Sieve algorithms for the discrete logarithm in medium characteristic finite fields

Laurent Grémy

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Algorithmes de crible pour le logarithme discret dans les corps finis de moyenne caractéristique

Sieve algorithms for the discrete logarithm in medium characteristic finite fields

THÈSE

présentée et soutenue publiquement le 29 septembre 2017

pour l'obtention du

Doctorat de l'Université de Lorraine

mention Informatique

par

Laurent Grémy

Composition du jury

Président :	Emmanuel Jeandel, professeur de l'Université de Lorraine
Rapporteurs :	Fabien Laguillaumie, professeur de l'Université de Lyon 1 Reynald Lercier, ingénieur de l'armement à la DGA et chercheur associé de l'Université de Rennes 1
Examinatrice :	Nadia Heninger, assistant professor de l'Université de Pennsylvanie
Directeurs de thèse :	Pierrick Gaudry, directeur de recherche au CNRS Marion Videau, maître de conférence de l'Université de Lorraine et en détachement chez Quarkslab

CNRS **Inria** **LORIA – UMR 7503**

Remerciements

Au cours de ces quatres années de thèse, j'ai eu l'occasion de recevoir le soutien de nombreuses personnes. Je tiens, à travers ces quelques lignes, à leur exprimer ma profonde gratitude et prie par avance celles qui se trouveraient à en être absentes de bien vouloir m'en excuser.

Je ne saurais assez remercier Marion et Pierrick. J'ai eu la très grande chance qu'ils acceptent de me prendre en thèse et de me trouver un financement, ce qui ne fut pas des plus aisé. Leur disponibilité, tant pour nos réunions hebdomadaires que pour des questions ponctuelles, leur enthousiasme et leur bienveillance m'ont été très précieux.

Je remercie Nadia Heninger, Emmanuel Jeandel, Fabien Laguillaumie et Reynald Lercier de me faire l'honneur de prendre part à mon jury. Je remercie mes deux rapporteurs d'avoir pris sur leurs vacances pour relire ma thèse et je les remercie encore une fois d'avoir accepté. Je remercie également Emmanuel d'avoir été mon référent de thèse.

À ce sujet, je remercie également François, pour son travail que j'ai apprécié dans le suivi des doctorants et qu'il a dû interrompre lors de ma quatrième année. Je leur sais gré à tous deux d'avoir assuré mon suivi, complémentaire de celui de Marion et Pierrick. J'en profite pour remercier tous les services d'Inria et du LORIA qui ont eu mon dossier entre les mains et qui ont toujours su le gérer avec beaucoup de compétence, en particulier nos assistantes d'équipe, Emmanuelle et Sophie, ainsi qu'Aurélie dans mon suivi RH. Je remercie aussi Christelle.

Je remercie tous mes coauteurs. Je tiens tout particulièrement à remercier Aurore, François et Emmanuel. Ils m'ont permis de vivre une soumission d'article particulièrement paisible, infirmant ces terribles histoires de soumissions à la dernière minute que l'on raconte aux jeunes doctorants pour leur faire peur.

Je remercie tous les membres, de passage ou non, de l'équipe Caramel-Caramba pour leur capacité à s'intéresser à un nombre très vaste de sujets, ce qui conduit toujours à des discussions très animées. Je tiens à remercier tous mes cobureaux : Razvan, pour l'intérêt qu'il a porté à mon travail et pour m'avoir versé dans l'art (occulte) de manipuler les fonctions L, Emmanuel, Pierre-Jean, Cyril, pour le temps qu'il a consacré à nos discussions autour de NFS, Hamza, pour sa sagesse et sa spontanéité, Svyatoslav, pour ses aphorismes, Simon, pour les points cultures qu'il m'a distillés durant ces deux dernières années, Marion, Aurore et Shashank. Grâce à mes deux derniers cobureaux, j'ai découvert, pendant ma rédaction, des présentations de (exT)NFS qui m'ont éclairées et qui, je l'espère, ont enrichi ce manuscrit. Je tiens aussi tout particulièrement à remercier Maike, sans qui mon anglais ne serait jamais devenu ce qu'il est aujourd'hui : le fait que ce manuscrit soit rédigé en anglais lui doit beaucoup. Je remercie également Pierre-Jean et Paul, notamment pour m'avoir permis d'améliorer et de concrétiser certaines parties des algorithmes présentés dans le chapitre 6. Je remercie Jérémie, pour avoir décrypté les différentes décisions de

nos tutelles, Stéphane, pour avoir toujours répondu patiemment à mes questions sur Debian, `git` et tant d'autres choses, Hugo, mon cousin de thèse qui a rédigé le si utile `howto-jesoutiens.txt`, Enea, qui sait apprécier la culture, Thomas, l'éternel stagiaire de troisième qui a participé à CADO-NFS et Luc, qui fut mon professeur de mathématiques en prépa. Je remercie Marine, pour ses conseils lyonnais et pour m'avoir permis d'enseigner la cryptographie et la sécurité.

À ce sujet, je remercie tous les enseignants avec qui j'ai travaillé ou qui ont suivi mon travail à l'UFR MI, notamment Geoffray, Yacine, Armelle, Pascal, Gilles, Romain, Laurent, et les élèves que j'ai suivis de la L2 au M1. Au PLG, j'ai eu l'occasion de pouvoir manger avec les doctorants du CEREFIGE qui m'ont particulièrement bien accueilli et qui ont rendu plus agréable mes longues journées au PLG. Je remercie également les enseignants de la FST, notamment Sylvain, Marie, Emmanuel et Jean, ainsi que mon équipe pour le demi-ATER que j'ai pu effectuer pour ma dernière année.

Je remercie les membres du conseil de laboratoire pour m'avoir accueilli et aidé à comprendre les ressorts, parfois complexes, des décisions, subies ou assumées. Je remercie notamment les rédacteurs des comptes-rendus avec qui j'ai travaillé.

Je remercie tous mes compagnons, parfois d'infortunes, doctorants, qui pour une part, ont participé au pique-nique des doctorants : Simon, Jean-Christophe, toujours enthousiaste pour s'engager dans les différents conseils (de l'école doctorale, du laboratoire, …), Răzvan, qui m'a fait découvrir le pique-nique, Mériem, notre amie du CRAN, Anne, notre amie de l'IECL, Renaud, Pierre, Joseph, une ancienne connaissance lilloise, Éric, avec qui j'ai tant échangé, Hubert, qui joue comme moi le rôle du dernier des mohicans, Hugo, Ludovic, Aurélien, Caterina et tant d'autres. J'en profite pour remercier Dominique, notre directeur de l'école doctorale, et tous les services qui m'ont permis de faire et soutenir ma thèse. Je remercie également tous les doctorants qui ont rédigé avant moi et grâce à qui j'ai pu trouver des renseignements précieux.

Je ne saurais que trop remercier ceux qui m'ont permis de faire mes premiers pas dans la recherche : Matthieu, Christelle et Sylvain à Nîmes. Je remercie l'équipe Calcul Formel à Lille de m'avoir accueilli pour poursuivre ces premiers pas, notamment Charles pour avoir proposé ce stage sur l'algorithme de Raghavendra et LPN, mais également François, François, Alexandre et Adrien pour leurs conseils et leur aide au moment de chercher une thèse. Je remercie également toute l'équipe AriC qui me permet de continuer dans la recherche, et particulièrement Damien pour sa confiance, Gilles pour ses conseils, ainsi que mes cobureaux actuels, Chitchanok et Fabrice.

Je remercie toute la communauté de Sainte Bernadette pour son accueil chaleureux, et notamment tous ceux qui ont animé le groupe des étudiants : Alix-Michelle, Bernard, Bolivard, Brenda, Cathy, Chilande, Françoise, Gaspard, Guillaume, Henri, Lynda, qui fut aussi une des mes élèves, Mariam, Marie-Odile, Têrêsa, et tant d'autres. Je remercie également Christiane, Blandine et Élisabeth pour leur accueil et leur bienveillance.

Enfin, je remercie ma famille, et tout spécialement mes parents, pour leur soutien constant. Depuis 2007, mes études m'ont amené à changer plusieurs fois de lieux et mes multiples déménagements n'auraient jamais pu se faire sans leur aide à tous deux, me déchargeant d'une grosse partie des problématiques matérielles. Malgré les difficultés, ils ont toujours su se montrer disponibles pour m'aider, et notamment, ces 4 derniers mois de conférences, déménagements et soutenance n'auraient pas pu se faire sans leur détermination.

Je remercie aussi toutes les personnes que j'ai rencontrées avant de commencer cette thèse (enseignants, amis, connaissances, …). Elles m'ont permis de m'amener jusqu'ici, ce que je ne regrette absolument pas.

Contents

Introduction

The computation of discrete logarithms is supposed to be a hard problem in general. Exploiting this hardness and the mathematical structure of well chosen groups, Diffie and Hellman [55], with the help of Merkle [95], explained in 1976 how two parties can agree on a secret number using an insecure channel, without the possibility for a third party to recover easily this number. This paved the way to a new type of cryptography, called asymmetric or public-key cryptography.

Before that date, cryptography was symmetric or secret key: the key to encrypt is the one to decrypt. It should therefore be only known by the parties that exchange messages.

From the beginnings of cryptography to the end of World War II, cryptanalysts were more or less able to break all the deployed cryptosystems in the wild. The scytale of the ancient Greeks, the Caesar cipher, the Vigenère cipher, the code of Mary Stuart (whose deciphering lead to her death) and the Enigma machine, all these systems were broken, even if the information about the break of the Enigma cipher was not right away public. One main primitive survived, the one of Vernam, now called one-time pad, which is essentially a Vigenère cipher with the length of the key larger than the length of the message and a random key. Evaluating the security of the cryptosystems proposed by cryptographers is the main goal of the cryptanalysis.

Since the second half of the century, new cryptosystems have undergone a public standardization process, so that the community can study their security. But even if a symmetric system is secure, the difficulty to exchange securely a key between the parties remains. The growth of electronic communications and the need of cryptography for different entities (states, companies, citizens, ...) all around the world worsen this problem of key management when only symmetric cryptography is available. In 1976, the Diffie–Hellman mechanism to agree on a secret between two parties, which becomes the key of a symmetric cryptosystem, solved this problem.

Moreover, asymmetric cryptography is not only a way to exchange keys, and becomes an integral part of cryptography with the raise of RSA [155] and ElGamal [59], the first public-key encryption schemes. In these cryptosystems, a key is divided into two parts: a private one, owned by only one party, and a public one, known by possibly anybody. The security relies either on the integer factorization (for RSA) or on the discrete logarithm problem in the multiplicative subgroup of a finite field (for ElGamal).

One way to evaluate the security is to try to solve efficiently the underlying hard mathematical problems. There always exits an algorithm that solves the problem by trying all the possible solutions. Such an exhaustive approach is also called a brute-force search. For a secure symmetric system, like the AES

cipher, this is currently the best algorithm. In a first approximation, it means that if the key used by AES is n-bit long, an attacker must try on average 2^{n-1} choices to recover the key: such a cryptosystem have a security level of n bits. A security level of 128 bits is considered to be long term [7, Annexe B.1]. To reach the same level of security, RSA and ElGamal must have keys of size 3,072 bits, due to algorithms that have better complexities than the brute-force algorithm.

However, the security of a cryptosystem is not dependent only on the hardness of the underlying mathematical problem. For example, the school book Diffie–Hellman protocol on an unauthenticated channel is not robust against a man-in-the-middle attack. Besides, the security can be downgraded due to legal limitations, as the export of cryptography from the United States that limited the sizes of keys: this limitation has been used to run the Logjam attack on the Diffie–Hellman key exchange [6]. Other types of attacks are listed in Figure 1. In this thesis, we focus on mathematical attacks on the discrete logarithm problem, see Figure 2.

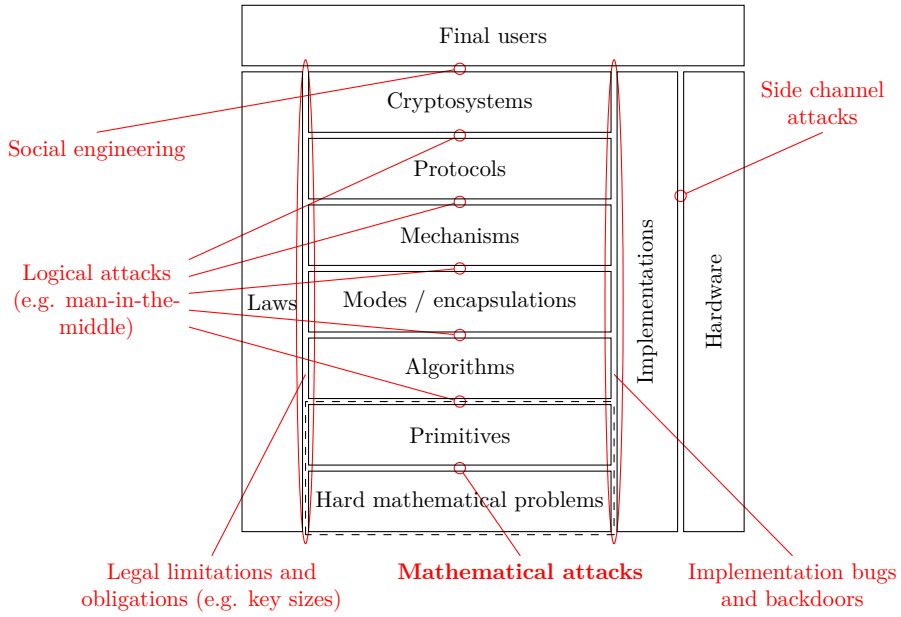


Figure 1 – Stack of abstraction layers in cryptography with possible attacks.

The number field sieve algorithm

To compute discrete logarithms in finite fields of large sizes, the best known approach uses the number field sieve (NFS) algorithm. NFS was first used in the factorization context at the end of the 80s [127]. The complexity of this algorithm is subexponential, which is expressed thanks to the L function defined as $L_N(\alpha, c) = \exp((c + o(1))(\log N)^{\alpha}(\log\log N)^{1-\alpha})$ [147, 126]. More precisely, the complexities of the variants of NFS reach $\alpha = 1/3$ and $c \leq (128/9)^{1/3}$.

The use of NFS in the context of discrete logarithms defined on prime fields is due to Gordon [77]. However, it was not the first $L(1/3)$ algorithm on finite

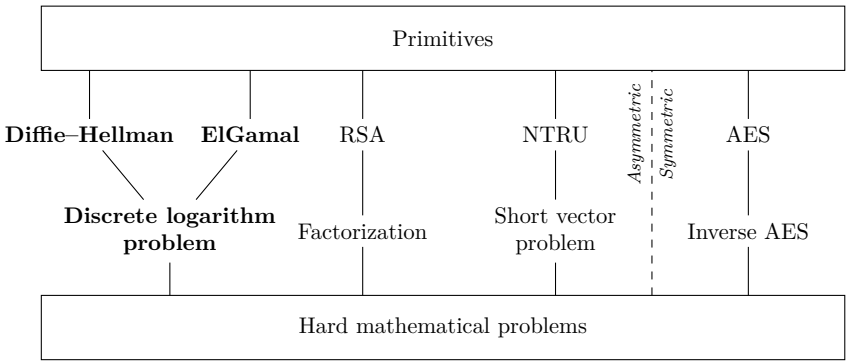


Figure 2 – Some links between hard mathematical problems and primitives.

fields: in 1984, Coppersmith proposed an algorithm to compute discrete logarithms on finite fields of characteristic 2, $\mathbb{F}_{2^n}$ [49]. The reader interested in the history of these algorithms is invited to read the survey of Joux, Odlyzko and Pierrot [107], and the one of Guillevic and Morain [89].

We distinguish three types of finite fields $\mathbb{F}_{p^n}$, depending on the characteristic p or the extension n:

- small p (typically, $p = 2$ and $p = 3$): in these fields, since 2014, there exists a quasi-polynomial algorithm [21, 80].
- large p (typically $n = 1$): in these fields, the discrete logarithm variant of the runs in about $L_{p^n}(1/3, (64/9)^{1/3})$.
- medium p (typically $n = 6$): in these fields, the situation evolved during the last three years.

In this thesis, we focus on the last case. The first NFS variant in this field was introduced in 2006 by Joux, Lercier, Smart and Vercauteren [106] with a running time in $L_{p^n}(1/3, (128/9)^{1/3})$. Since 2014, there were a lot of improvements to NFS. The most noticeable one is the extended tower number field sieve (exTNFS) algorithm [114], which runs in the general case in $L_{p^n}(1/3, (64/9)^{1/3})$, when n is composite, and when n is prime in $L_{p^n}(1/3, (96/9)^{1/3})$ [20, 143].

All the best known algorithms to compute discrete logarithms in finite fields are index calculus algorithms [123]. This family can be split into three main steps, once a nice representation of the field is chosen: relation collection, linear algebra and individual logarithm computation. Since the use of the quadratic sieve by Pomerance [148], we know that the relation collection can be efficiently performed using sieving algorithms, similar to the one of Eratosthenes to enumerate prime integers. With the NFS variants, the cost of the individual logarithm computation is negligible compared to the one of the relation collection and the linear algebra steps. The costs of these two phases depend on a large number of parameters. It is therefore usual to try to find the best trade-off between the two costs. However, this trade-off does not mean that the costs are balanced. In medium characteristics, the cost of the relation collection seems larger than the one of the linear algebra. Indeed, in the computation of Zajac [185], the running time to perform the sieving step is around two times

longer than the one of the linear algebra step, and around 40 times longer in the computation of Hayasaka, Aoki, Kobayashi and Takagi [93].

In this thesis, we focus on the relation collection for NFS. In large characteristics, the relation collection is performed by enumerating elements of a two-dimensional lattice. In medium characteristic, the dimension of the lattice may be larger. It is therefore needed to design algorithms that work efficiently in this type of lattices, and to try to reduce the cost of the relation collection.

Summary of contributions

The main contributions of this thesis revolve around the relation collection step for NFS in dimension higher than two, and especially the design, analysis and implementation of sieve algorithms on the one hand and the handling of record computations.

Polynomial selections. In the NFS variants, the input of the relation collection includes, among other parameters, polynomials to represent $\mathbb{F}_{p^n}$: the best their *quality* are, the lowest is the running time of the relation collection. We extend in three dimensions the two-dimensional quality criteria in an article coauthored with Pierrick Gaudry and Marion Videau [72]. We also propose a modification in the way to define these polynomials, to take into account the specificities of the sieve algorithms used to collect the relations.

Sieve algorithms. The relation collection is divided into three main steps: initialization of the norms, sieving and cofactorization. In addition to the line sieve, described by Zajac [185], we describe in [72], two new sieve algorithms in three dimensions, one of them being an extension of the two-dimensional sieve of Franke–Kleinjung. In Chapter 6, we propose a general framework in which these three sieve algorithms in three dimensions appear as particular cases to sieve in higher dimensions, resulting in two algorithms to sieve in any small dimensions.

Implementation. We implement a complete relation collection in higher dimensions in CADO-NFS. Our implementation allows to use all the variants of NFS, except the (ex)TNFS one. We include dedicated sieve algorithms in three dimensions and some of them allow to sieve in any higher dimensions.

Record computations. Our implementation was used to perform five computations of discrete logarithms. We summarize the results of these computations in Chapter 8. The computations are described in three articles: one in [72], concerns specifically the polynomial selection and the relation collection in $\mathbb{F}_{p^6}$, another one coauthored with Aurore Guillevic and François Morain [84], reports the first complete computation in $\mathbb{F}_{p^5}$, and the last one coauthored with Aurore Guillevic, François Morain and Emmanuel Thomé [85], reports four complete computations in $\mathbb{F}_{p^6}$, establishing a new record.

Other contributions. We wish also to highlight a few results we obtained during the writing of this manuscript.

Analysis of the MNFS variant of Zajac. In 2008, Zajac described a variant of NFS [185, Section 6.4]. Even if Zajac described some technicalities about this variant, its complexity was not analyzed. We analyze it in Appendix D and show that the complexity is the same as for another variant proposed in 2014 by Barbulescu and Pierrot [24], that is around $L_{p^n}(1/3, 2.40)$.

TNFS. TNFS is the precursor of exTNFS. Barbulescu, Gaudry and Kleinjung analyzed the complexity of this algorithm by using a specific setting [22]. We show in Chapter 5 that another polynomial selection is available, ensuring the same complexity, especially for the individual logarithm computation.

A small implementation of NFS. To highlight the different steps of NFS and the links between them, we provide in Appendix B a small implementation (less than 350 lines of code) of NFS on prime fields in Sage, except for the computation of an individual logarithm. This implementation, essentially focused on the relation collection, is described in Chapter 3.

Database of computations of discrete logarithms. The reports of computations of a discrete logarithm are done in non-uniformed ways, as in articles or emails to the NMBRTHRY list (`https://listserv.nodak.edu/cgi-bin/wa.exe?A0=NMBRTHRY`). With Aurore Guillevic, we propose a database that collects all the computations of discrete logarithms in a unified way with the references of the computations [83].

Outline of the manuscript

This thesis is divided into three parts: the first part provides some background on the mathematical and algorithmic sides of the manuscript. We begin by focusing on the use of discrete logarithms on finite fields in cryptography and describe some algorithms to compute discrete logarithms in Chapter 1. Before presenting NFS, we focus in Chapter 2 on the sieve algorithms, especially the use of the Eratosthenes sieve to factor integers in an interval. Even if we describe these sieve algorithms over the integers, a one-dimensional set, the notions we introduce in this chapter is used in all the subsequent chapters. We conclude this part by describing NFS on prime fields in Chapter 3. This chapter introduces the key notions of NFS and covers all the steps of the algorithm.

The second part is a focus on algorithms to solve the discrete logarithm problem in medium characteristics. We describe two variants of NFS, the high degree variant (NFS-HD) in Chapter 4 and the exTNFS one in Chapter 5. In Chapter 4, we describe two quality criteria in three dimensions to select the best polynomials coming from the polynomial selections. We also describe the different variants of NFS-HD. In Chapter 5, we quickly describe some of the variants of exTNFS. As exTNFS is a new algorithm, we also list some of the challenges that remain to be solved in a near future to perform practical computations using this algorithm. The NFS-HD and the exTNFS algorithms use an high-dimensional relation collection: we describe in Chapter 6 two generic algorithms to sieve in any small dimensions. We study them and show how the three sieve algorithms we described in three dimensions are covered by this general framework.

The third part concerns the practical results of our sieve algorithms. In Chapter 7, we justify some of the choices we did in our implementation. We also describe how our code is integrated in CADO-NFS and the challenges we need to solve to provide an automatic tool, as done by the `cado-nfs.py` script for prime field and factorization. Finally, we describe how we have managed the record computations we did, by describing how we found the parameters of the relation collection and how we run the computations on a cluster.

Part I

Discrete logarithms in finite fields

Chapter 1

Discrete logarithm

Computing discrete logarithm is at the heart of some widely deployed asymmetric cryptographic primitives. The hardness of computing discrete logarithms ensures the security of these primitives. This hardness depends on the mathematical structure in which discrete logarithms lie. Depending on this structure, there exist different algorithms to compute discrete logarithms, and the largest computations help cryptographers derive which security is guaranteed according to the size of the keys.

Let G be a multiplicative group and $\cdot$ denote the group operation between elements of G. Let a be an element of G, we denote by a^k, where k is an integer, the result of composing k times a with itself using $\cdot$, as $a^k = a \cdot a \cdot \cdots \cdot a$. This is an *exponentiation* of a to the power k. The inverse of an element a in the group is denoted by a^{-1}. The group G is *finite* when its cardinal n is finite and *cyclic* when there exists an element g such that $G = \{1 = g^0, g, g^2, \ldots, g^{(n-1)}\}$. Elements like g are called generators of the group. In the following, we call group a finite cyclic group.

Definition 1.1 (Discrete logarithm)**.** Let G be a group generated by g and n the cardinality of the group. Let h be an element of G. The discrete logarithm of h in basis g is the element k in $[0, n[$ such that $h = g^k$. This element k is often denoted by $\log_g h$.

Definition 1.2 (Discrete logarithm problem (DLP))**.** Given G, g and h as in Definition 1.1, the discrete logarithm problem is to compute the integer $k = \log_g h$. In general, n is assumed to be known.

1.1 Cryptography and discrete logarithm

In all of this section, the attackers are passive, that is they can only read the messages exchanged between the different parties but they cannot modify or fake a message. All the described cryptosystems will be secure under this type of attackers.

1.1.1 The Diffie–Hellman key exchange

The Diffie–Hellman key exchange [55] is used by two parties Alice and Bob to agree on a secret key. Only Alice and Bob can compute the key, if the group to which the key belongs, has some computational properties, to be defined in Section 1.1.3. Let G be a group of cardinality n, g be a generator of this group. Let these three elements be publicly available, following a given standard, so that Alice and Bob can have access to these pieces of information. To exchange a key K over an insecure medium, Alice chooses K_a, an element in $[0, n[$ and sends g^{K_a} to Bob. For his part, Bob chooses K_b in $[0, n[$ and sends g^{K_b} to Alice. The common shared key is $K = (g^{K_b})^{K_a} = (g^{K_a})^{K_b}$. This protocol is now standardized among others in ANSI X9.42 [8] and is the basis of many popular protocols over the Internet, as TLS [54]. As the key is built during the exchange, the protocol is not strictly speaking a key exchange but a key agreement.

From the point of view of the attacker, the information he has access to is made of the public parameters and the elements g^{K_a} and g^{K_b}. The security of this protocol relies on the difficulty for an attacker to compute the key K.

Definition 1.3 (Computational Diffie–Hellman problem)**.** Let G, g and n be as in Definition 1.1, and let k and k' be two elements in $[0, n[$. Then the computational Diffie–Hellman problem is to compute $g^{kk'}$ from g, g^k and $g^{k'}$.

If the discrete logarithm problem in a group is easy, the computational Diffie–Hellman problem becomes easy. However, if the computational Diffie–Hellman problem is easy in a group, it is not necessarily the case with the discrete logarithm problem [133]. The security is enhanced when it is hard to answer the Decisional Diffie–Hellman problem, that is given a, b and c randomly chosen in $[0, n[$, it is difficult to distinguish the two distributions (g^a, g^b, g^{ab}) and (g^a, g^b, g^c).

1.1.2 The ElGamal encryption

The ElGamal encryption [59] is a well-known public-key encryption method. It is derived from the Diffie–Hellman key exchange. Let the two parties be Alice and Bob. Bob wants to send a ciphertext to Alice.

Key generation. Alice creates a key pair, that is a public key and a private key. To do that, she chooses a group G of cardinality n and a generator g of this group. Her private key K_a is an element of $[0, n[$. The public key k_a is composed of the group G, the generator g, the cardinality n of the group and $h = g^{K_a}$: the public key k_a is the 3-tuple $\{G, g, h\}$.

Encryption. This public key is sent to Bob. He can encrypt his message m, an element of G, as follows. Bob chooses an integer r in $[0, n[$ and computes $c_0 = m \cdot h^r$. He also computes $c_1 = g^r$ and sends the message composed of c_0 and c_1.

Decryption. To decrypt the message (c_0, c_1) sent by Bob, Alice computes $c_0(c_1^{K_a})^{-1}$. She gets the message m because $m = mh^r\left((g^r)^{K_A}\right)^{-1}$.

This encryption scheme is among other things integrated in the GNU Privacy Guard software on two different types of group. The security of this encryp-

tion scheme relies mainly on the hardness of solving the Computational Diffie–Hellman problem on the group G chosen by Alice. As in the Diffie–Hellman key exchange, the security is enhanced when Decisional Diffie–Hellman problem is hard.

1.1.3 Proposed groups

Requirements

From these first two cryptographic schemes based on the computational hardness of computing discrete logarithms, we can deduce some requirements on the group G in order to have a strong security and a reasonable efficiency.

- To ensure a strong security, the DLP must be computationally hard to solve in G. Ideally, the difficulty of computing a discrete logarithm should be exponential in the size of n with the best known algorithm.
- One of the qualities of an encryption scheme is computational efficiency in terms of running time. The basic operation of the two previous schemes is the exponentiation, and it is required that it can be done in polynomial time in the size of n. It is also required that the elements of G can be represented in memory with $O(\log n)$ bits.

Choice of group

Two different groups are nowadays widely deployed in asymmetric cryptography. Historically, Diffie and Hellman proposed to perform the key exchange in $\mathbb{F}_{p^n}^*$, the multiplicative group of finite fields of cardinality p^n, where p is a prime and n is an integer. It fulfills all the requirements. Indeed, the elements of $\mathbb{F}_{p^n}^*$ can be represented in $n\lfloor \log_2(p) \rfloor$ bits and, using the binary exponentiation, we can compute a^k in $O(\log k)$ operations, each of them taking polynomial time in $O(n \log p)$. Computing discrete logarithms in these groups is more or less difficult, depending on the choice of p and n. These different cases depend on the relative size of n and $\log p$ and are discussed in Section 1.3.3.

Another interesting group is the group of rational points of an elliptic curve. Let $\mathcal{E}$ be an elliptic curve defined over the field $\mathbb{F}_q$. Using Hasse's theorem on elliptic curves, the number of points of $\mathcal{E}(\mathbb{F}_q)$ is equal to $q + 1 - t$, where $|t| < 2\sqrt{q}$. A point (x, y) defined on $\mathcal{E}(\mathbb{F}_q)$ can be represented with $2 \log_2 q$ bits, and the double-and-add algorithm allows to perform the exponentiation in $O(\log k)$ operations in $\mathcal{E}$, each of them taking polynomial time in $\log q$. The best known algorithms to compute discrete logarithms in the group of rational points of an elliptic curve have an exponential complexity in $\log q$, except for some specific classes of curves.

1.1.4 Signature schemes

Signature schemes are used to authenticate and provide non-repudiation of a message, which means guaranteeing that its author is Alice and that anyone can check that only she can be the author. In the following section, the signature schemes are described using the multiplicative subgroup $\mathbb{F}_p^*$ of a prime field, keeping in mind that their variants on other finite fields and elliptic curves also exist.

The ElGamal signature

Alice wants to sign a message to be sent to Bob. She has a key pair, like in Section 1.1.2. Her private key is the element K_a in $[0, p-1[$. Her public key k_a is composed of the prime p, a generator g and $h = g^{K_a}$.

Signature generation. Let m in $[0, p-1[$ be the message. To generate the signature of the message, Alice chooses an integer e in $[0, p-1[$ such that e and $p-1$ are coprime. She then computes $r = g^e \bmod p$ and $s = (m - K_a r)e^{-1} \bmod (p-1)$. Alice sends to Bob the triple $\{m, r, s\}$.

Verification Bob receives $\{m, r, s\}$ and has a copy of the public key of Alice. To verify the signature $\{r, s\}$ of m, he compares $h^r r^s \bmod p$ and $g^m \bmod p$. The equality validates the fact that m is signed by Alice. Indeed we have, during the signature generation of Alice, the equality $m = K_a r + se \bmod (p-1)$ and, by Fermat's little theorem, $g^m = g^{K_a r} g^{se} = h^r r^s \pmod p$.

DSA

The Digital Signature Algorithm (DSA) was proposed in 1991 by the National Institute of Standards and Technology [113]. It is a variant of the ElGamal signature. It uses groups such that their cardinality $p-1$ has a large prime factor, q. Let g_q be a generator of the subgroup of order q of the group $\mathbb{F}_p^*$. We can find g_q with g, the generator of $\mathbb{F}_p^*$, by computing $g^{(p-1)/q} \bmod p$. The private key of Alice is an element K_a in $[0, q[$. Her public key is then a little bit modified compared to the scheme of ElGamal: it is composed of the parameters $(p, q, g_q, h = g_q^{K_a} \bmod p)$.

Signature generation. To sign a message m in $[0, q[$, Alice chooses a random integer e in $[0, q[$ and computes $r = ({g_q}^e \bmod p) \bmod q$. She also computes $s = (m + K_a r) \bmod q$. The signature of m is then (r, s) and (m, r, s) is sent to Bob.

Verification To verify the signature of m, Bob needs to have the public key of Alice. He computes $w = s^{-1} \bmod q$, $u_0 = mw \bmod q$, $u_1 = rw \bmod q$ and $v = (g_q^{u_1} h^{u_2} \bmod p) \bmod q$. If $v = r$, Alice has signed the message m.

Using DSA is faster than using the ElGamal signature scheme, because the operation are done in a subgroup where computations can be faster. The cardinality q of this subgroup can be much smaller than $p-1$.

1.1.5 Pairing-based cryptography

Pairings were introduced in cryptography in 1993 by Menezes, Okamoto and Vanstone [135] and by Frey and Rück [64] as a tool to attack the discrete logarithm problem in several families of elliptic curves. Pairings will be thereafter used as a constructive tool in many cryptosystems.

Definition 1.4 (Pairings [35])**.** Let G_0 and G_1 be additive groups and G_T be a multiplicative group such that these three groups have the same order. A pairing is a map $e : G_0 \times G_1 \to G_T$ which is

1. bilinear: for all P_0, P_1 in G_0 and Q_0, Q_1 in G_1,
   - $e(P_0 + P_1, Q) = e(P_0, Q)e(P_1, Q)$ and
   - $e(P, Q_0 + Q_1) = e(P, Q_0)e(P, Q_1)$;
2. non-degenerate: for all non neutral P in G_0, there exists Q in G_1 such that $e(P, Q) \neq 1$ and for all non neutral element Q in G_1, there exists P in G_0 such that $e(P, Q) \neq 1$;
3. computable in polynomial time in the input size.

In cryptography, the groups G_0 and G_1 are groups of rational points of elliptic curves and G_T is a multiplicative subgroup of a finite field. To be secure, the discrete logarithm problem in these three groups must be difficult. With this requirement, one can build some cryptosystems like, among other things, the 3-partite Diffie–Hellman key exchange proposed by Joux [99], the identity-based encryption of Boneh and Franklin [36], the traitor tracing scheme of Mitsunari, Sakai and Kasahara [138] and the Boneh–Lynn–Shacham short signature scheme [37].

We briefly describe this signature scheme. Let the cardinality of G_0 , G_1 and G_T be a prime n and g_1 a generator of G_1. Let m in G_0 be the message that Alice wants to sign. Alice creates her secret key K_a by selecting a random integer in $[0, n[$ and computes her public key $k_a = K_a g_1$ which is in G_1. To sign m, Alice computes the signature s by computing $K_a m$, an element in G_0. To verify the signature, Bob checks if $e(s, g_1) = e(m, k_a)$.

1.1.6 Torus-based cryptography

To conclude the usage of discrete logarithms in cryptography, we will give a brief overview of the building-blocks of the torus-based cryptography. An interested reader can found more informations in the book of Galbraith [66, Chapter 6]. Torus-based cryptography can be viewed as the same idea as the DSA: use a subgroup of $\mathbb{F}_{p^n}$ to have an efficient arithmetic, but ensure the security by recovering the computations in a larger group.

Let consider the finite field $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/\varphi(X)$, where p is a prime and φ is an irreducible polynomial of degree 2. The group $\mathbb{F}_{p^2}^*$ always admits two subgroups, one of cardinality $p-1$ which is $\mathbb{F}_p^*$ and another of cardinality $p+1$, denoted by $\mathbb{T}_2(\mathbb{F}_p)$ called torus. Naively, an element of $\mathbb{T}_2(\mathbb{F}_p)$ is represented by a polynomial of degree one with coefficient in $\mathbb{F}_p$ and the multiplication of two elements of the torus require three multiplications over $\mathbb{F}_p$ using the Karatsuba algorithm and a reduction modulo the polynomial φ. Using Lucas sequences, also used in the $p+1$ factoring algorithm [184] and for primality tests [149, 156], allows us to represent an element of the torus by only one element in $\mathbb{F}_p$ and the multiplication of two such elements requires only one multiplication over $\mathbb{F}_p$. Cryptosystems using Lucas sequences in such a way are called LUC.

Instead of considering the finite field $\mathbb{F}_{p^2}$, Lenstra and Verheul, the designers of XTR [129], use $\mathbb{F}_{p^6}$. They exploit the subgroup $\mathbb{T}_6(\mathbb{F}_p)$ of cardinality p^2-p+1. An elements of this subgroup can be represented by only two elements in $\mathbb{F}_p$ and the multiplication of two elements requires three multiplications over $\mathbb{F}_p$. The cryptosystem CEILIDH [157], introduced by Rubin and Silverberg, can be viewed as a generalization of the LUC and XTR cryptosystems, allowing us

to perform the whole ElGamal encryption scheme, where XTR uses an agreed secret key to perform a part of the encryption.

1.2 Generic algorithms

Let G be a group of cardinality n and g a generator of this group. Let t be an element of G. In this section, we want to compute k, the discrete logarithm of t in basis g. The three algorithms we are going to describe are generic, in the sense that if we just know G, n, g and have access to an oracle that takes two elements a, b of G and returns the result ab, we can solve the DLP in this group, with a better complexity than the exhaustive search, which runs in time $O(n)$.

1.2.1 Pohlig–Hellman

The Pohlig–Hellman algorithm [144] uses the subgroups of G if n is composite. Let ℓ be a prime and e be an integer such that ℓ^e divides n. Then, there exists a unique subgroup of G of order ℓ^e generated by $g_0 = g^{n/\ell^e}$. Let t be an element of G such that its discrete logarithm is k. Then, t mapped in the subgroup of order ℓ^e is equal to ${g_0}^{k \bmod \ell^e}$.

Let n be a composite number whose factorization is equal to $\prod_{i=0}^{j} {p_i}^{e_i}$, with e_i an integer and p_i prime, and let p_j be the largest prime factor. By computing k_i, for i in $[0, j]$, such that $(g^{n/{p_i}^{e_i}})^{k_i} = t^{n/{p_i}^{e_i}}$, we can then reconstruct k by the Chinese Remainder Theorem. The complexity of solving the DLP in G is then dominated by the complexity of solving the DLP in a subgroup of G of order ${p_j}^{e_j}$.

Let us consider the multiplicative subgroup of G of order ${p_j}^{e_j}$. A generator of this subgroup is $g_j = g^{n/{p_j}^{e_j}}$ and t mapped in this subgroup is denoted by t_j. Let us now consider the base-p_j expansion of k_j: we write $k_j = k_{j,0} + k_{j,1}p_j + \cdots + k_{j,e_j-1}{p_j}^{e_j-1}$, with $k_{j,0}, k_{j,1}, \ldots, k_{j,e_j-1}$ in $[0, p_j[$. We have ${t_j}^{{p_j}^{e_j-1}} = {g_j}^{k_j{p_j}^{e_j-1}} = {g_j}^{k_{j,0}{p_j}^{e_j-1}}$. Let $t_{j,0} = {t_j}^{{p_j}^{e_j-1}}$ and $g_{j,0} = {g_j}^{{p_j}^{e_j-1}}$. Computing $k_{j,0}$ can be done by solving the DLP of $t_{j,0}$ in basis $g_{j,0}$ in the group of prime order p_j. We use similar computations for the other coefficients $k_{j,1}, k_{j,2}, \ldots, k_{j,e_j-1}$.

The Pohlig–Hellman reduction allows us to reduce solving a discrete logarithm in G to solving discrete logarithms in subgroups of G of cardinality p_i. This algorithm needs $O(\sum_{i=0}^{j} e_i(\log n + \sqrt{p_i}))$ group operations. Therefore, in the rest of this section, we assume that the cardinality of G will be prime.

The two following algorithms run in time $O(\sqrt{n})$ if n is prime. Shoup [169] has proved that no generic algorithm can have a complexity below $\Omega(\sqrt{n})$.

1.2.2 Shanks' algorithm

Shanks' algorithm, also called *baby-step giant-step*, was described in [167]. The idea is to write k as $k_b r + k_g$, with $r = \lceil \sqrt{n} \rceil$ and k_b, k_g in $[0, r[$. We can then rewrite $g^{k_b r + k_g} = t$ as $g^{k_g} = t(g^{-r})^{k_b}$. By computing all the possible values for g^{k_g} (giant steps), with k_g in $[0, r[$, and $t(g^{-r})^{k_b}$ (baby steps), with k_b in $[0, r[$, we eventually find a collision for a couple (k_b, k_g). The discrete logarithm k of t in basis g is then $k = k_b r + k_g$.

This algorithm needs to perform about $2\lceil\sqrt{n}\rceil$ group operations, therefore the expected running time is in $O(\sqrt{n})$. The algorithm needs to store about $\lceil\sqrt{n}\rceil$ group elements, therefore the memory complexity is in $O(\sqrt{n})$.

To improve the constant in the $O(\sqrt{n})$, Bernstein and Lange [32] and Galbraith, Wang, and Zhang [68] proposed to use two giant steps and one baby step, waiting for a collision between any of the three sets. The resulting algorithm needs $1.25\sqrt{N}$ group operations in total.

1.2.3 Pollard algorithms

Pollard rho

The Pollard rho algorithm [146] runs in the same time complexity as Shanks' one, but uses a constant memory complexity. The idea is to find a relation involving powers of t and g. To that purpose, the algorithm performs a random walk on the elements of the group G of the form $t^a g^b$, with a and b in $[0, n[$. Let, for all integer i, a_i and b_i be in $[0, n[$ and $x_i = t^{a_i} g^{b_i}$. The random walk allows to find a collision, that is, two integers i_0 and i_1 such that $x_{i_0} = x_{i_1}$.

Let w be a function that simulates a random walk on the elements of G. This pseudo-random walk is used to generate $x_{i+1} = w(x_i)$. If the x_i's seem uniformly and independently distributed on G, then, by the birthday paradox, the expected number of calls to w until we get a collision is in $(\sqrt{n\pi/2})$. To have a memory complexity in $O(1)$, we need to use the Floyds's cycle finding trick and compare x_i with x_{2i} at each step. The name "rho" comes from the shape of the path followed by the random walk: after the first collision, the pseudo random walk enters a cycle and the pseudo-random walk seems to describe the Greek letter ρ, as shown in Figure 1.1.

A collision occurs when a, b, a', b' in $[0, n[$ are such that $t^a g^b = t^{a'} g^{b'}$. To compute the discrete logarithm k of t in basis g, we compute $(b - b')/(a' - a) \bmod n$. The algorithm fails if $a = a'$. In such a case, we choose another x_0 in G.

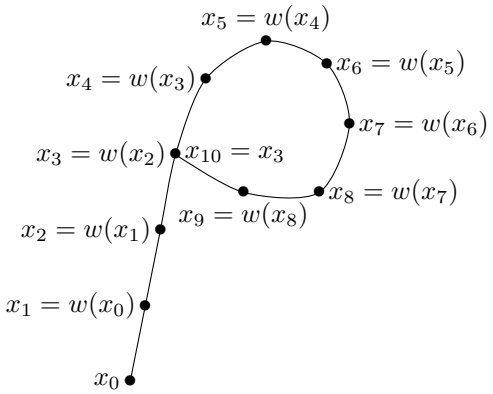


Figure 1.1 – Typical orbit for the Pollard rho algorithm.

Pollard kangaroo

The Pollard kangaroo algorithm is used when the discrete logarithm k of t is known to lie in a small interval. Although the complexity of this algorithm is the same as the Pollard rho or Pohlig–Hellman algorithms, the practical computation is faster. We do not detail this algorithm, and refer to the book of Galbraith [66, Chapter 14] and the improvements of Galbraith, Pollard and Ruprai [67].

1.3 Index calculus algorithms

Index calculus algorithms are nowadays the best family of algorithms to compute discrete logarithms in large finite fields or to factorize large numbers. They are the focus of this thesis.

1.3.1 General description

Let $\mathbb{F}_{p^n}^*$ be a multiplicative group of cardinality $p^n - 1$ and g be a generator of this group. Index calculus algorithms can be decomposed into two main steps: the first step (divided here in relation collection and linear algebra) is used to compute the logarithm of a subset $\mathcal{F}$ of elements of *small* sizes in $\mathbb{F}_{p^n}^*$. This subset is used in a second step to compute the discrete logarithm of a *large* target. The first step can therefore be considered as a precomputation for the individual logarithm step.

Relation collection

Let $\mathcal{F}$ be a subset of $\mathbb{F}_{p^n}^*$ containing small elements of $\mathbb{F}_{p^n}^*$. The relation collection is used to collect multiplicative relations involving the elements of $\mathcal{F}$. A relation is then of the form $\prod_i {f_i}^{m_i} = \prod_j {f_j}^{m_j}$, where f_i and f_j belong to $\mathcal{F}$ and m_i and m_j are integers modulo $p^n - 1$. By taking the logarithm of this multiplicative relation, we get a linear equation involving the logarithms of these small elements, that is $\sum_i m_i \log_g f_i = \sum_j m_j \log_g f_j$. We continue to collect relations until the system built by the linear relations involving unknown logarithms is overdetermined.

Linear algebra

The linear algebra step is used to compute the values of the logarithms of the elements of $\mathcal{F}$. As the system given by the relation collection is overdetermined and consistent, there exists a unique solution of this system. Using the Pohlig–Hellman reduction, we then solve the system modulo each prime involved in the factorization of the cardinality of $\mathbb{F}_{p^n}^*$. At the end of this step, all the discrete logarithms of the elements of $\mathcal{F}$ in basis g are found.

Individual logarithm

Let h be a large element of $\mathbb{F}_{p^n}^*$. At the end of this step, the discrete logarithm of h will be expressed as a linear combination of logarithms of elements of $\mathcal{F}$. This linear relation is generally not easy to find. We first rewrite the logarithm of h in

terms of some smaller elements, not all in $\mathcal{F}$ and for each smaller element, redo this procedure until the involved logarithms used to decompose the logarithm of h all belong to $\mathcal{F}$.

1.3.2 A first subexponential algorithm for prime fields

In this section, we instantiate the group G by a multiplicative group of a prime field $\mathbb{F}_p^*$, where p is a prime. The cardinality of this group is then $p-1$. Let g be a generator of this group. From now on, the best algorithms we have described in Section 1.2 are in $O(\sqrt{p})$. We describe here a subexponential algorithm proposed by Adleman at the end of the 70's [2] to find k such that $t = g^k$, where t is in $\mathbb{F}_p^*$ and k is an integer.

Smoothness

Before describing the Adleman algorithm, we define the smoothness of an integer, an important notion used in the index calculus algorithm family.

Definition 1.5 (Integer smoothness)**.** Let B be an integer. An integer n is B-smooth if the largest prime factor of n is strictly less than B.

Let B and $n \geq B$ be integers, we are interested in the evaluation of the number $\psi(n, B)$ of integers less than n that are B-smooth. Then, the probability for a random integer less than n to be B-smooth is equal to $P(n, B) = \psi(n, B)/n$. This probability cannot be quickly computed but we can obtain an asymptotic formula. In 1983, Canfield, Erdős and Pomerance gave a good approximation of it.

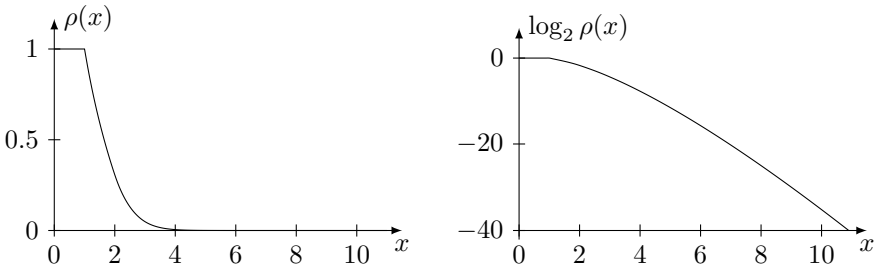
Theorem 1.1 (Smoothness probability [44])**.** *Let $\varepsilon > 0$ be fixed and u be such that $3 \leq u \leq (1-\varepsilon)\log n/\log\log n$. An approximation of $\psi(n, n^{1/u})$ is $n\exp(-u(\log u + \log\log u - 1 + o(1)))$.*

Then, the probability for an integer less than n to be B-smooth is equal to $u^{-u(1+o(1))}$, where $u \geq 1$ is the ratio between the size of n and the size of B, $u = \log n/\log B$. When u is large enough, $u^{-u+o(1)}$ is very close to $\rho(u)$ [96, Corollary 1.3], where ρ is the Dickman function. The Dickman function is the unique continuous function defined on positive real numbers satisfying the delay differential equation $x\rho'(x) + \rho(x-1) = 0$ when $x > 1$ and $\rho(x) = 1$ when $x \leq 1$. The Dickman function is plotted in Figure 1.2.

Adleman algorithm

Let $B < p$ be a smoothness bound to be defined later. We define the factor base $\mathcal{F}$ as the primes less than B. To find relations involving the elements of $\mathcal{F}$, we pick a random r in $[0, p-1[$ and raise g to the power r: if g^r viewed as an integer is B-smooth, then we have a relation between elements of $\mathcal{F}$ and g^r. We need to have at least $\#\mathcal{F} = \pi(B)$ relations to have a chance to build a system of maximum rank, where π is the prime-counting function.

Let M be a matrix of size $\#\mathcal{F} \times \#\mathcal{F}$. We label the columns of M by primes in $\mathcal{F}$. A row of this matrix corresponds to a relation: the coefficients of the unknowns are written in the corresponding columns. Once each needed relation is encoded as a row of M, we solve the system $Mx = y$, where x and y are in



Graph of $\rho(x)$. Graph of $\log_2 \rho(x)$.

Figure 1.2 – Graph of the functions $\rho(x)$ and $\log_2 \rho(x)$ for x in $[0, 11.0]$.

$(\mathbb{F}_p^*)^{\#\mathcal{F}}$ and where each coefficient of the vector y contains the random power of g for the corresponding relation. This computation is performed modulo each of the prime factors of the cardinality $p-1$ of the group. Then, we use the Chinese Remainder Theorem to compute all the discrete logarithms in $\mathbb{F}_p^*$.

Finally, to compute the discrete logarithm of t, we compute tg^r for different r in $[0, p-1[$, until we find a relation involving only elements of $\mathcal{F}$. For such an r, $tg^r = \prod_{i=0}^{j} {f_i}^{m_i}$ with m_i integers and f_i in $\mathcal{F}$. The discrete logarithm k of t in basis g is then $k = (-r + \sum_{i=0}^{j} m_i \log_g f_i) \bmod (p-1)$.

Complexity analysis

The subexponential function. The vast majority of index calculus algorithms have a subexponential complexity, that is a complexity smaller than exponential but larger than polynomial. The range between the two extreme complexities is covered by a real number α in $[0, 1]$. For an input of size $\log q$ and given a positive constant c, the subexponential function, also called L function, is given by $L_q(\alpha, c) = \exp((c + o(1))(\log q)^\alpha (\log\log q)^{1-\alpha})$.

