses. He has reconstructed many of the propositions and their proofs by himself. This did not come as a surprise, because that is what Viet has been doing all the time during his studies.

The thesis is not only theoretical. The problem of integer factorization is central for the safety of the public key cryptography we use every day. The fastest method in effective use today for factorization of large integers (e.g. RSA modulus) is the General Number Field Sieve (GNFS). In Viet's thesis, the history and mathematical foundation of this method are explained.

Furthermore, Viet has written a large amount of code for demonstrating the GNFS method and for distributing the computations to a computer network (GRID). Viet not only constructed this distributed computing environment but was also responsible for the production runs that led to the discovery of a remarkable structure of words in the field of theoretical computer science.

For us it has been a great pleasure to follow the development of this outstanding thesis.

Jouko Teeriaho Senior Lecturer, Thesis Supervisor
Veikko Keränen Principal Lecturer of Mathematics

Chapter 1

Introduction to integer factorization

With the exception of Shor's algorithm [29, p. 14], the General Number Field Sieve (GNFS) is known as the fastest method for factoring large integers, as long as quantum computers are yet to be rigorously invented. Like other modern sieving techniques such as the Quadratic Sieve and Dixon's algorithm, the GNFS exploits the same idea of *quadratic residues* to form two independent squares, for which the Euclidean algorithm is then applied to construct the prime factors [4, p. 10].

In addition, due to the large amount of computation required, the GNFS was designed to support parallel computing, which further speeds up the actual computation by the number of processors involved. As a result, various achievements have been recorded in efforts of integer factorization, such as the 193-digit challenge number RSA-640 in 2005 [34]. This sequence of breakthroughs has urgently raised a question about the security level of the RSA algorithm, a public key encryption method being used in almost every major information system nowadays.

Before diving deeper into the complexity of the GNFS, it is important to understand how integer factorization could be used in the cryptanalysis of the RSA encryption method. Thus, this chapter begins by giving a brief introduction to the operation of the RSA algorithm, after which it points out the principle which strongly depends on the difficulty of factoring an integer. In the end of the chapter, the *Quadratic Sieve* algorithm is presented as a proof that the RSA algorithm is being seriously considered for its security issues.

1.1 The Public Key Infrastructure (PKI)

The main cryptographic concern of the GNFS is in *asymmetric encryption*, a methodology of securing transmission of data over the public. The cryptographic idea was first initiated in 1976 which introduced the concept of public-private key pairs [28, Chapter 2, p. 31]. In the latter stage of development, with the burst of the World Wide Web internationally, the PKI was designed as a solution for the security of Internet communications, especially in the aspect of object authentication, such as human or computers.

In practice, a PKI allows remote users to explicitly identify themselves or authenticate the others before exchanging any information. This can be facilitated with the use of a unique certificate for each entity, or object. In order for these certificates to be safely valid, they must be created, signed and monitored by the so-called *Certificate Authority* (CA), an organization which is recognized worldwide as a leader in the field of information security, such as VeriSign or Geotrust. Moreover, each of the well known CAs can act as the root of a hierarchical tree on which each node is a smaller CA responsible for certain areas of business, but they are trusted as an employee of the root CA. This structure divides the workload pushing on the root CA and provides the scalability as the number of certificates grows up. [19, pp. 4-5]

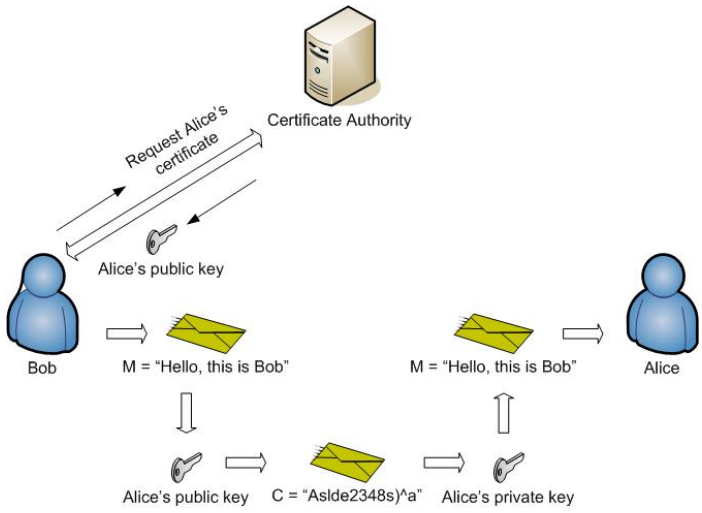

Figure 1.1: A visual demonstration of general PKI

Theoretically, to assure a secured communication, Bob first queries the CA for a certificate which can be used to authenticate Alice. Upon receiving Alice's certificate from the CA, Bob extracts from it a *public key* e that can be used to latter facilitate the underlying asymmetric algorithm. Depending on the nature of the algorithm indicated in the certificate, Bob invokes a transformation E to encrypt his outgoing message M with the help of the key e, and thus he obtains a cipher $C = E_e(M)$.

As the original message scrambled, its content C transmitting over the public seems to consist of random characters, which are meaningless to everyone but Alice, the owner of the private key. Indeed, knowing the encryption protocol, Alice can make use of a decryption transformation D in conjunction with his *private key* d to reconstruct the actual message, resulting $M = D_d(C)$. [16, Chapter 8, p. 283]

Based on specific communication protocols (e.g. *Secure Socket Layer*, or SSL), PKI can be applied in conjunction with other cryptographic methods such as *symmetric encryptions* to form a protected and effective data transmission environment. Meanwhile, its security feature relies on the difficulty of reversing the sniffed cipher C back to its original value M, even though attackers may be aware of the public key e as well as the two transformations E and D. As a result, it is the design of the encryption algorithm that is vital for the success of the PKI.

1.2 RSA cryptographic algorithm

Named after its three creators (Rivest, Shamir and Adleman), the RSA algorithm was publicly released in 1977 as a strong and simple asymmetric algorithm used in PKI applications. Moreover, this method also benefits from an advantage of security dependency on each particular key size being used, thus it provides the possibility of highly secured communication as long as the key pair is maintained with acceptable length. [28, Chapter 19, pp. 466-467]

In its simplest form, the RSA operates on the basis of two randomly chosen large prime integers, namely p and q. The product n of these primes, called the RSA modulus, is expected to be very difficult for factorization back to its factors. Meanwhile, an RSA public exponent e is chosen to be relatively prime to Euler's totient function $\phi(n)$, i.e., $\gcd(e, \phi(n)) = 1$. This consequently forms the RSA private exponent d such that $d \cdot e \equiv 1 \pmod{\phi(n)}$,

i.e., $d \cdot e = k \cdot \phi(n) + 1$. Apparently, the public and private keys are of the forms (e, n) and (d, p, q), respectively. [27, Chapter 3, p. 6]

Since every calculation takes place numerically, a message needing to be encrypted must be broken into chunks of characters so that each block can be encoded into an integer M that does not exceed the RSA modulus, i.e., $M < n$. This condition is required by the encryption transformation E_e that produces the ciphered version C of M, where C is defined as

$$C = E_e(M) = M^e (\text{mod } n) \tag{1.1}$$

The encrypted message C is then transmitted over whatever intended communication protocols (HTTP, SMTP, etc.) before reaching the targeting receiver at the other end of the communication channel. Upon complete receipt of the message, the receiver then uses a special transformation D to decrypt the message, as given below for the case of the RSA [4, Chapter 1, p. 2]:

$$C^d \equiv (M^e)^d \equiv M^{ed} \equiv M^{k \cdot \phi(n)+1} \equiv (M^{\phi(n)})^k \cdot M \equiv M \ (\text{mod } n)$$

This decryption algorithm assumes the following proposition, which indeed is true with the support of Euler's theorem and the nature of the RSA modulus n as a product of *distinct* primes:

Proposition 1.2.1. *Let n be a positive integer of distinct prime factors, then for every positive integer $M < n$ it holds true that $M \cdot (M^{\phi(n)})^k \equiv M \ (\text{mod } n)$, with k being any positive integer.*

1.3 Special purpose attacks to the RSA

Like many other security mechanisms, there have been various attempts by cryptanalysts trying to identify the weaknesses of the RSA under both general and special circumstances. These activities even though did not make any direct contribution to the evolvement of the algorithm, they have posed serious considerations on how the RSA should be deployed so as to take its full advantage and to prevent unexpected side effects.

Among those attack vectors, a method called *adaptive chosen ciphertext attack* is a well known example of such popular attempts. As its name implies, this attack works by selectively manipulating a ciphertext before it is released for the decryption process. Particularly, in order to get a message M sent by Bob, the attacker Eve needs to sniff the ciphertext C produced

by a cryptosystem ENCRYPTOR as in (1.1). To further perform the attack, Eve first chooses a random integer $R < n$ which he then uses to generate an altered ciphertext $\bar{C}$ of C such that $\bar{C} = C \cdot X^e \pmod{n}$, where $X = R^e \pmod{n}$. By sending $\bar{C}$ to ENCRYPTOR, Eve expects to receive a plaintext, yet meaningful message $\bar{M} = \bar{C}^d \pmod{n}$. Using $\bar{M}$ and R, the original message M can be derived as follows[28, p. 471]:

$$\bar{M} \cdot R^{-1} \equiv \bar{C}^d \cdot R^{-1} \equiv C^d \cdot X^{ed} \cdot X^{-1} \equiv C^d \cdot X^{k\phi(n)} \equiv C^d \equiv M \pmod{n}$$

Considering possible countermeasures for this security vulnerability, it is advisable that the RSA is to be used in conjunction with other cryptographic methods such as hash functions to make a presigned message. Thus, the decrypting system would have a way to determine whether the decrypted message is invalid, and discard it before it reaches the eavesdropper Eve.

Apart from the theoretical context, the RSA in early days can also be compromised when it comes into implementation. As an example, the interesting *timing attack* is primitively used to discover the private exponent d of the RSA private key stored inside a cryptosystem. In this case, the attack treats d as a binary number of k bits $d_{k-1}d_{k-2}\cdots d_1d_0$ and assumes that the cryptosystem makes use of *successive squares* as the computing mechanism for the encryption process, based on the observation that

$$C \equiv M^{d_0}(M^2)^{d_1}(M^4)^{d_2}\cdots(M^{2^{k-1}})^{k-1} \equiv \prod_{i=0}^{k-1} M^{2^i d_i} \pmod{n}$$

In computing a particular bit d_i of the private exponent, the attacker tries to generate a number of messages M satisfying that the computation of $C_{i-1}M^{2^i} \pmod{n}$ is extremely slow, where $C_{i-1} = \prod_{j=0}^{i-1} M^{2^j d_j} \pmod{n}$. Obviously, the above computation will not take place unless d_i is set. By observing the computation for a large number of messages, an averagely slow response would mean that this bit is 1, otherwise it is 0. After retrieving d_i, the attacker is able to compute C_i and repeat the process until the final bit is found. [13, p. 2]

Despite the complexity of the method, it is explicitly assumed that the attacker is aware of the entire operation of the target system. Moreover, it is relatively easy to overcome this problem, as the system designer may just implement a time padding scheme to reduce the differences in encrypting time among different messages of closed length, and thus significantly affects the prediction of the attack.

Other forms of attacks also include (but not limited to) breaking small exponents in RSA keys [13, pp. 3-11], or exploiting misuse of the RSA algorithm such as in *common modulus attacks*. However, the success of such attacks did not imply any weaknesses in the principle of the RSA, but rather pointed out the necessity of an appropriate implementation, and that the RSA must be deployed in cooperation with other security measures such as symmetric encryptions or hash functions to produce a safer environment.

1.4 The idea of integer factorization

It is obvious that the aforementioned cryptanalytic methods only exploit specific aspects of the RSA algorithm. Most of them are feasible in theory and can hardly be practically implemented due to the severity of their assumptions. In fact, these types of attacks did not take into consideration the security dependency that the RSA encryption relies on. Considering the case of *asymmetric encryptions* in general and of the RSA specifically, the level of security relies on how hard it is to determine the private key from the public key. In other words, given an RSA modulus n and its corresponding public exponent e, it should be practically impossible to compute the appropriate private exponent d which represents the private part of the key pair.

In order to extract d, it is important to note that $d \cdot e \equiv 1 \pmod{\phi(n)}$. If the value of $\phi(n)$ is known, it is relatively straightforward to compute d as the *multiplicative inverse* of e, using the extended Euclidean algorithm [16, Chapter 2, p. 71]. The problem of finding the private key now turns into seeking the value of the totient function ϕ of n, which is the product $(p-1)(q-1)$ if we consider the simplest form of the RSA modulus as $p \cdot q$, where p and q are two distinct prime numbers. Likewise, computing $\phi(n)$ requires the attacker to hold the values of p and q, which can only be collected by factoring n - the RSA modulus.

For small value of n, e.g., less than 15 digits, it is feasible to use a $brute force$ method that tries all possible prime numbers p and check if they are divisible by n. A success will automatically indicate q, and hence reveal the value of the private key. However, for such reason nowadays most cryptosystems are using RSA keys with modulus of length 1024 bits, or approximately 300 digits in decimal length. Breaking them using this $brute force$ method would require thousands of PCs running for thousands of years which is practically impossible. In this case, it is necessary to employ a better sieving technique to enormously reduce the number of prime divisions.

Following this idea, there exist several factoring methods, each of which has achieved certain successes upon their creations in factoring n. However, it seems that most of these methods base on the same principle that dates back to Fermat's time when he originally used the idea of *congruence of squares*. In particular, by finding two independent integer x and y such that $x^2 \equiv y^2 \pmod{n}$, it is possible to construct the factors of n using $\gcd(x-y, n)$ and $\gcd(x+y, n)$. To be more precise, it is easy to see that

$$x^2 - y^2 = (x-y)(x+y) = kn$$

for some integer k. Since n comprises two distinct primes p and q, it is necessary that n divides either $(x-y)$ or $(x+y)$, or alternatively its two factors are equally distributed, e.g., $p|(x-y)$ and $q|(x+y)$. There is however a higher probability that the latter situation would suffice, in which case the prime factors can be collected from $\gcd(x-y, n)$ and $\gcd(x+y, n)$. [4, p. 4]

As an extensive example of factoring methods inheriting the above Fermat's idea, the *Quadratic Sieve* restricts the search of x and y to a *rational factor base*, defined as

Definition 1.4.1. *A* rational factor base F *of bound* M *is a non-empty set of prime numbers less than* M*, i.e.,*

$$F = \{p_i \mid p_i \in \mathbb{Z} \text{ and } p_i < M\}$$

In this scenario, an integer is considered *smooth* over a factor base F if all of its prime factors belong to F. As soon as a factor base F of size k is constructed, the algorithm starts the sieving process by trying to find another set S such that for each $s \in S$, $f(s) = s^2 - n$ is smooth over F. The sieving process terminates when $v = |S| > |F|$, in which case a set $U \subseteq S$ can be derived satisfying that

$$\prod_{s_i \in U} f(s_i) = p_1^{2m_1} p_2^{2m_2} \cdots p_k^{2m_k} = \left(\prod_{i=1}^{k} p_i^{m_i} \right)^2 = x^2 \tag{1.2}$$

In explaining the existence of U, notes that for each $s_i \in S$, the corresponding $f(s_i)$ can be associated with a k-dimension vector $\{e_{1,i}, e_{2,i}, \ldots, e_{k,i}\}$ due to the structure of its value:

$$f(s) = p_1^{e_{1,i}} p_2^{e_{2,i}} \cdots p_k^{e_{k,i}}$$

Likewise, the above subset U can be found by solving the following system of equations:

$$\begin{cases} t_1 e_{1,1} + t_2 e_{1,2} + \cdots + t_v e_{1,v} & = 0 \pmod 2 \\ t_1 e_{2,1} + t_2 e_{2,2} + \cdots + t_v e_{2,v} & = 0 \pmod 2 \\ & \vdots \\ t_1 e_{k,1} + t_2 e_{k,2} + \cdots + t_v e_{k,v} & = 0 \pmod 2 \end{cases} \tag{1.3}$$

Since $v = |S| > |F| = k$, it is guaranteed that (1.3) can be solved to get a non-trivial solution of the form $\{t_1, t_2, \ldots, t_v\}$. Moreover, the value of each t_i can either be 1 or 0 which in turn indicates whether s_i should or should not belong to U, respectively.

Meanwhile, the original assumption on $f(s)$ also implies that $s^2 \equiv f(s) \pmod n$ which leads to the value of y as

$$y^2 = \prod_{s_i \in U} s_i^2 \tag{1.4}$$

The combination of (1.2) and (1.4) results in the congruence of squares:

$$x^2 = \prod_{s_i \in U} f(s_i) \equiv \prod_{s_i \in U} s_i^2 = y^2 \pmod n$$

Considering a practical implementation, in order to solve (1.3) one would employ the *Gaussian elimination* method if k and v are small. Otherwise, the *block Lanczos's algorithm* can be used as a better technique in terms of performance, and it indeed does not take the majority of the total running time. On the contrary, the sieving step in the Quadratic Sieve seems to take most of the time as it is directly affected by the size of n. As such, a trivial sieving technique is to search for s within a large interval $[-B, B]$ and divides $f(s)$ by each element in F. If the value of $f(s)$ finally arrives 1, the original $f(s)$ can be added to the set of smooth values. Despite its parallel feature in computing, this method still requires a lot of computations when B is large.

On the other hand, instead of trying to divide $f(s)$ by a prime p for every s in $[-B, B]$, one could alternatively fix each p in F and find an integer $s \in [-B, B]$ such that $f(s) \equiv 0 \pmod p$. Due to the assumed structure of $f(s)$, the problem turns into finding the square root of n modulo p since $s^2 - n \equiv 0 \pmod p$, or equivalently $s^2 \equiv n \pmod p$. Once an instance of s has been found, all other square roots of n modulo p can only be of the form

$s + kp$ for some integer k, as shown below:

$$f(s+kq) = (s+kp)^2 - n = s^2 + 2skp + k^2p^2 - n = f(s) + p(2sk + k^2p) \equiv 0 \pmod{p} \tag{1.5}$$

This method thus allow the algorithm to eliminate most failed trial divisions. Of crucial importance concerning this technique, n must be the quadratic residue modulo each element of F. Indeed, each p of the rational factor base can be chosen such that $n^{(p-1)/2} \equiv 1 \pmod{p}$, according to Euler's quadratic residue theorem [32, Chapter 6, p. 189]. In addition, as for the problem of finding a square root of n that generates others, it can be computed using the Shanks Tonelli algorithm. Eventually, the combination of these techniques has led to a significant success of the Quadratic Sieve in the past two decades, with the best being the factorization of an RSA challenge number of length 129 digits [10].

To be honest, much of the Quadratic Sieve is derived from its predecessor called *Dixon's algorithm*. The only difference in principle between these two generations are the forms of polynomials in use. Particularly, Dixon's algorithm uses $f(x) = x^2 \pmod{n}$, which makes it feasible to produce the congruence of squares as inherited by the method described in the Quadratic Sieve [4, p. 4]. Due to this designation, even though an instance of s may be determined such that $f(s)$ is divisible by some prime p in the factor base F, there is no such way in Dixon's algorithm to infer other values without having to perform trial divisions. This is probably the main reason for the lack of success associated with Dixon's algorithm. As shown above, the major improvement in the Quadratic Sieve appears at the sieving step with a remarkable time reduction.

However, concerning the best record mentioned, its broken modulus was not of large enough value to threaten the security of the RSA, since it is much less than the square root of what today's cryptosystem would use as the modulus of choice. Thus, a quest for a new method, particularly on the sieving technique is necessary before declaring the end of the RSA encryption. One of the primitive thought was to consider reducing the size of $f(s)$ to increase the probability that it would be smooth over the factor base F. Such observation later on led to the creation of the *Number Field Sieve* which indeed has significantly exploited this idea.

Chapter 2

Sieving in the General Number Field Sieve

In the previous section, the Quadratic Sieve was introduced as a standout success in the field of integer factorization. Being developed from Dixon's algorithm, it has overcome the performance issue in sieving techniques by using a different form of polynomial which however has the same effect in producing the congruence of squares.

Apart from its achievements, the Quadratic Sieve did not seem to make any significant adjustment in the size of $f(x)$. In the case of Dixon's algorithm, it is clear that the value of $f(x)$ is concealed from 0 to n, with a suggestion that $x > \lceil \sqrt{n} \rceil$. Similarly, this interval in the Quadratic Sieve is of the same absolute size except that it starts from $-n$ to 0. For n of size 50 decimal digits or larger, one has to find $f(s)$ of randomly close length which are smooth over a factor base whose elements are comparatively trivial. Such smoothness in practice appears with a relatively low probability. One way to increase this likelihood is to enlarge the factor base, but then more smooth values are needed for the cardinality to exceed $|F|$, which again yields a contradiction in terms of performance.

As there is almost no other improvement in sieving over the ring of integers $\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$, it is worth to try forming a square using a sieving technique over other algebraic structures and map it back to the corresponding integer square. In developing this idea, it may be necessary to devise other means of representing polynomial values and "prime numbers" as well as the division operation and a sieving method over a special algebraic structure. In the meantime, another trend would be to consider using polynomials of degree other than 2. Embracing these suggestions, the GNFS becomes isolated from

the simplicity introduced by any other factoring methods previously created, with the use of rings involving complex numbers.

2.1 Constructing the congruence of squares

Recall that in the Quadratic Sieve, one square is produced using the product of $f(x) = x^2 - n$ for some integers $x \in U$, whereas the other square is computed as the square of the product of all elements in U. This makes the algorithm itself fairly simple and straightforward in implementation since the mathematical context is restricted within the ring of integers $\mathbb{Z}$. As for the GNFS, the integer s used to form y^2 is replaced by a structure $a + bm$ for some fixed integer m that increases the flexibility for the sieving step with two variables a and b.

In general, the GNFS performs the difference in squares by using the following ring[26, p. 238], provided that the modulus n to be factored is identified:

Definition 2.1.1. *Let $f(x) = x^d + c_1 x^{d-1} + \cdots + c_{d-1}x + c_d$ be a monic, irreducible* polynomial with integer coefficients such that there exists $m \in \mathbb{Z}/n\mathbb{Z}$ for which n divides $f(m)$, i.e., $f(m) \equiv 0 \pmod{n}$. Considering an arbitrarily complex root θ of $f(x)$, the following polynomial ring $\mathbb{Z}[\theta]$ is defined as*

$$\mathbb{Z}[\theta] = \left\{ k_{d-1}\theta^{d-1} + k_{d-2}\theta^{d-2} + \cdots + k_1\theta + k_0 \mid \{k_i\} \subset \mathbb{Z} \right\}$$

In order to verify the availability of this definition, the "*Fundamental theorem of algebra*"[26, p. 473] states that for each polynomial $f(x)$ of degree d, there exist exactly d complex roots θ_i such that

$$f(x) = (x - \theta_1)(x - \theta_2)\cdots(x - \theta_d) = \prod_{i=1}^{d}(x - \theta_i) \tag{2.1}$$

As the underlying principle is to perform a sieving step over $\mathbb{Z}[\theta]$ and hence get a perfect square $\beta^2 \in \mathbb{Z}[\theta]$, it is necessary to define a mechanism that converts this square from $\mathbb{Z}[\theta]$ back to its corresponding integer square. This mapping can indeed be explicitly defined as

* $f(x) \in \mathbb{Z}[\theta]$ is monic if its coefficient of highest order is 1. $f(x)$ is irreducible if $f(x) = g(x)h(x)$ implies either $g(x) = c$ or $h(x) = c$ for some $c \in \mathbb{Z}$ and $g(x), h(x) \in \mathbb{Z}[\theta]$.

Proposition 2.1.1. *Let $f(x)$ be a monic, irreducible polynomial with integer coefficients and $m \in \mathbb{Z}_n$ satisfying $f(m) \equiv 0 \pmod{n}$. There exists a surjective ring homomorphism[†] $\phi : \mathbb{Z}[\theta] \to \mathbb{Z}/n\mathbb{Z}$ that maps polynomials in θ from $\mathbb{Z}[\theta]$ to appropriate polynomials in m over $\mathbb{Z}/n\mathbb{Z}$.*

Proof. The mapping ϕ can be defined such that $\phi(1) = 1 \pmod{n}$ and $\phi(\theta) = m \pmod{n}$. Such construction implies that for every polynomial $f(m)$ over $\mathbb{Z}/n\mathbb{Z}$, there exists a polynomial $f(\theta)$ created by substituting m to θ satisfying that $\phi(f(\theta)) = f(m) \pmod{n}$, thus making the mapping surjective. Moreover, for $a = \sum_{i=0}^{d-1} a_i\theta^i$ and $b = \sum_{i=0}^{d-1} b_i\theta^i$ in $\mathbb{Z}[\theta]$ the following binary operations using conventional polynomial multiplication and addition are clearly defined:

$$\begin{aligned}
\phi(ab) &= \phi\left(\left(\sum_{i=0}^{d-1} a_i\theta^i\right)\left(\sum_{i=0}^{d-1} b_i\theta^i\right)\right) = \phi\left(\sum_{i=0}^{2d-2}\left(\sum_{j=0}^{i} a_j b_{i-j}\right)\theta^i\right) \quad ^{\ddagger} \\
&= \sum_{i=0}^{2d-2}\left(\sum_{j=0}^{i} a_j b_{i-j}\right) m^i = \left(\sum_{i=0}^{d-1} a_i m^i\right)\left(\sum_{i=0}^{d} b_i m^i\right) \pmod{n} \\
&= \phi\left(\sum_{i=0}^{d-1} a_i\theta^i\right)\phi\left(\sum_{i=0}^{d} b_i\theta^i\right) = \phi(a)\,\phi(b)
\end{aligned}$$

$$\begin{aligned}
\phi(a+b) &= \phi\left(\left(\sum_{i=0}^{d-1} a_i\theta^i\right) + \left(\sum_{i=0}^{d-1} b_i\theta^i\right)\right) = \phi\left(\sum_{i=0}^{d-1}(a_i+b_i)\,\theta^i\right) \\
&= \sum_{i=0}^{d-1}(a_i+b_i)\,m^i = \left(\sum_{i=0}^{d-1} a_i m^i\right) + \left(\sum_{i=0}^{d} b_i m^i\right) \pmod{n} \\
&= \phi\left(\sum_{i=0}^{d-1} a_i\theta^i\right) + \phi\left(\sum_{i=0}^{d} b_i\theta^i\right) = \phi(a) + \phi(b)
\end{aligned}$$

□

Assume that after the sieving step and some further refinements, a set U of pairs (a, b) similar to that in §1.4, along with an *algebraic integer*[§] $\beta \in \mathbb{Z}[\theta]$ and $y \in \mathbb{Z}$ have been found such that

$$\beta^2 = \prod_{(a,b)\in U}(a + b\theta) \quad \textit{and} \quad y^2 = \prod_{(a,b)\in U}(a + bm) \tag{2.2}$$

[†]A surjective ring homomorphism $\phi : A \to B$ is a mapping from the ring A to the ring B such that ϕ is surjective and $\phi(a \cdot b) = \phi(a)\phi(b)$ for $a, b \in A$.

[‡]See [26, p. 237] for details

[§]See the next section for the explicit definition

By letting $x = \phi(\beta) \in \mathbb{Z}/n\mathbb{Z}$, one could easily produce the congruence of squares as follows:

$$\begin{aligned} x^2 = x \cdot x &= \phi(\beta)\,\phi(\beta) = \phi(\beta \cdot \beta) = \phi\left(\beta^2\right) \\ &= \phi\left(\prod_{(a,b)\in U} (a + b\theta)\right) \\ &\equiv \prod_{(a,b)\in U} (a + bm) \equiv y^2 \pmod{n} \end{aligned} \tag{2.3}$$

After x and y have been computed, the remaining task is to check whether $\gcd(x - y, n)$ and $\gcd(x + y, n)$ give non-trivial factors other than n and 1, with the probability of 2/3, i.e., approximately 66.67%. If trivial factors occur, an iteration of the whole sieving process needs to be restarted again until the prime factors p and q are found.

2.2 A possibility for squares in $\mathbb{Q}(\theta)$

One of the most important considerations in the GNFS is the construction of the algebraic square β^2 in the polynomial ring $\mathbb{Z}[\theta]$. Note that since $\mathbb{Z}[\theta]$ uses conventional polynomial binary multiplication and addition, it is also a commutative ring, i.e., $a \cdot b = b \cdot a$ for $a, b \in \mathbb{Z}[\theta]$. Moreover, it is clear that in this case 1 and 0 can be considered as the multiplicative and additive identities, respectively. As the ring of integers $\mathbb{Z}$ is sufficiently an *integral domain*, it follows along the lines of [26, Lemma 3.24] that $ab = 0$ implies either $a = 0$ or $b = 0$ for $a, b \in \mathbb{Z}[\theta]$. Thus $\mathbb{Z}[\theta]$ is as well an integral domain. As a result, an important field containing $\mathbb{Z}[\theta]$ can be defined as follows:

Proposition 2.2.1. *Let θ be a complex root of a monic, irreducible polynomial $f(x)$ with rational coefficients. Let $\mathbb{Z}[\theta]$ be the polynomial ring of θ over $\mathbb{Z}$. The polynomial ring of θ over $\mathbb{Q}$, denoted as $\mathbb{Q}(\theta)$, is the field of fraction of $\mathbb{Z}[\theta]$.*

Proof. Since $\mathbb{Z}[\theta]$ is a subring of $\mathbb{Q}(\theta)$, it follows similarly from the above explanation that $\mathbb{Q}(\theta)$ is also an integral domain. Furthermore, since $f(x)$ is irreducible, then $\gcd(f(x), g(x)) = 1$ for every $g(x) \in \mathbb{Q}[x] \setminus \{f(x)\}$. Using the *Euclidean extended algorithm* as given in [26, Theorem 3.71], there exist $h(x), t(x) \in \mathbb{Q}[x]$ such that

$$f(x)h(x) + g(x)t(x) = 1 \tag{2.4}$$

In (2.4) one could substitute x by θ to get $g(\theta)t(\theta) = 1$ by the assumption. In other words, $t(\theta)$ is the inverse polynomial of $g(\theta)$, denoted as $g^{-1}(\theta)$, such that $g(\theta)g^{-1}(\theta) = g^{-1}(\theta)g(\theta) = 1$. This lets $\mathbb{Q}(\theta)$ to be qualified as a field.

On the other hand, without loss of generality, one could assume that $g(\theta)$ is of the form $\sum_{i=0}^{d-1} \left(\frac{a_i}{b_i}\theta^d\right)$ where d is the degree of $f(x)$ and $a_i, b_i \in \mathbb{Z}$. By converting each fractional coefficient in $g(\theta)$ to common denominator, i.e., $\prod_{i=0}^{d-1} b_i$, $g(\theta)$ can now be represented in the form

$$g(\theta) = \frac{1}{\prod_{i=0}^{d-1} b_i} \sum_{i=0}^{d-1} \left(\theta^i a_i \prod_{j \neq i, j=0}^{d-1} b_i \right)$$

In other words, $g(\theta)$ can be factorized into the product $a^{-1}b$ for $a, b \in \mathbb{Z}[\theta]$. This makes $\mathbb{Q}(\theta)$ completely a field of fraction of $\mathbb{Z}[\theta]$, as can be seen in [26, Theorem 3.21]. □

While the actual sieving step may not take place in $\mathbb{Q}(\theta)$, the existence of such field is important as it gives rise to the following concept which is crucial for the operation of the GNFS:

Definition 2.2.1. *An algebraic integer is a complex root of a monic, irreducible polynomial with integer coefficients. The set of all possible algebraic integers $\gamma \in \mathbb{Q}(\theta)$ is called the ring of algebraic integers, denoted as $\mathfrak{O}$.*

Thus, if a perfect square β^2 is found such that $\beta \in \mathfrak{O}$, then $\beta^2 \cdot f'(\theta)^2$ also forms a square in $\mathbb{Z}[\theta]$ since $\beta \cdot f'(\theta) \in \mathbb{Z}[\theta]$. Likewise, with $x = \phi(\beta \cdot f'(\theta))$, the manipulation in (2.3) can now be altered to

$$\begin{aligned} x^2 = x \cdot x &= \phi\left(\beta \cdot f'(\theta)\right) \phi\left(\beta \cdot f'(\theta)\right) = \phi\left(\beta^2 \cdot f'(\theta)^2\right) \\ &= \phi\left(f'(\theta)^2 \prod_{(a,b)\in U} (a + b\theta) \right) \\ &\equiv f'(m)^2 \prod_{(a,b)\in U} (a + bm) \equiv y^2 \pmod{n} \end{aligned}$$

Simply state, this observation implies that it is possible to apply a similar idea as in §2.1, but to find smooth values and a perfect square in $\mathfrak{O}$ rather than in $\mathbb{Z}[\theta]$. At the same time, it is also clear that there is a higher probability for an algebraic integer α to be smooth over a fixed factor base A if it belongs to $\mathfrak{O}$ rather than $\mathbb{Z}[\theta]$, since its coefficients can be of rational form rather than integers. However in practice, this improvement of replacing

$\mathbb{Z}[\theta]$ by $\mathfrak{O}$ is not well established as it adds more overheads with the use of rational arithmetic. Unfortunately, as §2.5 and §3.1 show, the main sieving technique of the GNFS is only correct when it is applied in $\mathfrak{O}$, whereas in $\mathbb{Z}[\theta]$ the theory is not always true. As shown in §3.2, this problem later on causes a necessity for a compensation to make sure that the sieve can still be implemented in $\mathbb{Z}[\theta]$ without failure.

2.3 Sieving for rational smooth values

In preparing for the sieving process, it is necessary to create the rational factor base F which contains prime numbers of bound M as in §1.4, except that in this case there is no restriction in choosing those prime candidates. After F has been constructed, one could start the sieving process by trying to divide by each $p \in F$ all values of the form $(a + bm)$ for $a, b \in \mathbb{Z}$ and m already chosen with the monic, irreducible polynomial $f(x)$ as in Definition 2.1.1. In the end, only those pairs (a, b) for which the remaining quotients arrived at ± 1 are selected as they are completely smooth over F.

Considering the selection of a and b, it is crucial that $\gcd(a, b) = 1$. Otherwise, the sum $a + bm$ must be of the form $\gcd(a, b)(a_0 + b_0 m)$. This phenomenon also appears in the same way with the corresponding algebraic integer $a + b\theta$, i.e., $a + b\theta = \gcd(a, b)(a_0 + b_0\theta)$. Furthermore, if (a, b) produces a completely smooth value, then so does (a_0, b_0), and in case they are chosen as candidates for the set U that forms the perfect squares, their product would create a smaller congruence of squares, e.g., $\phi((a_0 + b_0\theta)^2) \equiv (a_0 + b_0 m)^2 \pmod{n}$ which is redundant since it does not improve the probability that the containing congruence of squares would produce a non-trivial factor of n. Thus, it is necessary to prevent this redundancy from scratch by requiring all smooth candidates of the form (a, b) to satisfy that $\gcd(a, b) = 1$.

Moreover, in order to support parallel computing, it is recommended to fix b to some small integer due to the fact that bm is of significant value when the modulus n is large. After that, a bound R can be chosen so that $[-R, R]$ gives an acceptable interval for a. Then, for each a in $[-R, R]$, a value of $a + bm$ is checked if it is smooth by trial divisions to every prime $p \in F$. In practice, instead of checking the smoothness for each $a + bm$ all at once, a better method is to perform trial divisions for all $a + bm$ by each prime p before moving to the other prime in F. Note that using the latter mechanism, an improvement for the sieve can be devised similar to that in

(1.5). Indeed, for each $p \in F$ it is only required to find a value a_0 in $[-R, R]$ such that $a_0 + bm \equiv 0 \pmod{p}$. All other values of a in $[-R, R]$ such that $p|a + bm$ are then concealed to the form $a_0 + kp$ for $k \in \mathbb{Z}$. Thus, regardless of the size of R, only one trial division is needed for each prime in F, making the total number of trials is $|F|$.

On the other hand, in order to produce a list of ± 1 in the end of the sieving process, each $t = a + bm$ divisible by some prime p must be updated with t/p. In the sieving step, these updates create a large amount of divisions, which remarkably decrease the overall performance. A suggestion is to somehow switch the operation from division to subtraction since the latter is much faster in practical computing. This leads to the idea of using the *natural logarithm* as a representative for both $a + bm$ and p. A division of $a + bm$ by p would simply mean a subtraction of $ln(a + bm)$ by $ln(p)$. Thus, given a fixed $b \in \mathbb{Z}^+$, a prime $p \in F$, and the smallest a_0 in $[-R, R]$ such that $a_0 + bm \equiv 0 \pmod{p}$, it is relatively straightforward to form the sieving process, as given in Algorithm 2.3.1.

Algorithm 2.3.1: RationalSieve

```
   Input: b, R, a_0
 1 begin
 2     for a ← −R to R do  log[a] ← ln(a + bm);
 3     count ← (a_0 + bm)/p;
 4     k ← a_0;
 5     while k < R do
 6         log[k] ← log[k] − ln(p);
 7         e ← 0;
 8         while p^{e+1}|count do  e ← e + 1;
 9         log[k] ← log[k] − e · ln(p);
10         k ← k + p;
11         count ← count + 1;
12     endw
13     result ← {};
14     for a ← −R to R do
15         if log[a] < 0.69 then  result ← result ∪ (a, b);
16     endfor
17     ▷ Return result;
18 end
```

This algorithm in practice also covers situations in which $a+bm$ contains p^k as one of its factors. In such cases, the variable e in Algorithm 2.3.1 is computed as the exponent k of p in the factorization of $a + bm$. Thus, $ln(a + bm)$ can safely be subtracted with $e \cdot ln(p)$ as shown above. In fact, this method is in contrast with the one explained in [4, p. 25] that spontaneously ignores the exponent k of p using some *fudged factors*. Moreover, an improvement in [14, Section 4.2, pp. 26-27] suggests that the subtraction could be further replaced by the addition, for which the *log* list is initialized with 0 and added by $ln(p)$ every time $p|(a + bm)$.

Finally, each element in the *log* list in correspondence with $a + bm$ is considered smooth and selected if its value is 0 (or $ln(a + bm)$ when the additive variation is used), since $ln(1) = 0$. In programming practice, as there is no way to get the exact value of the natural logarithm, it is necessary to allow errors in the final selection, that is, if $log[a_i] < 0.7 = ln(2)$, the corresponding a_i+bm is smooth even though $log[a_i]$ is not exactly 0 due to the approximity of computations. Note also that it is not necessary to compute $ln(a+bm)$ many times since the value of a is almost trivial compared to bm and hence $ln(a + bm) \approx ln(bm)$. Thus, one would only need to generate the *log* array with $ln(bm)$ and still be able to grasp all the smooth values.