Proposition 1.1 (L-arithmetic)**.** *Let q, a_b, a_n, b, n, be five positive real numbers. We have*

- $L_q(a_b, b)L_q(a_n, n) = \begin{cases} L_q(a_b, b) & \text{if } a_b > a_n; \\ L_q(a_n, n) & \text{if } a_n > a_b; \\ L_q(a_b, b+n) & \text{if } a_n = a_b. \end{cases}$
- $L_q(a_b, b) + L_q(a_n, n) = \begin{cases} L_q(a_b, b) & \text{if } a_b > a_n; \\ L_q(a_n, n) & \text{if } a_n > a_b; \\ L_q(a_b, \max(b, n)) & \text{if } a_n = a_b. \end{cases}$
- $L_q(a_b, b)^n = L_q(a_b, nb)$.
- $L_{L_q(a_b,b)}(a_n, n) = L_q(a_n a_b, nb^{a_n} a_b^{1-a_n})$.

Corollary 1.1 (of Theorem 1.1)**.** *Let q, a_b, a_n, b, n be five positive real numbers. Let B be an integer bounded by $L_q(a_b, b)$ and N bounded by $L_q(a_n, n)$. The probability of N to be B-smooth is equal to $L_q(a_n - a_b, (a_n - a_b)n/b)^{-1}$.*

If we work in a field $\mathbb{F}_q$, we often use the piece of notation $L(\alpha)$ instead of $L_q(\alpha, c)$, considering that q is implicit and c is not needed in a first approximation, because any modification of α changes $L_q(\alpha, c)$ more than any modification of c would do with a constant α.

The algorithm. We now prove that the Adleman algorithm has a complexity in $L(1/2)$. We begin by the relation collection. All the elements of $\mathbb{F}_p^*$ of the form g^r are less than p, that is in $L_p(1, 1)$. Let the smoothness bound B be equal to $L_p(a_b, b)$ with a_b, b two real numbers. By Corollary 1.1, the probability of g^r to be B-smooth is then equal to $P = L_p(1-a_b, -(1-a_b)1/b)$. The cardinality of $\mathcal{F}$ is bounded by $\pi(B)$, we therefore look for $\pi(B) < B$ relations. The average number of B-smoothness tests is then less than B/P. The smoothness tests can be done naively by trial division, each test costing B divisions. The total cost of the relation collection is then equal to $L_p(a_b, 2b)L_p(1 - a_b, (1 - a_b)/b)$. Since the first argument of the L function dominates the result, we choose $a_b = 1 - a_b = 1/2$ and then, the complexity of the relation collection step is $L_p(1/2, 2b + 1/(2b))$.

The linear algebra step can be performed by Gaussian elimination. The time complexity of this algorithm is polynomial in the dimension of the matrix M and can be upper-bounded by $O(B^3)$. We need to solve at most $\log p$ systems, but, written with L-functions, the coefficient is absorbed in the $o(1)$ and then, the linear algebra step can be performed in $L_p(1/2, 3b)$. The cost of the precomputation step using Adleman algorithm is then given by $L_p(1/2, 2b + 1/(2b)) + L_p(1/2, 3b)$. Intuitively, the cost of the precomputation must be balanced between the relation collection and the linear algebra steps, we therefore need to have $L_p(1/2, 2b + 1/(2b)) = L_p(1/2, 3b)$: this occurs when $2b + 1/(2b) = 3b$, that is $b = \sqrt{2}/2$. The cost of the precomputation seems then to be in $L_p(1/2, 3\sqrt{2}/2)$. But, by using L-arithmetic, we know that $L_p(1/2, 2b + 1/(2b)) + L_p(1/2, 3b) = L_p(1/2, \max(2b + 1/(2b)), 3b)$. The analysis of the function $\max(2b + 1/(2b), 3b)$ shows that the minimal complexity is reached when $b = 1/2$, that is a complexity in $L_p(1/2, 2)$, as shown in Figure 1.3.

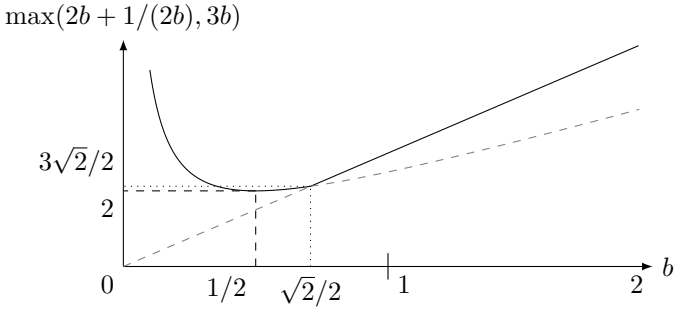


Figure 1.3 – Graph of the function $\max(2b + 1/(2b), 3b)$ for b in $[0.1, 2[$.

The cost of the individual logarithm step is the cost to find one relation, that is $B/P = L_p(1/2, 3/2)$.

To find B-smooth numbers less than p, we can use the elliptic curve factorization method [130], which has a complexity in $L_B(1/2, \sqrt{2})$. By using this

algorithm in the relation collection step and the Wiedemann algorithm to solve sparse linear systems (presented in Section 3.3.2), we can reach the complexity $L_p(1/2, \sqrt{2})$ for the precomputation step, and $L_p(1/2, \sqrt{2}/2)$ for the individual logarithm step.

1.3.3 Today's algorithms and choices of finite fields

Nowadays, the index calculus family covers all the spectrum of the different finite fields, with complexity at most $L(1/3)$. Let $\mathbb{F}_{p^n}$ be a finite field where p is a prime. We distinguish three types of finite fields, as summarized in Figure 1.4. For each case, we give the different variants and the corresponding complexities.

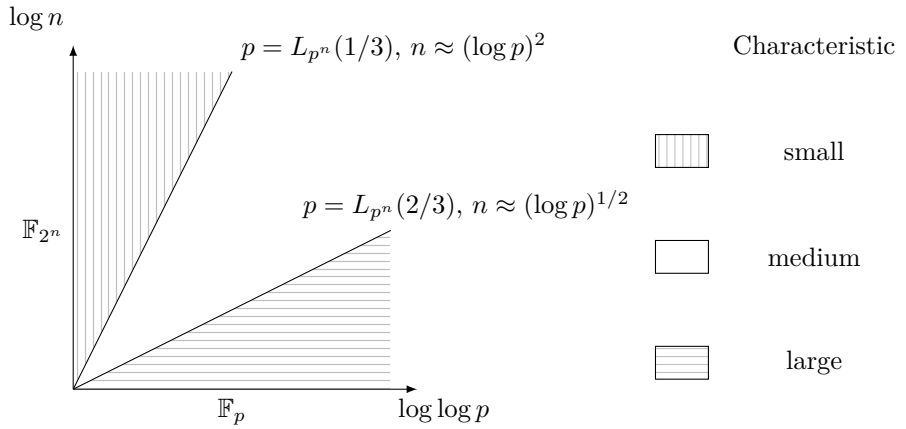


Figure 1.4 – Different domains of finite fields depending on their characteristic.

Small characteristic

The small characteristic case occurs when $\log p$ is small compared to n, typically the cases when $p = 2$ and $p = 3$. This type of fields allows to use the Frobenius map $x \mapsto x^p$ as a way to derive a relation from another for free. Because of the structure of the relation collection of the best index calculus algorithm, we also need to consider the B-smoothness of polynomials lying in $\mathbb{F}_p[x]$. A polynomial in $\mathbb{F}_p[x]$ is B-smooth if its largest irreducible factor is of degree less than B. This test can be done in polynomial time in the degree of the input polynomial. These two advantages allow, among other things, to reach smaller complexities than in the other cases. Before the end of 2012, the best algorithms ran in $L(1/3)$, due to the work of Coppersmith [49], Adleman [3], Adleman–Huang [5] and Joux–Lercier [105] resulting in the function field sieve algorithm of complexity $L_{p^n}(1/3, (32/9)^{1/3})$.

In the beginning of 2013, Joux [102] proposed a new algorithm of complexity $L(1/4)$, and in the middle of the same year, Barbulescu, Gaudry, Joux and Thomé [21] proposed the first quasi-polynomial algorithm, that is a complexity in $(n \log p)^{O(\log(n \log p))}$. Some improvements were proposed afterwards by Granger, Kleinjung, and Zumbrägel [80] and Joux and Pierrot [109].

Large characteristic

The finite fields $\mathbb{F}_{p^n}$ are said to be of large characteristic if n is very small, typically n is less than 4. The first $L(1/3)$ algorithm was proposed by Gordon in 1993 [77], running in time $L_p(1/3, 9^{1/3})$. In his thesis [165], Schirokauer obtains the nowadays $L_p(1/3, (64/9)^{1/3}) \approx L_p(1/3, 1.93)$ complexity. The algorithm that reaches this complexity, the number field sieve (NFS) algorithm, will be more detailed in Chapter 3. A variation of NFS, called multiple number field sieve due to Coppersmith [48] and Commeine–Semaev [47], and refined by Barbulescu and Pierrot [24], can reach a better complexity in $L_p(1/3, ((92+26\sqrt{13})/27)^{1/3}) \approx L_p(1/3, 1.91)$. When p has a special form, it can be exploited thanks to the special number field sieve algorithm of Gordon [76], to reach a complexity lower than in the general case, precisely $L_p(1/3, (32/9)^{1/3}) \approx L_p(1/3, 1.53)$. The majority of these algorithms are not relevant when $n > 1$. The tower number field sieve (TNFS) algorithm of Barbulescu, Gaudry and Kleinjung [22], from a former idea of Schirokauer [162], allows also to reach an $L_{p^n}(1/3, (64/9)^{1/3})$ complexity in the general case, and $L_{p^n}(1/3, (32/9)^{1/3})$ in special cases.

Medium characteristic

Work in this area started in 2006, when Joux, Lercier, Smart and Vercautern described the first $L(1/3)$ algorithm, whose complexity was $L_{p^n}(1/3, (128/9)^{1/3}) \approx L_{p^n}(1/3, 2.43)$ [106]. In 2014, Barbulescu and Pierrot improved the constant to $((4(46+13\sqrt{13}))/27)^{1/3} \approx 2.40$ [24]. In 2015, Barbulescu, Gaudry, Guillevic and Morain [20] reduced the complexity to $(96/9)^{1/3} \approx 2.21$ and Pierrot, by combining the two last improvements, reached the constant $((8(9+4\sqrt{6}))/15)^{1/3} \approx 2.16$ [143].

When n is composite, the complexity drops to $(64/9)^{1/3} \approx 1.93$ by using the extended tower number field sieve (exTNFS) algorithm proposed by Kim and Barbulescu [114], and in favorable cases, it is possible to reach $(48/9)^{1/3} \approx 1.75$. There exist many variants of exTNFS, with a multiple field variant and a variant when p has a special form. The first algorithm that exploits this special form is the one of Joux and Pierrot [108]; in this case, exTNFS improves the complexity to $(32/9)^{1/3} \approx 1.53$. We detail all these types of NFS in Part II.

Records

If lowering the overall complexity of index calculus algorithms to solve the DLP on finite fields is an achievement, we still need to implement these algorithms to compute discrete logarithms if we want to evaluate the practical impact of these theoretical improvements. Indeed, we can give as an example that the complexity of NFS to factor large integers is the same as the one to compute discrete logarithms in the multiplicative group of a finite field of prime characteristic. However the RSA-768 challenge solved at the end of 2009 [120] took less than 15 million core hours, compared to the 46 million core hours necessary to solve a DLP of the same size [122] in 2016. We report in Table 1.1 some of the biggest computations of discrete logarithms in $\mathbb{F}_{p^n}^*$, and in Figure 1.5 and Figure 1.6 the evolution of the records. Except for the medium characteristic, the records seem to follow the improvements of the algorithms. In the table

and the two figures, p is a prime and n is an integer. Guillevic and us provide a complete list of the computations of discrete logarithms in finite field in [83].

Finite field	Date	Bit size	Algorithm	Cost: CPU days	Authors
$\mathbb{F}^*_{2^n}$	01/2014	9234	QPA	$13.6 \cdot 10^3$	Granger,Kleinjung and Zumbrägel [81]
	05/2013	6168	$L(1/4)$	23	Joux [100]
$\mathbb{F}^*_{2^p}$	10/2014	1279	$L(1/4)$ and QPA	$1.28 \cdot 10^3$	Kleinjung [119]
	04/2013	809	FFS	805	Caramel Team [17]
$\mathbb{F}^*_{3^n}$	07/2016	4841	QPA	$73.1 \cdot 10^3$	Adj, Canales-Martinez, Cruz-Cortés, Menezes, Oliveira, Rodríguez-Henríquez and Rivera-Zamarripa [1]
	09/2014	3796	QPA	359	Joux and Pierrot [110]
$\mathbb{F}^*_{p^{57}}$	01/2013	1425	FFS	513	Joux [101]
$\mathbb{F}^*_{p^{47}}$	12/2012	1175	FFS	534	Joux [101]
$\mathbb{F}^*_{p^{12}}$	11/2013	201	NFS	11	Hayasaka, Aoki Kobayashi and Takagi [93]
$\mathbb{F}^*_{p^6}$	02/2008	240	NFS	38	Zajac [185]
$\mathbb{F}^*_{p^4}$	10/2015	392	NFS	510	Barbulescu, Gaudry, Guillevic and Morain [87]
$\mathbb{F}^*_{p^3}$	08/2016	593	NFS	$8.4 \cdot 10^3$	Gaudry, Guillevic and Morain [73]
$\mathbb{F}^*_{p^2}$	06/2014	595	NFS	175	Barbulescu, Gaudry, Guillevic and Morain [19]
$\mathbb{F}^*_p$	10/2016	1024	SNFS	$136 \cdot 10^3$	Fried, Gaudry, Heninger and Thomé [65]
	06/2016	768	NFS	$1.94 \cdot 10^6$	Kleinjung, Diem, Lenstra, Priplata, and Stahlke [122]
	06/2014	596	NFS	$61 \cdot 10^3$	Bouvier, Gaudry, Imbert, Jeljeli and Thomé [40]

Table 1.1 – Discrete logarithm records on finite fields.

Discussion on prime field

The cryptographic primitives whose security rely on the hardness of the DLP mostly use prime fields for the computations. In this section, we discuss about the choice of the prime p used to define $\mathbb{F}_p$.

"Safe" prime and small subgroups. With the Pohlig–Hellman algorithm (see Section 1.2.1) we have seen that solving a discrete logarithm in $\mathbb{F}^*_p$ is as difficult as solving a discrete logarithm in the subgroup whose order is the largest factor of $(p-1)$. It follows that if the largest factor of $(p-1)$ is too small, even if $(p-1)$ is large, the computation of a discrete logarithm is easy. To increase the hardness of computing discrete logarithms in $\mathbb{F}^*_p$, we look for a prime such that $p-1 = 2p'$, where p' is a prime. With this definition, p is a Sophie Germain prime. The density of Sophie Germain primes less than n is equal to $2Cn/(\ln n)^2$, where C is the twin-prime constant, equal to $\prod_{p'>2} p'(p'-2)/(p'-1)^2 \approx 0.66$ [170, Conjecture 5.24]. Such a prime p is called "safe" because the order of the largest subgroup of $\mathbb{F}^*_p$ is of order $(p-1)/2$. If 2,048 is a good bit

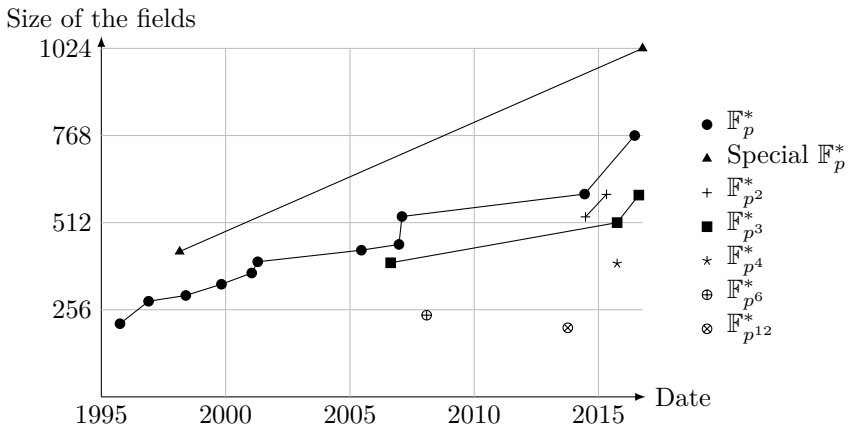


Figure 1.5 – Evolution of the records with NFS.

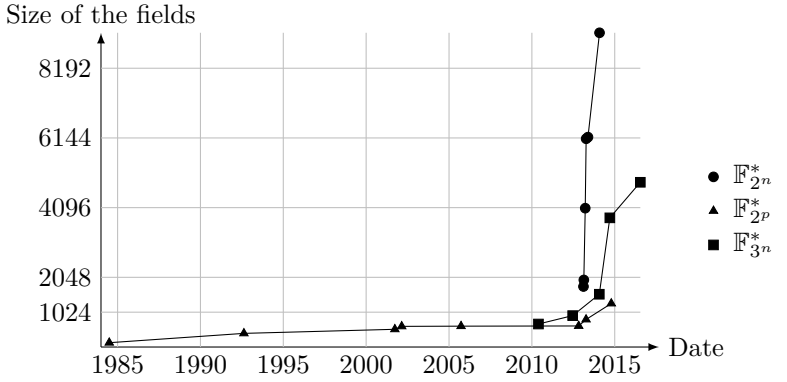


Figure 1.6 – Evolution of the records in small characteristic.

size for p, then there exist more than 2^{2026} safe primes, compared to more than 2^{2036} primes, which is more than enough in both cases.

Instead of using safe primes, DSA, described in Section 1.1.4, uses prime fields with a composite characteristic, to achieve a quick signature scheme. Parameters for p and q, the order of the subgroup in which the computations are performed, are proposed in [113], and for p of size 2,048 bits, it is recommended to use q of size 224 and 256 bits. Such a q is likely less than the largest prime dividing $p-1$; it is chosen to balance the cost of computing a discrete logarithm in the subgroup of order p with the Pollard rho algorithm and in the group $\mathbb{F}_p^*$ with NFS. A discussion on the consequences of such a choice is proposed in [179], with applications in real-life cryptography.

Common prime. As noted previously, the choice of p is important, and to avoid some bad choices, one can recommend to use a prime with some good properties, defined in some trusted standard. This seems reasonable, but the use of the same prime p in a large number of applications may decrease significantly the security of this prime. Indeed, once the precomputation step of an

index calculus is performed given p, one can perform the individual logarithm step as many times as there exist discrete logarithms to be computed. Let consider a p of size 512 bits. Using NFS, if the precomputation step is reachable, the individual logarithm step can be performed quickly. Using the CADO-NFS implementation [176], the authors of the Logjam attack [6] computed a single discrete logarithm in less than 10 minutes using a single core computer, while the precomputation costs more than 10 years on a single core. With this attack, and by downgrading a TLS connection, the authors showed that "82% of vulnerable servers [the TLS downgraded hack] use a single 512-bit group". They also observed that "breaking the single, most common 1024-bit prime used by web servers would allow passive eavesdropping on connections to 18% of the Top 1 Million HTTPS domains. A second prime would allow passive decryption of connections to 66% of VPN servers and 26% of SSH servers".

Special prime. As seen with the Dual EC pseudorandom number generator trapdoor [33], using standardized elements without knowing how they were produced can potentially decrease the security of a system. In the context of computing discrete logarithms on the multiplicative group of a finite field, it is possible, with the Gordon algorithm [76], to build a prime that can pass all the requirements for DSA, but, for the one who knows how this prime was built, there exists a way to forge fake signatures, as shown in the article of Fried, Gaudry, Henninger and Thomé [65]. This drawback can be avoided using the recommendations of NIST's FIPS 186 [113, Appendix A].

Chapter 2

Sieve algorithms

Sieve algorithms are used in number theory to enumerate elements of a set which verify a given arithmetic property. The first described sieve is attributed to Eratosthenes and is used to find prime numbers. This first algorithm allows many variants to practically improve the expected running time of the sieve, for instance the use of wheel factorizations. The main idea of the sieve of Eratosthenes to find prime numbers will not be improved before the sieve of Atkin and Bernstein [9], about 2,000 years later. The idea of the sieve of Eratosthenes can also be used to generate numbers with other interesting arithmetic properties, as the B-smoothness.

In this chapter, we will describe sieve algorithms that look for a given property of elements in an integer interval $[I, J[$. In some of them, it is possible to reduce the number of considered elements in $[I, J[$ by only looking for odd or even integers, or more generically integers in a set of the form $b\mathbb{Z} + c$, with b and c integers. We note that $b\mathbb{Z}$ is a sublattice of $\mathbb{Z}$ of basis b and $b\mathbb{Z} + c$ is a translate of this sublattice. This notion of translate of a sublattice will be ubiquitous in Chapter 6, where we sieve in higher dimension.

Our model of computation is a RAM (Random-Access Machine) with a direct access memory. The classical arithmetic operations have a unit cost, just like the comparisons, array indexing, assignement and branching.

2.1 Sieve of Eratosthenes and variants

The sieve of Eratosthenes is used to find prime numbers in an integer interval $[0, N[$. In this section, we will describe the classical algorithm and some variants around it. These algorithms use the fact that if p is a prime, all its multiple cannot be primes.

2.1.1 Schoolbook algorithm

The algorithm is often described as an array containing all the integers between $[0, N[$ and cross out the non-prime elements, as depicted in Figure 2.1. At the end of the algorithm, the non-struck elements are prime. By definition, 0 and 1 are not prime, so the first prime number is 2. In the interval $[0, N[$, the

multiples of 2 greater than 2, that are $\{4, 6, \ldots, 2\lfloor N/2 \rfloor\}$, cannot be primes. Once we have enumerated all these non-prime elements, we look for the next prime greater than 2. This element is 3, all its multiples larger than 3, that are $\{6, 9, \ldots, 3\lfloor N/3 \rfloor\}$, are not primes. The next prime element is 5, 4 is not a prime, because it is a multiple of 2. Once again, all the multiples of 5 larger than 5 are not prime and we look for the new prime, that is 7. We use this procedure as long as the next possible element is less than N.

Figure 2.1 – Representation of the sieve of Eratosthenes in $[0, 30[$.

A description of such an algorithm is given in the following:

1. Initialize a boolean array A of size N with **True** values. Set $A[0]$ and $A[1]$ to **False**.
2. While $p < N$
   (a) Go to next position p where $A[p]$ is equal to **True**; report p.
   (b) For $2 \leq k \leq \lfloor N/p \rfloor$, set $A[kp]$ to **False**.

We now briefly study the time complexity of this algorithm. In each iteration of the while loop of Item 2, we perform less than N/p updates in the array A if p is prime. The time complexity of the sieve of Eratosthenes is $O(\sum_{p \text{ prime}}^{N} N/p) = O(N \log \log N)$ by Mertens' theorems [136]. The space complexity is obviously in $O(N)$.

Comparison with exhaustive search

There exist many primality tests. The fastest probabilistic primality test is the one of Miller–Rabin: given an integer n, the algorithm returns an indication of the primality of n with a running time in $O(\log n)$, if a constant number of witnesses are used. Instead of using the sieve of Eratosthenes to find all the probable primes in the interval $[0, N[$, we can try to perform the Miller–Rabin primality test on all the numbers in the interval. The time complexity of such an algorithm is then in $O(\sum_{n=2}^{N} \log n)$, that is the exhaustive search runs in $O(N \log N)$. The use of the sieve of Eratosthenes to find primes in $[0, N[$ is therefore more efficient in term of running time than exhaustive search using Miller–Rabin primality test.

First improvements

The for loop on k in Item 2b can be done on the restricted interval $[p, \lfloor N/p \rfloor]$ because the multiples of p less than p^2 were treated previously. Then, the while loop in Item 2 is performed when p is less than $\sqrt{N}$. It results in the following algorithm:

1. Initialize the array A of size N with **True** values. Set $A[0]$ and $A[1]$ to **False**.
2. While $p < \sqrt{N}$

(a) Go to next position p where $A[p]$ is equal to **True**; report p.

(b) For $p \leq k \leq \lfloor N/p \rfloor$, set $A[kp]$ to **False**.

Even considering this, the time complexity is not improved in a major way and one of the drawbacks of the schoolbook sieve of Eratosthenes, the huge amount of memory needed to store all the possible primes, is not improved. In the following, we will describe some variations around the sieve of Eratosthenes that improve the space or/and the time complexity.

2.1.2 Segmented sieve

From one of the previous remarks, we know that we can find all the primes up to N by sieving with all the primes up to $\sqrt{N}$. Instead of discovering the primes up to $\sqrt{N}$ during the procedure of the sieve of Eratosthenes, that is what we do during the Item 2a of the schoolbook algorithm, we can precomput these primes by a recursive call to the sieve of Eratosthenes and doing the while loop of Item 2 only on the precomputed primes. This recursive call has a negligible cost in time, $O(\sqrt{N} \log\log N)$, and in memory, $O(\sqrt{N})$, compared to the costs to sieve up to N.

The segmented sieve is a way to decrease the memory requirement of the sieve of Eratosthenes by looking for primes in slices of length B. It was proposed by Singleton [171] and used by Brent [42] and Bays–Hudson [27]. Let I_k be an interval of length B of the form $[kB, (k+1)B[$, for an integer $k \leq \lfloor N/B \rfloor - 1$. With the precomputed primes up to $\sqrt{N}$, we will sieve in all the segments I_k one after the other. At step k, a boolean array A of size B stores the primality or not of the elements of I_k and an element a of I_k corresponds to the index $a - kB$. By concatenating all these arrays, we get the array A of the classical sieve. It just suffices to find the first location of a multiple of a prime p in I_k. The algorithm can be described as follows:

1. Compute the primes up to $\sqrt{N}$.
2. For all interval I_k
   (a) Initialize an array A with B cells set to **True**.
   (b) For each precomputed primes p, set the cells corresponding to multiples of p in I_k to **False**.
   (c) For all the positions i such that $A[i]$ is **True**, report that $i + kB$ is prime.

The sieving step performing in Item 2b can be done as in the classical sieve of Eratosthenes. Indeed, if i_0 in I_k is a multiple of a prime p and i_1 is its location in the corresponding array A, then $i_0 + p$ is also a multiple of p, corresponding to the location $i_1 + p$ in the array A. The first location in A can be computed as $(p - kB) \bmod p$. But from one k to the other, it is possible to keep the next position for p to avoid this computation. Storing this information requires to add $O(\sqrt{N})$ in the memory complexity.

The number of visited cells is the same as in the classical sieve, the time complexity is therefore $O(N \log\log N)$. The space complexity is equal to about $O(\sqrt{N}) + O(B)$. Then, if $B \approx \sqrt{N}$, the space complexity of the segmented

sieve is equal to $O(\sqrt{N})$. A drawback with this sieve algorithm is the number of hits per segment if p is greater than $N^{1/4}$, which is zero on average. From an implementation point of view, the primes can be divided in families depending on the number of hit per segment. This is done in the implementation of `primesieve` [182].

2.1.3 Wheel sieves

We will introduce the wheel sieve as an improvement of the sieving in congruence classes. We begin by describing a specific sieve for the primes in the congruence class 1 modulo 4, but the complexities in other congruence classes is the same. We then show the advantage of the wheel sieve as an extension of the sieve of the primes in the congruence class 1 modulo 2 and finally, the combination of the wheel and the segmented sieves.

Sieving in a congruence class

In this section, we want to enumerate the primes in the range $[0, N[$ such that the remainder of their division by 4 is equal to 1. The prime verifying these properties can be written as $4k + 1$, where k is an integer. Let A be an array which stores at index k by a boolean the primality of $1 + 4k$. If p can be written as $1 + 4k$, then its square is $4(4k^2 + 2k) + 1$ and is located at index $4k^2 + 2k$ in A. Therefore, to remove all the multiple of a prime p larger than its square, it suffices to look at the indices $(p^2 - 1)/4 + ip$ of A. If $p \equiv 1 \bmod 4$ is prime, the cell at index $(p - 1)/4$ will not be modified. The procedure runs as follow:

1. Compute the primes up to $\sqrt{N}$.
2. Create an array A indexed from 0 to $(N-1)/4-1$, initialized with **True**.
3. for all primes p up to $\sqrt{N}$
   (a) Compute $k = (p^2 - 1)/4 - 1$.
   (b) While $k < (N-1)/4-1$, store **False** in $A[k]$ and compute $k = k+p$.
4. All the locations k where $A[k]$ is equal to **True** give a prime $p = 1+4(k+1)$.

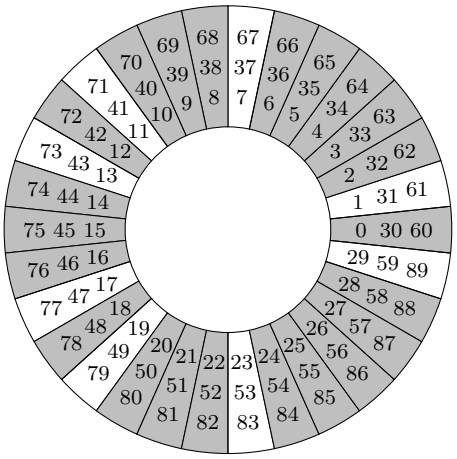
Combining with a segmented like algorithm, we can reduce the memory complexity to $O(\sqrt{N})$ and the time complexity is in $O(N \log \log N)$.

Wheel sieve

During the first iterations of the while loop of the classical sieve described in Section 2.1.1, we deal with small primes that generate a lot of hits. It is especially annoying when $p = 2$, because we already know that 2 is the only even prime. It seems therefore interesting to consider only the odd integers larger than 2, that is the translate of the sublattice of the odd integers, that is elements of the form $2k + 3$, with $k \geq 0$. The primality of an odd integer a larger than 2 is stored in the boolean array A at index $(a - 3)/2$.

Keeping the idea of removing the even numbers, we also can delete the multiples of other small primes. Let W be the product of the first n primes. If m in $[0, W[$ is divisible by one of the n primes p, p divides also $m + kW$, for

any k. By removing the multiples of the n primes in the interval $[0, W[$, we just consider $\phi(W)$ elements, where ϕ is the Euler's totient function. The set of those elements coprime to W is denoted by $\mathcal{P}$. The possible primes above W are therefore of the form $kW + i$, where $k > 0$ and i is in $\mathcal{P}$. Contrarily to the previous algorithms, the set of such integers is not a translate of a single lattice but a union of those. To store these numbers in a boolean array A, each cell representing an element greater than the one in the previous cell, we need a function I, that returns, given an element in $\mathcal{P}$, the index of this element in $\mathcal{P}$ (that is, if $W = 30$, $\mathcal{P} = \{1, 7, 11, 13, 17, 19, 23, 29\}$, $I(11) = 2$). It is therefore more complicated to enumerate the multiples of a prime because these multiples are not regularly stored in A. An example of the array A when $W = 30$ is given in Figure 2.2.



0	1	2	3	4	5
1	7	11	13	17	19

5	6	7	8	9	10
19	23	29	31	37	41

11	12	13	14	15	...
43	47	49	53	59	...

Figure 2.2 – Wheel factorization with $W = 30 = 2 \cdot 3 \cdot 5$ (no primes in the gray parts, instead of $2, 3, 5$) and produced array.

Let k be an integer. We want to find the primes in the interval $[0, kW[$. With the sieve of Eratosthenes, we need to stores kW booleans, instead of $k\phi(W)$ with the wheel sieve. We therefore save a factor around 3.75 when $W = 2 \cdot 3 \cdot 5$ and around 4.38 when $W = 2 \cdot 3 \cdot 5 \cdot 7$.

Segmented wheel sieve

Instead of considering the whole array A as we just described, we can consider a radius of the wheel, formed by integers of the form $kW + i$. It is convenient to sieve along a radius of the wheel, because we are in a translate of a lattice. In a boolean array, we store the primality of $a = k_0W + i$ at index k_0. As in the segmented wheel, to find the prime up to N, we need to know all the primes up to $\sqrt{N}$. The algorithm can be described as follow:

1. Compute W as the product of the first n primes and report these as prime.
2. Compute the primes up to $\sqrt{N}$ and remove those that divide W.
3. For each of the coprimes residue i modulo W
   (a) Initialize a boolean array A of size $\lfloor N/W \rfloor$ with **True**

(b) For each precomputed primes p up to $\sqrt{N}$, set to **False** the cells corresponding to multiples of p.

(c) Report that $i + kW$ is prime if $A[k]$ is **True**.

The space complexity of this algorithm is then equal to $O(\sqrt{N} + N/W)$. The loop on the coprimes at Item 3 is executed $\phi(W)$ times and it can be shown that $\phi(W) = O(W/\log\log W)$. There are about $N/(Wp)$ updates in A at Item 3b for each prime p, therefore the number of all the updates in A is in $O((N/W)\log\log N)$. For a prime, finding its first multiple in A can be done by the extended Euclidean algorithm in $O(\log N)$ basic operation and we need to do that for all the prime up to $\sqrt{N}$, a number in $O(\sqrt{N}/\log N)$. Puting all the time complexities together, we get a time complexity equals to $O(W/\log\log W(N/W\log\log N + \sqrt{N}))$. By taking $W = O(\sqrt{N})$, we can get a time complexity in $O(N)$ and a space complexity in $O(\sqrt{N})$. More details about the segmented wheel sieve can be found in the articles of Pritchard [153] and Sorenson [173].

2.2 Other sieves

In this section, we present two sieve algorithms. The first one was introduced by Pritchard in [154] and is inspired by the sieve of Eratosthenes while the second uses a different idea. At the end of this section, we summarize the time and space complexities for most of the sieve algorithms in the literature.

2.2.1 Composite sieve

In the classical sieve of Eratosthenes, some of the composite numbers are crossed out by different primes. In this sieve (described without considering a possible wheel), a composite is crossed out exactly once. Let a be a composite integer, p be the smallest prime factor of a and f be the cofactor that is $a = pf$. The prime p is therefore smaller or equal to the smallest prime factor of f. Then, by considering all the possible cofactors f in $[2, \lfloor N/2 \rfloor]$ and all the primes up to the least prime factor of f, we can generated all the composite integers up to N. As in the schoolbook sieve, we need to store the primality of all the integers up to N. The sieve works as follow:

1. Compute the primes up to $\sqrt{N}$.
2. Initialize the array A of size N with **True** value. Set $A[0]$ and $A[1]$ to **False**.
3. For $2 \leq f \leq \lfloor N/2 \rfloor$
   (a) For each prime p up to $\sqrt{N}$
      i. If p is less or equal to N/f, set $A[pf]$ to **False**.
      ii. If p divides f, break the loop.
4. Report indices of the **True** locations in A.

The space complexity is obviously in $O(N)$, as the time complexity because any composite in $[0, N[$ is visited only once. Using a wheel, we can get a time complexity in $O(N/\log\log N)$ and a space complexity in $O(N^{1+o(1)})$ and the segmented sieve combined with the wheel reduces once again the space complexity.

2.2.2 Sieving by quadratic forms

In this section, we will consider the writing of integers modulo 12. Except 2 and 3, all the primes can be written as $12k_0 + k_1$, with k_1 in $\{1, 5, 7, 11\}$. The primes are therefore congruent to 1 modulo 4 or 1 modulo 6 or 11 modulo 12. A first but inefficient idea will be to sieve these congruence classes. An other idea is to consider the writing of the primes in these families using binary quadratic forms, the Atkin–Bernstein sieve [9].

Instead of removing the multiple of a prime, the primes are discovered by looking for irreducible binary quadratic forms. There exist three forms to find the primes, listed in the Section 6 of the article of Atkin and Bernstein, following these three theorems:

Theorem 2.1. *Let n be a squarefree integer such that $n \equiv 1 \bmod 4$. Then, n is prime if and only if the cardinality of $\{(x, y) : x > 0, y > 0, 4x^2 + y^2 = n\}$ is odd.*

Theorem 2.2. *Let n be a squarefree integer such that $n \equiv 1 \bmod 6$. Then, n is prime if and only if the cardinality of $\{(x, y) : x > 0, y > 0, 3x^2 + y^2 = n\}$ is odd.*

Theorem 2.3. *Let n be a squarefree integer such that $n \equiv 11 \bmod 12$. Then, n is prime if and only if the cardinality of $\{(x, y) : x > y > 0, 3x^2 - y^2 = n\}$ is odd.*

In these three theorems, if n is the integer to be tested, we can bound x and y in the interval $[1, \sqrt{n}]$. Like in the sieve of Eratosthenes, we look for all the primes in the interval $[0, N[$. The primes 2 and 3 cannot be detected using these quadratic forms. A naive algorithm of the sieve of Atkin–Bernstein can be written as follow:

1. Initialize an array P with **False**, and set $P[2]$ and $P[3]$ to **True**
2. Initialize an array L with **False**
3. For the tuple (a, b, c, d) in $\{(4, 1, 1, 4), (3, 1, 1, 6), (3, -1, 11, 12)\}$
   (a) For x and y in $[1, \sqrt{N}]$, if $0 < ax^2 + by^2 < N$ and $ax^2 + by^2 \equiv c \bmod d$, negate $L[ax^2 + by^2]$.
   (b) For i in $[0, N[$, set $P[i]$ to **True** if $L[i]$ is **True**
   (c) Reinitialize L with all the cells to **False**
4. For i in $[0, N[$, if $P[i]$ is **True** and i is squarefree, report i.

The step summarized in Item 3a enumerates $O(N)$ pairs and then, the time complexity of these steps is in $O(N)$. The time complexity of the squarefree

elimination by sieving is also in $O(\sum_{p=2}^{\sqrt{N}} N/p^2) = O(N)$. The total time complexity is therefore in $O(N)$. The memory complexity is also in $O(N)$. An efficient algorithm described in [9] using wheel sieve and segmented sieve allows to have a running time in $O(N/\log\log N)$ with a memory complexity in $N^{1/2+o(1)}$. The `primegen` software [29] implements this algorithm.

2.2.3 Complexities of some sieve algorithms

In this section, we summarize in Table 2.1 the space and time complexities of some sieve algorithms. Several sieve algorithms have been previously described and we mention some other to be more complete about the sieves available to find prime numbers. We do in addition two comments that are not obvious using only the data we summarized:

- the implementation of the Bennion's "hopping sieve" by Galway has less cache misses than the one of the segmented sieve [69, Section 5].
- the pseudosquares prime sieve is faster than the exhaustive search using Miller–Rabin primality test and achieve the best space complexity [174].

Algorithm	Time	Space	References
Eratosthenes	$O(N \log\log N)$	$O(N)$	Section 2.1.1
Segmented sieve	$O(N \log\log N)$	$O(N^{1/2})$	Section 2.1.2
Segmented wheel	$O(N)$	$O(N^{1/2+o(1)})$	Section 2.1.3
Composite	$O(N)$	$O(N)$	Section 2.2.1
Segmented wheel composite	$O(N/\log\log N)$	$O(N^{1+o(1)})$	[151, 152, 56, 173]
Hopping sieve	$O(N \log\log N)$	$O(N^{1/2})$	[69]
Quadatic form	$O(N)$	$O(N)$	Section 2.2.2
Segmented wheel quadatic form	$O(N/\log\log N)$	$O(N^{1/2+o(1)})$	[9]
Dissected quadatic form	$O(N)$	$O(N^{1/3})$	[70, 71]
Hybrid sieve	$O(N \log\log N)$	$O(N^{1/4})$ (conjectured)	[71]
Pseudosquares prime sieve	$O(N \log N)$	$O(\log^2 N)$	[174]

Table 2.1 – Complexity of the sieves.

2.3 Sieve of Eratosthenes in different contexts

2.3.1 Perfect number

As a toy example of what can be done with a sieve, let now consider the enumeration of the perfect numbers. A perfect number is an integer for which the sum of its divisors (excluding itself) is equal to itself. Let N be an integer and consider the perfect numbers in $[1, N[$. To determine if an integer a is perfect, we need to find all its divisors, and not only its prime factors. Then, for all integer i less than $\lfloor N/2 \rfloor$, we need to record that for all integer $1 < k < N/i$, the integer ki is divisible by i.

The array A used for sieving now contains integers instead of booleans. Each index i of the array A correspond to the integer $i + 1$. A simple procedure to find perfect numbers in $[1, N[$ can be described as follow:

1. Initialize the array A with a 0 in each cell.
2. For i in $[1, \lfloor N/2 \rfloor[$
   (a) Set $k = 2i$.
   (b) While $k < N$, add i to $A[k]$ and set k to $k + i$.
3. For i in $[1, N[$, if $i = A[i-1]$, report i.

The time complexity of this algorithm in $O(\sum_{i=1}^{N/2} N/i) = O(N \log N)$ and the memory complexity is obviously in $O(N \log N)$. To factor any number, we can use a similar approach by considering primes and prime powers. This is the goal of the next algorithm to find smooth numbers.

2.3.2 Smooth numbers

Let B_0 be an integer. In this section, we look for B_0-smooth numbers, that is numbers in an interval $[I, J[$ for which the largest prime factor is less than B_0, as in Definition 1.5.

A first algorithm

Let us describe the idea of this algorithm before a more formal description. Let $\mathcal{P}$ be the set of primes strictly less than B_0. To find B_0-smooth integers, we want to remove the contribution of $p^a < J$, with p in $\mathcal{P}$ and a a positive integer, on all the integers in $[I, J[$. Removing the contribution of p^a, with $a > 0$, is performed by removing a times the contribution of p.

1. create an array L indexed from 0 to $J - I - 1$ initialized with the integers in $[I, J[$;
2. for prime $p < B_0$
   (a) set a to 1;
   (b) while $p^a < J$
      i. find the first multiple of p^a greater than I and compute its index i in L;
      ii. while $i < J - I$, divide $L[i]$ by p and set i to $i + p^a$;
      iii. increment a;
3. for i in $[0, J - I[$, if $L[i] = 1$, then $i + I$ is B_0-smooth.

In order to reduce the memory size of the array L, we can store an approximation of the logarithm of the integers. We then need to subtract the logarithm of p to $L[i]$, instead of dividing by p. To know if $i + I$ is B_0-smooth, it suffices to perform the test $L[i] = 0$ but of course, we must be careful with approximations. From now on, we consider that this change will be applied in the previous algorithm.

This algorithm is quite powerful if $J - I$ is greater than B_0, but when $J - I \ll B_0$, the number of updates in the array L is less than 1 on average. The following section present a strategy to find smooth numbers more efficiently.

Algorithm in short interval

In the following algorithm, the sieving procedures do not give the exhaustive list of B_0-smooth integers in $[I, J[$, but are faster and give a large subset.

Remarks on the smoothness probability. A simple observation on B_0-smooth numbers shows that a B_0-smooth number is often divisible by many *small* primes. For example, if $I = 1$, $J = 2^{20}$, $B_0 = 2^{16} - 1$ and $B_1 = 2^8 + 7$, the number of B_0-smooth numbers is equal to 846,695 and the number of B_1-smooth numbers is 173,552, that is more than 20% of the B_0-smooth integers are B_1-smooth. When B_1 is equal to $2^{12} + 3$, the proportion is about 65%. Let now briefly study the B_0-smoothness probability of $n = ab$, with n, a and b three integers. Without knowledge on the form of n, the B_0-smoothness probability of n is more or less equal to u^{-u}, with $u = \log(n)/\log(B_0)$ by Theorem 1.1. The probability of n to be B_0-smooth is also equal to the probability of a and b to be simultaneously B_0-smooth. If a and b are of the same size, this probability become $((u/2)^{-u/2})^2$, that is larger than u^{-u} by a factor 2^u. Then, we can infer that if a number is divisible by many *small* integers, its probability of smoothness is larger than for a random integer of the same size. This intuition will be used in the next paragraph.

A new algorithm. When the sieving step seems to be uninteresting, like in the situation described before when the interval is too small, we can try to have a compromise between the sieving step and a more exhaustive search algorithm. The goal of the sieving step is to distinguish between promising B_0-smooth number and almost doubtless not B_0-smooth integers. To make this distinction, we use what is called a *threshold*. Each prime we sieve with the above algorithm remove its contribution in the array L. Then, if the remaining value in $L[i]$ is smallest than the original value, there exist some primes less than B_0 which divide $i + I$. The threshold T is used to decide which are the promising B_0-smooth numbers: if the value stored in $L[i]$ is less than T, then we hope that $i + I$ is B_0-smooth and we compute the full factorization of $i + I$ to verify if it is really B_0-smooth.

Here, $B_1 < B_0$ is called the sieving bound, while B_0 is the smoothness bound. We now rewrite the previous algorithm taking into account this idea.

1. create an array L indexed from 0 to $J - I - 1$ initialized with the logarithm of the integers in $[I, J[$;
2. for all the prime p strictly less than B_1
   (a) let $a = 1$;
   (b) while $p^a < B_1$
      i. find the first multiple of p^a greater than I and compute its index i in L;
      ii. while $i < J - I$, subtract $\log p$ to $L[i]$ and set i to $i + p^a$;
      iii. increment a;
3. for i in $[0, J - I[$, if $L[i] \leq T$, factorize $i + I$ and test if it is B_0-smooth.

Setting B_1 to B_0 and T to 0, we recover the first algorithm. Setting B_1 to 0 and T to $\log J$, we perform an exhaustive search. With such an algorithm, we are aware of the possibility of missing some B_0-smooth integer in $[I, J[$. We need to find parameters that achieve a good compromise between the time spent during the sieving step, the time of the factorization step and a small number of false positives. The parameters need therefore to be carefully selected.

Choice of parameters. The previous algorithm is composed by two main steps. To adjust the parameters of this algorithm, we must analyze carefully the cost of these two steps. The sieving cost, summarized in Item 2, is equal to $(J-I)\log\log B_1$. For the factorization step, summarized in Item 3, the cost of one individual B_0-smooth test is equal to $L_{B_0}(1/2, \sqrt{2})$ using the ECM algorithm [130]. This test is performed $(J-I)\pi_{I,J}(T, B_1)$, where $\pi_{I,J}(T, B_1)$ is the proportion of integers that are not B_1-smooth and whose remaining value is less than T, also called *survivors*. The total cost of the algorithm to find B_0-smooth integer using a sieving up to B_1 and a factorization step on survivors is equal to $(J-I)(\log\log B_1 + \pi_{I,J}(T, B_1)L_{B_0}(1/2, \sqrt{2}))$. It must be compared with the cost of the first sieving algorithm which is $(J-I)(\log\log B_0)$. The second algorithm is therefore more efficient as the first one if $\log\log B_1 + \pi_{I,J}(T, B_1)L_{B_0}(1/2, \sqrt{2}) < \log\log B_0$. Furthermore, the second algorithm must report a *sufficient* number of B_0-smooth integers in $[I, J[$, say almost 90% of them. The theoretical estimation of this number and the proportion $\pi_{I,J}(T, B_1)$ is related to the number of k-semismooth integers, that are integers having exactly k prime factors between B_0 and B_1, and can be hard to compute, see for example the theses of Cavalar [45, Chapter 2] and Ekkelkamp [58, Chapter 2]. Adjusting the parameters can also be done empirically, and it is often the chosen way.

Dividing the search space

Let us now consider, in this last section, a way to deal with interval $[I, J[$ of a large length, where it is impossible to store the needed informations to test the smoothness of all the elements in $[I, J[$. A possible way will be to use a sieve procedure close to the one of the segmented sieve. In this section, we describe an other way based on the sublattices of $\mathbb{Z}$. This is a one-dimensional version of the well known special-$\mathfrak{Q}$ sieving, that is used in dimension 2 in the classical NFS algorithm, described in Section 3.2.3.

Let q be a prime less than B_0. Let us consider the sublattice Λ_q of $\mathbb{Z}$ which contains elements divisible by q. The intersection of Λ_q and $[I, J[$ is denoted by Λ'_q and contains almost $(J-I)/q$ integers. In the following, we consider only prime q such that the number of elements in Λ'_q is small enough to store in memory all the needed informations (that is essentially the logarithm of each element) to perform the sieve algorithm and find B_0-smooth numbers. The set of such primes is denoted by $[B_2, B_3[$. Inside a set Λ'_q, we remove the contribution of q. Then, we apply a sieving procedure, which can be the same as the previous one: for each element of Λ'_q, we remove the contribution of the prime p, and its power, less than a bound B_1 and if the remaining factor is smaller than a threshold T, implying that the element have a good chance to be B_0-smooth, we factorize it and report it, if it is B_0-smooth. The complete description of the algorithm is:

1. for all prime q in $[B_2, B_3[$
   (a) create an array L indexed from 0 to $\#\Lambda'_q - 1$ initialized with the logarithm of the integers in Λ'_q;
   (b) in each cell of L, subtract $\log q$;
   (c) for all the prime $p \neq q$ strictly less than B_1
      i. let $a = 1$;
      ii. while $p^a < B_1$
         A. find the first multiple of p^a in Λ'_q greater than I and compute its index i in L;
         B. while $i < \#\Lambda'_q$, subtract $\log p$ to $L[i]$ and set i to $i + p^a$;
         C. increment a;
   (d) for i in $[0, \#\Lambda'_q[$, if $L[i] \leq T$, factorize $q(i + \lceil I/q \rceil)$ and test if it is B_0-smooth.

Before describing the classical choice of the parameters, the advantages and the drawbacks of this sieve algorithm, we will define what we call a *duplicate*.

Definition 2.1 (Duplicate)**.** Let N be a B_0-smooth numbers in $[I, J[$. If N is divisible by the primes q and p in $[B_2, B_3[$, where $B_3 < B_0$, using the sieve algorithm above, the integer N will be probably reported two times, when we consider the set Λ'_q and Λ'_p: the integer N is a duplicate.

We will first describe obvious constraints on the parameters:

- the bounds B_2 and B_3 should be the largest possible to decrease the number of duplicate.
- the bound B_3 should be chosen such that $\#\Lambda'_q$ is sufficiently large, where q is the largest possible in $[B_2, B_3[$.
- the bound B_2 should be chosen such that $\#\Lambda'_q$ fit into memory, where q is the smallest possible in $[B_2, B_3[$.
- it is interesting to sieve if $\#\Lambda'_q/B_1$ is larger than 1.

Classically, the bound B_1 is often set to be less than q, in order to report less duplicates. Some advantages and drawbacks of this sieve algorithm are summarized in Table 2.2. With a correct choice of parameters, we hope that the advantages counterbalance the drawbacks, mainly in term of running time. We will develop below the advantages and the drawbacks, by comparing this sieve algorithm, called *special-q method*, with a segmented sieve algorithm that reaches the same goal.