2.4 Sieving for algebraic smooth values

As the main difference to other types of sieving methods, the GNFS besides a traditional sieve over $\mathbb{Z}/n\mathbb{Z}$ also operates a parallel search for smooth values over an algebraic structure, that is, the ring $\mathbb{Z}[\theta]$ of polynomials in θ with integer coefficients. As a result, this is the most difficult part both to construct the theoretical background and to implement it, since there exists no explicit way of representing this ring in a computer system. A solution for this problem is to find a mapping that converts the theoretical ideas from smooth values, primes, and operators to their unique representatives in $\mathbb{Z}/n\mathbb{Z}$ so as to enable a fruitful deployment using computer programming.

As the first step, according to (2.1), the following result defines several monomorphisms which map the field of fraction $\mathbb{Q}(\theta)$ to the set of complex number $\mathbb{C}$:

Proposition 2.4.1. *Given a monic, irreducible polynomial $f(x)$ of degree d with integer coefficients and a root $\theta \in \mathbb{C}$, there exist exactly d ring monomorphisms, namely $\sigma_i : \mathbb{Q}(\theta) \to \mathbb{C}$ such that $\sigma_i(1) = 1$ and $\sigma_i(\theta) = \theta_i$ for some*

root $\theta_i \in \mathbb{C}$ of $f(x)$. Moreover, each $\alpha \in \mathbb{Q}(\theta)$ has d conjugates including itself, defined in turn as $\sigma_i(\alpha)$.

While this proposition can be verified in [30, Theorem 1.8, p.23], such conjugates give rise for the definition of an important mapping that partially satisfies our needs:

Proposition 2.4.2. *Let $f(x)$ be a monic, irreducible polynomial of degree d with integer coefficients and $\mathbb{Q}(\theta)$ be the corresponding field of polynomial with rational coefficients in θ, a complex root of $f(x)$. For each algebraic integer $\alpha \in \mathbb{Q}(\theta)$, the product of its conjugates, namely the Norm of α, denoted as $N(\alpha)$, is an integer in $\mathbb{Z}$.*

Proof. The proof may begin with a proof that the following *field polynomial* of α has integer coefficients:

$$f_\alpha(x) = \prod_{i=1}^{d}(x - \sigma_i(\alpha)) \tag{2.5}$$

Indeed, it can easily be seen from [1, Theorem 4.3.4, pp. 153-154] that there exists a monic, irreducible polynomial $g(x) \in \mathbb{Q}[x]$ of least degree k such that $g(\alpha) = 0$. In this case $g(x)$ is called the *minimal polynomial* of α. Moreover, since α is an algebraic integer, it follows that there also exists a polynomial $t(x) \in \mathbb{Z}[x] \subset \mathbb{Q}[x]$ with integer coefficients such that $t(\alpha) = 0$.

Using the division algorithm $t(x)$ can be analyzed as $g(x) \cdot h(x) + r(x)$ where $deg(r) < deg(g)$. Thus, the assumption $g(\alpha) = t(\alpha) = 0$ implies that $r(\alpha) = 0$, and hence $r(x) = 0$, otherwise it would contradict with the finding that $g(x)$ is the *minimal polynomial*. This means that $g(x)|t(x)$, and since $t(x) \in \mathbb{Z}[x]$, a result in [30, Lemma 1.4, pp. 19-20] shows that even if $g(x) \in \mathbb{Q}[x]$, it can be turned into a minimal polynomial $G(x)$ in $\mathbb{Z}[x]$ by multiplying with a scalar $\lambda \in \mathbb{Q}$.

Meanwhile, it is important to note that $f_\alpha(\alpha) = 0$ since there exists $1 \leq i \leq d$ such that $\sigma_i(\theta) = \theta$, which makes a factor of $f_\alpha(x)$ becomes $(x-\alpha)$. Thus, the result from [30, Theorem 2.5(a), p. 43] shows that $f_\alpha(x) \in \mathbb{Z}[x]$ as it is a power of $G(x) \in \mathbb{Z}[x]$ described above.

Moving a step further, based on the definition of $N(\alpha) = \prod_{i=1}^{d} \sigma_i(\alpha)$, the formula of $f_\alpha(x)$ in (2.5) can be expanded to

$$f_\alpha(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_1x + a_0$$

where $a_0 = (-1)^d N(\alpha) - k_0$ for some $k_0 \in \mathbb{Z}$ as the coefficient of the lowest order of $f(x)$. Thus, it turns out that $N(\alpha) \in \mathbb{Z}$. □

In the case of $a + b\theta \in \mathbb{Q}(\theta)$, the norm from Proposition 2.4.2 can be further justified [4, p. 19] as

$$\begin{aligned} N(a+b\theta) &= \prod_{i=1}^{d} \sigma_i(a+b\theta) = \prod_{i=1}^{d}(a+b\theta_i) \\ &= \prod_{i=1}^{d}(-b)\left(-\frac{a}{b}-\theta_i\right) = (-b)^d \prod_{i=1}^{d}\left(-\frac{a}{b}-\theta_i\right) \\ &= (-b)^d f\left(-\frac{a}{b}\right) \end{aligned} \tag{2.6}$$

which can be used to check whether $a+b\theta$ is smooth over an algebraic factor base A. In fact, the value of the norm function can be used as a representative for each algebraic integer $a+b\theta$, and furthermore its value can be sieved over a set of prime integers to resemble the process of sieving $a+b\theta$ over some algebraic primes $p \in \mathbb{Z}[\theta]$. While the underlying principles of this analogy are explained and proven in the next section, the consideration on how to practically implement the sieve is what concentrated the most in this section.

That being said, each $(a+b\theta)$ can be partially checked for its smoothness by dividing $N(a+b\theta)$ to some primes p in another rational factor base A_1 which is the representative of the algebraic factor base A. As a result, the simplest method is to first compute the value of $N(a+b\theta)$ and perform the trial divisions by each $p \in A_1$. In the initial stage, this would be painful concerning the complexity since $N(a+b\theta)$ is of polynomial form, and the number of such trial divisions could be expanded to billions if the modulus n is large enough.

Therefore, it is necessary to devise a method that determines how to sieve $N(a+b\theta)$ over some prime p in a more reasonable manner. Fortunately, such method is relatively straightforward, as shown in the following result [14, p. 58]:

Proposition 2.4.3. *Given a monic, irreducible polynomial $f(x)$ with integer coefficients and one of its roots $\theta \in \mathbb{C}$, the norm function of $a+b\theta \in \mathbb{Z}[\theta]$ is divisible by some prime p if there exists an integer r such that $a \equiv -br \pmod{p}$ and $f(r) \equiv 0 \pmod{p}$. In general, the exponent of p as a factor of $N(a+b\theta)$ can be summarized as*

$$e_{p,r}(a+b\theta) = \begin{cases} ord_p(N(a+b\theta)) & if\ a \equiv -br \pmod{p} \\ 0 & otherwise \end{cases} \tag{2.7}$$

where $ord_p(k)$ is the number of factors p in k.

Proof. By the definition of $N(a + b\theta)$ it is obvious that $(-b)^d f(-a/b) \equiv 0 \pmod{p}$ in case $p|N(a + b\theta)$. This logically implies either $p|(-b)^d$ or $p|f(-a/b)$. Suppose the former holds, i.e., $p|b$, and that $N(a + b\theta)$ is expressed as follows:

$$\begin{aligned} N(a + b\theta) &= (-b)^d f\left(\frac{-a}{b}\right) \\ &= (-b)^d \left(\left(\frac{-a}{b}\right)^d + k_{d-1}\left(\frac{-a}{b}\right)^{d-1} + \cdots + k_1\left(\frac{-a}{b}\right) + k_0\right) \\ &= a^d - k_{d-1}ba^{d-1} + k_{d-2}b^2a^{d-2} + \cdots + k_1(-b)^{d-1}a + k_0(-b)^d \end{aligned}$$

This shows that besides other terms of $N(a + b\theta)$ which contain b as factors and hence are divisible by p, the first term a^d must also be divisible by p, which means $p|a$ and $\gcd(a, b) \geq p > 1$. However, note that for a reason described in §2.3 only those pairs (a, b) for which $\gcd(a, b) = 1$ are used. This contradiction implies $p \nmid b$ and $f(-a/b) \equiv 0 \pmod{p}$. Thus, if $p|N(a + b\theta)$, there must exists an integer r such that $f(-a/b) \equiv f(r) \equiv 0 \pmod{p}$, i.e., $-a/b \equiv r \pmod{p}$ or conveniently $a \equiv -br \pmod{p}$. This fact is true as long as $f(x) \equiv f(y) \pmod{p}$ implies $x \equiv y \pmod{p}$. □

According to this result, since the value of r is independent of a and b, it is necessary that for each prime p in the factor base A_1, a set R_p containing r such that $f(r) \equiv 0 \pmod{p}$ should be pre-generated from the very beginning of the sieving step so that (a, b) could be test for the divisibility of $N(a + b\theta)$ by p without having to compute $f(-a/b)$. In fact, as will be shown in the next section, each pair (p, r) computed for this convenience is actually a unique representative of a special prime in $\mathbb{Z}[\theta]$. This further explains why the above divisions and sieving idea in $\mathbb{Z}$ are valid for checking the smoothness in $\mathbb{Z}[\theta]$.

Meanwhile, Proposition 2.4.3 also allows the sieving of algebraic integers to be devised in a similar manner as that for rational smooth values. Indeed, given a fix value of b and a prime (p, r), it is sufficient to find a_0 such that $a_0 \equiv -br \pmod{p}$. By using a_0, all other values of a such that $p|N(a + b\theta)$ will come under the form $a_0 + kp$ for some $k \in \mathbb{Z}$.

Likewise, as in the rational sieve, a problem arises concerning the situation in which $N(a + b\theta)$ is divisible by the power p^k for some $k \in \mathbb{Z}^+$. Unfortunately, the above idea does not give a solution to identify how many

times $N(a + b\theta)$ should be divided by p rather than performing trial divisions. Due to the wide range of a, this problem not only forces $N(a + b\theta)$ to be computed but also produces a significant amount of waste divisions. Otherwise, If these exponents are eventually ignored, many smooth values yet remarkable would be overlooked. However, as the following result shows, such indications can be achieved without much computing efforts:

Proposition 2.4.4. *Let $f(x)$ be a monic, irreducible polynomial with integer coefficients $\{k_i\}$ and a root $\theta \in \mathbb{Z}$. Given a prime p and an algebraic integer $a + b\theta \in \mathbb{Z}[\theta]$, the value of the following expression is independent of k:*

$$N\left[(a + (k+1)p^m) + b\theta\right] - N[(a + kp^m) + b\theta] \pmod{p^{m+1}} \tag{2.8}$$

for some arbitrary $k \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$.

Proof. In order to simplify the idea behind, the two norm functions can be expanded to the following sums:

$$\begin{aligned} N_1 &= N\left[(a + (k+1)p^m) + b\theta\right] \\ &= (-b)^d f\left(\frac{a + (k+1)p^m}{-b}\right) \\ &= (-b)^d \sum_{i=0}^{d} k_i \left(\frac{a + (k+1)p^m}{-b}\right)^i \\ &= \sum_{i=0}^{d} k_i (-b)^{d-i} (a + (k+1)p^m)^i \end{aligned} \tag{2.9}$$

$$\begin{aligned} N_2 &= N\left[(a + kp^m) + b\theta\right] \\ &= (-b)^d f\left(\frac{a + kp^m}{-b}\right) \\ &= (-b)^d \sum_{i=0}^{d} k_i \left(\frac{a + kp^m}{-b}\right)^i \\ &= \sum_{i=0}^{d} k_i (-b)^{d-i} (a + kp^m)^i \end{aligned} \tag{2.10}$$

Note that each term of the sums in (2.9) and (2.10) differ by at most p^m. Therefore, their differences modulo p^{m+1} would be simplified as follows:

$$
\begin{aligned}
N_1 - N_2 &= \sum_{i=0}^{d} k_i(-b)^{d-i}(a+(k+1)p^m)^i - \sum_{i=0}^{d} k_i(-b)^{d-i}(a+kp^m)^i \pmod{p^{m+1}} \\
&= \sum_{i=1}^{d} \left(k_i(-b)^{d-i} p^m \sum_{j=0}^{i-1} (a+(k+1)p^m)^j (a+kp^m)^{d-j-1} \right) \pmod{p^{m+1}} \\
&= p^m \sum_{i=1}^{d} \left(k_i(-b)^{d-i} \sum_{j=0}^{i-1} a^j a^{d-j-1} \right) \pmod{p^{m+1}}\text{¶} \\
&= p^m \sum_{i=1}^{d} \left(k_i(-b)^{d-1} i a^{i-1} \right) \pmod{p^{m+1}}
\end{aligned}
$$

□

Simply state, Proposition 2.4.4 indicates that if the remainder δ_0 of $N(\alpha)$ and δ_1 of $N(\alpha + p^m)$ are known modulo p^{m+1}, then it is easy to find the remainder of $N(\alpha + kp^m) \pmod{p^{m+1}}$ as $\delta_1 + k(\delta_2 - \delta_1)$ without having to compute $N(\alpha + kp^m)$ and try to divide it by p^{m+1}. In this case, such common distance $\delta_2 - \delta_1$ between the two remainders can be called the *remainder unit* with regard to the exponent $m+1$ of the modulus.

As the remaining obstacle for this sieving technique, it is important that initial values for entries in the *log* array need to be generated by the same meaning as in Algorithm 2.3.1. Recall that in the previous section, this array could be initialized with a single value of $ln(bm)$ since a plays a subtle role in the computation of the natural logarithm. Unfortunately, this does not hold for the norm of $a + b\theta$ which varies significantly for each a. A solution in this case is to approximately solve the equation $N'(a + b\theta) = 0$ to locate the *critical points* on the graph of $N(a + b\theta)$. By computing the natural logarithm at each of such point as well as the beginning and ending of the interval, one could approximate the value of $ln(N(a + b\theta))$ to a certain extent. Even though the computing technique behind this idea is out of concern, it should make no difficulties to solve such equations, especially when using software such as *Mathematica*.

Given a fixed value of b, a prime pair (p, r) and the sieving interval $[A, B]$, after the *log* array has been initialized, it is ready to begin the sieve by p, as

¶Since the sum is multiplied by p^m, all of its inner terms whose values contain p are canceled.

illustrated in Algorithm 2.4.1.

Algorithm 2.4.1: AlgebraicSieve

Input: b, p, r, A, B, initialized logarithm list $log[\,]$
Output: List of pairs (a, b) smooth over prime (p, r)

```
1  begin
2      a ← p((−br + A) ÷ p);
3      ▷ Initialize unit[ ] to 0;
4      ▷ Initialize Remainder[ ] to 0;
5      while a < B do
6          e ← 0;
7          while Remainder[e] = 0 do
8              e ← e + 1;
9              if unit[e] = 0 then
10                 unit[e] ← N(a + p^e + bθ) − N(a + bθ) (mod p^{e+1});
11                 Remainder[e] ← N(a + bθ) (mod p^{e+1});
12             else
13                 Remainder[e] ← Remainder[e] + unit[e] (mod p^{e+1});
14             endif
15         endw
16         log[a] ← log[a] − e · ln(p);
17         a ← a + p;
18     endw
19     result ← {};
20     for a ← A to B do
21         if log[a] < ln(2) then result ← result ∪ (a, b);
22     endfor
23     ▷ Return result;
24 end
```

As an important note, the array *unit* in the above algorithm acts as the indicator for the *remainder distances* discussed previously. In fact, once a unit has been computed, say $unit[2]$ for example, it is no more required to compute $N(a + b\theta) \pmod{p^3}$, whereas this value can still be found by simply adding $unit[2]$ to the remainder δ from the previous iteration, i.e., $\delta = N(a - p^2 + b\theta) \pmod{p^3}$. Therefore, the array *Remainder* is created to hold these previous remainders modulo each p^e. Thus, assume that the whole sieving process employs b within an interval $[B_1, B_2]$, the total number of *norm* computations would be effectively concealed within $(B_2 - B_1)\#A_1$,

where A_1 is the set of pairs (p, r) that represent algebraic primes in $\mathbb{Z}[\theta]$.

2.5 Justifying the algebraic factor base

In the previous section, the idea of smoothness of an algebraic integer $(a + b\theta) \in \mathbb{Z}[\theta]$ is introduced by using a *norm* mapping that converts $a + b\theta$ into an integer. This integer is then checked for smoothness in a similar manner as in rational sieving. However, the idea of mapping primes from $\mathbb{Z}[\theta]$ to a prime integer has not been clarified explicitly except the use of a prime pair (p, r). Indeed, the definition of the norm function does not reveal a way on how a prime in $\mathbb{Z}[\theta]$ would look like when it is mapped into $\mathbb{Z}$, i.e., whether their norm values are really primes in $\mathbb{Z}$.

Alternatively, the actual idea in this mapping technique is to use another algebraic structure which shares the same application of the norm function, but it moreover provides a mean of identifying the relationship between primes in $\mathbb{Z}[\theta]$ and $\mathbb{Z}$. In practice, when constructing the mapping into A_1 in the previous section form the algebraic factor base $A \subset \mathbb{Z}[\theta]$, it is important to mention the following concept:

Definition 2.5.1. *Given a commutative ring R, an ideal I of R is a subset of R such that $xr, x + y \in I$ for every $x, y \in I$ and $r \in R$. If I can be generated from a single element $a \in R$, then I is called a principal ideal, denoted as $\langle a \rangle$. Moreover, if I is a proper ideal of R and $ab \in I$ implies either $a \in I$ or $b \in I$, then I is a prime ideal. Of crucial importance, the binary operations for any two ideals I and J can be defined as follows:*

$$I + J = \{a + b | a \in I \text{ and } b \in J\}$$
$$I \cdot J = \left\{ \sum_{i=0}^{k} a_i b_i | a_i \in I \text{ and } b_i \in J \text{ and } k \in \mathbb{Z} \right\}$$

For safety reason, in this case the ring $\mathfrak{O}$ will be used instead of $\mathbb{Z}[\theta]$ as the context to explain the idea of prime and factorization of algebraic integers. This is because $\mathbb{Z}[\theta]$ is not fully qualified for such operation as briefly noted in the previous section. However, as will be shown in the next chapter, a method to compensate the lack of features in $\mathbb{Z}[\theta]$ can heuristically be overcome by using *quadric characters* which help in securing the mapping of an unknownly claimed perfect square from $\mathbb{Z}[\theta]$ to an integer square in $\mathbb{Z}$.

To begin the discussion, the most important characteristic of $\mathfrak{O}$ can be concluded in the following result:

Proposition 2.5.1. *Given a monic, irreducible polynomial $f(x)$ with integer coefficients and a root $\theta \in \mathbb{C}$, every ideal of the ring $\mathfrak{O}$ of algebraic integers in $\mathbb{Q}(\theta)$ are principal. In addition, each non-zero ideal $\langle\alpha\rangle$ in $\mathfrak{O}$ can uniquely factors into a product of prime ideals $\mathfrak{p}_i \subset \mathfrak{O}$, that is,*

$$\langle\alpha\rangle = \prod_{i=0}^{k} \mathfrak{p}_i^{e_i} \tag{2.11}$$

for some $k, e_i \in \mathbb{Z}^+$.

Due to the above properties, the ring $\mathfrak{O}$ is called both as a *principal ideal domain* (PID) and a *unique factorization domain* (UFD). In fact, a domain once proved to be a PID automatically implies itself to be a UFD [26, Theorem 7.16, p. 528]. Following this condition, a similar concept of *norm* can be defined for ideals of $\mathfrak{O}$, whether or not they are prime:

Proposition 2.5.2. *Let the norm of an ideal I of a ring R, denoted as $N(I)$, be the size of the corresponding quotient ring R/I. If I is a prime ideal, then there exists a prime p and a positive integer k such that $N(I) = p^k$. Otherwise, if R is a UFD, then for each $\langle\alpha\rangle \subset R$ such that*

$$\langle\alpha\rangle = \prod_{i=0}^{k} \mathfrak{p}_i^{e_i}$$

for some unique set $\{\mathfrak{p}_i\}$ and $\{e_i\}$, the norm of $\langle\alpha\rangle$, i.e., $N(\langle\alpha\rangle)$ can be factorized as

$$|N(\alpha)| = N(\langle\alpha\rangle) = \prod_{i=0}^{k} N(\mathfrak{p}_i)^{e_i} = \prod_{i=0}^{k} p_i^{e_i k_i} \tag{2.12}$$

for p_i, k_i such that $N(\mathfrak{p}_i) = p_i^{k_i}$.

In order to simplify the formula in (2.12), it is sufficient to consider only those prime ideals $\mathfrak{p}$ whose norms are prime instead of powers of prime. These ideals are hence given a special name, called as *first degree prime ideals*. Indeed, the use of *first degree prime ideals* has a strong relationship with the set of pairs (p, r) introduced in the previous section for sieving algebraic integers. Furthermore, the sufficiency of using this type of ideals can be shown by the following result:

Proposition 2.5.3. *Let $f(x)$ be a monic, irreducible polynomial with integer coefficients and a root $\theta \in \mathbb{C}$. The set of all first degree prime ideals in $\mathbb{Z}[\theta]$*

is in bijective correspondence[‖] *with the set of all pairs* (p, r) *for* $p, r \in \mathbb{Z}$ *such that* $f(r) \equiv 0 \pmod{p}$.

Thus, if an algebraic integer $(a + b\theta) \in \mathfrak{O}$ satisfies that $a \equiv -br \pmod{p}$ for some pair (p, r) described above, it means $p|N(a + b\theta)$ and hence there exists a first degree prime ideal $\mathfrak{p}$ with $N(\mathfrak{p}) = p$ such that $\mathfrak{p}|\langle a + b\theta\rangle$. Analogously, the process of sieving an algebraic integer in $\mathfrak{O}$ can be turned into sieving an ideal in the same ring.

However, the prime ideals in Proposition 2.5.1 for factoring are that of the ring $\mathfrak{O}$, whereas those used in the previous section for the sieving step lie within the ring $\mathbb{Z}[\theta]$. This inconsistence creates some obstructions since a prime ideal in $\mathbb{Z}[\theta]$ is not necessarily the one in $\mathfrak{O}$ and vice versa. Moving a step further, an approach to address this problem is to check whether the result in Proposition 2.5.1 would hold if the prime ideals in $\mathbb{Z}[\theta]$ were used. Fortunately, as it is the case, the following result confirms such feasibility [14, Proposition 7.1, pp. 63-64]:

Proposition 2.5.4. *Let* K *be an algebraic number field, with* K^* *being its multiplicative group, i.e.,* $K \setminus \{0\}$. *Let* A *be an order of* K. *For each prime ideal* $\mathfrak{p} \subset A$, *there exists a group homomorphism* $l_{\mathfrak{p}} : K^* \to \mathbb{Z}$ *such that*

(*a*) *if* $x \in A$, *then* $l_{\mathfrak{p}}(x) \geq 0$
(*b*) *if* $x \in A$, *then* $l_{\mathfrak{p}}x > 0$ *iff* $x \in \mathfrak{p}$
(*c*) *if* $x \in K^*, \exists P = \{\mathfrak{p}_i\}$ *of finite order such that* $l_{\mathfrak{p}_i}(x) > 0$ *and*
$$\prod_{\mathfrak{p} \in P} (N(\mathfrak{p}))^{l_{\mathfrak{p}}(x)} = |N(x)|$$

Applying this proposition to the current situation, the number field K is assigned to the field $\mathbb{Q}(\theta)$ of polynomials in θ with rational coefficients. Consequently, the corresponding order A of $\mathbb{Q}(\theta)$ can be mapped to the ring $\mathbb{Z}[\theta]$. Thus, considering the claim (c) of this result, it is exactly what was mentioned in (2.12). This raises a hope for the feasibility of the sieving technique described in the previous section.

[‖]Set A is said to be in bijective correspondence with set B, if there exists a mapping $\phi : A \to B$ such that for all $y \in B$ there exist a unique $x \in A$ with $\phi(x) = y$.

Chapter 3

Computing perfect squares with matrix equations

Recall from the previous section that some inconsistences have been encountered when dealing with the sieving of the algebraic integers in $\mathbb{Z}[\theta]$. As described in §2.4, the practical sieving step is built upon the ring $\mathbb{Z}[\theta]$ of polynomials in θ with integer coefficients. The main benefit to set up the algorithm on this ring is in its ability which allows a direct and simple mapping from a perfect square $\beta^2 \in \mathbb{Z}[\theta]$ to a perfect square x^2 with $x \in \mathbb{Z}$ as shown in Proposition 2.1.1.

On the other hand, as §2.5 illustrates, the actual mathematical context which satisfies described in §2.4 is not $\mathbb{Z}[\theta]$, but its containing ring $\mathfrak{O}$. This inconsistence raises a question on whether the algorithm for sieving algebraic integers previously explained is any longer valid. Following Proposition 2.5.4, a compensation mapping is created to make sure that the sieving algorithm works at least in its principle. Unfortunately, this mapping does not guarantee that the outcome of the sieve would potentially produce a perfect square, firstly in $\mathfrak{O}$, then in $\mathbb{Z}[\theta]$, and finally in $\mathbb{Z}$.

In filling the above gaps, a number of techniques has been created independently by the interested community. As an example, one could explicitly move from the ring $\mathbb{Z}[\theta]$ to $\mathfrak{O}$ by using prime ideals in $\mathfrak{O}$ [14, p. 61]. On the contrary, this chapter describes a less complicated method that fulfills the requirements and hence increases the probability that the smooth values produced by the previous sieving technique would be sufficient to produce the perfect square. In addition, it also discusses in details how the final set U is chosen from the large set of smooth values that is finally used to construct the algebraic square.

3.1 The shortcomings of $\mathbb{Z}[\theta]$

Note that in the principle described in §1.4 one attempts to find among the set S of smooth values a subset U such that the product of its elements contains only factors of even prime powers. In the rational sieve, this is enough to ensure that the above product is a perfect square. However, as briefly mentioned in §2.5, the algebraic sieve does not directly check for smoothness of algebraic integers due to limitation of computing techniques. Instead, an indirect method using the *norm* mapping is invoked to check for smoothness. This means that even if norm values of certain algebraic integers in $U_A \subset A$ are smooth, and that their norm product is a square, it is still possible that the actual product of algebraic integers in U_A is not a perfect square. This at the final stage may lead to a non perfect square in $\mathbb{Z}$ when the homomorphism ϕ is used.

On the other hand, it is of crucial importance that even though the squareness of a norm product alone does not fully imply a perfect square in $\mathbb{Z}[\theta]$, the inverse is always true, which makes this partial condition indispensable in selecting smooth values. Before diving into this assertion, it is necessary to justify the value of the homomorphism $l_{\mathfrak{p}}$ introduced in the previous section:

Proposition 3.1.1. *Let a and b be two coprime integers. Given a first degree prime $\mathfrak{p} \subset \mathbb{Z}[\theta]$ that maps to a pair (p, r) using an isomorphism in Proposition 2.5.3, then $l_{\mathfrak{p}}(a + b\theta) = e_{p,r}(a + b\theta)$.*

Proof. Since $\mathfrak{p}$ corresponds to the pair (p, r), it follows by Proposition 2.5.3 along the lines of [4, p. 16] that $N(\mathfrak{p}) = p$. According to Proposition 2.5.4(c) the absolute value of $N(a + b\theta)$ contains factor $p^{l_{\mathfrak{p}}(a+b\theta)}$.

Meanwhile, by assumption the notation $e_{p,r}(a+b\theta)$ stands for the number of p in the factors of $N(a + b\theta)$. Thus $l_{\mathfrak{p}}(a + b\theta) = e_{p,r}(a + b\theta)$. □

Based on this result, it is straightforward to show that the squareness of a norm product is important in determining a square algebraic integer. The following result addresses this hypothesis:

Proposition 3.1.2. *let U be a set of algebraic integers $a + b\theta \in \mathfrak{O}$ such that their product forms a square of an algebraic integer in $\mathfrak{O}$. Then the norm of such square is also a perfect square in $\mathbb{Z}$, i.e.,*

$$\sum_{a+b\theta \in U} l_{\mathfrak{p}}(a + b\theta) \equiv 0 \pmod 2$$

for all first degree prime ideals $\mathfrak{p} \subset \mathbb{Z}[\theta]$.

Proof. Let $\beta^2 \in \mathfrak{O}$ be the square formed by the algebraic integers of concern. Recall from Proposition 2.4.1 that since σ_i are monomorphisms, $\sigma_i(\beta^2) = \sigma_i(\beta)\sigma_i(\beta)$. By applying this to the computation of $N(\beta^2)$ similar to that in (2.6) it follows that:

$$N(\beta^2) = \prod_{i=1}^{d} \sigma_i(\beta^2) = \prod_{i=1}^{d} \sigma_i(\beta)^2 = \left(\prod_{i=1}^{d} \sigma_i(\beta) \right)^2 = (N(\beta))^2$$

Let $\mathfrak{p} \subset \mathbb{Z}[\theta]$ be a first degree prime ideal corresponding to a pair (p, r) such that $p^{k_p} | N(\beta)$ and hence $e_{p,r}(\beta) = k_p$ for the largest possible $k_p \in \mathbb{Z}$. It follows that $p^{2k_p} | (N(\beta))^2 = N(\beta^2)$ and hence $e_{p,r}(\beta^2) = 2k_p \equiv 0 \pmod 2$. Thus the proposition can be proven as in [14, p. 60]:

$$\begin{aligned} \sum_{a+b\theta \in U} l_{\mathfrak{p}}(a + b\theta) &\equiv \sum_{a+b\theta \in U} e_{p,r}(a + b\theta) \\ &\equiv e_{p,r} \left(\prod_{a+b\theta \in U} a + b\theta \right) \\ &\equiv e_{p,r} \left(\beta^2 \right) \equiv 2k_p \equiv 0 \pmod 2 \end{aligned}$$

□

Apart from the above indicator which determines a square of an algebraic integer in $\mathfrak{O}$, there are obstructions which prevent it from fully attesting the prediction. First of all, as explained in §2.5, the underlying technique used in the algebraic sieve actually utilizes the operations on ideals in $\mathfrak{O}$ instead of their generators, i.e., algebraic integers in $\mathfrak{O}$. Therefore, in order for the perfect square in $\mathfrak{O}$ to be trustworthy, it is necessary that its corresponding principal ideal must also be a square of an ideal in $\mathfrak{O}$, that is,

$$\prod_{a+b\theta \in U} (a + b\theta)\mathfrak{O} = \mathfrak{o}^2 = \beta^2 \mathfrak{O} = \left\langle \beta^2 \right\rangle \tag{3.1}$$

for the refined set U described above. However, since all the primes ideals used to factor each ideal generated by an element in U are ideals in $\mathbb{Z}[\theta]$ rather than $\mathfrak{O}$, the resulting ideal may not be a square of an ideal in $\mathfrak{O}$.

Furthermore, even if the square of an ideal can be generated by an algebraic integer $\beta^2 \in \mathfrak{O}$ as in (3.1), it is still not necessary the case that $\prod_{a+b\theta \in U}(a + b\theta) = \beta^2$ since the results from both sides could generate the same ideal even though their values are different. Thus, it is theoretically required that these gaps must be overcome by some means, or otherwise the probability for the perfect square to appear will plummet drastically.

3.2 The use of quadratic characters

The idea of using quadratic characters was originated from an observation about the feature of *quadratic residues* throughout the whole idea of integer factorization. Indeed, the initial thought states that if an integer a is truly a perfect square, then for each prime $p \in \mathbb{Z}$ for which $p \nmid a$, either $a \bmod p$ is itself a perfect square or a is quadratic residue modulo p, i.e., there exists $x \in \mathbb{Z}$ such that $x^2 \equiv a \pmod{p}$. In a more precise term, the following concept is used to specify whether an integer is a quadratic residue:

Definition 3.2.1. *Given $a \in \mathbb{Z}$ and a prime $p \in \mathbb{Z}$ such that $p \nmid a$, the Legendre symbol, denoted as $\left(\frac{a}{p}\right)$, can be defined as follows:*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } \exists \gamma \in \mathbb{Z}: \ \gamma^2 \equiv a \pmod{p} \\ -1 & \text{if } \nexists \gamma \in \mathbb{Z}: \ \gamma^2 \equiv a \pmod{p} \\ 0 & \text{if } p|a \end{cases}$$

Thus, assume that there is no way to check whether a is a perfect square, an alternative is to try testing whether a is quadratic residue modulo some primes $p \nmid a$. This does not explicitly guarantee that a is a perfect square, but the more number of tests are to be invoked, the more likely that a is a perfect square in $\mathbb{Z}$. However, in the case of the GNFS, it is not necessary to test the perfect square in $\mathbb{Z}$, but rather in $\mathbb{Z}[\theta]$. Therefore, a similar meaning to the above test must be built to test the set U in (3.1) over some different primes in $\mathbb{Z}[\theta]$. As such, the following result integrates the use of the Legendre symbol into justifying the squareness of an algebraic integer:

Proposition 3.2.1. *Let U be the set of $a + b\theta \in \mathfrak{O}$ such that the product of elements in U results in a perfect square $\beta^2 \in \mathfrak{O}$. Let $\mathfrak{p} \subset \mathbb{Z}[\theta]$ be a first degree prime ideal in correspondence to a pair (p, r) where p is prime such that $\beta \notin \mathfrak{p}$, i.e., $a \not\equiv -br \pmod{p}$. Then the following holds:*

$$\prod_{a+b\theta \in U} \left(\frac{a+br}{p}\right) = 1 \tag{3.2}$$

Proof. Note that the fraction in (3.2) is the Legendre symbol instead of the conventional fraction. In this case a ring homomorphism $\phi : \mathfrak{O} \to \{\pm 1\}$ can be defined for some $\theta \in \mathfrak{O}$ and $k \in \mathbb{Q}$ as

$$\phi(\theta) = \left(\frac{r}{p}\right) \quad \textit{and} \quad \phi(k) = \left(\frac{k}{p}\right)$$

Thus, it is obvious that $\phi(\beta^2) = 1$. On the other hand, since ϕ is a homomorphism and the product of element in U forms a square β^2 in $\mathfrak{O}$, it follows that

$$1 = \phi(\beta^2) = \phi\left(\prod_{a+b\theta \in U} a + b\theta\right) = \prod_{a+b\theta \in U} \phi(a + b\theta) = \prod_{a+b\theta \in U} \left(\frac{a+br}{p}\right)$$

□

As a result, the set of such pairs (p, r) corresponding to the first degree prime ideals $\mathfrak{p} \subset \mathbb{Z}[\theta]$ used for the above test is named as the *quadratic character base*, denoted as Q. The value of $\#Q$ greatly affects the likelihood that the algebraic integer formed by the refined set U of smooth values is really a perfect square. Moreover, as the above result only assures for a product to be a square in $\mathfrak{O}$, in order to turn it into a square in $\mathbb{Z}[\theta]$, another condition is required. Note from an improvement in §2.2 that a perfect square β^2 in $\mathfrak{O}$ can be turned into a corresponding square in $\mathbb{Z}[\theta]$ by multiplying it with $f'(\theta)^2$. Hence, the test is only effective for $f'(\theta)^2\beta^2 \in \mathbb{Z}[\theta]$ if $f'(\theta) \notin \mathfrak{p}$, i.e., $p \nmid f'(r)$ for $\mathfrak{p} \in Q$.

3.3 Finding the perfect square

At this stage the idea of the GNFS has led to the existence of three different bases: the rational factor base F with $\#F = v_f$, the algebraic factor base A with $\#A = v_a$, and the quadratic character base Q with $\#Q = v_q$. Assume that after the sieving step the set S contains a number of pairs (a, b), each of which satisfies that $a + bm$ is smooth over F and $a + b\theta$ is smooth over A for some $\theta \in C$ such that $f(\theta) = 0$, which is not of our concern. Then, the next step concentrates on finding a subset $U \subset S$ with $\#U = v_u$ which satisfies (2.2). In other words, assume that for each $(a_i, b_i) \in U$ such that

$$a_i + b_i m = \prod_{p_j \in F} p_j^{k_{i,j}} \tag{3.3}$$

$$\prod_{(a_i,b_i)\in U} a_i + b_i m = \prod_{(a_i,b_i)\in U} \left(\prod_{p_j \in F} p_i^{k_{i,j}}\right) = \prod_{p_i \in F} p_i^{\sum_{j=1}^{v_u} k_{j,i}}$$

then the following holds for $1 \leq i \leq v_f$:

$$\sum_{j=1}^{v_u} k_{i,j} \equiv 0 \pmod 2 \tag{3.4}$$

Thus, in order to find the set U that satisfies (3.4), it is necessary to represent (3.3) for each pair $(a, b) \in S$ under some form convenient for computing techniques. Note also that since this selection process only considers whether the product is a square, it is sufficient that each $k_{j,i}$ in (3.3) is reduced modulo 2. The simplest way of doing this is to use a v_f-dimensional $\mathbb{F}_2$-vector for each $(a_i, b_i) \in S$:

$$\left\langle k_{i,1}, k_{i,2}, \ldots, k_{i,v_f-1}, k_{i,v_f} \right\rangle \pmod 2 \tag{3.5}$$

Similarly, suppose for each $(a_i, b_i) \in U$ such that

$$a_i + b_i\theta = \prod_{\mathfrak{p}_j \in A} \mathfrak{p}_j^{l_{\mathfrak{p}_j}(a_i+b_i\theta)}$$

$$\prod_{(a_i,b_i)\in U} a_i + b_i\theta = \prod_{(a_i,b_i)\in U} \left(\prod_{\mathfrak{p}_i \in A} \mathfrak{p}_i^{l_{\mathfrak{p}_i}(a_i+b_i\theta)} \right) = \prod_{\mathfrak{p}_i \in A} \mathfrak{p}_i^{\sum_{j=1}^{v_u} l_{\mathfrak{p}_i}(a_j+b_j\theta)}$$

then the similar behavior to (3.4) emerges for each $\mathfrak{p}_i \in A$:

$$\sum_{j=1}^{v_u} l_{\mathfrak{p}_i}(a_j + b_j\theta) \equiv 0 \pmod 2 \tag{3.6}$$

This causes another type of vector to be constructed in the same manner as (3.5) for each $(a_i, b_i) \in S$, but with v_a dimensions:

$$\left\langle l_{\mathfrak{p}_1}(a_i + b_i\theta), l_{\mathfrak{p}_2}(a_i + b_i\theta), \ldots, l_{\mathfrak{p}_{v_a-1}}(a_i + b_i\theta), l_{\mathfrak{p}_{v_a}}(a_i + b_i\theta) \right\rangle \pmod 2 \tag{3.7}$$

At the same time, in satisfying the condition specified in Proposition 3.2.1, a number of different values need to be associated with each $(a_i, b_i) \in S$. They in particular form a v_q-dimensional vector as follows:

$$\left\langle \chi_{q_1}(a_i + b_i s_1), \chi_{q_2}(a_i + b_i s_2), \ldots, \chi_{q_{v_q}}(a_i + b_i s_{v_q}) \right\rangle \tag{3.8}$$

for

$$\chi_p(x) = \left\{ \begin{array}{ll} 0 & \left(\frac{x}{q}\right) = 1 \\ 1 & otherwise \end{array} \right\}$$

Using this vector structure, if the sum of all vectors of $(a, b) \in U$ for each dimension results an even number, then the condition in Proposition 3.2.1 matches, otherwise the product from U is not a square. This structure turns the multiplicative process to the additive one due to the observation that $(-1)(-1) = 1$ can be switched into $1 + 1 \equiv 0 \pmod 2$.

Note that since for each $(a, b) \in S$ there exist distinct instances of vectors (3.5), (3.7), and (3.8), it is even simpler to merge them into a single vector [14, pp. 69-70]. In addition, it is also important to determine the sign of $a + bm$ produced from each $(a, b) \in S$ to measure the overall sign of y^2 mentioned in (2.2). Otherwise if the result is for instance -16, it is not a square even though 16 is a square. In overall, the proper $\mathbb{F}_2$-vector u_i will be of size $1 + v_f + v_a + v_q$ with structure of coordinates as below:

$$\begin{aligned}
&\Big\langle \left\{ \begin{array}{ll} 0 & a_i + b_i m > 0 \\ 1 & \textit{otherwise} \end{array} \right\}, \\
&\qquad k_{i,1} \pmod 2, \qquad k_{i,2} \pmod 2, \quad \ldots, \quad k_{i,v_f} \pmod 2, \\
&l_{\mathfrak{p}_1}(a_i + b_i\theta) \pmod 2, \quad l_{\mathfrak{p}_2}(a_i + b_i\theta) \pmod 2, \quad \ldots, \quad l_{\mathfrak{p}_{v_a}}(a_i + b_i\theta) \pmod 2, \\
&\qquad \chi_{q_1}(a_i + b_i s_1), \qquad \chi_{q_2}(a_i + b_i s_2), \quad \ldots, \quad \chi_{q_{v_q}}(a_i + b_i s_{v_q}) \Big\rangle
\end{aligned}$$

As mentioned, the first dimension of the above vector indicates whether the corresponding $a + bm$ is a positive integer. In the final selection of the set U, if the sum of all values of elements in U results in a even number for this dimension, the product of all $a + bm$ formed by elements in U will be a positive integer, and vice versa. Moreover, in order to assure such a set U can be found satisfying all of the above conditions, it is necessary that the set S of found smooth values must contains more than $1 + v_f + v_a + v_q$ elements, i.e., $v_s = \#S > 1 + v_f + v_a + v_q$. As indicated in [26, Lemma 4.17, p.333], this claim is true as long as the following result holds:

Proposition 3.3.1. *Let V be a vector space spanned by a set of n vectors $u_1, u_2, \ldots, u_n$. If $v_1, v_2, \ldots, v_m \in V$ and $m > n$, then these vectors are linearly dependent.*

Note that the vector space of concern has $d = 1 + v_f + v_a + v_q$ dimensions over $\mathbb{F}_2$, so it is spanned by d unit vectors. Thus since $v_s > d$, all vector u_i $(1 \leq i \leq v_s)$ formed by elements in S are linearly dependent. In other words, there exists a scalar list $c_1, c_2, \ldots, c_{v_s}$ such that

$$\sum_{i=1}^{v_s} c_i u_i = \mathbf{0} \pmod 2 \tag{3.9}$$

where $\mathbf{0}$ is a vector of d dimensions with all components are 0. In this case, the selection of elements in U is straightforward: if c_i is 1 then $(a_i, b_i) \in S$ is picked to the set U, or discarded otherwise.

3.4 The Gaussian elimination method

In order to find the set $\{c_i\}$ as in (3.9), it is necessary that a coefficients matrix $\{k_{i,j}\}$ is formed with size $(1+v_f+v_a+v_q)\times v_s$, where each $k_{i,j}$ corresponds to the i-th component of a vector u_j representing $(a_j, b_j) \in S$. In other words, each column of this matrix $\boldsymbol{B}$ is the vector representing $(a_j, b_j) \in S$. A vector variable $\boldsymbol{x}$ consists of candidates for c_i with a zero-vector $\mathbf{0}$ are declared so as to construct the following equation:

$$\boldsymbol{B}\boldsymbol{x} = \mathbf{0} \pmod 2 \tag{3.10}$$

By solving this equation, a non-trivial solution of $\boldsymbol{x}$ is guaranteed to be obtained which returns the set $\{c_i\}$ as desired. Indeed, the simplest way to deal with (3.10) is to use the *Gaussian elimination* (GE) which reduces the matrix $\boldsymbol{B}$ into *row echelon form* and solves the corresponding system of linear equations from the bottom. In the standard algorithm [33, p. 151], there is however a high probability that the GE will fail to solve the above equation [33, p. 153]. In dealing with this problem, it is necessary to recall the following important concept concerning this method:

Definition 3.4.1. *In Gaussian elimination, a pivot row of a matrix is a row whose first non-zero component is of greatest value within its containing column vector.*

That is to say, if $k_{i,j}$ of a matrix of size $m \times n$ is the first non-zero coefficient in row i and $k_{i,j} \geq k_{t,j}$ for every $1 \leq t \leq m$, then row i is a pivot row. With this concept, the original version of the GE can be modified to increase its stability, i.e., to enhance the probability that a reasonable solution will return, regardless of whether it is trivial or not. The summary of this version of GE is then followed by Algorithm 3.4.1 below.

However, since the complexity of the GE for reducing a matrix of size $n \times m$ is $\mathcal{O}(n \cdot m^2)$, it is very slow for this method to finish if n exceeds 10^7 if the GE is to be applied in the general way as specified by Algorithm 3.4.1. In this situation, it is worth to note that the matrix $\boldsymbol{B}$ has some certain features which could be considered to reduce the amount of computation required.

Algorithm 3.4.1: GaussianWithPivoting

Input: matrix B of size $n \times m$
Output: Row echelon form of B

```
1  begin
2      i ← 1;
3      j ← 1;
4      while i < n or j < m do
           /* Find the j-th pivot row for reduction          */
5          pivot ← i;
6          for k ← i + 1 to m do
7              if |B[k, j]| > |B[i, j]| then
8                  pivot ← k;
9              endif
10         endfor
11         if B[pivot, j] ≠ 0 then
               /* Push the pivoting row up                   */
12             ▷ Switch row i and pivot;
               /* Reduce the pivot row i                     */
13             B[i] ← B[i]/B[i, j];
14             for u ← i + 1 to n do
                   /* Reduce row u by the reduced pivot row i */
15                 B[u] ← B[u] − B[i]B[u, j];
16             endfor
17             i ← i + 1;
18         endif
19         j ← j + 1;
20     endw
21 end
```

Considering an example in which $\boldsymbol{B}$ is roughly of size $10^6 \times 10^6$. This implicitly means the factor bases each contains some 500000 primes. Therefore, every smooth value in accord with its corresponding factor base would contains a trivial number of distinct prime factors compared to the whole base. In other words, the number of 1s occurred in the vector representative of each smooth value associated with elements of the factor bases is very small. On the other hand, note that it is conjecturally sufficient to use a relatively small-size quadratic character base Q (e.g. 100 elements) to harden the square, and so this does not affect the sparsity of 1s in each column vector of $\boldsymbol{B}$. Thus, the matrix $\boldsymbol{B}$ is said to be sparse as it contains mostly 0s. Applying this comment to the GE, it is obvious that the *For* loop on lines

14-16 of Algorithm 3.4.1 may not need to have complexity $\mathcal{O}(n)$, but instead $\mathcal{O}(d)$ where d is the average appearance of 1s in each column vector of $\boldsymbol{B}$, heuristically given by few hundreds.

Meanwhile, since the matrix is concerned over $\mathbb{Z}/2\mathbb{Z}$, it is also not necessary to use the *For* loop on lines 6-9. This is because the pivot row to be chosen is always the row containing the first non-zero component encountered in column j, and by some special way of storing the matrix, this row can be obtained immediately without being searched through the whole matrix.

In overall, the final complexity of the GE could be easily reduced from $\mathcal{O}(nm^2)$ to $\mathcal{O}(ndm)$. With this requirement, the method offered by the GE is still able to deal with matrix of large size, provided that some parallel computing technique must be devised to speed up the computation process. In fact, if a number of independent pivot rows have been successfully found, then all other rows can be independently reduced by these rows at the same time without having to wait for each other to complete. The idea of this concurrent process can be summarized as in Algorithm 3.4.2.

In order to facilitate this algorithm, the matrix $\boldsymbol{B}$ needs to be stored somehow so that it would be feasible to load it into physical memory when performing computation. Since $\boldsymbol{B}$ is vertically sparse, a good idea is to store only non-zero coefficients along with their positions and a linking method for non-zero coefficients in the same row. The linking method is helpful not only when performing row operation, but also when finding pivot rows where leading coefficients lie on the middle vector columns of the matrix. For such reason, each non-zero coefficient needs to be linked with its left and right closest non-zero coefficients on the same row.

As mentioned earlier, even with this improvement, the GE should only be used for reducing small enough matrix. A faster *iterative method* should be used to cope with larger matrices, such as those with size $10^7 \times 10^7$. In the following sections, a method contributed by Lanczos and other researchers is introduced which uses a different approach to the problem and hence significantly lessens the overall running time for solving this matrix equation.

3.5 Standard Lanczos's algorithm

Invented by Cornelius Lanczos in early 1950s and deployed widely since mid-1970s, Lanczos's algorithm became a very fast algorithm for computing *eigen-*

Algorithm 3.4.2: ParallelGauss

Input: matrix B of size $n \times m$
Output: Row echelon form of B

```
1  begin
2      i ← 1; j ← 1;
3      while j < n do
4          pivotSet ← {};
5          while j-th pivot row not exist do
6              j ← j + 1;
7          endw
8          while j-th pivot row exist do
               /* Find and swap j-th pivot row with row i        */
9              ▷ Add j-th pivot row to pivotSet;
10             i ← i + 1; j ← j + 1;
11         endw
           /* Concurrently reduce other rows with pivotSet      */
12         for k where B[k, j] ≠ 0 do
13             for pivot ∈ pivotSet do
14                 B[k] ← B[k] − B[k, j]pivot;
15             endfor
16         endfor
17     endw
18 end
```

values of large, sparse, and symmetric matrices. Moreover, this method is destined to be used in computer program to replace the simple GE algorithm, with support of parallel computing as well. In principle, Lanczos's algorithm and many other iterative methods try to represent the solutions either by a number of deterministic vectors or projecting them onto different subspaces and combine these representations to get the final result. The reason why such methods are regarded as iterative is due to the fact that the construction of the sequence of vectors and subspaces must be computed in order, iteratively using the same algorithm.

As a result, given a symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, a vector $\boldsymbol{b} \in \mathbb{R}^n$, the standard Lanczos's algorithm (iteration) solves the matrix equation $\boldsymbol{Ax} = \boldsymbol{b}$ by firstly computing the following sequence of vectors, as described by P.L.Montgomery in [18, pp. 108-109]:

$$\begin{aligned} &\boldsymbol{w}_0 = \boldsymbol{b} \\ &\boldsymbol{w}_i = \boldsymbol{A}\boldsymbol{w}_{i-1} - \sum_{j=0}^{i-1} c_{i,j}\boldsymbol{w}_j \quad \textit{for} \quad c_{i,j} = \frac{\langle \boldsymbol{A}\boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_{i-1}\rangle}{\langle \boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_j\rangle} = \frac{(\boldsymbol{A}\boldsymbol{w}_j)^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w}_{i-1}}{\boldsymbol{w}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w}_j} \end{aligned} \tag{3.11}$$

where $1 \leq i \leq m$ for some least integer $m \geq 0$ such that $\boldsymbol{w}_{m+1} = \boldsymbol{0}$. Following this construction, the algorithm can be simplified by inferring an important result as below:

Proposition 3.5.1. *Given the sequence of* $\{\boldsymbol{w}_i\}$ *as in* (3.11)*, then for every* $\boldsymbol{w}_i$ *and* $\boldsymbol{w}_j$ *with* $i \neq j$*, the inner product* $\boldsymbol{w}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w}_j$*, denoted as* $\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_j\rangle$*, is* $\boldsymbol{0}$*, and hence the two vectors* $\boldsymbol{w}_i$ *and* $\boldsymbol{w}_j$ *are said to be conjugated with respect to* $\boldsymbol{A}$*.*

Proof. The proof can be done using mathematical induction twice. At the first stage, one needs to prove that the proposition holds for $i = 0$ and $1 \leq j \leq m$. Indeed, considering $j = 1$, the base case follows as:

$$\begin{aligned} \langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_1\rangle &= \left\langle \boldsymbol{w}_0, \boldsymbol{A}\left(\boldsymbol{A} - \frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\right)\boldsymbol{w}_0 \right\rangle \\ &= \langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{A}\boldsymbol{w}_0\rangle - \left\langle \boldsymbol{w}_0, \boldsymbol{A}\frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\boldsymbol{w}_0 \right\rangle \\ &= \boldsymbol{w}_0^{\mathrm{T}}\boldsymbol{A}\boldsymbol{A}\boldsymbol{w}_0 - \boldsymbol{w}_0^{\mathrm{T}}\boldsymbol{A}\frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\boldsymbol{w}_0 \end{aligned} \tag{3.12}$$

Since $\boldsymbol{A}$ is symmetric, it is obvious that $\boldsymbol{A} = \boldsymbol{A}^{\mathrm{T}}$. Furthermore, with the fact that $a^{\mathrm{T}}b^{\mathrm{T}} = (ba)^{\mathrm{T}}$ and the fraction in (3.12) is in $\mathbb{R}$, the manipulation then continues as:

$$\begin{aligned}
(3.12) &= \boldsymbol{w}_0^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w}_0 - \boldsymbol{w}_0^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\boldsymbol{w}_0 \\
&= (\boldsymbol{A}\boldsymbol{w}_0)^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{w}_0) - (\boldsymbol{A}\boldsymbol{w}_0)^{\mathrm{T}}\frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\boldsymbol{w}_0 \\
&= \langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle - \left\langle \boldsymbol{A}\boldsymbol{w}_0, \frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\boldsymbol{w}_0 \right\rangle \\
&= \langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle - \frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{w}_0\rangle = 0
\end{aligned}$$

Assume then that the proposition holds for $i = 0$ and $j = k < m$, the case $j = k + 1$ can now be considered as follows:

$$\begin{aligned}
\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_{k+1}\rangle &= \left\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{A}\boldsymbol{w}_k - \sum_{t=0}^{k}\frac{\langle \boldsymbol{A}\boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_t\rangle}\boldsymbol{A}\boldsymbol{w}_t \right\rangle \\
&= \langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_k\rangle - \sum_{t=0}^{k}\frac{\langle \boldsymbol{A}\boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_t\rangle}\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_t\rangle \\
&= \langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_k\rangle - \frac{\langle \boldsymbol{A}\boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle}\langle \boldsymbol{w}_0, \boldsymbol{A}\boldsymbol{w}_0\rangle = 0
\end{aligned}$$

Thus, the principle of mathematical induction implies that the proposition holds for $i = 0$, and this result can be used as the base case to prove that the same applies when $i = k + 1$, assumed that it is true when $i = k$:

$$\begin{aligned}
\langle \boldsymbol{w}_{k+1}, \boldsymbol{A}\boldsymbol{w}_j\rangle &= \left\langle \boldsymbol{A}\boldsymbol{w}_k - \sum_{t=0}^{k}\frac{\langle \boldsymbol{A}\boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_t\rangle}\boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_j \right\rangle \\
&= \langle \boldsymbol{A}\boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_j\rangle - \sum_{t=0}^{k}\frac{\langle \boldsymbol{A}\boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_t\rangle}\langle \boldsymbol{w}_t, \boldsymbol{A}\boldsymbol{w}_j\rangle \\
&= \langle \boldsymbol{A}\boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_j\rangle - \frac{\langle \boldsymbol{A}\boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_j\rangle}\langle \boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_j\rangle \overset{j\neq k}{=} 0 \\
&\overset{j=k}{=} \langle \boldsymbol{A}\boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_k\rangle - \frac{\langle \boldsymbol{A}\boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_k\rangle}{\langle \boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_k\rangle}\langle \boldsymbol{w}_k, \boldsymbol{A}\boldsymbol{w}_k\rangle = 0
\end{aligned}$$

□

According to the above proposition, the vectors $\boldsymbol{w}_i$ are independent of each others, which means their *span* is well defined. On the other hand, this result leads to a simplification of the computation of those $\boldsymbol{w}_i$ in the following observation [24, Lemma 2.1.1]:

Proposition 3.5.2. *Given the constants $c_{i,j}$ defined above, then $c_{i,j} = 0$ for every $j < i - 2$.*

Therefore, most of the terms appeared in the computation of $\boldsymbol{w}_i$ will vanish as they are multiplied by 0, except for the closet two terms:

$$\boldsymbol{w}_i = \boldsymbol{A}\boldsymbol{w}_{i-1} - c_{i,i-1}\boldsymbol{w}_{i-1} - c_{i,i-2}\boldsymbol{w}_{i-2} \quad \textit{for} \quad i \geq 2 \tag{3.13}$$

The solution is then revealed by the following equation:

Proposition 3.5.3. *Given a symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, a vector $\boldsymbol{b} \in \mathbb{R}^n$ and a collection of vectors $\boldsymbol{w}_i$ constructed as above. Then the following holds:*

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}\sum_{i=0}^{m} \frac{\langle \boldsymbol{w}_i, \boldsymbol{b}\rangle}{\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_i\rangle}\boldsymbol{w}_i = \boldsymbol{b} \tag{3.14}$$

Proof. Firstly, it is important to prove that $\langle \boldsymbol{A}\boldsymbol{x}, \boldsymbol{w}_i\rangle = \langle \boldsymbol{b}, \boldsymbol{w}_i\rangle$. Indeed, since the fraction in the composition of $\boldsymbol{x}$ is in $\mathbb{R}$,

$$\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{x}\rangle = \sum_{j=0}^{m} \frac{\langle \boldsymbol{w}_j, \boldsymbol{b}\rangle}{\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_i\rangle} \langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_j\rangle = \frac{\langle \boldsymbol{w}_i, \boldsymbol{b}\rangle}{\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_i\rangle} \langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_i\rangle = \langle \boldsymbol{w}_i, \boldsymbol{b}\rangle$$

and hence $\langle \boldsymbol{A}\boldsymbol{x}, \boldsymbol{w}_i\rangle = \langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{x}\rangle^{\mathrm{T}} = \langle \boldsymbol{w}_i, \boldsymbol{b}\rangle^{\mathrm{T}} = \langle \boldsymbol{b}, \boldsymbol{w}_i\rangle$

Moreover, from (3.11) and the construction of $\boldsymbol{x}$ it can be inferred that $\boldsymbol{b}, \boldsymbol{x} \in \mathit{Span}\{\boldsymbol{w}_i\}$. Note also that $\boldsymbol{A}\boldsymbol{w}_i \in \mathit{Span}\{\boldsymbol{w}_i\}$ by (3.11), hence $\boldsymbol{A}\boldsymbol{x} \in \mathit{Span}\{\boldsymbol{A}\boldsymbol{w}_i\} \subset \mathit{Span}\{\boldsymbol{w}_i\}$. Thus $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} \in \mathit{Span}\{\boldsymbol{w}_i\}$ and can be represented as a combination of $\boldsymbol{w}_i$. It is now straightforward to prove the proposition as follows:

$$\begin{aligned}
\langle \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\rangle &= \left\langle \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \sum_{i=0}^{m} k_i\boldsymbol{w}_i \right\rangle \\
&= \sum_{i=0}^{m} k_i \langle \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \boldsymbol{w}_i\rangle = \sum_{i=0}^{m} k_i \left(\langle \boldsymbol{A}\boldsymbol{x}, \boldsymbol{w}_i\rangle \ - \langle \boldsymbol{b}, \boldsymbol{w}_i\rangle\right) \\
&= \sum_{i=0}^{m} k_i \left(\langle \boldsymbol{A}\boldsymbol{x}, \boldsymbol{w}_i\rangle \ - \langle \boldsymbol{b}, \boldsymbol{w}_i\rangle\right) = 0 \Leftrightarrow \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}
\end{aligned}$$

□

In the construction of $\boldsymbol{w}_i$ described by (3.13), the only computation at each iteration is the multiplication of the matrix $\boldsymbol{A}$ with $\boldsymbol{w}_{i-1}$. Due to the fact that $\boldsymbol{A}$ is sparse and the total number of non-zero entries is dn for d much less than n, this multiplication can be taken with complexity $\mathcal{O}(dn)$ by representing the matrix $\boldsymbol{A}$ by its list of non-zero entries and multiply them one by one to each corresponding component of the vector $\boldsymbol{w}_i$. Since the number of $\boldsymbol{w}_i$ is at most n so that they can be linearly independent, the overall complexity of the method is $\mathcal{O}(dn^2)$, which is approximately the same as the GE, except that it is much simpler to implement than in the case of the GE.

The problem, however, comes into place when this method is applied to solve the matrix equation created by the GNFS. In particular, the vector $\boldsymbol{b}$ is $\boldsymbol{0}$ in the first place, which means no $\boldsymbol{w}_i$ will return. In addition, since this matrix equation of the GNFS is concerned in $\mathbb{Z}/2\mathbb{Z}$, it is very likely that a vector $\boldsymbol{w}_i$ if created is self-orthogonal, i.e., $\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_i \rangle = 0$, making the coefficients $c_{i,j}$ in (3.13) meaningless.

To address this problem, the vector $\boldsymbol{b}$ can simply be produced from the combination of some vector columns in $\boldsymbol{A}$ and multiplied by 2 to to get even components. In order to generate the symmetric matrix $\boldsymbol{A}$ from $\boldsymbol{B}$, the matrix equation can be turned into $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b}$ where there is a high probability that a solution would satisfy the original one. The remaining task is to apply Lanczos's algorithm over $\mathbb{R}$ instead of $\mathbb{Z}/2\mathbb{Z}$ with some hope to have a non-trivial solution. This solution when modulo 2 will be the desired result for the matrix equation of the GNFS.

The next section describes how this algorithm could be further improved to bundle a number of vectors $\boldsymbol{w}_i$ into blocks and perform the computation for each iteration the same as above but with the number of iterations greatly reduced. In addition, it will also secure the chance that a solution can be found, unlike the application of the standard version as mentioned above.

3.6 The Block Lanczos's algorithm

After the success of the standard Lanczos's algorithm, some attempts were spent to make enhancements over this method. In 1995, Montgomery published a new improvement of Lanczos's idea which groups the vectors $\boldsymbol{w}_i$ into blocks of vectors, or matrices. This new method requires less iterations while still possesses the same computing speed at each iteration, and hence it came

up with a positive achievement for solving matrix equations.

In this context, instead of finding pairwise $\boldsymbol{A}$-conjugate vectors $\boldsymbol{w}_i$, the algorithm targets at collecting a number of pairwise $\boldsymbol{A}$-orthogonal subspaces $\mathcal{W}_i$ of $\mathbb{F}_2^n$. For the possibility of this method, the following definition is in effective use:

Definition 3.6.1. *Let $\boldsymbol{A}$ be an $n \times n$ symmetric matrix. A subspace $\mathcal{W} \subseteq \mathcal{K}^n$ is called $\boldsymbol{A}$-**invertible** if its any basis $\boldsymbol{W}$satisfies that $\boldsymbol{W}^{\mathrm{T}} \boldsymbol{A}\, \boldsymbol{W}$ is invertible.*

Definition 3.6.2. *Let $\boldsymbol{A}$ be an $n \times n$ symmetric matrix. Two subspaces $\mathcal{W}_i, \mathcal{W}_j \subset \mathcal{K}^n$ are said to be $\boldsymbol{A}$-**orthogonal** if for every $\boldsymbol{w}_i \in \mathcal{W}_i$ and $\boldsymbol{w}_j \in \mathcal{W}_j$, $\langle \boldsymbol{w}_i, \boldsymbol{A}\boldsymbol{w}_j \rangle = \langle \boldsymbol{w}_j, \boldsymbol{A}\boldsymbol{w}_i \rangle = 0$.*

In addition, the process of projecting a vector onto an $\boldsymbol{A}$-invertible subspace can be done as below [24, pp. 11-12]:

Proposition 3.6.1. *If a subspace $\mathcal{W} \subseteq \mathcal{K}^n$ is $\boldsymbol{A}$-invertible, then for every $\boldsymbol{u} \in \mathcal{K}^n$, there exists uniquely a vector $\boldsymbol{w} \in \mathcal{W}$ and a vector $\boldsymbol{v} \in \mathcal{K}^n$ such that $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{v}$ where $\boldsymbol{v}$ is $\boldsymbol{A}$-orthogonal to $\mathcal{W}$. Furthermore, if column vectors of some matrix $\boldsymbol{W}$span $\mathcal{W}$, then*

$$\boldsymbol{w} = \boldsymbol{W}(\, \boldsymbol{W}^{\mathrm{T}} \boldsymbol{A}\, \boldsymbol{W})^{-1}\, \boldsymbol{W}^{\mathrm{T}} \boldsymbol{A}\, \boldsymbol{u}$$

Thus, the vector $\boldsymbol{w}$ in Proposition 3.6.1 is the projection of $\boldsymbol{u}$ onto the subspace $\mathcal{W}$. Similar to the standard version, the next proposition summarizes the heart of the block Lanczos's algorithm:

Proposition 3.6.2. *Let $\boldsymbol{A} \in \mathcal{K}^{n \times n}$. Let $\{\mathcal{W}_i\}_{i=0}^m$ be a set of pairwise $\boldsymbol{A}$-orthogonal subspaces such that each $\mathcal{W}_i$ is $\boldsymbol{A}$-invertible and $\mathcal{W} = \sum \mathcal{W}_i$ is $\boldsymbol{A}$-invariant. Then for each $\boldsymbol{b} \in \mathcal{W}$, the set of vectors $\{\boldsymbol{w}_i\}_{i=0}^m$ where $\boldsymbol{w}_i \in \mathcal{W}_i$ and that each $\boldsymbol{A}\boldsymbol{w}_i - \boldsymbol{b}$ is orthogonal to $\mathcal{W}_i$, forms the solution for the equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. Equivalently, $\boldsymbol{x}$ can be computed as below:*

$$\boldsymbol{x} = \sum_{i=0}^{m} \boldsymbol{W}_i(\, \boldsymbol{W}_i^{\mathrm{T}} \boldsymbol{A}\, \boldsymbol{W}_i)^{-1}\, \boldsymbol{W}_i^{\mathrm{T}} \boldsymbol{b} \tag{3.15}$$

where column vectors of $\boldsymbol{W}_i$ form a basis for $\mathcal{W}_i$.

In order to compute the subspaces $\mathcal{W}_i$, or more precisely the matrices $\boldsymbol{W}_i$, it is possible to use the following iterative algorithm:

Proposition 3.6.3. *Let $\boldsymbol{A} \in \mathcal{K}^{n\times n}$ and $\boldsymbol{V}_0$ be an $\boldsymbol{A}$-invertible matrix in $\mathcal{K}^{n\times N}$. The matrices $\boldsymbol{W}_i$ that form the bases for subspaces $\mathcal{W}_i$ described in Proposition 3.6.2 can be constructed as follows:*

$$\begin{aligned} \boldsymbol{W}_i &= \boldsymbol{V}_i \boldsymbol{S}_i \\ \boldsymbol{V}_{i+1} &= \boldsymbol{A}\,\boldsymbol{W}_i \boldsymbol{S}_i^{\mathrm{T}} + \boldsymbol{V}_i - \sum_{j=0}^{i} \boldsymbol{W}_j \boldsymbol{C}_{i+1,j} \end{aligned} \tag{3.16}$$

where each $\boldsymbol{S}_i$ is such that the $\boldsymbol{A}$-invertible matrix $\boldsymbol{W}_i$ is the largest collection of columns in $\boldsymbol{V}_i$. Moreover, each $\boldsymbol{C}_{i,j}$ must be of the form:

$$\boldsymbol{C}_{i+1,j} = (\,\boldsymbol{W}_j^{\mathrm{T}} \boldsymbol{A}\,\boldsymbol{W}_j)^{-1}\,\boldsymbol{W}_j^{\mathrm{T}} \boldsymbol{A}(\boldsymbol{A}\,\boldsymbol{W}_i \boldsymbol{S}_i^{\mathrm{T}} + \boldsymbol{V}_i) \tag{3.17}$$

The main idea for using $\boldsymbol{S}_i$ bases on the fact that $\boldsymbol{V}_i$ may not be $\boldsymbol{A}$-invertible, and thus cannot be chosen to be $\boldsymbol{W}_i$. However, a number of columns in $\boldsymbol{V}_i$ may be chosen which satisfy this condition. In order to simplify the computation of $\boldsymbol{V}_i$, a similar observation to the standard Lanczos's algorithm can be done to reduce it to a few recurrences:

Proposition 3.6.4. *If $i > j$, then $\boldsymbol{V}_i^{\mathrm{T}} \boldsymbol{A}\,\boldsymbol{W}_j = 0$ and hence $\boldsymbol{C}_{i+1,j}$ in* (3.17) *can be reduced to:*

$$\boldsymbol{C}_{i+1,j} = (\,\boldsymbol{W}_j^{\mathrm{T}} \boldsymbol{A}\,\boldsymbol{W}_j)^{-1}\,\boldsymbol{W}_j^{\mathrm{T}} \boldsymbol{A}^2\,\boldsymbol{W}_i \boldsymbol{S}_i^{\mathrm{T}} \tag{3.18}$$

Moreover, if the process of selecting columns in $\boldsymbol{V}_i$ is more careful so that each column of $\boldsymbol{V}_i$ belongs either to $\boldsymbol{W}_i$ or $\boldsymbol{W}_{i+1}$, then most of the terms in computing $\boldsymbol{V}_i$ could be successfully canceled , that is,

Proposition 3.6.5. *If the span of column vectors in $\boldsymbol{V}_i$, denoted $\langle\,\boldsymbol{V}_i\rangle$, is a subset of $\sum_{j=0}^{i+1} \mathcal{W}_j$, then for $j < i-2$, $\boldsymbol{C}_{i+1,j} = 0$*

With this result, the construction of $\boldsymbol{V}_i$ in (3.16) is simplified to three terms recurrence, i.e.,

$$\boldsymbol{V}_{i+1} = \boldsymbol{A}\,\boldsymbol{W}_i \boldsymbol{S}_i^{\mathrm{T}} + \boldsymbol{V}_i - \boldsymbol{W}_i \boldsymbol{C}_{i+1,i} - \boldsymbol{W}_{i-1} \boldsymbol{C}_{i+1,i-1} - \boldsymbol{W}_{i-2} \boldsymbol{C}_{i+1,i-2} \tag{3.19}$$

Note that since the size of the matrix A is large, even though the algorithm is equipped with the result above, it is still time consuming to naively compute each iteration in such way. Therefore, Montgomery gave further clarification on what to compute and what to be inferred [18, pp.112-113]:

Proposition 3.6.6. *The computation of $\boldsymbol{V}_i$ as in* (3.19) *can be turned into*

$$\boldsymbol{V}_{i+1} = \boldsymbol{A}\,\boldsymbol{V}_i \boldsymbol{S}_i \boldsymbol{S}_i^{\mathrm{T}} + \boldsymbol{V}_i \boldsymbol{D}_{i+1} + \boldsymbol{V}_{i-1} \boldsymbol{E}_{i+1} + \boldsymbol{V}_{i-2} \boldsymbol{F}_{i+1} \tag{3.20}$$

where

$$\begin{aligned}
\boldsymbol{D}_{i+1} &= \boldsymbol{I}_N - \boldsymbol{W}_i^{\text{inv}} \left(\boldsymbol{V}_i^{\text{T}} \boldsymbol{A}^2 \boldsymbol{V}_i \boldsymbol{S}_i \boldsymbol{S}_i^{\text{T}} + \boldsymbol{V}_i^{\text{T}} \boldsymbol{A} \boldsymbol{V}_i \right) \\
\boldsymbol{E}_{i+1} &= -\boldsymbol{W}_{i-1}^{\text{inv}} \boldsymbol{V}_i^{\text{T}} \boldsymbol{A} \boldsymbol{V}_i \boldsymbol{S}_i \boldsymbol{S}_i^{\text{T}} \\
\boldsymbol{F}_{i+1} &= -\boldsymbol{W}_{i-2}^{\text{inv}} \left(\boldsymbol{I}_N - \boldsymbol{V}_{i-1}^{\text{T}} \boldsymbol{A} \boldsymbol{V}_{i-1} \boldsymbol{W}_{i-1}^{\text{inv}} \right) \left(\boldsymbol{V}_{i-1}^{\text{T}} \boldsymbol{A}^2 \boldsymbol{V}_{i-1} \boldsymbol{S}_{i-1} \boldsymbol{S}_{i-1}^{\text{T}} + \boldsymbol{V}_{i-1}^{\text{T}} \boldsymbol{A} \boldsymbol{V}_{i-1} \right) \boldsymbol{S}_i \boldsymbol{S}_i^{\text{T}} \\
\boldsymbol{W}_i^{\text{inv}} &= \boldsymbol{S}_i \left(\boldsymbol{S}_i^{\text{T}} \boldsymbol{V}_i^{\text{T}} \boldsymbol{A} \boldsymbol{V}_i \boldsymbol{S}_i \right)^{-1} \boldsymbol{S}_i^{\text{T}}
\end{aligned}$$

Since for any arbitrary vector $\boldsymbol{b}$, the process of finding $\boldsymbol{x}$ such that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is exactly the same, the blocked version of Lanczos's algorithm can also be applied to find the solution for the equation $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B}$ where $\boldsymbol{X}$ and $\boldsymbol{B}$ are matrices of size $n \times N$. In other words, it is analogous to the process of finding solutions for N different equations at the same time. Note also that it is convenient to substitute $\boldsymbol{W}_i^{\text{inv}}$ into (3.15) to get

$$\boldsymbol{x} = \sum_{i=0}^{m} \boldsymbol{V}_i \boldsymbol{S}_i (\boldsymbol{W}_i^{\text{T}} \boldsymbol{A} \boldsymbol{W}_i)^{-1} \boldsymbol{S}_i^{\text{T}} \boldsymbol{V}_i^{\text{T}} \boldsymbol{b} = \sum_{i=0}^{m} \boldsymbol{V}_i \boldsymbol{W}_i^{\text{inv}} \boldsymbol{V}_i^{\text{T}} \boldsymbol{b} \tag{3.21}$$

Meanwhile, even though the formulae in Proposition 3.6.6 look complicated, they in fact have been categorized into groups of known and unknown components so that the computation at each iteration can be explicitly specified, as shown in Algorithm 3.6.1.

According to this algorithm, the heaviest matrix operations within each iteration fall into the following points:

- Line 18: in the case of the GNFS, the matrix $\boldsymbol{A}$ is formed by $\boldsymbol{B}^{\text{T}}\boldsymbol{B}$. Hence, there are two ways to multiply $\boldsymbol{A}$ by $\boldsymbol{V}_i$, either as $\left(\boldsymbol{B}^{\text{T}}\boldsymbol{B}\right)\boldsymbol{V}_i$ or $\boldsymbol{B}^{\text{T}}(\boldsymbol{B}\boldsymbol{V}_i)$. The former will compute $\boldsymbol{A}$ first in an extremely fast manner, but later on it would face a problem in computing $\boldsymbol{A}\boldsymbol{V}_i$ since $\boldsymbol{A}$ can no longer be guaranteed as sparse, even if $\boldsymbol{B}$ is sparse. The latter method, i.e., $\boldsymbol{B}^{\text{T}}(\boldsymbol{B}\boldsymbol{V}_i)$, performs twice the computation of an $m \times n$ and an $n \times N$ matrices with complexity $\mathcal{O}(dn)$ using *outer product*. In this case dn stands for the number of non-zero entries in $\boldsymbol{B}$. Using this method avoids us from storing the matrix $\boldsymbol{A}$ and restricts the theoretical running time to $\mathcal{O}(dn)$.

- Line 19: using Coppersmith's method mentioned in [24, pp. 38-40], the multiplication of an $N \times n$ by an $n \times N$ matrices is reduced from $\mathcal{O}(Nn)$ to $\mathcal{O}(n)$.

- Line 20: by storing the sum of each column vector of a_i, the computation of $a_i^{\text{T}} a$ would require only $\mathcal{O}(N^2)$ where $a_i \in \mathbb{F}_2^{n \times N}$.