Memory and parallelization. These two features are shared by the two algorithms. Indeed, the length of the segmented sieve can be chosen to fit into memory, as the cardinality of each Λ'_q in the special-q method. During the description of the segmented sieve, we showed that, given the location of a hit of a prime in a segment, we can compute the location of the next hit in another segment, which gives an advantage if we consider the natural sequence of the segments. If we treat each segment independently, we cannot therefore use the previous advantage, but the segments can be treated in parallel, as we can treat each Λ'_q independently.

Advantages	Drawbacks
• do not explore numbers divisible by a factor larger than B_0 • fit into memory • easily parallelizable	• miss (B_2-1)-smooth numbers • generates duplicates

Table 2.2 – Advantages and drawbacks of the special-q method.

Completeness and duplicates Using the segmented sieve will divide in many contiguous subsets of $[I, J[$ such that, putting altogether these subsets, we cover exactly all the elements in $[I, J[$. If we consider that the parameters B_1 and T are sufficiently well designed to report all the B_0-smooth numbers in all the segment, we then report all the B_0-smooth integers in $[I, J[$.

The use of the special-q method covers differently the interval $[I, J[$. If we combine all the sets Λ'_q, we have no guarantee that putting altogether these sets, we can cover the whole interval $[I, J[$. It is obvious that, if N in $[I, J[$ is (B_2-1)-smooth, then it will never be reported by any Λ'_q, where q is in $[B_2, B_3[$, of the special-q method. Therefore, the set of reported B_0-smooth integers by the special-q method is not complete.

However, this drawback implies an advantage. In the interval $[I, J[$, there exist many elements of the form ab, where a is a product of small prime and b is a prime, or a product of primes, larger than B_0. These elements are always considered by the segmented sieve and always ignored by the special-q method.

Finally, the special-q method implies necessarily to deal with duplicates. The number of duplicates can be small if the interval $[B_2, B_3[$ contains a small number of primes and the bounds are relatively large. Using the segmented sieve, we do not need to deal with duplicates, because the intersection of each segment is empty.

Chapter 3

The number field sieve algorithm in prime fields

The number field sieve (NFS) algorithm to compute discrete logarithms is a variant of NFS to factor large integers [127]. In this chapter, we focus on NFS in prime fields. The different variants for extension fields will be presented in Part II.

The NFS algorithm is an index calculus algorithm, and follows the description given in Section 1.3. In this chapter, our target finite field is $\mathbb{F}_p$, where p is a prime number. Let f_0 and f_1 be two irreducible polynomials with integer coefficients sharing a common root m modulo p. Let ℓ be the largest prime factor of $p-1$. The major difference with the index calculus algorithm sketched in Section 1.3 is the relation collection. We first describe a general overview of NFS and especially the relation collection before going into details.

Let K_0 be a number field defined as $\mathbb{Q}[x]/f_0(x) = \mathbb{Q}(\theta_0)$, where θ_0 is a root of f_0. Let $\mathcal{O}_0$ be the ring of integers of K_0. Let ν_0 be the map from $\mathbb{Z}[x]$ to K_0, which maps x to θ_0, and ρ_0 be the map from K_0 to $\mathbb{F}_p$, which maps θ_0 to m modulo p. Let $K_1 = \mathbb{Q}(\theta_1)$, $\mathcal{O}_1$, ν_1 and ρ_1 be likewise defined. We can then build the typical commutative diagram, as in Figure 3.1. Indeed, for an integer polynomial a, we have $\rho_0(\nu_0(a)) = \rho_1(\nu_1(a))$.

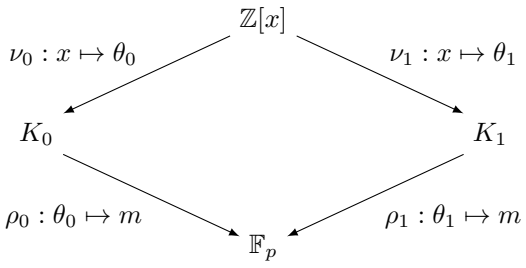


Figure 3.1 – The NFS diagram to compute discrete logarithms in $\mathbb{F}_p$.

Instead of performing the relation collection directly in $\mathbb{F}_p^*$, we perform the relation collection in the rings of integers $\mathcal{O}_0$ and $\mathcal{O}_1$. Let B_0 and B_1 be two

integers, called *smoothness bounds* or *large prime bounds*. The factor base $\mathcal{F}_0$ (respectively $\mathcal{F}_1$) is the set of prime ideals in $\mathcal{O}_0$ (respectively $\mathcal{O}_1$) of norms less than B_0 (respectively B_1). The factor bases also contain prime ideals dividing the leading coefficient of the polynomials f_0 and f_1, to take into account that θ_0 and θ_1 are not necessarily algebraic integers.

Let a be a polynomial in $\mathbb{Z}[x]$. We say that the principal ideal $\langle a(\theta_0)\rangle$ is B_0-smooth if it completely factors into prime ideals of $\mathcal{F}_0$. We get a relation if $a(\theta_0)\mathcal{O}_0 = \prod_{\mathfrak{Q}\in\mathcal{F}_0} \mathfrak{Q}^{\mathrm{val}_{\mathfrak{Q}}\, a(\theta_0)}$ is B_0-smooth and $a(\theta_1)\mathcal{O}_1 = \prod_{\mathfrak{R}\in\mathcal{F}_1} \mathfrak{R}^{\mathrm{val}_{\mathfrak{R}}\, a(\theta_1)}$ is B_1-smooth. Instead of performing the factorization over ideals, we factorize the norm of $a(\theta_0)$ (respectively $a(\theta_1)$), defined as $\pm \mathrm{lc}(f_0)^{\deg a} \mathrm{Norm}\,(a(\theta_0)) = \mathrm{Res}\,(f_0, a)$ (respectively $\pm \mathrm{lc}(f_1)^{\deg a} \mathrm{Norm}\,(a(\theta_1)) = \mathrm{Res}\,(f_1, a)$) which is a rational. The B_0-smoothness of $\langle a(\theta_0)\rangle$ is then defined as the B_0-smoothness of the resultant between a and f_0, likewise on the side 1. Then, to find a relation, we consider the smoothness of norms instead of the smoothness of $a(\theta_0)$ and $a(\theta_1)$. Furthermore, knowing the factorization of a norm allows us to find the factorization into ideals almost for free. Complexity analysis shows that a must be of degree one to reach the $L(1/3)$ complexity, we then write $a = a_0 + a_1 x$. The set of possible pairs (a_0, a_1) is often restricted to a search space $\mathcal{S}$, a finite subset of $\mathbb{Z}^2$.

A relation can be transformed in a linear relation involving the virtual logarithms of the ideals [163]. To be valid, this linear relation must involve the Schirokauer maps [164], labeled $\lambda_{f_0,i}$ for i in $[0, r_0[$, where r_0 is the unit rank of K_0, and $\lambda_{f_1,i}$ for i in $[0, r_1[$, where r_1 is the unit rank of K_1. The unit rank is equal to $n_0 + n_1 - 1$, according to the Dirichlet's unit theorem, where n_0 is the number of real roots and n_1 the number of conjugate pairs of complex roots of the polynomial that defines the number field. To avoid to deal with fractional ideals, we use the following results.

Proposition 3.1 ([60, Section 9])**.** *Let $f(x) = \sum_{i=0}^{d} c_i x^i$ with coprime integer coefficients and θ a root of f. Let $J = \langle c_d, c_d\theta + c_{d-1}, c_d^2\theta + c_{d-1}\theta + c_{d-2}, \ldots, \sum_{i=1}^{d} c_i\theta^{i-1}\rangle$. The ideal J has norm $|c_d|$, $J\langle 1, \theta\rangle = (1)$ and for integers a_0 and a_1, $\langle a_0 + \theta a_1\rangle J$ is an integral ideal.*

Let J_0 be defined as in Proposition 3.1 for f_0 and J_1 likewise for f_1. The norm of the integral ideal $\langle a_0 + a_1\theta_0\rangle J_0$ is equal to $\pm \mathrm{Res}\,(f_0, a_0 + a_1 x)$ and that of $\langle a_0 + a_1\theta_1\rangle J_1$ is equal to $\pm \mathrm{Res}\,(f_1, a_0 + a_1 x)$. A relation can therefore be written as the equality

$$\mathrm{vlog}\, J_0 + \sum_{\mathfrak{Q}\in\mathcal{F}_0} \mathrm{val}_{\mathfrak{Q}}\,(a_0 + a_1\theta_0)\,\mathrm{vlog}\,\mathfrak{Q} + \sum_{i=0}^{r_0-1} \lambda_{f_0,i}(a_0 + a_1\theta_0)\,\mathrm{vlog}\,\lambda_{f_0,i} \equiv$$
$$\mathrm{vlog}\, J_1 + \sum_{\mathfrak{R}\in\mathcal{F}_1} \mathrm{val}_{\mathfrak{R}}\,(a_0 + a_1\theta_1)\,\mathrm{vlog}\,\mathfrak{R} + \sum_{i=0}^{r_1-1} \lambda_{f_1,i}(a_0 + a_1\theta_1)\,\mathrm{vlog}\,\lambda_{f_1,i} \bmod \ell.$$

Once we have found more than $\#\mathcal{F}_0 + \#\mathcal{F}_1 + r_0 + r_1$ relations, we put the relations as rows of a matrix M, which must have a right kernel of dimension 1, whose columns are indexed by the prime ideals in $\mathcal{F}_0$ and $\mathcal{F}_1$ and the characters (Schirokauer maps). We then compute a right non-zero kernel vector $\boldsymbol{w}$ of this matrix M; the entries of $\boldsymbol{w}$ give the virtual logarithms of the elements of the factor basis. Knowing these virtual logarithms, we try to compute the virtual

logarithm of an ideal of large norm in one of the number fields by rewriting the target element with some ideals of smaller norm while the involved ideals are those whose virtual logarithm is already known.

3.1 Polynomial selection

To ensure a high smoothness probability and find many relations during the relation collection, the choice of the polynomial pair (f_0, f_1) is crucial. We will describe in the following the two main ways to construct a polynomial pair and begin by describing criteria to determine which pair is the best in term of smoothness probability.

3.1.1 Quality criteria

Let F_0 (respectively F_1) be the homogenization of f_0 (respectively f_1), that is $F_0(a_0, a_1) = f_0(-a_0/a_1)a_1^{\deg f_0}$ (respectively $F_1(a_0, a_1) = f_1(-a_0/a_1)a_1^{\deg f_1}$). The polynomial F_0 (respectively F_1) represents the resultant between an element $a_0 + a_1 x$ and f_0 (respectively f_1). As will be explained in Section 3.2.1, the search space $\mathcal{S}$ has the form $[I_0^m, I_0^M[\times[0, I_1^M[$, where I_0^m, I_0^M and I_1^M are three integers. Moreover, we only consider the pairs (a_0, a_1) in $\mathcal{S}$ such that a_0 and a_1 are coprime.

Size property

A first criterion to estimate the number of relations we can find with $\mathcal{S}$ is to compute the sum of the smoothness probability in both sides, which can be computed as

$$\sum_{\substack{a_0 \in [I_0^m, I_0^M[,\ a_1 \in [0, I_1^M[\\ \gcd(a_0, a_1) = 1}} \rho\left(\frac{\ln|F_0(a_0, a_1)|}{\ln B_0}\right) \rho\left(\frac{\ln|F_1(a_0, a_1)|}{\ln B_1}\right), \tag{3.1}$$

where ρ is the Dickman function.

A simplified and easier criterion is to compute an upper bound of the sizes of the norms corresponding to both polynomials. This is enough for getting the optimal theoretical complexities, and can also be used as a first filter in practice. The maximum of F_0 and F_1 in $\mathcal{S}$ is often reached when $a_0 = I_0^M$ (we suppose here that $|I_0^m| \approx |I_0^M|$) and $a_1 = I_1^M$, because the resultant is very sensitive to the infinity norm of the polynomials [34, Theorem 7]. Then, the sum given in Equation (3.1) can be rewritten as

$$\frac{6}{\pi^2}(I_0^M - I_0^m) I_1^M \rho\left(\frac{\ln|F_0(I_0^M, I_1^M)|}{\ln B_0}\right) \rho\left(\frac{\ln|F_1(I_0^M, I_1^M)|}{\ln B_1}\right), \tag{3.2}$$

where the factor $6/\pi^2$ takes into account the probability of two integers to be coprime.

The first criterion is then to select the polynomial pair for which the sizes of the norms on both sides are minimal. As a first approximation, this minimum can be reached by minimizing the sum of the sizes of norm. Indeed, since the Dickman rho function is convexe, the product $\rho(x_0)\rho(x_1)$ is larger than $\rho(x_0 + x_1)$, and since $B_0 \approx B_1$, Equation (3.2) can be roughly estimated as

$$\frac{6}{\pi^2}(I_0^M - I_0^m)I_1^M \rho\left(\frac{\ln|F_0(I_0^M, I_1^M)| + \ln|F_1(I_0^M, I_1^M)|}{\ln B_0}\right).$$

This can be refined by considering that, in a list of polynomial pairs that yield a similar value for the sum of the sizes of the norms, it is better to choose the pair for which the norms have sizes as close as possible to each other.

Local property

A drawback of Equation (3.1) is that the norms are considered as if they were random integers of a given size. However, this assumption is not verified, because the divisibility properties of norms are not exactly the same as for random integers. To measure the difference in terms of smoothness probability between these two integers, Murphy introduces in his thesis [140, Section 3.2] the α quantity, that depends on the polynomial that defines the number field. Then, if the size of a norm is N (the size is given by the natural logarithm) in the number field K_0, its probability of smoothness is about the same as the one of a random integer of size $N + \alpha(f_0)$. We then look for negative α quantities. The formal definition of the α quantity is obtained as a sum of local contributions:

Definition 3.1. Let f be an irreducible polynomial in $\mathbb{Z}[x]$, and F be its homogenization. The quantity $\alpha(f)$ is defined as $\alpha(f) = \sum_{\ell \text{ prime}} \alpha_\ell(f)$ with, for all prime ℓ,

$$\begin{aligned}\alpha_\ell(f) = \ln(\ell)\big[&\mathbb{A}(\mathrm{val}_\ell(n), n \in \mathbb{Z}) \\ &- \mathbb{A}(\mathrm{val}_\ell(F(a_0, a_1), (a_0, a_1) \in \mathbb{Z}^2 \text{ and } \gcd(a_0, a_1) = 1))\big],\end{aligned}$$

where $\mathbb{A}(\cdot)$ is the average value and val_ℓ the ℓ-adic valuation.

The average value $\mathbb{A}(\cdot)$ is defined here by taking the limit of the average value of the quantity for increasingly large finite subsets of the whole set considered. These subsets are chosen to be centered balls of increasing radius. The convergence of the series definition $\alpha(f)$ is proved in the article of Barbulescu and Lachand [23].

Let f and F be as in Definition 3.1. In this case, when f has only simple roots modulo ℓ, we can get explicit formulæ for $\mathbb{A}(\mathrm{val}_\ell(n), n \in \mathbb{Z})$ and $\mathbb{A}(\mathrm{val}_\ell(F(a, b), (a, b) \in \mathbb{Z}^2 \text{ and } \gcd(a, b) = 1))$. The first term can be written as $1/(\ell - 1)$. The second term is equal to $\ell n_\ell/(\ell^2 - 1)$, where n_ℓ is the number of simple roots of f modulo ℓ. Then, we can write $\alpha(f)$ as $\sum_{\ell \text{ prime}} \ln(\ell)[1/(\ell - 1) - \ell n_\ell/(\ell^2 - 1)]$. A sketch of a proof for this formula is given in Appendix C.

Global property

Taking the α quantities into account, we can then rewrite our estimation of the number of relations given in Equation (3.1):

$$\sum_{\substack{a_0 \in [I_0^m, I_0^M[,\ a_1 \in [0, I_1^M[\\ \gcd(a_0, a_1) = 1}} \prod_{i=0}^{1} \rho\left(\frac{\ln|F_i(a_0, a_1)| + \alpha(f_i)}{\ln B_i}\right). \tag{3.3}$$

The number of relations given by Equation (3.3) is somehow difficult to estimate and the simplification given in Equation (3.2) is too rough. A way to

approximate this large sum is to select (a_{0_k}, a_{1_k}) as points of an ellipse that approximates the shape of the rectangle $\mathcal{S}$: explanations on why this simplification is a good approximation of the number of relations in $\mathcal{S}$ is given in [140, Section 5.2]. Let K be the number of points (a_{0_k}, a_{1_k}), for k in $[0, K[$, regularly spaced on the ellipse. By computing the average value of the smoothness probability of $F_0(a_{0_k}, a_{1_k})$ and $F_1(a_{0_k}, a_{1_k})$, we obtain an approximation of the number of relations that we can get in $\mathcal{S} = [I_0^m, I_0^M[\times[0, I_1^M[$ as

$$(I_0^M - I_0^m) I_1^M \left[\frac{1}{K} \sum_{k=0}^{K-1} \prod_{i=0}^{1} \rho\left(\frac{|\ln F_i(a_{0_k}, a_{1_k})| + \alpha(f_i)}{\ln B_i} \right) \right]. \tag{3.4}$$

The Murphy E quantity [140, Chapter 5] is precisely the quantity between the square brackets in Equation (3.4). The number of real roots of a polynomial tends to increase the Murphy E quantity. The shape of the set $\mathcal{S}$ must of course takes into account the repartition of norms of about the same size, the Figure 3.2 shows some isonorms, using the polynomial f of the 768-bit record [122].

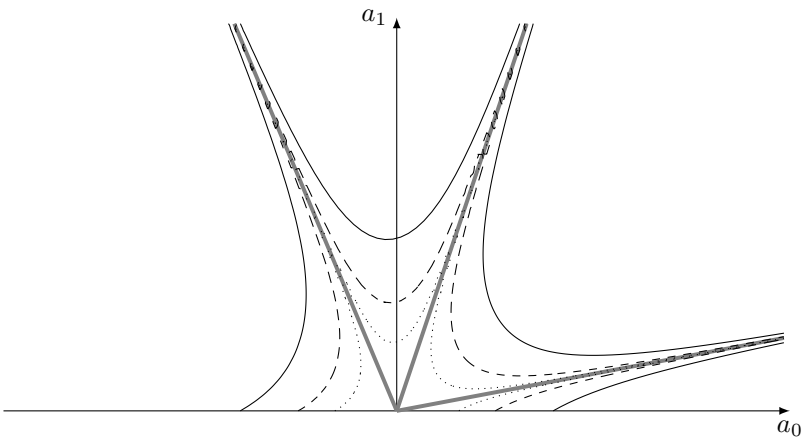


Figure 3.2 – Different isonorms for (a_0, a_1) pairs, where the real roots of f are the slopes of the gray lines.

3.1.2 Generation of polynomial pairs

There exist two popular ways to construct the polynomial pair (f_0, f_1). The resulting properties of the two polynomial selections will be summarized in Table 3.1.

Common method (base-m)

This first method is the classical method used in NFS to factor integers, called the base-m method. One chooses an integer m whose size will be determined later. The polynomial f_0 is defined by $x - m$ and the coefficients b_i of $f_1(x) = \sum_i b_i x^i$ are given by the expansion of p in basis m, as $p = \sum_i b_i m^i$, with b_i in $[0, m[$. The coefficients are in general in $O(m)$.

There exist some ameliorations of this polynomial selection, generally described for the factorization. A first one is proposed in Murphy's thesis [140,

Chapter 5]: by choosing the degree d and the leading coefficient b_d of f_1, we can build a polynomial pair (f_0, f_1) by choosing the root $m = \lfloor (p/b_d)^{1/d} \rfloor$ and computing the other coefficients of f_1 such that $f_1(m) = p$. Then, the coefficient of b_{d-1} has magnitude db_d. The second improvement is described in [116, 117]. It allows to find two polynomials with a common root $m = m_0/m_1 \bmod p$ with $f_0 = m_1 x - m_0$ and f_1 of degree d such that $f_1(m_0/m_1)m_1^d$ is equal to $\pm p$. With the appropriate algorithm, we can find, given b_d, the coefficient b_{d-1} such that $|b_{d-1}| \leq db_d$ and the other coefficients have the same size as m. Another improvement is to use rotations and translations, to reduce the size of the coefficients b_{d-2} and b_{d-3} [140, 12]. A translation of the pair (f_0, f_1) is the pair $(f_0(x+k), f_1(x+k))$ for some integer k, and a rotation of (f_0, f_1) is the pair $(f_0, f_1 + \lambda f_0)$ for some polynomial λ. A last improvement is to have more freedom about the selection of f_1 by considering that $|f_1(m_0/m_1)m_1^d|$ can be a multiple of p: a suitable multiple can be computed thanks to a lattice reduction [12, Section 3.3], that counterbalance the increasing of the norm. Some of these methods were used in the polynomial selection done to compute discrete logarithms in $\mathbb{F}_p^*$, where p was 596-bit long [40].

Joux–Lercier method

This polynomial selection proposed by Joux and Lercier [103] allows us to find two polynomials of degree d and $d+1$, one with small coefficients. This polynomial selection was used to compute discrete logarithms in $\mathbb{F}_p^*$, where p was of size 768 bits [122].

Choice of the polynomials. Let f_0 be a polynomial of degree $d+1$ with small coefficients and a root m modulo p. Contrary to the base-m method, the root m is deduced from the polynomial f_0. We need to find a polynomial f_1 with the same root modulo p. Let j be the degree of f_1. The sets of valid polynomial f_1 can be described as linear combinations of polynomials px^i for i in $[0, j]$ and polynomials $(-m^k + x^k)$ for k in $[1, j]$. We can observe that for i in $[0, j]$, the polynomial px^i is equal to $m^i p + p(-m^i + x^i)$, then the sets of polynomial f_1 can be described by $\lambda_0 p + \sum_{i=1}^{j} \lambda_i(-m^i + x^i)$, where the λ_i are integers. By choosing adapted λ_i, we can reduce the size of the coefficients. To find a polynomial f_1 with the smallest possible coefficients, it suffices to find a short vector of the lattice for which a basis is given by the following vectors of length $j+1$: $\{(p, 0, 0, \ldots, 0), (-m, 1, 0, 0, \ldots, 0), (-m^2, 0, 1, 0, 0, \ldots, 0), \ldots, (-m^j, 0, 0, \ldots, 0, 1)\}$. The coefficients $(b_0, b_1, \ldots, b_d)$ of this short vector define the polynomial $f_1 = \sum_{i=0}^{j} b_i x^i$. The coefficients of f_1 are more or less in $O(p^{1/(j+1)})$, according to Theorem A.1. Once the coefficients of the polynomial f_1 are found, it remains to check if the polynomial f_1 is irreducible, which is practically always the case.

Remark 3.1. By setting d to 0, the Joux–Lercier polynomial selection is equivalent to the base-m method. The case $d = 1$ is equivalent to the polynomials defined in the article of Coppersmith–Odlzyko–Schroeppel [52], that is $f_0 = x^2 + 1$ and $f_1 = b_0 + b_1 x$.

Choice of the parameters. We first discuss the parameter j. An upper bound on the norms in the field K_1 is $2^{j/2}(j+1)^{1/2} \max(I_0^M, I_1^M)^j p^{1/(j+1)}$,

following [34]. Therefore, the norms in K_1 decrease when j increases. Then, the degree j of f_1 should not be too small. However, if $j > d$, the shortest vector of the lattice generated by the basis $\mathcal{B}$ will give the coefficients of f_0, because these coefficients are small. The second shortest vector is not guaranteed to have small coefficients. With Theorem A.2, we can infer that if λ_1 is very short, then the other successive minima must be quite large, especially the second minimum. Then, $j = d$ seems to be the best compromise.

To find the parameter d that allows to reach the minimal product of norms in both sides, we can use the fact that the product of the norms is bounded by $2^{(2d+1)/2}(d+1)^{1/2}(d+2)^{1/2}\max(I_0^M, I_1^M)^{2d+1}p^{1/(d+1)}$. This function in d, when I_0^M and I_1^M are fixed, admits a global minimum, we then can find a good approximation of the d that reaches the best first quality criterion given in Equation 3.2.

Properties of the base-m and Joux–Lercier polynomial selections

The two polynomial selections described above give different polynomials, which have different properties summarized in Table 3.1.

Variant	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$
Base-m	1	$p^{1/(d_m+1)}$	d_m	$p^{1/(d_m+1)}$
Joux–Lercier	$d_{\mathrm{JL}}+1$	small	d_{JL}	$p^{1/(d_{\mathrm{JL}}+1)}$

Table 3.1 – Polynomial selection for NFS in $\mathbb{F}_p$.

Concerning the base-m method, the complexity analysis of Schirokauer [164] shows that there exists an optimal value of d_m that reaches the complexity in $L_p(1/3, (64/9)^{1/3})$. As in NFS to factor integers, the degree d_m grows like $(3\log p/\log\log p)^{1/3}$.

For the Joux–Lercier polynomial selection, the complexity analysis due to Commeine–Semaev [47] sets the optimal degree d_{JL} to be close $d_m/2$, that is $d_{\mathrm{JL}} = (3/8\log p/\log\log p)^{1/3}$.

3.2 Relation collection

Once the polynomial pair (f_0, f_1) is chosen according to some quality criteria, we can begin the relation collection of the NFS algorithm. A simple procedure to find relations is to map the polynomial $a_0 + a_1 x$, where (a_0, a_1) is in $\mathcal{S}$, in the two number fields, then computing the norms in both number fields and keeping (a_0, a_1) if the norms are doubly smooth. This simple procedure is however costly because testing for smoothness is not a simple task. We will detail here some methods to improve the running time of the relation collection.

3.2.1 Preliminaries

In this section, we consider a prime ideal $\mathfrak{Q}$ of $\mathcal{O}_0$ of norm q^d, where q is a prime and d is the degree of the ideal. Except in the few cases where q divides the discriminant of f_0, the ideal $\mathfrak{Q}$ can be represented by a pair (q, r), where the polynomial r is of degree d and is a factor of f_0 modulo q. In particular,

the prime ideal $\mathfrak{Q}$ of norm q and degree one will be denoted in the following by $(q, x + \rho)$ where $-\rho$ is a root of f_0 modulo q and ρ in $[0, q[$. From [43, Corollary 5.5], if a_0 and a_1 are two coprime integers and $\mathfrak{Q}$ is a non-zero ideal of $\mathcal{O}_0$ containing $a_0 + a_1\theta_0$, then $\mathfrak{Q}$ is of degree 1. Furthermore, the ideal $\mathfrak{Q}$ of degree one contains $a_0 + a_1\theta_0$ if and only if $a_0 + a_1 x \equiv 0 \bmod (q, x + \rho)$. In this chapter, we only consider ideals of degree one, denoted by $\mathfrak{Q} = (q, x + \rho)$. If a_0 and a_1 are not coprime, the factorization of $a_0 + a_1\theta_0$ into ideals involves the same prime ideals as $(a_0 - a_1\theta_0)/\gcd(a_0, a_1)$, plus a few others of norm dividing $\gcd(a_0, a_1)$. When a_0 and a_1 are not coprime, we do not get a new interesting relation: this is again the relation obtained with $(a_0/\gcd(a_0, a_1), a_1/\gcd(a_0, a_1))$ up to some factors.

Let us consider an integer polynomial $a_0 + a_1 x$. If the ideal factorization of $a_0 + a_1\theta_0$ involves $\mathfrak{Q}$, the ideal factorization of $-a_0 - a_1\theta_0$ involves also $\mathfrak{Q}$. The relation given by $-a_0 - a_1 x$ is therefore the same as the relation given by $a_0 + a_1 x$. We can therefore consider polynomials with a_0 an integer and a_1 a positive integer. These two descriptions explain why, in Section 3.1, during the estimation of the number of relations, we only consider coprime a_0 and a_1 with a_1 a positive integer.

If the ideal factorization of $a_0 + a_1\theta_0$ involves the ideal $\mathfrak{Q} = (q, x + \rho)$, then $a_0 + a_1 x \equiv 0 \bmod (q, x + \rho)$. It follows that the ideal $\mathfrak{Q}$ is also involved in the factorization of $(a_0 + kq) + a_1\theta_0$, where k is an integer. The ideal $\mathfrak{Q}$ is also involved in $(a_0 + k\rho) + (a_1 + k)\theta_0$, because $a_0 + a_1 x + k(x + \rho) \equiv a_0 + a_1 x \equiv 0 \bmod (q, x + \rho)$. Then, the set of valid pairs (a_0, a_1) such that an ideal $\mathfrak{Q} = (q, x + \rho)$ divides $a_0 + a_1\theta_0$ is the lattice generated by $\{(q, 0), (\rho, 1)\}$, called the $\mathfrak{Q}$-lattice.

With this background, we can describe what we do in practice for the relation collection. The smoothness test can be performed with any factoring algorithm, from trial division to the NFS algorithm to factorize integers. The algorithm we use in practice is the ECM, because it depends on the size of the factors we look for. But, even with the ECM, the smoothness test is costly. A way to improve the practical running time of the relation collection is to look for a subset of $\mathcal{S}$ that contains (a_0, a_1) pairs which are doubly smooth with a higher probability than a random pair. We perform an enumeration step to remove the contribution of the small ideals of $\mathcal{F}_0$ and $\mathcal{F}_1$ from the corresponding norms of all the polynomials $a_0 + a_1 x$ with (a_0, a_1) in $\mathcal{S}$. If a pair is marked by a lot of small ideals, this pair has a greater chance to be doubly-smooth. This is the same idea as the threshold idea of Section 2.3.2.

Let us now give a short description of the relation collection using this improvement. It uses two integers: b_0 less than B_0 and b_1 less than B_1, called *enumeration bounds* (called factor base bound in the classical literature), t_0 and t_1 the *thresholds*. The smoothness bounds B_0 and B_1 are also called *large prime bounds*.

Selection. for i in $[0, 1]$,

> **Initialization.** compute the norm in K_i of all the polynomials $a_0 + a_1 x$ with (a_0, a_1) in $\mathcal{S}$ and store them in an array T_i indexed by (a_0, a_1),
>
> **Enumeration.** for all prime ideals $\mathfrak{Q}$ in $\mathcal{F}_i$ of norms below b_i, compute the $\mathfrak{Q}$-lattice and divide by q all the cells at index (a_0, a_1),

such that (a_0, a_1) are in $\mathcal{S}$ and in the $\mathfrak{Q}$-lattice,

Cofactorization. for all coprime pairs (a_0, a_1) in $\mathcal{S}$, if $T_0[(a_0, a_1)]$ is less than t_0 and $T_1[(a_0, a_1)]$ is less than t_1, perform the full factorization of the norm of $a_0 + a_1 x$ in K_0 and K_1. If the norms are smooth for both sides, the pair (a_0, a_1) gives a valid relation.

The enumeration of the (a_0, a_1) pairs in a $\mathfrak{Q}$-lattice can be efficiently performed by sieving, using the following algorithms. We use the term *sieve bounds* instead of *enumeration bounds*.

3.2.2 The sieving algorithms

Let $\mathfrak{Q}$ be the prime ideal of degree one $(q, x + \rho)$ of $\mathcal{O}_0$, with $-\rho$ a root of f_0 modulo q, and $\mathcal{S} = [I_0^m, I_0^M[\times[0, I_1^M[$.

The line sieve

The line sieve (also called *sieving by rows*) is one of the most basic sieving procedures, directly derived from the Eratosthenes sieve presented in Section 2.1. We know that if $a_0 + a_1\theta_0$ is in $\mathfrak{Q}$, then $a_0 + kq + a_1\theta_0$ with k an integer is also in $\mathfrak{Q}$. Then, we can perform a quite similar sieving procedure as the one of Eratosthenes: by setting a_1, and finding an initial a_0 such that $a_0 + a_1\theta_0$ is in $\mathfrak{Q}$, it suffices to add or remove q as long as we stay in the search space $\mathcal{S}$. Line sieve can be described as the simple following procedure to collect all the elements of form $a_0 + kq + a_1\theta_0$, where k is an integer, whose norm is a multiple of q. Let consider the sieving step on the side 0, keeping in mind that the same applies to the side 1:

- For a_1 in $[0, I_1^M[$
  1. Find a_0 such that (a_0, a_1) is in $\mathcal{S}$ and is in the $\mathfrak{Q}$-lattice,
  2. For all integers k such that $I_0^m \leq a_0 + kq < I_0^M$, divide $T_0[(a_0 + kq, a_1)]$ by q and store the result in $T_0[(a_0 + kq, a_1)]$,

This sieving procedure is quite efficient when the norm q of $\mathfrak{Q}$ is less than the size $I_0^M - I_0^m$ of $[I_0^m, I_0^M[$. Indeed, there always exists at least one element (a_0, a_1) in the $\mathfrak{Q}$-lattice for each a_1 in $[0, I_1^M[$. When $q > I_0^M - I_0^m$, the number of hits per line is at most one and most of the time is spent in the first step described in Item 1, and it is better to use an other sieving algorithm.

A first lattice sieve

To tackle the drawback of the line sieve, a first procedure, described by Pollard in [145] and called *sieving by vector* was proposed. This algorithm is not as efficient as the one of Franke and Kleinjung [61] that will be described in Section 6.2.2, and called *lattice sieve* in this manuscript.

We first perform a lattice reduction on the basis $\{(q, 0), (\rho, 1)\}$ that generate the $\mathfrak{Q}$-lattice, which is here a Gaussian reduction since the $\mathfrak{Q}$-lattice is of dimension 2. We obtain two vectors $\boldsymbol{u_0}$ and $\boldsymbol{u_1}$. Then, by doing small linear combinations of these two vectors, we can cover a region that contains $\mathcal{S}$. The

linear combination $b_0\boldsymbol{u_0} + b_1\boldsymbol{u_1}$ can be done by considering coprime b_0 and b_1. To be sure to cover at least the search space $\mathcal{S}$, it suffices to find λ_0 and λ_1 such that $\lambda_0\boldsymbol{u_0} + \lambda_1\boldsymbol{u_1}$ generates a corner of $\mathcal{S}$, using an algorithm to solve the closest vector problem. Then, by finding the minimum and the maximum of λ_0 and λ_1 for the four corners, we can find bounds on b_0 and b_1. We can describe this first lattice sieve as:

1. compute $\boldsymbol{u_0}$ and $\boldsymbol{u_1}$, a reduced basis of the $\mathfrak{Q}$-lattice,
2. for b_0 and b_1 small and coprime
   - if $(a_0, b_0) = b_0\boldsymbol{u_0} + b_1\boldsymbol{u_1}$ is in $\mathcal{S}$, divide $T_0[(a_0, a_1)]$ by q and store the result in $T_0[(a_0, a_1)]$.

3.2.3 Dividing the search space

The enumeration step needs a huge amount of memory: for record computations, $\mathcal{S}$ contains a lot of pairs, much more than 2^{55} for the computation of a discrete logarithm in a 596-bit field [40]. A first way to reduce the memory requirement is to store the logarithm of the norm instead of the norm, but it still requires too much memory. On the running time aspect, the enumeration step can be parallelized: the loop on the prime ideals $\mathfrak{Q}$ can be split in some ranges executed in parallel. But, the small prime ideals will hit a lot, and then we must carefully implement the accesses to the arrays T_0 and T_1. To reduce the amount of memory and to have a high-level parallelization, Pollard proposed the special-$\mathfrak{Q}$ method [145], improving an idea of Davis and Holridge [53]. In addition, the special-$\mathfrak{Q}$ method allows to help the smoothness test, because a medium-to-large factor of the norm is already known. This is a two-dimensional equivalent of what was explained in one dimension at the end of the previous chapter, with essentially the same advantages and drawbacks.

Let Λ be the lattice of all possible pairs (a_0, a_1) mapped to the fields K_0 and K_1. This lattice can be generated by the basis $\{(1, 0), (0, 1)\}$. A $\mathfrak{Q}$-lattice is a sublattice of Λ, made of elements (a_0, a_1) such that all the ideal factorization of $a_0 + a_1\theta_0$ involves $\mathfrak{Q}$. Let $M_\mathfrak{Q}$ be the 2×2 matrix whose rows contain the vectors of the reduced basis of the $\mathfrak{Q}$-lattice and (c_0, c_1) be an element of the $\mathfrak{Q}$-lattice. To compute the coordinates (a_0, a_1) of (c_0, c_1) in Λ, it suffices to compute $(a_0, a_1) = (c_0, c_1)M_\mathfrak{Q}$, as showed in Figure 3.3. In the $\mathfrak{Q}$-lattice, we can use the line sieve and the lattice sieve as in Λ, for which we originally described the sieve algorithms.

To avoid many redundant work, we only consider not so small $\mathfrak{Q}$, that is $\mathfrak{Q}$ such that its norm is larger than the sieving bound and less than the smoothness bound. The pairs (c_0, c_1) are elements of a search space $\mathcal{H}$, a subset of $\mathbb{Z}^2$ such that c_1 is non-negative. Let H_0^m, H_0^M, H_1^M be three integers such that H_0^M is larger than H_0^m. We define the search space as $\mathcal{H} = [H_0^m, H_0^M[\times[0, H_1^M[$. Because we use the same search space $\mathcal{H}$ for all the different special-$\mathfrak{Q}$ lattices, we cannot ensure that an element of this lattice and in $\mathcal{H}$ give exactly an element $(a_0, a_1) = (c_0, c_1)M_\mathfrak{Q}$ in Λ and $\mathcal{S}$, but because $\#\mathcal{H} \ll \#\mathcal{S}$ ($\#\mathcal{H}$ is equal to 2^{31} in our example), (a_0, a_1) is almost always in $\mathcal{S}$. We set the special-$\mathfrak{Q}$ on the side 0, but it can be set on the side 1 without difficulty, and we can then rewrite the filter step as follows, where $\boldsymbol{c} = (c_0, c_1)$:

Selection. For $\mathfrak{Q}$ in $\mathcal{F}_0$ of norms in $]b_0, B_0[$,

Extract lattice. Compute the matrix $M_{\mathfrak{Q}}$ of the special-$\mathfrak{Q}$-lattice,

Sieve. For i in $[0, 1]$,

Initialization. For all $\boldsymbol{c}$ in $\mathcal{H}$, compute the norm in K_i of the polynomial $a_0 + a_1 x$ with $(a_0, a_1) = \boldsymbol{c}M_{\mathfrak{Q}}$ and store them in a two-dimensional array T_i indexed by $\boldsymbol{c}$,

Enumeration. For all prime ideals $\mathfrak{R}$ in $\mathcal{F}_i$ of norms below $b_i < B_i$, compute the $\mathfrak{R}$-lattice and divide by r the cells indexed by $\boldsymbol{c}$, such that $\boldsymbol{c}$ is in the $\mathfrak{R}$-lattice and the $\mathfrak{Q}$-lattice,

Cofactorization. For all $\boldsymbol{c}$, if $T_0[\boldsymbol{c}]/q$ is less than t_0 and $T_1[\boldsymbol{c}]$ is less than t_1, compute $(a_0, a_1) = \boldsymbol{c}M_{\mathfrak{Q}}$ and if (a_0, a_1) is in $\mathcal{S}$ and a_0 and a_1 are coprime, perform the full factorization of the norm of $a_0 + a_1 x$ in K_0 and K_1 and if the norms are really B_i-smooth in both sides, (a_0, a_1) gives a valid relation.

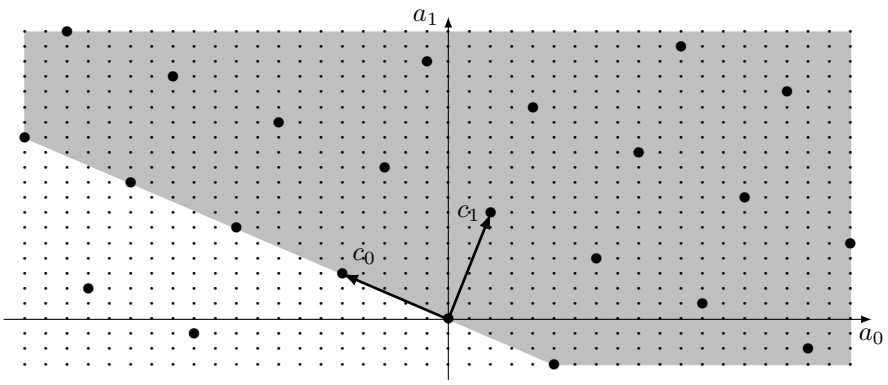


Figure 3.3 – Extracting the $\mathfrak{Q}$-lattice; the gray part is the part explored by (c_0, c_1) in $\mathcal{H}$.

3.3 Linear algebra

In the index calculus algorithms, the linear algebra step is often the bottleneck of the computation. Even if the relation collection was constrained to give a small matrix, two computations were unfeasible using powerful computers, one in 1993 by Gordon and McCurley for the computation of a discrete logarithm in $\mathbb{F}_{2^{503}}$ [78] and another in 2015 for the computation of a discrete logarithm in $\mathbb{F}_{2^{1039}}$ [97]. This difficulty comes from many factors as, among other things, the size of the matrix, the size of the prime ℓ, the need of communication during the computation and the memory cost. If we refer to the computation [40], the number of prime ideals of degree one, that is almost the number of columns of the matrix, is less than $2^{26.3}$ and the number of relations, that is the number of rows of the matrix, is less than $2^{27.4}$. The maximum *weight* of the rows, the

number of non-zero coefficients, is at most 20. The number of empty columns is less than $2^{19.2}$.

We now consider that the number of independent and unique relations is larger than the number of prime ideals of degree one in both number fields, meaning that the system is overdetermined. Using the data of the 596-bit computation [40], the produced matrix is very sparse. This allows us to use some well adapted algorithm to improve the computation of the right kernel of the matrix. The so-called *filtering* step is used to reduce the size of the matrix, without increasing too much the weight. Finally, we apply an algorithm to compute the right kernel of the matrix, which takes advantage of the sparsity of the matrix.

3.3.1 Conditioning of the matrix

The Schirokauer maps

The use of the Schirokauer maps. The first step of the linear algebra is the transformation of the multiplicative relations into linear relations. In the following of this section, we assume for simplicity that f_0 is equal to $x - m$ and f_1 is of degree d.

Let us consider that $a = a_0 + a_1 x$ gives a valid relation. We know that $a_0 + a_1 m = \prod_j q_j^{c_j}$ is smooth and the factorization of its norm is equal to its factorization into ideals. The element $a_0 + a_1\theta_1$ is also smooth and let us denote by $\prod_i \mathfrak{Q}_i^{e_i}$ its factorization in ideals. Let h_1 be the class number of K_1. The ideals $\mathfrak{Q}_i^{h_1}$ are principal ideals and then, $\mathfrak{Q}_i^{h_1} = (\pi_i)$, where π_i is in K_1. By raising $a_0 + a_1\theta_1$ to h_1, we can therefore write an equality between principal ideals as $(a_0 + a_1\theta_1)^{h_1} = (\prod \pi_i^{e_i})$. For a principal ideal, two different generators differ from a unity. Hence, there exists a unit ε of K_1, such that the equality $(a_0 + a_1\theta_0)^{h_1} = \varepsilon(\prod \pi_i^{e_i})$ holds.

To compute the unit ε, we use the Dirichlet's unit theorem. Let U_{K_1} be the group of units in K_1: it is isomorph to $\mathbb{Z}/(n\mathbb{Z}) \times \mathbb{Z}^{r_1}$, where n is the number of units of finite order and r_1 is the unit rank of K_1. Let $\varepsilon_{0,1,\ldots,r_1-1}$ be generators of the units of infinite order. The unit ε is equal to $\zeta \prod_{i=0}^{r_1-1} \varepsilon_i^{g_i}$, where ζ is a torsion unit and g_i are integers. We can now write an equality after mapping all the quantities to $\mathbb{F}_p^*$: the element $(a_0 + a_1 x)^{h_1}$ yields $(\zeta \prod_{i=0}^{r_1-1} \varepsilon_i^{g_i}) \prod_i \pi_i^{e_i} = \prod_j q_j^{c_j h_1}$. The logarithm of ζ modulo ℓ is equal to 0 if ℓ and n are coprime, then we write the previous relation as a additive expression by taking its logarithm modulo ℓ, that is $\sum_{i=0}^{r_1-1} g_i \log \varepsilon_i + \sum_i e_i \log \pi_i(m) \equiv h_1 \sum_j c_j \log q_j \bmod \ell$. If h_1 and ℓ are coprime (note that it is not checked during a practical computation), we can divide the last expression by h_1. This completely explicit approach can be done for very specific number fields. However, computing the class numbers, the generators π_i and the units is in general as costly as computing a discrete logarithm, that is why we use the Schirokauer maps.

Without details, Schirokauer maps are r_1 independent maps $(S_0, S_1, \ldots, S_{r_1-1})$ from K_1^* to $\mathbb{Z}/\ell\mathbb{Z}$, such that the relation induced by a doubly-smooth $a = a_0 + a_1 x$ can be written as a linear relation: $\sum_{i=0}^{r_1-1} S_i(a_0 + a_1\theta_1) \operatorname{vlog} S_i + \sum_i e_i \operatorname{vlog} \mathfrak{Q}_i \equiv \sum_j c_j \operatorname{vlog} q_j \bmod \ell$, where $\operatorname{vlog} S_i$, $\operatorname{vlog} \mathfrak{Q}_i$ and $\operatorname{vlog} q_j$ are unknowns called *virtual logarithms*. In this case, where side 0 is rational, the virtual logarithm $\operatorname{vlog} q_j$ coincide exactly with the usual discrete logarithm.

Computation of the Schirokauer maps. Let z be in K_1 and $S_i(z)$ be the r_1 Schirokauer maps of z. Let $f_{1,i}$ be the factors of f_1 modulo ℓ. We assume that there are no multiplicities. By the Chinese remainder theorem, the algebra $(\mathbb{Z}/\ell\mathbb{Z})[x]/f_1(x)$ is isomorphic to $\prod_i(\mathbb{Z}/\ell\mathbb{Z})[x]/f_{1,i}(x)$. Let s be the least common multiple of $\ell^{\deg f_{1,i}} - 1$, then, if z_0 is in $(\mathbb{Z}/\ell\mathbb{Z})[x]/f$, $z_0^s = 1$ by the Fermat's little theorem. The same holds for z, that is $z^s \equiv 1 \bmod \ell$. By computing in $((\mathbb{Z}/\ell^2\mathbb{Z})[x]/f_1(x)$ the expression $z^s \bmod \ell^2$, we obtain that $z \equiv 1 + \ell W(\theta_1) \bmod \ell^2$, where $W(\theta_1) = W_0 + W_1\theta_1 + \ldots + W_{d-1}\theta_1^{d-1}$ has coefficient in $\mathbb{Z}/(\ell\mathbb{Z})$. The Schirokauer maps $S_i(z)$ can be taken as random linear combinations of the W_i.

Filtering

The filtering step is used to reduce the size of the matrix. Even if this step is not taken into account in the theoretical analysis, ignoring filtering makes a practical computation impossible. At the end of the filtering step, the matrix is square of dimension N and has rank $N - 1$, with a subsequent smaller size and a larger but reasonable weight. In our example, the weight after filtering is 150 on average, and the size of the matrix decreases to $2^{22.8}$. The weight is therefore increased by a factor less than 8, and the size is divided by a factor around 11. We briefly describe some algorithms to perform this task, and refer to the works of LaMacchia–Odlyzko [124] and Pomerance–Smith [150] as well as the theses of Cavalar [45, Chapter 3] and Bouvier [41, Chapter 5] for more information.

The first step of the filtering step is the *singleton* removal. A singleton is a prime ideal that occurs only one time over all the relations. It appears in the matrix as a column of weight 1. There exists a unique relation involving this ideal and then, its discrete logarithm can be computed knowing all the virtual logarithms of the other ideals involved in this relation and those of the corresponding Schirokauer maps. This column can be deleted, as well as the row corresponding to the relation. We still keep apart the relation for later use, after the linear algebra step.

The second step is the *clique* removal. If an ideal is involved in only two relations, and if one of the relations is removed, then it creates a singleton which is removed as presented previously. Then, we remove two rows and a column. Generically, each step of the clique removal removes two rows and one column, but sometimes two columns can be removed. As for the singleton removal, we need to keep track of the removed relations. Knowing which cliques are the most interesting to be removed is done by computing the weight of a clique. We perform these two removals until the matrix is square.

Remark 3.2. In the context of the filtering step, a clique is a connected component, but this is not the same definition in graph theory. The word is however used in the CADO-NFS software [176], in the theses of Cavalar [45, Chapter 3] and Bouvier [41, Chapter 5] and in the article on the computation of the RSA-768 integer [120].

Once we have a square matrix, finally, a structured Gaussian elimination is performed to create a singleton and then reduce by one the number of rows and columns. Contrarily to the removals, this elimination increases the weight of the matrix. Let us consider a column of weight 2: the two rows of the matrix

that have a non-zero coefficient on the shared column can be linearly combined to obtain a new row with a zero on the shared column. By replacing one of the original row with this new row, we create a singleton, and can apply the singleton removal. This mechanism is performed as long as the average weight of the rows is less than the targeted weight.

3.3.2 Linear algebra

There exist two main algorithm families to solve linear algebra problems, the *direct methods* and the *iterative methods*. Before giving a bird's-eye view of the Wiedemann algorithm, we begin by discussing about the choice of the best algorithm family in our context. Let M be a matrix of size N produced at the end of the filtering, with γ the average number of coefficients per row.

Choice of the solver

The direct methods are classical algorithms to solve numerical linear algebra problems. These algorithms are for example the Gaussian elimination, the Cholesky decomposition, LU decomposition and the QR decomposition [74] with some improvements, like the one of Bouillaguet and Delaplace [39]. These methods require $O(N^\omega)$ operations in $\mathbb{F}_\ell$, where ω equals 3 with the naive algorithm to perform matrix multiplications and 2.81 if the Strassen algorithm is used. As mentioned in Section 3.3.1, Gaussian elimination densifies the matrix, and it is the case for all these algorithms. Then, the space complexity of these algorithms is in $O(N^2)$.