Algorithm 3.6.1: BlockLanczos

Input: $\boldsymbol{Y} \in \mathcal{K}^{n\times N}$, symmetric $\boldsymbol{A} \in \mathcal{K}^{n\times n}$
Output: $\boldsymbol{X} \in \mathcal{K}^{n\times N}$ such that $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{Y}$, last $\boldsymbol{V}_i$

begin
  /* Algorithm initialization */
  $\boldsymbol{V}_{-2} \leftarrow \boldsymbol{V}_{-1} \leftarrow \boldsymbol{0}$; /* $V_i \in \mathcal{K}^{n\times N}$ */
  $\boldsymbol{V}_0 \leftarrow \boldsymbol{Y}$;
  $d_{-1} \leftarrow \boldsymbol{S}_{-1}\boldsymbol{S}_{-1}^{\mathrm{T}} \leftarrow \boldsymbol{I}_N$; /* $S_i \in \mathcal{K}^{N\times Rank(V_i)}$, $S_i S_i^{\mathrm{T}} \in \mathcal{K}^{N\times N}$ */
  $\boldsymbol{W}_{-2}^{\mathrm{inv}} \leftarrow \boldsymbol{W}_{-1}^{\mathrm{inv}} \leftarrow \boldsymbol{0}$; /* $W_i^{\mathrm{inv}} \in \mathcal{K}^{N\times N}$ */
  $a_{-1} \leftarrow b_{-1} \leftarrow c_{-1} \leftarrow \boldsymbol{0}$; /* $a_i, b_i, c_i \in \mathcal{K}^{n\times n}$ */
  $a_0 \leftarrow \boldsymbol{A}\,\boldsymbol{V}_0$;
  $b_0 \leftarrow \boldsymbol{V}_0^{\mathrm{T}} a_0$;
  $c_0 \leftarrow a_0^{\mathrm{T}} a_0$;
  ▷ Compute $\boldsymbol{W}_0^{\mathrm{inv}}$ and $d_0 = \boldsymbol{S}_0\boldsymbol{S}_0^{\mathrm{T}}$ from d_{-1} and b_0;
  $\boldsymbol{X} \leftarrow \boldsymbol{V}_0\,\boldsymbol{W}_0^{\mathrm{inv}}\,\boldsymbol{V}_i^{\mathrm{T}}\,\boldsymbol{Y}$; /* $X \in \mathcal{K}^{n\times N}$ */
  /* Main iterating loop */
  $i \leftarrow 0$;
  while $b_i \neq \boldsymbol{0}$ **do**
    $\boldsymbol{D}_{i+1} \leftarrow \boldsymbol{I}_N - \boldsymbol{W}_i^{\mathrm{inv}}\,(c_i d_i + b_i)$; /* $D_i \in \mathcal{K}^{N\times N}$ */
    $\boldsymbol{E}_{i+1} \leftarrow -\boldsymbol{W}_{i-1}^{\mathrm{inv}} b_i d_i$; /* $E_i \in \mathcal{K}^{N\times N}$ */
    $\boldsymbol{F}_{i+1} \leftarrow -\boldsymbol{W}_{i-2}^{\mathrm{inv}}\left(\boldsymbol{I}_N - a_{i-1}\,\boldsymbol{W}_{i-1}^{\mathrm{inv}}\right)(c_{i-1}d_{i-1} + b_{i-1})\,d_i$; /* $F_i \in \mathcal{K}^{N\times N}$ */
    $\boldsymbol{V}_{i+1} \leftarrow a_i d_i + \boldsymbol{V}_i\boldsymbol{D}_{i+1} + \boldsymbol{V}_{i-1}\boldsymbol{E}_{i+1} + \boldsymbol{V}_{i-2}\boldsymbol{F}_{i+1}$;
    $a_{i+1} \leftarrow \boldsymbol{A}\,\boldsymbol{V}_{i+1}$;
    $b_{i+1} \leftarrow \boldsymbol{V}_{i+1}^{\mathrm{T}} a_{i+1}$;
    $c_{i+1} \leftarrow a_{i+1}^{\mathrm{T}} a_{i+1}$;
    ▷ Compute $\boldsymbol{W}_{i+1}^{\mathrm{inv}}$ and $d_{i+1} = \boldsymbol{S}_{i+1}\boldsymbol{S}_{i+1}^{\mathrm{T}}$ from d_i and b_{i+1};
    $\boldsymbol{X} \leftarrow \boldsymbol{X} + \boldsymbol{V}_{i+1}\,\boldsymbol{W}_{i=1}^{\mathrm{inv}}\,\boldsymbol{V}_{i+1}^{\mathrm{T}}\,\boldsymbol{Y}$; /* Derived from (3.21) */
    $i \leftarrow i + 1$;
  endw
end

- Line 21: this step requires at most complexity $\mathcal{O}(N^2)$ which is relatively small, as will be shown in the next section.

- Line 17: there exist four inner products between $n \times N$ and $N \times N$ matrices as well as four corresponding additions among them. The additions can be done easily with $\mathcal{O}(n)$ complexity. Meanwhile, each inner product can be done also with $\mathcal{O}(n)$ complexity by dividing the longer matrix into blocks of $N \times N$ matrices and multiply each by the targeting $N \times N$ matrix using Coppersmith's method with complexity $\mathcal{O}(N)$ and thus in overall will be $\mathcal{O}(N(n/N)) = \mathcal{O}(n)$.

- Line 22: the computation of $\boldsymbol{V}_{i+1}^{\mathrm{T}} \boldsymbol{Y}$ and $\boldsymbol{V}_{i+1} \boldsymbol{W}_{i+1}^{\mathrm{inv}}$ would both require $\mathcal{O}(n)$ as well as their final multiplication. So the total running time of this step would be $\mathcal{O}(n)$.

The above points indicate that the complexity at each iteration is $\mathcal{O}(dn)$. As claimed by Montgomery [18, p. 118], the conjectured number of iterations is at most $\mathcal{O}(n/N)$ for if the input matrix $\boldsymbol{Y}$ is completely random. Combined these two results, the blocked version of Lanczos's algorithm has complexity $\mathcal{O}(dn/N)$, which is N times faster than Gaussian elimination. The main advantage in this algorithm is the use of simultaneous binary operations in N-bit computer since the computation uses as much time as would be done for 1 bit, whereas the method specified in the GE cannot exploit this idea with large and sparse matrices.

3.7 Constraints of Block Lanczos's algorithm

In order to select $\boldsymbol{S}_i$ and $\boldsymbol{W}_i^{\mathrm{inv}}$ so that the condition of Proposition 3.6.5 is satisfied, a separate algorithm is invoked which acts as a closed procedure in the implementation. It is necessary that the following compact result is employed in understanding this selection process:

Proposition 3.7.1. *Given the sequence of $\boldsymbol{V}_i$ computed as in Algorithm 3.6.1, then for every $\boldsymbol{V}_i$, all of its columns which were not included in $\boldsymbol{W}_i$ appear in $\boldsymbol{V}_{i+1}$ at the exact column positions.*

Proof. Let $\boldsymbol{S}_i$ be the selecting matrix for $\boldsymbol{W}_i$, then it's clear that $\boldsymbol{S}_i \boldsymbol{S}_i^{\mathrm{T}}$ have similar structure to $\boldsymbol{I}_N$, except that some coefficients on the diagonal are 0 as they stand for columns of $\boldsymbol{V}_i$ not selected to be included in $\boldsymbol{W}_i$ in the previous iteration. Considering the formula used to compute $\boldsymbol{V}_{i+1}$ in Proposition 3.6.6, the following comments can be made for the four terms:

- Term 1,3,4: Since each of these terms contains $\boldsymbol{S}_i\boldsymbol{S}_i^{\mathrm{T}}$, all columns whose positions match those columns not selected in the previous iteration are $\boldsymbol{0}$ vectors.
- Term 2: It is easy to check that the multiplication $\boldsymbol{S}_i\boldsymbol{X}\boldsymbol{S}_i^{\mathrm{T}}$ would result in an $N \times N$ matrix with $\boldsymbol{0}$ column vectors at positions match with columns not selected into $\boldsymbol{W}_i$ from $\boldsymbol{V}_i$. Thus it is also true for $\boldsymbol{W}_i^{\mathrm{inv}}$ by assumption, and hence ensures that in the diagonal of $\boldsymbol{D}_{i+1}$, all positions indicating columns not selected will have value of 1. By multiplying $\boldsymbol{D}_{i+1}$ by $\boldsymbol{V}_i$, all the columns not selected in $\boldsymbol{V}_i$ now appear in the second term at their exact positions.

This implies that the sum of these terms, i.e., $\boldsymbol{V}_{i+1}$ contains all the columns not selected in the previous iteration, at their exact positions. □

Another important result is required to make sure that the selection process successfully finds $\boldsymbol{W}_i$ which is $\boldsymbol{A}$-invertible. This result was explicitly proved in [24, Lemma 3.3.1], while its idea can be summarized as below:

Proposition 3.7.2. *Let $\boldsymbol{V}$ be a symmetric matrix of size $N \times N$ with $r = Rank(\boldsymbol{V})$. Let $\boldsymbol{V}'$ be a matrix formed by selecting any r linearly independent rows of $\boldsymbol{V}$. Let $\boldsymbol{V}''$ be a matrix formed by selecting some columns of $\boldsymbol{V}'$ whose positions are the same with positions of rows selected by $\boldsymbol{V}'$ from $\boldsymbol{V}$. Then $\boldsymbol{V}''$ is invertible.*

Thus if $\boldsymbol{V}'' = \boldsymbol{W}_i^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{W}_i$ is found from $\boldsymbol{V} = \boldsymbol{V}_i^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{V}_i$, then the condition is satisfied. In order to select a set of columns in $\boldsymbol{V}_i$ which form an $\boldsymbol{A}$-invertible matrix, an implicit version of the GE is in used to reduce $\boldsymbol{V}_i^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{V}_i$ and select the pivot columns (rows) as the desired choices. In addition, since there must be an assurance that all previously not selected columns must now be selected as well, these columns are switched to the beginning of $\boldsymbol{V}_i^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{V}_i$ so that during the reduction process, they will be chosen as pivot columns. Proposed by Montgomery [18, p. 116], the method can be summarized as in Algorithm 3.7.1.

To sum up, all the components required by Block Lanczos have so far been revealed. Note that from the beginning a symmetric matrix $\boldsymbol{A}$ and a matrix $\boldsymbol{Y}$ also need to be initialized. As previously mentioned, the matrix $\boldsymbol{A}$ can be of the form $\boldsymbol{B}^{\mathrm{T}}\boldsymbol{B}$. To form $\boldsymbol{Y}$ for Algorithm 3.6.1, the algorithm may start by randomly initializing a matrix $\boldsymbol{Y}_0$ of binary coefficients. Then, by letting $\boldsymbol{Y} = \boldsymbol{A}\,\boldsymbol{Y}_0$, this matrix is ready as the appropriate input, so that the solution $\boldsymbol{X}$ satisfies that $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{A}\,\boldsymbol{Y}_0$. Note however that the terminating condition for Algorithm 3.6.1 is $\boldsymbol{V}_i^{\mathrm{T}}\boldsymbol{A}\,\boldsymbol{V}_i = 0$, hence it might happen that the

Algorithm 3.7.1: SelectWinvAndSi

Input: Matrix $b_i \in \mathcal{K}^{N\times N}$ in Algorithm 3.6.1, $\boldsymbol{S}_{i-1}\boldsymbol{S}_{i-1}^{\mathrm{T}}$
Output: $\boldsymbol{S}_i\boldsymbol{S}_i^{\mathrm{T}}$, $\boldsymbol{W}_i^{\mathrm{inv}}$

begin
  $\boldsymbol{M} \leftarrow (b_i|\boldsymbol{I}_N)$; /* Join I_N to the left of b */
  ▷ Switch columns of $\boldsymbol{M}$ whose positions correspond to zero columns in $\boldsymbol{S}_{i-1}\boldsymbol{S}_{i-1}^{\mathrm{T}}$ to the beginning of $\boldsymbol{M}$;
  $ss \leftarrow \boldsymbol{0}$; /* Initialize $S_iS_i^{\mathrm{T}} \in \mathcal{K}^{N\times N}$ */
  for $j \leftarrow 1$ **to** N **do**
    $k \leftarrow j$;
    while $k \leq N$ **do**
      if $\boldsymbol{M}[k, j] \neq 0$ **then** ▷ Break this loop;
      $k \leftarrow k + 1$;
    endw
    if $k \leq N$ **then** /* If $M[k, j] \neq 0$ */
      ▷ Exchange row j and k of $\boldsymbol{M}$;
      $ss[j, j] = 1$; /* Add column j of V_i into W_i */
      ▷ Divide row j of $\boldsymbol{M}$ by $\boldsymbol{M}[j, j]$; /* Omit if use F_2 */
      /* Zero the rest of column j */
      ▷ Reduce rows of $\boldsymbol{M}$ by row j;
    else /* If column j of M is a zero vector */
      $k \leftarrow j$;
      while $k \leq N$ **do**
        if $\boldsymbol{M}[k, j + N] \neq 0$ **then** ▷ Break this loop;
        $k \leftarrow k + 1$;
      endw
      if $k \leq N$ **then** /* Assert that $M[k, j + N] \neq 0$ */
        ▷ Exchange row j and k of $\boldsymbol{M}$;
        /* Zero the rest of column $j + N$ */
        ▷ Reduce rows of $\boldsymbol{M}$ by row j;
      endif
    endif
  endfor
  $\boldsymbol{W}_i^{\mathrm{inv}} \leftarrow$ Right half of $\boldsymbol{M}$;
  $\boldsymbol{S}_i\boldsymbol{S}_i^{\mathrm{T}} \leftarrow ss$;
end

algorithm terminates with non-zero value of $\boldsymbol{V}_i$, and thus $\boldsymbol{A}\boldsymbol{X} \neq \boldsymbol{A}\boldsymbol{Y}_0$. However, for a reason described in [18, p. 114], the vectors contained in $\boldsymbol{X} - \boldsymbol{Y}_0$ and the last $\boldsymbol{V}_i$ are in the nullspace of $\boldsymbol{A}$. Thus, if we let $\boldsymbol{Z} = (\boldsymbol{X} - \boldsymbol{Y}_0 | \, \boldsymbol{V}_i)$, and use the Gaussian elimination to find the nullspace $\boldsymbol{U}$ of $\boldsymbol{B}\boldsymbol{Z}$, then $\boldsymbol{Z}\boldsymbol{U}$ is the desired solution. The main algorithm for solving the matrix equation is then followed in Algorithm 3.7.2.

Algorithm 3.7.2: FindDependencies

Input: Matrix $\boldsymbol{B} \in \mathbb{F}_2^{m \times n}$ representing smooth values in S with size n
Output: A matrix $\boldsymbol{X}_0 \in \mathbb{F}_2^{m \times N}$ as solutions

begin
  $\boldsymbol{A} \leftarrow \boldsymbol{B}^{\mathrm{T}}\boldsymbol{B}$; /* $\boldsymbol{A} \in \mathbb{F}_2^{n \times n}$ */
  $\boldsymbol{Y}_0 \leftarrow Random(0, 1)$; /* $\boldsymbol{Y}_0 \in \mathbb{F}_2^{n \times N}$ */
  $\boldsymbol{Y} \leftarrow \boldsymbol{A}\boldsymbol{Y}_0$; /* $\boldsymbol{Y} \in \mathbb{F}_2^{n \times N}$ */
  $(\boldsymbol{X}, \boldsymbol{V}_i) \leftarrow BlockLanczos(\boldsymbol{Y}, \boldsymbol{A})$; /* $\boldsymbol{X}, \boldsymbol{V}_i \in \mathbb{F}_2^{n \times N}$ */
  $\boldsymbol{Z} \leftarrow (\boldsymbol{X} - \boldsymbol{Y}_0) | \, \boldsymbol{V}_i$; /* $\boldsymbol{Z} \in \mathbb{F}_2^{n \times 2N}$ */
  $\boldsymbol{U} \leftarrow Kernel(\boldsymbol{B}\boldsymbol{Z})$; /* $\boldsymbol{U} \in \mathbb{F}_2^{m \times k}$ where $k < 2N$ */
  $\boldsymbol{X}_0 \leftarrow \boldsymbol{Z}\boldsymbol{U}$; /* $\boldsymbol{X}_0 \in \mathbb{F}_2^{n \times k}$ */
end

Chapter 4

Extracting square roots in $\mathbb{Z}$

Following the matrix equation, a number of pairs (a, b) remain as compositions of the final perfect squares of x and y mentioned in the beginning. As for the rational side, computing the square root is just a matter of simple efforts since prime factorization of each $a + bm$ is well known, and so is their product:

$$y^2 = f'(m) \prod_{(a,b)\in U} a + bm \tag{4.1}$$

By dividing the total appearance of each $p \in F$ in y^2 by 2, the final product of these primes along with their exponents would easily results in y. The remaining task is now to deal with algebraic numbers $a + b\theta$ and their product. In fact, this task poses a question in the performance of the GNFS since careless consideration would make its computation as slow as the whole algorithm. Indeed, an intuitive way is to continuously taking polynomial multiplication of $a + b\theta$ modulo $f(\theta)$. Then one may try to factor this polynomial and map it to $\mathbb{Z}$ using the homomorphism ϕ used in Proposition 2.1.1. This surely gives an independent perfect square in $\mathbb{Z}$. However, since there might be a large number of pairs (a, b) appeared in the set U that forms the algebraic square, coefficients of the polynomial in θ representing this square may reach several millions of digits, making the computation practically impossible.

Alternatively, another strategy is to find the representations in $\mathbb{Z}[\theta]$ of all the primes in the algebraic factor base A and compute the square root in the same way as with rational square root. This method involves finding exact (q, s) such that $q + s\theta$ represent the primes $(p, r) \in A$. Once again, this does not seem to be feasible due to the large amount of prime ideals involved, and the computing technique that leads to their complex forms has yet been

optimized.

As suggested by J.M. Couveignes [7], an acceptable solution was introduced that benefits from the Chinese remainder theorem. In principle, the method struggles to find images of the square root in many different finite fields $\mathbb{F}_p$ for a number of primes p. After enough primes have been collected such that their product P is greater than the actual square root, the square can be computed indirectly in $\mathbb{Z}/P\mathbb{Z}$ with conventional method. With this description, the complexity of the square root algorithm depends on the time taken for multiplying $\#U$-bit numbers repeating $ln(\#U)$ number of times [14, pp. 100-101]. With U of large size such as 10 millions, this method is once again prohibitive. In addition, Couveignes's algorithm also requires that the original polynomial $f(x)$ in use is monic with odd degree. So far, this does not cause any trouble with the principle of the GNFS, but it would prevent further improvements to speed up the whole algorithm.

In what follows a significantly better idea will be represented whose creation named after P. Montgomery [17], with slight modification by Phong Nguyen [21]. Beside its great improvement over the performance, this method makes no assumption on the polynomial $f(x)$ except its irreducibility. In other words, $f(x)$ may also be non-monic with even degree. This gives rise to the improvement of the whole GNFS algorithm, as can be seen afterward.

4.1 A modification to the algebraic context

In the description of the mathematical background for the GNFS, it is clear that the reason for requiring the base polynomial $f(x)$ to be monic is to make sure that its root θ belongs to $\mathfrak{O}$, the ring of algebraic integers. This can be verified from the concept of $\mathfrak{O}$ in Definition 2.2.1. Given this condition, the ring $\mathbb{Z}[\theta]$ is sufficiently an order in $\mathfrak{O}$ and thus secures a number of operations on its scope, especially for Proposition 2.5.4.

On the other hand, if a non-monic polynomial $f(x)$ is introduced, this is not necessarily the case. In other words, those theorems appeared in Chapter 2 may no longer hold true, and hence a question on the probability of success of the algorithm should be raised. In order to cope with this change, instead of relying on $\mathbb{Z}[\theta]$, it is necessary a different ring may be used as a replacement. For this reason, θ should also be replaced with a new generation:

Proposition 4.1.1. *Let $f(x)$ be an irreducible polynomial of degree d with integer coefficients. Let θ be one of its complex roots, then there exists an algebraic integer $\hat{\theta}$ of the form $k_d\theta$ where k_d is the highest degree coefficient of $f(x)$.*

Proof. The proof can be facilitated with a monic irreducible polynomial $T(x) = k_d^{d-1} f(x/k_d)$. It is obvious that $\hat{\theta}$ is a root of $T(x)$, and hence $\hat{\theta} \in \mathfrak{O} \subset \mathbb{Q}(\hat{\theta}) = \mathbb{Q}(\theta)$ since $T(x)$ is monic irreducible with integer coefficients. □

Even though $\hat{\theta}$ satisfies that $\mathbb{Z}[\hat{\theta}]$ is a ring in $\mathfrak{O}$, there is no evidence that it is qualified for Proposition 2.5.3, i.e., having all the first degree prime ideals defined by (p, r) pairs. Note that for the reason described in Proposition 2.4.3, if a pair (a, b) satisfies that $p|N(a - b\theta)^*$, then $p \nmid b$. This is however not necessarily the case when $f(x)$ is non-monic. Indeed, considering the situation in which $p|b$ and $p|k_d$:

$$\begin{aligned} N(a - b\theta) &= b^d f(\frac{a}{b}) \\ &= b^d(k_d\frac{a^d}{b^d} + k_{d-1}\frac{a^{d-1}}{b^{d-1}} + \cdots + k_1\frac{a}{b} + k_0 \\ &= k_d a^d + k_{d-1}a^{d-1}b + \cdots + k_1 a b^{d-1} + k_0 b^d \\ &= p\frac{k_d}{p}a^d + p\frac{b}{p}M = pN \end{aligned} \tag{4.2}$$

Thus, even if $p|N(a-b\theta)$, there is no such r that $a \equiv -br \pmod{p}$, or that r is categorized as ∞ since $b \equiv -a/r \equiv 0 \pmod{p}$. In order to take (p, ∞) into considerations, a solution is to make use of a larger ring [14, pp. 88-89]that contains $\mathbb{Z}[\hat{\theta}]$, as shown below:

Proposition 4.1.2. *Let θ be a complex root of a polynomial with integer coefficients $f(x)$. Let $\beta_i = \sum_{j=0}^{d-i-1} k_{j+i+1}\theta^j$ where k_j are coefficients of $f(x)$. Let $A = \mathbb{Z} + \sum_{i=0}^{d-2} \mathbb{Z}\beta_i$, then A is an order in $\mathbb{Q}(\theta)$ satisfying $\mathbb{Z}[\hat{\theta}] \subset A = \mathbb{Z}[\theta] \cap \mathbb{Z}[\theta^{-1}]$.*

Due to this special construction, the order A clearly satisfies the requirement in Proposition 2.5.3, as the following result has shown:

Proposition 4.1.3. *Let A be an order of a field K defined in Proposition 4.1.2. Let $R(p)$ be the set of $r \in \mathbb{Z}/p\mathbb{Z}$ for which $f(r) \equiv 0 \pmod{p}$, together*

*Since $N(a - b\theta)$ has simpler form than $N(a + b\theta)$, it is more convenient to use b as $-b$, and still get the same norm.

with ∞ if $p|k_d$. Then the set of prime ideals $\mathfrak{p}$ of A is in bijective correspondence with the set of pairs (p, r) for $r \in R(p)$. Moreover, for each $a - b\theta$, the computation of $l_{\mathfrak{p}}$ in Proposition 2.5.4 can be modified to what follows:

$$l_{\mathfrak{p}}(a - b\theta) = \begin{cases} e_{p,r}(a - b\theta) & \text{if } r \neq \infty \\ e_{p,r}(a - b\theta) - v_p(k_d) & \text{if } r = \infty \end{cases}$$

where $v_p(x)$ stands for the p-adic valuation[†] of x and $e_{p,r}(a-b\theta) = v_p(b^d f(a/b))$.

Proof. For $r \neq \infty$, it is clear that every element of $\mathfrak{p}$ is divisible by p, and hence they all belong to the kernel of the mapping $\psi_{p,r} : \mathbb{Z}[\theta] \to \mathbb{Z}/p\mathbb{Z}$ that sends θ to r. Since $\mathfrak{p} \in A$, then $\mathfrak{p} = A \cap ker(\psi_{p,r})$ is indeed a first degree prime ideal of A. The same can be applied to $r = \infty$, except that $\psi_{p,\infty} : \mathbb{Z}[\theta^{-1}] \to \mathbb{Z}/p\mathbb{Z}$ is used that sends θ^{-1} to r^{-1}, i.e., 0. Thus, the first part of the proposition holds. Considering the computation of $l_{\mathfrak{p}}$, it is necessary to recall the norm formula, with respect to the ring $\mathbb{Z}[\hat{\theta}]$:

$$\begin{aligned} N(a - b\theta) &= N\left(a - b\frac{\hat{\theta}}{k_d}\right) \\ &= \left(\frac{b}{k_d}\right)^d T\left(\frac{ak_d}{b}\right) \\ &= \left(\frac{b}{k_d}\right)^d \left(\frac{a^d k_d^d}{b^d} + k_{d-1}\frac{a^{d-1}k_d^{d-1}}{b^{d-1}} + k_{d-2}k_d\frac{a^{d-1}k_d^{d-1}}{b^{d-1}+} \cdots + k_0 k_d^{d-1}\right) \\ &= a^d + k_{d-1}a^{d-1}\frac{b}{k_d} + k_{d-2}a^{d-2}\frac{b^2}{k_d} + \cdots + k_0\frac{b^d}{k_d} \\ &= \frac{k_d a^d + k_{d-1}a^{d-1}b + k_{d-2}a^{d-2}b^2 + \cdots + k_0 b^d}{k_d} \\ &= \frac{1}{k_d} b^d f\left(\frac{a}{b}\right) \end{aligned}$$

Since $l_{\mathfrak{p}}(a - b\theta) = v_p(N(a - b\theta))$, the proof is complete as one computes the p-adic valuation of both sides. □

In this case, if $f(x)$ of non-monic form is to be used throughout the GNFS, then it is crucial to add also a number of pairs (p, ∞) to the algebraic factor base A and recompute $l_{\mathfrak{p}}$ from $e_{p,\infty}$ before moving to solving the matrix equation. On the other hand, since all of the orders in use so far are neither

[†] $v_p(x)$ is simply the number of p as factors in x. This has the same meaning when using the concept of $\mathfrak{p}$-adic valuation of an ideal, which is also called the ramification index.

PID nor UFD, there are chances that even if the norm of $a - b\theta$ has been completely factorized, the ideal generating from it is divisible by other prime ideals not included in the prime factorization of $N(a - b\theta)$. In such cases, these unknown prime ideals are called *exceptional prime ideals*. As indirectly shown by [21, Theorem 1], all the prime ideals lying above a prime p dividing $[\mathfrak{O} : A]$[‡] would necessarily be treated as exceptional ideals.

Thus, after receiving a smooth algebraic integer $a-b\theta$, it is also crucial to find the valuation of each exceptional prime ideal $\mathfrak{p}$ at $\langle a - b\theta\rangle$, and add them as entries to the matrix equation. Note that by this definition, some "good" first degree prime ideals already in the algebraic factor base A may appeared as exceptional just because they lie above a prime p dividing $[\mathfrak{O} : A]$, and without careful consideration, some prime ideals might occur twice in the factorization of the ideal generated by an algebraic integer $a - b\theta$. While details of this process is beyond this context, a good illustration of dealing with exceptional primes is presented with the final example in Chapter 7.

4.2 Square root algorithm

Since most computations are lifted from algebraic integers to their corresponding principal ideals, the method described in this algorithm also turns the problem of factoring an algebraic integer to decomposing prime ideals of an ideal. Therefore, it is important to define some fractional concepts of an ideal I for all the prime ideals used in the factor bases, as what follows:

$$Numer(I) = \mathfrak{p}_0^{max(v_{\mathfrak{p}_0}(I),0)}\mathfrak{p}_1^{max(v_{\mathfrak{p}_1}(I),0)}\cdots\mathfrak{p}_k^{max(v_{\mathfrak{p}_k}(I),0)} = \prod_{v_{\mathfrak{p}_i}(I)>0}\mathfrak{p}_i^{v_{\mathfrak{p}_i}(I)}$$

$$Denom(I) = \mathfrak{p}_0^{max(-v_{\mathfrak{p}_0}(I),0)}\mathfrak{p}_1^{max(-v_{\mathfrak{p}_1}(I),0)}\cdots\mathfrak{p}_k^{max(-v_{\mathfrak{p}_k}(I),0)} = \prod_{v_{\mathfrak{p}_i}(I)<0}\mathfrak{p}_i^{-v_{\mathfrak{p}_i}(I)}$$

Of crucial importance, it is obvious that $I = Numer(I)/Denom(I)$ and $N(I) = N(Numer(I))/N(Denom(I))$. Let γ be the perfect square generated by compositing all algebraic integers $a - b\theta \in U$, the algorithm finds $\sqrt{\gamma}$ by factoring the corresponding ideal $\langle\gamma\rangle$ to get $\sqrt{\langle\gamma\rangle}$ and approximate it back to $\sqrt{\gamma}$. The algorithm involves the following steps, as summarized in [21, pp. 5-6]:

- Try to simplify the prime composition of $Numer(\langle\gamma\rangle)$ and $Denom(\langle\gamma\rangle)$ by deciding whether to put each ideal $\langle a - b\theta\rangle$ to the numerator (resp.

[‡] $[\mathfrak{O} : A]$ indicates the degree of $\mathfrak{O}$ as the vector space over A, i.e., the dimension of the vector space $\mathfrak{O}$ over A.

denominator) of $\langle\gamma\rangle$ so as to cancel some prime ideals appeared in denominator (resp. numerator).

- Let γ_0 be the simplified version of γ, start to further reduce γ_0 using an iterative approach until it is computable using the *Chinese remainder theorem*. In particular, at each step l, try to find $s_l \in \{-1, +1\}$ and $\delta_l \in \mathfrak{O}$ such that $\gamma_l = \gamma_{l-1}\delta_l^{-2s_l}$ is somehow less complicated than γ_{l-1}. Record each δ_l as the approximation at each step l so that they can be used in the later stage to form the square root.

- Let α be the final version of γ_L which is assumed to be simple enough so that it is possible to apply Cipolla's algorithm to find $\sqrt{\alpha}$. Thus, the square root of γ can be obtained as follows:

$$\sqrt{\gamma} = \sqrt{\alpha}\prod_{l=0}^{L}\delta_l^{s_l}$$

As briefly mentioned, there has been a slight improvement to the original algorithm by Montgomery. In fact, this change relates to the way how operations among ideals are performed, including addition, multiplication, etc. In principle, the best method for doing computation in $\mathfrak{O}$ is to find its basis as vectors in $\mathbb{Z}[\hat{\theta}]$ and perform all operations on these vectors. As suggested by Montgomery [17, p. 10], the power basis $\{1, \hat{\theta}, \hat{\theta}^2, \ldots, \hat{\theta}^{d-1}\}$ is used for this purpose. In this modification however, it is better to use an integral basis of $\mathfrak{O}$, that is, the set of d linearly independent vectors, under the form of polynomials in $\hat{\theta}$ with rational coefficients. In overall, the whole algorithm can be summarized as in Algorithm 4.2.1.

4.3 Computing the integral basis of $\mathfrak{O}$

The idea of constructing the integral basis of $\mathfrak{O}$ was based on the fact that there exists a method to enlarge an order finitely many times so that it finally becomes the order $\mathfrak{O}$. During each enlargement, a new integral basis is computed to facilitate the next iteration, and after the last step, the basis obtained is supposed to successfully spans $\mathfrak{O}$. Before diving into this method, it is necessary to recall some characteristics of $\mathfrak{O}$ as well as rings closed to $\mathfrak{O}$ [5, p. 303]:

Proposition 4.3.1. *Let $\mathfrak{O}$ be the ring of algebraic integers in the field K for some $\theta \in \mathfrak{O}$. Then $R \subset \mathfrak{O}$ for any other order $R \subset \mathbb{Q}(\theta)$. In this case $\mathfrak{O}$ is called the maximal order of K.*

Algorithm 4.2.1: SquareRoot

Input: A set U of pair (a, b).
Output: The square root of $\gamma = \prod_{(a,b)\in U}(a - b\theta)$.

```
1  begin
2     ComputeIntegralBasis(f(x)) ;                    /* See §4.3 */
3     γ_0 ← 1;
4     foreach (a, b) ∈ U do
         /* Use greedy algorithm to assess this                  */
5        if Numer(⟨γ_0⟩) ">" Denom(⟨γ_0⟩) then
6           γ_0 ← γ_0(a − bθ)^{-1};
7        else
8           γ_0 ← γ_0(a − bθ);
9        endif
10    endfch
11    Approx ← 1;
12    while γ_l is complex do          /* For this loop, see §4.4 */
13       if Numer(⟨γ_l⟩) ">" Denom(⟨γ_l⟩) then
14          s_l ← 1 ;                 /* Simplifying Numer(⟨γ_l⟩) */
15       else
16          s_l ← −1 ;                /* Simplifying Denom(⟨γ_l⟩) */
17       endif
18       δ_l ← SelectGoodDelta(γ_l, s_l);
19       γ_{l+1} ← γ_l δ_l^{−2s_l};
20       Approx ← Approx · δ_l^{s_l};
21       l ← l + 1;
22    endw
23    α ← γ_l;
24    Sqrt ← FindSquareRoot(α);
25    Sqrt ← Sqrt · Approx;
26 end
```

Definition 4.3.1. *Let $\mathcal{O}$ be an order in a number field K. Let p be a prime number in $\mathbb{Z}$. If $p \nmid [\mathfrak{O} : \mathcal{O}]$, then $\mathcal{O}$ is called p-maximal. Furthermore, the p-radical I_p of $\mathcal{O}$ can be uniquely defined as:*

$$I_p = \{x \in \mathcal{O} | \exists m \geq 1 \text{ such that } x^m \in p\mathcal{O}\}$$

The heart of the Round 2 algorithm, as originated by Pohst and Zassenhaus, is followed by an important observation, as described below:

Proposition 4.3.2. *Let $\mathcal{O}$ be an order in a number field K. Let p be prime number in $\mathbb{Z}$. Let $\mathcal{O}' = \{x \in K | xI_p \subset I_p\}$. If $\mathcal{O}$ is p-maximal, then $\mathcal{O} = \mathcal{O}'$, otherwise, $\mathcal{O} \subsetneq \mathcal{O}'$ and $p | [\mathcal{O}' : \mathcal{O}] | p^m$ for some $m \in \mathbb{Z}^+$.*

While the proof is given along the lines of [5, pp. 304-305], the idea is clear on how the algorithm proceeds on enlarging some smaller ring to achieve $\mathfrak{O}$. Indeed, note that the ring $\mathcal{O}'$ in the above proposition is also an order, and the method may start with the ring $\mathbb{Z}[\hat{\theta}]$ since its integral basis is well known, as represented by the power basis in $\hat{\theta}$ up to the exponent $d-1$. By iteratively enlarging $\mathbb{Z}[\hat{\theta}]$, the algorithm tries to alter $\mathbb{Z}[\hat{\theta}]$ so that it becomes p-maximal for each prime number in $\mathbb{Z}$. After finitely many steps, it can be sure that the ring $\mathcal{O}$ derived from $\mathbb{Z}[\hat{\theta}]$ is p-maximal for every prime p, and thus $\mathcal{O} = \mathfrak{O}$, along with its integral basis growing at every iteration.

In fact, the original ring $\mathbb{Z}[\hat{\theta}]$ is already p-maximal for most primes p. Therefore, it is only necessary to find the remaining primes, and try to enlarge this ring over them to reach the maximal order $\mathfrak{O}$. For this reason, the primes p which $\mathbb{Z}[\hat{\theta}]$ is yet p-maximal can be obtained from the following results [5, Theorem 4.4.4]:

Proposition 4.3.3. *Let $\hat{\theta}$ be a complex root of a monic irreducible polynomial $T(x)$ with integer coefficients. Let $f = [\mathfrak{O} : \mathbb{Z}[\hat{\theta}]]$, then*

$$disc(T) = disc(\mathbb{Q}(\hat{\theta}))f^2 \tag{4.3}$$

where $disc(T)$[§] and $disc(\mathbb{Q}(\hat{\theta}))$ stand for the discriminants of $T(x)$ and $\mathbb{Q}(\hat{\theta})$, respectively.

Thus, if the value of $disc(T)$ is known, it is then simple to find the set of primes p dividing $[\mathfrak{O} : \mathbb{Z}[\hat{\theta}]]$ by seeking for primes p such that $p^2 | disc(T)$. Note that however the converse of this is not true, that is, if $p^2 | disc(T)$, it

[§]The discriminant of $T(x)$ is defined as $(-1)^{d(d-1)/2} R(T, T')/k_d$, where $T'(x)$ is the derivative of $T(x)$, as well as $R(A, B)$ is referred as the resultant of A and B. See [5, p. 119] for details.

is not always true that $p|[\mathfrak{O} : \mathbb{Z}[\hat{\theta}]]$. In such cases, the enlargement process can still be tried over these primes with no result or the Dedekind test might be performed before any enlargement to check for the divisibility, as can be seen in [5, pp. 305-308]. Nevertheless, due to the fact that the list of primes p above is relatively small, whether or not the Dedekind's test is used, the performance of this strategy will not be remarkably affected.

From the other perspective, it is also important to define a mean of operations among the algebraic structures within $\mathfrak{O}$ which is indispensable for this method to operate. Indeed, note that since every well defined structure within our consideration has a unique basis, the idea is to somehow use them to identify each structure, represented under some computable forms. This leads to the use of matrices of Hermite Normal Form (HNF):

Definition 4.3.2. *Let $\boldsymbol{M}$ be a matrix in $\mathcal{K}^{m\times n}$. $\boldsymbol{M}$ is said to be of HNF form if there exists an $r \leq n$ such that*

1. *For $r+1 \leq j \leq n$, $m_{j-r,j} \geq 1$, $m_{i,j} = 0$ if $i > j - r$ and $0 \leq m_{k-r,j} < m_{k-r,k}$ if $k < j$.*
2. *For $1 \leq j \leq r$ and $1 \leq i \leq m$, $m_{i,j} = 0$.*

Visually speaking, one may think of an HNF matrix as an upper triangular matrix if all of its zero columns and rows are removed, or that this is always true if the matrix is square. This remaining part of the HNF matrix gives rise to the representation of the vectors that span each algebraic structure. In this context, it is sufficient to use the concept of *module*[¶] in the number field K, as it is common to all structures used in this method. The following result confirms the use of HNF matrices on representing modules:

Proposition 4.3.4. *Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be n $\mathbb{Z}$-linearly independent elements of the number field K that spans a module R. Let M be a module in K, then there exists a unique basis $\omega_1, \omega_2, \ldots, \omega_n$ of M of the form:*

$$\omega_i = \frac{1}{d}\left(\sum_{i=1}^{n} w_{i,j}\alpha_i\right)$$

where d is the smallest positive integer such that $d\omega_i \in \mathbb{Z}[\hat{\theta}]$. Furthermore, the matrix $\boldsymbol{W}$ formed by $w_{i,j}$ is in HNF form, and hence together with d, they uniquely identify the module M.

[¶]A module M in K is a finitely generated $\mathbb{Z}-$*submodule* of K with maximal rank, i.e., $rank(M) = deg(K)$.

In this case, since $R = \mathbb{Z}[\hat{\theta}]$ and $\alpha_i = \hat{\theta}^{i-1}$, each ω_i is a polynomial in $\hat{\theta}$ with coefficients represented by the i-th column of the matrix $\boldsymbol{W}$. With this support, one can start computing for each prime p the p-maximal order $\mathcal{O}'$ containing $\mathbb{Z}[\hat{\theta}]$. To compute $\mathcal{O}'$ as in Proposition 4.3.2, it is required that the p-radical of $\mathbb{Z}[\hat{\theta}]$ be found, along with a pair $(\boldsymbol{W}, d)$ representing it. The radical can be found by an observation as follows [5, Lemma 6.1.6]:

Proposition 4.3.5. *Let $\mathcal{O}$ be an order in a number field K, let $R = \mathcal{O}/p\mathcal{O}$, then if $p^j \geq n$ for some integer j and the degree n of K, then the radical of R is the kernel of the map $x \to x^{p^j}$.*

Assume that the ring $\mathcal{O}$ has basis $\omega_1, \omega_2, \ldots, \omega_n$, then $\overline{\omega}_1, \overline{\omega}_2, \ldots, \overline{\omega}_n$[‖] will apparently be the basis of R. We then attempt to find $\overline{a}_{i,k}$ such that

$$\overline{\beta_k} = \overline{\omega}_k^{p^j} = \sum_{i=1}^{n} \overline{a}_{i,k}\overline{\omega}_i \tag{4.4}$$

Since $\overline{\omega}_k$ are linearly independent, then so are $\overline{\beta}_k$. By forming the matrix $\overline{\boldsymbol{A}}$ containing $\overline{a}_{i,k}$ and change it into HNF form, the basis for the mapping above is thus revealed. Then, finding the radical $\overline{I}_p$ of R is only a matter of computing the kernel of the matrix $\overline{\boldsymbol{A}}$, appeared as a matrix $\boldsymbol{D}$ representing a basis of j elements $\overline{\sigma}_1, \ldots, \overline{\sigma}_j$. Then, note that it was agreed from the beginning that every module is represented in its HNF form, this must be applied also to I_p. That being said, $\boldsymbol{D}$ must be converted into its HNF form, and in order to do so, it must be supplemented so that it would have more columns than rows before the conversion process takes place. This supplement can be done by joining the basis of $\overline{I}_p$ with that of $p\mathcal{O}$, i.e., $\overline{\sigma}_1, \ldots, \overline{\sigma}_j, p\omega_1, \ldots, p\omega_n$.

With the p-radical I_p of $\mathcal{O}$, or more precisely, the integral basis $\psi_1, \ldots, \psi_n$ of I_p, the basis of $\mathcal{O}'$ can be easily computed, according to the following result:

Proposition 4.3.6. *Let $\mathcal{O}$ be an order in the number field K. Let $\mathcal{O}'$ be the order as defined in Proposition 4.3.2. Let U be the kernel of the mapping $\alpha \to (\overline{\beta} \to \overline{\alpha\beta})$ from $\mathcal{O}$ to $End(I_p/pI_p)$, the set of endomorphisms in I_p/pI_p. Then $\mathcal{O}' = \frac{1}{p}U$.*

Since I_p is known from the previous step, the basis for I_p/pI_p can be trivially produced as the image of the basis of I_p in I_p/pI_p. Similar to the

[‖] $\overline{\omega}_i$ is simply $\omega_i \pmod{p}$. The same notation may be applied to other categories such as matrices, coefficients, modules, etc.

previous one, this step begins by computing the basis for the mapping described above, represented by a matrix $\boldsymbol{B}$. This matrix is of size $n^2 \times n$, where its coefficients $\boldsymbol{B}_{(i,j),k}$satisfy that:

$$\omega_k \psi_j \equiv \sum_{i=1}^{n} \boldsymbol{B}_{(i,j),k} \psi_i \pmod{p} \tag{4.5}$$

By finding the kernel of $\boldsymbol{B}$, the basis of $\overline{U}$ is thus revealed. The integral basis of U can be obtained in the same way by supplementing it with $p\mathcal{O}$. In practice, a reasonable version of the Round 2 algorithm was successfully included in a *state-of-the-art* implementation of the NFS community, namely the GGNFS. Nevertheless, this implementation can be summarized as in Algorithm 4.3.1.

Algorithm 4.3.1: Round2Zassenhaus

Input: a monic irreducible polynomial $T(x)$ of degree d and root $\hat{\theta}$
Output: the integral basis of $\mathfrak{O}$ in $\mathbb{Q}(\hat{\theta})$

begin
  ▷ Compute the discriminant d_T of $T(x)$;
  ▷ Compute a set P of primes p where $p^2 | d_T$;
  $\boldsymbol{W}_{\mathfrak{O}} \leftarrow \boldsymbol{I}_d$; /* Set $\mathfrak{O} = \mathbb{Z}[\hat{\theta}]$ with HNF basis $W_{\mathfrak{O}}$ */
  foreach $p \in P$ **do**
    $\boldsymbol{W}_{\mathcal{O}'} \leftarrow \boldsymbol{I}_d$; /* Set $\mathcal{O}' = \mathbb{Z}[\hat{\theta}]$ with HNF basis $W_{\mathcal{O}'}$ */
    ▷ Compute least $j \in \mathbb{Z}^+$ such that $p^j > d$;
    repeat
      $\boldsymbol{W}_{\mathcal{O}} \leftarrow \boldsymbol{W}_{\mathcal{O}'}$; /* Set $\mathcal{O}$ to previous enlargement */
      ▷ Compute $\overline{\boldsymbol{A}}$ of coefficients $\overline{a}_{i,k}$ as in (4.4);
      $\boldsymbol{D} \leftarrow Kernel(\overline{\boldsymbol{A}})$;
      $\boldsymbol{D} \leftarrow (\boldsymbol{D} | p\boldsymbol{W}_{\mathcal{O}})$; /* Supplement $\overline{I_p}$ with $p\mathcal{O}$ */
      $\boldsymbol{D} \leftarrow HNF(\boldsymbol{D}) \pmod{p}$; /* Get basis ψ_i of I_p/pI_p */
      ▷ Compute $\boldsymbol{B}$ of coefficients $\boldsymbol{B}_{(i,j),k}$ as in (4.5);
      $\boldsymbol{B} \leftarrow Kernel(\boldsymbol{B}) \pmod{p}$; /* Get $\overline{U}$ */
      $\boldsymbol{B} \leftarrow (\boldsymbol{B} | p\boldsymbol{W}_{\mathcal{O}})$; /* Supplement $\overline{U}$ with $p\mathcal{O}$ */
      $\boldsymbol{B} \leftarrow HNF(\boldsymbol{B})$; /* Get U */
      $\boldsymbol{W}_{\mathcal{O}'} \leftarrow \boldsymbol{B}/p$; /* Get $\mathcal{O}'$ */
    until $\boldsymbol{W}_{\mathcal{O}} = \boldsymbol{W}_{\mathcal{O}'}$ /* Until $\mathcal{O}$ is p-maximal */;
    $\boldsymbol{W}_{\mathfrak{O}} \leftarrow \boldsymbol{W}_{\mathfrak{O}} + \boldsymbol{W}_{\mathcal{O}}$ /* Enlarge $\mathfrak{O}$ by p-maximal $\mathcal{O}$ */
  endfch
end

4.4 Selecting good approximations for γ

As mentioned in the main algorithm, at each step of the approximation, a value δ_l must be found so that once its square is multiplied with γ_l, the result would be "simpler". In term of γ_l itself, this idea is not clear on how the simplicity can be recognized and assessed. However, such measurement can be applied to the ideal it generates, i.e., $\langle\gamma_l\rangle$. Indeed, considering the notions of $Numer(\langle\gamma_l\rangle)$ and $Denom(\langle\gamma_l\rangle)$, this measurement process turns into finding δ_l such that the numerator and denominator of the ideal generated by $\gamma_l\delta_l^{-2s_l}$ would have simpler prime decompositions. Therefore, at each step l the algorithm needs to cancel out some prime factors in either the numerator or denominator of $\langle\gamma_l\rangle$ by multiplying them to the opposite part of the fractions. These extra multiplications are then recorded as approximation to the original ideal, i.e., the actual perfect square in $\mathbb{Z}[\theta]$.

Without loss of generality it can be assumed that at step l the numerator $H_N = Numer(\langle\gamma_l\rangle)$ is chosen to be simplified as it contains more prime factors than $Denom(\langle\gamma_l\rangle)$. With some predefined limit LLL_{max}, the method would try to find a largest possible ideal I_l with I_l^2 dividing H_N and $N(I_l)$ very close to LLL_{max}. Since I_l consists of some prime factors in H_N, multiplying it with the denominator will result in a simpler numerator. The problem is that if the manipulation is done by ideal computation, an obstacle arises when trying to find γ_{l+1} generating it and record the approximation.

For this reason, instead of updating $\langle\gamma_l\rangle$ to $\langle\gamma_{l+1}\rangle$ and find γ_{l+1}, it is safer to update γ_l to γ_{l+1} and find its corresponding ideal $\langle\gamma_{l+1}\rangle$. This is where δ_l comes in as a way of representing I_l. Note however that, even if δ_l is chosen from I_l, there is no guarantee that δ_l would only contain prime factors in I_l, and hence it may produces more changes than that expected by I_l. Due to this obstruction, $\delta_l \in I_l$ must be computed so that it is as small as possible, i.e., δ_l such that $N(\langle\delta_l\rangle / I_l)$ is of minimum value, making $\langle\delta_l\rangle$ as close to I_l as possible. This is equal to the problem of finding short elements in I_l and hence triggers the use of the *lattice reduction* algorithm, since an ideal is also a lattice**.

At the first stage, once an ideal I_l is chosen for conditions described from the beginning, it is important that its HNF identification matrix is formed so that its integral basis is known. The lattice reduction algorithm must

**See the next section for clear concepts of a lattice and how the lattice reduction technique operates.

then be invoked to reduced this basis to a basis where elements are almost pairwise orthogonal. Let this basis of I_l be $\mu_1, \ldots, \mu_n$ and $v_{i,j}$ such that

$$\mu_i = \sum_{j=1}^{n} v_{i,j}\omega_j \tag{4.6}$$

Denoted by $\Delta(\mathbb{K})$ the discriminant of the field $\mathbb{K} = \mathbb{Q}(\hat{\theta})$ and $\sigma_k(\gamma_l)$ the k-th conjugate (embedding) of γ_l as defined in Proposition 2.4.1. Assume that these factors are well known from the previous approximation, some other local constants need to be computed. These in turn are:

$$c^d = \frac{LLL_{max}}{N(I_l)}\sqrt{\frac{|N_{\mathbb{K}}(\gamma_l)|^{s_l}}{|\Delta(\mathbb{K})|}}$$
$$\lambda_k = \frac{c}{|\sigma_k(\gamma_l)|^{s_l/2}}$$

Given these values and the composition of the basis in (4.6), for each of the μ_i, one can construct a unique column vector of the form

$$\Omega\mu_i = \langle v_{i,1}, v_{i,2}, \ldots, v_{i,n}, \lambda_1\sigma_1(\mu_i), \lambda_2\sigma_2(\mu_i), \ldots, \lambda_n\sigma_n(\mu_i)\rangle$$

The n such column vectors result in a $2n \times n$ matrix $\boldsymbol{F}$ satisfying the following result:

Proposition 4.4.1. *Let $\boldsymbol{F}$ be the $2n \times n$ matrix formed by joining together the transformations $\Omega\mu_i$ for $i = \overline{1,n}$. Then this matrix satisfies:*

1. *The determinant of the first n rows of $\boldsymbol{F}$ is in absolute value equal to $|N(I_l)|$.*
2. *The determinant of the last n rows of $\boldsymbol{F}$ is in absolute value equal to LLL_{max}.*

The second lattice reducing process can then be applied to reduce the matrix $\boldsymbol{F}$ so as to get shorter vectors in the first d columns of $\boldsymbol{F}$. As the final stage, δ_l can be chosen among the vectors representing these columns so that its magnitude $|\delta_l|$ is of minimum value, which is possibly the best value that can be chosen. In general, this shortest vector is normally located in the first column of the matrix due to the nature of the lattice reduction technique. Despite this effort, to make sure that the algorithm works correctly, it is still important to check whether δ_l contains primes not in I_p. If this is the case, let H be the ideal consisting of primes in $\langle\delta_l\rangle$ but I_p, and so the ideal

representing γ_{l+1} must be changed by H in the same manner. In other words, let G_l be the ideal generated from γ_l, then

$$\gamma_{l+1} = \gamma_l \delta^{-2s_l} \Longleftrightarrow G_{l+1} = \left(\frac{I_l}{H}\right)^{-2s_l} G_l$$

On termination of this phase, a condition can be specified which requires that the parameter $\mathcal{C}(G_l) = N(Numer(G_l))N(Denom(G_l))$ as well as its two factors are close to 1. This makes sure that γ_l will be extremely simple, and hence allows the Chinese remainder theorem to easily compute its coefficients. In practice, as shown in [21, p. 11], the number of iterations required before achieving these conditions is at most $2\lceil log_2\mathcal{C}(\sqrt{\langle\gamma\rangle})\rceil$ which can be easily computed from the beginning. To give an example implementation, Algorithm 4.4.1 explains the tasks required for completing the approximating phase.

Remark. As suggested by Nguyen in [21, pp. 11,13-17], the value of LLL_{max} can be determined from the beginning so that it is much larger than a constant C, as described in the following proposition:

Proposition 4.4.2. *Let I_l and δ_l be the corresponding components at any step l of the approximation phase, then there exist a constant C depending only on the underlying number field $\mathbb{K}$ such that:*

$$|N_{\mathbb{K}}(\delta_l)| \leq C \times N(I_l) \tag{4.7}$$

In fact, the value of the constant C can be computed as follows:

$$\begin{aligned}
C_1 &= max_{1\leq j\leq n}\sqrt{\sum_{i=1}^{n}|\sigma_j(\omega_i)|^2} \\
C_2 &= 2^{\frac{n(n-1)}{4}} n^n 2^{n+1} \\
C_3 &= \sqrt{1+nC_1} \\
C &= 2^{\frac{n(n-1)}{2}} max(1, C_1^n) C_2 C_3^n
\end{aligned}$$

where n is the degree of $\mathbb{K}$, or more deterministically, the degree of the base polynomial $f(x)$ chosen at the beginning of the sieving algorithm. Also, considering this square root approximation, it is better to use addition with logarithm than multiplication with power as illustrated in the pseudocode. This is to prevent any problems with overflows, memory consumption, and performance constraints required by the computations. In terms of time

Algorithm 4.4.1: SelectApproximation

Input: Set U where $\gamma = \prod_{(a,b)\in U}(a - b\theta)$, HNF basis $\boldsymbol{W}_{\mathfrak{O}}$
Output: α simplified from γ

begin
  $G_0 \leftarrow \langle\gamma\rangle$;
  $l \leftarrow 0$;
  while $\mathcal{C}(G_l) > 1$ ***or*** $N(Numer(G_l)) > 1$ ***or*** $N(Denom(G_l)) > 1$ **do**
    if $N(Numer(G_l) > N(Denom(G_l)$ **then**
      $s_l \leftarrow 1$;
      $\boldsymbol{M} \leftarrow Numer(G_l)$;
    else
      $s_l \leftarrow -1$;
      $\boldsymbol{M} \leftarrow Denom(G_l)$;
    endif
    $c \leftarrow \sqrt[n]{\frac{LLL_{max}}{N(I_l)}\sqrt{\frac{|N_{\mathbb{K}}(\gamma_l)|^{s_l}}{|\Delta(\mathbb{K})|}}}$;
    for $k \leftarrow 1$ **to** n **do**
      $\lambda_k \leftarrow \frac{c}{|\sigma_k(\gamma_l)|^{s_l/2}}$;
    endfor
    $I_l \leftarrow \mathbb{Z}[\hat{\theta}]$; `/* Use` I_n `for` W_{I_l} `*/`
    while $N(I_l) < LLL_{max}$ **do** `/* Select large` I_l `*/`
      ▷ Select $\mathfrak{p}^{2k}$ in $Numer(G_l)$;
      $I_l \leftarrow I_l \cdot \mathfrak{p}^{2k}$;
    endw
    $\{\mu_i\} \leftarrow LLLReduce(I_l)$; `/* See §4.5 */`
    ▷ Compute $\boldsymbol{F}$ as in Proposition 4.4.1;
    $\boldsymbol{F} \leftarrow LLLReduce(\boldsymbol{F})$; `/* See §4.5 */`
    ▷ Let $\boldsymbol{F}$ be its first d rows;
    $\delta_l \leftarrow \infty$;
    for $i \leftarrow 1$ **to** n **do** `/* Select smallest` δ_l `*/`
      $\delta_l \leftarrow Min(\delta_l, \boldsymbol{F}_i)$;
    endfor
    $H \leftarrow \frac{\langle\delta_l\rangle}{I_l}$;
    $G_{l+1} \leftarrow \left(\frac{I_l}{H}\right)^{-2s_l} G_l$;
    $\gamma_{l+1} \leftarrow \gamma_l \delta_l^{-2s_l}$;
    $l \leftarrow l + 1$;
  endw
  $\alpha \leftarrow \gamma_l$;
end

complexity, it is easy to see that this step does not take much time since the LLL reduction is applied only for matrices of trivial size $2n \times n$. In the worst case, the number of steps required before those conditions with $\mathcal{C}(G_l)$, $N(Numer(G_l))$, and $N(Denom(G_l))$ are matched may reach $\mathcal{O}(\#U)$ at maximum, and even when running within one computer, this is still acceptable.

4.5 Lattice reduction with the LLL algorithm

Named after A.K. Lenstra, H.W. Lenstra, and L. Lovasz, the LLL reduction algorithm was introduced in 1982 as a breakthrough for computing lattice reduced bases. As an application to the square root approximation process above, this algorithm helps in finding the shortest possible vector that closely acts as a generator for the whole lattice, with minor difference. To briefly sketch this idea, it is worth to recall the core concept of lattice, as can be seen in the following definitions [5, pp.79-81]:

Definition 4.5.1. *Let V be a vector space over some field K. Let q be a mapping from V to K such that for all $\lambda \in K$ and $x, x', y \in V$:*

1. $q(\lambda \cdot x) = \lambda^2 q(x)$
2. *Let* $b(x, y) = \frac{1}{2}(q(x+y)-q(x)-q(y))$, *then* $b(x+x', y) = b(x, y)+b(x', y)$ *and* $b(\lambda \cdot x, y) = \lambda b(x, y)$

The map q is called a quadratic form.

Definition 4.5.2. *A lattice L is a free $\mathbb{Z}$-module of finite rank together with a quadratic form q on $L \otimes \mathbb{R}$ such that $q(x) > 0$ $\forall x \in L$*

In order to geometrically represent a lattice (L, q), it is sufficient to consider it as a discrete subgroup of rank n of the Euclidean vector space $L \otimes R$. In other words, let $\{\boldsymbol{b}_i\}$ be a $\mathbb{Z}$-basis of L, then

$$L = \mathbb{Z}\boldsymbol{b}_1 + \mathbb{Z}\boldsymbol{b}_2 + \cdots + \mathbb{Z}\boldsymbol{b}_n$$

Given the mapping q such that $b(\boldsymbol{b}_i, \boldsymbol{b}_j) = \boldsymbol{b}_i \cdot \boldsymbol{b}_j$, a matrix $\boldsymbol{Q}$ can be formed with $\boldsymbol{Q}_{i,j} = b(\boldsymbol{b}_i, \boldsymbol{b}_j)$ such that for each $\boldsymbol{x} \in L$, an equivalence can be obtained as follows:

$$\boldsymbol{x} = \sum_{i=1}^{n} x_i \boldsymbol{b}_i \Longleftrightarrow q(\boldsymbol{x}) = \sum_{1 \le i,j \le n} \boldsymbol{Q}_{i,j} x_i x_j$$

While the matrix $\boldsymbol{Q}$ uniquely identifies the quadratic form q with respect to the given basis $\{\boldsymbol{b}_i\}$ of L, it is also called the *Gram matrix* of the $\{\boldsymbol{b}_i\}$

where $\sqrt{det(\boldsymbol{Q})}$ is called the determinant of L, denoted as $d(L)$. The use of the matrix $\boldsymbol{Q}$, though very important when dealing with floating-point errors in computations of the LLL algorithm [22, pp.11-12], can be waived out in this case since the two cases of its applications described previously can be relaxed with integer operations, i.e., the basis $\{\boldsymbol{b}_i\}$ can be rounded up so that it becomes integral.

As a starting point, notice that from any given basis $\{\boldsymbol{b}_i\}$ of a lattice (L, q), it is possible to construct an orthogonal basis $\{\boldsymbol{b}_i^*\}$ by inductively using the following Gram-Schmidt orthogonalization process:

Proposition 4.5.1. *Let $\{\boldsymbol{b}_i\}$ be a basis of a lattice L of rank n, define by induction $\{\boldsymbol{b}_i^*\}$ such that:*

$$\boldsymbol{b}_i^* = \boldsymbol{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\boldsymbol{b}_j^* \quad (1 \leq i \leq n), \text{ where } \quad \mu_{i,j} = \frac{\boldsymbol{b}_i \cdot \boldsymbol{b}_j^*}{\boldsymbol{b}_j^* \cdot \boldsymbol{b}_j^*} \quad (1 \leq j \leq i \leq n)$$

The set $\{\boldsymbol{b}_i^*\}$ then forms an orthogonal basis of L. In addition, if $d(L)$ is the determinant of L, then $d(L)^2 = \prod_{1 \leq i \leq n} \|\boldsymbol{b}_i^*\|^2$

Proof. The proof of orthogonality of $\{\boldsymbol{b}_i^*\}$ can be done using mathematical induction. The base case starts with $i = 2$ where $\boldsymbol{b}_1^* = \boldsymbol{b}_1$, hence:

$$\boldsymbol{b}_2^* \cdot \boldsymbol{b}_1^* = \left(\boldsymbol{b}_2 - \frac{\boldsymbol{b}_2 \cdot \boldsymbol{b}_1^*}{\boldsymbol{b}_1^* \cdot \boldsymbol{b}_1^*}\boldsymbol{b}_1^*\right)\boldsymbol{b}_1^* = \boldsymbol{b}_2 \cdot \boldsymbol{b}_1^* - \frac{\boldsymbol{b}_2 \cdot \boldsymbol{b}_1^*}{\boldsymbol{b}_1^* \cdot \boldsymbol{b}_1^*}\boldsymbol{b}_1^* \cdot \boldsymbol{b}_1^* = 0$$

Similarly, assume that the set $\{\boldsymbol{b}_k^*\}$ satisfies the proposition, the case $k+1$ can be proceeded as follows:

$$\boldsymbol{b}_{k+1}^* \cdot \boldsymbol{b}_i^* = \left(\boldsymbol{b}_{k+1} - \sum_{j=1}^{k} \mu_{k+1,j}\boldsymbol{b}_j^*\right)\boldsymbol{b}_i^* = \boldsymbol{b}_{k+1} \cdot \boldsymbol{b}_i^* - \frac{\boldsymbol{b}_{k+1} \cdot \boldsymbol{b}_i^*}{\boldsymbol{b}_i^* \cdot \boldsymbol{b}_i^*}\boldsymbol{b}_i^* \cdot \boldsymbol{b}_i^* = 0$$

This completely proved the first part of the proposition. Also, since $\{\boldsymbol{b}_i^*\}$ is a basis of L, the matrix $\boldsymbol{Q}$ created from it satisfies $d(L)^2 = det(\boldsymbol{Q})$. Due to the orthogonality of $\{\boldsymbol{b}_i^*\}$, only the diagonal of $\boldsymbol{Q}$ contains non-zero elements, hence the second part of the proposition follows. □

The idea of lattice reduction is to find a basis with vectors of shortest magnitude, which also implies that they are nearly orthogonal. In LLL-algorithm, a distinct criteria for bases being reduced, namely LLL-reduction, is what follows.

Definition 4.5.3. *Using the notations in Proposition 4.5.1, a basis* $\{\boldsymbol{b}_i\}$ *is called LLL-reduced if it satisfies that*

$$|\mu_{i,j}| \leq \frac{1}{2} \quad \text{for} \quad 1 \leq j < i \leq n$$
$$|\boldsymbol{b}_i^* + \mu_{i,i-1}\boldsymbol{b}_{i-1}^*|^2 \geq \frac{3}{4}|\boldsymbol{b}_{i-1}^*|^2 \quad \text{for} \quad 1 < i \leq n$$

The latter condition can also be expressed as:

$$|\boldsymbol{b}_i^*|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right)|\boldsymbol{b}_{i-1}^*|^2$$

Since $\{\boldsymbol{b}_i^*\}$ was chosen inductively, the same type of algorithm can be applied to find $\{\boldsymbol{b}_i\}$ satisfying the above conditions. Indeed, assume that in the case $k = 2$ (since it is meaningless with $k = 1$), the vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{k-1}$ are already reduced. The problem is now to reduce $\boldsymbol{b}_k$ while still retaining the previous vectors reduced. Starting with the first condition, i.e., $|\mu_{k,j}| \leq 1/2$ for $j < k$, one can assume that there exists l such that $|\mu_{k,j}| \leq 1/2$ for $l \leq j \leq k$ (the worst case would be $l = k$). Then, the basis must be modified such that l can be decreased by 1, and iteratively until $l = 0$. Note that according to the proof of Proposition 4.5.1 above, it is easy to check that if $l < j$, then $\boldsymbol{b}_l\boldsymbol{b}_j^* = 0$. Thus, if one chooses $q = \lfloor \mu_{k,l} \rceil$ and replaces $\boldsymbol{b}_k$ by $\boldsymbol{b}_k - q\boldsymbol{b}_l$, it follows that $\mu_{k,j}$ remains unchanged for $j > l$ since

$$\mu_{k,j} = \frac{\boldsymbol{b}_k \cdot \boldsymbol{b}_j^*}{\boldsymbol{b}_j^* \cdot \boldsymbol{b}_j^*} - q\frac{\boldsymbol{b}_l \cdot \boldsymbol{b}_j^*}{\boldsymbol{b}_j^* \cdot \boldsymbol{b}_j^*} = \frac{\boldsymbol{b}_k \cdot \boldsymbol{b}_j^*}{\boldsymbol{b}_j^* \cdot \boldsymbol{b}_j^*}$$

Meanwhile, it is also straightforward to verify that $\mu_{l,l} = 1$ for every l by the construction of $\{\boldsymbol{b}_l^*\}$:

$$\boldsymbol{b}_l^* \cdot \boldsymbol{b}_l^* = \left(\boldsymbol{b}_l - \sum_{j=1}^{l-1} \mu_{l,j}\boldsymbol{b}_j^*\right)\boldsymbol{b}_l^* = \boldsymbol{b}_l \cdot \boldsymbol{b}_l^*$$

Thus, replacing $\boldsymbol{b}_k$ by $\boldsymbol{b}_k - q\boldsymbol{b}_l$ also implies $\mu_{k,l}$ to be replaced by $\mu_{k,l} - q$ and hence it is less than $1/2$ in absolute value by the selection of q. Therefore, l can be decreased by 1 as $\mu_{k,j} \leq 1/2$ for $l - 1 < j < k$. Repeating this process until $l = 0$ will make sure that the case k satisfies the first condition.

Considering the second condition, namely *Lovász condition*, if $|\boldsymbol{b}_i^*|^2 < \left(3/4 - \mu_{i,i-1}^2\right)|\boldsymbol{b}_{i-1}^*|^2$ then we can interchange $\boldsymbol{b}_k$ and $\boldsymbol{b}_{k-1}$ and start working with the first condition in the lower case $k - 1$. Initially speaking, this idea

seems to continue forever, but as a proof in [5, pp. 88-89], it indeed terminates at a reasonable time.

Beside the main idea of the LLL algorithm, an improvement can be made so that the computation is simpler. Note that in the two applications of LLL reduction described in the previous section, the input bases can be rounded up to integral coefficients. Thus, the LLL algorithm above can be modified so that all computations will be done over $\mathbb{Z}$ instead of using floating-point or rational arithmetic. The obstruction is that $\mu_{i,j}$ and $|\boldsymbol{b}_i^*|^2$ can be rational, and with the frequent use of GCD computations for rational arithmetic, the algorithm will be slowed down. Alternatively, a method can be devised that cares about these values only in abstract terms, that is, to indirectly represent them. The following results enlighten this idea by introducing some new notations, as illustrated in [5, pp. 92-94]:

Proposition 4.5.2. *Let $\{\boldsymbol{b}_i\}$ be an integral basis of a lattice L of rank n, and hence the Gram matrix $\boldsymbol{Q}$ is also integral. For each $i \leq n$ forms a matrix $\boldsymbol{M}$ of size $i \times i$ in which $\boldsymbol{M}_{r,s} = \boldsymbol{b}_r \cdot \boldsymbol{b}_s$ for $1 \leq r, s \leq i$. Then d_i can be defined such that*

$$d_i = det(\boldsymbol{M}) = \prod_{1 \leq j \leq i} |\boldsymbol{b}_j^*|^2 \tag{4.8}$$

Also, for all $j < i$ it can be concluded that $d_{i-1}|\boldsymbol{b}_i^*|^2 \in \mathbb{Z}$, $d_j \mu_{i,j} \in \mathbb{Z}$, and for $j < m \leq i$,

$$d_j \sum_{1 \leq k \leq j} \mu_{i,k}\mu_{m,k}|\boldsymbol{b}_k^*|^2 \in \mathbb{Z}$$

Proposition 4.5.3. *With the notations used in Proposition 4.5.2, define $\lambda_{i,j} = d_j \mu_{i,j}$ for $j < i$. Then $\lambda_{i,j} \in \mathbb{Z}$ and $\lambda_{i,i} = d_i$. In addition, for each pair (i, j) with $j \leq i$, $u_{(i,j),k}$ can be inductively defined with $u_{(i,j),0} = \boldsymbol{b}_i \cdot \boldsymbol{b}_j$ and for $1 \leq k < j$,*

$$u_{(i,j),k} = \frac{d_k u_{(i,j),k-1} - \lambda_{i,k}\lambda_{j,k}}{d_{k-1}}$$

where $u_{(i,j),k} \in \mathbb{Z}$ and $u_{(i,j),j-1} = \lambda_{i,j}$.

Thus, instead of recomputing $\mu_{i,j}$ as in the original algorithm using rational arithmetic, one could alternatively represents them using pairs $(d_j, \lambda_{i,j})$ where $\lambda_{i,j}$ can be computed by induction on $u_{(i,j),k}$. In summary, the modification can be ordered, as given in pseudocode form in Algorithm 4.5.1.

Algorithm 4.5.1: IntegralLLLReduction

Input: An integral basis $\{\boldsymbol{b}_i\}$ of L of rank n
Output: a $\boldsymbol{H}$ matrix contains coordinates of LLL-reduced basis

```
1  begin
       /* Initialize algorithm                                          */
2      k ← 2 ;                              /* The case to work with */
3      k_max ← 1 ;  /* Indicate what vectors have been reduced */
4      d_0 ← 1 ;                                          /* See (4.8) */
5      d_1 ← b_1 · b_1;
6      H ← I_n;
7      while k ≤ n do
8          if k > k_max then
9              k_max ← k;
10             for j ← 1to k do
11                 ▷ Compute u_{(k,j),j−1} as in Proposition 4.5.3;
12                 if j < k then
13                     λ_{k,j} ← u_{(k,j),j−1};
14                 else
15                     d_k ← u_{(k,j),j−1};
16                 endif
17             endfor
18         endif
19         REDI(k, k − 1) ;        /* If μ_{k,k−1} > 1/2, b_k ← b_k − q b_{k−1} */
           /* If |b*_k|^2 < (3/4 − μ^2_{k,k−1}) |b*_{k−1}|^2                  */
20         if d_k d_{k−2} < (3/4) d^2_{k−1} − λ^d_{k,k−1} then
21             SWAPI(k) ;                          /* Swap b_k and b_{k−1} */
22             k ← max(2, k − 1) ;        /* Come back to case k − 1 */
23         else
24             for l = k − 2 to 1 do  REDI(k, l);
25             k ← k + 1;
26         endif
27     endw
28 end
```

Algorithm 4.5.2: Sub algorithm REDI

Input: k, l
Output: $\boldsymbol{H}, \{\lambda_{i,j}\}, \{\boldsymbol{b}_i\}, \{d_i\}$ modified, if any
Data: $\boldsymbol{H}, \{\lambda_{i,j}\}, \{\boldsymbol{b}_i\}, \{d_i\}$

begin
  if $|2\lambda_{k,l}| > d_l$ **then**
    $q \leftarrow 2\lambda_{k,l} + d_l \ div \ 2d_l$; /* $q = \left\lfloor \frac{\lambda_{k,l}}{d_l} \right\rceil = \lfloor \mu_{k,l} \rceil$ */
    $\boldsymbol{H}_k \leftarrow \boldsymbol{H}_k - q\boldsymbol{H}_l$; /* Denote by H_i the column of H */
    $\boldsymbol{b}_k \leftarrow \boldsymbol{b}_k - q\boldsymbol{b}_l$;
    $\lambda_{k,l} \leftarrow \lambda_{k,l} - qd_l$;
    for $i \leftarrow 1$ **to** $l - 1$ **do** $\lambda_{k,i} \leftarrow \lambda_{k,i} - q\lambda_{l,i}$;
  endif
end

Algorithm 4.5.3: Sub algorithm SWAPI

Input: k
Output: $\boldsymbol{H}, \{\lambda_{i,j}\}, \{\boldsymbol{b}_i\}, \{d_i\}$ modified, if any
Data: $\boldsymbol{H}, \{\lambda_{i,j}\}, \{\boldsymbol{b}_i\}, \{d_i\}$

begin
  ▷ Exchange $\boldsymbol{H}_k, \boldsymbol{H}_{k-1}$;
  ▷ Exchange $\boldsymbol{b}_k, \boldsymbol{b}_{k-1}$;
  for $j \leftarrow 1$ **to** $k - 2$ **do** ▷ Exchange $\lambda_{k,j}, \lambda_{k-1,j}$;
  $\lambda \leftarrow \lambda_{k,k-1}$;
  $B \leftarrow (d_{k-2}d_k + \lambda^2)/d_{k-1}$;
  for $i \leftarrow k + 1$ **to** k_{max} **do**
    $t \leftarrow \lambda_{i,k}$;
    $\lambda_{i,k} \leftarrow (d_k\lambda_{i,k-1} - \lambda t)/d_{k-1}$;
    $\lambda_{i,k-1} \leftarrow (Bt + \lambda\lambda_{i,k})/d_k$;
  endfor
  $d_{k-1} \leftarrow B$;
end

4.6 Finding square root of approximation α

Recall that in Algorithm 4.2.1, after the approximation process has been done, a "small" algebraic integer α is returned which is also a perfect square in $\mathbb{Z}[\hat{\theta}]$. At this point, the remainder of this algorithm is to find the square root of α, with an advantage that the coefficients of α as a polynomial in $\hat{\theta}$ are small integers, and thus finding its square root would not require much amount of computation. The first problem is to construct α. However, due to the size of the set U, it is not a good idea to compute α directly by multiplying each element of U as it may come back to the original problem with coefficient size. Fortunately, there exists an explicit algorithm that suits the case whose principle relies on a popular observation, namely the Chinese remainder theorem (CRT). The idea behind this algorithm can be pointed out from the beginning of the theorem, as can be seen from the following special case of the CRT:

Proposition 4.6.1. *(Chinese remainder theorem) Let $p_1, \ldots, p_k$ be pairwise coprime integers and $x_1, \ldots, x_k$ be arbitrary integers. Then there exists $x \in \mathbb{Z}$ such that $x \equiv x_i \pmod{p_i}$ for $1 \leq i \leq k$. Moreover, x is unique modulo the product of $p_1, \ldots, p_k$.*

Proof. Let $P = \prod_{i=1}^{k} p_i$ and $P_i = P/p_i$. Since p_i are pairwise coprime, then for each i it is obvious that $\gcd(P_i, p_i) = 1$, and hence there exists an integer a_i such that $a_i P_i \equiv 1 \pmod{p_i}$. The solution can then be constructed as follows:

$$x = \sum_{i=1}^{k} a_i P_i x_i \tag{4.9}$$

To check back the conditions, note that $P_i \equiv 0 \pmod{p_j}$ for $i \neq j$, thus the result follows for $1 \leq j \leq k$:

$$x = \sum_{i=1}^{k} a_i P_i x_i \equiv a_j P_j x_j \equiv x_j \pmod{p_j}$$

On the other hand, assume $x \equiv y \pmod{P}$ with $x \neq y$, then for $1 \leq i \leq k$ there exists $m_i \in \mathbb{Z}^+$ such that $x - y = m_i p_i$. Since p_i are pairwise coprime, then $p_j | m_i$ for $1 \leq j \leq k$ and $j \neq i$, and hence $P_i | m_i$ or $P | x - y$. This implies either $x = y$ or only one of them can be in $[0, P - 1]$. □

In this case p_i can be freely selected to be prime integers to avoid computing their greatest common divisors. Considering the leftover square, this theorem also holds when it is applied to find the polynomial form of α in $\mathbb{Z}[\hat{\theta}]$

from a large number of its factors. Additively, the set of primes p_i should be chosen so that for each p_i the polynomial $f(x)$ is irreducible over $\mathbb{Z}/p_i\mathbb{Z}$, and thus these primes are called inert primes. This is to make sure that the operation over each finite field is correct as was done over $\mathbb{Z}$. Other than that, in order to know how many inert primes that should be chosen to bound the coefficients of α, a number of experiments should be done to find this bound, which is normally represented in its logarithmic form. Based on the above statements, Algorithm 4.6 gives a demonstration on how α should be factored.

As can be seen from the pseudocode, this method has an advantage over trivial multiplication that all computations are done over some finite field $\mathbb{F}_p$, and hence the coefficients of α are restricted to be small. In practice, this improvement is so significant that even this method performs $\mathcal{O}(\#IPB)$ more polynomial multiplications than the trivial method, the time required for it to finish is much more reasonable.

By applying the CRT, one could assume that α, the approximation of γ, is obtained. The rest of the algorithm is now to find its square root before multiplying it with a set of $\delta_i^{s_i}$ aforementioned. Note that in most cases this part can be skipped since α is already in $\mathbb{Z}$ due to the effectiveness of the previous approximation step. Nevertheless, to make sure that the algorithm works well in general, it is necessary to consider the case when α arrives as a polynomial in θ of degree $d \geq 1$. Since then, there exist various types of methods that can be used to find $\sqrt{\alpha}$, either by factoring its polynomial form, or finding its square root in a finite field. In this context, an idea created by M. Cipolla is used as it was successfully implemented in the GGNFS, a reliable implementation of the GNFS algorithm. To understand this idea, it is required that the following result be used, as its proof is given in [15, pp. 50-51,54]:

Proposition 4.6.2. *Let $\mathbb{K}$ be a finite field of order q and $\boldsymbol{L}$ be its finite extension of order q^k. For each $\alpha \in \boldsymbol{L}$ define k embeddings $\sigma_i = \alpha^{q_i}$ for $0 \leq i \leq k-1$, called the conjugates of α. Then the norm of α, defined as*

$$N(\alpha) = \prod_{i=0}^{k-1} \sigma_i(\alpha)$$

is a mapping from $\boldsymbol{L}$ onto $\mathbb{K}$.