The two major algorithms in the iterative method family are the Lanczos [125] and the Wiedemann [183] algorithms, with their block variants found independently by Coppersmith [50] and Montgomery [139] for the block Lanczos, and Coppersmith for the block Wiedemann [51] on $\mathbb{F}_2$, generalized in all finite fields by Kaltofen [112]. These algorithms essentially use the matrix-vector product operations. They need $O(N)$ matrix-vector product operations, and because the matrix is sparse in our context, the number of operation in $\mathbb{F}_\ell$ is in $O(\gamma N)$. The whole computation has therefore a running time in $O(\gamma N^2)$ operations in the field. We know that $\gamma \ll N$, then the iterative methods are well-suited in our context. Moreover, there exist many ways to store a sparse matrix [13, Section 10.1] and the memory complexity cannot be worst than $O(\gamma + N)$. The theses of Jeljeli [97, Chapter 5] and Vialla [181, Chapter 5] describe some formats and their impact on the matrix-vector product implementation.

It then seems obvious that, to solve the linear algebra problems coming from the discrete logarithm context, the best family is the one of iterative methods.

The Wiedemann algorithm

Even if the Lanczos algorithm seems to have better result in computation of medium size, according to the work on Factoring as a Service [180], we only consider the Wiedemann algorithm here, because it is easier to parallelize. For a description of the Lanczos algorithm, we refer to Eberly–Kaltofen [57] and, for the latest improvements, to a chapter book of Thomé [178].

Let consider the minimal polynomial μ_M of M, defined as $\sum_{i=0}^{d} m_i x^i$, where $m_0, m_1, \ldots, m_d$ are in $\mathbb{F}_\ell$. The Cayley–Hamilton theorem states that $d \leq N$. Since the rank of the $N \times N$ matrix M is $N-1$, then 0 is an eigenvalue of M and a root of μ_M; the coefficient m_0 is therefore equal to 0. Assuming μ_M is known, it is possible to deduce a kernel vector $\boldsymbol{w} \neq \mathbf{0}$ of M. Since $\mu_M(M) = 0$, there exists a vector $\boldsymbol{x}$ with coefficients in $\mathbb{F}_\ell$, such that $M(\sum_{i=1}^{d} m_i M^{i-1})\boldsymbol{x} = \mathbf{0}$. The vector $\sum_{i=1}^{d} m_i M^{i-1}\boldsymbol{x}$ is a vector of the kernel of M and is a non-zero vector if $\boldsymbol{x}$ is in the kernel of the matrix $M' = \sum_{i=1}^{d} m_i M^{i-1}$. This matrix has a kernel of dimension at most $N-1$ and if $\boldsymbol{x}$ is randomly chosen in $(\mathbb{F}_\ell)^N$, the vector $\boldsymbol{x}$ is not in the kernel of M' with probability $\ell^{N-1}/\ell^N = 1/\ell$. Given the coefficients m_i and a suitable vector $\boldsymbol{x}$, the number of operations in $\mathbb{F}_\ell$ to compute $\boldsymbol{w} = \sum_{i=1}^{d} m_i M^{i-1}\boldsymbol{x}$ is equal to $O(\gamma N^2)$.

There exist different ways to compute the coefficients $m_1, m_2, \ldots, m_d$. Computing the characteristic polynomial by $\det(xI_N - M)$ and factorizing it to find the minimal polynomial is too costly. Another strategy is to consider that, for all vectors $\boldsymbol{x_0}$ and $\boldsymbol{x_1}$ of size N in $\mathbb{F}_\ell$, $\sum_{i=1}^{d} m_i \boldsymbol{x_0}^{\mathrm{T}} M^{i-1} \boldsymbol{x_1} = 0$. The coefficients m_i are the coefficients of the linear sequence defined by the $\boldsymbol{x_0}^{\mathrm{T}} M^{i-1} \boldsymbol{x_1}$. Let $\boldsymbol{x_0}^{\mathrm{T}}$ and $\boldsymbol{x_1}$ be randomly chosen. Then, with high probability according to Kaltofen [112], we can compute the d coefficients $m_1, \ldots, m_d$ by computing the $2d$ coefficients $\boldsymbol{x_0}^{\mathrm{T}} M^{i-1} \boldsymbol{x_1}$, for i in $[1, 2d]$. The naive method needs $O(N^3)$ operations in $\mathbb{F}_\ell$ but there are some faster methods, such as the Berlekamp–Massey algorithm [131, 28], which needs $O(N^2)$ operations in $\mathbb{F}_\ell$. We can summarize the important steps of the Wiedemann algorithm as follows, where $\boldsymbol{x_0}$, $\boldsymbol{x_1}$ are two vectors of dimension N with randomly chosen coefficients in $\mathbb{F}_\ell$:

Krylov. Compute $\lambda_1, \ldots, \lambda_{2N}$ such that $\lambda_i = \boldsymbol{x_0}^{\mathrm{T}} M^i \boldsymbol{x_1}$.

Linear generator. Compute the linear generator $F(x) = \sum_{i=1}^{N} m_i x^i$ such that $\sum_{i=1}^{N} m_i \lambda_{k+i} = 0$ for k in $[0, N]$.

Evaluation. Compute $\boldsymbol{w} = F(M)\boldsymbol{x_1}$.

Improvements of the Wiedemann algorithm

There exist many improvements of the Wiedemann algorithm and we list three of them.

Block Wiedemann. The first improvement is the block version, that allows to distribute on several processes the computations of two steps of the Wiedemann algorithm: the **Krylov** and **Evaluation** steps. The idea is to consider a block of c vectors instead of just one vector during the Krylov step. This implies an additional cost of $\widetilde{O}(c^2 N)$ during the linear generator computation, but the algorithm can now be parallelized at a high level.

Double-matrix product. The *double-matrix product* [118, 121] was first used in the factorization of seventeen Mersenne numbers with the idea of the *factorization factory* of Coppersmith [48]. Briefly, if M_{raw} is the matrix with the Schirokauer maps, the filter produces $M = M_{\mathrm{raw}} \cdot M_0 \cdot M_1$, with the matrices M_0 and M_1 storing the operations performed by the filtering step. Using double-matrix product, we consider $M_2 = M_{\mathrm{raw}} \cdot M_0$ and instead of computing

the matrix-vector product with M, we consider this product by M_1 and after by M_2. If the sum of the weights of M_1 and M_2 is smaller than the weight of M, this method is advantageous.

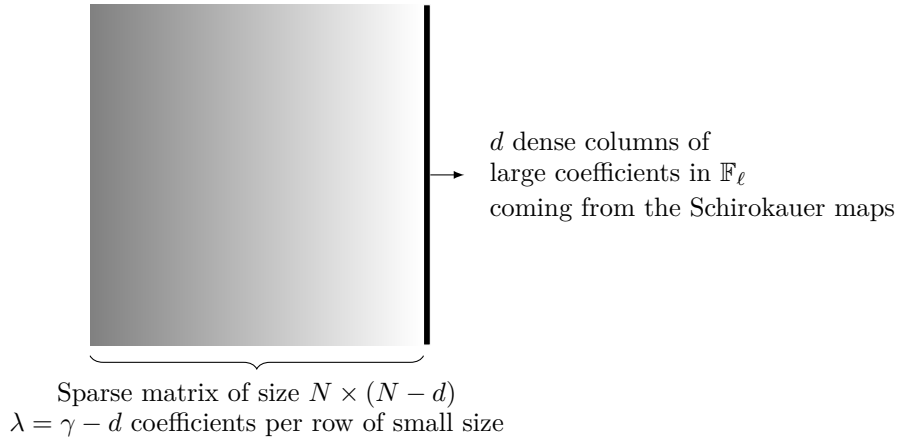


Figure 3.4 – A sparse matrix given after filtering.

Tackle the Schirokauer maps. The input matrix of the block Wiedemann algorithm, represented in Figure 3.4, is made of dense columns of large elements due to the computation of the Schirokauer maps. The impact of these columns on the matrix-vector product is important because each multiplication of integer coefficients close to ℓ needs to be followed by a reduction modulo ℓ. An idea that goes back to Coppersmith [51] is to use these heavy columns as part of the $\boldsymbol{x_1}$ block in the description above. An implementation available in CADO-NFS, used in the computation of the 1024-bit SNFS [65], and an article by Joux and Pierrot [98] allow to perform the computation of the matrix-vector products as if the d dense columns were not in the matrix. If we use the block Wiedemann algorithm with a block size $c \geq d$, it reduces the complexity of the algorithm to $O(\lambda N^2) + \widetilde{O}(c^2 N)$ operations in $\mathbb{F}_\ell$, thus avoiding the expensive contribution dN^2.

3.4 Individual logarithm

Let T be an arbitrarily large element of $\mathbb{F}_p^*$ and g be a generator of $\mathbb{F}_p^*$. In this last step, we are looking for the discrete logarithm k of T in basis g modulo ℓ, assuming that the virtual logarithms of all the factor base elements are known. A careful analysis of the complexity of this phase can be found in the articles of Commeine–Semaev [47] and that of Fried, Gaudry, Heninger and Thomé [65, Appendix A].

The probability to find an integer e such that $\log_g(T^e)$ can be completely written in terms of the precomputed logarithms is very small. Therefore, the individual-logarithm computation cannot be done in one step. The goal is then to build a tree from the target T to all the precomputed logarithms, using logarithms of elements of intermediate sizes, to allow at each step of the descent

to express a logarithm of an element of size N in terms of logarithms of elements of size less than N. This tree is depicted in Figure 3.5. This is what is called the descent step.

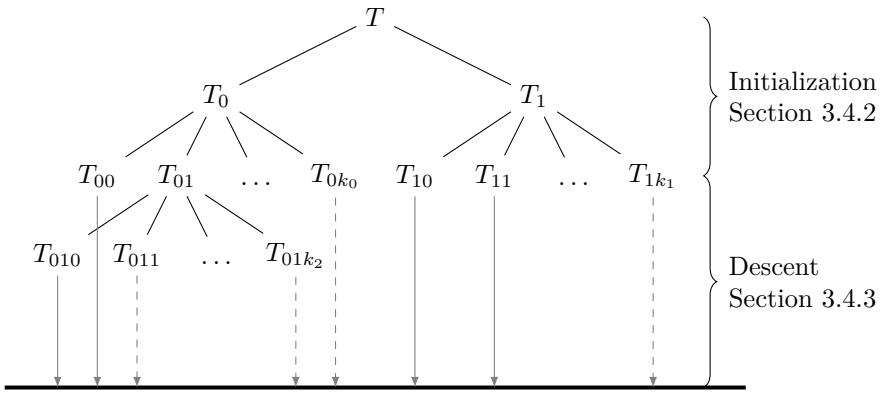


Figure 3.5 – Descent tree.

3.4.1 Lifting elements

Before describing the descent steps, we will describe how to lift an element T in $\mathbb{F}_p^*$ in a number field. We assume that T viewed as an integer is prime.

Trivial case

For simplicity, we assume in the following that T is lifted in a number field K_0 defined by a linear polynomial $f_0 = m_0 + xm_1$. Then, there exists a unique prime ideal of degree one above T, which is $\mathfrak{T} = (T, x + (-m_0/m_1 \bmod T))$. By abuse of notation, the ideal above T in the number field will be denoted by T too in the following.

General case

Let K_1 be defined by a polynomial f_1 of degree d with integer coefficients. Let $\mathfrak{P}$ be the ideal of $\mathcal{O}_1$ defined as $(p, x-m)$. Let z be an element of K_1 such that z modulo $\mathfrak{P}$ is equal to T. There exist z_0 and z_1 in K_1 such that $z \equiv z_0 z_1^{-1} \bmod \mathfrak{P}$, where z_0 is equal to $z_{0,0} + z_{0,1}\theta_1 + \cdots + z_{0,d-1}\theta_1^{d-1}$ and z_1 to $z_{1,0} + z_{1,1}\theta_1 + \cdots + z_{1,d-1}\theta_1^{d-1}$. The $2d$ integers $(z_{0,0}, z_{0,1}, \ldots, z_{0,d-1}, z_{1,0}, z_{1,1}, \ldots, z_{1,d-1})$ verify the equation $z \equiv z_0 z_1^{-1} \bmod \mathfrak{P}$ if they are equal to a linear combination of the row vectors of the matrix L equals to,

$$\left(\begin{array}{ccccc|c} p & & & & & \\ -m & 1 & & & & \\ -m^2 & & 1 & & & \\ \vdots & & & \ddots & & \\ -m^{d-1} & & & & 1 & \\ \hline & & TI_d & & & I_d \end{array}\right).$$

The shortest vector of the lattice generated by the rows of L has infinity norm around $p^{1/(2d)}$ (see Theorem A.1), then the coefficients of z_0 and z_1 must be relatively small. The norms of z_0 and z_1 are about in $(d+1)^{(d-1)/2}d^{d/2}(2d)^{d/2}p^{1/2}$ $\|f_1\|_\infty^{d-1} = O(p^{1/2}\|f_1\|_\infty^{d-1})$. We hope that the factorizations into ideals of z_0 and z_1 involve only ideals of degree 1, which is highly probable, or ideals of small degrees. The largest norm of these ideals must be in $L(2/3)$ to ensure a complexity of the descent in $L(1/3)$, as in the initialization of the descent. The elements z_0 and z_1 must be doubly $L(2/3)$-smooth which happens with probability $L(1/3)$ when $\|f_1\|_\infty$ is in $O(1)$. If we want to lift in a field defined by f_1 whose $\|f_1\|_\infty$ is in $O(p^{1/d})$, the complexity remains in $L(2/3)$, but with a largest constant incompatible with the descent complexity.

3.4.2 Initialization of the descent

From here, we assume that there is a rational side. The goal is to express T^e with elements less than a bound B_{init} which is larger than the factor base bounds. The value of B_{init} must be in $L(2/3)$ to allow an $L(1/3)$ overall complexity.

Improved smoothness test

The first step of the descent is to look for the B_{init}-smoothness of T^e for a random integer e. This step can be improved using the idea of the early-abort strategy of Pomerance [147] proposed by Barbulescu [14, Chapter 4] in the NFS context and summarized in [88]. As with some sieving procedures, the goal is to detect promising B_{init}-smooth numbers. The strategy here is to remove with the ECM the small factors of T^e below a bound B' less than B_{init}. Depending on the size of the remaining unfactored part of T^e, T^e is fully factorized or not. If the full factorization allows to reach the B_{init}-smoothness bound, then we continue the descent, otherwise we pick a new random integer e. This first filter can be itself decomposed in other filters of the same type, decreasing the size of the bound B' in each new filter.

Rational reconstruction and initial splitting

Instead of directly using the early-abort strategy, we can look for the rational reconstruction of $T^e \equiv u/v \bmod p$, with u and v in $O(\sqrt{p})$. We test the B_{init}-smoothness of these two elements and if both are B_{init}-smooth, then we can continue the descent. Otherwise, we pick a new random element e. This can be combined with the early-abort strategy.

A further improvement of this method is to use a sieving procedure, as described in Joux–Lercier [103]. The idea is to write T^e using two different rational reconstructions, that is $T^e \equiv u_0/v_0 \equiv u_1/v_1 \bmod p$, with u_0, u_1, v_0 and v_1 of size about the half of the size of p. Then, for any integers k_0 and k_1,

$T^e \equiv (k_0u_0 + k_1u_1)/(k_0v_0 + k_1v_1) \bmod p$. Finding a pair (k_0, k_1) can be done by sieving as in the relation collection presented in Section 3.2. Indeed, the polynomial $G_0(k_0, k_1) = k_0u_0 + k_1u_1$ can be viewed as the homogenization of the linear polynomial $g_0(x) = u_0 + xu_1$, likewise for $G_1(k_0, k_1) = k_0v_0 + k_1v_1$ with $g_1(x) = v_0 + xv_1$. Then, the search of (k_0, k_1) can be done with g_0 and g_1 playing the role of the polynomials that define the number fields, new factor bases $\mathcal{G}_{g_0}$ and $\mathcal{G}_{g_1}$ and the smoothness bounds both equal to B_{init}. We can also set a special-$\mathfrak{Q}$. If t is the norm of $\mathfrak{T}$, then the rational reconstruction can be done by looking for u_0 of roughly the same size as the size $v_0 + t$, and the same for u_1 with $v_1 + t$. Setting a special-$\mathfrak{Q}$ on the side 1, the objects become of the same size as without the special-$\mathfrak{Q}$. With the special-$\mathfrak{Q}$ method, this allows to not modify e if we do not find doubly smooth relations, and pick a new special-$\mathfrak{Q}$ of norm almost Q. This method is called *tkewness* (for skewness: as skewness is often associated to polynomial selection in NFS, a word was invented, taking the letter coming after "s") in the CADO-NFS software [176].

3.4.3 Descent step

After the initialization of the descent, there probably exist some elements in $\mathbb{F}_p^*$ of unknown logarithm in the factorization of T^e. We need then to descend each of them individually. The methods described in the previous section cannot be applied because the size of the elements becomes too small to use these methods. We can use the special-$\mathfrak{Q}$ method as follow.

Let q in $\mathbb{F}_p^*$ be an element whose logarithm is unknown and such that q is a prime. Let $\mathfrak{Q}$ the prime ideal above q of degree one in the number field K_0 or K_1. We use a similar procedure of the one described in Section 3.2.3, but with a simple modification: the smoothness test on the side 0 must be done without considering the norm q of $\mathfrak{Q}$. Indeed, the smoothness bounds are below the norm of $\mathfrak{Q}$ to allow to continue the descent. In some specific case, when the descent is not possible, we allow that the smoothness bounds are a bit larger than q to involve new ideals that we hope to yield an easier descent path in a next step.

As noted in [65, Appendix A], the descent step can be done faster at some point by mapping integer polynomials of degree $t - 1$ higher than one. The sieving algorithms to perform this will be described in Section 6.4. The major drawback of this descent step using polynomials of degree $t - 1$ is the degree of the prime ideals involved in a relation, which can be less or equal to $t - 1$. The virtual logarithms of these prime ideals are not known but can be found by redoing a small relation collection step, using these unknown ideals as special-$\mathfrak{Q}$ and keeping a relation involving the prime ideals of degree one whose virtual logarithms are already known.

3.4.4 Individual logarithm procedure

We put together the two steps described above in a description that is close to an algorithm:

Reduction. Pick some random power e, compute T^e.

Initialization. Compute two rational reconstructions of $T^e = u_0/v_0 = u_1/v_1$ and find by sieving (k_0, k_1) such that $k_0u_0 + k_1u_1$ and $k_0v_0 + k_1v_1$

are B_{init}-smooth.

Descent. For all the prime factors q of $k_0u_0 + k_1u_1$ and $k_0v_0 + k_1v_1$ for which $\log_g q$ is unknown

1. Build an empty list L.
2. Compute the ideal $\mathfrak{Q}$ above q in one number field, say K_0, and add $\mathfrak{Q}$ to L.
3. While L is not empty
   - Perform a special-$\mathfrak{Q}$ descent to find a relation (a, b) and add the ideal of unknown virtual logarithm to L.

Reconstruction. Knowing all the intermediate logarithms, compute $k = \log_g T$.

At each step of the descent by special-$\mathfrak{Q}$, we need to adjust the parameters of the sieve and, even if the complexity analysis gives a bound on the smoothness bound depending on the size of the special-$\mathfrak{Q}$ that are processed, this bound cannot be used as it is in the practical descent. Furthermore, when a special-$\mathfrak{Q}$ cannot be descended given a smoothness bound, the choice between increasing the sieving region, or increasing the smoothness bound or both or something else, as the extreme choice of wasting the relation involving this special-$\mathfrak{Q}$ to look for a new one is hard, even if the overall complexity remains in $L(1/3)$.

3.5 A small example

In order to give an example of some steps of the NFS algorithm, we propose a very simple implementation of the different steps of NFS to compute discrete logarithms in $\mathbb{F}_p^*$, where p is a prime and ℓ the largest factor of $p-1$, using the Sage software [177]. This implementation can be found in Appendix B.

We use the base-m polynomial selection where m equals $\lfloor p^{1/(d+1)} \rfloor$, where d is the degree of $f_1 = f_{1,0} + f_{1,1}x + \ldots + f_{1,d}x^d$. The ideals whose norm divides $f_{1,d}$ and $f_{1,d-1}$ and those whose norm divides the discriminant of the polynomial are difficult to process: during the relation collection, if a factor of the resultant between a polynomial $a_0 + a_1x$ and f_1 involves a prime equals to these avoided norms, we forget this relation. Of course, in a real implementation, this would not be the case.

The relation collection is done by applying line sieving inside the special-$\mathfrak{Q}$ method, the special-$\mathfrak{Q}$ is set on side 1. Let $\{\boldsymbol{b_0}, \boldsymbol{b_1}\}$ be the basis of the special-$\mathfrak{Q}$-lattice used to extract all the elements $a_0 + a_1\theta_1$ in the special-$\mathfrak{Q}$. The coefficients a_0 and a_1 of such an element are a linear combination of $\boldsymbol{b_0}$ and $\boldsymbol{b_1}$, say $(a_0, a_1) = c_0\boldsymbol{b_0} + c_1\boldsymbol{b_1}$, where c_0 and c_1 are integers. For each ideal $\mathfrak{R} = (r, x + \rho)$ to be sieved, we need to consider a basis of the $\mathfrak{R}$-lattice inside the special-$\mathfrak{Q}$-lattice. The elements $a_0 + a_1\theta_1$ in $\mathfrak{R}$ verify the equation $a_0 + a_1x \equiv 0 \bmod (r, x + \rho)$, which can be rewritten as $c_0\boldsymbol{b_0}[0] + c_1\boldsymbol{b_1}[0] - \rho(c_0\boldsymbol{b_0}[1] + c_1\boldsymbol{b_1}[1]) \equiv 0 \bmod r$. If $\boldsymbol{b_0}[0] - \rho\boldsymbol{b_1}[0]$ is not zero modulo r, then the $\mathfrak{R}$-lattice extracted from the special-$\mathfrak{Q}$-lattice has basis $\{(r, 0), (\rho_0, 1)\}$, where $\rho_0 \equiv (\rho\boldsymbol{b_1}[1] - \boldsymbol{b_1}[0])/(\boldsymbol{b_0}[0] - \rho\boldsymbol{b_0}[1]) \bmod r$. We perform the enumeration of the special-$\mathfrak{Q}$-lattice for many special-$\mathfrak{Q}$.

Once the number of relations is larger than the number of ideals in the factor basis, we compute an almost complete factorization in ideals of the relations. Indeed, at this step, the factorization of the resultants between $a = a_0 + a_1 x$ and the polynomials f_i is known, but the factorization of $a_0 + a_1\theta_i$ is not known. This factorization is computed by taking for all prime factor p of the resultant, the corresponding ideals $(p, x+\rho)$, where $x+\rho$ is a degree one factor of a modulo p. In this factorization, we forget some of the ideals whose norms divide the leading coefficients of the polynomial f_i, but these factors are the same for all relations and can be replaced by a column of 1 in the matrix. The matrix is built by packing the left columns of the matrix with the ideals on the side 0, the right columns with the ideals on side 1, the column of 1 and the Schirokauer maps on side 1. We then can compute the right kernel modulo ℓ of the matrix and find the virtual logarithms of almost all the ideals, the virtual logarithm of the column of 1 and the one of the Schirokauer maps.

To verify if the computation is correct, we map in the rational side, the side 0, all the prime q less than the smoothness bound and test if its virtual logarithm v_q verify $q^{v_r(p-1)/\ell} \equiv r^{v_q(p-1)/\ell} \bmod p$, for an ideal of norm r of virtual logarithm v_r, if . We note that the virtual logarithms are given in an arbitrarily basis and to have the virtual logarithms in a chosen basis, it suffices to divides them by the virtual logarithm of the chosen element.

In our example, we take $p = 2\ell + 1$, where $\ell = 3141592653589793238462773$ and therefore, p is of size 83 bits. Such a p requires to perform a polynomial selection with a polynomial f_1 of degree 3, if we use the base-m method. We choose the same smoothness bounds on both side, which is equal to 2^{12}, the same sieving bound, which is equal to 2^{10}, and the same threshold, which is equal to 2^{36}. All the special-$\mathfrak{Q}$, set on side 1, have norm between the sieving bound and the smoothness bound. The sieving region for all the special-$\mathfrak{Q}$ is equal to $[-2^7, 2^7[\times[0, 2^7[$. This corresponds to a sieving region for the (a_0, a_1) pairs of about $[-2^{20}, 2^{20}[\times[0, 2^{20}[$. We collect 3325 raw relations and even if we use the special-$\mathfrak{Q}$ method, we do not have duplicate relations. All these relations allow to find the virtual logarithm modulo ℓ of 1187 ideals, that is, all except one ideals in the factor basis.

3.6 The special and multiple NFS algorithms

3.6.1 The special NFS algorithm

The special NFS algorithm was used many times in the context of integer factorization, in particular for the Cunningham Project whose aim is to give the factorization of the numbers $b^n \pm 1$, where b is in $\{2, 3, 5, 6, 7, 10, 11, 12\}$, and especially the Mersenne numbers of the form $2^n - 1$. The adaptation in the context of the discrete logarithm computation is due to Gordon [76]. It exploits the special form of the prime p defining $\mathbb{F}_p^*$.

In order to have an efficient arithmetic modulo the prime p, one can choose p as $2^n - c$, where c is a small integer. However, by choosing $f_0(x) = x - 2^{n/d}$ and $f_1(x) = x^d - c$, where d is a divisor of n, we get a polynomial f_0 of degree 1 and of infinity norm in $O(p^{1/d})$, and f_1 of degree d and of infinity norm in $O(1)$, instead of $O(p^{1/d})$ with the base-m method. The complexity analysis, if c and d match the requirements, shows that the constant term of the NFS complexity

decreases, from $(64/9)^{1/3}$ to $(32/9)^{1/3}$. Such "efficient" Mersenne-like primes must therefore be avoided for discrete logarithm cryptography.

An application of special NFS is the building of trapdoored primes. To construct such a prime, given a certain size S, the polynomials f_0 and f_1 are first chosen to reach the following requirement, given by the complexity analysis: the size of the resultant between f_0 and f_1 must be very close to S, f_1 is chosen to have degree d and coefficients in $O(1)$ with a good α quantity, and f_0 must be a linear polynomial $f_0 = f_{0,0} + xf_{0,1}$ with $f_{0,0} \approx f_{0,1}$ in $O(2^{(\log_2 p)/d})$. The degree d is equal to $(3 \log p / \log\log p)^{1/3}$, as in the base-$m$ method. Details can be found in [76, Section 5] and in [65, Algorithm 1].

3.6.2 The multiple NFS algorithm

The multiple number field sieve algorithm is a variant of NFS which was first proposed by Coppersmith in [48] in the factorization context, adapted to compute discrete logarithms in large characteristic by Matyukhin in [132] and refined by Commeine and Semaev [47]. We use here the formalism used in the Barbulescu–Pierrot article [24]. It briefly consists in proposing many ways instead of one to find relations.

If we analyze the two polynomial selections described in Section 3.1 and summarized in Table 3.1, we note that the two sides are asymmetric, that is the polynomials f_0 and f_1 have a different degree and infinity norm. Let us consider the Joux–Lercier polynomial selection. We can consider different polynomials f_1 during the lattice reduction. Indeed, in the original NFS algorithm, we look for a polynomial f_1 with the smallest possible coefficients and then look for the smallest vector of the lattice built during the polynomial selection, but all the vectors given by a lattice reduction can be more or less chosen equivalently. Let f_1 be the classical polynomial chosen during this polynomial selection and f_2 by the polynomial whose coefficients are given by the second smallest vector of the reduced basis. Using the polynomial pair (f_0, f_2) instead of (f_0, f_1) is valid for the original NFS algorithm, as for $(f_0, \lambda_0 f_1 + \lambda_1 f_2)$, with λ_0 and λ_1 two integers. We can define $V-1$ polynomials $f_1, f_2 \ldots, f_{V-1}$ as small linear combinations of f_1 and f_2, for some integer V. The degree of all these polynomials is the same. The smoothness probability of the norms in the number fields $K_i = \mathbb{Q}[x]/f_i(x)$ for i in $[1, V[$ is therefore roughly the same, as the number of relations involving ideals of $\mathcal{O}_0$ and $\mathcal{O}_i$. The side 0 plays an important role during the relation collection of MNFS, and each relation must involve ideals of $\mathcal{O}_0$. The MNFS algorithm implies a modified commutative diagram, as represented in Figure 3.6.

One can try to find a relation involving ideals of $\mathcal{O}_i$ and $\mathcal{O}_j$ for $j > i \geq 1$. We recall that all the polynomials f_i with $i \geq 1$ have the same degree, say d, and the same infinity norm of magnitude $p^{1/(d+1)}$, compared to the degree $d+1$ of f_0 and small coefficients. Let E be the value of the bound of the coefficients of the polynomial a mapped in K_i. The complexity analysis gives that $E = L_p(1/3, (8/9)^{1/3})$. An upper bound [34] of the norms in K_0 is $2^{(d+1)/2}(d+2)^{1/2}E^{d+1}$ and an upper bound in K_i is $2^{(d)/2}(d+1)^{1/2}E^d p^{1/(d+1)}$, with $i \geq 1$. The norms in K_i are therefore much larger than in K_0, with $i \geq 1$. Then, even if there maybe exist relations involving ideals of $\mathcal{O}_i$ and $\mathcal{O}_j$ for $j > i \geq 1$, the search of such a relation is substantially more time-consuming than only in $\mathcal{O}_0$ and $\mathcal{O}_i$. We then build an asymmetric commutative diagram, as shown in Figure 3.6 with the bold arrows.

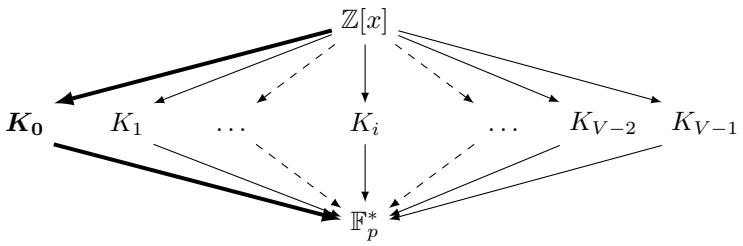


Figure 3.6 – The multiple NFS diagram for $\mathbb{F}_p$.

The linear algebra stays unchanged, except that we need to consider the Schirokauer maps in all the number fields. The individual logarithm computation can be done theoretically using only two sides. The complexity analysis shows that the constant term of the NFS complexity decreases, from $(64/9)^{1/3} \approx 1.93$ to $((92 + 26\sqrt{13})/27)^{1/3} \approx 1.91$.

Part II

Discrete logarithm in medium characteristic

Chapter 4

The high-degree variant of the number field sieve algorithm

Adleman–DeMarrais introduced in [4] the first subexponential algorithm to compute discrete logarithms in medium characteristic fields and has a running time in $L(1/2)$. In 2006, Joux, Lercier, Smart and Vercauteren [106] described the high-degree variant of the number field sieve algorithm that achieves a running time in $L_{p^n}(1/3, 2.43)$. This variant is called high-degree because the polynomials that give relations can have a degree larger than 1. Based on this variant, the complexity of the algorithm was further improved to the complexity $L_{p^n}(1/3, 2.16)$ using the variant of Pierrot [143], a combination of the multiple number field sieve algorithm with a polynomial selection introduced by Barbulescu, Gaudry, Guillevic and Morain [20].

The NFS-HD algorithm is an index-calculus algorithm that shares many properties with NFS for prime field: we therefore keep the same pieces of notation introduced in Chapter 3. The fields targeted by NFS-HD are of the form $\mathbb{F}_{p^n}$, where $n > 1$ is a small integer and p is a medium prime [106], that is $L_{p^n}(1/3, \cdot) < p < L_{p^n}(2/3, \cdot)$.

Let f_0 and f_1 be two irreducible polynomials with integer coefficients, and sharing a common irreducible factor φ of degree n modulo p. Keeping the same notation for the number field K_0 defined by f_0 and K_1 defined by f_1, we form the commutative diagram of Figure 4.1. We force that the degree of f_0 is less than or equal to the one of f_1. We reserve the notation a for the integer polynomial that will be sent through the diagram. Its degree is set to $t - 1$ so that we perform the relation collection in dimension t, and in the classical NFS algorithm, we have $t = 2$.

We limit a sieving region, or equivalently a search space, by setting the bounds on the coefficients, in this case the coefficients of a.

Definition 4.1. Let $t \geq 1$. A t-sieving region $\mathcal{S}$ is a t-dimensional box of the form $[I_0^m, I_0^M[\times[I_1^m, I_1^M[\times\cdots\times[I_{t-1}^m, I_{t-1}^M[$, where all the $[I_i^m, I_i^M[$ are integer

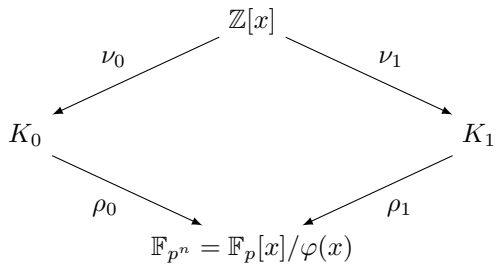


Figure 4.1 – The NFS-HD diagram to compute discrete logarithms in $\mathbb{F}_{p^n}$.

intervals. The t-sieving region, also called sieving region when t is implicit, must contain the element $(0, 0, \ldots, 0)$.

By abuse of notation, we say that a polynomial $a = a_0 + a_1 x + \cdots + a_{t-1}x^{t-1}$ is in a sieving region $\mathcal{S}$ if $(a_0, a_1, \ldots, a_{t-1}) = \boldsymbol{a}$ is in $\mathcal{S}$.

The relation collection is essentially conducted like over a prime field, the major difference is the modification of the searching space $\mathcal{S}$ and then, the sieving algorithm used to improve the running time has to be modified.

The linear algebra is performed exactly as for a prime field. The number of Schirokauer maps can be larger, but thanks to the algorithm that uses the Schirokauer maps as input vectors, the problem of dealing with at most $2n$ Schirokauer maps can be essentially avoided, see Section 3.3.2.

Finally, the individual logarithm step can be conducted as over a prime field, that is using a special-$\mathfrak{Q}$ descent for example. Guillevic proposes in [86, 88] a way to improve the initial splitting step of the computation with an algorithm that allows to reduce the coefficients of the targeted elements.

4.1 Polynomial selections

In this section, we will present six different algorithms that produce two valid polynomials (f_0, f_1). In order to choose the best pair, we can use the following strategies: we first estimate the size of the norms given the degree $t-1$ of the polynomial a, the extension n and a size for the characteristic p for all the polynomial selections. We select the two or three polynomial selections that give the smallest estimated norms and distinguish the best pair using the α and the Murphy-E quantity, as in Section 3.1. The quality criteria need however to be adapted to higher dimension. This is not a simple matter, and we concentrate on dimension 3.

4.1.1 Quality criteria in 3 dimensions

Size properties

The first estimate we use is the size of the norms on both sides. An upper bound of the product of two norms is $(\|f_0\|_\infty \|f_1\|_\infty)^{t-1} E^{2(\deg f_0 + \deg f_1)/t}$, by following [20, Section 4.1], where E is the bound on a 2-sieving region: typical value of E can be found in [20, Table 2]. Another strategy to estimate the product of the two norms is to build some typical polynomials f_0 and f_1 and

sample some polynomials a in an approximate sieving region bounded by $E^{1/t}$ and computing the average of the product of the norms. Whatever the way to compute the upper bound, we select the two or three best polynomial selections, according to different value of t, that reach the smallest product of the norms.

Local properties

In Chapter 3, we have introduced the α quantity as a way to correct the estimation of the smoothness probability of a norm to be smooth. We recall that, if the norms have a size N (expressed in base e) on average in a number field defined by f_0 (respectively f_1), the smoothness probability of these norms is estimated by the smoothness probability of integers of size $N+\alpha(f_0)$ (respectively $N+\alpha(f_1)$). The definition of the α quantity can be found in Definition 3.1 and we just recall that $\alpha(f_0) = \sum_{\ell \text{ prime}} \alpha_\ell(f_0)$, where for all prime ℓ,

$$\begin{aligned}\alpha_\ell(f) = \ln(\ell)\big[\, &\mathbb{A}(\mathrm{val}_\ell(n), n \in \mathbb{Z}) \\ &- \mathbb{A}(\mathrm{val}_\ell(\mathrm{Res}_x(f(x), a(x)), \text{ where } a \in \mathbb{Z}[x], \deg a = t-1, a \text{ irreducible}))\big],\end{aligned}$$

where val_ℓ is the ℓ-adic valuation and $\mathbb{A}(\cdot)$ is the average value, defined by taking the limit of the average value of the quantity for increasingly large finite subsets of the whole set considered. To disambiguate this choice, we take these subsets as intersections of $\mathcal{S}$ with centered balls of increasing radius. Another potential issue with this definition is the convergence of the series defining $\alpha(f)$. We leave it as a conjecture: since adapting the proof of [23] goes beyond the scope of this thesis. When $t = 3$, and if ℓ does not divide the leading coefficient of f or its discriminant, then we have the equality

$$\alpha_\ell(f) = \frac{\ln(\ell)}{\ell - 1}\left(1 - n_1\frac{\ell(\ell+1)}{\ell^2+\ell+1} - 2n_2\frac{\ell^2}{(\ell+1)(\ell^2+\ell+1)}\right),$$

where n_1 and n_2 are the number of linear (respectively, degree-2) irreducible factors of f modulo ℓ. The proof of this computation is taken from our article [72] and can be found in Appendix C.

Global property

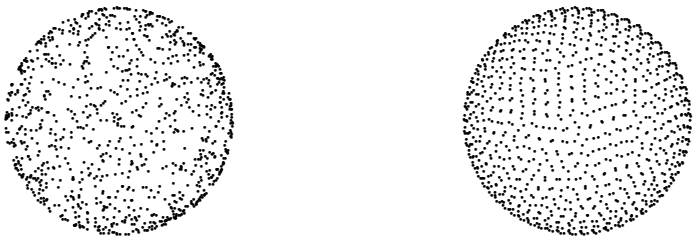
As for the two-dimensional case, we can define a similar formula to Equation 3.3 to estimate the number of relations in the sieving region $\mathcal{S}$ by

$$\sum_{\substack{a \in \mathcal{S} \\ a \text{ irreducible}}} \rho\left(\frac{\ln|\operatorname{Res}(f_0, a)| + \alpha(f_0)}{\ln B_0}\right)\rho\left(\frac{\ln|\operatorname{Res}(f_1, a)| + \alpha(f_1)}{\ln B_1}\right). \tag{4.1}$$

Because the relation collection is often performed with the special-$\mathfrak{Q}$ method (see below for a generalization of the method presented in Section 3.2.3). We can take it into account by considering that we put the special-$\mathfrak{Q}$s, for example on side 0, and that the size of a typical special-$\mathfrak{Q}$ is $\ln Q$. Equation (4.1) becomes

$$\sum_{\substack{a \in \mathcal{S} \\ a \text{ irreducible}}} \rho\left(\frac{\ln|\operatorname{Res}(f_0, a)| + \alpha(f_0) - \ln Q}{\ln B_0}\right)\rho\left(\frac{\ln|\operatorname{Res}(f_1, a)| + \alpha(f_1)}{\ln B_1}\right). \tag{4.2}$$

As in the two-dimensional case, evaluating exactly this formula is equivalent to performing the full relation collection, but for efficiency reasons, we would like to avoid to do that when selecting the best polynomials. In the two-dimensional case, we walk in the boundary of the searching space: we follow this idea in the three-dimensional case. The rationale is that when we multiply a polynomial a by a scalar r, the resultant between a and f is multiplied by $r^{\deg f}$, and therefore the sizes of the norms on a line through the origin are well controlled once one value on it is known. With the special-$\mathfrak{Q}$ method, since we are dealing with many ideals $\mathfrak{Q}$, each of them favoring some direction, viewed globally the general shape of the sieving region will be a sphere, or an ellipsoid if there is some skewness on f_0 and f_1. Hence, we will use this approximation for computing a Murphy E value and compare the polynomial pairs: we pick a sphere or an ellipsoid corresponding to our sieving region, and perform a Monte Carlo evaluation of the integral on its surface. In practice, we found it convenient to use a Fibonacci sphere [75] as evaluation points, see Figure 4.2: indeed, the points on a Fibonacci sphere are regularly spaced on the sphere.



Random points. Fibonacci sphere.

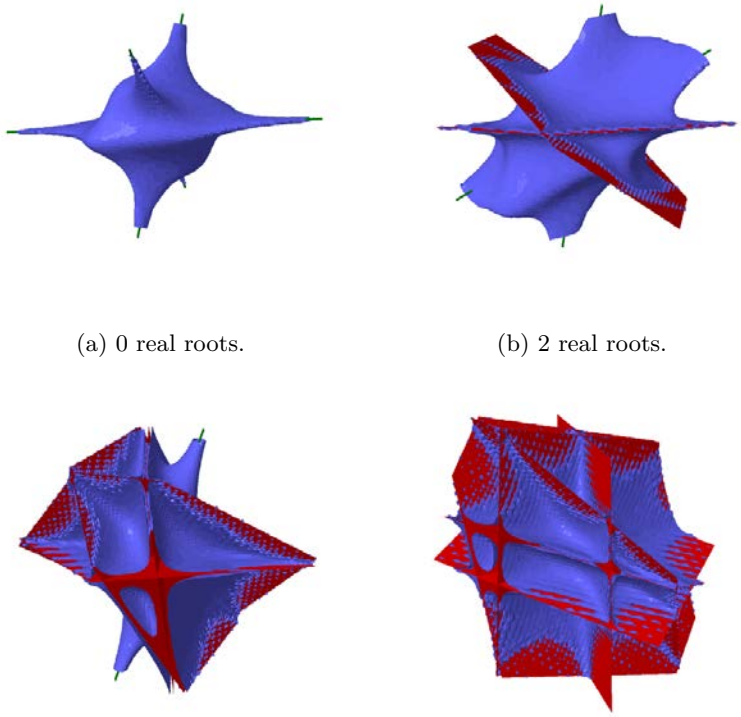
Figure 4.2 – Points on a sphere.

As in NFS for prime fields (see Section 3.1.1), the number of real roots of a polynomial f tends to increase the Murphy E quantity. A real root in f will correspond to a plane in the direction of which the isonorm sphere is deformed with a bump. In the three-dimensional case, we also need to consider the complex roots. The isonorm sphere is therefore also stretched in the direction of the lines corresponding to polynomials close to irreducible factors of degree 2.

Figure 4.3 illustrates how the sphere is modified in the directions corresponding to real and complex roots of f . This clearly shows that the spherical approximation is not accurate enough and justifies the use of the more precise Murphy E estimate.

Explicit Galois actions

The possibility of an explicit, easy to compute, Galois action, having the same expression for f_0 and f_1 is an additional criteria to take into account. For a polynomial f_i, and a homography $\sigma(x) = n(x)/d(x)$, we define $f_i^\sigma(x) = f_i(\sigma(x))d(x)^{\deg f_i}$. Then, σ is said to be an explicit Galois action for f_i if f_i^σ is proportional to f_i. In that case, σ is an automorphism of the number field K_i. In this field, $a(\sigma(x))$ is a conjugate of $a(x)$: they have the same norm. In our context we need to work with polynomials, so we consider $a^\sigma = a(\sigma(x))d(x)^{\deg a}$.



(a) 0 real roots. (b) 2 real roots.

(c) 4 real roots. (d) 6 real roots.

Figure 4.3 – Isonorms for polynomials of degree 6 with real roots. The red planes correspond to real roots and the green lines to complex roots.

The norm is therefore multiplied by the norm of $d(x)^{\deg a}$, which is typically a small, smooth number.

Let consider that σ is an explicit Galois action of order k for f_0. It is possible to build a t-searching space $\mathcal{S}'$ which is a $1/k$-portion of $\mathcal{S}$ and such that, for all a in $\mathcal{S}'$, the polynomials $a^{(\sigma^i)}$ are in $\mathcal{S}$, where i is in $[0, k[$. Therefore, the relation collection on side 0 can be done by enumerating only the polynomials a in $\mathcal{S}'$, and then recover all the polynomials a in $\mathcal{S}$ that have by construction a smooth norms on this side. On side 1, the relation collection is performed classically. If the time to perform the relation collection on the whole of $\mathcal{S}$ on side 0 is T_0 and T_1 on side 1, the acceleration factor is equal to $(T_0 + T_1)/(T_0/k + T_1)$, which tends to 2 when k is large.

But if σ is an explicit Galois action of order k for both f_0 and f_1, and if a is a polynomial that has a smooth norm on both sides and hence yields a relation, then a^σ also yields a relation: we can deduce the $k-1$ additional relations for free, by letting σ act on a, allowing us to have an acceleration factor equals to k. In a special-$\mathfrak{Q}$ context, it is simple to organize the computation in order to save a factor k in the relation collection phase. Indeed, the special-$\mathfrak{Q}$s can be organized in orbits of k conjugate ideals, and if a polynomial a yields a

principal ideal divisible by $\mathfrak{Q}$, then a^σ yields a principal ideal divisible by $\mathfrak{Q}^\sigma$. It is therefore enough to sieve only one of the special-$\mathfrak{Q}$s per orbit, and to derive relations for the other special-$\mathfrak{Q}$s under conjugation by σ.

Remark 4.1. On prime fields, due to the polynomial selections used in that case, it is not possible to have the same Galois action on both sides, but it is possible to enforce a Galois action in one side. This is what was done for the computation in a prime field of size 431 bits by Joux and Lercier [104].

4.1.2 Generation of polynomial pairs

There exists many polynomial selections for NFS-HD to define the polynomial pair (f_0, f_1). Each of them allows to reach a different shape for the polynomial pair, sometimes resulting in a different complexity. The names of the polynomial selections are the one used in the article of Barbulescu, Gaudry, Guillevic and Morain [20].

$\mathbf{JLSV_0}$

This first polynomial selection is one of the polynomial selection described in [106, Section 2.1] to reach the $L(1/3)$ complexity. The polynomial f_0 is chosen to have degree n, small coefficients and to be irreducible in $\mathbb{F}_p$. The polynomial f_1 is equal to the sum or the difference of p to f_0. Such polynomials were used in [106] to compute discrete logarithms over $\mathbb{F}_{p^3}^*$, where p^3 was 394-bit long, by Zajac in $\mathbb{F}_{p^6}^*$, where p^6 was 240-bit long [185], and by Hayasaka, Aoki, Kobayashi and Takagi in $\mathbb{F}_{p^{12}}^*$, where p^{12} was 203-bit long [93]. This polynomial selection produces unbalanced polynomials in term of infinity norm and it is not possible to enforce a Galois action on both sides.

$\mathbf{JLSV_1}$

Original description. This polynomial selection was first described in [106, Section 2.3]. It was proposed to balance the infinity norm of the two polynomials. To build f_0, we choose two polynomials g_0 and g_1 of degree n with small coefficients and c_0 an integer close to $\sqrt{p}$. If $g_0 + c_0 g_1$ is irreducible over $\mathbb{F}_p$, then we set f_0 to $g_0 + c_0 g_1$. Thanks to the extended Euclidean algorithm, we can compute c_1 and c_2 of size about $\log \sqrt{p}$ such that $c_0 \equiv c_1/c_2 \bmod p$. We define f_1 as $c_2 g_0 + c_1 g_1$.

Taking into account the special-$\mathfrak{Q}$s. Let consider that we set a special-$\mathfrak{Q}$ (see Section 3.2.3 or Section 4.2.2) on side 1. Because we know that the norms on this side are divisible by Q, the norm of a typical special-$\mathfrak{Q}$, the resulting norm on this side is about $\mathrm{Res}(a, f_1)/Q$. We remarked in Section 3.1 that, given a list of polynomial pairs that yield a similar value for the sum of the sizes of the norms, it is better to choose the pairs for which the norms have sizes close to each other. Using this remark, we look for polynomials such that $\mathrm{Res}(a, f_1)/Q \approx \mathrm{Res}(a, f_0)$. Using a crude upper bound, the ratio between the norms on both sides is $\mathrm{Res}(a, f_1)/\mathrm{Res}(a, f_0) \approx \|f_1\|_\infty^{t-1}/\|f_0\|_\infty^{t-1} \approx Q$. We therefore need to select polynomials with unbalanced infinity norms, but we keep the gap between the two infinity norms under control. We can perform

such a task using an unbalanced extended Euclidean algorithm, as shown in the following.

Let $0 \leq \varepsilon < 1/2$ be the variable that helps us to control the balancing between the two norms. The infinity norm of f_0 must be smaller than the one of f_1, we therefore try to have the infinity norm of f_1 close to $p^{1/2-\varepsilon}$. Using a similar construction of the one of JLSV_1, it suffices to chose c_0 close to $p^{1/2-\varepsilon}$. To have the product of the two norms close to the one of the original JLSV_1, we need to have the infinity norm of f_1 close to $p^{1/2+\varepsilon}$ (reaching a smaller bound will decrease the complexity of NFS-HD and seems impossible with the method we use). It can be done by using again the extended Euclidean algorithm to compute $c_0 \equiv c_1/c_2 \bmod p$, with c_1 close to $p^{1/2-\varepsilon}$ and c_2 close to $p^{1/2+\varepsilon}$. We define again f_1 as $c_2 g_0 + c_1 g_1$, which have an infinity norm close to $p^{1/2+\varepsilon}$. The ratio between the two norms is therefore equal to $p^{2\varepsilon(t-1)}$, which must be as close as possible to Q, and equality is reached by fixing $\varepsilon = \log_p(Q)/(2(t-1))$.

Exploiting a Galois action. The construction of the polynomials allows the possibility to enforce the same Galois automorphism on both polynomials. Barbulescu, Gaudry, Guillevic and Morain reproduce a list of particular forms for g_0 and g_1 for n in $2, 3, 4, 6$ to use nice automorphisms [19, Table 4]. The combination of the enforcement of a Galois action of order 6 and the unbalanced form to take into account the special-$\mathfrak{Q}$s was used in our article with Gaudry and Videau [72] to perform the relation collection in three different $\mathbb{F}_{p^6}$, where p^6 were 240-bit, 300-bit and 389-bit long. The original version combined with a Galois automorphism of order 3 was also used in the computation over $\mathbb{F}_{p^3}$, where p^3 was 508-bit long by Guillevic and Morain [90], and over $\mathbb{F}_{p^4}$ with a 392-bit long characteristic by Barbulescu, Gaudry, Guillevic and Morain [87].