According to this proposition, it is clear that $N(\alpha) = \alpha^{\frac{q^k-1}{q-1}} \in \mathbb{F}_q$. For our particular case, $\boldsymbol{L}$ is referred to as a quadratic extension of the finite

Algorithm 4.6.1: CRTAlgorithm

Input: A set V of (δ_i, s_i) and a set U of $(a_i - b_i\theta, e_i)$, MaxModulus
Output: $\alpha = (\prod_{i=1}^{\#U}(a_i - b_i\theta)^{e_i}) \prod_{i=1}^{\#V} \delta_i^{-2s_i}$

begin
  $IPB \leftarrow \{\}$; `/* Initialize the set of inert primes */`
  $P \leftarrow 1$; `/* Initialize` P `in Proposition 4.6 */`
  $p \leftarrow 2$;
  while $P < MaxModulus$ **do**
    while $f(x)$ *is not irreducible over* $\mathbb{F}_p$ ***or*** $p|disc(f)$ **do**
      $p \leftarrow NextPrime(p)$; `/* Find next prime >` p `*/`
    endw
    $IPB \leftarrow IPB \cup p$;
    $P \leftarrow pP$;
  endw
  `/* Computing` α_p`, the CRT residues of` α `*/`
  foreach $p \in IPB$ **do**
    foreach $(a_i - b_i\theta, e_i) \in U$ **do**
      $\alpha_p \leftarrow \alpha_p(a_i - b_i\theta)^{e_i} \pmod{(f(x), p)}$;
    endfch
    foreach $(\delta_i, s_i) \in V$ **do**
      $\alpha_p \leftarrow \alpha_p \delta_i^{-2s_i} \pmod{(f(x), p)}$;
    endfch
  endfch
  $\alpha \leftarrow 0$;
  foreach $p \in IPB$ **do**
    $a_p \leftarrow \left(\frac{P}{p}\right)^{-1} \pmod{p}$;
    $\alpha \leftarrow \alpha + a_p \frac{P}{p} \alpha_p \pmod{P}$;
  endfch
end

field $\mathbb{F}_q$ with q is a prime integer p raised to the power of $deg(f(x))$, i.e., p^n. Considering Cipolla's algorithm as described in [3, pp. 157-159], it is assumed that there exists an element $x \in \boldsymbol{L}$ such that $N(x) = \alpha$. It is clear, thought, that the polynomial form of α is in $\mathbb{F}_q$, and hence such x is assured to exist, since according to the above proposition the norm map is onto. In addition, as the underlying field is quadratic, the norm of x can be simplified to $N(x) = x^{q+1} = \alpha$, directly by the definition of this mapping. Thus, the square root of α can be easily obtained with $x^{(q+1)/2}$. This leads to the complete algorithm of finding $\sqrt{\alpha}$, as given in Algorithm 4.6.2.

Algorithm 4.6.2: CipollaSquareRoot

Input: $\alpha \in \mathbb{F}_q$
Output: $\sqrt{\alpha}$

begin
  $t \leftarrow Random() \in \mathbb{F}_q$;
  while $t^2 - 4\alpha$ *is a square* **do**
    $t \leftarrow Random() \in \mathbb{F}_q$;
  endw
  $f(X) \leftarrow X^2 - tX + \alpha$;
  $e \leftarrow \frac{q+1}{2}$;
  $b(X) \leftarrow 1$;
  $s(X) \leftarrow X$;
  `/* Use of successive squares method */`
  while $e > 0$ **do**
    if $e \pmod 2 = 1$ **then**
      $e \leftarrow e - 1$;
      $b(X) \leftarrow Xb(X) \pmod{f(X)}$;
    endif
    if $e > 0$ **then**
      $e \leftarrow \frac{e}{2}$;
      $s(X) \leftarrow s(X)^2$;
      $b(X) \leftarrow s(X)b(X) \pmod{f(X)}$;
    endif
  endw
  $\sqrt{\alpha} \leftarrow b(X)$;
end

In terms of complexity, as the running time of Cipolla's algorithm takes $\mathcal{O}((log_{10}q)^3)$, there is not much computing resource to be concerned in this

particular application. Also, this stage completely concludes the square root algorithm, as well as hopefully terminates the whole GNFS algorithm, provided that the greatest common divisors found are not trivial. Otherwise, either the algorithm should be restarted from the beginning, or it should find another set U of smooth values which also forms the perfect squares in both $\mathbb{Z}$ and $\mathbb{Z}[\theta]$.

Remark. Note that in Line 21 it was not clear that $b(X)$ is a square root of α since it may contains variable X in its polynomial form. In order to verify that this algorithm is valid, it can be assumed that $b(X) = kX + m$ for some k and m in $\mathbb{F}_q$. Moreover, the quadratic extension $\boldsymbol{L}$ used in this algorithm is $\mathbb{F}_q[X]/(X^2 - tX + \alpha)$, where t was chosen as in Line 3. From this assumption, the following result confirms that $b(X)$ is indeed in $\mathbb{F}_q$:

Proposition 4.6.3. *Let $\mathbb{K}$ be a finite field of odd order q, with $\alpha, t \in \mathbb{K}$ such that α is a square and $t^2 - 4\alpha$ is not a square. Then the value of $x^{(q+1)/2}$ in the quadratic extension $\boldsymbol{L} = \mathbb{K}[X]/(X^2 - tX + \alpha)$ is $\sqrt{a}$ in $\mathbb{K}$.*

Proof. To equip the proof, recall firstly that if $t^2 - 4\alpha$ is not a square, then $(t^2 - 4\alpha)^{(q-1)/2} = -1$. Indeed, in the finite field $\mathbb{K}$, there exists a *primitive element* γ generating $\mathbb{K}$ and $i \in \mathbb{Z}$ such that $\gamma^i = (t^2 - 4\alpha)$. Since there are as many *primitive elements* as $\phi(q-1)$, it can always be assumed that $\gamma \neq t^2 - 4\alpha$. Assume on the contrary that $(t^2 - 4\alpha)^{(q-1)/2} = 1$, then $\gamma^{i(q-1)/2} = 1$, and it turns out that $i > 2$ as i must be odd and $i \neq 1$. This leads to the fact that $(q-1)|i(q-1)/2$ since $\gamma^{q-1} = 1$ following *Lagrange's theorem*[††], and hence $2|i$ which causes a contradiction.

Let $g(X) = X^2 - tX + \alpha$, and express it in the form:

$$g(X) = X^2 - 2\frac{t}{2}X + \left(\frac{t}{2}\right)^2 - \left(\frac{t}{2}\right)^2 + \alpha = \left(\frac{t}{2} - X\right)^2 - \left(\frac{t}{2}\right)^2 + \alpha$$

Let $X_1 = \frac{t}{2} - X$, we can now observe the following manipulation where $(t^2 - 4\alpha)/4 = (t^2 - 4\alpha)(2^{-1})^2$ is also not a square since q is odd:

$$X_1^q = X_1(X_1^2)^{\frac{q-1}{2}} = X_1(X_1^2 - g(X))^{\frac{q-1}{2}} = X_1\left(\left(\frac{t}{2}\right)^2 - \alpha\right)^{\frac{q-1}{2}} = -X_1$$

[††]In Lagrange's theorem, the order of every subgroup H of G always divides the order of G, i.e., $|G| = k|H|$ for some $k \in \mathbb{Z}^+$.

Then,

$$X^q = \left(\frac{t}{2} - X_1\right)^q = \left(\frac{t}{2}\right)^q - \binom{q}{1}\left(\frac{t}{2}\right)^{q-1} X_1 + \cdots + \binom{q}{1}\frac{t}{2}X_1^{q-1} + (-X_1)^q$$
$$= \frac{t}{2} + X_1$$

Considering the image of $X^{(q+1)/2}$ in $\boldsymbol{L}$, i.e., $b(X)$ it follows that:

$$k^2X^2 + 2kmX + m^2 = (kX + m)^2 = b(X)^2 = X^{q+1} = \left(\frac{t}{2} + X_1\right)\left(\frac{t}{2} - X_1\right)$$
$$= \left(\frac{t}{2}\right)^2 - X_1^2 = \alpha$$

Thus, the variable X on the left-hand size must be canceled, making $k = 0$, and hence $b(X) = m = \sqrt{\alpha}$. □

Chapter 5

Completing the sieving part

So far the idea behind the GNFS has been revealed as it differs from the Quadratic Sieve by the use of the ring $\mathbb{Z}[\theta]$ for finding a perfect square in the field of complex numbers $\mathbb{C}$. Given a monic, irreducible polynomial $f(x)$ with integer coefficients and an integer m such that $f(m) \equiv 0 \pmod{n}$, the algorithm starts its sieving technique by finding enough pairs (a, b) so that $a + bm$ is smooth over a rational factor base F and $a + b\theta$ is smooth over an algebraic factor base A. After the sieving step, the remaining tasks are somewhat similar to that explained in the Quadratic Sieve algorithm with a slight modification to further enhance the perfect square and finally to compute the square roots.

Originally speaking, it has so far been assumed that everything necessary to start an instance of the GNFS has been generated. In the actual implementation, these initial factors are unfortunately not trivial to be chosen. Recall that in the case of the Quadratic Sieve, the polynomial $f(x) = x^2 - n$ is chosen regardless of the value of n. On the contrary, the GNFS provides a flexibility to select a good polynomial that makes certain positive distinctions to the other sieving techniques. Besides its obvious benefits, this advantage raises a question on how to measure if a polynomial is good, and how to choose a good one.

On the other hand, as long as a polynomial $f(x)$ is chosen, it is also necessary to form the factor bases as the second component of the sieving step. However, it turns out that this is not as easy as in the Quadratic Sieve unless one chooses to construct the factor bases using the brute force method. By addressing these initial issues, this chapter is devoted to somehow overcome the initial constraints before properly executing the algorithm. It starts by mentioning some concepts used in assessing the quality of a polynomial used

in the GNFS, after which it describes an algorithm suggested by B. Murphy in [20], with a significant improvement from T. Kleinjung in [12]. In addition, at the end of this chapter, a different technique called *lattice sieve* is introduced that serves for finding smooth values. In fact, this technique excessively predominates the line sieving method described in §2.3.

5.1 Generalizations of polynomial selection

In the case of the GNFS, there are in fact two polynomials to be considered in this selection process. Recall that in (2.2) the polynomial $f(x)$ is used to form the ring for smooth values $(a+b\theta)$, as well as the corresponding smooth values $a+bm \in \mathbb{Z}$. Also, in (4.2) b was replaced by $-b$ in order to simplify the norm function without affecting the result. In this manner, $a-bm$ can be treated in the same way as $a-b\theta$ if one thinks of the polynomial $g(x) = x-m$, and m being a root θ_g of $g(x)$, with $a-bm \in \mathbb{Z}[\theta_g]$. The fortunate fact is that $g(x)$ is linear, and that it has $m \in \mathbb{Z}$ as the only root, hence equalizing $\mathbb{Z}[\theta_g]$ with $\mathbb{Z}$. Thus, the process of constructing squares in §2.1 can be generalized so that there is no distinction between sieving in the algebraic and rational sides. In fact, it can be assumed that the algorithm makes use of two polynomials $f_1(x)$ and $f_2(x)$ having the same root m modulo n, while their complex roots are θ_1 and θ_2, respectively. Accordingly, it is possible to define two separate mappings $\phi_1 : \mathbb{Z}[\theta_1] \to \mathbb{Z}/n\mathbb{Z}$ and $\phi_2 : \mathbb{Z}[\theta_2] \to \mathbb{Z}/n\mathbb{Z}$ that map θ_1 and θ_2 to m. Suppose that a set U of pairs (a, b) is formed so that their products form perfect squares in both rings, then these squares are indeed congruent modulo n, since

$$\gamma_1^2 = \phi_1\left(\prod_{(a,b)\in U}(a-b\theta_1)\right) \equiv \prod_{(a,b)\in U}(a-bm) \pmod{n}$$

$$\gamma_2^2 = \phi_2\left(\prod_{(a,b)\in U}(a-b\theta_2)\right) \equiv \prod_{(a,b)\in U}(a-bm) \pmod{n}$$

In order to select the polynomial pair that satisfies the above conditions while they are practically independent, the simplest way is to find $f_1(x)$ and $f_2(x)$ associated with m such that $f_1(x) = n$, whereas $f_2(m) = 0$. A technique that sufficiently addresses this idea is referred to as the base-m method. As its name implies, given a common root m, the method determines the "digits" of n as a number in base m. Assume that these "digits" starting from least

significant one are $a_0, a_1, \ldots, a_d$ with $|a_i| < m$, the polynomial $f_1(x)$ can be constructed as follows:

$$f_1(x) = a_d x^d + a_1 x^{d-1} + \cdots + a_1 x + a_0$$

$f_2(x)$ can then be chosen to be $x - m$. As a result, it can be easily verified that these two polynomials possess the same root m modulo n. In addition, the value of m can be found in a relatively optimal way by the selection of the degree d of $f_1(x)$. According to a conjectured complexity analysis given in [14, pp. 76-84], the GNFS can be optimized so that its running time is at most $L_n[\frac{1}{3}, (64/9)^{1/3} + o(1)]$, where

$$L_n[u, v] = e^{v(\ln n)^u (\ln \ln n)^{1-y}}$$

As a necessary condition to partially satisfy this optimization, the degree d of $f_1(x)$ can be selectively computed from the following formula, with its dependency on n:

$$d = \left(3^{\frac{1}{3}} + o(1)\right) \left(\frac{\ln n}{\ln \ln n}\right)^{\frac{1}{3}}$$

Since the value of d is located, it is then convenient to approximate a good choice of m bounded by the d-th root of n, i.e., $\sqrt[d]{n}$. In addition, a sieve-like process can be invoked to search for m close to this value so that the coefficients of $f_1(x)$ are minimized to a certain extent. Nevertheless, this method only focuses on finding valid pairs of $f_1(x)$ and $f_2(x)$, whereas the technique described in the next section concentrates on a more efficient way of weighting polynomials and selecting the best one among those candidates.

5.2 Defining polynomial yield

Generally speaking, the yield of a polynomial $f(x)$ is simply the number of smooth values obtained from the norm $N(x - y\theta)$ over a sieving region $\mathcal{A}^*$. According to this definition, a polynomial $f_1(x)$ is preferred over another polynomial $f_2(x)$ if there exists a method to estimate that its yield is higher, i.e., $f_1(x)$ possesses higher probability of having smooth values than $f_2(x)$ over the same sieving region. Consequently, this value will mainly be used as the mean for categorizing and selecting the "best" polynomial for the sieving step. In addition, the selection process can be simplified by concerning only *skewed polynomials*, i.e., polynomials whose coefficients are in ascending order. More precisely, let $f(x)$ be a skewed polynomial of degree d, then

*The range of a and b used for finding smooth values. For the line sieving technique described in Chapter 2, $\mathcal{A} = [-A, A] \times [1, B] \cap \mathbb{Z}^2$.

$|a_d| < |a_{d-1}| < \ldots < |a_0|$ and the ratio a_i/a_{i-1} should be approximately constant for $1 \leq i \leq d$.

In principle, there are two criteria that can be used to assess the yield, namely size and root properties. *Size properties* refer to the average size of $N(a - b\theta)$ over a given sieving region. Obviously, this feature directly affects the yield of a polynomial. In other words, the smaller the size of $N(a - b\theta)$, the higher the probability that it can be decomposed into primes within the factor base. Therefore, the corresponding task is to choose $f_1(x)$ and $f_2(x)$ so that $N_1(a, b)^\dagger = N(a - b\theta_1)$ and $N_2(a, b) = N(a - b\theta_2)$ are optimized. Note that since $f_2(x)$ is linear, the value of $N_2(a, b)$ does not vary remarkably among the selections of pairs $(a, b) \in \mathcal{A}$, or by choices of $m \approx \sqrt[d]{n}$. This leaves $f_1(x)$ as the main concern in the selection process. In particular, given a pair of polynomial found from the previous method, there exist two simple ways of altering $f_1(x)$ and $f_2(x)$ whilst preserving its root m modulo n, defined as follows:

1. Translation by $t \in \mathbb{Z}$: Let $g_1(x) = f_1(x - t)$ and $g_2(x) = f_2(x - t)$ and $m_t = m + t$, then $g_1(m_t) \equiv g_2(m_t) \pmod{n}$.
2. Rotation by $P(x)$ with $deg(P) < deg(f)$: Let $f_P(x) = f_1(x) + P(x)(x - m)$, then $f_P(m) \equiv f_2(m) \pmod{n}$.

Also, with respect to the size properties of $f_1(x)$, the value of $N_1(a, b)$ is affected by the *shape* of the region $\mathcal{A}$ used in the sieving step. In other words, given a rectangle region $\mathcal{A}$ of size $2A \times B$, the average size of $N_1(a, b)$ depends on the ratio between the length and height of $\mathcal{A}$, namely the skewness s of $\mathcal{A}$ given by $s = \frac{A}{B}$. Note that the skewness s is distinct from the skewness defined previously for a polynomial. However, as long as the algorithm attempts to find a good polynomial, s can be thought of as a compensation for balancing the skewness of $f_1(x)$.

In this manner, since the yield of $f_1(x)$ is not affected by the size of $\mathcal{A}$, it can be reduced to the simplest form when $A = \sqrt{s}$ and $B = 1/\sqrt{s}$, and hence a concrete measure of the size properties of $f_1(x)$ follows:

$$\int_{\mathcal{A}} N_1^2(x, y)dxdy = \int_0^{\frac{1}{\sqrt{s}}} \int_0^{\sqrt{s}} N_1^2(x, y)dxdy$$

From the other perspective, *root properties* of $f_1(x)$ is literally defined as the distribution of roots of $N_1(x, y)$ modulo p^k for some small primes

†Note that from now on this notation will be used to represent the norm, as it is shorter than the conventional one.

p and $k > 1$. Unlike size properties, the importances of root properties in polynomial selection is however unclear when it comes to estimating the yield. Thus, in order to explain its contributions to this aspect, it is necessary that a relating concept be mentioned, as shown below:

Definition 5.2.1. *Let $v_p(v)$ be the p-adic valuation of a number v, then $cont_p(v)$ is defined to be the mean of $v_p(v)$ across some sample S, i.e.,*

$$cont_p(v) \approx \frac{\sum_{v \in S} v_p(v)}{\#S}$$

Considering a random integer i_r generated uniformly so that $i_r < r$, then it is obvious that the probability of i_r dividing p^k for some prime p and $k > 1$ is $1/p^k$, thus $cont_p(i_r)$ can be computed as:

$$cont_p(i_r) = \frac{1}{p} + \frac{1}{p^2} + \frac{1}{p^3} + \cdots = \frac{1}{p-1}$$

On the other hand, let q_p be the number of roots modulo p of $N_1(x, y)$, then for a reason explained in [20, p. 46-47], it is easy to check that the expected valuation of p in $N_1(x, y)$ is of the form

$$cont_p(N_1) = q_p \frac{p}{p^2 - 1}$$

In fact, given a small bound B, if $N_1(x, y)$ is treated as a random integer i_r, then the probability that $N_1(x, y)$ is smooth over some factor base A is the same as that of γ, where

$$\ln \gamma = \ln i_r - \sum_{p \leq B} cont_p(i_r) \ln p = \ln i_r - \sum_{p \leq B} \frac{\ln p}{p-1}$$

On the contrary, if the value of $N_1(x, y)$ behaves as it normally should, then the likelihood that it is smooth over the same factor base can be computed differently. In this case this value is equal to that of β given by

$$\ln \beta = \ln N_1(x, y) - \sum_{p \leq B} cont_p(N_1) \ln p = \ln N_1(x, y) - \sum_{p \leq B} q_p \frac{p}{p^2 - 1} \ln p$$

Let $\alpha(N_1) = \ln \beta - \ln \gamma$, then α can be thought of as a measurement on how root properties of $f_1(x)$ would affect the chance that $N_1(x, y)$ is smooth. In practice, if α appears to be much less than 0, then since $\ln \beta = \ln \gamma + \alpha$, $N_1(x, y)$ is analogously the same as a random integer of size β, where $\beta = \gamma e^{\alpha}$. In other words, $N_1(x, y)$ behaves as if it is a random integer of a smaller size

than it normally is. This confirms the impact of root properties on the yield of $f_1(x)$. To sum up all the aforementioned observations, Algorithm 5.2.1 devises a reasonable process to find a list of good polynomials that possesses both root and size properties.

5.3 Selecting the best polynomial

After the execution of Algorithm 5.2.1, it is very likely that a huge number of so far "good" polynomials will be produced, all of which to a certain extent would produce more smooth values than normal. The problem however, is due to the fact that it is still impractical to make sample on their yields as they may count to millions. Alternatively, if a less expensive method of rating polynomials is available, then the best polynomials of similar yields can be isolated, thereafter they may be sieved over some sample interval to find the best one for the sieving step.

A simple idea suggested in [20, pp. 85-88] is to turn these polynomials into polar coordinates. In particular, for each (x, y) there exists a pair (r, θ) such that $x = r\ \cos\theta$ and $y = r\ \sin\theta$. Note that in the case of skewed polynomials sieving over a skewed region, it is necessary that $s = x/y$, and thus it is better to use $x = \sqrt{s}r\ \cos\theta$ and $y = 1/\sqrt{s}r\ \sin\theta$. Then due to the nature of $N_i(x, y)$, i.e., a homogeneous polynomial, it can be converted in to a new coordinate as:

$$N_i(x, y) = r^d N_i\left(\sqrt{s}\cos\theta, \frac{1}{\sqrt{s}}\sin\theta\right)$$

Note that for each $f_i(x)$ in the list above, since $N_i(x, y)$ contain the common factor r^d, this factor cannot be used in the rating process and hence can be assumed to be 1 without affecting the ranking algorithm. As a result, it is sufficient to concentrate only on the value of $N_i\,(s_1 cos\theta, s_2 sin\theta)$ for $s_1 = \sqrt{s}$ and $s_2 = 1/\sqrt{s}$. Moving a step further, for the sake of simplicity one could define the following notation:

$$u_{N_j}(\theta_i) = \frac{\ln|N_j(s_1\cos\theta_i, s_2\sin\theta_i)| + \alpha(N_j)}{\ln B_{N_j}}$$

where B_{N_j} is a good bound for the factor base in accord with $f_j(x)$ and $N_j(x, y)$. Literally speaking, $u_{N_j}(\theta_i)$ is simply the combination of root and size properties with respect to a bound B_{N_j}, sampling from pairs (x, y) of the form $(s_1\cos\theta_i, s_2\sin\theta_i)$. Furthermore, u_{N_j} becomes effective in this ranking process when it is pointed by the *Dickman function* [8], defined by

Algorithm 5.2.1: PolynomialSelection

Input: A number n, F_{max}, limit J_0, J_1, list P of small primes
Output: A list of polynomials with $f(m) \equiv 0 \pmod{n}$

begin
  $d \leftarrow \left(3^{\frac{1}{3}} + o(1)\right) \left(\frac{\ln n}{\ln \ln n}\right)^{\frac{1}{3}}$;
  $count \leftarrow 0$; `/* Number of polynomials found */`
  while $count < F_{max}$ **do**
    $a_{d-1} \leftarrow \infty$; $a_{d-2} \leftarrow \infty$;
    while a_{d-1} *and* a_{d-2} *are large* **do**
      ▷ Find a_d dividing many small p_i^k for primes $p_i \in P$;
      $m \leftarrow \left\lfloor \left(\frac{n}{a_d}\right)^{\frac{1}{d}} \right\rfloor$;
      ▷ Compute $f_m(x)$;
    endw
    $x_t \leftarrow x - t$; $m_t \leftarrow m - t$; `/* Translate` $f_m(x)$ `by` t `*/`
    $f(x) \leftarrow f_m(x_t) + (c_1 x + c_0)(x - m_t)$; `/* Rotate` $f_m(x_t)$ `*/`
    ▷ Find min $P(t, c_0, c_1) \leftarrow \int_0^{\frac{1}{\sqrt{s}}} \int_0^{\sqrt{s}} N_1^2(x, y) dx dy$;
    $I(f, s) \leftarrow 1/2 \ln P(t, c_0, c_1)$;
    if $I(f, s)$ *is small* **then** `/* With good root properties */`
      for $j_1 \leftarrow -J_1$ **to** J_1 **do**
        for $j_0 \leftarrow -J_0$ **to** J_0 **do**
          $f_{j_0,j_1}(x) \leftarrow f(x) + (j_1 x - j_0)(x - m)$;
          foreach $p \in P$ **do**
            $q_p \leftarrow CountRoot(f_{j_0,j_1}(x) \equiv 0 \pmod{p})$;
            if $p | a_d$ **then** $q_p \leftarrow q_p + \frac{1}{p} + \frac{1}{p^2} + \cdots + \frac{1}{p^{v_p(a_d)}}$ ‡;
            $cont_p(N_{j_0,j_1}) \leftarrow q_p \frac{p}{p^2 - 1}$;
          endfch
          $\alpha(N_{j_0,j_1}) \leftarrow \sum_{p \in P} \left(\frac{1}{p-1} - cont_p(N_{j_0,j_1})\right) \ln p$;
          if $I(f, s) + \alpha(N_{j_0,j_1})$ *is small* **then**
            ▷ Output $f_{j_0,j_1}(x)$;
            $count \leftarrow count + 1$;
          endif
        endfor
      endfor
    endif
  endw
end

Definition 5.3.1. *Let $P_j(n)$ denote the j-th largest factor of n, define by $\psi(r, B)$ the number of $n \leq r$ such that their largest factor, i.e., $P_1(n)$ is bounded by B. Then for $v \in \mathbb{R}$ and $v > 0$ the Dickman function can be defined as*

$$\rho(v) = \lim_{r \to \infty} \frac{\psi(r, r^{1/v})}{r} \text{for } v > 1$$

or 1 otherwise.

In other words, $\rho(v)$ heuristically determines the probability that a uniformly random integer i_r is B-smooth with $v = \ln r / \ln B$. By applying the Dickman function, $\rho(u_{N_j}(\theta_i))$ indicates the probability for i_r to be B-smooth where $r \approx u_{N_j}(\theta_i)$, which is exactly the yield of $f_j(x)$. To effectively compute $\rho(u)$, the following result gives a deeper analysis on the characteristics of the Dickman function, according to a study by Norton given in [23]:

Proposition 5.3.1. *Let $\rho(u)$ be the Dickman function of u, then $\rho(u)$ satisfies the following equations:*

$$\begin{cases} \rho(u) = 1 & \text{for } 0 \leq u \leq 1 \\ u\rho'(u) + \rho(u-1) = 0 & \text{for } u > 1 \end{cases} \tag{5.1}$$

This result shows that $\rho(u)$ is a *delay differential equation*[§] since the derivate of $\rho(u)$ depends on the value of ρ taken previously, i.e., at $u - 1$. This observation implies that $\rho(u)$ is piecewise analytic, that is, for each $k \in \mathbb{Z}^+$ there exists an analytic function $\rho_k(u)$ such that $\rho_k(u) = \rho(u)$ for $k - 1 \leq u \leq k$, and such functions can be dependently computed in the following way [2, pp. 1706-1707]:

Proposition 5.3.2. *Let k be a positive integer and $0 \leq \xi \leq 1$. Define $\rho_k(k - \xi)$ such that*

$$\rho_k(k - \xi) = \sum_{i=0}^{\infty} c_{k,i} \xi^i$$

[§]A delay differential equation is a differential equation in which the derivate of the unknown function depends on values of that function at both present and previous states, that is, $f'(x)$ depends on $f(x)$ and $f(x - \tau)$ for some $\tau > 0$.

where $c_{k,i}$ are computed inductively as follows:

$$c_{1,0} = 1,\ c_{1,i} = 0 \qquad \text{for } i \geq 1 \tag{5.2}$$

$$c_{2,0} = 1 - \ln 2,\ c_{2,i} = \frac{1}{i2^i} \qquad \text{for } i \geq 1 \tag{5.3}$$

$$c_{k,i} = \sum_{j=0}^{i-1} \frac{c_{k-1,j}}{ik^{i-j}} \qquad \text{for } k > 2 \text{ and } i > 0 \tag{5.4}$$

$$c_{k,0} = \frac{1}{k-1} \sum_{j=1}^{\infty} \frac{c_{k,j}}{j+1} \qquad \text{for } k > 2 \tag{5.5}$$

Then for each u and a positive integer l such that $l - 1 \leq u \leq l$, the Dickman function $\rho(u)$ agrees with $\rho_l(u)$, i.e., assume that $\xi = l - u$, then

$$\rho(u) = \rho(l - \xi) = \rho_l(l - \xi) = \sum_{i=0}^{\infty} c_{k,i}\xi^i \tag{5.6}$$

Thus, the computation of $\rho(u)$ becomes simple as it is possible to initiatively compute $c_{k,i}$ up to a certain precision K, then $\rho(u)$ can be mapped to a corresponding analytic function ρ_l in (5.6), provided that $l \leq K$. More precisely, Algorithm 5.3.1 gives formal steps that suffice this computation.

Furthermore, in order to scale this result over all possible pairs (x, y), it is necessary to find the mean of $\rho(u_{N_j}(\theta_i))$ for θ_i ranging from 0 to π. This can be approximately done by dividing $[0, \pi]$ to as small as possible sub-interval, e.g., $k = 1000$, as suggested in [20, p. 87]. Each sub-interval i can then be represented by θ_i, the mean of θ within that sub-interval. Thus for each polynomial $f_1(x)$ and the corresponding norm $N_1(x, y)$, its rating point can be computed as

$$\mathbb{E}(N_1) = \sum_{i=1}^{k} \rho(u_{N_1}(\theta_i))^{\|}$$

In the end, depending on the optimization of the algorithm, a few polynomials $f_1(x)$ whose $\mathbb{E}(N_1)$ are of highest value will be picked up for further sampling before the best one is found. Since the number of these polynomials is trivial, the final testing phase thus becomes practical. As a result, a simple implementation of the overall idea can be found in Algorithm 5.3.2.

¶In order to compute (5.5) and (5.6), it is necessary to replace ∞ by some constant whose value suffices for approximation purposes. As suggested in [2, pp. 1706-1707], 55 is a good choice, and it is assigned to $infinity$.

‖Note that in [20, p. 87], this formula also considers the linear polynomial of the pair (f_1, f_2), but since its value does not dominate the value of $\mathbb{E}(N_{j,1}, N_{j,2})$, it can be omitted.

Algorithm 5.3.1: DickmanFunction

Input: a real number $u > 0$
Output: $\rho(u)$

begin
  $K \leftarrow \lceil u \rceil$; /* Get the bound K of u */
  $infinity \leftarrow 55$¶;
  $\xi \leftarrow K - u$;
  $c_{1,0} \leftarrow 1$;
  for $i \leftarrow 1$ **to** $infinity$ **do** $c_{1,i} \leftarrow 0$;
  $c_{2,0} \leftarrow 1 - \ln 2$;
  for $i \leftarrow 1$ **to** $infinity$ **do** $c_{2,i} \leftarrow \frac{1}{i2^i}$;
  for $k \leftarrow 3$**to** K **do**
    for $i \leftarrow 1$ **to** $infinity$ **do** $c_{k,i} \leftarrow \sum_{j=0}^{i-1} \frac{c_{k-1,j}}{ik^{i-j}}$;
    $c_{k,0} \leftarrow \frac{1}{k-1} \sum_{j=1}^{infinity} \frac{c_{k,j}}{j+1}$;
  endfor
  $\rho(u) \leftarrow \sum_{i=0}^{infinity} c_{K,i}\xi^i$;
end

5.4 Improvement to polynomial selection

In 2005, T. Kleinjung suggested a better method for selecting the polynomial pairs $(f_1(x), f_2(x))$ as a significant replacement to the beginning of Murphy's algorithm. In particular, the improvement was based on the observation that the common root m of $f_1(x)$ and $f_2(x)$ modulo n need not be an integer. The problem with a rational m is that the mappings ϕ_1 and ϕ_2 may produce perfect squares not in $\mathbb{Z}$ and hence ruin the result. This is fortunately not the case since there is always a way for canceling the denominators. In fact, let $f_2(x) = px - m$ for $\gcd(p, m) = 1$, it is clear that $m_1 = \frac{m}{p}$ satisfies $f_2(m_1) \equiv 0 \pmod{n}$. The following result proves that $f_1(x)$ can also be found having the same root modulo n:

Proposition 5.4.1. *Let $n, d \in \mathbb{Z}^+$ and $a_d, p, m \in \mathbb{Z}$ such that $n \equiv a_d m^d \pmod{p}$ and $\gcd(p, m) = 1$. Let $\tilde{m} = \sqrt[d]{\frac{n}{a_d}}$, then there exists a polynomial $f_1(x) = \sum_{i=0}^{d} a_i x^i$ such that*

- $f_1(\frac{m}{p})p^d = n$
- $|a_{d-1}| < p + da_d\frac{m-\tilde{m}}{p}$
- $|a_i| < p + m$ *for* $0 \leq i \leq d-2$

Algorithm 5.3.2: FindBestPolynomial

Input: List of polynomial P with their skewnesses and bounds, f_{max}
Output: List of polynomial T where $\#T = f_{max}$

begin
  $k \leftarrow 1000$;
  foreach $f_{1,j}(x) \in P$ **do**
    $N_{1,j}(x,y) \leftarrow y^d f\left(\frac{x}{y}\right)$;
    $s_1 \leftarrow \sqrt{s_j}$;
    $s_2 \leftarrow 1/\sqrt{s_j}$;
    $u_{N_{1,j}}(x) \leftarrow \frac{\ln|N_{1,j}(s_1 \cos x, s_2 \sin x)| + \alpha(N_{1,j})}{\ln B_{N_{1,j}}}$;
    $\mathbb{E}(N_{1,j}) \leftarrow 0$;
    for $i \leftarrow 1$**to** k **do**
      $\theta \leftarrow \frac{\pi}{k} i + \frac{\pi}{2k}$; `/* Middle value of` $[\frac{\pi}{k}i, \frac{\pi}{k}(i+1)]$ `*/`
      $\mathbb{E}(N_{1,j}) \leftarrow \mathbb{E}(N_{1,j}) + \rho(u_{N_{1,j}}(\theta))$;
    endfor
    $R[j] \leftarrow \{\mathbb{E}(N_{1,j}), f_{1,j}(x)\}$;
  endfch
  $Quicksort(R)$;
  $\triangleright$ Return last f_{max} polynomials in R;
end

Proof. Let $r_d = n$ and inductively construct r_i and a_i for $0 \leq i \leq d-1$ as follows:

$$r_i = \frac{r_{i+1} - a_{i+1}m^{i+1}}{p}$$
$$a_i = \left\lfloor \frac{r_i}{m^i} \right\rfloor + \sigma_i$$

for $0 \leq \sigma_i < p$ such that $r_i \equiv a_i m^i \pmod{p}$. Along the lines of [12, Lemma 2.1], it is clear that such construction satisfies the conditions listed above. □

Note that since $f_1(\frac{m}{p})p^d \equiv 0 \pmod{n}$ and $p^d \equiv 0 \pmod{n}$, it is obvious that m/p is a root of $f_1(x)$ modulo n. Also, by the original assumption, the common root m_1 is explicitly a rational, and hence if it is merely applied for the GNFS, the final perfect squares may not be integers. In this case, instead of using the pair (a, b) for which $a - bm_1$ and $a - b\theta$ are smooth, the algorithm can use (pa, pb) as a replacement whilst preserving the congruence of squares. In fact, this makes no difference to the complexity of the algorithm since the GNFS algorithm can be carried out with pairs (a, b) and multiplied with a power of p after the square root algorithm.

On the other hand, from the second point of the above result it is easily to see that Kleinjung's idea has a significant impact on size properties of polynomial yield. More precisely, this algorithm tends to find polynomial whose three leading coefficients are small and bounded. In order to investigate this advantage, it is first required that the following concept be introduced that gives rise to the bounds of these coefficients:

Definition 5.4.1. *Let $f(x) = \sum_{i=0}^{d} a_i x^i$, then for each skewness s for the sieving region of $f(x)$ define*

$$\sup(f, s) = \max_i |a_i s^{i-\frac{d}{2}}|$$

Then the sup-norm of $f(x)$ can be defined as

$$\sup(f) = \min_{s>0} \sup(f, s)$$

Given $M < \sqrt[d+1]{n}$ as the bound of $\sup(f)$, then the upper bound of the skewness s is $\left(\frac{M}{a_d}\right)^{2/d}$ if one considers the formula of $\sup(f, s)$. In addition, assume in the worst case that $|a_1| \approx \tilde{m}$, the optimal range of the skewness s can be formulated as

$$s_{min} = \left(\frac{M}{\tilde{m}}\right)^{\frac{2}{2-d}} \leq s \leq \left(\frac{M}{a_d}\right)^{\frac{2}{d}} = s_{max} \tag{5.7}$$

This interval indicates the bounds for each of the coefficients of $f(x)$, as given below

$$a_{i,max} = \begin{cases} Ms_{min}^{\frac{d}{2}-i} & \text{for } \frac{d}{2} \leq i < d \\ Ms_{max}^{\frac{d}{2}-i} & \text{for } i < \frac{d}{2} \end{cases} \tag{5.8}$$

As an example, this value when applying to a_d, a_{d-1} and a_{d-2} are respectively $\left(\frac{M^{2d-2}}{n}\right)^{\frac{1}{d-3}}$, $\frac{M^2}{\tilde{m}}$, and $\left(\frac{M^{2d-6}}{\tilde{m}^{d-4}}\right)^{\frac{1}{d-2}}$ if one substitutes s_{min} by its formula given in (5.7).

With each selection of a_d, it is now possible to seek for a value of a_{d-1} of size $\max(p, a_d)$. According to the second condition of Proposition 5.4.1, this can only be the case if m differs from $\tilde{m}$ by at most a small scalar of p. To continue on this strategy, it can be assumed that p is a small integer decomposing into some small primes p_i such that $p_i \equiv 1 \pmod{d}$ and $\gcd(p, a_d n) = 1$. It is now a question to find solutions of the equation $n \equiv a_d x^d \pmod{p}$, in which case it can be verified that the number of such solutions is either 0 or d^l. Indeed, for each i within $[1, l]$ there exist $x_{i,1}, \ldots, x_{i,d} \in \mathbb{F}_p$ dividing $\frac{p}{p_i}$ such that these are d solutions of the equation $n \equiv a_d x^d \pmod{p_i}$. Following this idea, each of the d^l solutions of $n \equiv a_d x^d \pmod{p}$ can then be represented by a unique vector $\mu = (\mu_1, \ldots, \mu_l)$ with $1 \leq \mu_i \leq d$ such that

$$x_\mu = \sum_{i=1}^{l} x_{i,\mu_i}$$

Given $m_0 > \tilde{m}$ as the smallest integer dividing p, then for each x_μ, $m_\mu = m_0 + x_\mu$ is also a solution of $n \equiv a_d x^d \pmod{p}$, whereas m_μ is very close to $\tilde{m}$. Thus, if m is computed using the above method, then the bound of a_{d-1} can be successfully minimized.