JLSV_2

Original description. This polynomial selection is not particularly well designed for the medium characteristic, but we recall it for completeness. It was originally described in [106, Section 3.2] for finite fields $\mathbb{F}_{p^n}$ of large characteristic. It is now outperformed by the generalized Joux–Lercier, described in the next paragraph. Let g_0 be a polynomial of degree n with small coefficients and irreducible over $\mathbb{F}_p$ and assume for the moment we set f_0 equal to this polynomial (we will show in the following that this is not the best choice)). We consider a polynomial f_1 of degree $d_2 \geq n$. We must ensure that, f_0 and f_1 share a same irreducible factor φ of degree n modulo p. Without loss of generality, the polynomial φ is monic and we can therefore write that $f_1 \equiv k\varphi \bmod p$, where k is a polynomial of degree $d_2 - n$. The coefficients of φ are labeled as φ_i, where i is in $[0, n]$ and $\varphi_n = 1$. We can build valid polynomials f_1 as linear combinations of polynomials px^j, where j is in $[0, d_2]$, and of the polynomials $\varphi(x)x^m$, where m is in $[0, d_2 - n]$. We can remark that the polynomial px^q, where q is in $[n, d_2]$ can be rewritten as a linear combination of $\varphi(x)x^m$ and px^j, where m is in $[0, d_2 - n]$ and j is in $[0, n[$. To find a polynomial f_1 with the smallest possible coefficients, it suffices to find a short vector of the lattice for which a basis is given by the rows of the following $(d_2 + 1) \times (d_2 + 1)$ matrix

$$M_{\varphi,d_2,n} = \begin{pmatrix} p & & & & & & & \\ & p & & & & & & \\ & & \ddots & & & & & \\ & & & p & & & & \\ \varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & \varphi_n & & & \\ & \varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & \varphi_n & & \\ & & \ddots & \ddots & & \ddots & \ddots & \\ & & & \varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & \varphi_n \end{pmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right\} n \\ \\ \\ \\ \left.\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right\} d_2 + 1 - n \end{matrix} \tag{4.3}$$

Remark 4.2. The notation $M_{\varphi,d_2,n}$ will be used again for the GJL and Sarkar–Singh polynomial selections. This corresponds to the matrix given in [20, Algorithm 2], because the authors force f_0 to be monic and the coefficients of f_0 are smaller than p, therefore $\varphi = f_0$.

The volume of the lattice described previously is equal to p^n and we know that the infinity norm of the shortest lattice vector is not much larger than $p^{n/(d_2+1)}$ following Theorem A.1. The infinity norm of f_1 is then in $O(p^{n/(d_2+1)})$. But, because f_0 has small coefficients and has degree n, the shortest vector has the coefficients of f_0 and the second shortest vector has most probably large coefficients.

Because of this drawbacks, the authors of [106] propose to use f_0 with larger coefficients: instead of setting f_0 to g_0, they build f_0 as $g_0(x + W)$, where W is a constant to be defined latter. The largest coefficient of f_0 is close to $\mathrm{lc}(g_0)W^n \approx W^n$. If W is sufficiently large, the shortest vector of the previous lattice will probably not be formed by the coefficients of f_0 and can therefore define the coefficients of f_1. To balance the infinity norms of f_0 and f_1, we need to have $W^n \approx p^{n/(d_2+1)}$, that is $W \approx p^{1/(d_2+1)}$. To ensure that the shortest vector is not made of the coefficients of f_0, the coefficient W must be larger than $p^{1/(d_2+1)}$. The output coefficients describe not necessarily an irreducible polynomial, it is therefore needed to test the irreducibility of the possible polynomial f_1.

Remarks. We can observe that the size of the coefficients of f_0 are in geometric progression of ratio W: indeed the leading coefficient is in $O(1)$, the term in x^{d_2-1} is in $O(W)$ and so on to the constant term in $O(W^n)$. We say that f_0 has a skewness W. But, with the classical basis reduction, the coefficients of f_1 are almost all of the same size, that is a skewness of 1. During the relation collection, it is better to have the same skewness for both polynomials, or, close to it. The optimal skewness for f_1 is obtained when $d_2 = n$ because the volume of the lattice $p^n = W^{n(d_2+1)}$ is equal to the product of the expected values of the coefficients of f_1, that is $(W^{d_2}, W^{d_2-1}, \ldots, 1)$. If d_2 is larger than n, the expected values for f_1 are in $(W^{bd_2}, W^{b(d_2-1)}, \ldots, 1)$, where $b = 2n/d_2$. To obtain such a polynomial f_1, it is possible to perform a skew basis reduction with weights $(W^{-bd_2}, W^{-b(d_2-1)}, \ldots, 1)$ (or equivalently $(1, W^b, \ldots, W^{bd_2})$), as described in Section A.1.

The choice of a polynomial g_0 with small coefficients seems not necessary to define the polynomial f_0: it seems sufficient to build a polynomial f_0 with

coefficients in $O(W^n)$ to reach the same bounds as the original description. It is however maybe harder to enforce a Galois action in such a polynomial than in the original construction. With such a construction, the skew basis reduction is not necessary.

Generalized Joux–Lercier

The generalized Joux–Lercier (GJL) was introduced by Barbulescu, Gaudry, Guillevic and Morain [20, Section 3.2] as an extension of the Joux–Lercier polynomial selection for prime fields, described in Section 3.1: with the same reasoning, the gap between the degrees of f_0 and f_1 is 1.

First, we select an irreducible polynomial f_1 with small coefficients of degree $d_3 + 1$, where $d_3 \geq n$. To be valid, this polynomial must have an irreducible factor $\varphi(x) = \varphi_0 + \varphi_1 x + \cdots + \varphi_n x^n$ of degree n modulo p. Without loss of generality, this factor can be considered as monic. The polynomial φ is a valid candidate to be the polynomial that defines $\mathbb{F}_{p^n}$. To define f_0, we know that φ is a common factor of f_0 and f_1 modulo p. The valid candidates to define f_0 are therefore the linear combinations of px^i for i in $[0, d_3]$ and $x^j\varphi$, for j in $[0, d_3 - n]$. We can remark that px^i for i in $[n, d_3]$ can be written as $px^i = px^{n-i}\varphi - \sum_{k=0}^{n-1} \varphi_k p x^{k-n-i}$. We can therefore describe all the valid polynomials f_0 as $\sum_{i=0}^{n-1} \lambda_i p x^i + \sum_{i=0}^{d_3-n} \lambda_{i+n} x^i \varphi$, where the λ_i are integers. To minimize the coefficients of f_0, we can find a short vector in the lattice whose basis vectors are rows of the $(d_3+1) \times (d_3+1)$ matrix $M_{\varphi,d_3,n}$ following Equation (4.3).

If f_0 is irreducible, then the pair (f_0, f_1) is valid. The infinity norm of the shortest vector is bounded by $p^{n/(d_3+1)}$, following Theorem A.1. The coefficients of f_0 are therefore bounded by $O(p^{n/(d_3+1)})$. Due to the construction of the polynomials, it is not possible to enforce the same Galois action on both sides, but it can be done on the side 1.

Conjugation

This method was proposed by Barbulescu, Gaudry, Guillevic and Morain [20, Section 3.3] and has a theoretical impact on the complexity of NFS-HD in the general case, allowing us to reach a complexity in $L_{p^n}(1/3, 2.21)$.

We begin the polynomial selection by choosing three irreducible integer polynomials μ, g_0 and g_1, with small coefficients, where g_0 is of degree n, g_1 is of degree less or equal to n, and $\mu(x) = \mu_0 + \mu_1 x + x^2$ is quadratic and monic. If μ has two roots λ_0 and λ_1 in $\mathbb{F}_p$ and $g_0 + \lambda_0 g_1$ is irreducible over $\mathbb{F}_p$, we can define f_0 and f_1, otherwise, we need to find new polynomials g_0, g_1 and μ. We know that $\mu(x) \equiv (x - \lambda_0)(x - \lambda_1) \bmod p$. By evaluating μ in $-g_0/g_1$ and multiplying by g_1^2, we get the irreducible polynomial $(g_0 + \lambda_0 g_1)(g_0 + \lambda_1 g_1) = \mu_0 g_1^2 - \mu_1 g_0 g_1 + g_0^2$ and we set f_1 to this polynomial, therefore f_1 has degree $2n$ and small coefficients. A degree n factor of f_1 modulo p is $g_0 + \lambda_0 g_1$, and we now look for a polynomial f_0 that has this polynomial as a factor modulo p: a valid choice will be $f_0 = g_0 + \lambda_0 g_1$, but λ_0 has almost the same size as p. Thanks to the extended Euclidean algorithm, we can compute two integers b_0 and b_1 in $O(\sqrt{p})$ such that $\lambda_0 = b_1/b_0 \bmod p$ and then, we can set f_0 to $b_0 g_0 + b_1 g_1$: this polynomial has coefficients in $O(\sqrt{p})$ and degree n.

In order to have a monic polynomial f_1, the degree of g_0 is chosen to be smaller than n. The construction of f_1 given in this section corresponds to the one given in the original description, consisting in the computation of $\mathrm{Res}_y(\mu(y), g_0 + yg_1) = (g_0 + \lambda_0 g_1)(g_0 + \lambda_1 g_1)$.

As with the JLSV_1 polynomial selection, this construction allows the possibility to enforce a common Galois automorphism on both polynomials. This construction was used in some practical computation of discrete logarithms: two $\mathbb{F}_{p^2}$ of size 529 bits [19] and of size 595 bits [20] by Barbulescu, Gaudry, Guillevic and Morain, and three $\mathbb{F}_{p^3}$ of size respectively 512 bits by the same team [15], 508 bits by Guillevic, Morain and Thomé [91] and 592 bits by Gaudry, Guillevic and Morain [73].

Sarkar–Singh

This polynomial selection, called $\mathcal{A}$ by the authors, was introduced by Sarkar and Singh [160, Section 5]. It can be seen as a generalization of the GJL and the conjugation polynomial selections. This polynomial selection improves the complexity of the boundary case, that is when p is between the large and the medium characteristic.

The algorithm to perform the polynomial selection $\mathcal{A}$ needs as input the characteristic p, the extension degree n, a divisor d_5 of n and an integer r larger or equal to n/d_5. Let k be equal to n/d_5. We give here a quick overview of the polynomial selection before going into details. The following process is repeated until f_0 and f_1 are irreducible over $\mathbb{Z}$ and φ is irreducible over $\mathbb{F}_p$.

1. Select a polynomial μ of degree $r+1$, with small coefficients, irreducible over $\mathbb{Z}$ and which has an irreducible factor μ' of degree k modulo p.
2. Select g_0 and g_1 with small coefficients with g_0 of degree d_5 and g_1 of degree less or equal to d_5.
3. Define
   (a) $f_1 = \mathrm{Res}_y(\mu(y), g_0 + yg_1)$.
   (b) $\varphi = \mathrm{Res}_y(\mu'(y), g_0 + yg_1) \bmod p$.
   (c) ψ_0 as the polynomial whose coefficients are given by the shortest vector of the lattice generated by the rows of $M_{\mu',r,n}$, defined in Equation (4.3).
   (d) $f_0 = \mathrm{Res}_y(\psi_0(y), g_0 + yg_1)$.

We now explain why this polynomial selection gives a valid pair (f_0, f_1). The polynomial μ of degree r is irreducible over $\mathbb{Z}$, then the polynomial $f_1 = \mu(-g_0/g_1)g_1^r = \mathrm{Res}_y(\mu(y), g_0 + yg_1)$ is irreducible. The polynomial μ' is an irreducible factor of μ modulo p, then $\varphi = \mu'(-g_0/g_1)g_1^k = \mathrm{Res}_y(\mu'(y), g_0 + yg_1)$ divides f_1 modulo p. The polynomial ψ_0, which has generically degree $r+1$, is a linear combination of μ' and polynomial px^i, for i in $[0, r[$, then modulo p, the polynomial μ' divides ψ_0. Finally, f_0 is a multiple of φ modulo p, then f_0 and f_1 are divisible by φ, which defines $\mathbb{F}_{p^n}$.

With this description, f_1 has degree $d_5 r$ and small coefficients and f_0 has degree $d_5(r+1)$ and coefficients in $p^{n/(d_5(r+1))}$, following the classical analysis that involves short vectors of a lattice. We can remark that, if $d_5 = 1$, the

produced polynomials have the same shape as the one using the GJL polynomial selection, and if $d_5 = n$ and $r = k = 1$, we have the bounds of the conjugation method. As for the JLSV_1 and conjugation method, it is possible to enforce a Galois action of order d_5 for f_0 and f_1 by taking special forms for g_0 and g_1.

Summary of the polynomial selections

The following table summarizes the different polynomial selections described previously. The "Galois action" column is set to "none" if a Galois action of the same order cannot be applied on both sides, and to the maximal order of a possible Galois action otherwise. In order to have more room to select the polynomial f_0 and f_1, the infinity norm of f_1 in GJL and conjugation polynomial selection are often chosen to be in $O(\log p)$ instead of $O(1)$. This implies that the polynomial μ defined in conjugation and $\mathcal{A}$ has coefficients in $O(\log p)$.

The bounds on the different variables are the following. The variable ε for JLSV_1 is in $[0, 1/2[$ and depends on the size of the special-$\mathfrak{Q}$s we set. The degree d_2 in JLSV_2 must be larger or equal to n, as for the degree d_3 for the generalized Joux–Lercier. The degree d_5 for the polynomial selection $\mathcal{A}$ described by Sarkar and Singh is a positive divisor of n, and r is an integer larger or equal to n/d_5.

Variant	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$	Common Galois action
JLSV_0	n	small	n	p	none
JLSV_1	n	$p^{1/2-\varepsilon}$	n	$p^{1/2+\varepsilon}$	n
JLSV_2	n	$p^{n/(d_2+1)}$	$d_2 \geq n$	$p^{n/(d_2+1)}$	$\gcd(n, d_2)$
GJL	$d_3 \geq n$	$p^{n/(d_3+1)}$	$d_3 + 1$	small	none
Conjugation	n	$p^{1/2}$	$2n$	small	n
$\mathcal{A}$	$d_5 r \geq n$	$p^{n/(d_5(r+1))}$	$d_5(r+1)$	small	d_5

Table 4.1 – Polynomial selections for NFS-HD in $\mathbb{F}_{p^n}$, where d_5 divides n and $r \geq .n/d_5$.

In Section 4.4, we give a general framework to get the theoretical complexity. In the general case, the best polynomial selection is the conjugation one. However, the situation is not as clear when we perform practically a computation, and we often need to test two or more polynomial selectionss to select the best polynomial pair.

4.1.3 Practical results

About JLSV_0 and JLSV_2

The JLSV_0 and JLSV_2 polynomial selections seem to be surpassed by respectively the JLSV_1 and GJL polynomial selections. The JLSV_0 gives a polynomial f_1 skewed for only one coefficients and to balance this, we need to have the coefficients of the polynomial a with the skewness of f_1, that is $a_i/a_{i+1} = p^{1/n}$: this was observed by Zajac [185, Section 8.2]. This shape of sieving region is hardly compatible with the special-$\mathfrak{Q}$ method. The norm are clearly unbalanced and no Galois action can be enforced. This is why we prefer to use the JLSV_1 method, instead of the JLSV_0 polynomial selection.

Concerning the JLSV_2 polynomial selection, we use a similar reasoning as the one in [19, Section 6.1.2]. Let us consider an upper bound of the product of the norms which is $E^{d_2+n}p^{2n/(d_2+1)}$, where E is a bound on the sieving region and the degree of the polynomials we sieve is 1. If we try to take into account the skewness of the polynomial f_0, this product can be optimistically decreased to $E^{d_2+n}p^{3n/(2(d_2+1))}$. Using the GJL polynomial selection, the bound on the product of the norms drops to $E^{2d_3+1}p^{n/(d_3+1)}$. The ratio of the product of the norms given by the JLSV_2 and GJL polynomial selections is about $p^{1/n-2}E$, which is much smaller than 1 in practice. Another way to show that the GJL polynomial selection is closer to the optimal than the JLSV_2 polynomial selection is to compare the resultant between f_0 and f_1 for each polynomial selection. Using the bound on the resultant given in [34], we have $\text{Res}(f_0, f_1) \approx (n+1)^{d_2/2}(d_2+1)^{n/2}p^{n^2/(d_2+1)}p^{d_2 n/(d_2+1)}$ with the JLSV_2 polynomial selection and $\text{Res}(f_0, f_1) \approx (d_3+1)^{(d_3+1)/2}(d_3+2)^{d_3/2}p^n$ with the GJL polynomial selection. We know that p^n must divide the resultant between the two polynomials, and a good polynomial selection should not exceed too much p^n. Forgetting the smaller term in the two resultants, we get that $\text{Res}(f_0, f_1) = p^{n(n+d_2)/(d_2+1)}$ for the JLSV_2 polynomial selection, instead of $\text{Res}(f_0, f_1) = p^n$ for the GJL polynomial selection. Therefore, the GJL polynomial selection seems to outperform the JLSV_2 polynomial selection in any case.

Short discussion on the extension degree n

In our arsenal of polynomial selections, we now consider the JLSV_1, GJL, conjugation and $\mathcal{A}$ polynomial selections.

Prime extension. When n is a prime, the parameter d_5 of the polynomial selection $\mathcal{A}$ can take the value 1 or n. The resultant between f_0 and f_1 is bounded by the formula given in [34], which can be separated on two parts: the significant one, which depends only on n, and the frequently avoided part, which is equal to $(d_5 r+1)^{(d_5(r+1))/2}(d_5(r+1)+1)^{d_5 r/2}$. We reach the minimum when $d_5 = 1$ and $r = n$ or when $d_5 = n$ and $r = 1$; there are the GJL or conjugation methods. The polynomial selection $\mathcal{A}$ contains therefore only the GJL and conjugation methods.

If a Galois action can occur on the two sides, it seems difficult that the possible smaller norms reached by the GJL polynomial selection can compensate the advantage of using the Galois action. The only remaining choices are the two polynomial selections: the JLSV_1 and conjugation methods.

Composite extension. For simplicity, we do not cover the case where $r > n/d_5$ for the polynomial selection $\mathcal{A}$. As for the prime extension, the GJL polynomial selection seems not to be competitive when a Galois action occurs. It is however necessary to deal with all the divisors of n for the polynomial selection $\mathcal{A}$.

The following paragraph can also be applied in prime extension. If we use the unbalanced version of the JLSV_1 polynomial selection, it seems easier to find polynomials with a good, very negative, α quantity: the polynomial f_1 has coefficients larger than f_0 and adding or removing a small multiple of f_0 does

not affect the infinity norm or the Galois action and can decrease $\alpha(f_1)$. It is not possible to do such a technique for all the other polynomial selections without changing the degree or the infinity norm of one of the polynomials. This method can also be applied for the balanced JLSV_1 polynomial selection, with less freedom.

Experimental results

There exist in the literature two reports about the comparison of polynomial selections: one about an $\mathbb{F}_{p^3}^*$ of size 508 bits [91] with a two-dimensional sieving step and one about an $\mathbb{F}_{p^6}^*$ of size 300 bits, which can be found in our article [72], with a three-dimensional sieving step. The number of relations per $\mathfrak{Q}$ is given without taking into account the Galois action, then, for a fair comparison, the number of relations per $\mathfrak{Q}$ must be multiplied by the order of the possible Galois action.

As remarked before, the GJL polynomial in Table 4.2 cannot be competitive, even if the number of relations per $\mathfrak{Q}$ is larger than using the JLSV_1 method. We can also notice that the Murphy-E are close to each other and that the Murphy-E function takes smaller values for the GJL than for the JLSV_1 polynomials, but the number of relations does not follow this order. This problem arises also in factorization (see for example the bug number 21311 of CADO-NFS). The data of Table 4.2 are extracted from the corresponding article, with some additional data provided by the authors.

	JLSV_1	Conjugation	GJL
α values	$-3.0, -2.8$	$-4.16, -2.94$	$-2.1, 1.2$
Murphy E	$2^{-39.9}$	$2^{-39.5}$	$2^{-40.9}$
Special-$\mathfrak{q}$ side	0 and 1	0 and 1	0
Infinity norms	$2^{84.7}, 2^{85.3}$	$2^{85.1}, 2^{8.4}$	$2^{129.4}, 2^{2}$
Galois action	3	3	none
Smoothness bounds	$2^{27}, 2^{27}$	$2^{27}, 2^{27}$	$2^{28}, 2^{26}$
Relations per $\mathfrak{Q}$	4.2	5.9	4.9

Table 4.2 – Experiments on sieving in a 508-bit $\mathbb{F}_{p^3}$ (data from [91]).

The size of norms for Table 4.3 are given after removing the contribution of the special-$\mathfrak{Q}$s. The smallest size of the $\mathbb{F}_{p^6}$ is suitable for the unbalanced JLSV_1, but the trend reported in this table is probably not as important for larger sizes. The smoothness bounds are the same for all the polynomial selection, which are $(2^{25}, 2^{25})$.

	Unbal. JLSV_1	Conjugation	$\mathcal{A}$ (8, 6)	$\mathcal{A}$ (9, 6)
α values	$-12, -4.9$	$-6.4, -0.8$	$-4.6, 1.2$	$-6.5, 1.9$
Murphy E	$2^{-19.0}$	$2^{-27.3}$	$2^{-22.5}$	$2^{-23.0}$
Special-$\mathfrak{q}$ side	1	1	0	0
Average of norms	$2^{128}, 2^{139}$	$2^{148}, 2^{251}$	2^{143}, 2^{153}	2^{144}, 2^{186}
Galois action	6	6	2	3
Relations per $\mathfrak{Q}$	25.7	0.2	1.2	0.9

Table 4.3 – Experiments on sieving in a 300-bit $\mathbb{F}_{p^6}$.

4.2 Relation collection

After the polynomial selection, we have a representation of the field that helps us to perform the relation collection the most efficiently. Keeping the notation of Section 3.2.1, we can anew give the classical way to perform the relation collection. We recall that the enumeration bounds b_0 and b_1 are respectively less than B_0 and B_1, t_0 and t_1 are the thresholds, that is the largest remaining norms we cofactorize.

Selection. For i in $[0, 1]$,

- **Initialization.** Compute the norm in K_i of all the polynomials a in $\mathcal{S}$ and store them in an array T_i indexed by $(a_0, a_1, \ldots, a_{t-1})$,
- **Enumeration.** For all prime ideals $\mathfrak{Q}$ in $\mathcal{F}_i$ of norms below b_i, compute the $\mathfrak{Q}$-lattice and divide by q all the cells at index $(a_0, a_1, \ldots, a_{t-1})$, such that a is in $\mathcal{S}$ and in the $\mathfrak{Q}$-lattice,

Cofactorization. For all coprime tuple $(a_0, a_1, \ldots, a_{t-1})$ in $\mathcal{S}$, if $T_0[(a_0, a_1, \ldots, a_{t-1})]$ is less than t_0 and $T_1[(a_0, a_1, \ldots, a_{t-1})]$ is less than t_1, perform the full factorization of the norm of $a_0 + a_1 x + \cdots + a_{t-1}x^{t-1}$ in K_0 and K_1. If the norms are smooth for both sides, the polynomial a gives a valid relation.

If $t = 2$, we get the description given in Section 3.2.1. The parameter t is however very important in NFS-HD, because using polynomials of degree higher than 1, that is dealing with lattice of dimension higher than 2, allows us to reach the complexity in $L(1/3)$, which can be impossible given p and n if t is equal to 2. In the following, we will describe how to define the ideals and how to use the special-$\mathfrak{Q}$ method. The sieving algorithms will be described in Chapter 6.

4.2.1 Building the lattice of an ideal

Let $\mathfrak{Q}$ be a prime ideal of $\mathcal{O}_0$ of norm q^d, where q is a prime and d is the degree of the ideal. Except in few cases, we can represent $\mathfrak{Q}$ as a pair (q, g), where g is a polynomial of degree d dividing f_0 modulo q. Given t, we take into account during the sieving step all the possible $\mathfrak{Q}$ that can appear in the factorization of the norm of a on the side 0. The ideals that can be involved in this factorization have necessarily a degree d less than t: let us write g as $g = g_0 + g_1 x + \cdots + g_d x^d$ with the leading coefficient of g equal to 1. We know that, if the ideal $\mathfrak{Q}$ of degree one contains $a(\theta_0)$, where a is of degree one, the $a \equiv 0 \bmod (q, g)$. The same occurs if a is of degree $t-1$ and the degree d of $\mathfrak{Q}$ is less or equal to $t-1$. Such polynomials a can therefore be written as $\lambda_i q x^i + \mu_j x^j g(x)$, where i is in $[0, t[$, j is in $[0, t-d[$ and λ_i and μ_j are integers. We can remark that, when $i \geq d$, qx^i can be written as linear combinations of $x^j g$ and qx^k, where j is in $[0, i-d]$ and k in $[0, d]$. Then, the coefficients of a polynomial a in $\mathfrak{Q}$ are given by a linear combination of the basis of a lattice, where the basis vectors are rows of the following $t \times t$ matrix

$$M_{\mathfrak{Q}} = \begin{pmatrix} q & & & & & & & \\ & q & & & & & & \\ & & \ddots & & & & & \\ & & & q & & & & \\ g_0 & g_1 & \cdots & g_{d-1} & 1 & & & \\ & g_0 & g_1 & \cdots & g_{d-1} & 1 & & \\ & & \ddots & \ddots & & \ddots & \ddots & \\ & & & g_0 & g_1 & \cdots & g_{d-1} & 1 \end{pmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right\} d \\ \left.\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right\} t-d \end{matrix} . \quad (4.4)$$

4.2.2 Dividing the search space

In higher dimension, the special-$\mathfrak{Q}$ method can be applied as in two dimensions. In two dimensions, it is not possible to perform the computation of discrete logarithms in a large size finite field without applying a special-$\mathfrak{Q}$ method. It seems reasonable that the same thing applies in higher dimensions, but the first use of a special-$\mathfrak{Q}$ method in higher dimensions to perform the relation collection reported by Zajac in [186] in an $\mathbb{F}_{p^6}$ of size 240 bits was not conclusive. In 2015, Hayasaka, Aoki, Kobayashi and Takagi attacked again the same field, using the same polynomial selection, but applying a special-$\mathfrak{Q}$ method described by the same authors in [93] and an adapted sieve for the three-dimensional case and achieved the computation of Zajac in about the same CPU core time [94]. In the following, we will especially describe the approach of [93] to compute the $\mathfrak{R}$-lattice in a special-$\mathfrak{Q}$-lattice, that is a description of the set of polynomials a that are in the ideals $\mathfrak{R}$ and $\mathfrak{Q}$. We denote by (q, g) the ideal $\mathfrak{Q}$ and by (r, h) the ideal $\mathfrak{R}$, where q and r are primes and g and h are integer polynomials of degree respectively $d_{\mathfrak{Q}}$ and $d_{\mathfrak{R}}$. A polynomial a in $\mathfrak{Q}$ and $\mathfrak{R}$ verify the two modular equations $a \equiv 0 \bmod (q, g)$ and $a \equiv 0 \bmod (r, h)$.

The polynomial a is in $\mathfrak{Q}$ and the coefficients of a are given by linear combinations of the rows of $M_{\mathfrak{Q}}$ (4.4). To enumerate the polynomial a in $\mathcal{S}$ that are in $\mathfrak{Q}$, we compute $\boldsymbol{a} = \boldsymbol{c}M_{\mathfrak{Q}}$, where $\boldsymbol{c}$ is an integer vector in $\mathcal{H} = [H_0^m, H_0^M[\times[H_1^m, H_1^M[\times \cdots \times [H_{t-1}^m, H_{t-1}^M[$, a t-sieving region. The matrix $M_{\mathfrak{Q}}$ is composed by sparse vectors with large coefficients, which is not convenient to try to enumerate the polynomials a in $\mathcal{S}$ that are in $\mathfrak{Q}$. We explore a part of the intersection of the $\mathfrak{Q}$-lattice and the searching space $\mathcal{S}$ by performing linear combinations of a basis of the $\mathfrak{Q}$-lattice, the coefficients of the linear combination are bounded by $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times \cdots \times [H_{t-1}^m, H_{t-1}^M[$: it is therefore common to deal with a reduced basis of the $\mathfrak{Q}$-lattice, given by a basis reduction of $M_{\mathfrak{Q}}$, eventually with some skewness if $\mathcal{S}$ is skew. Let $M_{\mathfrak{Q}}^{\mathrm{BR}}$ be the $t \times t$ matrix whose rows form a reduced basis of $\mathfrak{Q}$: we therefore enumerate $\boldsymbol{a} = \boldsymbol{c}M_{\mathfrak{Q}}^{\mathrm{BR}}$.

The polynomial a is also in $\mathfrak{R}$. In the $\mathfrak{R}$-lattice, we know that the coefficients of a polynomial a are given by a linear combination of the rows of the matrix $M_{\mathfrak{R}}$. We use a modified version of the matrix $M_{\mathfrak{R}}$, denoted $\mathcal{M}_{\mathfrak{R}}$, which results of some linear combinations of the rows of the matrix to get

$$\mathcal{M}_{\mathfrak{R}} = \left(\begin{array}{c|c} rI_{d_{\mathfrak{R}}} & 0 \\ \hline T_{\mathfrak{R}} & I_{t-d_{\mathfrak{R}}} \end{array}\right), \quad (4.5)$$

where I_k is the identity matrix of size $k \times k$. The coefficients $\boldsymbol{a}$ of a are given by a linear combination of $\mathcal{M}_{\mathfrak{R}}$, that is $\boldsymbol{a} = \boldsymbol{d}\mathcal{M}_{\mathfrak{R}}$, where $\boldsymbol{d}$ is an integer vector of dimension t. Because $(a_{d_{\mathfrak{R}}}, a_{d_{\mathfrak{R}}+1}, \ldots, a_{t-1}) = (d_{d_{\mathfrak{R}}}, d_{d_{\mathfrak{R}}+1}, \ldots, d_{t-1})$, we obtain the relation $(a_0, a_1, \ldots, a_{d_{\mathfrak{R}}-1}) \equiv (a_{d_{\mathfrak{R}}}, a_{d_{\mathfrak{R}}+1}, \ldots, a_{t-1})T_{\mathfrak{R}} \bmod r$. By replacing the coefficients of a by their expression $\boldsymbol{a} = \boldsymbol{c}M_{\mathfrak{Q}}^{\mathrm{BR}}$, we can obtain a relation involving $\mathfrak{Q}$, $\mathfrak{R}$ and $\boldsymbol{c}$. Let $M_{\mathfrak{Q}}^{\mathrm{BR}}$ be divided into two blocks $(M_{\mathfrak{Q}}^0 | M_{\mathfrak{Q}}^1)$ such that $(a_0, a_1, \ldots, a_{d_{\mathfrak{R}}-1}) = \boldsymbol{c}M_0$ and $(a_{d_{\mathfrak{R}}}, a_{d_{\mathfrak{R}}+1}, \ldots, a_{t-1}) = \boldsymbol{c}M_1$. We obtain therefore the relation

$$\boldsymbol{c}(M_{\mathfrak{Q}}^0 - M_{\mathfrak{Q}}^1 T_{\mathfrak{R}}) \equiv 0 \bmod r. \tag{4.6}$$

By looking for t linearly independent vectors that solve Equation 4.6, we can build the matrix of the $\mathfrak{R}$-lattice in the $\mathfrak{Q}$-lattice, which generically follows the form

$$M_{\mathfrak{Q},\mathfrak{R}} = \left(\begin{array}{c|c} rI_{d_{\mathfrak{R}}} & 0 \\ \hline * & I_{t-d_{\mathfrak{R}}} \end{array}\right). \tag{4.7}$$

The goal of the sieve algorithms described in Chapter 6 is to enumerate all the elements in the intersection of the lattice formed by the rows of $M_{\mathfrak{Q},\mathfrak{R}}$ and the sieving region $\mathcal{H}$.

4.3 Individual logarithm

At this step, we have found almost all the virtual logarithms of the factor basis, coming from the linear algebra step. Using these virtual logarithms, we want to compute the discrete logarithm of a target T in $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/\varphi(x)$ modulo ℓ, a large prime factor of $p^n - 1$. As in Chapter 3, we need to lift $h = h_0 + h_1 x + \ldots + h_{n-1}x^{n-1}$ in one of the number fields. We describe first the rational reconstruction method, similar to the one described in Section 3.4.1, and a new method proposed by Guillevic in [86, 88] that improves the norm of z, the element in the number field K_1 defined by the irreducible polynomial f_1 of degree d such that z in K_1 maps to T in $\mathbb{F}_{p^n}$ (the element T could be lifted on the side 0 in the same way). The target T can always be considered as monic thanks to the following lemma:

Lemma 4.1 ([86, Lemma 2])**.** *Let T be an element of $\mathbb{F}_{p^n}^*$. Let ℓ be a divisor of the order of $\mathbb{F}_{p^n}^*$ that does not divide the order of the multiplicative group of a proper subfield of $\mathbb{F}_{p^n}$. Let $T' = uT$, where u is in a proper subfield of $\mathbb{F}_{p^n}$. Then,* $\log T = \log T' \bmod \ell$.

Let T' be equal to $T/\operatorname{lc}(T)$ in $\mathbb{F}_{p^n}$. From Lemma 4.1, we get $\log T \equiv \log T' \bmod \ell$. We view the coefficients $T_0, T_1, \ldots, T_{n-1}$ in $\mathbb{F}_p$ of T as integers. A simple way to lift T on K_1 is to use the lift $T_0 + T_1\theta_1 + \ldots + T_{n-2}\theta_1^{n-2} + \theta_1^{n-1}$. The norm of this element is large, typically in $O(p^d \|f_1\|_\infty^{n-1})$, and we try to reduce it with the two following methods, to have a better probability to involve ideals of norms below a smoothness bound B for the booting step.

4.3.1 Rational reconstruction over number field

Let $\mathfrak{P}$ be the ideal of $\mathcal{O}_1$ defined as $(p, \varphi(x))$. As in prime field, let z_0 and z_1 be two elements in K_1 such that $T = z_0/z_1 \bmod \mathfrak{P}$, where z_0 is equal to

$z_{0,0} + z_{0,1}\theta_1 + \cdots + z_{0,d-1}\theta_1^{d-1}$ and z_1 to $z_{1,0} + z_{1,1}\theta_1 + \cdots + z_{1,d-1}\theta_1^{d-1}$. These $2d$ integers verify the equation $z = z_0/z_1 \bmod \mathfrak{P}$ if they are equal to a linear combination of vectors of the lattice whose basis vectors are rows of the following $2d \times 2d$ matrix, where I_d is the $d \times d$ identity matrix,

$$\left(\begin{array}{cccccccc|c}
 & & pI_n & & & & & & \\
\hline
\varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & 1 & & & & \\
 & \varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & 1 & & & \\
 & & \ddots & \ddots & & \ddots & \ddots & & \\
 & & & \varphi_0 & \varphi_1 & \cdots & \varphi_{n-1} & 1 & \\
\hline
 & & & & T(\theta_1) & & & & \\
 & & & & T(\theta_1)\theta_1 & & & & \\
 & & & & \vdots & & & & I_d \\
 & & & & T(\theta_1)\theta_1^{d-1} & & & &
\end{array}\right)
\begin{array}{l}
\left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} n \\
\left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} d-n \\
\left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} d
\end{array}.$$

By finding the shortest vector of this lattice, we can find z_0 and z_1 with infinity norm close to $p^{n/(2d)}$ (see Theorem A.1. The norms in K_1 of z_0 and z_1 are therefore bounded by $\|f_1\|_\infty^{d-1} p^{n/2}$. For some of the polynomial selections in our arsenal, we summarize in Table 4.4 the norms of $z_{\{0,1\}}$, the product of the norms of z_0 and z_1 that gives the bound on the complexity of the B-smoothness test for z_0 and z_1 and the norm of the targeted element T.

Variant	$\mathrm{Norm}(z_{\{0,1\}}(\theta_1)) = N_1$	N_1^2	$\mathrm{Norm}(T(\theta_1))$
JLSV_1 (balanced)	$p^{n-1/2}$	p^{2n-1}	$p^{(3n-1)/2}$
GJL	$p^{n/2}$	p^n	p^{d_3}
Conjugation	$p^{n/2}$	p^n	p^{2n}
$\mathcal{A}$	$p^{n/2}$	p^n	$p^{d_5(r+1)}$

Table 4.4 – Bound on the norm of $z_{\{0,1\}}$ for different polynomial selections.

We report only the norm of z_0 and z_1 for a lift on side 1: indeed, the lift on side 0 is either the same or larger. We can remark that the only polynomial selection that cannot allow us to have a rational reconstruction which gives less chance to be B-smooth than multiplying T by a random power of the generator g is for the JLSV_1 polynomial selection.

4.3.2 Reducing the coefficients of the target

Using Lemma 4.1, Guillevic observes that there exist many elements in $\mathbb{F}_{p^n}^*$ that have the same discrete logarithm: the number of such elements is finite, there necessarily exists one with the smallest norm in K_1, that is essentially with the smallest coefficients. The goal of [88, Algorithm 4] is to find one of the elements with small coefficients, let us call it T'. This section is based on a description of Guillevic in [88, Section 4]. To find a preimage of T with small coefficients, we use different properties. The first one is that if we add a multiple of px^i

to T, where i is in $[0, d[$, then $\log(T + jpx^i) \equiv \log T \bmod \ell$, where j is in $\mathbb{F}_p$. The second one is that if we add a multiple $x^i\varphi$ to T, where i is in $[0, d-n[$, then $\log(T + jx^i\varphi) = \log T \bmod \ell$, where j is in $\mathbb{F}_p$. The last one, following Lemma 4.1, is to use a basis of the largest subfield of $\mathbb{F}_{p^n}$.

These three properties help us to find a suitable polynomial T'. Let k be the largest proper divisor of n: the field $\mathbb{F}_{p^k}$ is a proper subfield of $\mathbb{F}_{p^n}$. Let $(1, u, u^2, \ldots, u^{k-1})$ be a polynomial basis of this subfield, where u is a polynomial in $\mathbb{F}^*_{p^n}$. Using Lemma 4.1, we know that multiplying T by a linear combination of the basis polynomials does not change its logarithm: the linear combination achieving the lowest coefficients is obtained by finding the shortest vector of the lattice generated by the coefficients of $(T, uT, u^2T, \ldots, u^{k-1}T)$. This lattice admits an echelon form and even a reduced echelon form $\mathcal{E} = (e_0, e_1, \ldots, e_{k-1})$ because each basis vector is defined in $\mathbb{F}_p$ and then each basis vector can be multiplied by the inverse of the pivot. This basis E gives some of the basis vectors $\mathcal{B}$ of the lattice that generates the coefficients of T'. We add to $\mathcal{B}$ the vectors formed by the coefficients of $x^i\varphi(x)$, for i in $[0, d-n[$. For now, the basis does not contain the contribution of px^i, for i in $[0, d[$. As usual, the polynomial px^i for i in $[n-d, d[$ can be generated by a linear combination of u^iT and $x^j\varphi(x)$, where i is in $[0, d[$ and j in $[0, d-n[$. The basis $\mathcal{B}$ is therefore composed by the coefficients of:

- px^i, for i in $[0, n-k[$;
- u^iT, for i in $[0, k[$;
- $x^i\varphi(x)$, for i in $[0, d-n[$.

The volume of this lattice is equal to p^{n-k}. The smallest vector of this lattice gives the smallest possible coefficients of the polynomial T' and have a good chance to reach a smallest norm in K_1 than the one of T: as usual, the infinity norm of T' is close to $p^{(n-k)/d}$: we summarize the result of this method in Table 4.5.

Variant	Section 4.3.1	Norm$(T(\theta_1))$	Norm$(T'(\theta_1))$
JLSV_1 (balanced)	p^{2n-1}	$p^{(3n-1)/2}$	$p^{(3n-1)/2-k}$
GJL	p^n	p^{d_3}	p^{n-k}
Conjugation	p^n	p^{2n}	p^{n-k}
$\mathcal{A}$	p^n	$p^{d_5(r+1)}$	p^{n-k}

Table 4.5 – Bound on the norm of T' using Guillevic's method for different polynomial selections and comparison with rational reconstruction.

4.4 Complexity analysis

In this section, we give a general result for the complexity of NFS, given a polynomial selection. Let f_0 be a polynomial of degree $k_{0,0}n$ and of infinity norm equal to $p^{k_{0,1}}$ and f_1 be a polynomial of degree $k_{1,0}n$ and of infinity norm equal to $p^{k_{1,1}}$, where $k_{i,j}$ are real numbers and $k_{i,0} \geq 1$. Let E be the infinity norm of the polynomial a of degree t and B be the smoothness bound for both sides. We

recall that the resultant between a and f_0 can be approximated by $E^{k_{0,0}n}p^{k_{0,1}t}$. Following [24, 143], we define $p = L_Q(l_p, c_p)$, where l_p is in $]1/3, 2/3[$ and $Q = p^n$, $B = L_Q(1/3, c_b)$, $E = L_Q(l_p - 1/3, c_e c_p)$ and $t = c_t/c_p(\log Q/\log\log Q)^{2/3-l_p}$. By following the analysis given in the previous articles or like in Appendix D, we get

- $c_e c_t = 2c_b$;
- the norm on side 0 is in $L_Q(2/3, k_{0,0}c_e + k_{0,1}c_t)$;
- the norm on side 1 is in $L_Q(2/3, k_{1,0}c_e + k_{1,1}c_t)$;
- if P is the probability of a polynomial a to be doubly smooth, $B = 1/P$.

Using Corollary 1.1, the probability of smoothness on side 0 is equal to $L_Q(1/3, -(k_{0,0}c_e + k_{0,1}c_t)/(3c_b))$ and $L_Q(1/3, -(k_{1,0}c_e + k_{1,1}c_t)/(3c_b))$ on side 1. We obtain that $3c_b^2 = k_{0,0}c_e + k_{0,1}c_t + k_{1,0}c_e + k_{1,1}c_t$, then $3c_b^2 c_t - 2(k_{0,0} + k_{1,0})c_b - (k_{0,1} + k_{1,1})c_t^2 = 0$. We try to minimize the overall complexity of NFS, in $L_Q(1/3, 2c_b)$, under this constraint. Using Lagrange multipliers, we get that $c_t = 3c_b^2/(2(k_{0,1} + k_{1,1}))$. Using the value of c_t in the constraint, we get $c_b = (8(k_{0,0} + k_{1,0})(k_{0,1} + k_{1,1})/9)^{1/3}$, that is a complexity of NFS in

$$L_Q\left(\frac{1}{3}, \sqrt[3]{\frac{64\,C}{9}}\right) = L_Q\left(\frac{1}{3}, \sqrt[3]{\frac{64\,(k_{0,0} + k_{1,0})(k_{0,1} + k_{1,1})}{9}}\right).$$

Variant	$k_{0,0}$	$k_{0,1}$	$k_{1,0}$	$k_{1,1}$	C
JLSV_0	1	0	1	1	2
JLSV_1	1	1/2	1	1/2	2
JLSV_2	1	$n/(d_2+1)$	d_2/n	$n/(d_2+1)$	$2(n+d_2)/(d_2+1)$
GJL	d_3/n	$n/(d_3+1)$	$(d_3+1)/n$	0	$1 + d_3/(d_3+1)$
Conjugation	1	1/2	2	0	3/2
$\mathcal{A}$	$d_5 r/n$	$n/(d_5(r+1))$	$d_5(r+1)/n$	0	$1 + r/(r+1)$

Table 4.6 – Coefficients of the polynomial selections for NFS-HD in $\mathbb{F}_{p^n}$.

For the example of the JLSV_0, JLSV_1 and conjugation polynomial selections, we get, using Table 4.6, the announced complexity, that is $(128/9)^{1/3}$ and $(96/9)^{1/3}$. We leave at an open question the possibility to find a new polynomial selection that reach a lower complexity than the obtained with the conjugation polynomial selection. An important constraint is that $\text{Res}(f_0, f_1) = p^n$, that is $k_{0,1}k_{1,0} + k_{0,0}k_{1,1} = 1$.

Let $d \geq 1$. If f_0 has degree n and infinity norm equals to $p^{1/d}$ and f_1 has degree dn and infinity norm equals to $O(1)$, we can reach a better complexity, in $L_Q(1/3, (64/9(d+1)/d)^{1/3})$, than the one reached by the conjugation polynomial selection. However, as of current knowledge, this complexity can be reached only when p has a special form, as we will describe in the following section.

4.5 The special and multiple NFS algorithms

4.5.1 The special NFS algorithm

As in prime field, a special construction for the characteristic of the targeted field $\mathbb{F}_{p^n}$ can be used to improved the running time of NFS to compute dis-

crete logarithms. The special NFS variant of Joux–Pierrot [108] is particularly well designed for computing discrete logarithms in fields used for pairing-based cryptography: indeed, it is hard to ensure that a pairing maps to a finite field $\mathbb{F}_{p^n}$ with a small n, and there exist constructions (see for example the survey of Freeman, Scott and Teske [63]) that allow to reach a nice field $\mathbb{F}_{p^n}$, where p is defined as the evaluation of a polynomial P in a variable u, where P has degree d that does not depend on p and small coefficients and u is small compared to p. Let f_0 be an irreducible polynomial of degree n equal to $\lambda x^n + r(x) - u$, where r is a degree d_r polynomial with small coefficients and λ a small integer, often equal to 1. The polynomial f_1 is defined as $P(\lambda x^n + r(x))$ and is a valid polynomial because $f_1 = P(f_0 + u) \equiv p \bmod f_0$ and then, f_0 divides f_1 modulo p. With such a construction, the polynomial f_0 has coefficients in $O(p^d)$ and degree n, and f_1 has coefficients in $O((d_r + 1)^\lambda)$ and degree dn. Some configurations for the choice of d_r are reported in [108], resulting in a complexity of $L_{p^n}(1/3, (64/9 \cdot (d+1)/d)^{1/3})$.

4.5.2 The multiple NFS algorithm

The multiple NFS algorithm we will describe here comes from the development of multiple NFS presented in Section 3.6.2 and we keep the same pieces of notation: the integer V is the number of number fields needed to reach the best complexity. It seems that, for all polynomial selections described in Section 4.1, we can derive a multiple NFS variant: in the following, we list those that allowed historically to reduce the complexity of NFS in medium characteristic and in the boundary case $p = L_{p^n}(2/3, \cdot)$. We begin with a variant proposed by Zajac in 2008, never analyzed before, and continue with the classical MNFS variant, the one of Barbulescu–Pierrot [24], the one of Pierrot [143] and finally the one of Sarkar–Singh [160].

The first proposition of adapting to $\mathbb{F}_{p^n}$ the multiple number field sieve algorithm over prime field was described in 2008 by Zajac [185, Section 6.4]. The construction is derived from the JLSV_0 polynomial selection. Let f_0 be a polynomial of degree n with small coefficients. It is possible to build a valid polynomial f_1 as $f_0 + ph_1$, where h_1 is a polynomial, if $f_0 + ph_1$ is irreducible. To not increase the degree of f_1, we choose a polynomial h_1 of degree less than n: it is possible to produce many polynomials f_i as $f_0 + ph_i$, where h_i is a polynomial of degree less than n and i is in $[1, V[$, that are equal to f_0 modulo p. If the f_i are irreducible, the number field K_i defined as $\mathbb{Q}[x]/f_i(x)$ are defined to be compatible in a multiple number field sieve approach. If E is the bound on the coefficients of the polynomial a, the bound of the norm of a in K_0 is almost equal to E^n and $E^n p^t$ in the other number fields. This leads to an asymmetric MNFS algorithm, that is a relation is given by a polynomial a if the norm of a is smooth in K_0 and an other K_i, as depicted in Figure 3.6. We analyze the complexity in Appendix D and show that the complexity of this variant is equal to $L_{p^n}(1/3, 2.40)$.

In 2014, Barbulescu and Pierrot introduced in [24] an algorithm that can be applied on the medium characteristic case using multiple number fields that reach also $L_{p^n}(1/3, 2.40)$. This MNFS algorithm is obtained by extending the JLSV_1 polynomial selection. Let f_0 and f_1 be the two polynomials built during the JLSV_1 polynomial selection described in Section 4.1. If the polynomial $f_0 + f_1$ is irreducible, this polynomial shares a common factor with f_0 and f_1 in

$\mathbb{F}_{p^n}$. More generically, all the polynomials $f_i = \alpha_{i,0} f_0 + \alpha_{i,1} f_1$ have a common factor, where i is in $[3, V[$ and $\alpha_{i,0}$ and $\alpha_{i,1}$ are two integers: to not increase the norm on the side i, the coefficients $\alpha_{i,0}$ and $\alpha_{i,1}$ are in $\sqrt{V}$. In all sides, the norms are almost equal: we therefore need to perform a symmetric MNFS algorithm, that is a relation can be found by involving at least two sides, but there does not exists a favored one as in almost all the MNFS variants, as show in Figure 4.4.

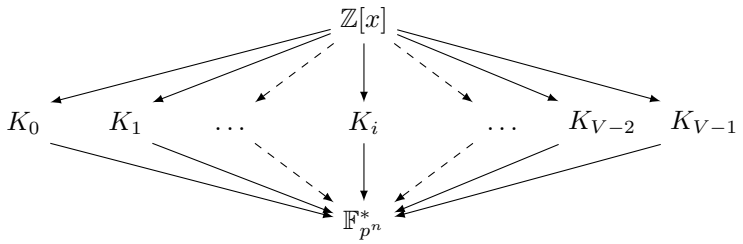


Figure 4.4 – The symmetric multiple NFS diagram for $\mathbb{F}_{p^n}$ using the Barbulescu–Pierrot variant: no particular field is favored.

In 2015, Pierrot [143] proposed to use the conjugation polynomial selection of Barbulescu, Gaudry, Guillevic and Morain [19] as the polynomial selection to define multiple number fields. This variant allows to reach a new complexity in $L_{p^n}(1/3, 2.16)$. Let consider that f_0 and f_1 are as defined in the description of the conjugation polynomial selection described in Section 4.1. The norm on side 1 is smaller than in side 0: it seems then interesting to try to have a polynomial with the same properties than the one of f_1 but it is does not seem to be possible. We therefore stick to build polynomials with the same shape as the one of f_0. This automatically leads to an asymmetric MNFS algorithm, where the side 1 is the favored side. To build the $V-2$ other polynomials $f_2, f_3, \ldots, f_{V-1}$, we use the fact that the rational reconstruction of λ_0 can be equal to $b_1/b_0 \equiv b_3/b_2 \bmod p$, where b_2 and b_3 are in $O(\sqrt{p})$ and the rational reconstruction (b_0, b_1) and (b_2, b_3) are linearly independent over $\mathbb{Q}$. It is therefore possible to build a polynomial f_2 that have a common factor with f_0 and f_1 as $f_2 = b_2 g_0 + b_3 g_1$. To build the $V-3$ other polynomials, we can consider linear combination of f_0 and f_2 as $f_i = \alpha_{i,0} f_0 + \alpha_{i,1} f_2$, where i is in $[3, V[$ and $\alpha_{i,0}$ and $\alpha_{i,1}$ are integers in $O(\sqrt{V})$. The norms on side $0, 2, 3, \ldots, V-1$ have almost the same size, larger than the norms on side 1. Therefore, the relation collection is performed involving the factorization in ideals in the side 1 and an other side, which is in $\{0, 2, 3, \ldots, V-1\}$, as depicted in Figure 4.5.

Concerning the boundary case, that is $p = L_{p^n}(2/3, \cdot)$, the best suited polynomial selection is the one of Sarkar–Singh, the polynomial selection $\mathcal{A}$. Let f_0 and f_1 be defined as in the original description. A valid polynomial f_2 can be built using ψ_1, a polynomial whose coefficients are given by the second minimum of the lattice defined by the rows of $M_{\mu',r,n}$, as $\mathrm{Res}_y(\psi_1(y), g_0 + y g_1)$, which can have the same shape as the shortest vector. If f_2 is irreducible, it is possible to build $V-3$ other polynomials f_i, where i is in $[3, V[$, as $f_i = \alpha_{i,0} f_0 + \alpha_{i,1} f_2$, where $\alpha_{i,0}$ and $\alpha_{i,1}$ are integers in $O(\sqrt{V})$. As in MNFS using the conjugation polynomial selection, the side 1 is predominant to find relations. The different complexity depending on the parameters r, d and k are summarized in [160,

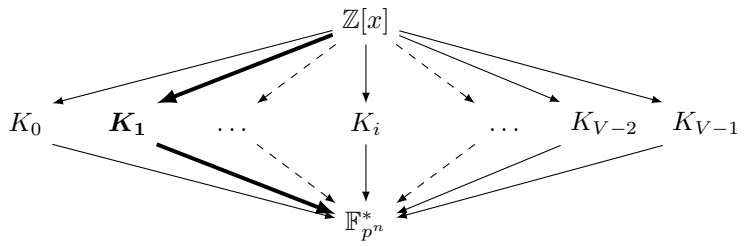


Figure 4.5 – The asymmetric multiple NFS diagram for $\mathbb{F}_{p^n}$ using the Pierrot variant: the side 1 is predominant.

Figure 4].

Chapter 5

The extended tower number field sieve algorithm

The extended tower number field sieve (exTNFS) algorithm, developed by Kim and Barbulescu [114], is the algorithm that reaches the best know complexity for finite fields of medium characteristic, when the extension degree is composite. The exTNFS algorithm has a complexity between $L_{p^n}(1/3, (64/9)^{1/3})$ and $L_{p^n}(1/3, (48/9)^{1/3})$, this lowest complexity arising when n has a factor of an appropriate size depending on the size of p^n. It is based on the tower number field sieve (TNFS) algorithm, an idea of Schirokauer [162] for the large characteristic, analyzed by Barbulescu, Gaudry and Kleinjung in [22] to have a complexity of $L_{p^n}(1/3, (64/9)^{1/3})$, and advantageous when p has a special form.

The TNFS and exTNFS algorithms are relatively young index calculus algorithms for which, there is therefore few hindsight on these algorithms. Indeed, there exists no implementation of these two algorithms to perform practical records. In this chapter, we try to propose a short state of the art of (ex)TNFS and some practical challenges, especially for the polynomial selection and relation collection. We skip the linear algebra step, since dealing with a large number of Schirokauer maps seems under control. We begin by presenting TNFS as an introduction of exTNFS.

5.1 Prelimiaries: the tower NFS algorithm

The TNFS algorithm uses a different representation of the target field $\mathbb{F}_{p^n}$ from the one in the classical NFS algorithm. In NFS, the field $\mathbb{F}_p$ is represented as $\mathbb{Z}/p\mathbb{Z}$, and $\mathbb{F}_{p^n}$ as $\mathbb{F}_p[x]/\varphi(x)$ where φ is a polynomial of degree n over $\mathbb{F}_p$. With TNFS, the field $\mathbb{F}_{p^n}$ is viewed as R/pR, where R is the quotient ring $\mathbb{Z}[t]/h(t)$ and h is a polynomial of degree n irreducible over $\mathbb{F}_p$.

Let first consider the tower of number fields, as represented in Figure 5.1. Let ι be a root of h and let $\mathbb{Q}(\iota)$ be the number field defined by h as $\mathbb{Q}[t]/h(t)$. The polynomial h is irreducible modulo p, there exists therefore a unique ideal $\mathfrak{p}$ over p in $\mathbb{Q}(\iota)$. Let f_0 and f_1 be two irreducible polynomials over R sharing

a same root m modulo $\mathfrak{p}$ in R. Let K_0 (respectively K_1) be the number field defined by $\mathbb{Q}(\iota)[x]/f_0(x) = \mathbb{Q}(\iota, \theta_0)$ (respectively $\mathbb{Q}(\iota)[x]/f_1(x) = \mathbb{Q}(\iota, \theta_1)$).

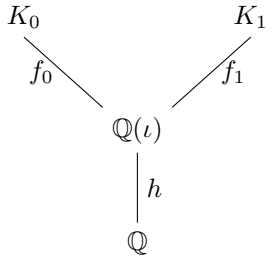


Figure 5.1 – Tower of number fields.

The conditions on h, f_0 and f_1 impose that there exist a ring homomorphism from $R[x] = \mathbb{Z}[\iota][x]$ to $\mathbb{F}_{p^n}$ involving K_0 and an other involving K_1. This allows us to build a commutative diagram, as for the previous variant of NFS, as depicted in Figure 5.2.

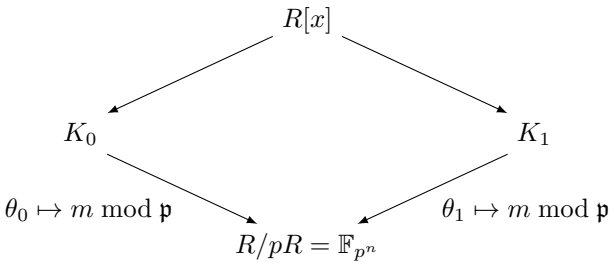


Figure 5.2 – The TNFS diagram to compute discrete logarithms in $\mathbb{F}_{p^n}$.

As in the classical NFS algorithm, it is sufficient to use polynomials a in $R[x]$ of degree 1 in x [22]. In this context, the polynomial a is defined as $a = a_0(t) + a_1(t)x$, where the polynomials a_0 and a_1 are of degree $n-1$ over $\mathbb{Z}$. The total number of coefficients of a polynomial a is therefore $2n$: to define the bounds on the coefficients, we need a $(2n)$-searching space $\mathcal{S}$. To obtain relations, we test the smoothness as ideals of a mapped in K_0 and K_1: if it is doubly smooth, the polynomial a gives a relation. The norm of a in K_0 is given by $\mathrm{Res}_t(\mathrm{Res}_x(a, f_0), h)$ and the same things occurs on side 1. Let $F_0(a_0, a_1)$ the homogenization of f_0 such that $F_0(a_0, a_1) = f_0(-a_0/a_1)a_1^{\deg f_0}$. Because a has degree 1, the norm of a is equal to $\mathrm{Res}_t(F_0(a_0, a_1), h)$, which is an integer. An upper bound for this norm is equal to $(\deg f_0 + 1)^{3n/2}(n+1)^{(3\deg f_0+1)n/2}\|a\|_\infty^{n\deg f_0}\|f_0\|_\infty^n\|h\|_\infty^{(n-1)\deg f_0}$. The relation collection will be performed quite the same way as in exTNFS case, we then refer to Section 5.4 for more details.

The individual logarithm step must be analyzed carefully, because lifting an element in one of the number fields can in some cases be non negligible, as it is the case in NFS.

Remark 5.1. The algorithm PiRaTh, from the first names of the authors, in [143, Figure 4] corresponds to TNFS.

5.1.1 Polynomial selections

The polynomial h is less constrained than the polynomial f_0 and f_1: it must only have degree n and be irreducible modulo p. As remarked during the computation of an upper bound of the norms, its infinity norm is raised to a potentially large power: we therefore look for a polynomial h with small coefficients. If we can enforce a Galois action of order n in h, the coefficients of h can be taken a little bit larger, hopping that the practical gain during the relation collection will largely counterbalance the fact that the norms are larger: one can save a factor of n in the relation collection.

The authors of [22] remark that their is no gain by taking the coefficients of f_0 and f_1 in the whole R and it suffices to consider the polynomial over $\mathbb{Z} \subset R$. Let the common root m of f_0 and f_1 be therefore in $\mathbb{Z}$. A way to find the polynomials f_0 and f_1 is to perform the polynomial selection described for prime field, see Section 3.1. We just recall in Table 5.1 the shape of the polynomial selections, where d_m and d_{JL} are positive integers.

Variant	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$
Base-m	1	$p^{1/(d_m+1)}$	d_m	$p^{1/(d_m+1)}$
Joux–Lercier	$d_{\text{JL}}+1$	small	d_{JL}	$p^{1/(d_{\text{JL}}+1)}$

Table 5.1 – Polynomial selection for TNFS in $\mathbb{F}_{p^n}$.

5.1.2 Individual logarithm

In this section, we want to compute the discrete logarithm of $T = T_0 + T_1 t + \cdots + T_{n-1} t^{n-1}$ an element of $\mathbb{F}_{p^n}$, where the T_i are large element of $\mathbb{F}_p$. The goal is that the individual logarithm step is kept negligible during the complexity analysis. We describe here the booting step of the individual logarithm, since the special-$\mathfrak{Q}$-descent is under control. We show that using the Joux–Lercier polynomial selection allows, as for the base-m polynomial selection analyzed in the original article [22], to have a negligible cost for the individual logarithm computation.

Using the base-m polynomial selection

As in NFS, we lift T on the rational side, that is the side 0. Let $\mathfrak{t}$ the ideal above T in $\mathbb{Q}(\iota)$, that is $\mathfrak{t} = (T, h \bmod T)$. The ideal above $\mathfrak{t}$ of K_0 is equal to $\mathfrak{T} = (\mathfrak{t}, m_0 + m_1 x \bmod \mathfrak{t})$, if $f_0(x,t)$ is equal to $m_0 + m_1 x$. The norm of the ideal $\mathfrak{T}$ in K_0 is bounded essentially by $p^n p^{n/(d_m+1)} = p^{n(1+o(1))}$. Following [22, Section 3.5], this is sufficient to obtain the needed complexity.

Using the Joux–Lercier polynomial selection

Let the ideal $\mathfrak{P}$ of K_0 be defined as $(\mathfrak{p}, x-m) = ((p, h(t)), x-m)$. Let z, z_0, z_1 be three elements of K_0 such that $T \equiv z \equiv z_0/z_1 \bmod \mathfrak{P}$. Let d be equal to $d_{JL}+1$ and let z_i be equal to $(z_{i,0,0} + z_{i,0,1}t + \cdots + z_{i,0,n-1}t^{n-1}) + (z_{i,1,0} + z_{i,1,1}t + \cdots + z_{i,1,n-1}t^{n-1})x + \cdots + (z_{i,d-1,0} + z_{i,d-1,1}t + \cdots + z_{i,d-1,n-1}t^{n-1})x^{d-1}$. The coefficients of z_0 and z_1, listed as

$$(z_{0,0,0}, z_{0,0,1}, \ldots, z_{0,0,n-1}, z_{0,1,0}, z_{0,1,1}, \ldots, z_{0,1,n-1}, \ldots, z_{0,d-1,0}, z_{0,d-1,1}, \ldots, z_{0,d-1,n-1},$$
$$z_{1,0,0}, z_{1,0,1}, \ldots, z_{1,0,n-1}, z_{1,1,0}, z_{1,1,1}, \ldots, z_{1,1,n-1}, \ldots, z_{1,d-1,0}, z_{1,d-1,1}, \ldots, z_{1,d-1,n-1}),$$

are valid if they are given by a linear combination of vectors of the lattice for which a basis is given by the rows of the following $(2nd) \times (2nd)$ matrix, where I_n is the identity matrix of size $n \times n$, i an integer in $[0, n[$ and j an integer in $[0, d[$,

$$\left(\begin{array}{ccccc|c} pI_n & & & & & \\ -mI_n & I_n & & & & \\ -m^2 I_n & & I_n & & & \\ \vdots & & & \ddots & & \\ -m^{d-1} I_{n-1} & & & & I_{n-1} & \\ \hline \multicolumn{5}{c|}{t^i x^j T \bmod (p, h, x-m)} & I_{nd} \end{array}\right).$$

The infinity norm of the shortest vector is around $p^{n/(2nd)}$ following Theorem A.1, the infinity norm of z_0 and z_1 is therefore close to $p^{1/(2d)}$. The norm of z_0 and z_1 in K_0 is almost bounded by $p^{n/2}$: the product of the norm to be tested for smoothness is then close to p^n. We can therefore reach the expected complexity bound, which is $L_{p^n}(1/3, (64/9)^{1/3})$ for the general case.

5.1.3 The multiple and special TNFS algorithms

As for the classical NFS algorithm, there exist a multiple variant (MTNFS) and a special variant (STNFS) of TNFS. The complexity achieved by MTNFS is the same than the one achieved by MNFS in large characteristic [24], that is $L_{p^n}(1/3, ((92 + 26\sqrt{13})/27)^{1/3})$. The STNFS variant reach the same complexity as SNFS [76] on prime field, which is $L_p(1/3, (32/9)^{1/3})$, but the complexity covers the whole high characteristic finite fields with the complexity $L_{p^n}(1/3, (32/9)^{1/3})$.

5.2 General framework for exTNFS

The extended tower NFS algorithm can be viewed as a modification of TNFS. Indeed, in TNFS, we shift the extension not on the number fields, as in NFS, but directly on the polynomials a mapped in the number fields. Using exTNFS, the extension degree $n = \eta\kappa$ is divided in two parts: the part η is shifted on the polynomial a, and the part κ on the number fields.

Let us now be more precise. For simplicity, we consider that η and κ are coprime. We can represent $\mathbb{F}_{p^n}$ as $\mathbb{F}_{(p^\eta)^\kappa}$. Let h be an integer polynomial of degree η irreducible over $\mathbb{F}_p$ and ι be a root of h. We define $\mathbb{F}_{p^\eta}$ as R/pR, where R is the number field $\mathbb{Z}[t]/h(t)$. As in TNFS, we look for two ring homomorphisms from $R[x] = \mathbb{Z}[\iota][x]$ to $\mathbb{F}_{p^n}$ involving for one a number field K_0 defined by f_0 and for the other a number field K_1 defined by f_1, where f_0 and f_1 are two polynomials over R sharing a common irreducible factor φ of degree κ. With this setting, we can define $\mathbb{F}_{p^n} = \mathbb{F}_{(p^\eta)^\kappa} \approx (R/pR)[x]/\varphi(x)$. This provides then the following commutative diagram.

Remark 5.2. Examining Figure 5.3 allows us to recognize some classical diagrams:

- if $R = \mathbb{Z}$, the diagram corresponds to NFS
  - for non-prime fields if φ has degree larger than 1.

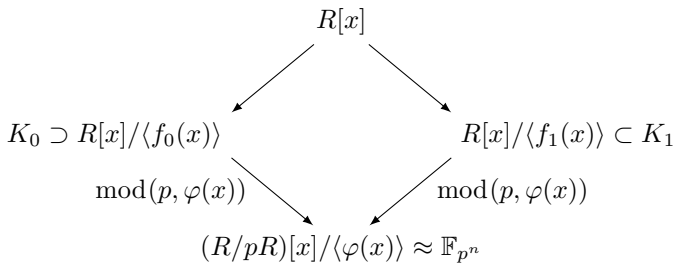


Figure 5.3 – The exTNFS diagram to compute discrete logarithms in $\mathbb{F}_{p^n}$.

  - for for prime fields if φ has degree 1.
- if $\varphi(x) = x - m$, this is the diagram for TNFS.

We now will give some details and list some challenges on the polynomial selection and the relation collection. We do not take care of the individual logarithm step, since it is under controlled thanks to the work of Guillevic [88, Section 4.2]: a particular case is studied in [187]. We do not detail how are defined the case of the multiple exTNFS (MexTNFS) algorithm and the case where p has a special form (SexTNFS): the reached complexity are, in the general case, the one in fields of large characteristic, that is $L_{p^n}(1/3, (32/9)^{1/3}$ for SexTNFS and $L_{p^n}(1/3, ((92+26\sqrt{13})/27)^{1/3}$ for MexTNFS. The end of this chapter will discuss about the cryptographic consequences of the new complexity reaches by exTNFS.

5.3 Polynomial selections

Instead of the classical NFS algorithm, we have three polynomials to select (we then use the term polynomial triple), instead of the usual two. Classically, the coefficients of h are chosen to be small.

5.3.1 Literature on polynomial selection for exTNFS

In all the works described in this section, the polynomial h is considered as fixed with small coefficients: no other condition is required. The goal is then to define f_0 and f_1, as in the classical NFS algorithm.

The polynomial selections available for exTNFS are basically variations of the ones proposed for the number field sieve algorithm for medium and large characteristic (see Section 4.1) as described by Kim and Barbulescu [114]. Sarkar and Singh propose extensions of the polynomial selection $\mathcal{A}$ [161, 158], called $\mathcal{B}$ and $\mathcal{C}$, allowing us to compute discrete logarithm in $\mathbb{F}_{p^n}$ for any composite n. For now on, the polynomials that defines K_0 and K_1 have coefficients in R and not only in $\mathbb{Z}$. Kim and Jeong propose a generalization of the classical JLSV_2 and conjugation polynomial selections [115] (called gJLSV and gConj) that reach a better complexity than the one obtained by Sarkar–Singh. Finally, Sarkar–Singh propose a new generalization of Kim–Jeong [159], the polynomial selection $\mathcal{D}$, that improves the complexity found by Kim and Jeong in some cases. For simplicity, we can keep in mind that, when n is composite and not a

prime power, the general complexity is in $L_{p^n}(1/3, (64/9)^{1/3})$. A more detailed description of these works can be found in [134, Section 4].

The shape of the polynomials f_0 and f_1 defined in all the previous articles are summarized in Table 5.2, where d_5 (respectively d_6 and d_9) is a factor of κ and r_5 (respectively r_6 and r_9) is larger or equal to κ/d_5 (respectively κ/d_6 and κ/d_9).

	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$
JLSV$_1$ (in $\mathbb{Z}$)	κ	$p^{1/2}$	κ	$p^{1/2}$
JLSV$_2$ (in $\mathbb{Z}$)	κ	$p^{\kappa/(d_2+1)}$	$d_2 \geq \kappa$	$p^{\kappa/(d_3+1)}$
GJL (in $\mathbb{Z}$)	$d_3 \geq \kappa$	$p^{\kappa/(d_3+1)}$	d_3+1	small
Conj (in $\mathbb{Z}$)	κ	$p^{1/2}$	2κ	small
$\mathcal{B}$ (in $\mathbb{Z}$)	$d_5 r_5 \geq \kappa$	$p^{n/(d_5(r_5+1))}$	$d_5(r_5+1)$	small
$\mathcal{C}$ (in R)	$d_6 r_6 \geq \kappa$	$p^{(r_6(\eta+1)+\kappa/d_6)/(r_6\eta+1)}$	$d_6(r_6+1)$	small
gJLSV (in R)	κ	$p^{\kappa/(d_7+1)}$	$d_7 \geq \kappa$	$p^{\kappa/(d_7+1)}$
gConj (in R)	κ	$p^{1/2}$	2κ	small
$\mathcal{D}$ (in R)	$d_9 r_9 \geq \kappa$	$p^{\kappa/(d_9(r_9+1))}$	$d_9(r_9+1)$	small

Table 5.2 – Polynomial selections for exTNFS in $\mathbb{F}_{p^n}$, where d_5, d_6 and d_9 divide κ and $r_i \geq \kappa/d_i$.

Remark 5.3. In the $\mathcal{C}$ polynomial selection, the value η can theoretically be replaced by any value in $[1, \eta]$: it is the parameter λ in the origin article.

If dealing with polynomials f_0 and f_1 with coefficients over R seems theoretically promising, dealing with integer polynomials f_0 and f_1 as proposed in the article that describes first exTNFS is yet a challenge, because the quality criteria, that is the equivalent of the α and Murphy-E functions, are not described.

5.3.2 Quality criteria

In this section, we will describe some of the available choices for the polynomial selection step, our practical experiments and the challenges we need to solve to select the best polynomial triple: we focus on integer polynomials h, f_0 and f_1.

Galois actions

As in the classical NFS variant for the medium characteristic, we can hope to have an important speed-up by using Galois actions. A Galois action of order $k_0 \leq \eta$ can be enforced in the polynomial h and a common Galois action of order $k_1 \leq \kappa$ can be shared by f_0 and f_1. This allows us to emulate a Galois action in the classical NFS algorithm of order $k_0 k_1$. The particular case of $h = t^2 + 1$, which have a Galois action of order $k_0 = 2$, is detailed in [22, Section 7.1].

Practical experiments

We extend the implementation we did of the three-dimensional relation collection, described in Chapter 7, to propose an implementation of exTNFS with a four-dimensional relation collection. If the norms of the elements for a 389-bit

$\mathbb{F}_{p^6}$ seems to have the same size than the one for a 300-bit $\mathbb{F}_{p^6}$ using the classical NFS algorithm, the smoothness of the elements are experimentally much worse. This is probably due to a not-so-good polynomial triple: indeed, the size of the norms is mainly dependent of the size of the coefficient, but not on the quality of a triple.

It is therefore necessary to explicit the quality criteria for the exTNFS case. Considering only integer polynomials triples, the difficulty consists in distinguishing good triples, because the generation itself is the same than for the classical NFS. We describe first a workaround to distinguish such triples.

A fake α-function

Since an α-function is not available yet in the literature for exTNFS, we try to simulate one, and call it β-function. Our goal is not to estimate how many bits in base e we gain by using a given triple, but just to distinguish between two triples the most promising one. We will show in the following that there is a degree 1 ideal $\mathfrak{Q}$ of norm q in K_0 (respectively K_1) if h have a root modulo q and if f_0 (respectively f_1) have a root modulo q.

Therefore, one can choose an irreducible integer polynomial h of small coefficients having sufficiently many roots modulo small primes (and possibly having a Galois action of maximal order). The list of such small primes is denoted by L and we define our β-function as $\beta(f) = \sum_{\ell \in L} \alpha_\ell(f)$, where ℓ is a prime in the list L and α_ℓ is defined either as in Section 3.1.1 or in Section 4.1.1.

Remark 5.4. The use of the quantity $\alpha_\ell(f)$ is nonsense but is a simple way to take into account the contribution of the small primes.

Challenge: precise quality criteria

The β-function is obviously not elegant, but we believe that, in the absence of a true α-function, the β-function allows us to select not so bad polynomial triples. It seems however feasible to have a well defined α-function in particular cases, especially in the case $h = t^2 + 1$. One of the difficulties is to find the equivalent notion of the irreducibility of a in the definition of α_ℓ to avoid to consider duplicated polynomials a.

Once the α-function is defined, we can hope to find a formula to compute the equivalent of the Murphy-E quantity. If there exists an equivalent of the Fibonacci sphere in dimension higher than 3, it seems therefore not so difficult to compute this quantity.

5.4 Relation collection

As for the polynomial selection, we consider only polynomial triples defined over the integers. We will show briefly how we can use the special-$\mathfrak{Q}$ method, as in the classical NFS algorithm, the remaining task being to understand how we can define a relation in exTNFS.

5.4.1 Defining the ideals

The complexity of exTNFS shows, as the one about TNFS, that the degree in the variable x of the polynomial a can be taken equal to 1. We then can

write $a(x)$ as $a_0(t) + a_1(t)x$, where a_0 and a_1 are two polynomials of degree $\eta - 1$ in $\mathbb{Z}[t]$. The factorization of a in prime ideals in K_0 involves ideals $\mathfrak{R}$ which can be represented as $(\mathfrak{r}, x - \rho(t))$, where $\mathfrak{r}$ is a prime ideal in the number field $\mathbb{Z}[t]/h(t)$ and $\rho(t)$ a root of f modulo $\mathfrak{r}$. The prime ideal $\mathfrak{r}$ is written as $(r, h_r(t))$, with r the norm of $\mathfrak{r}$ and h_r a polynomial of degree d which divides h modulo r. The lattice of polynomials a involving $\mathfrak{R}$ in its ideal factorization is generated by $\{r, rt, \ldots, rt^{d-1}, h_r(t), th_r(t), \ldots, t^{\eta-d-1}h_r(t), x - \rho(t), t\,(x - \rho(t)), \ldots, t^{\eta-1}\,(x - \rho(t))\}$. We denote by $M_{\mathfrak{R}}$ the matrix whose rows are the vector of this basis. We can define as well ideals of larger degree, but as ideals of inertia degree 1 are more numerous (the same apply for ideal $\mathfrak{r}$ of inertia degree $d = 1$), we only deal with them.

5.4.2 Relation

A relation in exTNFS is given by a polynomial $a(x, t) = a_0(t)+a_1(t)x$. The norm of this polynomial mapped into K_0 (respectively K_1), is, as in TNFS, equal to $\mathrm{Res}_t(\mathrm{Res}_x(a(x, t), f_0(x, t)), h(t))$ (respectively $\mathrm{Res}_t(\mathrm{Res}_x(a(x, t), f_1(x, t)), h(t))$). Let us consider the mapping into K_0. As in TNFS, this resultant can be rewritten as $\mathrm{Res}(f_0(-a_1/a_0)a_0^{\deg f_0}, h)$. The quantity is upper bounded by $(\deg f_0 + 1)^{3\eta/2}(\eta + 1)^{(3\deg f_0+1)\eta/2}\|a\|_\infty^{\eta \deg f_0}\|f_0\|_\infty^{\eta}\|h\|_\infty^{\deg f_0(\eta-1)}$.

As in Section 5.3.2 for the definition of α, a problem during the relation collection is the definition of the polynomials a that give relations. In the classical NFS algorithm, the polynomials a must verify:

- a is irreducible over $\mathbb{Z}$,
- the leading coefficient of a is positive.

The polynomials a in exTNFS can be described by 2η coefficients, half of them describing $a_0(t) = a_{0,0}+a_{0,1}t+\ldots+a_{0,\eta-1}t^{\eta-1}$ and the other half $a_1(t) = a_{1,0}+a_{1,1}t+\ldots+a_{1,\eta-1}t^{\eta-1}$. The coefficients $a_{1,\eta-1}$ is forced to be positive, to avoid to deal with a relation and its opposite, which is a translation of the second condition. But, the translation of the first condition is not obvious: morally, we look for polynomials a irreducible over R. The conditions to be checked are not well defined, and the irreducibility over R is maybe not the only condition to give a relation.

5.4.3 Dividing the search space

Let $\mathfrak{Q}$ be an ideal of K_0, $M_{\mathfrak{Q}}$ be a $2\eta \times 2\eta$ matrix whose rows are vectors of a basis of a $\mathfrak{Q}$-lattice. The coefficients of the polynomial a whose norm in K_0 is divisible by $\mathfrak{Q}$ is given by $\boldsymbol{a} = \boldsymbol{c}M_{\mathfrak{Q}}$, with $\boldsymbol{c}$ in $\mathbb{Z}^{2\eta}$. In this $\mathfrak{Q}$-lattice, we want to enumerate the polynomials a divisible by ideals $\mathfrak{R}$.

Let consider an $\mathfrak{R}$-lattice $\Lambda_{\mathfrak{R}}$. Modulo r, the first basis vector of $\Lambda_{\mathfrak{R}}$ is therefore equals to $\mathbf{0}$, its rank become $2\eta - 1$. The coefficients of a polynomial a involving the ideal $\mathfrak{R}$ in its ideal factorization can be generated by a linear equation modulo r. This relation can be written as $\boldsymbol{a}U_{\mathfrak{R}} \equiv \mathbf{0} \bmod r$, with $U_{\mathfrak{R}}$ a $2\eta \times 1$ matrix. In the $\mathfrak{Q}$-lattice, $\boldsymbol{a} = \boldsymbol{c}M_{\mathfrak{Q}}$, and combining this relation with the one in the $\mathfrak{R}$-lattice, we obtain $\boldsymbol{c}M_{\mathfrak{Q}}U_{\mathfrak{R}} \equiv 0 \bmod r$. The vectors $\boldsymbol{c} \in \mathbb{Z}^{2\eta}$ verifying this relation are element of a lattice, which a basis is formed by the rows of the matrix $M_{\mathfrak{Q}\mathfrak{R}}$, which can be written as, $\alpha_{\{0,1,\ldots,t-2\}}$ in $\mathbb{Z}/r\mathbb{Z}$,

$$M_{\mathfrak{Q}\mathfrak{R}} = \begin{pmatrix} \boldsymbol{b_0} \\ \boldsymbol{b_1} \\ \vdots \\ \vdots \\ \boldsymbol{b_{t-1}} \end{pmatrix} = \begin{pmatrix} r & 0 & 0 & \cdots & 0 \\ \alpha_0 & 1 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \alpha_{t-2} & 0 & \cdots & 0 & 1 \end{pmatrix}. \tag{5.1}$$

Challenges

There remain therefore two main challenges to run the relation collection:

- define the conditions on the polynomials a to be a valid relation,
- enumerate the elements of the intersection of a sieving region and the lattice generated by $M_{\mathfrak{Q}\mathfrak{R}}$.

The first challenge is quite the same for the polynomial selection to define the α quantity. To solve the second challenge, we recall that the relation collection can be divided in three parts: initialization of the norms, sieving and cofactorization. The cofactorization is about the factorization of integers and the provenance of this integers is not taken into account. It is therefore not a problem for exTNFS: cofactoring with ECM chains is described in Section 7.4.1. The situation is less clear for the initialization of the norms. In Section 7.1, we describe a general algorithm to initialize the norms, but accuracy and running time are not guaranteed for dimensions larger than 4. The matrix in Equation (5.1) as the same form as in Equation (4.7). In Chapter 6, we will describe and analyze sieve algorithms to enumerate elements of such lattices.

Therefore, the second challenge has some solution, at least in dimension 4. The first challenge remains.

5.5 Cryptographic consequences

A quick look at the cryptosystems whose security relies of the hardness of computing discrete logarithms in medium characteristic finite fields shows that:

- XTR [129] is defined over $\mathbb{F}_{p^6}$,
- pairings using BN [26] curves and BLS12 [25] curves are defined over $\mathbb{F}_{p^{12}}$,
- pairings using KSS [111] curves are defined over $\mathbb{F}_{p^{18}}$,
- pairings using BLS24 [25] curves are defined over $\mathbb{F}_{p^{24}}$.

We can observe that the extension degree is always composite and some of the systems were proposed before 2006.

Before 2006 and the article of Joux, Lercier, Smart and Vercauteren [106], the best complexity to compute discrete logarithms in medium characteristic was in $L(1/2)$ [79]. The complexity of the $L_{p^n}(1/3, c^{1/3})$ algorithms was improved, from $c = 128/9$ in 2006 to $c = 64/9$ today. It is obvious that the parameters designed for cryptosystems before 2014 to reach a given security level need to be updated.

This is what Menezes, Sarkar and Singh did in [134], and more recently Barbulescu and Duquesne [18]. However, even if these articles try to be the closest possible to what it can be expected empirically, the validation of these theoretical results will have to be confirmed by a complete implementation of exTNFS, and some of the challenges that are listed in this chapter should be solved to run these practical experiments. For example, it is assumed that a relation collection in dimension 24 or 36 exist, which is not currently the case. In the next chapter, we propose algorithms to sieve in small dimensions.

Chapter 6

Sieving for the number field sieve algorithms

The relation collection of the NFS algorithms can be performed efficiently with a sieving strategy, a result known since the use of the quadratic sieve of Pomerance to factorize large integers [148]. Instead of the classical NFS algorithm where the relation collection involve polynomials of degree one (dimension two), the relation collection in the context of the medium characteristic must be done with polynomial of degree higher than one, see Chapter 3 and Chapter 4. If sieving in dimension two is well described in the literature, sieving in higher dimension received significantly less attention.

In this chapter, we will present efficient algorithms to sieve in dimension higher or equal to 2. We will begin with a short remainder about the line sieve and the sieve of Franke–Kleinjung, an efficient way to sieve in two dimensions when the line sieve becomes inefficient. We then describe a general algorithm to sieve in any small dimension, with a specialization to the 3-dimensional case.

Let Λ be a full-rank lattice of dimension t. Let $\mathcal{H}$ be the t-sieving region equal to $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\cdots\times[H_{t-1}^m, H_{t-1}^M[$, following Definition 4.1: we recall that H_k^m are negative and H_k^M are positive. The length of an interval $[H_k^m, H_k^M[$ is denoted by the integer I_k. We recall that the goal of the sieving step is, given the lattice Λ and the sieving region $\mathcal{H}$, to find all the elements of Λ that lie in $\mathcal{H}$. In the following, we consider that a basis of Λ is of the form

$$\mathcal{B} = \{(r, 0, 0, \ldots, 0), (\lambda_0, 1, 0, 0 \ldots, 0), \ldots, (\lambda_{t-1}, 0, 0, \ldots, 0, 1)\},$$

where r is a prime and the λ_i are non-negative and less than r. The vectors of the basis $\mathcal{B}$ are denoted by $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$. The special form of this basis comes from the basis of a degree 1 prime ideal in a special-$\mathfrak{Q}$-lattice, as explained in Chapter 4 and Chapter 5.

We believe that our description of the general algorithms of this chapter can be adapted to other forms of the basis like the Hermite normal form, but we do not discuss it further.

For an integer k in $[0, t[$, we define the extended sieving region $\mathcal{H}_k$ as $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\ldots\times[H_k^m, H_k^M[\times\mathbb{Z}^{t-(k+1)}$: with this definition, the sieving

region $\mathcal{H}$ is equal to $\mathcal{H}_{t-1}$. This extended sieving region will be used to define the three sorts of vectors we introduce to describe our sieve algorithms.

Remark 6.1. Let ℓ be less than t. The expected number of elements in the intersection of the lattice Λ of volume r and a cuboid $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\cdots\times[H_\ell^m, H_\ell^M[\times\{c_{\ell+1}\}\times\{c_{\ell+2}\}\times\cdots\times\{c_{t-1}\}$, where $(c_{\ell+1}, c_{\ell+2}, \ldots, c_{t-1})$ are in $\mathbb{Z}^{t-(\ell+1)}$, is close to $I_0 I_1 \cdots I_\ell / r$.

6.1 Transition-vectors

A key notion for all our algorithms is the *transition-vectors*, that generalize the vectors introduced in the Franke–Kleinjung algorithm (see Section 6.2.2) which are 1–transition-vectors.

Definition 6.1. Let k be in $[0, t[$. A *k–transition-vector* is an element $\boldsymbol{v} \neq \mathbf{0}$ of Λ such that there exist $\boldsymbol{c}$ and $\boldsymbol{c_n}$ in the intersection of the lattice Λ and the extended sieving region $\mathcal{H}_k$, with $\boldsymbol{c_n} = \boldsymbol{c} + \boldsymbol{v}$ such that the $t-1-k$ last coordinates of $\boldsymbol{c}$ and $\boldsymbol{c_n}$ are equal and the coordinate $\boldsymbol{c_n}[k]$ must be the smallest possible larger than $\boldsymbol{c}[k]$.

In other words, a k–transition-vector allows to jump from one vector in the intersection of Λ and $\mathcal{H}$ to another one with a different coordinate k, without missing any vectors. A set of k–transition-vectors is *complete* if it contains all the possible k–transition-vectors. Given an algorithm $\mathcal{E}$ that uses transition-vectors to perform the enumeration of the elements in the intersection of Λ and $\mathcal{H}$, a group of i sets of k–transition-vectors are *$(i-1)$-suitable* if they allow to reach all the elements in the intersection of the form $(\cdot, \cdot, \ldots, \cdot, c_{i+1}, c_{i+2}, \ldots, c_{t-1})$, where k is in $[0, i[$, i in $[0, t[$ and $(c_{i+1}, c_{i+2}, \ldots, c_{t-1})$ in $[H_{i+1}^m, H_{i+1}^M[\times[H_{i+2}^m, H_{i+2}^M[\times\cdots\times[H_{t-1}^m, H_{t-1}^M[$. Note that is is quite impossible to know, given a group of i sets of transition-vectors without knowledge on how they were produced, if a set is complete or if the i sets are $(i-1)$-suitable without performing the generalized line sieve, described later in Section 6.4.4: this is mainly due to the fact that it is impossible to determine if an element of a lattice is a transition-vector without performing the generalized line sieve, because of the condition on the coordinate $\boldsymbol{c_n}[k]$ in Definition 6.1 that requires there does not exist an element $\boldsymbol{c'}$ with a coordinate k between the one of $\boldsymbol{c}$ and the one of $\boldsymbol{c_n}$.

Remark 6.2. The shape of the lattices we consider imposes that, if there exists a 0–transition-vector $\boldsymbol{v}$, then $\boldsymbol{v}$ is the only element of the set of 0–transition-vectors.

6.2 Reminders in 2 dimensions

In this section, we set $t = 2$, implying that the basis $\mathcal{B} = \{\boldsymbol{b_0}, \boldsymbol{b_1}\}$ is equal to $\{(r, 0), (\lambda_0, 1)\}$. We call line a 1-dimensional subset of the lattice Λ parallel to the abscissa axis and plane the whole elements of Λ. We are looking for elements that are in the intersection of Λ and the sieving region $\mathcal{H}$, which is equal in this context to $[H_0^m, H_0^M[\times[H_1^m, H_1^M[$ (classically, the value H_1^m is set to 0). The line sieve, becomes inefficient when there is less than one element per line ($r > I_0$ following Remark 6.1). This is why the lattice sieve, which is the

sieve of Franke and Kleinjung, is used, an other algorithm to quickly enumerate the elements in the plane.

6.2.1 Line sieve

In this first section, we give anew a description of the line sieve, already described in Section 3.2.2, in order to show how we can rewrite this algorithm to fit into the general description of our algorithms. During the line sieve, the expected number of elements per line is greater than 1. Sieving in a line is performed by a procedure similar to the sieve of Eratosthenes. To begin this procedure, one needs to find a starting point in the line. An element (c_0, c_1) of the lattice is the linear combination of the two basis vectors and can therefore be written as $(c_0, c_1) = e_0\boldsymbol{b_0} + e_1\boldsymbol{b_1}$. Given the ordinate e_1 in $[H_1^m, H_1^M[$, a possible starting point is found if its abscissa $e_0 r + e_1\lambda_0$, where $e_0 = \lceil (H_0^m - e_1\lambda_0)/r \rceil$, is less than H_0^M: this starting point is the one with the smallest possible abscissa in the line that fit into the sieving region. To enumerate the other elements of the line, we add to this starting point multiples of $\boldsymbol{b_0}$: $\boldsymbol{b_0}$ is indeed the 0–transition-vector. An algorithm to perform the line sieve is the following:

1. Set e_1 to H_1^m.
2. While $e_1 < H_1^M$
   (a) Find the element (c_0, c_1) of the lattice with the smallest possible abscissa.
   (b) Enumerate the elements of the line by adding multiple of $\boldsymbol{b_0}$ to this starting point until their abscissa becomes larger than H_0^M.
   (c) Increment e_1.

This algorithm spends a lot of time by finding a starting point because of the divisions and ceilings. A way to improve efficiency is to use the information given by a previous line to find the starting point of the following line. Indeed, from an element $e_0\boldsymbol{b_0} + e_1\boldsymbol{b_1}$, if $(e_1 + 1) < H_1^M$, a valid element of the lattice is given by $e_0\boldsymbol{b_0} + (e_1 + 1)\boldsymbol{b_1}$. If this point is not in the sieving region, we must subtract $\boldsymbol{b_0}$.

If the vectors $\boldsymbol{b_1} + k\boldsymbol{b_0}$, where k is an integer, fit in the sieving region $\mathcal{H}$, they are therefore 1–transition-vectors: given the following algorithm, the set of 1–transition-vectors $\{\boldsymbol{b_1}, \boldsymbol{b_1} - \boldsymbol{b_0}\}$ and the 0–transition-vector are 1-suitable.

1. Set $\boldsymbol{c}$ to $\boldsymbol{0}$.
2. While $\boldsymbol{c}[1] < H_1^M$
   (a) Enumerate the elements of the line by adding positive or negative multiples of $\boldsymbol{b_0}$ to $\boldsymbol{c}$.
   (b) Set $\boldsymbol{c}$ to $\boldsymbol{c} + \boldsymbol{b_1}$ and subtract $\boldsymbol{b_0}$ if $\boldsymbol{c}$ does not fit in $\mathcal{H}$.
3. Set $\boldsymbol{c}$ to $-\boldsymbol{b_1}$.
4. While $\boldsymbol{c}[1] \geq H_1^m$
   (a) Enumerate the elements of the line by adding positive or negative multiples of $\boldsymbol{b_0}$ to $\boldsymbol{c}$.

(b) Set $\boldsymbol{c}$ to $\boldsymbol{c} - \boldsymbol{b_1}$ and add $\boldsymbol{b_0}$ if $\boldsymbol{c}$ does not fit in $\mathcal{H}$.

Even if H_1^m is classically set to 0, we describe the case of negative ordinate for completeness in Item 4 and in order to correspond to the generic algorithm in the following. The complete pseudo-code is given in Appendix E.1. Item 3 of the previous algorithm sets $\boldsymbol{c}$ to $-\boldsymbol{b_1}$ instead of $\mathbf{0}$ to avoid to sieve again the line $(\cdot, 0)$.

If the volume of Λ becomes larger than I_0, the average number of elements per line is less than 1. Sieving in a line becomes expensive because if there exists a point in the line, it is the only one in the line, and the cost of discovering one element is the same as the one to discover no element.

6.2.2 Lattice sieve

In this section, we assume that λ_0 is a non-zero coefficient. Otherwise, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}\}$ of the lattice is orthogonal and the elements to be sieved are of the form $(0, e_1)$, with e_1 in $[H_1^m, H_1^M[$, which can be processed specifically and efficiently.

The lattice sieve is used when there is less than one element per line. We can therefore sort these elements by their increasing c_1-coordinate. Furthermore, there exist no 0–transition-vector. If we perform a line sieve, as presented in the previous section, or the sieve by vector briefly described in Section 3.2.2, and sort the elements found by increasing c_1 coordinate, we can observe that the set of 1–transition-vectors is composed of at most three vectors, as illustrated in Figure 6.1.

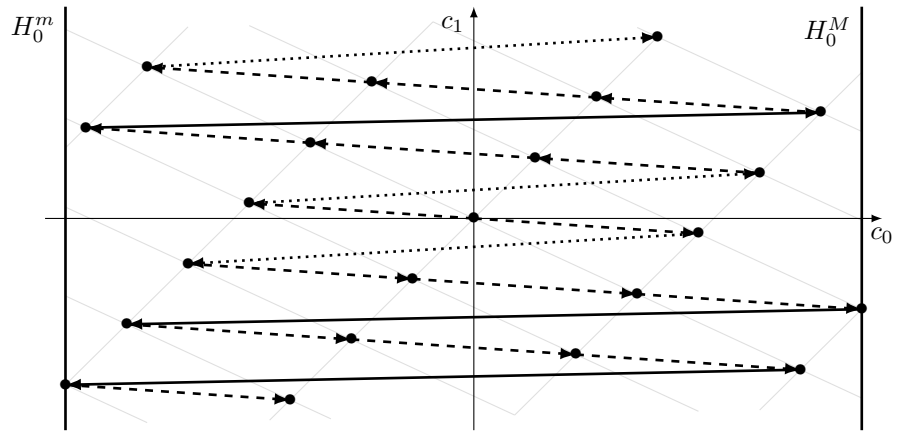


Figure 6.1 – Three 1–transition-vectors.

Preliminaries

Franke and Kleinjung proved in [61] that, given a lattice Λ of basis $\mathcal{B}$ and a sieving region $\mathcal{H}$, there exists a basis $\{\boldsymbol{u}, \boldsymbol{v}\}$ of Λ, well adapted to perform efficiently the enumeration of the elements of Λ that are in $\mathcal{H}$. The basis $\{\boldsymbol{u}, \boldsymbol{v}\}$ is described in Proposition 6.1. The only three possible 1–transition-vectors are $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{u} + \boldsymbol{v}$, as shown in Proposition 6.2 and Corollary 6.1, and the enumeration can be easily managed, as described at the end of this section.

Proposition 6.1 ([61, Proposition 1])**.** *Let $\mathcal{H}$ be a sieving region, Λ be a lattice of basis $\mathcal{B}$ and volume $r \geq I_0$. There exists a unique basis $\{\boldsymbol{u}, \boldsymbol{v}\}$ of Λ such that:*

- *the coordinates $\boldsymbol{u}[1]$ and $\boldsymbol{v}[1]$ are positive;*
- *the coordinates $\boldsymbol{u}[0]$ and $\boldsymbol{v}[0]$ verify $-I_0 < \boldsymbol{u}[0] \leq 0 \leq \boldsymbol{v}[0] < I_0$ and $\boldsymbol{v}[0] - \boldsymbol{u}[0] \geq I_0$.*

Proposition 6.2 ([61, Proposition 2])**.** *Let $\mathcal{H}$, Λ, $\boldsymbol{u}$ and $\boldsymbol{v}$ be as in the previous proposition. Let (c_0, c_1) be in the intersection of Λ and $\mathcal{H}_0$. Then the element of this intersection with the smallest ordinate larger than c_1 is obtained by adding to (c_0, c_1) the following vector:*

$$\begin{cases} \boldsymbol{u} & \text{if } c_0 \geq H_0^m - \boldsymbol{u}[0]. \\ \boldsymbol{v} & \text{if } c_0 < H_0^M - \boldsymbol{v}[0]. \\ \boldsymbol{u} + \boldsymbol{v} & \text{if } H_0^M - \boldsymbol{v}[0] \leq c_0 < H_0^m - \boldsymbol{u}[0]. \end{cases}$$

Remark 6.3. In Figure 6.1, the vector $\boldsymbol{u}$ is dashed, the vector $\boldsymbol{u} + \boldsymbol{v}$ is dotted and the vector $\boldsymbol{v}$ is solid.

We can also go in decreasing order instead of increasing order with respect to the c_1-coordinate, which is useful if we did not start from the bottom end of the sieving region.

Corollary 6.1. *Let $\mathcal{H}$, Λ, $\boldsymbol{u}$ and $\boldsymbol{v}$ be as in the previous proposition. Let (c_0, c_1) be in the intersection of Λ and $\mathcal{H}_0$. Then the element of this intersection with the largest ordinate smaller than c_1 is obtained by subtracting from (c_0, c_1) the following vector:*

$$\begin{cases} \boldsymbol{u} & \text{if } c_0 < H_0^M + \boldsymbol{u}[0]. \\ \boldsymbol{v} & \text{if } c_0 \geq H_0^m + \boldsymbol{v}[0]. \\ \boldsymbol{u} + \boldsymbol{v} & \text{if } H_0^M + \boldsymbol{u}[0] \leq c_0 < H_0^m + \boldsymbol{v}[0]. \end{cases}$$

Proof. The proofs of Proposition 6.1 and Proposition 6.2 are given in [61].

The proof of Corollary 6.1 is derived from the Proposition 6.2. Let $\mathcal{H}$, Λ, $\boldsymbol{u}$ and $\boldsymbol{v}$ be as in Proposition 6.2. Let (c_0, c_1) be an element of the intersection of Λ and $[H_0^m, H_0^M[\times\mathbb{Z}$. Let (c_0', c_1') be the element resulting from the addition of $\boldsymbol{u}$ to (c_0, c_1). The condition on c_0 in Proposition 6.2, that is $H_0^m - \boldsymbol{u}[0] \leq c_0 < H_0^M$ is translated into the condition $H_0^m \leq c_0 + \boldsymbol{u}[0] < H_0^M + \boldsymbol{u}[0]$. The coordinate c_0' is equal to $c_0 + \boldsymbol{u}[0]$, that is what we claim for the first condition in Corollary 6.1. The same idea applies on $\boldsymbol{v}$ and $\boldsymbol{u} + \boldsymbol{v}$ to prove the result of Corollary 6.1. □

Enumeration *à la* Franke–Kleinjung

First, we need to compute the basis verifying the properties listed in Proposition 6.1. We can remark that the ordinates of $\boldsymbol{u}$ and $\boldsymbol{v}$ are relatively small and the abscissae are relatively large. More precisely, we mean that $\boldsymbol{u}[0]$ and $\boldsymbol{v}[0]$ are in $O(I_0)$ and $\boldsymbol{u}[1]$ and $\boldsymbol{v}[1]$ are in $O(r/I_0)$. We can therefore perform a weighted basis reduction with weight $w = (1/I_0, I_0/r)$ (or $w = (r, I_0^2)$) but with no guarantee of the correctness of the two output vectors: it can be necessary to do a small linear combination of these vectors to fit into the bounds given in Proposition 6.1. However, we can compute them efficiently and correctly by

applying a Gaussian reduction with stopping criteria that allows to reach the bounds given in Proposition 6.1, as shown in the proof of this proposition in [61], resulting in Function `reduce-qlattice`.

Function `reduce-qlattice`($\boldsymbol{b_0}$, $\boldsymbol{b_1}$, I_0).