As the next step, while a_{d-2} can be computed as in Proposition 5.4.1, there exists a less expensive method to test for a_{d-2} bounded by $a_{d-2,max}$, after which it can be found by the method in Proposition 5.4.1. This makes sure that the resources are not wasted to compute a_{d-2} whose value does not match the condition. In principle, the method bases on the following result:

Proposition 5.4.2. *Denote by $a_{i,\mu}$ the coefficient a_i generated from the choice of m_μ, the sequence $e_{i,j}$ is defined such that*

- $e_{1,j} \equiv a_{d-1,(j,1,\ldots,1)} \pmod{p}$ *for* $j \geq 1$
- $e_{i,1} = 0$ *for* $i > 1$

- $e_{i,j} \equiv a_{d-1,(1,\ldots,1,j,1,\ldots,1)} - a_{d-1,(1,\ldots,1)} \pmod{p}$ *for* $i > 1$ *and* $j > 1$

Also, let $f_0 = \frac{n - a_d m_0^d}{p^2 m_0^{d-1}}$ *and for* $1 \leq i \leq l$, $1 \leq j \leq d$ *define*

$$f_{i,j} = -\frac{a_d d x_{i,j}}{p^2} - \frac{e_{i,j}}{p}$$

Then for each vector μ *representing a solution of* $n \equiv a_d x^d \pmod{p}$,

$$\frac{a_{d-2,\mu}}{m_0} \approx f_0 + \sum_{i=1}^{l} f_{i,\mu_i}$$

Thus, for each μ, one could test the size of $a_{d-2,\mu}$ by computing f_0 and $f_{i,j}$, after which $\frac{a_{d-2,\mu}}{m_0}$ can be compared to $\frac{a_{d-2,\max}}{m_0}$ to check whether this coefficient should be selected as a good candidate. Likewise, the pseudocode given in Algorithm 5.4.1 provides a summary of the explanations above, as can be seen in [12, Algorithm 3.6].

5.5 Constructing the factor base

As mentioned in the beginning of this chapter, a trivial method for gathering the first degree prime ideals serving as the algebraic factor base is simply to search for each p the interval $[1, p-1]$ to find r satisfying $f(r) \equiv 0 \pmod{p}$. This method is sufficient if the bound for the factor base is small, e.g., few thousands. Assume that one need to factor a large modulus n, and that the bound is required to be 10^6, then this part of the GNFS may consume remarkable efforts to be completed. Therefore, a better idea is to somehow find the roots of $f(x)$ in the corresponding finite field $\mathbb{F}_p$ in a more active manner.

In this context, since the polynomial $f(x) \in \mathbb{F}_p[x]$ has small coefficients, it is convenient to factor it into distinct, monic, linear factors whose roots can be easily retrieved. The problem now turns into finding an effective algorithm for factoring $f(x)$ over $\mathbb{F}_p$. To facilitate this idea, there exists a popular result that serves as a main principle throughout the algorithm, as shown below:

Proposition 5.5.1. *Let* $p \in \mathbb{Z}^+$ *be a prime and let* $\mathbb{F}_p$ *be the finite field of* p *elements, then* $x^p - x \in \mathbb{F}_p[x]$ *can be factored as*

$$x^p - x = \prod_{i=0}^{p} (x - i)$$

Algorithm 5.4.1: SelectSmallPolynomials

Input: A number n, degree $d \geq 4$, bound $M < \sqrt[d+1]{n}$, l, p_{max}
Output: List of m for $f(x)$ with small a_d, a_{d-1}, a_{d-2}

begin
  $\mathcal{P} \leftarrow \{\}$; `/* Set of primes` p_i `used to form` p `*/`
  $q \leftarrow 0$;
  while $q \leq p_{max}$ **do**
    $q \leftarrow NextPrime()$;
    if $q \equiv 1 \pmod{d}$ **then** $\mathcal{P} \leftarrow \mathcal{P} \cap \{q\}$;
  endw
  for $a_d \leftarrow 1$ **to** $\left(\frac{M^{2d-2}}{n}\right)^{\frac{1}{d-3}}$ **do**
    $\tilde{m} \leftarrow \sqrt[d]{\frac{n}{a_d}}$;
    $a_{d-1,\max} \leftarrow \frac{M^2}{\tilde{m}}$;
    $a_{d-2,\max} \leftarrow \left(\frac{M^{2d-6}}{\tilde{m}^{d-4}}\right)^{\frac{1}{d-2}}$;
    $\mathcal{Q} \leftarrow \{\}$;
    foreach $q \in \mathcal{P}$ **do**
      $\{r_{q,1}, \ldots, r_{q,d}\} \leftarrow Solve[n \equiv a_d x^d \pmod{q}]$;
      if $r_{q,1} \neq r_{q,2}$ **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cap \{q\}$;
    endfch
    foreach $\mathcal{P}' \subset \mathcal{Q} |\ \#\mathcal{P}' = l, p = \prod_{q \in \mathcal{P}'} q \leq a_{d-1,\max}$ **do**
      for $i \leftarrow 1$ **to** $\#\mathcal{P}'$ **do**
        $q \leftarrow \mathcal{P}'[i]$;
        for $j \leftarrow 1$ **to** d **do**
          $x_{i,j} \leftarrow r_{q,j} + q \cdot Solve[r_{q,j} + qx \equiv 0 \pmod{\frac{p}{q}}]$;
        endfor
      endfor
      $m_0 \leftarrow \tilde{m} + Solve[\tilde{m} + x \equiv 0 \pmod{p}]$;
      foreach $\mu \leftarrow (\mu_1, \ldots, \mu_l) \mid 1 \leq \mu_i \leq d$ **do**
        ▷ Compute $\{a_{d-1,\mu}\}$; `/* See Proposition 5.4.1 */`
      endfch
      ▷ Compute $\{e_{i,j}\}$; `/* See Proposition 5.4.2 */`
      $f_0 \leftarrow \frac{n - a_d m_0^d}{p^2 m_0^{d-1}}$;
      ▷ Compute $\{f_{i,j}\}$; `/* See Proposition 5.4.2 */`
      foreach $\mu \leftarrow (\mu_1, \ldots, \mu_l) \mid 1 \leq \mu_i \leq d$ **do**
        if $|\frac{a_{d-2,\max}}{m_0}| > |f_0 + \sum_{i=1}^{l} f_{i,\mu_l}|$ **then**
          ▷ Select $m = m_0 + \sum_{i-1}^{l} x_{i,\mu_i}$;
        endif
      endfch
    endfch
  endfor
end

According to this result, if $r \in \mathbb{F}_p$ is a root of $f(x)$, then $x - r$ appears in the factorization of both $f(x)$ and $x^p - x$ over $\mathbb{F}_p$. This means that the greatest common divisor (GCD) $g(x)$ of $f(x)$ and $x^p - x$ over $\mathbb{F}_p$ contains all monic, linear factors of $f(x)$. If $deg(g) \leq 2$, the process becomes trivial as finding roots of quadratic and linear polynomials has been well studied. We thus assume on the contrary that $deg(g) > 2$, and follows an idea that $g(x)$ can be further factored into smaller factors using the same strategy as was done to isolate $f(x)$.

Note that since $g(x)|x^p - x$, it can be expressed in the form $\prod_{i=0}^{k}(x - x_i)$, and hence $g(x - b) = \prod_{i=0}^{k}(x - x_i - b) = \prod_{i=0}^{k}(x - t_i)$ should also divide $x^p - x$. On the other hand, observe that

$$x^p - x = x\left(x^{p-1} - 1\right) = x\left(\left(x^{\frac{p-1}{2}}\right)^2 - 1\right) = x\left(x^{\frac{p-1}{2}} - 1\right)\left(x^{\frac{p-1}{2}} + 1\right) \text{ **}$$

Considering all the cases that can happen, then either $g(x)$ divides one of the above factors of $x^p - x$, or its factors appears in at least two of them. In the latter case, $g(x)$ can be factored by finding the GCDs between $g(x)$ and each of these factors. Note that to check if $\gcd(x, g(x)) \neq 1$, it is trivial since it is the case only if the free term of $g(x)$ is 0. If this happens, then $g(x)$ can be divided by x and the Euclidean algorithm can be implemented to find GCD between $g(x)$ and either $x^{\frac{p-1}{2}} - 1$ or $x^{\frac{p-1}{2}} + 1$. In case the factors of $g(x)$ are trivial, i.e., 1 and $g(x)$ itself, this process can be repeated with different values of b, chosen uniformly random from 0 to $p - 1$. Furthermore, by recursively executing this algorithm, all the linear factors of $g(x)$ will be found, each of which gives rise to a root of $f(x)$ over $\mathbb{F}_p$. To clearly demonstrate the overall idea, Algorithm 5.5.1 explains the method from the programming perspective, while the sub-algorithm Algorithm 5.5.2 technically describes how the recursion should be done.

**Note that since p is a prime and $p - 1$ is even, this is always the case.

Algorithm 5.5.1: FactorPolynomialInFiniteField

Input: A polynomial $f(x)$, a prime p
Output: Set R contains linear factors of $f(x)$ in $\mathbb{F}_p[x]$

```
1  begin
2      f_p(x) ← f(x) (mod p);
3      g(x) ← gcd(f(x), x^p − x) (mod p);
4      R ← {} ;                          /* Set of factors of f(x) */
5      FindFactors(g(x), R, p, 0);
6      Return R;
7  end
```

Algorithm 5.5.2: FindFactors

Input: A polynomial $g(x)$, a set R, a prime p, a number $b_p \in \mathbb{F}_p$
Output: a set R containing factors of $g(x)$ in $\mathbb{F}_p[x]$

```
1  begin
2      if deg(g) = 0 then  ▷ Terminate algorithm;
3      if deg(g) = 1 then
4          R ← R ∩ {g(x + b_p)};
5          ▷ Terminate algorithm;
6      endif
7      g_1(x) ← 1;
8      while deg(g_1) = 1|| deg(g_1) = deg(g) do
9          b ← Random(0, p − 1);
10         g_1(x) ← gcd(g(x − b), x^((p−1)/2) − 1);
11     endw
12     g_2(x) ← g(x−b)/g_1(x);
13     R ← R ∩ FindFactors(g_1(x), R, p, b + b_p);
14     R ← R ∩ FindFactors(g_2(x), R, p, b + b_p);
15     Return R;
16 end
```

5.6 The lattice sieving technique

In §2.3 and §2.4 it has been shown that the line sieving technique can be effectively used to find smooth values of the form (a, b) over a particular sieving region. The problem with this method however, emerges with the fact that the sieving part seems to dominate the total running time of the GNFS, and that this technique is wasting most of its time computing pairs

(a, b) whose values are not smooth. Consequently, a suggestion in coping with this performance issue is to find a way to isolate a wide range of these bad pairs right from the beginning and sieve only those with high probabilities of being smooth.

To further exploit this idea, the *lattice sieve* (LS) method was introduced by J. Pollard that later supersedes the line sieving algorithm in finding many smooth values in less amount of time. As an initialization, assume that the two factor bases are of close bound B_1, the LS defines a prime p in the factor base to be *small* if $p < B_0$ for fixed B_0 ranging from $0.1B_1$ to $0.5B_1$, or *medium* otherwise. Then, the sieve is repeated for each medium prime q $(B_0 < q \leq B_1)$ in the factor base. At each of these iterations, instead of sieving for every pairs (a, b) appearing in the sieving region $\mathcal{A}$ as in line sieving, the LS only considers those (a, b) whose norm can be sieved (divisible) by q. Since there exist two types of norm, i.e., $N_1(a, b)$ and $N_2(a, b)$, the first sieve considers the one with possibly smaller value. Since then, the rest of the iteration is similar to that of the line sieving.

The first problem so far is how to represent the pairs (a, b) dividing q in a efficient way so that further sieving by other primes should be convenient. As suggested by Pollard, these pairs form a lattice $L(q)$ within the (a, b) plane of the sieving region. This means that a basis of at most 2 elements can be found such that it generate every other pairs of the lattice. Assume without loss of generality that this basis, preferably reduced, consists of two points $V_1 = (a_1, b_1)$ and $V_2 = (a_2, b_2)$. As a result, if a point $(a, b) \in \mathcal{A}$ can be sieved by q, there exists a pair (c, d) such that

$$(a, b) = cV_1 + dV_2 = (ca_1 + da_2, cb_1 + db_2) \tag{5.9}$$

Note that since $\gcd(a, b) = 1$, it is necessary that $\gcd(c, d) = 1$ since $\gcd(c, d)$ is divisible by $\gcd(a, b)$, emerged from (5.9). However, since (c, d) should be used to represent (a, b), the inverse does not always hold. Indeed, it is possible that even if $\gcd(c, d) = 1$, there still exists some number k such that $\gcd(a, b) = k$. In that case, if (a, b) is smooth, then so is $(a/k, b/k)$, which gives us the coprime pair (a_1, b_1). Thus, it is sufficient to consider only coprime pairs (c, d), and in the later stage the final results will be checked for their validity.

At this point, the lattice $L(q)$ can be represented by an array of the form $A[-C \ldots C, 1 \ldots D]$ whose elements point to the values in $\mathcal{A}$ dividing q. The sieving process is then identical to that of the conventional line sieving to

find values that are smooth over two factor bases. More precisely, assume that $N_2(a, b)$ was chosen for constructing the reduced basis, then the process involves two steps, which can be done in arbitrary order:

- Sieve the values $N_2(a, b)$ corresponding to $A[c, d]$ for primes $p < q$. In this particular case, $N_2(a, b)$ is normally of the form $ka - bm$.
- Sieve the values $N_1(a, b)$ corresponding to $A[c, d]$ for all other p in the factor base.

From this moment toward, it is sufficient to use the line sieving method described in §2.3 to execute the above two steps. That being said, assume that the array A needs to be sieved for a prime p in the former step, then a shortest vector (c, d) in $L(q)$ must be found such that

$$cN_2(a_1, b_1) + dN_2(a_2, b_2) \equiv 0 \pmod{p} \tag{5.10}$$

Then, by shifting the values of c and d iteratively by p, the corresponding elements in the array A will be sieved, and the iterations terminates when $c > C$ and $d > D$. In special case, i.e., $N_2(a_1, b_1) \equiv 0 \pmod{p}$ (resp. $N_2(a_2, b_2) \equiv 0 \pmod{p}$), then for each choice of d (resp. c) the whole d-th column (resp. c-th row) will be sieved. To express this sieving method in a more formal view, Algorithm 5.6.1 addresses the simplest form of the lattice, with many of its aspects are available for improvements and optimizations when coming into practice.

Remark. To get the reduced basis of any two independent vector V_1 and V_2, an LLL reduction technique described in §4.5 can be employed to give the precise solution. On the other hand, since the dimension is 2, there exists a less expensive method to achieve an acceptable result for this task. This method resembles the Euclidean algorithm for finding the greatest common divisor. In practice, this iterative technique compares $|V_1|$ and $|V_2|$ at each iteration and reduces the length of the greater one. Assume that $|V_1| < |V_2|$ at one step, then a new V_2 can be achieved from:

$$V_2 = V_2 - \frac{V_2 \cdot V_1}{V_1 \cdot V_1} V_1$$

Finally, the loop terminates once $V_1 \cdot V_2 \approx 0$, i.e., the vectors are relatively orthogonal.

Considering the improvement in the complexity of the lattice sieve, note that since for each medium prime p, the sieve only considers those pairs (a, b)

Algorithm 5.6.1: LatticeSieve

Input: $f_1(x)$, $f_2(x)$, the bounds B_0 and B_1, a skewness s, sieving bound M, factor bases F_1 and F_2
Output: List of smooth values S

```
1  begin
2      S ← {};
3      foreach (q, r)|(q, r) ∈ F_2, q > B_0 do
4          V_1 ← (q, 0√s); V_2 ← (r, 1/√s) ;        /* Get skewed basis */
5          ReduceBasis(V_1, V_2) ;                  /* Assume that |V_1| < |V_2| */
6          C ← (M + a_2)/a_1;
7          D ← (|V_1|/|V_2|)C;
8          ▷ Initialize array A to contain 1;
9          q_1 ← B_1; q_2 ← q;
10         for i ← 1 to 2 do
11             foreach (p, r)|(p, r) ∈ F_i, p_1 < q_i do
12                 ▷ Find smallest (c, d):
                   cN_i(a_1, b_1) + dN_i(a_2, b_2) ≡ 0 (mod p);
13                 while c < C do
14                     if N_i(a_2, b_2) ≡ 0 (mod p) then
15                         for d_0 ← 1 to D do A[c, d_0] ← A[c, d_0]p;
16                     else
17                         d_0 ← d;
18                         while d_0 < D do
19                             A[c, d_0] ← A[c, d_0]p;
20                             d_0 ← d_0 + p;
21                         endw
22                     endif
23                     c ← c + p;
24                 endw
25             endfch
26         endfor
27         for c ← −C to C do
28             for d ← 1 to D do
29                 a ← ca_1 + ca_2; b ← cb_1 + cb_2 ; /* Reconstruct (a, b) */
30                 if A[c, d] = N_1(a, b)N_2(a, b) then S ← S ∩ {a, b};
31             endfor
32         endfor
33     endfch
34 end
```

whose norms dividing p, the complexity is reduced to $\frac{1}{p}$ for $B_0 < p \leq B_1$, and hence the total reduction will be the factor

$$W = \sum_{B_0 < q \leq B_1} \frac{1}{q} \approx \frac{\log\left(\frac{B_1}{B_0}\right)}{\log(B_1)}$$

Chapter 6

The GRID environment

As mentioned earlier, the GNFS algorithm was designed in a way that its implementation supports parallel computing, at least to a certain extent. Indeed, this parallel scheme can be applied during executing each step of this algorithm. Considering the polynomial selection phase, a wide range of computers can be assigned with different intervals of a_d as inputs for Algorithm 5.4.1 to produce distinct polynomials, after which a central server will be responsible for selecting the best candidate for the sieving step. Then, due to the design of the sieving techniques, the sieving region can be divided for distribution over the GRID network in the case of line sieving, or that the set of medium primes q can be splitted if one uses the lattice sieve as an alternative.

Moving toward the next phase, again each iteration of the Block Lanczos's method can be equally distributed, and results from these parallel computations are then collected and assembled to form the input for the next iteration. Even though there exist data dependencies among these steps, it is practically possible to use distributed computing, especially when the matrix size extends to millions of columns and rows. Finally, when several dependencies have been found, each of them can be used independently by the square root algorithm to check whether the corresponding roots yield non-trivial factors of n, hence making this ending process also distributable.

Installed at Rovaniemi University of Applied Sciences (RAMK) in 2005, the GRID network has been utilized in many kinds of research projects. Due to the need of computing resources by the GNFS, this chapter aims at describing a simple and standardized GRID model, namely *Condor* which is sufficient for experimenting this algorithm. In addition, this chapter also contributes a new extension in system design to further enhance Condor in managing and utilizing its features for the special case at RAMK. Otherwise,

it can also be regarded as a document supporting further developments and redesigning, should the initial architecture be examined.

6.1 An overview of Condor's operation

In the distributed computing community, Condor plays a vital role as a *de facto* model for parallel computing. Being developed at Wisconsin university, Condor acts as a resource manager for a computer system and appropriately schedules jobs execution using the computing power under its management. In the simplest form, a typical Condor's system is made up of three components: the *job submitter*, the *Condor manager*, and the *Condor pool* containing a number frequently *idle* computers called *Condor workers*, as can be seen in Figure 6.1. [31, pp. 1-4]

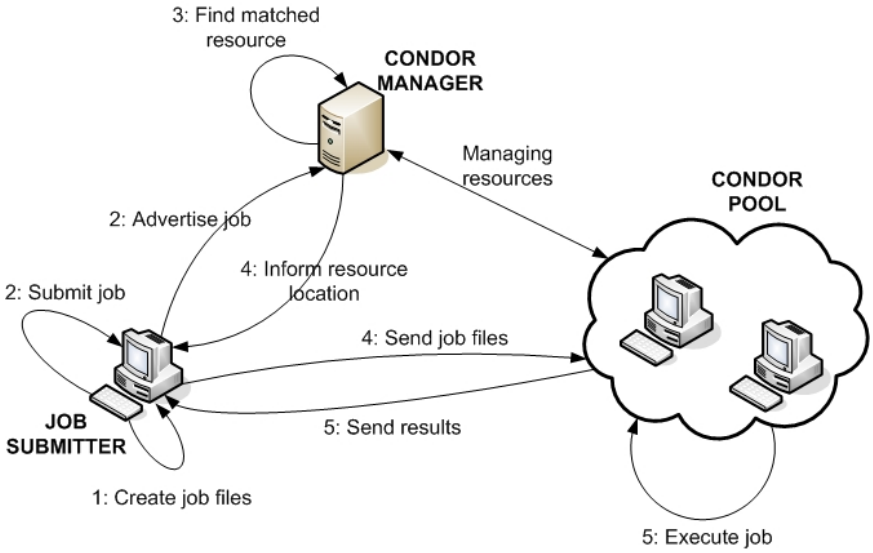

Figure 6.1: A sample of Condor network operation

In Condor's terms, a *job* is referred to as a need of executing a program for certain results. In this aspect, a proper job consists of an executing program, its input and an instruction on how it should be run. Considering the life cycle of a job within a Condor environment, the following steps show the

process from when the job is to be created until results are delivered back to the job submitter [31, pp. 27-30]:

1. The user creates the desired executing program(s) and input files. At the same time, the user should also create an instruction file which indicates how the program should be run, on what type of resources (platform, memory, etc.) should it be executed. From this point, this instruction file is referred to as the *submit file*.

2. As the user submits the submit file using a job submitter on his/her own premise, the job information will be advertised to the Condor manager for resource matching.

3. After the arrival of the job, a so-called *ClassAd* mechanism running within the Condor manager will be responsible for finding available resources in the Condor pool that match the job requirements. If no resource is found, the job is added to a queue, serving in *FIFO* manner. [25]

4. In case an available resource is found satisfying the job's conditions, the Condor manager informs the submitter on the location (e.g. IP address) of the resource, after which the submitter sends all the job files needed by the computation to the resource computer(s).

5. By receiving the job files from the submitter, the resource starts executing the job files. After the execution has finished, all results are directed back to the submitter, ending the life cycle of the job.

Beside the core functionality, Condor also provides a comfortable monitoring tools for users and system administrators to manage jobs and resources status. Indeed, in order to trace the process of the submitted jobs, the command `condor_q` can be issued directly in the submitter batch console to show job relevant information such as job ID, status, running time, memory occupation, etc. On the other hand, the Condor manager's administrator can monitor the Condor pool by issuing the `condor_status` command that shows a real time report on the status of each computer participating in the pool. Also, with the availability of other job and resource management commands, Condor achieves a number of remarkable advantages, as shown below:

- Scalability: since Condor uses as simple *client-server* structure, management of large Condor pools can be done with inexpensive computing resources. Moreover, as machines in the Condor environment operate independently of each others, a Condor pool can be expanded by simply

installing Condor instances on new machines and connecting them to the network. After that, these new resources can be effectively managed and utilized by Condor managers in the same way as for the existing ones.

- Reliability: the most important consideration in scientific computing is the ability to provide error-free computations. Condor supports this feature by having the Condor manager to check if jobs have been incorrectly executed due to system corruptions. In such cases, depending on the initial configuration, it either restarts jobs on other resources or notifies the submitters about the situation, thus preventing any further mistakes and wastes of computing power.

- Manageability: since Condor operates with highest privileges in every participating machine, it provides a great ability to control and monitor all the parameters of each resource as well as jobs executing on it. In addition, it becomes evident when Condor is scheduled to run a large program dividing into millions of jobs. While keeping user's efforts away from management issues, Condor automatically distributes these jobs for parallel computing continuously on all available resources, and returns results as they are correctly computed. In addition, it becomes evident when Condor is scheduled to run a large program dividing into millions of jobs. While keeping user's efforts away from management issues, Condor automatically distributes these jobs for parallel computing continuously on all available resources, and returns results as they are correctly computed.

- Saving computing power: it is very likely nowadays that most computer systems are not dedicated for distributed computing, such as university and company networks. The fact is that computing power in such systems is being wasted from time to time as computers are mostly in idle state, and even when they are active, many activities do not take full advantages of their capabilities. As a solution, Condor was also designed for utilizing unused resources for parallel computations, while securing acceptable resources for local system operation and other user activities.

Due to the above observations, the computer system at RAMK is seen to be suitable for deploying Condor as a mean of combining educational activities and a powerful distributed computing environment. Indeed, as the following sections show, an extension to this system is made to further facilitate the nature of research at RAMK, not only for experimenting the GNFS, but also for other scientific activities.

6.2 A projects scheduling mechanism

As explained in the previous section, the Condor software offers a powerful method for dealing with management issues of distributed computing over a large computer system. In particular, it defines the concept of *job* as the unit of computation in the GRID environment. The problem with this definition however, arises when it comes to distinguishing among jobs belonging to different computations, that is, when several programs exist, each of which consists of many jobs to be executed. In such cases, if all the jobs are submitted at the same time, most of them shall not be executed in near future due to the limitation of available resources. As a result, these jobs during their idle state would consume a huge amount of monitoring resource from the Condor manager.

To address this issue, we introduce a concept of *project* that groups jobs of the same computing purpose into a single instance. In other words, assume that a computation is divided into many jobs for distributed computing, then these jobs must be assigned to a single project as they are parts of the same computation. Moreover, instead of submitting these jobs all at once, the project only sends to Condor certain number of jobs for every pre-configured period, and terminates after all of its jobs have been correctly executed. Similar to that of a job, the following steps show how a project behaves during its life cycle:

1. The user creates the executing program and a number of jobs, each of which contains several files necessary as input for the executions. All of these files are then submitted using a method specified by the software extension to a project managing server, namely the *project controller*.

2. The project controller creates a project for these files along with a dedicated working directory and declares this project active by giving it a child *process*.

3. For every given period, e.g., 30 seconds, the project contacts the Condor manager to check how many machines are available in the Condor pool that match the job conditions specified by the user.

4. If n such resources exist, the project creates valid submit file for n pending jobs and submits them to the Condor manager. Since then, with regards to the Condor environment, the project controller plays a role of an automatic job submitter.

5. Submitted jobs are handled within Condor environment and returned as soon as they have been correctly executed. Meanwhile, the project checks for the validity of these results, and in case they are corrupted, the corresponding jobs are reset for new execution. Otherwise, as long as there exist incomplete jobs, the process returns to Step 3.

6. When all the results have been successfully received, the project formats them into an accessible form and sends the final outcome back to the user. After this task, the project ends its life cycle and hence is terminated.

In addition to the above concept, a network extension is also deployed that serves for the purposes of managing many different projects and simplifying the user interface. This extension behaves as an intermediate system connecting and administering communications from both users and the Condor environment. As Figure 6.2 shows, a Web environment suffices these conditions as it supports simple and reliable communications with users over the Internet, and at the same time with the Condor environment using a project database.

According this figure, the overall operation with respect to a project life cycle can be literally explained as follows:

1. The project is firstly created within user's premise, after that it will be advertised to the *Web server* located remotely over the Internet. At the same time, depending on the instruction from this Web server, project files are uploaded to a *project storage*.

2. Upon receipt of the submitting project, the Web server adds it to the *project database*, along with a location pointing to where its files are stored.

3. The Condor environment, or more precisely the project controller, frequently checks the project database for new projects. If an entry is found, the controller gets the corresponding project and its files specified in the database for execution.

4. The Condor environment starts executing the project until all of its jobs are correctly computed. Note that in this case several projects may be executed at a time.

5. Once the project is correctly executed, its complete status is returned to the database, along with the result sending back to a specific location in the project storage.

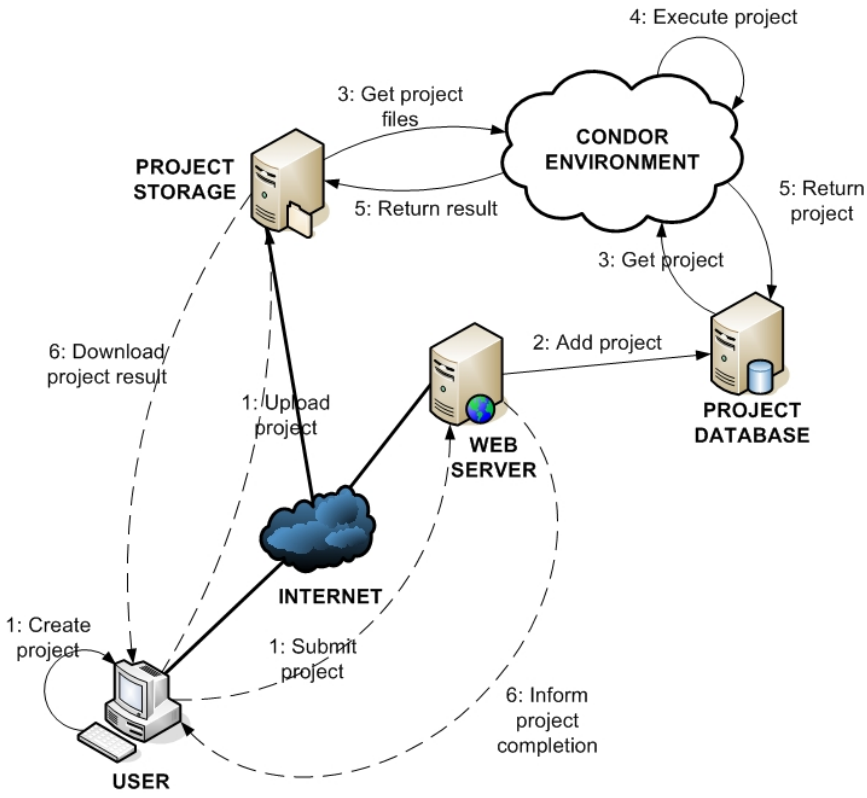

Figure 6.2: A network diagram for the Web environment

6. As the Web server notifies the user on the completion of the project, the user may download the result from a storage specified by information from the Web server.

In overall, apart from the benefit of saving computing resource for the Condor manager, the above design also provides a flexibility for making a friendly user interface to replace the direct use of batch system provided by Condor. Indeed, Web technologies can now be used to support users with simpler tools for submitting and managing their own projects. This method is also useful if one considers it as a popular and reliable communication medium over the Internet.

6.3 Optimizing data transmission

In practice, one of the most important considerations with distributed computing is the availability of network infrastructure for sending and receiving job data as well as other administrative information. As for the case of Condor, it performs network operations using a special communication protocol to connect different components together. In executing each job, after an available Condor worker has been found, all the job files will be sent from the submitter to this worker. The worker then executes the appropriate file and returns the result once the execution finishes.

To explain the importance of this aspect, it is necessary to consider the scenario in which the above execution causes significant waste of network bandwidth. Indeed, we begin this example by introducing the following concepts:

- Common job files: within a project, a job file is regarded as *common* if it appears as an input for every job. Examples of such files include execution files, common configuration files, or common input file whose existences are normally crucial for error-free execution of each job.

- Individual job files: within a project, a job file is regarded as *individual* if it appears as an input for only one job. These files play a key role in the mean of distributed computing as they are normally divided out from a large input file to serve the parallel process. Thus, if two jobs possess different individual job files, it is normally true that the results produced by them are also different, and vice versa.

Concerning the example of a project given in Figure 6.3, it is easy to see that there exist 3 different common files and 4 individual files unique for each

```
-rwxr-xr-x 1 x x  215096 2007-09-20 19:34 CommonFile1
-rwxrwxr-x 1 x x   21096 2007-09-22 18:53 CommonFile2
-rwxrwxr-x 1 x x   34096 2008-02-07 17:35 CommonFile3
-rwxr-xr-x 1 x x    2288 2006-02-11 02:07 IndividualFile1
-rwxrwxr-x 1 x x    2646 2007-11-20 17:05 IndividualFile2
-rwxrwxr-x 1 x x    7656 2008-02-06 23:47 IndividualFile3
-rwxrwxr-x 1 x x    3456 2007-11-26 20:43 IndividualFile4
```

Figure 6.3: A sample set of job files

job of the project. Assume that one million such jobs exist for this project, and that the individual files are equally splitted out of 4 large input files, then the total amount of data transmission across the network would be:

$$\begin{aligned} S_0 &= 1000000(215096 + 21096 + 34096 + 2288 + 2646 + 7656 + 3456) \\ &= 286334000000 \text{ (byte)} \approx 273069 \text{ (MB)} \end{aligned}$$

On the other hand, if the large input files are not splitted and hence only one job is to be computed, the bandwidth consumed by this project is now significantly reduced, as given by:

$$\begin{aligned} S_1 &= 215096 + 21096 + 34096 + 1000000(2288 + 2646 + 7656 + 3456) \\ &= 16046270288 \text{ (byte)} \approx 15300 \text{ (MB)} \end{aligned}$$

The ratio $S_0/S_1 \approx 17.8$ shows that if one merely submits all the jobs of the above project, even a fast network infrastructure may not be sufficient to support the transmission, and that bandwidth is being wasted for unreasonable factors. Referring to the example, such difference is caused by the fact that common files are being sent multiple times along with the jobs, whereas this should not be the case since each Condor worker has already received a copy of these files after its first contribution to this project. Following this idea, suppose that there are 1000 workers in the Condor pool, and that each of them receives exactly one copy of common files during the execution of the project, then the total bandwidth consumed would be:

$$\begin{aligned} S_2 &= 1000(215096 + 21096 + 34096) + 1000000(2288 + 2646 + 7656 + 3456) \\ &= 16316288000 \text{ (byte)} \approx 15560 \text{ (MB)} \end{aligned}$$

As the difference between S_1 and S_2 is subtle, this strategy clearly produces a remarkable improvement compared to the trivial submission method mentioned above. The remaining task is to technically deploy it while maintaining the functionality of the Condor environment. Thus, instead of using the

```
-rwxr-xr-x 1 x x  215096 2007-09-20 19:34 CommonFile1
-rwxrwxr-x 1 x x   21096 2008-02-07 17:35 CommonFile2
-rwxrwxr-x 1 x x   34096 2008-02-07 17:35 CommonFile3
drwxr-xr-x 2 x x    4096 2006-02-11 02:07 IndividualDir1
drwxr-xr-x 2 x x    4096 2006-02-11 02:07 IndividualDir2
drwxrwxr-x 2 x x    4096 2007-11-26 20:43 IndividualDir3
drwxr-xr-x 2 x x    4096 2006-02-11 02:07 IndividualDir4
```

Figure 6.4: A sample set of project files

conventional file transfer mechanism provided by Condor, we attempt to use a normal file sharing system such as NFS or Samba for on-demand transmission of job files. In addition, since the two file types need to be distinguished from the beginning of the project execution, the project files must follows a standard directory structure, of which an example of the above project can be seen in Figure 6.4. A proper explanation for this model is then followed:

- The common files are located directly in the project root directory. In fact, any file appears in this location will be categorized as common job files.
- Assume that there are n jobs for a project, then each directory appears in the project root directory is called an *individual directory*, and it must contain exactly n files. Within each individual directory, each file is assigned as an individual file for a job.
- If there exist more than one individual directory, e.g., Figure 6.4, then when a job is being structured, the project will pick up from each individual directory an individual job file with directory index corresponding to that of the job. As an example, each individual file in Figure 6.3 is taken from each individual directory presented in Figure 6.4.

On the other hand, note that since Condor was not designed to support this feature, each component of the Condor environment must also be equipped with a special program in order to perform this task. In particular, while the project controller is scheduled to periodically submit as many jobs as possible, each Condor worker also needs to check whether it should download the common files before executing any job. In fact, within each worker, a separate hard disk space is used to temporarily store common files of those projects to which the worker has contributed. In addition, from

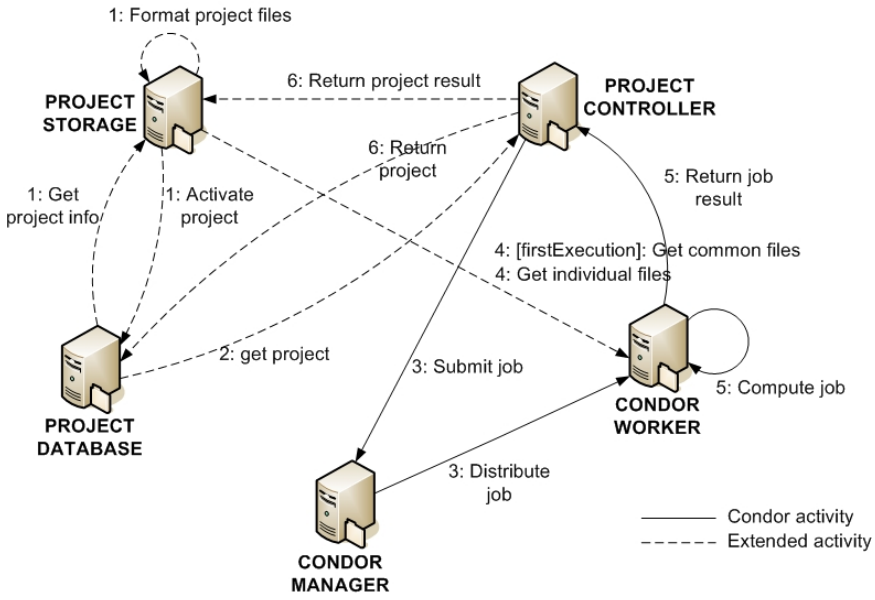

Figure 6.5: Modified project life cycle

the very beginning of each project submission, project files should arrive the project storage under a compressed form, and hence there must also exists a service running on this storage server to validate the structure of these files before the project can be activated for computation in the Condor environment. Assume that a project has already been added to the database and the project storage, the above observations lead to a network design illustrated by Figure 6.5, in which the communication among different machines can be literally described as follows:

1. The special service on the project storage queries the database for information regarding new projects. Based on the responses, it then validates the structure of the project files. If the structure matches the standard specified by Figure 6.4, the project is activated in the database so that it is available for computation.

2. The project controller gets the project from the database and prepares for its job submission routine.

3. Within the project execution process, each job is submitted to the

Condor manager, after which it is distributed to an appropriate Condor worker for remote computation.

4. Upon receipt of the job information, the Condor worker downloads the corresponding individual files from a location specified by the job information. In addition, if the common files of the project do not appear in the local storage, the worker downloads the common files from the storage.

5. Once all the necessary files are presented, the job is executed, after which its result is sent back to the project controller.

6. After the project has been fully computed with all of its jobs executed, the project result is returned to the project storage and its completed status is returned to the project database, ending the active period of the project.

Compared to the model presented by Figure 6.1, it is clear that this final design opts a more complicated structure when it comes to an implementation. However, for the reason described above, this complexity greatly reduces the minimum requirement concerning the capacity of the network infrastructure. Consequently, it allows the computer system at RAMK to be used as a distributed computing environment. As an example, [11, pp. 12-15, 25] presents a breakthrough result containing 200 new abelian square-free endomorphisms (resulting to a powerful substitution over four letters) produced by this system during computations for combinatorics on words research. In fact, it has opened new opportunities for a number of research at RAMK to continue their developments with the support of massive computing power.

Chapter 7

An empirical conclusion

As described in chapters 2 to 5, the GNFS algorithm involves several steps toward the factorization of any composite integer of general form in $\mathbb{Z}$. In particular, it begins by selecting a good polynomial $f_1(x)$ that forms the field of fraction $\mathbb{Q}[x]/(T(x))$ for some monic version $T(x)$ of $f_1(x)$, i.e., $T(x) = a_d^{d-1} f_1(x/a_d)$, as can be seen in Proposition 4.1.1. Moreover, the polynomial selection process should also make sure that $f_1(m) \equiv f_2(m) \equiv 0 (\text{mod } n)$ where $f_2(x) = px - m$, of which an explanation is located in §5.4.