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}\}$, the length I_0 of $[H_0^m, H_0^M[$
output: the reduced basis $\{\boldsymbol{u}, \boldsymbol{v}\}$ where $\boldsymbol{u}$ and $\boldsymbol{v}$ verify Proposition 6.1
$\boldsymbol{u} \leftarrow -\boldsymbol{b_0}$, $\boldsymbol{v} \leftarrow \boldsymbol{b_1}$;
while $\boldsymbol{v}[0] \geq I_0$ **do**
  $\boldsymbol{u} \leftarrow$ `reduce`$(\boldsymbol{u}, \boldsymbol{v})$; `// reduction of` $\boldsymbol{u}$ `by` $\boldsymbol{v}$
  if $\boldsymbol{u}[0] > -I_0$ **then** break;
  $\boldsymbol{v} \leftarrow$ `reduce`$(\boldsymbol{v}, \boldsymbol{u})$; `// reduction of` $\boldsymbol{v}$ `by` $\boldsymbol{u}$
  if $\boldsymbol{v}[0] < I_0$ **then** break;
  $\boldsymbol{u} \leftarrow$ `reduce`$(\boldsymbol{u}, \boldsymbol{v})$; `// reduction of` $\boldsymbol{u}$ `by` $\boldsymbol{v}$
  if $\boldsymbol{u}[0] > -I_0$ **then** break;
  $\boldsymbol{v} \leftarrow$ `reduce`$(\boldsymbol{v}, \boldsymbol{u})$; `// reduction of` $\boldsymbol{v}$ `by` $\boldsymbol{u}$
end
$k \leftarrow \boldsymbol{v}[0] - I_0 - \boldsymbol{u}[0]$;
if $\boldsymbol{v}[0] > -\boldsymbol{u}[0]$ **then** $\boldsymbol{v} \leftarrow \boldsymbol{v} - (\lfloor k/\boldsymbol{u}[0] \rfloor)\boldsymbol{u}$;
else $\boldsymbol{u} \leftarrow \boldsymbol{u} + (\lfloor k/\boldsymbol{v}[0] \rfloor)\boldsymbol{v}$;
return $(\boldsymbol{u}, \boldsymbol{v})$;

The reduce function in `reduce-qlattice` of $\boldsymbol{u}$ by $\boldsymbol{v}$ is the Euclidean operation that allows to have the absolute value of $\boldsymbol{u}[0]$ less than the absolute value of $\boldsymbol{v}[0]$ by removing the appropriate number of times $\boldsymbol{v}$ to $\boldsymbol{u}$. Finally, the sieving procedure is the following:

Basis reduction. Compute from $\boldsymbol{b_0}$ and $\boldsymbol{b_1}$ the two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ that reach the conditions of Proposition 6.1, with `reduce-qlattice`.

Enumerate positive ordinates. Let $\boldsymbol{c}$ be equal to $\boldsymbol{0}$. While $\boldsymbol{c}[1]$ is less than H_1^M

1. Report $\boldsymbol{c}$.
2. Following Proposition 6.2, add to $\boldsymbol{c}$ the 1–transition-vector corresponding to $\boldsymbol{c}[0]$.

Enumerate negative ordinates. Let $\boldsymbol{c}$ be equal to either $-\boldsymbol{u}$, or $-\boldsymbol{v}$ or $-(\boldsymbol{u}+\boldsymbol{v})$, according to which vector must be subtracted following Corollary 6.1 when the abscissa is zero. While $\boldsymbol{c}[1]$ is larger than H_1^m:

3. Report $\boldsymbol{c}$.
4. Following Corollary 6.1, subtract to $\boldsymbol{c}$ the 1–transition-vector corresponding to $\boldsymbol{c}[0]$.

The enumeration parts, concerning the positive and negative ordinates, can be depicted as in Figure 6.2. Depending on the abscissa of an element, we know which vector we need to add or subtract to continue the enumeration. The full pseudo-code of the lattice sieve is given in Algorithm E.2.

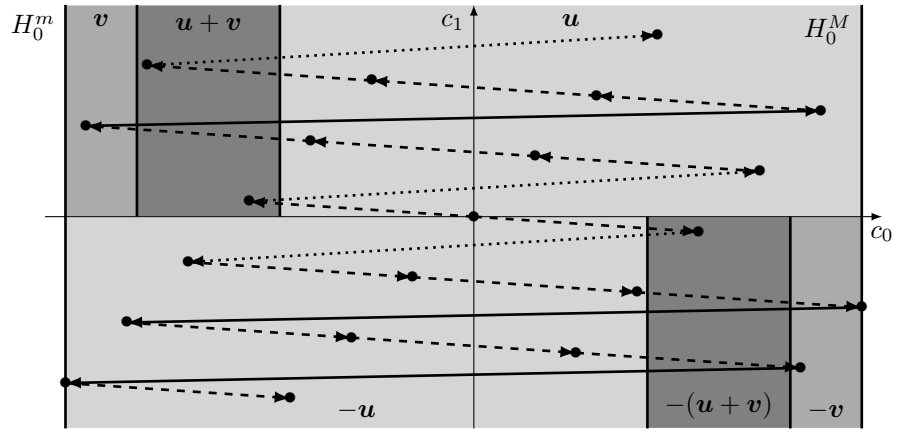


Figure 6.2 – Areas of elements for which the same 1–transition-vector is added.

6.2.3 Unification of the two sieves

To conclude this reminder, we propose a unification of the two sieves, namely the line sieve and the sieve of Franke–Kleinjung, assuming that λ_0 is a non-zero coefficient. Depending on the volume of the lattice Λ and the sieving region $\mathcal{H}$, we know how to build the possible transition-vectors:

- if the volume of Λ is less than I_0, the vector $\boldsymbol{b_0}$ is the 0–transition-vector and the 1–transition-vectors $\boldsymbol{b_1}$ and $\boldsymbol{b_1} - \boldsymbol{b_0}$ form two sets that are 1-suitable for the unified algorithm we propose in the following.
- if the volume of Λ is larger than I_0, there exists no 0–transition-vector and three 1–transition-vectors are computed using the output of Function `reduce-qlattice`.

Once the sets of 0–transition-vector and 1–transition-vectors are constituted, the enumeration of the elements in the intersection of Λ and $\mathcal{H}$ is simple: given an element $\boldsymbol{c}$ in this intersection, we try to find all the element in the line, that is add to $\boldsymbol{c}$ all the multiple of the 0–transition-vector, add a 1–transition-vector and apply on the new vector the two previous steps. The algorithm looks like:

Initialization.

1. Given Λ and $\mathcal{H}$, compute the sets of 0–transition-vector and 1–transition-vectors. If the 0–transition-vector exists, we call it $\boldsymbol{v}$, otherwise, the operations involving $\boldsymbol{v}$ must be skipped.
2. Set $\boldsymbol{c}$ to $\mathbf{0}$.

Enumeration.

3. While $\boldsymbol{c}[1] < H_1^M$:
   (a) Set $\boldsymbol{d}$ to $\boldsymbol{c}$.
   (b) While $\boldsymbol{c}[0] < H_0^M$, report $\boldsymbol{c}$ and add $\boldsymbol{v}$ to $\boldsymbol{c}$.
   (c) Set $\boldsymbol{d}$ to $\boldsymbol{c} - \boldsymbol{v}$.

   (d) While $\boldsymbol{c}[0] \geq H_0^m$, report $\boldsymbol{c}$ and subtract $\boldsymbol{v}$ to $\boldsymbol{c}$.
   (e) Set $\boldsymbol{c}$ to the addition of $\boldsymbol{d}$ and a 1–transition-vector such that $\boldsymbol{c}$ has the smallest possible positive ordinate and an abscissa in $[H_0^m, H_0^M[$.
4. Set $\boldsymbol{c}$ to the opposite of a 1–transition-vector such that $\boldsymbol{c}$ has the smallest possible negative ordinate and an abscissa in $[H_0^m, H_0^M[$.
5. While $\boldsymbol{c}[1] \geq H_1^m$:
   (a) Perform Item 3a, Item 3b, Item 3c and Item 3d.
   (b) Set $\boldsymbol{c}$ to the subtraction of $\boldsymbol{d}$ and a 1–transition-vector such that $\boldsymbol{c}$ has the smallest possible negative ordinate and an abscissa in $[H_0^m, H_0^M[$.

When the volume of Λ is less than I_0, we get exactly the algorithm of the line sieve, and when the volume of Λ is larger than I_0, the step in Item 3b, Item 3c and Item 3d are just the report of $\boldsymbol{c}$ and the final step in Item 3e can be performed according to Proposition 6.2.

Let $\mathcal{H}$ denote the sieving region and Λ the lattice of dimension t. In this case, an extension of the line sieve and the sieve of Franke–Kleinjung, also called *plane sieve*, can be performed, as shown at Section 6.4.4. But, when the volume of the lattice is larger than $I_0 I_1$, the average number of elements per plane is less than one and the plane sieve can be not as efficient as we can hope. In this chapter, we present a general algorithm to sieve in small dimensions. We first present a sieve algorithm with an oracle that produces the transition-vectors and then, because the oracle does not exist with the features we want, an algorithm that try to generate a subset of the transition-vectors and how to perform the enumeration with such a subset.

6.3 Sieve algorithm with oracle

In this section, given the lattice Λ and the sieving region $\mathcal{H}$, we consider that an oracle can produce a complete set of k–transition-vectors, where k is in $[0, t[$. The main idea of the enumeration algorithm is, given an element $\boldsymbol{c}$ in the intersection of Λ and $\mathcal{H}$, to modify its $t-1$ first coordinates, that is finding all the elements in $\Lambda \cap \mathcal{H}$ with the coordinates $(\cdot, \cdot, \cdots, \cdot, \boldsymbol{c}[t-1])$. Once the enumeration of those elements is performed, a $(t-1)$–transition-vector is added to $\boldsymbol{c}$ such that the new element fits in $\mathcal{H}$. And then, we perform recursively the enumeration of the elements with the last coordinate equal to $\boldsymbol{c}[t-1]$.

The vector $\mathbf{0}$ is, by definition, in the sieving region and is the starting point of our enumeration algorithm. The complete algorithm is the following (but for simplicity, we do not care if we report several times the same element):

Initialization.

1. Get the transition-vectors from the oracle $\mathcal{O}(\mathcal{H}, \Lambda)$.
2. Set $\boldsymbol{c}$ to $\mathbf{0}$ and k to $t-1$.

Enumeration.

3. While $\boldsymbol{c}[k] < H_k^M$:

   (a) Report $\boldsymbol{c}$.
   (b) If $k > 0$, call recursively this enumeration procedure with input $\boldsymbol{c}$ and $k-1$.
   (c) Add $\boldsymbol{v}$ to $\boldsymbol{c}$, where $\boldsymbol{v}$ is a k–transition-vector which implies that $(\boldsymbol{c}+\boldsymbol{v})[k]$ is the smallest possible value larger than $\boldsymbol{c}[k]$.
4. Recover $\boldsymbol{c}$ as it was when the procedure was called.
5. While $\boldsymbol{c}[k] \geq H_k^m$:
   (a) Report $\boldsymbol{c}$.
   (b) If $k > 0$, call recursively this enumeration procedure with input $\boldsymbol{c}$ and $k-1$.
   (c) Subtract $\boldsymbol{v}$ to $\boldsymbol{c}$, where $\boldsymbol{v}$ is a k–transition-vector which implies that $(\boldsymbol{c}-\boldsymbol{v})[k]$ is the smallest possible value smaller than $\boldsymbol{c}[k]$.

Remark 6.4. By unrolling the recursive calls, we get exactly the algorithm given in Section 6.2.3 when $t = 2$, modulo avoiding the duplicates.

Proposition 6.3. *Let Λ be a lattice and $\mathcal{H}$ be a sieving region of dimension t. The algorithm described previously reports at least once all the elements in the intersection of Λ and $\mathcal{H}$.*

Proof. We prove it by induction.

Let $k = 0$. The set of 0–transition-vectors is either empty or contains an element, say $\boldsymbol{v}$. If the set is empty, the two while loops are broken at the first iteration and only $\boldsymbol{c}$ is reported, which is correct: if another element was in the same line as $\boldsymbol{c}$, this would create a 0–transition-vector. Otherwise, from an element $\boldsymbol{c}$, we can find with the algorithm all the elements in the same line as the one of $\boldsymbol{c}$, that is $\boldsymbol{c} + \alpha\boldsymbol{v}$, where α is an integer.

Let $k = 1$. Let $\boldsymbol{c}$ be in the intersection of Λ and $\mathcal{H}$. All the elements of the form $(\cdot, \boldsymbol{c}[1], \boldsymbol{c}[2], \ldots, \boldsymbol{c}[t-1])$ are reported by the case $k = 0$. By the definition of the transition-vectors, if the set of 1–transition-vector is not empty, there exists at least one vector $\boldsymbol{v}$ in the set of the 1–transition-vectors that allows to reach a point in $\mathcal{H}_0$ with the smallest ordinate larger than the one of $\boldsymbol{c}$. There does not exist any element of $\Lambda \cap \mathcal{H}$ with an ordinate in $]\boldsymbol{c}[1], (\boldsymbol{c}+\boldsymbol{v})[1][$. Then, by considering all the additions or subtractions of a 1–transition-vector and the case $k = 0$, we cannot miss any element of the form $(\cdot, \cdot, c_2, c_3, \ldots, c_{t-1})$ in $\Lambda \cap \mathcal{H}$, where all the c_i are in $[H_i^m, H_i^M[$.

Let $k = k_0 < t$. Suppose that, given $\boldsymbol{c}$, the algorithm enumerate all the elements of the form $(\cdot, \cdot, \ldots, \cdot, \boldsymbol{c}[k_0+1], \boldsymbol{c}[k_0+2], \cdots, \boldsymbol{c}[t-1])$. Using the same argument as for the case $k = 1$, we can enumerate all the elements of the form $(\cdot, \cdot, \ldots, \cdot, c_{k_0+1}, c_{k_0+2}, \ldots, c_{t-1})$ in $\Lambda \cap \mathcal{H}$, where all the c_i are in $[H_i^m, H_i^M[$. □

The enumeration algorithm is described, however the oracle to get the sets of transition-vectors is not determined in our description. Algorithms to compute Graver basis [82] can be used to find t sets of transition-vectors that are $(t-1)$-suitable with respect to our enumeration algorithm. When the volume r of the lattice Λ is less than I_0, we prove in Section E.3 that the t sets of transition-vectors given in Section 6.4.4, which are $(t-1)$-suitable, are in the Graver basis. Such a proof can be extended with the sets of transition-vectors of Section 6.4.4 when $I_0 < r < I_1$. We have experimentally verified for other r that the t sets of transition-vectors obtained by the computation of a Graver basis are $(t-1)$-suitable.

Definition 6.2 (Graver basis)**.** Let Λ be a full-rank lattice of dimension t. Let $\mathcal{L}$ be the set of non-zero vectors of Λ in an orthant which cannot be written as the sum of two non-zero vectors of the lattice in this orthant. The Graver basis of a lattice is the intersection of the $\mathcal{L}$ defined in each orthant of the t-dimensional space.

Remark 6.5. Let Λ and $\mathcal{L}$ be as in Definition 6.2, for a given orthant. If a vector $\boldsymbol{v}$ is in $\mathcal{L}$, there does not exist an element $\boldsymbol{u}$ of Λ in the orthant such that $\boldsymbol{u}[i] \leq \boldsymbol{v}[i]$, for i in $[0, t[$.

We use the software `4ti2` [175] as an oracle, but the timing of the computation of the Graver basis often exceeds the time to perform a generalized plane sieve, whose a possible description can be found at the end of Section 6.4.4. In the generic case, it seems not possible to bound the cardinality of a Graver basis by a function of r and t, as it often contains an exponential number of vectors [142]. We report in Table 6.1 the number of vectors in the Graver basis for a given size and the number of generated *nearly-transition-vectors* (a weaker notion than transition-vectors, that includes the transition-vectors and, given a vector of $\boldsymbol{v}$ of the lattice, it is easy to verify if $\boldsymbol{v}$ is or not a nearly-transition-vectors, see Section 6.4.1), with respect to the sieving region $\mathcal{H} = [-2^5, 2^5[\times[-2^5, 2^5[\times[-2^5, 2^5[\times[0, 2^5[$.

Volume of lattice	Cardinality of Graver basis			Number of nearly-transition-vectors		
	min	average	max	min	average	max
$[0, 2^6[$	4	241	2827	4	119	489
$[2^6, 2^{12}[$	107	4217	132036	31	437	5173
$[2^{12}, 2^{18}[$	—	9839	—	—	98	—
$[2^{18}, 2^{23}[$	—	19778	—	—	8	—

Table 6.1 – Experiments on Graver basis thanks to the `graver` binary of `4ti2`.

Concerning the first two lines of Table 6.1, the description of the generalized sieves in Section 6.4.4 can be performed without the need for an algorithm to compute a Graver basis. As we can see, the number of vectors generated by the Graver basis computation is way too large, compared to what we need to get t sets of transition-vectors that are $(t-1)$-suitable. The results are obtained by considering 500 random lattices. For the last two lines, we just give an average number on 10 lattices, since dealing with some lattices of these volumes can require more than 20 hours of computation and more than 16 GB of memory, which is incompatible with the expected running time of our algorithm.

As this oracle spends a lot of time to give t sets of transition-vectors that are $(t-1)$-suitable, we therefore propose our own construction of an approximated oracle: we accept the fact that in some not-too-frequent cases, one or more sets of possible transition-vectors do not allow to have the t sets of transition-vectors being $(t-1)$-suitable and possibly contain rather than transition-vectors nearly-transition-vectors.

6.4 Sieve algorithm without oracle

If we do not have access to an oracle that generates complete or suitable sets of transition-vectors (and it is generally the case), we need to provide a specific algorithm that build transition-vectors. But, we cannot prove efficiently that a vector is a transition-vector. That is why we will define and use the nearly-transition-vectors: nearly-transition-vectors share the same properties than the transition-vectors except the condition that, between $\boldsymbol{c}$ in the intersection of $\mathcal{H}$ and Λ and $\boldsymbol{c}+\boldsymbol{v}$, where $\boldsymbol{v}$ is a k–nearly-transition-vector, there can exist an element $\boldsymbol{c'}$ with the coordinate k between the one of $\boldsymbol{c}$ and $\boldsymbol{c}+\boldsymbol{v}$. Verifying if $\boldsymbol{v}$ is a k–nearly-transition-vector is easy, we need to verify Property 6.2. This change does not impact drastically the enumeration algorithm, but by using the weaker notion of nearly-transition-vectors, we could miss a large number of elements. We will describe two different sieve algorithms: their major difference is the construction of the nearly-transition-vectors and imply modifications in the enumeration algorithms. Before describing the algorithm, we define a level of a sieving algorithm for a sieving region $\mathcal{H}$, a key notion for the rest of the description.

Definition 6.3 (Level)**.** Let Λ be a lattice and $\mathcal{H}$ be a sieving region. We define the maximum level of a sieve algorithm with respect to Λ and $\mathcal{H}$ as the minimal integer value $\ell \leq t-1$ such that the intersection of the cuboids $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\cdots\times[H_\ell^m, H_\ell^M[\times\{c_{\ell+1}\}\times\{c_{\ell_{\max}+2}\}\times\cdots\times\{c_{t-1}\}$, where $(c_{\ell_{\max}+1}, c_{\ell_{\max}+2}, \ldots, c_{t-1})$ are in $[H^m_{\ell_{\max}+1}, H^M_{\ell_{\max}+1}[\times[H^m_{\ell_{\max}+2}, H^M_{\ell_{\max}+2}[\times\cdots\times[H^m_{t-1}, H^M_{t-1}[$, and the lattice Λ contain more than one element on average.

In case $\mathcal{H}$ contains less than one element on average, we set the maximum level $\ell_{\max}$ to the value $t-1$.

In the following algorithms, the level will play a central role. It allows us to control the most efficiently which type of sieve is used.

Example 6.1. Let $\mathcal{H}$ be equal to $[-2^{15}, 2^{15}[\times[0, 2^{15}[$ and the volume $r = 2^{17}+29$ of the lattice Λ.With respect to $\mathcal{H}$ and Λ, the level is equal to 1. In this case, because $r > I_0$, the lattice sieve is the most efficient sieve to be used, and our general algorithms will degenerate in the lattice sieve. But, it is also possible to use the line sieve to enumerate the elements in the intersection of $\mathcal{H}$ and Λ.

The general algorithms degenerates in the line sieve when the level is equal to 0. In our sieve algorithms, the parameter ℓ can be replaced by any smaller integer: it will result in the call of a less efficient sieve in term of running time, but will enumerate all the expected elements.

6.4.1 Preliminaries

In this section, we define two new types of vectors that try to approximate the notion of transition-vectors and describe some properties that are shared by the two sieve algorithms we will describe.

Nearly-transition-vectors

Given a vector $\boldsymbol{v}$ of a lattice Λ and a sieving region $\mathcal{H}$, it is almost impossible to determine quickly if the vector $\boldsymbol{v}$ is a transition-vector or not, as described

in Section 6.1. A nearly-transition-vector shares almost the same properties as a transition-vector, the only difference is that we do not require that the coordinate $\boldsymbol{c_n}[k]$ of Definition 6.1 is the smallest possible larger than $\boldsymbol{c}[k]$

Definition 6.4. Let k be in $[0, t[$. A *k–nearly-transition-vector* is an element $\boldsymbol{v} \neq \boldsymbol{0}$ of Λ that allows to reach, from $\boldsymbol{c}$ in the intersection of Λ and $\mathcal{H}_k$, a new element $\boldsymbol{c_n} = \boldsymbol{c} + \boldsymbol{v}$ in this intersection, such that the $t-1-k$ last coordinates of $\boldsymbol{c}$ and $\boldsymbol{c_n}$ are equal and the coordinate $\boldsymbol{c_n}[k]$ must be larger than $\boldsymbol{c}[k]$.

Proposition 6.4. *Let k be in $[0, t[$. The vector $\boldsymbol{v}$ in the lattice Λ is a k–nearly-transition-vector if:*

1. *the coordinate k of $\boldsymbol{v}$ is positive,*
2. *for all j in $]k, t[$, $\boldsymbol{v}[j] = 0$,*
3. *for all j in $[0, k]$, $|\boldsymbol{v}[j]| < I_j$.*

A k–transition-vector is necessarily a k–nearly-transition-vector. Instead of the difficulty to show efficiently if, given a vector $\boldsymbol{v}$ of a lattice Λ and a sieving region $\mathcal{H}$, the element $\boldsymbol{v}$ is or not a transition-vector, it is possible to efficiently prove that $\boldsymbol{v}$ is or not a nearly-transition-vector, by only verifying the conditions of Proposition 6.4

Shape of the nearly-transition-vectors

We will first describe the shape of the nearly-transition-vectors for the two and three-dimensional cases and then generalize our observations.

Shape of the nearly-transition-vectors in two and three dimensions. In the 2-dimensional case, when $r < I_0$, that is $\ell = 0$, we apply the line sieve and we know that the 0–transition-vector is equal to $(r, 0)$. Given the algorithm in Section 6.3, the set of the 0–transition-vector and the set of 1–transition-vectors $\{(\lambda_0, 1), (\lambda_0 - r, 1)\}$ are 1-suitable. The shape of these transition-vectors, and then nearly-transition-vectors, is therefore in $(O(r), 1)$.

Remark 6.6. The notation $O(i)$ is used here to mean that the value is almost equal to the value i.

Still in the case $t = 2$, when $r > I_0$, that is $\ell = 1$, there does not exist a 0–transition-vector and the set of 1–transition-vectors is constituted by vectors in the set $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{u} + \boldsymbol{v}\}$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ are the two Franke–Kleinjung vectors. As remarked in Section 6.2.2, the first coordinates of $\boldsymbol{u}$ and $\boldsymbol{v}$ are in $O(I_0)$ and the second are in $O(r/I_0)$.

The shape of the nearly-transition-vectors in the three-dimensional case, given in [94, 72], are the following:

- when $\ell = 0$, the shape is equal to $(O(r), O(1), O(1))$;
- when $\ell = 1$, the shape is equal to $(O(I_0), O(r/I_0), O(1))$;
- when $\ell = 2$, the shape is equal to $(O(I_0), O(I_1), O(r/(I_0 I_1)))$.

It seems obvious, given a level ℓ, to try to generalize this shape as (we do not write the $O(\cdot)$ notation for clarity) $(I_0, I_1, \ldots, I_{\ell-1}, r/(I_0 \times I_1 \times \cdots \times I_{\ell-1}), 1, 1, \ldots, 1)$. In the following, we will show why this general shape is the expected one.

General shape of the nearly-transition-vectors. The general shape described previously is the one we can expect by doing a crude generalization of the cases of two and three dimensions; we will show that this is exactly what we need to have in the two situations where $\ell < t-1$ and $\ell = t-1$.

Level $\ell < t-1$. In this situation, the number of elements of the form $(\cdot, \cdot, \ldots, \cdot, c_{\ell+1}, c_{\ell+2}, \ldots, c_{t-1})$ where c_k is in $[H_k^m, H_k^M[$ is on average larger than one. These sets of elements are in cuboids where the $t-(\ell+1)$ last coordinates are fixed: we say that the dimension of such cuboids is equal to $\ell+1$. To try to explore all these cuboids, the k–nearly-transition-vectors, where k is in $[\ell+1, t-1[$, must have, a coordinate k equal to ideally 1, in order to exhaustively enumerate all the possible cuboids of dimension ℓ, and if not 1, then a value the smallest possible. Inside a cuboid of dimension $\ell+1$, the average number of elements is equal to $(I_0 \times I_1 \times \cdots \times I_\ell)/r$, larger than one, while the average number of elements in a cuboid of dimension ℓ included in a cuboid of dimension $\ell+1$ is less than one.

Hence, a k–nearly-transition-vector $\boldsymbol{c}$, where k is in $[0, \ell[$, has the coordinates $\boldsymbol{c}[j]$ in the magnitude of I_j, where j is in $[0, k]$, because we except less than one element per cuboid of dimension $k+1$. Following Definition 6.3, a ℓ–nearly-transition-vector verify the same property except for the coordinate ℓ which is in the order of $r/(I_0 \times I_1 \times \cdots \times I_{\ell-1})$, in order to find more than one element per cuboid of dimension $\ell+1$. Putting all together, the nearly-transition-vectors have more or less the form $(I_0, I_1, \ldots, I_{\ell-1}, r/(I_0 \times I_1 \times \cdots \times I_{\ell-1}), 1, 1, \ldots, 1)$.

Level $\ell = t-1$. If r is less than $I_0 \times I_1 \times \cdots \times I_{t-1}$ (it means that the volume of Λ is larger than the number of elements in $\mathcal{H}$, allowing us to expect more than one element per cuboid of dimension t), the shape of the nearly-transition-vectors is in $(I_0, I_1, \ldots, I_{t-2}, r/(I_0 \times I_1 \times \cdots \times I_{t-2}))$ using the same previous arguments as when $\ell < t-1$.

If r is larger than $I_0 \times I_1 \times \cdots \times I_{t-1}$, the shape of the vector is the same but the last coordinate is larger than I_{t-1}: this will give a vector with a not-so-small last coordinate, which is consistent with what we expect: all the coordinates except the last one must enumerate less than one element per cuboid of the corresponding dimension, and the last one must allow to find less than one element in $\mathcal{H}$.

Obtaining the shape. We describe in the following two algorithms to produce nearly-transition-vectors. These two ways result in two different but similar enumeration algorithms. Each of them has advantages and drawbacks, that we discuss later. To produce these nearly-transition-vectors, the two algorithms uses a skew lattice reduction, as described in Section A.1, with the same (vector of) weight w: we therefore define it before going into details.

Definition 6.5 (Weight)**.** Let ℓ be a level of a sieve with respect to Λ and $\mathcal{H}$. The shape of the nearly-transition-vectors at this level is equal to $s = (I_0, I_1, \ldots, I_{\ell-1}, r/(I_0 \cdot I_1 \cdot \cdots \cdot I_{\ell-1}), 1, 1, \ldots, 1)$. The weight w we use in the weighted lattice basis reduction is defined by the t-tuple $(1/s[0], 1/s[1], \ldots, 1/s[t-1])$.

Enumeration algorithm

The enumeration algorithm to enumerate the elements in the intersection of the lattice and the sieving region is similar to the one given in Section 6.3. This algorithm has a major drawback, the use of nearly-transition-vectors cannot ensure that the report of the elements in the intersection of the lattice and the sieving region is exhaustive. This is the reason of the use of skew-small-vectors to compute during the enumeration some missing nearly-transition-vectors, when possible. We describe the two possible situations on an example, assuming that $t = 2$ and the set of 1–transition-vectors is equal to $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{u} + \boldsymbol{v}\}$, where these three vectors are as in Figure 6.2.

Suitable nearly-transition-vectors. In our example, if only $\boldsymbol{u}$ and $\boldsymbol{u} + \boldsymbol{v}$ are in the set of 1–nearly-transition-vectors, we say that the set of 1–nearly-transition-vectors is suitable, which is a notion different but close to the one for a group of i sets of transition-vectors. If we consider the enumeration of the elements in $\mathcal{H}$, all the new elements have their abscissa in $[H_0^m, H_0^M[$. We indeed miss some elements in the enumeration as we lack the vector $\boldsymbol{v}$, but the enumeration is stopped when the bounds $[H_1^m, H_1^M[$ are reached and this condition is the only stopping criteria. We now give a more formal and general description of this case.

A set of k–nearly-transition-vectors is suitable if, given any element of the lattice such that its coordinates fit in $\mathcal{H}_k$, there exist at least one k–nearly-transition-vector to go from this element to an other in $\mathcal{H}_{k-1}$, meaning that we can find another one element that reach all the bounds of the k first intervals of the sieving region except for the coordinate k, that is a stopping criteria at a point of the enumeration. In this situation, the enumeration can fail to report some elements but the enumeration is stopped regularly. In the following algorithms, we try to reduce the number of missed elements by generating sufficiently many nearly-transition-vectors during the initialization of the enumeration but we do not try to test if this situation occurs during the enumeration and therefore do not propose mechanisms to avoid missing elements.

Lack of nearly-transition-vectors. In our example, if only $\boldsymbol{u}$ and $\boldsymbol{v}$ are in the set of 1–nearly-transition-vectors, we say that there is a lack in the set of 1–nearly-transition-vectors. If we consider the enumeration of the elements in the plane, we cannot always find a new element in the strip around the ordinate axis bounded by $[H_0^m, H_0^M[$. Contrary to the previous case, for which the enumeration is stopped when the bounds $[H_1^m, H_1^M[$ are reached, the enumeration can be stopped well before having reached these bounds, implying the missing of a possibly large proportion of the elements we hope to enumerate. We now give a more formal and general description of this case.

There is a lack in the set of k–nearly-transition-vectors if, given any element of the lattice such that its coordinates fit in $\mathcal{H}$, there are cases where there are no k–nearly-transition-vector to go from this element to another one in $\mathcal{H}_{k-1}$. The enumeration can be stopped even if there are elements left to be enumerated. At this stage, it is impossible to determine the reason of the lack of k–nearly-transition-vector. It could be because of a strong skewness of the lattice, or because the shape of the sieving region is unbalanced, or because the initialization procedure did not produce enough k–nearly-transition-vector. In

this case, we need to find a new k–nearly-transition-vector, computed on the fly during the enumeration. We want this situation to be very rare in order to get an efficient sieve. In this case, we propose strategies to find new nearly-transition-vectors: these strategies are also called *fall-back* stategies.

Fall-back strategies

To perform these strategies, we use the notion of *skew-small-vectors*. In the following, we will describe a general fall-back strategy, instantiated differently by the two enumeration algorithms.

Skew-small-vectors. The initialization step of the two enumeration algorithms build a lot of vectors having the specific shape we have described above. All the vectors we will produced by the initialization procedure are not nearly-transition-vectors, but have coordinates close to the one we target: these vectors will be called *skew-small-vectors*. Even if some vectors can be very small and seem to not respect the target shape, we still keep the name skew-small-vector. A k–nearly-transition-vector is necessarily a k–skew-small-vector.

Definition 6.6. Let k be in $[0, t[$. A *k–skew-small-vector* is an element $\boldsymbol{v} \neq \mathbf{0}$ of Λ that allows to reach, from $\boldsymbol{c}$ in Λ, a new element $\boldsymbol{c_n} = \boldsymbol{c} + \boldsymbol{v}$ in Λ, such that the $t - 1 - k$ last coordinates of $\boldsymbol{c}$ and $\boldsymbol{c_n}$ are equal and the coordinate $\boldsymbol{c_n}[k]$ must be larger than $\boldsymbol{c}[k]$.

Proposition 6.5. *Let k be in $[0, t[$. A vector $\boldsymbol{v}$ in the lattice Λ is a k–skew-small-vector if:*

1. *the coordinate k of $\boldsymbol{v}$ is positive,*
2. *for all $j \in]k, t[$, $\boldsymbol{v}[j] = 0$.*

Summary of the different types of vectors. We propose to depict in Figure 6.3 the three types of vectors on a 2-dimensional example. In this context, there are only one 0–transition-vector, 1–transition-vectors, 0–nearly-transition-vectors, 1–nearly-transition-vectors.

Let Λ be a lattice and $\mathcal{H}$ be a sieving region. Let $\boldsymbol{v}$ be a k–skew-small-vector and $\mathcal{H}_k$ be an extended sieving region. We distinguish three cases, according to the type of the vector $\boldsymbol{v}$:

- if $\boldsymbol{v}$ is a k–skew-small-vector, there is no guaranty that $\boldsymbol{c} + \boldsymbol{v}$ is in the extended sieving region if $\boldsymbol{c}$ is an element in the intersection of Λ and $\mathcal{H}$.
- if $\boldsymbol{v}$ is a k–nearly-transition-vector, there exists an element $\boldsymbol{c}$ in the intersection of Λ and $\mathcal{H}$ such that $\boldsymbol{c} + \boldsymbol{v}$ is in the extended sieving region.
- if $\boldsymbol{v}$ is a k–transition-vector, there exists an element $\boldsymbol{c}$ in the intersection of Λ and $\mathcal{H}$ such that $\boldsymbol{c} + \boldsymbol{v}$ is in the extended sieving region with the smallest possible coordinate k larger than the one of $\boldsymbol{c}$.

Let $\boldsymbol{c}$ be an element in the intersection of the lattice Λ and the sieving region $\mathcal{H}$, and $\boldsymbol{v}$ be a k–skew-small-vector. We give here the patterns of a k–skew-small-vector, which is the same for a k–nearly-transition-vector or a k–transition-vector, and the one of $\boldsymbol{c} + \boldsymbol{v}$.

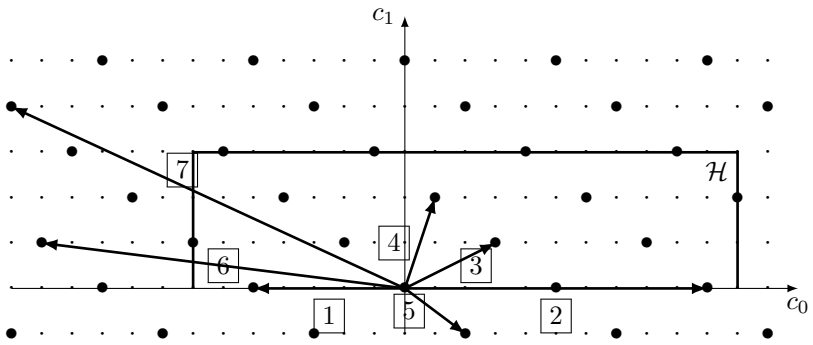


Figure 6.3 – Some vectors of the lattice. The vector $\boxed{1}$ is the opposite of the 0–transition-vector, $\boxed{2}$ is a 0–nearly-transition-vector, $\boxed{3}$ is a 1–transition-vector, $\boxed{4}$ is a 1–nearly-transition-vector, $\boxed{5}$ is the opposite of a 1–transition-vector, $\boxed{6}$ is a 1–transition-vector and $\boxed{7}$ is a 1–skew-small-vector.

	$\boldsymbol{c}[0]$	$\boldsymbol{c}[1]$	$\dots$	$\boldsymbol{c}[k]$	$\boldsymbol{c}[k+1]$	$\boldsymbol{c}[k+2]$	$\dots$	$\boldsymbol{c}[t-1]$
$+$	$\boldsymbol{v}[0]$	$\boldsymbol{v}[1]$	$\dots$	$\boldsymbol{v}[k] > 0$	0	0	$\dots$	0
	$(\boldsymbol{c}+\boldsymbol{v})[0]$	$(\boldsymbol{c}+\boldsymbol{v})[1]$	$\dots$	$(\boldsymbol{c}+\boldsymbol{v})[k]$	$\boldsymbol{c}[k+1]$	$\boldsymbol{c}[k+2]$	$\dots$	$\boldsymbol{c}[t-1]$

Generating nearly-transition-vectors on the fly. Let us consider the case when, during the enumeration, the addition of the known nearly-transition-vectors fails to land in $\mathcal{H}_{k-1}$, according to the previous description of a lack of nearly-transition-vectors. The case of the subtraction follows the same idea. The set of known nearly-transition-vectors can be computed by the two methods described in the following, one for each enumeration algorithm. The main idea is to store all the vectors produced by the initialization procedure, and not only keep the nearly-transition-vectors. We assume that the sets of j–skew-small-vectors, where j is in $[0, t[$, are not empty. Let consider the set of k–skew-small-vectors and the element $\boldsymbol{c}$ in $\Lambda \cap \mathcal{H}_k$. Adding a k–skew-small-vector, say $\boldsymbol{v}$, to $\boldsymbol{c}$ makes the result necessarily out of the sieving region. But, it is possible to minimize the coordinates of $\boldsymbol{c}+\boldsymbol{v}$ using $\boldsymbol{d}$, a linear combination of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \dots, \boldsymbol{b_{k-1}}\}$, and obtain a potential new k–nearly-transition-vector equal to $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$. This idea will be specialized depending on the context of our algorithms.

Bird's-eye view of the algorithms

The structure of the algorithm is similar to the one of Section 6.3. The input of the algorithm is a lattice Λ and a sieving region $\mathcal{H}$. The algorithm reports the elements in the intersection of Λ and $\mathcal{H}$.

Initialization.

1. Given $\mathcal{H}$ and Λ, call a procedure `findV` that returns some nearly-transition-vectors and skew-small-vectors.

2. Set $\boldsymbol{c}$ to $\mathbf{0}$ and k to $t-1$

Enumeration.

3. While $\boldsymbol{c}[k] < H_k^M$:
   (a) Report $\boldsymbol{c}$.
   (b) If $k > 0$, call recursively this enumeration procedure (`sieve`) with input $\boldsymbol{c}$ and $k-1$.
   (c) Add $\boldsymbol{v}$ to $\boldsymbol{c}$, where $\boldsymbol{v}$ is a k–nearly-transition-vector, such that $\boldsymbol{c}$ lands in $\mathcal{H}_{k-1}$ (`add`)
   (d) If not possible, call a fall-back strategy (`fbAdd`) that tried to produce a new element in $\mathcal{H}$, and therefore a new k–nearly-transition-vector, by using k–skew-small-vectors.
4. Recover $\boldsymbol{c}$ as it was when the procedure was called.
5. While $\boldsymbol{c}[k] \geq H_k^m$:
   (a) Perform Item 3a, Item 3b, Item 3c and Item 3d by considering $\boldsymbol{c} - \boldsymbol{v}$ instead of $\boldsymbol{c} + \boldsymbol{v}$.

This description allows us to propose the general form of the enumeration algorithm in Algorithm 6.1 and of the Function `sieve`. The sign `...` denotes that Function `sieve` can have additional arguments.

Function `sieve`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ, L, ...)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors T, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the list L that contains the elements in $\mathcal{H} \cap \Lambda$

$\boldsymbol{c_t} \leftarrow \boldsymbol{c}$;
while $\boldsymbol{c_t}[k] < H_k^M$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L, ...)$;
  $\boldsymbol{c_t} \leftarrow$ `add`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda, ...)$;
end
$\boldsymbol{c_t} \leftarrow$ `sub`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda)$;
while $\boldsymbol{c_t}[k] \geq H_k^m$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L, ...)$;
  $\boldsymbol{c_t} \leftarrow$ `sub`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda, ...)$;
end

Remark 6.7. From an implementation point of view, the k–nearly-transition-vectors are sorted by increasing k-coordinate and tested in this order.

The two procedures we need to instantiate in this general description are on the one hand the generation of the nearly-transition-vectors and the skew-small-vectors, in Item 1 (`findV`), and on the other hand the strategies to produce a new nearly-transition-vector, in Item 3d (`fbAdd`). These two procedures are

Algorithm 6.1: General structure of the enumeration algorithms.

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the sieving region $\mathcal{H}$, the level ℓ with respect to $\mathcal{H}$ and Λ, the bounds on the small linear combinations $\mathcal{A}$
output: list of visited point in $\Lambda \cap \mathcal{H}$
$(T, S) =$ `findV`$(\Lambda, \mathcal{H}, \ell, \mathcal{A})$; $L \leftarrow \emptyset$;
`sieve`$(t-1, \mathbf{0}, \mathcal{H}, T, S, \Lambda, L, ...)$;
remove duplicates of L;
return L;

linked together, and so the way we produce the nearly-transition-vectors and skew-small-vectors to ensure or try to ensure some properties will affect the way we use and design the fall-back strategies.

We briefly summarize in Table 6.2 the major differences between the two proposed algorithms. The justification of the choices will be given in the appropriate sections.

	`globalntvgen`	`localntvgen`
Initialization	Skew lattice reduction on the whole basis	Skew lattice reduction on the first $\ell + 1$ vectors and $t - (\ell + 1)$ closest vector problems solutions
Fall-back	Frequently used	Rarely used, aggressive strategy

Table 6.2 – Main features of the two sieve algorithms.

We choose the name `globalntvgen` for the first algorithm because all the nearly-transition-vectors are build from the whole skew basis of the initialization, instead of the second algorithm, for which all the nearly-transition-vectors are more controlled, named `localntvgen`.

6.4.2 First algorithm: `globalntvgen`

Initial generation of the nearly-transition-vectors

We apply a weighted basis reduction on the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ with weight w. Then, we perform small linear combinations of the output basis vectors. Each produced vector is at least a skew-small-vector and can be a nearly-transition-vector. We use k different lists to store the k–skew-small-vectors (that is all the produced vectors), and k other lists to store specifically the k–nearly-transition-vectors.

Generating nearly-transition-vectors on the fly

At this step, all the additions to $\boldsymbol{c}$ in the sieving region $\mathcal{H}$ of a k–nearly-transition-vector fail to land in $\mathcal{H}_{k-1}$. The addition of $\boldsymbol{v}$, a k–skew-small-vector, is necessarily out of $\mathcal{H}_{k-1}$. We try to minimize the coefficients of $\boldsymbol{c} + \boldsymbol{v}$ by using

the set of vectors $\mathcal{B}_k = \{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{k-1}}\}$. Indeed, we want to find an element with the same coordinate k than the one of $(\boldsymbol{c}+\boldsymbol{v})$, but with the k first coordinates smaller than $\boldsymbol{c}+\boldsymbol{v}$: the specific pattern of the vectors of $\mathcal{B}_k$ allows to reach such a goal. Let $\boldsymbol{d}$ be the vector subtracted to $\boldsymbol{c}+\boldsymbol{v}$ to shrink its coefficients. If $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ fits in the sieving region, $\boldsymbol{v}-\boldsymbol{d}$ is a new k–nearly-transition-vector. If not, set $\boldsymbol{c}$ to $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ and redo this procedure, until $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ fits in $\mathcal{H}$ or its coordinate k is larger than H_k^M. When this procedure is called, the set of k–skew-small-vectors have already been filled by the initialization step. The patterns of the different vectors is the following:

	$\boldsymbol{c}[0]$	$\boldsymbol{c}[1]$	$\ldots$	$\boldsymbol{c}[k-1]$	$\boldsymbol{c}[k]$	$\boldsymbol{c}[k+1]$	$\boldsymbol{c}[k+2]$	$\ldots$	$\boldsymbol{c}[t-1]$
$+$	$\boldsymbol{v}[0]$	$\boldsymbol{v}[1]$	$\ldots$	$\boldsymbol{v}[k-1]$	$\boldsymbol{v}[k]>0$	0	0	$\ldots$	0
$-$	$\boldsymbol{d}[0]$	$\boldsymbol{d}[1]$	$\ldots$	$\boldsymbol{d}[k-1]$	0	0	0	$\ldots$	0
	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[0]$	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[1]$	$\ldots$	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[k-1]$	$(\boldsymbol{c}+\boldsymbol{v})[k]$	$\boldsymbol{c}[k+1]$	$\boldsymbol{c}[k+2]$	$\ldots$	$\boldsymbol{c}[t-1]$

The different steps of this generation on the fly are the following, given $\boldsymbol{c}$ in $\Lambda \cap \mathcal{H}$ and k in $[0, t[$:

1. While $\boldsymbol{c}[k] < H_k^M$
   (a) For all k–skew-small-vectors $\boldsymbol{v}$
      i. Reduce the coefficients of $\boldsymbol{c}+\boldsymbol{v}$ by $\boldsymbol{d}$, a linear combination of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{k-1}}\}$.
      ii. If $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ is in $\mathcal{H}$, return $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$.
   (b) Set $\boldsymbol{c}$ to one of the vector $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ computed during the for loop.
2. Return fail.

If this procedure do not fail, the new element in $\mathcal{H}$ is the output of this procedure and $\boldsymbol{v}-\boldsymbol{d}$ is the new k–nearly-transition-vector, computed by the difference between the output and the input vector of this procedure. This new k–nearly-transition-vector is inserted in the corresponding lists (of k–nearly-transition-vectors and k–skew-small-vectors) for further use.

Complete algorithm

We now summarize all the steps of the algorithm to enumerate the largest possible number of elements in the intersection of the lattice and the sieving region.

The generation of the nearly-transition-vectors needs a set $\mathcal{A} = [A_0^m, A_0^M[\times [A_1^m, A_1^M[\times \cdots \times [A_{t-1}^m, A_{t-1}^M[$ defined by integer intervals. This set is used to bound the coefficients of the small linear combinations. The function `index`, used by Function `findV1` is a function that returns, given a vector $\boldsymbol{v}$, the highest index of a non-zero coordinate.

The enumeration algorithm is split into three main functions, as described previously. Function `sieve1`, which is nothing that Function `sieve` instantiated for the `globalntvgen`, is the recursive function called to perform all the steps of the enumeration and is written in Appendix E.2, as Algorithm E.3 that combine all the different function. Function `add1` is called to try to add a nearly-transition-vector and `fbAdd1` to try to find a new nearly-transition-vector, if `add1` fails to continue the enumeration. The extended sieving region in Line 1 of Function `add1` is used to stop regularly the enumeration thanks to

Function `findV1`(Λ, $\mathcal{H}$, ℓ, $\mathcal{A}$)

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the sieving region $\mathcal{H}$, the level ℓ with respect to Λ and $\mathcal{H}$, the bounds $\mathcal{A}$ on the small linear combinations
output: sets of nearly-transition-vectors and skew-small-vectors
$T \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$; $S \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$; `// sizes of` T `and` S `are` t
compute the weight w according to the shape of nearly-transition-vectors given by ℓ and $\mathcal{H}$;
$\{\boldsymbol{b'_0}, \boldsymbol{b'_1}, \ldots, \boldsymbol{b'_{t-1}}\} \leftarrow$ perform a skew basis reduction of Λ with weight w;
for *coprime* $(a_0, a_1, \ldots, a_{t-1}) \in \mathcal{A}$ **do**
  $\boldsymbol{v} \leftarrow a_0\boldsymbol{b'_0} + a_1\boldsymbol{b'_1} + \cdots + a_{t-1}\boldsymbol{b'_{t-1}}$;
  $k \leftarrow$ `index`$(\boldsymbol{v})$;
  if $\boldsymbol{v}[k] < 0$ **then** $\boldsymbol{v} \leftarrow -\boldsymbol{v}$;
  $S[k] \leftarrow S[k] \cup \{\boldsymbol{v}\}$;
  if $\boldsymbol{v}$ *is a* k*–nearly-transition-vector* **then** $T[k] \leftarrow T[k] \cup \{\boldsymbol{v}\}$;
end
for $0 \leq k < t$ **do** sort $S[k]$ and $T[k]$ by increasing k coordinate;
return (T, S);

an already existing nearly-transition-vector: indeed, if we reach all the k first bound except the $(k+1)$th ($[H_k^m, H_k^M[$), we can consider that there does not exist an element between the last element in $\mathcal{H}$ and the following element in $\mathcal{H}_{k-1}$. Function `sub1` and Function `fbSub1` are similar to Function `add1` and Function `fbAdd1`, but with subtraction, and will be described in Appendix E.2. The function `CVA` (Closest Vectors Around the targeted element) is a function that, given an element $\boldsymbol{c}$ of a lattice Λ and an integer k, returns some lattice vectors in the lattice generated by $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_k}\}$ close to the element $\boldsymbol{c}$.

A first experiment

One of the drawbacks of the algorithm is particularly visible in the case $\ell = 1$. Let us consider a basis of the lattice equal to $\{(881, 0, 0), (448, 1, 0), (268, 0, 1)\}$ and a sieving region equal to $[-2^7, 2^7[\times[-2^7, 2^7[\times[0, 2^6[$. A weighted lattice reduction can produce the basis $\{(15, 2, 0), (-165, 1, 1), (268, 0, 1)\}$. We however know that, in this case, a basis reduction on $\{(881, 0, 0), (448, 1, 0)\}$ to obtain the Franke–Kleinjung basis, gives $\{(15, 2, 0), (-253, 25, 0)\}$. Although this is a highly skewed basis, and therefore a bit rare, this situation occurs in practice. The small linear combinations will have difficulty to produce the second basis vector. With only the vector $(15, 2, 0)$, we are necessarily in the case of a lack of nearly-transition-vectors and producing the second vector with the procedure to compute nearly-transition-vectors on the fly would require to take into account very large intervals and make things prohibitively expensive without guarantee of results. With our implementation, we cannot find this second vector of the example.

The main explanation of this situation is the use of the skew basis reduction for two reasons. When the level is equal to 1 with respect to $\mathcal{H}$ and Λ, we know exactly the form of the 1–transition-vectors (and so the form of the

Function `fbAdd1`(k, $\boldsymbol{c}$, $\mathcal{H}$, S, Λ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ
output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

```
while c[k] < H_k^M do
    L ← ∅;
    for v ∈ S[k] do
        D ← CVA(c + v, Λ, k − 1);
        for d ∈ D do
            if c + v − d ∈ H then return c + v − d;
            L ← L ∪ {c + v − d};
        end
    end
    set c to an element of L;
end
return c;                                   // c is out of H
```

Function `add1`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ
output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

```
1  for v ∈ T[k] do
2      if c + v ∈ H_{k−1} then return c + v;
3  end
4  if k > 0 then
5      d ← fbAdd1(k, c, H, S, Λ);
6      if d ∈ H then T[k] ← T[k] ∪ {d − c}; S[k] ← S[k] ∪ {d − c};
7      c ← d;
8  else
9      c ← (H_0^M, H_1^M, ..., H_{t−1}^M);          // c is out of H
10 end
11 return c;
```

1–nearly-transition-vectors) and then, a part of a convenient basis thanks to Proposition 6.1 and in some cases, the skew basis combined with the small linear combination does not allow to verify this proposition. If the basis reduction applies only on the ℓ first vectors of the basis $\mathcal{B}$, we can more easily control what happens on the vectors (control the behavior of the 0 at the end of the vectors) and replace the skew basis reduction by a more appropriate algorithm, such as the one of Franke–Kleinjung when $\ell = 1$ and possibly the one of Hayasaka, Aoki, Kobayashi and Takagi [94] when $\ell = 2$, even if we do not know how to compute all the 2–transition-vectors with such a convenient basis. This is what

we do with `localntvgen`, similar to `globalntvgen`. The goal is to try to ensure that the sets of k–nearly-transition-vectors, for k in $[0, \ell]$, are at least suitable.