After the best polynomial has been selected with both good root and size properties, the algorithm turns into finding a number of pairs $(a, b) \in \mathbb{Z}^2$ such that the factors of $N_1(a, b) = b^d f_1(a/b)$ and $N_2(a, b) = b^d f_2(a/b)$ are bounded to some pre-defined algebraic and rational factor bases, respectively. For this purpose, §2.3 and §2.4 present a line sieving method for collecting such pairs, whereas §5.6 gives a more effective and faster technique, known as the lattice sieve as it makes use of lattice bases in searching the pairs. As soon as a set of pairs is found whose cardinality is greater than that of the factor bases together with an additional *quadratic character base* (QCB), the description in §3.3 helps to form a matrix equation that can be solved using the *Gaussian elimination* method, or preferably the *Block Lanczos's algorithm* if the matrix is of large size.

Once a non-trivial solution (dependency) of this equation is computed which indicates which pairs are to be picked up, Montgomery's square root algorithm can be invoked to multiply these pairs together and produce an algebraic square root β of the equation (2.2). Finally, by mapping β to $x \in \mathbb{Z}$, (2.3) shows that a congruence of square is found such that the factors of n can hopefully be computed from $\gcd(n, x - y)$ and $\gcd(n, x + y)$. If the value given is trivial, either a different dependency from the matrix equation is

Table 7.1: Polynomials resulted from *base-m* method

$f_1(x)$	$f_2(x)$
$25 + 7x + 2x^2 + 12x^3$	$-38 + x$
$9 + 11x + 2x^2 + 13x^3$	$-37 + x$
$35 + 18x + 6x^2 + 14x^3$	$-36 + x$
$3 + 4x + 15x^2 + 15x^3$	$-35 + x$
$26 + 18x + 13x^2 + 18x^3$	$-33 + x$
$11 + 4x + 6x^2 + 20x^3$	$-32 + x$

used, or the algorithm is restarted with a new instance of $f(x)$.

To ensure clear explanation of the whole process, this concluding chapter dedicates in giving an extended snapshot with simple input to illustrate the computations throughout the GNFS algorithm that factors the number 661643. In addition, the conclusion provides brief description on the development of integer factorization and possible trends for the GNFS.

7.1 Initialization of the algorithm

As the beginning of the algorithm, the polynomial selection phase is divided into three distinct steps, implemented using *Mathematica*, a well known *computer algebra system*. The first step involves a primitive search of polynomials $f(x)$ using either the *base-m* method or Kleinjung's trick in Algorithm 5.4.1. Since $n = 661643$ is small, the chosen degree for the polynomial is $d = 3$, and thus the Kleinjung's method is skipped out as it is more suitable for $d \geq 4$. In addition, while the interval for a_d is $[12, 20]$, the $base - m$ method produces several polynomials, as shown in Table 7.1.

Moving a step further, Algorithm 5.2.1 is implemented that optimizes the size properties of each polynomial in Table 7.1, after which it sieves through the variations of each such polynomial with sieving region $[J_0 \times J_1] = [10 \times 10]$ and a sample of the first 150 primes to find those with $\alpha < -1$. After 100 polynomials have been found, the final step computes the rating for each polynomial, sorts the list in descending order, and choose the first one. As illustrated in Table 7.2, an excerpt of the finding indicates that $f_1(x) = 25 + 7x + 2x^2 + 12x^3$ is chosen as one of its variation arrives at the rating of 986. With the optimized selection of $(c_0, c_1, t) = (0, 1, 1)$ and the pair $(j_0, j_1) = (1, -3)$, the optimized version of this polynomial then follows

Table 7.2: Final list of optimized polynomials

$f_1(x)$	s	c_0	c_1	t	(j_0, j_1)	α	Rating
$-70 + 119x - 36x^2 + 12x^3$	0.839476	0	1	1	(-2,-3)	-1.92051	902
$47 + 116x - 36x^2 + 12x^3$	0.839476	0	1	1	(1,-3)	-1.1785	986
$9 + 48x + x^2 + 13x^3$	0.778176	0	1	0	(0,-2)	-1.34214	955
$120 + 45x + x^2 + 13x^3$	0.778176	0	1	0	(3,-2)	-2.81245	706
$-28 + 12x + 2x^2 + 13x^3$	0.778176	0	1	0	(-1,-1)	-1.82316	890
$-65 - 24x - 34x^2 + 14x^3$	1.07131	0	1	1	(-2,1)	-1.11697	970
$-28 - 99x - 32x^2 + 14x^3$	1.07131	0	1	1	(-1,3)	-1.39869	881

Table 7.3: Algebraic factor base A

(r, p)	(r, p)	(r, p)	(r, p)	(r, p)	(r, p)	(r, p)
$(2, 2)$	$(7, 13)$	$(11, 19)$	$(21, 43)$	$(62, 83)$	$(18, 107)$	$(1, 139)$
$(2, 3)$	$(10, 13)$	$(9, 23)$	$(0, 47)$	$(76, 89)$	$(84, 107)$	$(16, 139)$
$(3, 3)$	$(12, 13)$	$(4, 37)$	$(36, 53)$	$(37, 97)$	$(42, 109)$	$(125, 139)$
$(3, 5)$	$(15, 17)$	$(29, 41)$	$(34, 71)$	$(57, 101)$	$(18, 113)$	
$(2, 7)$	$(4, 19)$	$(9, 43)$	$(70, 73)$	$(61, 103)$	$(114, 127)$	
$(2, 11)$	$(7, 19)$	$(16, 43)$	$(3, 79)$	$(8, 107)$	$(43, 137)$	

with $f_1(x) = 47 + 116x - 36x^2 + 12x^3$ and $f_2(x) = -39 + x$, i.e., $m = 39$ and $p = 1$.

After selecting the appropriate polynomial, choices of factor bases must also be made before the sieving step takes place. Based on the size of n, the algebraic and rational factor bases can be generated with primes up to the bound $M = 150$ using a generator written in *Mathematica*. As shown in Table 7.3 and Table 7.4, the cardinalities of these factor bases are 39 and 35, respectively. This implies that the sieving step must find at least 66 pairs (a, b) which are smooth over A and F to make sure that the matrix equation will give non trivial solutions.

In addition to the generation of the factor bases above, the generator is also responsible for computing the list of natural logarithm points to initialize the sieving array. In particular, this must be done for each value of b since it has been decided that the line siever is to be used as the sieving technique. As mentioned previously, the program computes this list by first finding the critical points of the norm function. Then, it constructs the list so that the

Table 7.4: Rational factor base F

(r,p)	(r,p)	(r,p)	(r,p)	(r,p)	(r,p)	(r,p)
$(1,2)$	$(0,13)$	$(8,31)$	$(39,53)$	$(39,73)$	$(39,101)$	$(39,127)$
$(0,3)$	$(5,17)$	$(2,37)$	$(39,59)$	$(39,79)$	$(39,103)$	$(39,131)$
$(4,5)$	$(1,19)$	$(39,41)$	$(39,61)$	$(39,83)$	$(39,107)$	$(39,137)$
$(4,7)$	$(16,23)$	$(39,43)$	$(39,67)$	$(39,89)$	$(39,109)$	$(39,139)$
$(6,11)$	$(10,29)$	$(39,47)$	$(39,71)$	$(39,97)$	$(39,113)$	$(39,149)$

difference in logarithm between any two continuous points is less than $\ln 2$. To further demonstrate this idea, Table 7.5 gives an example of the list with $b = 2$ and a ranging from -50000 to 50000 for the algebraic sieve. Also, this computation terminates the initialization phase for the main sieving step, of which details are given in the next section.

7.2 Sieving and forming the matrix equation

The sieving part takes place by using the idea of the line sieving technique described in §2.3 and §2.4. In particular, note that since it is possible to generate a polynomial $f_2(x)$ for the rational part, the sieve for the rational values can be generalized as in the algebraic side. That being said, it is sufficient to use only Algorithm 2.4.1 for both sieves and omit the implementation of Algorithm 2.3.1. Considering the sieving region, an original attempt shows that the sieving region $\mathcal{A} = [-50000, 50000] \times [1, 200]$ is sufficient for finding enough smooth pairs. This also implies that 400 samples of logarithm lists must be given for both rational and algebraic sieve, of which an example is presented in Table 7.5.

Technically speaking, the sieving program was implemented in C# using .NET library for *Windows* programming. Furthermore, the experiment shows that the time consumption in a 1500GHz laptop was less than 4 minutes for this search. After the sieving part, 635 pairs were found, whereas some of which are given in Table 7.6. In fact, only these pairs were considered for constructing the matrix equation. Additionally, note that in order to compensate the shortcomings of $\mathbb{Z}[\theta]$, it is necessary that a QCB is structured that adds several dimensions to each vector of the matrix equation. For this purpose, the QCB is constructed with 20 elements, each of which is of the form (s, q) such that q is prime, $f(s) \equiv 0 \pmod{q}$, $f'(s) \not\equiv 0 \pmod{q}$, and $a - bs \not\equiv 0 \pmod{q}$ for all pairs (a, b) in Table 7.6. To satisfy the first two

Table 7.5: List of log points for algebraic sieve with $b = 2$

$(a, \ln N_1(a,2))$	$(a, \ln N_1(a,2))$	$(a, \ln N_1(a,2))$	$(a, \ln N_1(a,2))$	$(a, \ln N_1(a,2))$
$(-50000, -34.94)$	$(-621, -21.79)$	$(-8, -9.55)$	$(68, 15.06)$	$(5211, 28.16)$
$(-39685, -34.25)$	$(-493, -21.1)$	$(-6, -8.93)$	$(85, 15.75)$	$(6564, 28.85)$
$(-31498, -33.56)$	$(-391, -20.4)$	$(-5, -8.56)$	$(106, 16.42)$	$(8269, 29.54)$
$(-25000, -32.87)$	$(-310, -19.71)$	$(-4, -8.13)$	$(133, 17.11)$	$(10417, 30.28)$
$(-19843, -32.17)$	$(-246, -19.03)$	$(-3, -7.59)$	$(167, 17.8)$	$(13124, 30.93)$
$(-15749, -31.48)$	$(-195, -18.34)$	$(1, 6.66)$	$(209, 18.48)$	$(16534, 31.62)$
$(-12500, -30.79)$	$(-155, -17.65)$	$(3, 7.28)$	$(262, 19.17)$	$(20831, 32.32)$
$(-9921, -30.1)$	$(-123, -16.97)$	$(5, 7.78)$	$(329, 19.86)$	$(26244, 33.01)$
$(-7874, -29.4)$	$(-98, -16.30)$	$(7, 8.35)$	$(414, 20.55)$	$(33064, 33.70)$
$(-6250, -28.71)$	$(-78, -15.64)$	$(9, 8.92)$	$(521, 21.24)$	$(41657, 34.4)$
$(-4961, -28.02)$	$(-62, -14.97)$	$(11, 9.45)$	$(655, 21.93)$	$(50000, 34.94)$
$(-3938, -27.32)$	$(-49, -14.29)$	$(13, 9.93)$	$(824, 22.62)$	
$(-3126, -26.63)$	$(-39, -13.64)$	$(16, 10.56)$	$(1037, 23.31)$	
$(-2481, -25.94)$	$(-31, -12.1)$	$(20, 11.25)$	$(1306, 24.01)$	
$(-1969, -25.24)$	$(-25, -12.4)$	$(24, 11.82)$	$(1644, 24.7)$	
$(-1563, -24.55)$	$(-20, -11.8)$	$(29, 12.41)$	$(2070, 25.39)$	
$(-1241, -23.86)$	$(-16, -11.22)$	$(36, 13.09)$	$(2607, 26.08)$	
$(-985, -23.17)$	$(-13, -10.7)$	$(44, 13.71)$	$(3284, 26.77)$	
$(-782, -22.48)$	$(-10, -10.06)$	$(55, 14.41)$	$(4137, 27.47)$	

Table 7.6: Found pairs smooth over A and F

(a,b)	(a,b)	(a,b)	(a,b)	(a,b)	(a,b)	(a,b)
$(-586,1)$	$(8,1)$	$(61,2)$	$(88,3)$	$(342,4)$	$(73,6)$	$(37,8)$
$(-313,1)$	$(9,1)$	$(85,2)$	$(94,3)$	$(386,4)$	$(133,6)$	$(69,8)$
$(-145,1)$	$(16,1)$	$(141,2)$	$(-909,4)$	$(-67,5)$	$(215,6)$	$(109,8)$
$(-94,1)$	$(18,1)$	$(171,2)$	$(-433,4)$	$(-59,5)$	$(1411,6)$	$(171,8)$
$(-70,1)$	$(23,1)$	$(193,2)$	$(-355,4)$	$(-41,5)$	$(-88,7)$	$(341,8)$
$(-53,1)$	$(125,1)$	$(-395,3)$	$(-141,4)$	$(-23,5)$	$(-46,7)$	$(355,8)$
$(-42,1)$	$(145,1)$	$(-83,3)$	$(-75,4)$	$(-18,5)$	$(-2,7)$	$(394,8)$
$(-37,1)$	$(-87,2)$	$(-55,3)$	$(-13,4)$	$(-17,5)$	$(11,7)$	$(-2347,9)$
$(-27,1)$	$(-51,2)$	$(-31,3)$	$(-3,4)$	$(-8,5)$	$(19,7)$	$(-100,9)$
$(-19,1)$	$(-29,2)$	$(-16,3)$	$(1,4)$	$(-1,5)$	$(20,7)$	$(-52,9)$
$(-14,1)$	$(-25,2)$	$(-8,3)$	$(9,4)$	$(21,5)$	$(36,7)$	$(-40,9)$
$(-13,1)$	$(-11,2)$	$(-5,3)$	$(15,4)$	$(24,5)$	$(47,7)$	$(-26,9)$
$(-12,1)$	$(-5,2)$	$(2,3)$	$(35,4)$	$(58,5)$	$(65,7)$	$(-1,9)$
$(-3,1)$	$(-3,2)$	$(4,3)$	$(41,4)$	$(94,5)$	$(96,7)$	$(7,9)$
$(-2,1)$	$(3,2)$	$(5,3)$	$(47,4)$	$(99,5)$	$(146,7)$	$(137,9)$
$(0,1)$	$(7,2)$	$(14,3)$	$(59,4)$	$(131,5)$	$(-83,8)$	$(232,9)$
$(2,1)$	$(11,2)$	$(17,3)$	$(127,4)$	$(206,5)$	$(-11,8)$	$(557,9)$
$(3,1)$	$(13,2)$	$(20,3)$	$(170,4)$	$(647,5)$	$(9,8)$	$(4957,9)$
$(4,1)$	$(19,2)$	$(31,3)$	$(197,4)$	$(-31,6)$	$(23,8)$	$(-541,10)$
$(7,1)$	$(41,2)$	$(50,3)$	$(282,4)$	$(47,6)$	$(27,8)$	$(-141,10)$

and the last conditions, we simply repeat the generator for finding algebraic primes exceeding the bound $M = 150$ set in the previous section. It remains then the test for the third condition to match until 20 of such primes are found. As a result, Table 7.7 shows those primes that were used in testing the squareness of the solution.

On the other hand, note that in §4.1 we observe the existence of exceptional prime ideals which may appear in the ideal factorization of each ideal formed by pairs (a,b). Moreover, this is especially the case when $a_d > 1$, and that those ideals need also to be considered to make sure that Montgomery's square root is valid. In order to cope with this additional requirement, such exceptional prime ideals must be found, after which their valuations against each pairs (a,b) have to be computed as additional dimensions for vector columns of the matrix equation.

Table 7.7: Algebraic primes in the quadratic character base

(s, q)	(s, q)	(s, q)
$(531377465, 1072693261)$	$(619630789, 1072693417)$	$(354766173, 1072693541)$
$(738039001, 1072693289)$	$(466592492, 1072693417)$	$(397002423, 1072693541)$
$(195059167, 1072693327)$	$(446695841, 1072693459)$	$(769228135, 1072693571)$
$(348109122, 1072693339)$	$(39570017, 1072693477)$	$(1010915920, 1072693607)$
$(124071865, 1072693367)$	$(92866501, 1072693483)$	$(350937741, 1072693607)$
$(417981257, 1072693409)$	$(12454749, 1072693493)$	$(783533556, 1072693607)$
$(1059163556, 1072693417)$	$(320924948, 1072693541)$	

Table 7.8: List of exceptional prime ideals

p	α	$N(\mathfrak{p})$
2	$\hat{\theta} + \hat{\theta}^2$	2
3	$\hat{\theta} + \hat{\theta}^2$	3
3	$2 + \hat{\theta}^2$	3

As suggested in [21, p. 8], these ideals can be computed by first constructing the ring of algebraic integers $\mathfrak{O}$, represented in its HNF form. After that, the *Buchmann-Lenstra* method can be used to decompose this maximal order into exceptional prime ideals. While the description of finding the integral basis for $\mathfrak{O}$ can be seen in §4.3, [5, p. 312-322] explains how the decomposition process takes place. In this case, an implementation of Algorithm 4.3.1 in the GGNFS gives the following representative of $\mathfrak{O}$ after continuously enlarging the order $\mathbb{Z}[\hat{\theta}]$ by the list of primes $\{2, 3\}$ as the square factors of $disc(T) = -14356230912$:

$$\boldsymbol{W}_{\mathfrak{O}} = \begin{pmatrix} 12 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad with \quad W_d = 12$$

Following this result, the decomposition process appears to successfully compute three distinct exceptional primes. As shown in Table 7.8, each these ideals are represented by two elements, namely a prime p and an algebraic integer α that it lies above, i.e., $\mathfrak{p} = p\mathfrak{O} + \alpha\mathfrak{O}$ [5, p. 194]. At this stage, it is sufficient to construct the matrix equation, with 3 additional dimensions to that in §3.3 to activate the exceptional prime ideals.

Table 7.9: Primes to be omitted from the matrix equation

Rational primes	Algebraic primes	Algebraic primes
$(39, 139)$	$(2, 2)$	$(16, 43)$
$(39, 149)$	$(2, 3)$	$(21, 43)$
	$(3, 3)^*$	$(18, 107)$
	$(10, 13)$	$(84, 107)$
	$(12, 13)$	$(16, 139)$
	$(11, 19)$	$(125, 139)$

Table 7.10: Sample vector representative of pairs (a, b)

(a, b)	$\{bf_2(\frac{a}{b}) < 0, [v_{p_i}(bf_2(\frac{a}{b}))], [l_{\mathfrak{p}_i}(a + b\theta)], [l_{\mathfrak{p}\mid\mathfrak{O}}(a + b\theta)], [\chi_{q_i}(a + bs_i)]\}$
$(-586, 1)$	{1, 00000000000000000000000000000000000000, 000010000000000110 00000000000000000000, 001, 10100101100000101101}
$(-313, 1)$	{1, 10001000000000000000000000000000000000, 000000000000010000 00000100000100000000, 011, 11110100001110011000}
$(-145, 1)$	{1, 10000000100000000000000000000000000000, 000010000001000000 00000000110000000000, 011, 11001001000011101111}

Also, as a simple trick to reduce the work required for solving the matrix equation, note that a prime 139 in F does not appear in factorizations of any pairs (a, b) given in Table 7.6. This means that no matter which pairs are to be chosen, the valuation of 139 in the final product will remain the same as 0, and hence the corresponding dimension can be skipped as it does not affect any selection. This observation also holds for the primes given in Table 7.9, which returns the total number of dimensions of the column vectors as $\dim(v) = 1 + \#F + \#A + \#Q + 3 - 14 = 84$. As Table 7.10 shows, each vector representing each pair (a, b) are reduced modulo 2 since the selection process only considers whether an entry is odd or even.

After binary vectors for those pairs (a, b) in Table 7.6 have been formed, they are joined together as column vectors of the matrix $\boldsymbol{B}$. As explained in Chapter 3, the next step invokes the Block Lanczos's algorithm to find a solution for the equation $\boldsymbol{BX} \equiv 0 \pmod 2$. Consequently, this process returns 55 vectors (dependencies) as the basis for the nullspace of $\boldsymbol{B}$ over $\mathbb{F}_2$,

*Note that $(2, 2), (2, 3)$, and $(3, 3)$ did appeared in the factorization of the pairs in Table 7.6, but since they are covered by the exceptional prime ideals, they are now cancelled.

Table 7.11: 28th dependency D in the nullspace of $\boldsymbol{B}$

$(a,b)^e$	$(a,b)^e$	$(a,b)^e$	$(a,b)^e$	$(a,b)^e$
$(-94,1)^1$	$(7,1)^1$	$(-25,2)^1$	$(14,3)^1$	$(-141,4)^{-1}$
$(-53,1)^{-1}$	$(9,1)^{-1}$	$(-3,2)^{-1}$	$(17,3)^1$	$(1411,6)^1$
$(-27,1)^1$	$(16,1)^1$	$(11,2)^1$	$(50,3)^1$	
$(-19,1)^{-1}$	$(-87,2)^1$	$(171,2)^{-1}$	$(88,3)^1$	
$(3,1)^{-1}$	$(-51,2)^{-1}$	$(-55,3)^{-1}$	$(94,3)^{-1}$	
$(4,1)^1$	$(-29,2)^{-1}$	$(2,3)^{-1}$	$(-433,4)^1$	

each of which contains a number of pairs chosen from the list in Table 7.6.

7.3 Finding possible square roots

Among 55 dependencies found from the previous step, some selections may give trivial factors of the modulus n after the algebraic square has been found. Thus, the square root algorithm must be applied to each dependency until a non trivial one is found, which then triggers the termination of the whole algorithm with a successful result. To produce the final solution for this example, an implementation of the square root algorithm in the GGNFS project was slightly modified so that it omits large prime variations used in general cases. As a result, the experiment shows that the dependency given in Table 7.11 finally returns the prime factors of 661643.

Indeed, following Algorithm 4.2.1, the first computation is to use the *greedy strategy* to decide which pairs are to be pulled down to the denominator, i.e., the valuation of the square at them change from 1 to -1. As those exponents in Table 7.11 illustrate, 12 out of 26 pairs of the dependency are inverted to simplify the algebraic square. Then, as the algorithm turns to finding approximation for the square root, it iteratively chooses a number of δ_l that contain factors of the square root. In particular, this process is repeated twice that produces two approximations as shown in Table 7.12.

Furthermore, as shown in the end of Algorithm 4.2.1, there still exists a leftover square α that needs to be factored. In order to find the square root of α, it is necessary to firstly construct its polynomial representative in θ . This is where the *Chinese Remainder Theorem* comes into place as it makes use of several primes larger than 2^{32}. Moreover, following Algorithm 4.6.1, the residue of α at each of these primes can be computed by multiplying

Table 7.12: Table of approximations δ_l needed

s_l	δ_l	HNF form of I_l
1	$\delta_1 = 10296 - 3024\theta + 996\theta^2$	$\begin{pmatrix} 804569003 & 240718291 & 172462067 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
-1	$\delta_2 = 1128 + 168\theta - 168\theta^2$	$\begin{pmatrix} 22985820 & 361524 & 7290264 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$

Table 7.13: Inert primes with residues of α

p	$\alpha \pmod p$
6,000,000,007	$19576 + 39552\theta + 2160000000000\theta^2$
6,000,000,277	$19576 + 39552\theta + 6480000296640\theta^2$
6,000,000,289	$19576 + 39552\theta + 6480000309600\theta^2$
6,000,000,341	$19576 + 39552\theta + 6480000365760\theta^2$
6,000,000,511	$19576 + 39552\theta + 2160000181440\theta^2$
6,000,000,569	$19576 + 39552\theta + 6480000612000\theta^2$

together all pairs (a, b) in Table 7.11 and approximations δ_i in Table 7.12, as the result is given in Table 7.13. By combining these residues together to the larger finite field using (4.9), the actual value of α can be expressed as follows:

$$\alpha(\theta) = 19576 + 39552\theta - 252\theta^2 \tag{7.1}$$

Since $\alpha \notin \mathbb{Z}$, it is not trivial to find its square root, and hence Cipolla's algorithm must be applied as an alternative. Note that in this case $\hat{\theta}$ is used instead of θ since $f_1(x)$ is not irreducible in $\mathbb{F}_{p^d}$ and the monic version of $f_1(x)$ must be used, that is, $T_1(x) = a_d^{d-1} f_1(x/a_d)$. This would mean that (7.1) must be turned into:

$$\begin{aligned} \alpha(\hat{\theta}) &= 19576 + 39552\theta - 252\theta^2 \\ &= \frac{1}{144}(2818944 + 474624\hat{\theta} - 252\hat{\theta}^2) \\ &= \frac{1}{4}(78304 + 13184\hat{\theta} - 7\hat{\theta}^2) \end{aligned} \tag{7.2}$$

In order to facilitate Cipolla's algorithm, α needs to be represented with integer coefficients. This can be done by multiplying (7.2) with the denominator

W_d of $\mathfrak{O}$ to eliminate the denominator 1/4. In fact, the multiplier should be W_d^2 to make sure that the square is retained. This makes a new version of α as

$$\gamma = W_d^2\alpha = \frac{144}{4}(78304 + 13184\hat{\theta} - 7\hat{\theta}^2) = 2818944 + 474624\hat{\theta} - 252\hat{\theta}^2$$

According to the description in Algorithm 4.6.2, this method starts by selecting a large enough prime $p \in \mathbb{Z}$ and a random polynomial $t(\hat{\theta}) \in \mathbb{F}_{p^d}$ such that $t^2 - 4\alpha(\hat{\theta})$ is not a square in $\mathbb{F}_{p^d}$. As the experiment shows, the following example gives an appropriate choice for these parameters:

$$\begin{aligned} p &= 533928446631273161556097, \\ t(\hat{\theta}) &= 1189641421 + 10252023621\hat{\theta} + 1350490027\hat{\theta}^2 \end{aligned}$$

With these initial values, the square root of γ can be retrieved using Cipolla's idea, as the following manipulation shows:

$$\begin{aligned} \sqrt{\gamma} &= x^{\frac{p^d+1}{2}} \pmod{(p, x^2 - t(\hat{\theta})x + \alpha(\hat{\theta}))} \\ &= 240 - 138\hat{\theta} + 2\hat{\theta}^2 \end{aligned}$$

Consequently, the square root of α is computed by dividing out $\sqrt{\gamma}$ by W_d, as shown below:

$$\begin{aligned} \sqrt{\alpha} &= \frac{1}{12}(240 - 138\hat{\theta} + 2\hat{\theta}^2) \\ &= \frac{1}{12}(240 - 138(12\theta) + 2(12\theta)^2) \\ &= 20 - 138\theta + 24\theta^2 \end{aligned}$$

Thus, the square root β in (2.2) of the algebraic square formed by multiplying together values in Table 7.11 can be computed as the product of those approximations in Table 7.12 and $\sqrt{\alpha}$, i.e.,

$$\begin{aligned} \beta &\equiv \delta_1\delta_2^{-1}\sqrt{\alpha} && \pmod{f} \\ &\equiv \frac{(10296 - 3024\theta + 996\theta^2)(20 - 138\theta + 24\theta^2)}{1128 + 168\theta - 168\theta^2} && \pmod{f} \\ &\equiv \frac{747642 - 237936\theta + 38328\theta^2}{1128 + 168\theta - 168\theta^2} && \pmod{f} \\ &\equiv (747642 - 237936\theta + 38328\theta^2)\left(\frac{190997}{212618835} - \frac{3955\theta}{28349178} + \frac{4228\theta^2}{70872945}\right) && \pmod{f} \\ &\equiv \frac{1381607087}{1915485} - \frac{26102477}{127699}\theta + \frac{33251188}{638495}\theta^2 && \pmod{f} \end{aligned}$$

Using the homomorphism ϕ mentioned in Proposition 2.1.1, the image of this algebraic square root in $\mathbb{Z}$ can be computed as follows:

$$\begin{aligned} x = \phi(\beta) &\equiv \frac{1381607087}{1915485} - \frac{26102477}{127699}39 + \frac{33251188}{638495}39^2 \pmod{661643} \\ &\equiv \frac{137836828886}{1915485} \pmod{661643} \\ &\equiv 137836828886 \cdot 243519 \pmod{661643} \\ &\equiv 590918 \pmod{661643} \end{aligned}$$

On the other hand, based on those pairs and exponents given in Table 7.11, the rational square root can be easily computed in the following manner:

$$\begin{aligned} y &\equiv \sqrt{\prod_{(a_i,b_i)\in D} (a_i - b_i 39)^{e_i}} \pmod{661643} \\ &\equiv \sqrt{\frac{5^2 \cdot 7^2 \cdot 11^2 \cdot 19^2 \cdot 67^2 \cdot 103^2}{3^{10} \cdot 23^2 \cdot 43^2}} \pmod{661643} \\ &\equiv \frac{5^1 \cdot 7^1 \cdot 11^1 \cdot 19^1 \cdot 67^1 \cdot 103^1}{3^5 \cdot 23^1 \cdot 43^1} \pmod{661643} \\ &\equiv \frac{50480815}{240327} \pmod{661643} \\ &\equiv 50480815 \cdot 44925 \pmod{661643} \\ &\equiv 420503 \pmod{661643} \end{aligned}$$

Finally, the prime factors of $n = 661643$ can be computed using the idea of congruence of squares described in §1.4. Indeed, as the following expressions show, the computed primes are clearly the non trivial factors of n:

$$\begin{aligned} p &= \gcd(590918 - 420503, 661643) = 541 \\ q &= \gcd(590918 + 420503, 661643) = 1223 \end{aligned}$$

7.4 The development of integer factorization

As the previous example shows, the GNFS has been proven for its validity in factoring integers of general forms. To conclude on behalf of this algorithm, this final section aims at giving an overview of the current development in the field of integer factorization, and particularly of the Number Field Sieve methods.

As mentioned in §1.4, one of the most prominent application of integer factorization is in cryptanalysis. In particular, it provides an ability to break

the main security principle of the RSA algorithm, the best known public key encryption method. In addition, several aspects of applied mathematics also benefit from integers that come along with their primes factors. As an example, if prime factors of an arbitrarily composite integer n are available, then it is simple to compute the *completely multiplicative function* on n, i.e., an arithmetic homomorphism $f(x)$ that is defined on $\mathbb{Z}$.

As for recent achievements that facilitate these applications, factoring algorithms are normally divided into two categories, namely special-purpose and general-purpose algorithms. Concerning the former type, algorithms of this category aim at factoring integer satisfying certain conditions. As a typical example, *Pollard's* ρ method described in [6, pp. 896-901] assumes that the least prime factor of n is relatively small since its performance is strongly dependent of this factor. Similarly, the Special Number Field Sieve (SNFS) method behaves in the same manner as the GNFS, except that it requires the modulus n to be of the form $r^e \pm s$ with relatively small r and s.

From the other perspective, general-purpose factoring algorithms possess a major advantage over the formers as they do not apply any restriction on the form of integers to be factored, i.e., they treat integers in a completely general manner. As shown in Chapter 1, several representatives for this category include the Quadratic Sieve as well as Dixon's algorithm, of which a number or records have been made within the past 20 years. Nevertheless, the fastest ever created algorithm of this kind is Shor's algorithm [9] that factors integers in polynomial time with complexity $\mathcal{O}((\ln n)^2 \cdot \ln \ln n)$. However, this method is subjected to operate on quantum computers which recently are not available due to lacks of technology advances.

Considering the development of the GNFS, it is currently the fastest factoring algorithm in use. In fact, the most consuming step of this algorithm is the sieving step which attempts to find a number of pairs (a, b) whose norms are smooth over some fixed factor bases. On the other hand, the complexity of this step as well as the later ones mostly depends on the choice of the polynomial $f_1(x)$ and $f_2(x)$ that are used to form the algebraic and rational rings, respectively. Indeed, if the yields of these polynomials together are small, it is more likely that any pair (a, b) would be smooth and hence the sieve would be quicker in finding smooth values. Furthermore, having polynomials with good size properties means that the size for each factor base can also be reduced, thus making the matrix equation simpler as it contains fewer dimensions, though the improvement may not be very significant. Fi-

nally, minor affects can also be found in the square root process as smaller coefficients and factor bases are considered by the computation.

Following this observation, further development to the GNFS will most likely concentrate on improving the polynomial selection process. This however does not mean that there is no place for innovating other parts such as sieving methods or solving matrix equation, which in practice still remain as major questions. In any case, the overall aim of this thesis is no more than to provide a compact documentation with elementary explanations on the recent development of the GNFS, as well as several other mathematical techniques that make it feasible in practice. Within this knowledge scope, it is hoped that this thesis contributes to inspiring mathematical and computer science communities to further improvements of factorization techniques in general, and of the GNFS specifically.

Bibliography

[1] Allenby R.B.J.T. *Rings, Fields and Groups - An introduction to Abstract Algebra*. Hodder Headline Group, London, second edition, 1991.

[2] Bach E. and Peralta R. Asymptotic semismoothness probabilities. *Mathematics of Computation*, 65(216):1701–1715, 1996. `citeseer.ist.psu.edu/bach96asymptotic.html`. Accessed: April 25, 2008.

[3] Bach E. and Shallit J. *Algorithmic number theory, Volume 1: Efficient algorithms*. MIT Press, Cambridge, Massachusets, 1996.

[4] Briggs M.E. An introduction to the general number field sieve. Master's thesis, Virginia Polytechnic Institute and State University, April 1998. `http://scholar.lib.vt.edu/theses/available/etd-32298-93111/unrestricted/etd.pdf`. Accessed: April 25, 2008.

[5] Cohen H. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, Berlin, third edition, 1996.

[6] Cormen T.H., Leiserson C.E., Rivest R.L., and Stein C. *Introduction to algorithms*. The MIT press, Cambridge, Massachusetts, second edition, 2001.

[7] Couveignes J. Computing a square root for the number field sieve. In Lenstra A. K. and Lenstra, Jr. H. W., editors, *The development of the number field sieve*, number 1554, pages 95–102. Springer-Verlag, 1993. `http://citeseer.ist.psu.edu/256874.html`. Accessed: April 25, 2008.

[8] Dickman K. On the frequency of numbers containing prime factors of a certain relative magnitude. *Astronomi oc Fysik*, Arkiv for Mathematik, 22A:1–14, 1930.

[9] Hayward M. Quantum computing and shor's algorithm. Feb 2005. `http://alumni.imsa.edu/~matth/quant/299/paper.pdf`. Accessed: April 25, 2008.

[10] Janeba M. Factoring challenge conquered - with a little help from williamete, 1994. `http://www.willamette.edu/~mjaneba/rsa129.html`. Accessed: April 25, 2008.

[11] Keranen V. Mathematica in word pattern avoidance research. *Computer Algebra and Differential Equations*, 67(3):12–27, 2007. `https://oa.doria.fi/bitstream/handle/10024/36060/CADE_2007.pdf?sequence=1`. Accessed: April 25, 2008.

[12] Kleinjung T. On polynomial selection for the general number field sieve. volume 75. American Mathematical Society, October 2006.

[13] Kocher P.C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996. `http://citeseer.ist.psu.edu/kocher96timing.html`. Accessed: April 25, 2008.

[14] Lenstra A.K. and (Eds) H.W.Jr. Lenstra. The development of the number field sieve. *Lecture Notes in Mathematics*, 1554, 1993.

[15] Lidl R. and Niederreiter H. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge, revised edition, 1994.

[16] Menezes A.J., Oorschot van P.C., and Vanstone S.A.

[17] Montgomery P.L. Square roots of products of algebraic numbers. June 1995. `ftp://ftp.cwi.nl/pub/pmontgom/sqrt.ps.gz`. Accessed: April 25, 2008.

[18] Montgomery P.L. A block lanczos algorithm for finding dependencies over gf(2). *Lecture Notes in Mathematics*, 1998. `http://www.mathmagic.cn/Crypt1981-1997/HTML/PDF/E95/106.PDF`. Accessed: April 25, 2008.

[19] MS A. Public key cryptography - applications algorithms and mathematical explanations. InfoSec Writers, 2007. `http://www.infosecwriters.com/text_resources/pdf/Public_Key_Cryptography_AMS.pdf`. Accessed: April 25, 2008.

[20] Murphy B. *Polynomial selection for the number field sieve integer factorisation algorithm*. PhD thesis, Australian National University, July 1999. `http://citeseer.ist.psu.edu/murphy99polynomial.html`. Accessed: April 25, 2008.

[21] Nguyen P. A Montgomery-like square root for the number field sieve. *Lecture Notes in Computer Science*, 1423:151–168, 1998. `http://citeseer.ist.psu.edu/nguyen98montgomerylike.html`. Accessed: April 25, 2008.

[22] Nguyen P. and Stehle D. An LLL algorithm with quadratic complexity. *Lecture notes in Computer Sciences*, October 2007.

[23] Norton K.K. *Numbers with small prime factors, and the least k-th power non-residue*. American Mathematical Society, Providence, Rhode Island, 1971.

[24] Peterson M. Parallel block lanczos for solving large binary systems, 2006. `http://dspace.lib.ttu.edu/bitstream/2346/1169/1/Peterson_Michael_Thesis.pdf`. Accessed: April 25, 2008.

[25] Raman R., Livny M., and Solomon M. Matchmaking: Distributed resource management for high throughput computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998. `http://www.cs.wisc.edu/condor/doc/hpdc98.ps`. Accessed: April 25, 2008.

[26] Rotman J.J. *A first course in Abstract algebra with applications*. Upper Saddle River, New Jersey 07458, third edition, 2006.

[27] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, June 2002. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`. Accessed: April 25, 2008.

[28] Scheneier B. *Applied Cryptography*. John Wiley & Sons, Inc., New York, second edition, 1996.

[29] Shor P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. `http://citeseer.ist.psu.edu/25085.html`. Accessed: April 25, 2008.

[30] Stewart I.N. and Tall D.O. *Algebraic Number Theory*. Chapman & Hall/CRC, second edition, 1987.

[31] Tannenbaum T., Wright D., Miller K., and Livny M. Condor – a distributed job scheduler. In Sterling Thomas, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001. `http://www.cs.wisc.edu/condor/doc/beowulf-chapter-rev1.pdf`. Accessed: April 25, 2008.

[32] Tattersall J.J. *Elementary Number Theory in Nine Chapters*. Cambridge University Press, New York, 1999.

[33] Trefethen L.N. and Bau D. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 3600 University City Science Center, Philadelphia, PA 19104-2688, 1997.

[34] Weisstein E.W. Rsa-640 factored, November 2005. `http://mathworld.wolfram.com/news/2005-11-08/rsa-640/`. Accessed: April 25, 2008.

ISSN: 1239-7733
ISBN: 978-952-5153-77-4 (nid.)
ISBN: 978-952-5153-78-1 (PDF)