6.4.3 Second algorithm: `localntvgen`

The `localntvgen` uses another strategy to build the nearly-transition-vectors. To take into account the shape of the nearly-transition-vectors we generate, we propose a different fall-back strategy. This algorithm is the best suited in the case $\ell = 1$, and we believe that this algorithm is better to enumerate all the elements in the intersection of the lattice and the sieving region. This can result in a drawback in terms of timing and we summarize drawbacks and advantages of these two sieve algorithms in the next section.

Initial generation of the nearly-transition-vectors

We apply a weighted basis reduction on $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$ with the weight w. With the output vectors, we perform some small linear combinations. Each linear combination gives a k–skew-small-vector and possibly a k–nearly-transition-vector, where k is in $[0, \ell + 1[$. We consider that these sets of nearly-transition-vectors are suitable, because, to build k–nearly-transition-vectors, we use linear combination of k–nearly-transition-vectors. However, we obviously cannot ensure that without performing a generalized line sieve. To build possible k–nearly-transition-vectors where k is in $[\ell + 1, t[$, we try to minimize the $(\ell+1)$th first coordinate of each $\boldsymbol{b_k}$ by a linear combination of the output vectors of the basis reduction of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$.

Remark 6.8. Another approach to generate possible k–nearly-transition-vectors where k is in $[\ell + 1, t[$ is to try to minimize the coordinates of $\boldsymbol{b_k}$ using the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{k-1}}\}$. The output vectors are then valid inputs for the enumeration described in the `globalntvgen`, but not for the `localntvgen`.

We give an overview of the patterns of the skew-small-vectors, and then nearly-transition-vectors, in the case $\ell = 2$ and $t = 6$ in Table 6.3.

k	`globalntvgen`	`localntvgen`	Remark 6.8
0	$(>0,0,0,0,0,0)$	$(>0,0,0,0,0,0)$	$(>0,0,0,0,0,0)$
1	$(\cdot,>0,0,0,0,0)$	$(\cdot,>0,0,0,0,0)$	$(\cdot,>0,0,0,0,0)$
2	$(\cdot,\cdot,>0,0,0,0)$	$(\cdot,\cdot,>0,0,0,0)$	$(\cdot,\cdot,>0,0,0,0)$
3	$(\cdot,\cdot,\cdot,>0,0,0)$	$(\cdot,\cdot,\cdot,1,0,0)$	$(\cdot,\cdot,\cdot,1,0,0)$
4	$(\cdot,\cdot,\cdot,\cdot,>0,0)$	$(\cdot,\cdot,\cdot,0,1,0)$	$(\cdot,\cdot,\cdot,\cdot,1,0)$
5	$(\cdot,\cdot,\cdot,\cdot,\cdot,>0)$	$(\cdot,\cdot,\cdot,0,0,1)$	$(\cdot,\cdot,\cdot,\cdot,\cdot,1)$

Table 6.3 – Overview of the patterns of the k–skew-small-vectors when $\ell = 2$.

Generating nearly-transition-vectors on the fly

With the previous initial generation of the nearly-transition-vectors, the patterns of the skew-small-vectors are more specific than the ones given in Proposition 6.5, especially when $k > \ell$. Such a k–skew-small-vector verifies that the coordinate ℓ to the coordinate $k - 1$ are equal to 0, the coordinate k is equal to 1 and the coordinate $(k + 1)$ to the coordinate $t - 1$ are equal to 0.

At this step, all the additions to $\boldsymbol{c}$ in the sieving region $\mathcal{H}$ of a k–nearly-transition-vector fail to land in $\mathcal{H}_{k-1}$. The addition of $\boldsymbol{v}$, a k–skew-small-vector, is necessarily out of $\mathcal{H}_{k-1}$. We try to minimize the coefficients of $\boldsymbol{c}+\boldsymbol{v}$ by the set of vectors resulting in the skew lattice reduction of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$ to keep unchanged the coordinate $(\ell+1)$ to the coordinate $t-1$ of $\boldsymbol{c}+\boldsymbol{v}$. Let $\boldsymbol{d}$ the vector subtracted to $\boldsymbol{c}+\boldsymbol{v}$ to shrink its coefficients. If $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ fits in the sieving region, a new element in the intersection of Λ and $\mathcal{H}$ is found, and therefore a new k–nearly-transition-vector. For instance, when $\ell = 2$, $t = 6$ and $k = 5$, the different vectors respect the patterns:

	$\boldsymbol{c}[0]$	$\boldsymbol{c}[1]$	$\boldsymbol{c}[2]$	$\boldsymbol{c}[3]$	$\boldsymbol{c}[4]$	$\boldsymbol{c}[5]$
$+$	$\boldsymbol{v}[0]$	$\boldsymbol{v}[1]$	$\boldsymbol{v}[2]$	0	0	1
$-$	$\boldsymbol{d}[0]$	$\boldsymbol{d}[1]$	$\boldsymbol{d}[2]$	0	0	0
	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[0]$	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[1]$	$(\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d})[2]$	$\boldsymbol{c}[3]$	$\boldsymbol{c}[4]$	$\boldsymbol{c}[5]+1$

If $k > \ell+1$, the coordinate $(\ell+1)$ to the coordinate $k-1$ of $\boldsymbol{c}$ have not been modified, and therefore, some cube of dimension $\ell+1$ were not explored, to try to find a new starting point: to explore it, we need to call this procedure with input $k-1$ and one of the vectors generated previously. If all the recursions fail to find a new element in the intersection of the lattice and the sieving region, we set $\boldsymbol{c}$ to $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ and redo this procedure with input k and $\boldsymbol{c}$, until a generated element fits in the $\mathcal{H}$ or its coordinate k is larger than H_k^M. The different steps of this generation are the following, given $\boldsymbol{c}$ in $\Lambda \cap \mathcal{H}$ and k in $[0, t[$:

1. While $\boldsymbol{c_t}[k] < H_k^M$
   (a) For all k–skew-small-vectors $\boldsymbol{v}$
      i. Reduce the coefficients of $\boldsymbol{c}+\boldsymbol{v}$ by $\boldsymbol{d}$, a linear combination of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$.
      ii. If $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ is in $\mathcal{H}$, return $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$.
   (b) Set $\boldsymbol{c}$ to one of the vector $\boldsymbol{c}+\boldsymbol{v}-\boldsymbol{d}$ computed during the for loop.
   (c) If $k-1 > \ell$, use this procedure (additive or subtractive case) with $\boldsymbol{c}$ and $k-1$ as inputs and return the result if it does not fail.
2. Return fail.

The possible k–nearly-transition-vector is then the subtraction of the output new element in the intersection of Λ and $\mathcal{H}$ found by this procedure and the input vector $\boldsymbol{c}$, and we store it for further use.

Remark 6.9. The instruction given in Item 1(a)i must be done a little bit more carefully. Indeed, if $\ell = t-1$, we want to modify, with the vector given by the small linear combination, the $t-1$ first coordinates of $\boldsymbol{c}+\boldsymbol{v}$, and not the whole coordinates, as written. Therefore, when $\ell = t-1$, we look for a linear combination of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-2}}\}$.

Complete algorithm

We now summarize all the steps of the `localntvgen` to enumerate as many elements as possible in the intersection of the lattice and the sieving region.

Function `findV2`(Λ, $\mathcal{H}$, ℓ, $\mathcal{A}$)

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the sieving region $\mathcal{H}$, the level ℓ with respect to Λ and $\mathcal{H}$, the bounds on the small linear combinations $\mathcal{A}$
output: sets of nearly-transition-vectors and skew-small-vectors
$T \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$; $S \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$; `// sizes of` T `and` S `are` t
compute the weight w according to the shape of nearly-transition-vectors given by ℓ and $\mathcal{H}$;
$\{\boldsymbol{b'_0}, \boldsymbol{b'_1}, \ldots, \boldsymbol{b'_\ell}\} \leftarrow$ perform a skew basis reduction of $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$ with weight w;
for *coprime* $(a_0, a_1, \ldots, a_\ell) \in \mathcal{A}$ **do**
  $\boldsymbol{v} = a_0\boldsymbol{b'_0} + a_1\boldsymbol{b'_1} + \cdots + a_\ell\boldsymbol{b'_\ell}$;
  $k \leftarrow$ `index`$(\boldsymbol{v})$;
  if $\boldsymbol{v}[k] > 0$ **then**
    $S[k] \leftarrow S[k] \cup \{\boldsymbol{v}\}$;
    if $\boldsymbol{v}$ *is a k–nearly-transition-vector* **then** $T[k] \leftarrow T[k] \cup \{\boldsymbol{v}\}$;
  end
end
for $\ell + 1 \leq k < t$ **do**
  $D \leftarrow$ `CVA`$(\boldsymbol{b_k}, \Lambda, \ell)$;
  for $\boldsymbol{v} \in D$ **do**
    $\boldsymbol{v} \leftarrow \boldsymbol{b_k} - \boldsymbol{v}$;
    if $\boldsymbol{v}[k] > 0$ **then**
      $S[k] \leftarrow S[k] \cup \{\boldsymbol{v}\}$;
      if $\boldsymbol{v}$ *is a k–nearly-transition-vector* **then** $T[k] \leftarrow T[k] \cup \{\boldsymbol{v}\}$;
    end
  end
end
for $0 \leq k < t$ **do** sort by increasing k coordinate $S[k]$ and $T[k]$;
return (T, S) ;

As in the generation of the nearly-transition-vectors for the `globalntvgen`, we need a set $\mathcal{A} = [A_0^m, A_0^M[\times[A_1^m, A_1^M[\times\cdots\times[A_\ell^m, A_\ell^M[$ and the function `index` to perform Function `findV2`.

Function `findV2` implies a modification of the functions that try to find a new nearly-transition-vector, that is Function `fbAdd2` and Function `fbSub2`. These two functions are called by Function `add2` and Function `sub2` only when $k > \ell$ or $k = t - 1$. The function `CVA` is the same as the one described in the `globalntvgen`. The sieve function for the `localntvgen` is the same as the one of the `globalntvgen` and is written in Appendix E.2.

Before giving some differences between the two algorithms, we describe two specializations of our enumeration algorithms, that are the generalized line and plane sieves. In the case of the generalized line sieve, both enumeration algorithms are the same and the initial generation is given in the next section. In the case of the plane sieve, the algorithm to generate the nearly-transition-vectors is a specialization of the one given for the `localntvgen`. It is maybe possible to modify the generation of nearly-transition-vectors for the `globalntvgen` (a

Function `fbAdd2`(k, $\boldsymbol{c}$, $\mathcal{H}$, S, Λ, ℓ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the level ℓ

output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

while $\boldsymbol{c}[k] < H_k^M$ **do**
  $L \leftarrow \emptyset$;
  for $\boldsymbol{v} \in S[k]$ **do**
    if $\ell = t - 1$ **then**
      $D \leftarrow \texttt{CVA}(\boldsymbol{c} + \boldsymbol{v}, \Lambda, t-2)$;
    else
      $D \leftarrow \texttt{CVA}(\boldsymbol{c} + \boldsymbol{v}, \Lambda, \ell)$;
    end
    for $\boldsymbol{d} \in D$ **do**
      if $\boldsymbol{c} + \boldsymbol{v} - \boldsymbol{d} \in \mathcal{H}$ **then return** $\boldsymbol{c} + \boldsymbol{v} - \boldsymbol{d}$;
      $L \leftarrow L \cup \{\boldsymbol{c} + \boldsymbol{v} - \boldsymbol{d}\}$;
    end
  end
  set $\boldsymbol{c}$ to an element of L;
  if $k - 1 > \ell$ **then**
    $\boldsymbol{d} \leftarrow \texttt{fbAdd2}(k-1, \boldsymbol{c}, \mathcal{H}, S, \Lambda, \ell)$; **if** $\boldsymbol{d} \in \mathcal{H}$ **then return** $\boldsymbol{d}$;
    $\boldsymbol{d} \leftarrow \texttt{fbSub2}(k-1, \boldsymbol{c}, \mathcal{H}, S, \Lambda, \ell)$; **if** $\boldsymbol{d} \in \mathcal{H}$ **then return** $\boldsymbol{d}$;
  end
end
return $\boldsymbol{c}$; `// c is out of` $\mathcal{H}$

Function `add2`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ, ℓ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the level ℓ

output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

for $\boldsymbol{v} \in T[k]$ **do**
  if $\boldsymbol{c} + \boldsymbol{v} \in \mathcal{H}_{k-1}$ **then return** $\boldsymbol{c} + \boldsymbol{v}$;
end
if $k > \ell$ *or* $k = t - 1$ **then**
  $\boldsymbol{d} \leftarrow \texttt{fbAdd2}(k, \boldsymbol{c}, \mathcal{H}, S, \Lambda, \ell)$; $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{d}$;
  if $\boldsymbol{c} \in \mathcal{H}$ **then** $T[k] \leftarrow T[k] \cup \{\boldsymbol{d} - \boldsymbol{c}\}$; $S[k] \leftarrow S[k] \cup \{\boldsymbol{d} - \boldsymbol{c}\}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{d}$;
else
  $\boldsymbol{c} \leftarrow (H_0^M, H_1^M, \ldots, H_{t-1}^M)$; `// c is out of` $\mathcal{H}$
end
return $\boldsymbol{c}$;

way will be to use Remark 6.8) to have, as described below, an algorithm that enumerates all the elements in the intersection of the lattice and the sieving region, but we do not discuss it.

6.4.4 Generalized line and plane sieves

There exist two particular cases that allow to get an algorithm which guarantees that all the elements in the intersection of the lattice and the sieving region are reported.

Generalized line sieve

When $\ell = 0$, it means that for all dimensions, cuboids contain on average more than one element, the sieve algorithm is equivalent to a generalized line sieve, see also the thesis of Zajac [185, Section 7] for another description. For the line sieve, we know that the suitable sets of transition-vectors, and necessarily the suitable sets of nearly-transition-vectors, is well defined. We do not perform the algorithm to produce new possible nearly-transition-vectors. If the following vectors fit in the sieving region, there are k–transition-vectors:

- the suitable set of 0–transition-vector is necessarily equal to $\{\boldsymbol{b_0}\}$.
- the suitable set of 1–transition-vectors is a subset of $\{\boldsymbol{b_1}, \boldsymbol{b_1} - \boldsymbol{b_0}\}$.
- the suitable set of 2–transition-vectors is a subset of $\{\boldsymbol{b_2}, \boldsymbol{b_2} - \boldsymbol{b_0}\}$.
- ...
- the suitable set of $(t-1)$–transition-vectors is a subset of $\{\boldsymbol{b_{t-1}}, \boldsymbol{b_{t-1}} - \boldsymbol{b_0}\}$.

It is the only case where we know how to compute all the suitable sets of transition-vectors. It is also the only time where there exists a 0–transition-vector.

Generalized plane sieve

Let us assume that I_0 is smaller than r. In this case, we can perform Algorithm `reduce-qlattice` to compute the reduced basis $\{\boldsymbol{u}, \boldsymbol{v}\}$ that gives the three possible 1–transition-vectors. It is not possible to guarantee that we can produce k–transition-vectors, where k is in $[2, t[$. We use the `localntvgen` to generate nearly-transition-vectors. It is indeed easier to solve the closest vector problem in two dimensions than in larger dimensions. We have therefore:

- the suitable set of 1–transition-vectors is a subset of $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{u} + \boldsymbol{v}\}$.
- some 2–nearly-transition-vectors can be computed by minimizing the coefficients of $\boldsymbol{b_2}$ using linear combinations of $\boldsymbol{u}$ and $\boldsymbol{v}$.
- some 3–nearly-transition-vectors can be computed by minimizing the coefficients of $\boldsymbol{b_3}$ using linear combinations of $\boldsymbol{u}$ and $\boldsymbol{v}$.
- ...

- some $(t-1)$–nearly-transition-vectors can be computed by minimizing the coefficients of $\boldsymbol{b_{t-1}}$ using linear combinations of $\boldsymbol{u}$ and $\boldsymbol{v}$.

To be sure that we do not miss any elements, if we need to compute nearly-transition-vectors on the fly, we must use only k–skew-small-vectors with a coordinate k equal to one. The minimization of the coordinates of a vector out of the sieving region must be done using only $\boldsymbol{u}$ and $\boldsymbol{v}$. Instead of using the function `CVA` in Function `fbAdd2` and Function `fbSub2`, we can use another approach. From an element $\boldsymbol{c}$ of the lattice Λ out of the extended sieving region $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\mathbb{Z}^{t-2}$, we can try to find an element $\boldsymbol{c_n} = (\cdot, \cdot, \boldsymbol{c}[2], \boldsymbol{c}[3], \ldots, \boldsymbol{c}[t-1])$ by using $\boldsymbol{u}$ or $\boldsymbol{v}$ to have the first coordinate of $\boldsymbol{c_n}$ in $[H_0^m, H_0^M[$ and after that, using the enumeration of Franke–Kleinjung to have the second coordinate in $[H_1^m, H_1^M[$. If it is not possible, the third to the coordinate $t-2$ must be modified using the skew-small-vectors according to Function `fbAdd2` and Function `fbSub2` to reach a new element in the intersection of Λ and $\mathcal{H}$.

6.4.5 Differences

The case $\ell = 0$ and $\ell = 1$ are treated before, so we only consider the cases where the level $\ell > 1$. In this section, we list some advantages and drawbacks of the two algorithms: we first discuss some theoretical features of the algorithms, and verify these assertions practically, trying also to avoid or reduce the impacts of the drawbacks, when possible.

Concerning the algorithms

We summarize in Table 6.4 some advantages and drawbacks of the two algorithms. In the rest of this section, we give some justifications of this classification.

	Advantages	Drawbacks
`globalntvgen`	• nearly-transition-vectors with small coordinates	• always fall-back • less provable
`localntvgen`	• less fall-back • more provable • `CVA` can be processed with a precomputation step	• nearly-transition-vectors with possibly larger coordinates • fall-back strategy can be long because of recursive calls

Table 6.4 – Some advantages and drawbacks of the two enumeration algorithms.

About the nearly-transition-vectors. The way to generate the nearly-transition-vectors is the main difference between the two algorithms. In Table 6.4, we claim that the coefficients of the nearly-transition-vectors have

small coordinates with the `globalntvgen`, and possibly large ones with the `localntvgen`. In fact, we want to say that the coordinates of the skew-small-vectors produced after the skew basis reduction with the `globalntvgen` are always smaller or equal to the skew-small-vectors produced after the skew basis reduction performed with the `localntvgen`: indeed, because the skew basis reduction with the `globalntvgen` is performed on the whole basis, contrary to the `localntvgen` (except when $\ell = t - 1$), the coefficients of the output vectors are necessarily smaller or equal to the one produced by the skew lattice reduction performed with the `localntvgen`, dealing with a subset of the basis vectors. The reduction of the remaining vectors in the `localntvgen` is once again less aggressive than with the `globalntvgen`.

Fall-back strategies. Highly dependent on the way to generate the nearly-transition-vectors, the fall-back strategies are different in the two algorithms. In the `localntvgen`, we assume that we have found at least suitable k–nearly-transition-vectors, where $k \in [0, \ell]$, a fact that cannot be proved. We therefore need to perform sufficiently many linear combinations, to ensure to discover all the possible nearly-transition-vectors. But, if it is possible to increase specifically the number of k–skew-small-vectors, where $k \in [0, \ell]$, in the `localntvgen`, it is not possible to do the same for the `globalntvgen`: this is why, each time we fail to leave the addition or subtraction of a nearly-transition-vector in the `globalntvgen`, we need to call the fall-back strategy. As this strategy modifies, at a depth k, the k first coordinates, we just need one call to the fall-back strategy to try to find, or not, a new element. Therefore, the fall-back strategy for the `globalntvgen` is often called.

Concerning the `localntvgen`, the call to the fall-back strategy is less frequent: indeed, if no k–nearly-transition-vector, where $k \leq \ell$, can be added or subtracted, we consider that it is not possible, at this depth of the sieving step, to find a new element, and then this depth is ignored. However, when $k - 1 > \ell$, the k–skew-small-vectors modify the ℓ first coordinates and the coordinate k, but not the others. We therefore need to use the fall-back strategy to modify this non-impacted coordinates, if we do not find a new element in the intersection of the lattice and the sieving region. This strategy can therefore be more costly than the one for the `globalntvgen`.

Provability. With the general line and plane sieves, we can prove that these two algorithms enumerate all the elements in the intersection of the lattice and the sieving region. These two algorithms are modifications of the `localntvgen`. With the specific shape of the k–nearly-transition-vectors and k–skew-small-vectors, that is a 1 for the coordinate k when $k > \ell$, we can have more guarantees that we enumerate all the elements than with the `globalntvgen`.

`CVA`. All the calls to the `CVA` function in the `localntvgen` are performed with the input $(\cdot, \Lambda, \ell)$. It is possible to compute only once an interesting representation of the lattice $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell}\}$ to solve the closest vector problem and, for each call to `CVA`, solve this problem by computing a matrix-vector product, using an idea close to the Babai's rounding technique [10]. To produce more interesting vectors, we do not only return the closest vector, but the vectors

that form the fundamental domain around $\boldsymbol{c}$ (it contains necessarily the closest vector).

Practical results

With the generalized line and plane sieves that corresponds to the case $\ell = 0$ and $\ell = 1$, we have two specific algorithms to enumerate efficiently and completely the elements of the intersection between the lattice and the sieving region, we then stick to level higher than or equal to 2. We perform our experiments with a 4-dimensional sieving ($t = 4$) where the level are equal to 2 and 3. On each level, we sample 1,000 random lattices and perform the enumeration of the elements of these lattices that are in the sieving region $\mathcal{H} = [-2^5, 2^5[\times[-2^5, 2^5[\times[-2^5, 2^5[\times[0, 2^5[$ that contains 2^{23} elements. We choose to generate almost the same number of vectors in the two algorithms, which is essentially dependent on the set $\mathcal{A}$ to perform the small linear combination. When $\ell = 2$, the set $\mathcal{A}$ is equal to $[-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$ for the `globalntvgen` and $[-2, 2] \times [-2, 2] \times [-1, 1]$ for the `localntvgen`. When $\ell = 3$, the set $\mathcal{A}$ is equal to $[-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$ for both algorithms. We summarize the results in Table 6.5 and Table 6.6.

	`globalntvgen`			`localntvgen`		
	min	mean	max	min	mean	max
Number of skew-small-vectors		40			41	
Number of nearly-transition-vectors at the beginning	4	**16.5**	33	1	12.3	24
Number of nearly-transition-vectors at the end	6	**18**	33	4	14	24
Expected number of elements	32	156.5	1978	32	156.5	1978
Number of elements	18	**149.7**	1983	18	139.7	1983
Number of call to the fall-back strategy	1	269.5	2854	0	**1.7**	8

Table 6.5 – Experiments at level 2.

	`globalntvgen`			`localntvgen`		
	min	mean	max	min	mean	max
Number of skew-small-vectors		40			40	
Number of nearly-transition-vectors at the beginning	1	**9.9**	26	1	**9.9**	26
Number of nearly-transition-vectors at the end	1	**10.2**	26	1	**10.2**	26
Expected number of elements	1	3.9	31	1	3.9	31
Number of elements	1	**4.8**	47	1	**4.8**	47
Number of call to the fall-back strategy	3	18	113	0	**0.8**	4

Table 6.6 – Experiments at level 3.

Which algorithm to choose?

The results of our simple Sage implementations seem to confirm some of the advantages and drawbacks we have anticipated. These implementations are available in CADO-NFS [176].

The `globalntvgen`. An unexpected advantage of this algorithm, probably due to the small coordinates of the nearly-transition-vectors, is the number of enumerated elements. This number is not, or for a tiny part, increased by the large number of call to the fall-back strategy, which is the major drawback highlighted by the practical experiment. A more careful analysis of the algorithm, especially the test to perform or not the fall-back strategy, is maybe a way to make this algorithm practicable. It seems too early to avoid to perform any fall-back strategy.

The `localntvgen`. A first good point of this algorithm is that the number of calls to the fall-back strategy is very low, even if the strategy seems to possibly use it a lot. It produces often a new useful nearly-transition-vector. The number of elements enumerated is a little bit less than with the previous algorithm, meaning probably that the generation of nearly-transition-vectors and skew-small-vectors may not be fully optimized, even if it is closer to the `globalntvgen`.

At this point, and given the description of the algorithms, the `localntvgen` seems to be the most efficient, because of the possible preprocessing for the computation of the closest vectors, and the small number of calls to the fall-back strategy. The number of enumerated elements is more than satisfying. However, the practical results do not exclude the `globalntvgen`, if we ignore the fall-back strategy we have proposed.

Implementation of the algorithms. Concerning the implementation of the line and plane sieves, it seems better to have a specific implementation: indeed, the algorithms can add some useful nearly-transition-vectors, but may be used in a suboptimal context, that is use a nearly-transition-vector where a smaller not already known nearly-transition-vector must be discovered. For these reasons, a specific implementation is required, maybe one for the two sieve algorithms if it is possible to merge the algorithms without impacting the running time and the correctness of the results.

Concerning the sieve for a higher level, the situation is not as clear as in the previous situation and the implementation choice will be conclusive with a real implementation, and not a prototype in Sage. We discuss one choice: a specific implementation given ℓ, or a general implementation (in other words, should we unroll the recursive calls or not).

Starting with the first remark of this paragraph, it seems that we need to have a specific implementation for each ℓ. It allows indeed more control on what happens during a recursive call, especially if some patterns can be observed and detected to have an expected and predictable behavior. We know that, if very small nearly-transition-vectors are involved in the skew basis reduction, it will be difficult to generate new nearly-transition-vectors on the fly, and we can then discard this step of the algorithms. Except this pattern, we do not find other predictable behavior and the conclusion we did about this pattern,

that is avoiding the generation of nearly-transition-vectors on the fly, must be done in each recursive call, and not for a specific one. Therefore, a generic implementation of the algorithms seems to be the best solution, at least in first approximation.

6.4.6 The 3-dimensional case

In this section, we rewrite the algorithms given in the article coauthored with Gaudry and Videau [72] to show that they are in adequacy with the general algorithms described previously, especially the `localntvgen`. It is obvious that the line sieve follows the general algorithm, so we stick to the plane and space sieves. Here, the dimension t is equal to 3, and the integer ℓ is equal to 0 in the case of the line sieve, 1 for the plane sieve and 2 for the space sieve.

Plane sieve

We recall here briefly the different steps of the plane sieve. The plane sieve follows the remarks given in the particular case $\ell = 1$ written in the previous section.

Initialization. In this section, we report the different steps of the initialization. We keep the order of the description given in the general algorithm and link it to the different items of [72]. As we show in Section 6.2.2, the Franke–Kleinjung algorithm conditions produce, given $\boldsymbol{b_0}$ and $\boldsymbol{b_1}$, two skew vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ with weight $(I_0, r/I_0, 1, 1, \ldots, 1)$: this is what is performed in Item 2 of the initialization of the plane sieve. Item 1 is the one that select $\boldsymbol{b_2}$ to be the vector whose coefficients are minimized using $\boldsymbol{u}$ and $\boldsymbol{v}$. A common way to minimize these coefficients is to compute a closest vector of $\boldsymbol{b_2}$, that is a linear combination of $\boldsymbol{u}$ and $\boldsymbol{v}$, and subtract it to $\boldsymbol{b_2}$, as mentioned in Item 3, resulting in a possible 2–nearly-transition-vector, at least a 2–skew-small-vector, with the last coordinate equal to 1. We increase the number of possible 2–nearly-transition-vectors by adding or subtracting $\boldsymbol{u}$ and $\boldsymbol{v}$ by Item 4. Then Item 5 returns possible 2–nearly-transition-vectors.

Enumeration. The enumeration algorithm is the same as the one in the description of the generalized plane sieve in Section 6.4.4.

Space sieve

We use the space sieve when $\ell = 2$. The notion of transition-vectors in [72] is not the same as the one in this document, so we keep the definition of this chapter, keeping in mind that transition-vectors in [72] are exactly 2–nearly-transition-vectors. To identify the different steps described in [72, page 342], we define paragraph 1 as the first paragraph after [72, Lemma 6].

Initialization. The initialization corresponds to the second paragraph, written after [72, Lemma 6]. The weighted LLL shares the same weights as the skew lattice reduction performed during the generation of nearly-transition-vectors using the `globalntvgen` and `localntvgen`. In [72, Algorithm 3], the list L_z

is split into lists: the list $L_{z=0}$ contains 1–nearly-transition-vectors and the list $L_{z\neq 0}$ contains 2–nearly-transition-vectors.

Enumeration. The enumeration process begins with $\mathbf{0}$. We note that, in the space sieve, the sieving region bounds the third coordinate in $[0, H_2^M[$, so the enumeration is done only on ascending third coordinate. In the while loop of [72, Algorithm 3], the first for loop tries to enumerate elements in a plane, this is what we do when we call the sieve algorithm recursively. The second for loop tries to add a 2–nearly-transition-vector. If it cannot find a valid new element in the intersection of Λ and $\mathcal{H}$, the plane sieve is called on the last valid point. This is what we do, but, in order to avoid to enumerate the plane twice, we begin the plane sieve with a point in Λ out of $\mathcal{H}$ in a plane above.

Another approach. Instead of trying to precompute some nearly-transition-vectors and skew-small-vectors, Hayasaka, Aoki, Kobayashi and Takagi propose in [94] a way to find an adapted basis in the 3-dimensional case, allowing them to find all the transition-vectors by doing positive linear combinations of the three basis vectors. Using an appropriate algorithm to compute on the fly the transition-vector allows to enumerate the elements in the intersection of the plane and the sieving region. We were unfortunately not able to reproduce the algorithm to compute the adapted basis or the enumeration algorithm. We believe that computing a small subset of possible transition-vectors is better than always computing the transition-vectors on the fly, but this precomputation is better if the basis is well adapted, as it will be with [94, Algorithm 1].

Part III

Practical results

Chapter 7

Implementation

The relation collection is divided in 3 main tasks, where the sieving algorithms described in Chapter 6 are one of the main step. A full implementation of the relation collection for the high degree variant of NFS, containing these sieving algorithms, have been implemented and integrated in the CADO-NFS software [176]. Even if the sieving algorithms are the main purpose of this thesis, the other steps must be implemented carefully to have a correct running time.

The only public available implementation of a relation collection for the high degree variant is the one of Zajac [185]. It contains an implementation of the line sieve for the three-dimensional case. The implementation we propose offer the way to deal with the special-$\mathfrak{Q}$-method, to sieve with the line, plane and space sieves, as presented in Section 6.4.6, and to use a Galois action of order 6. In the following, we will describe and explain our implementation choices. This chapter can be viewed as a documentation of some parts of our code and justification of our choices. We note that, in our description, the basis vectors of a lattice are rows of a matrix, but, in our C implementation, this is the columns of a matrix that form the basis of a lattice.

We recall first the parameters used to perform the relation collection, using a sieving step and the special-$\mathfrak{Q}$-method to accelerate the computation. Let $t-1$ be the degree of the polynomial a used to find a relation. The set of valid coefficients of a polynomial a is bounded by a t-searching space $\mathcal{S}$, but because of the large number of polynomials that have coefficients in $\mathcal{S}$, we divide $\mathcal{S}$ using the special-$\mathfrak{Q}$-method: if $M_{\mathfrak{Q}}$ is the matrix whose rows are the reduced basis vectors of the special-$\mathfrak{Q}$-lattice, a valid polynomial a has coefficients equal to $\boldsymbol{c}M_{\mathfrak{Q}}$, where $\boldsymbol{c}$ is an element of a t-sieving region $\mathcal{H}$, containing a constant number of elements, largely smaller than the number of elements contained in $\mathcal{S}$. For each norm on side 0 of a polynomial $a = \boldsymbol{c}M_{\mathfrak{Q}}$, we remove by sieving the factors smaller than the sieving bound b_0 in each norm, and for each norm below a threshold T_0, we keep the polynomial a of the corresponding norm, because it has a high probability to be B_0-smooth. We apply the same thing on side 1 and if a polynomial a is reported two times, we perform the complete factorization of both norms. If the norms are doubly-smooth, the polynomial a gives a relation. Let us give a bird's-eye view of the different steps of the algorithm we have implemented: we set the special-$\mathfrak{Q}$ on side 0, but it can be

set on side 1.

- For all the special-$\mathfrak{Q}$ whose norm is in $]b_0, B_0[$
  1. build the matrix $M_{\mathfrak{Q}}$ that represents the special-$\mathfrak{Q}$-lattice
  2. for all sides i:
     (a) initialize the norms on side i and if $i = 0$, remove from each norm the norm of the special-$\mathfrak{Q}$.
     (b) for all ideals on side i of norm smaller than b_i, use the sieving algorithm dedicated for the size of the ideals.
     (c) for all $\boldsymbol{c}$ in $\mathcal{H}$, if $a = \boldsymbol{c}M_{\mathfrak{Q}}$ have a resulting norm smaller than T_i, store the polynomial a in an array A_i.
  3. if the polynomial a is in A_0 and A_1
     (a) compute the norms of a on both sides and test it for smoothness: if the norms are doubly-smooth, report a and the factorization of the norms in both sides
     (b) if a Galois action is enforced, use the Galois action on a and report it.

In this description, we can justify a first implementing choice, the use of the array A_i. To store the norms of all the elements, we can store the norms of a on each side. However, only a few polynomials after the sieving step have a resulting norm less than the threshold of the corresponding side. Then, storing only the interesting information (a has a good chance to have a smooth norm), allows us to reduce drastically the memory footprint: it allows us to reduce by a factor almost 2 the memory footprint instead of using the basic strategy of storing the norms on both sides. Let us now go deeper into details of some important steps of the algorithms.

7.1 Initialization of norms

Let consider that the matrix $M_{\mathfrak{Q}}$ is given, see Section 7.2.1 for more information. Basically, the number of elements in $\mathcal{H}$ is large, around 2^{30} for the largest computation. We discuss here two important elements: in which data structure we store the norms in a first part, and how we compute the norms in a second part.

7.1.1 Storage

Data structure

The sieving region $\mathcal{H}$ is a t-sieving region. Each polynomial a is linked to a vector $\boldsymbol{c}$ in $\mathcal{H}$, we therefore index each cell of the array that store the norms by the coordinate of the corresponding vector $\boldsymbol{c}$. A first way to store the norm is to use a t-dimensional array A: we access the norm of $a = \boldsymbol{c}M_{\mathfrak{Q}}$ by looking at $A[\boldsymbol{c}[0]][\boldsymbol{c}[1]] \dots [\boldsymbol{c}[t-1]]$. To access one element, we need to query t arrays, by looking first in $A[\boldsymbol{c}[0]]$, then to the index $\boldsymbol{c}[1]$ of the previous array and so on to the index $(\boldsymbol{c}[t-1]-1)$ of the last array. These multiple indirections can slow

down the implementation, since we need to have around $(H_0^M - H_0^m)(H_1^M - H_1^m)\cdots(H_{t-1}^M - H_{t-1}^m)\sum_{p \text{ primes}}^{b_0} 1/p$ access to indices of the array A.

To avoid this problem of indirections, we store the $(H_0^M - H_0^m)(H_1^M - H_1^m)\cdots(H_{t-1}^M - H_{t-1}^m)$ elements of A in a one dimensional array. The norm of $a = \boldsymbol{c}M_\Omega$ is stored at index $(\boldsymbol{c}[0] - H_0^m) + (\boldsymbol{c}[1] - H_1^m)(H_0^M - H_0^m) + \ldots + (\boldsymbol{c}[t-1] - H_{t-1}^m)\prod_{k=0}^{t-2}(H_k^M - H_k^m)$. From an index in the one dimensional array, we can also compute the corresponding vector $\boldsymbol{c}$. Choosing $H_i^M - H_i^m$ to be a power of two can accelerate the computations by using bit shifting.

Storing one norms

The representation in memory of one norm is very important. Indeed, the basic choice is to store the exact result of a norm. But the norms we need to store are not integers of size 64 bits, or 128 bits (modern compiler allows us to use natively integers of size 128 bits). Storing the exact norm implies using `mpz_t`, which can have a prohibitive cost for the memory requirement. Let consider that the sizes of the norms are smaller than 2^{256}, which is the order of magnitude of what we require to store the norms given by the conjugation polynomial selection, see [72, Table 2]. Storing an integer of this size using a `mpz_t` require around 56 bytes.

A way to reduce the memory footprint is to store the logarithm of the norm, traditionally on a `char`. In the implementation of the relation collection with a two-dimensional relation collection in CADO-NFS [176], the basis is chosen to use all the bits of a `char`: in our implementation, it is possible to specify a basis, but we do not take care of fitting exactly in one byte. It remains to validate that storing the logarithm is sufficient for our purpose. During the sieving step, if we store the value of the norm, we will divide the norm by each factor found during the sieving step. This can be easily transform by subtracting from the logarithm of the norm the logarithm of the removed factor. The comparison with the threshold can also be done by comparing with the logarithm of the threshold. We can wonder if there is a loss of precision by using: there is in fact a loss of precision, because using a `char` will discard the fractional part of the logarithm. This problem can be partially avoided by using an appropriate threshold that take into account this imprecision.

7.1.2 Algorithm to initialize the norms

We have described how we store the norms for one side. Computing the norm of a on one side, say 0, is almost equivalent to computing the resultant between a and f_0, using the algorithm given in Appendix F. We recall that $\mathcal{H}$ contains between 2^{20} to 2^{30} elements. Computing the resultant for each polynomial a, whose coefficients are given by $\boldsymbol{c}M_\Omega$, is the first approach we can have: indeed, it is the most accurate method. However, it is often difficult to combine the best accuracy and the best running time. Indeed, computing 2^{20} to 2^{30} resultants is expensive. One other method is to use an upper bound on the resultant, typically the one given by Bistritz–Lifshitz in [34]. This method has the advantage to be the evaluation of a simple function, which can be accelerated by a few precomputations. Intermediately, we can compute the resultant by discarding a little bit of accuracy, using floating point computation on `double`, instead of preserving the whole precision by using `mpz_t`.

The main problem of all these approaches is that we need to compute for each cell of the array in which we store the norms the corresponding resultant, or an approximation of it, which represents too many computations. In the following, we will use a property of the resultant to speed up the computation.

Using the continuity of the resultant

The resultant can be viewed as the determinant of the Sylvester matrix formed by the coefficients of f_0 and a. The resultant is therefore a polynomial in the coefficients of a: the resultant is continuous and infinitely differentiable. This means that there exists a neighborhood of a where all the polynomials a' have the same size of norm, in base 2 for example. Using this idea, by finding some areas where the norm of the polynomials a have the same size, we can set to each of the polynomials the same value. It allows us to not compute the norm of all the polynomials a, but just one typical norm in the neighborhood. But, when we consider the special-$\mathfrak{Q}$-lattice, the location of two *contiguous* polynomials a can differ from a factor, maybe too large to have two contiguous polynomials a with the same size of norms. We will show that this factor is not so large, and allows us to use the continuity of the resultant, even in the special-$\mathfrak{Q}$-lattice.

Let consider the bound on the resultant between a and f_0 given in [34]: if we discard the factor depending only on the degrees of the polynomial, the logarithm of the resultant between a and f_0 is close to $\deg f_0 \log \|a\|_\infty + \deg a \log \|f_0\|_\infty$, where $\deg a = t-1$ in a vast majority of case. An upper bound of the infinity norm of $a = \boldsymbol{c}M_{\mathfrak{Q}}$ is given by $\|a\|_\infty = t\|\boldsymbol{c}\|_\infty Q^{1/t}$, where Q is the size of the special-$\mathfrak{Q}$. By considering the logarithm of this bound, we get $\log t + 1/t \log Q + \log \|\boldsymbol{c}\|_\infty$: $\log \|\boldsymbol{c}\|_\infty$ does not vary too much when $\|\boldsymbol{c}\|_\infty$ has not-too-large variation around a given value. Therefore, there exists a neighborhood $\mathcal{C}$ of $\boldsymbol{c}$ where the polynomials $a = \boldsymbol{c}M_{\mathfrak{Q}}$ have almost the same size of norms.

In the following, we will describe how to initialize the norms by searching a neighborhood of polynomials $a = \boldsymbol{c}M_{\mathfrak{Q}}$ which have the same size of norm by looking for portions, that are cuboids, of $\mathcal{H}$ with this property. We use the following recursive procedure, with $\mathcal{H}$ as the first given cuboid:

1. Compute the norm of the $2^t + 1$ polynomials $a = \boldsymbol{c}M_{\mathfrak{Q}}$, where $\boldsymbol{c}$ are the coordinates of the 2^t vertices of the cuboid and the coordinates of a random element inside the cuboid.

2. If the size of the norms are almost the same, say N, then set to N the norms of all the elements in this cuboid.

3. Othewise, divide the input cuboid in 2^t equal cuboids, and call this procedure recursively on each of them.

Experimental results

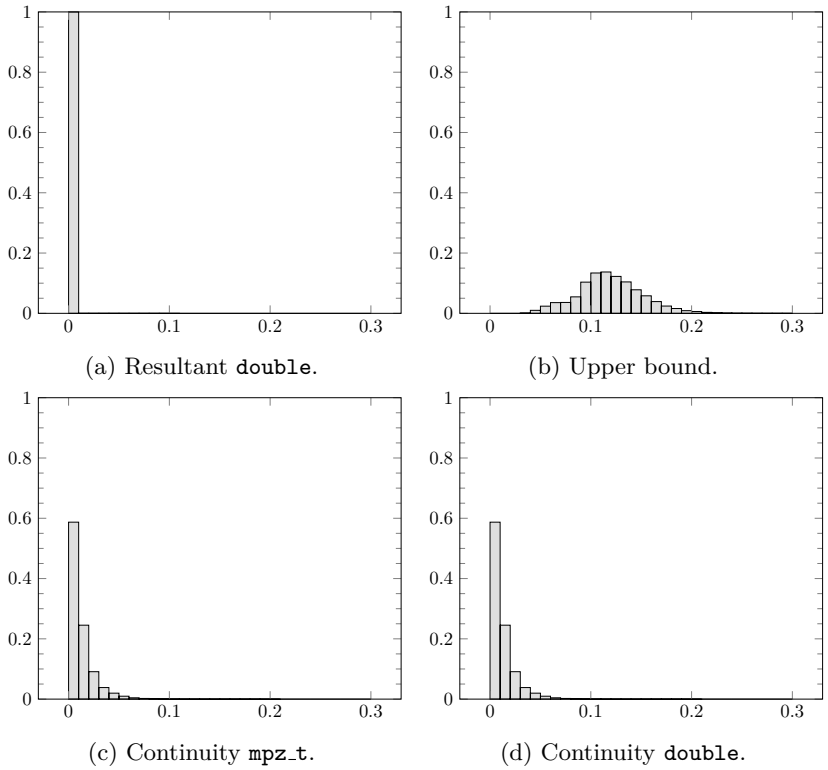
In this section, we try to validate the approach given previously to initialize the norms, by using the computation of the resultant with a floating point precision. We give the timing and the accuracy of all the methods described previously.

The average timing is found by using on side 1 the 991 first special-$\mathfrak{Q}$ (one per orbit of a Galois action of order 6) whose norm is largest than 2^{20} on a

sieving region $\mathcal{H} = [-2^7, 2^7[\times[-2^7, 2^7[\times[0, 2^7[$ using the polynomial designed to perform the relation collection step for an $\mathbb{F}_{p^6}^*$ of 300-bit size, see Section 8.2.3. We use an `Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz` and compile the code with `gcc (Debian 6.3.0-6) 6.3.0 20170205`. We count together the time to initialize both sides, because they have no reason to take advantage of the presence of the special-$\mathfrak{Q}$ or not, and in fact, have the same magnitude order. The timings are reported in Table 7.1. To compare the accuracy of our different initialization algorithms, we compute the relative error of the initialization of norms for 100 special-$\mathfrak{Q}$s distributed on the range of special-$\mathfrak{Q}$s used in our computations. The results are summarized on Figure 7.1. For the algorithm we used in our computations, called "Continuity `double`", we have a sufficient accuracy and the best timing. This is therefore the algorithm that is used by default in our implementation.

	Resultant		Upper bound	Continuity	
	`mpz_t`	`double`		`mpz_t`	`double`
Average timing (s)	86.8	18.2	1.29	0.890	0.251

Table 7.1 – Average timing for different initialization algorithms.



(a) Resultant `double`. (b) Upper bound.

(c) Continuity `mpz_t`. (d) Continuity `double`.

Figure 7.1 – Proportion of norms having the same relative error.

7.2 Sieving

Once the norms are initialized, the sieving step is performed: it consists to remove in the norms the contribution of the norm of the ideals that are involved in the factorization in ideals. The lattice described by the matrix in Equation (4.7) allows us to enumerate the elements $\boldsymbol{c}$ of the sieving region $\mathcal{H}$ such that $\mathfrak{Q}$ and $\mathfrak{R}$ are involved in the factorization of $a(\theta_0)$, where the coefficients of a are given by $\boldsymbol{c}M_{\mathfrak{Q}} = \boldsymbol{e}M_{\mathfrak{Q},\mathfrak{R}}M_{\mathfrak{Q}}$, where $\boldsymbol{e}$ are in $\mathbb{Z}^t$. We first described the preliminaries steps of the sieving: building what we call *sieving basis*, that is the set of all the ideals to be considered in the sieving step, building the matrices $M_{\mathfrak{Q}}$ and $M_{\mathfrak{Q},\mathfrak{R}}$ and finally, how we select the special-$\mathfrak{Q}$ we use in the orbit of a Galois action. We then described our choice for the implementation of the sieving algorithm, and gives some benchmarks, essentially for the space sieve.

Remark 7.1. Because the ideal of degree 1 are more numerous than the one of larger inertia degree, we only deal with ideals $\mathfrak{R}$ of inertia degree 1. However, dealing with special-$\mathfrak{Q}$s of larger inertia degree is not a problem, even if it is not useful for the relation collection, due to the too large norms it implies.

7.2.1 Dealing with ideals

Building the sieving basis

The sieving basis contain the ideals to be sieved. To build the sieving base on side 0, we use the procedure:

- For all prime r less than the sieving bound b_0
  1. If r divides the leading coefficient of f_0, the ideal (r, x) is a projective ideal.
  2. For all the simple factor h of f_0 modulo r
     (a) if the degree of h is 1, the ideal (r, h) is of inertia degree 1.
     (b) else, if $\deg h < t$, the ideal (r, h) is of inertia degree $\deg h$.

We distinguish the ideal of inertia degree equal to 1 and of larger inertia degree, because their representation are different in our implementation, even if we do not sieve the ideals of inertia degree larger than 1. To store an ideal, we store these four informations, to identify an ideal $\mathfrak{R} = (r, h)$:

- the prime r;
- the polynomial h, a factor of f_0 modulo r;
- the block matrix $T_{\mathfrak{R}}$ of size $(\deg h) \times (t - 1 - \deg h)$, see Equation (4.5);
- the logarithm in base 2 of $r^{\deg h}$.

The main difference between an ideal of inertia degree equal to 1 and of larger inertia degree is the way to store the block matrix $T_{\mathfrak{R}}$: indeed, an array of one dimension is sufficient to describe $T_{\mathfrak{R}}$ when the degree of h is 1, and a two-dimensional array is necessary when h have a larger degree.

Building the matrices of the ideals

In a special-$\mathfrak{Q}$-lattice, we need to build the lattice whose elements a involves the ideal $\mathfrak{R}$ of inertia degree equal to 1 in $a(\theta_0)$: this lattice have as basis vectors the rows of the matrix given in Equation (4.7). To build this matrix, we look for vectors verifying the Equation (4.6). Let $T_{\mathfrak{Q},\mathfrak{R}}$ be equal to $M^0_{\mathfrak{Q}} - M^1_{\mathfrak{Q}} T_{\mathfrak{R}} \bmod r$: we require that the first non-zero coefficient of $T_{\mathfrak{Q},\mathfrak{R}}$ is equal to 1. It is always possible by multiplying each coefficients of $T_{\mathfrak{Q},\mathfrak{R}}$ by the inverse of the first non-zero coefficient of $M^0_{\mathfrak{Q}} - M^1_{\mathfrak{Q}} T_{\mathfrak{R}}$. If $T_{\mathfrak{Q},\mathfrak{R}}$ has a 1 as its first coefficient, a basis of the $\mathfrak{R}$-lattice in the $\mathfrak{Q}$-lattice have basis vectors as rows of the matrix $M_{\mathfrak{Q},\mathfrak{R}}$ that has exactly the shape given in Equation (4.7). We assume that we are in this case. The second row of the matrix $M_{\mathfrak{Q},\mathfrak{R}}$ is equal to $(-T_{\mathfrak{Q},\mathfrak{R}}[1] + r, 1, 0, \ldots, 0)$, the third is equal to $(-T_{\mathfrak{Q},\mathfrak{R}}[2] + r, 0, 1, 0, 0, \ldots, 0)$ and so on to the last vector equals to $(-T_{\mathfrak{Q},\mathfrak{R}}[t-1] + r, 0, 0, \ldots, 0, 1)$. The matrix $M_{\mathfrak{Q},\mathfrak{R}}$ is therefore equal to

$$M_{\mathfrak{Q},\mathfrak{R}} = \left(\begin{array}{c|c} r & 0 \\ \hline -T_{\mathfrak{Q},\mathfrak{R}}[i] + r & I_{t-1} \end{array} \right), \tag{7.1}$$

where I_{t-1} is the identity matrix of size $(t-1) \times (t-1)$ and i is an integer in $[1, t[$.

Selecting the best special-$\mathfrak{Q}$

Let the special-$\mathfrak{Q}$s be set on side 0. When a Galois action of order k can be applied on both sides, there exist k conjugate special-$\mathfrak{Q}$s that have exactly the same norm. From the description on how to deal with a Galois action and the special-$\mathfrak{Q}$ method, see Section 4.1.1, we know that we can use one special-$\mathfrak{Q}$ in each orbit and apply the Galois action on the found relations to recover all the relations in each other special-$\mathfrak{Q}$ in the same orbit. For the k special-$\mathfrak{Q}$s in a same orbit, the portion of the space they covered are different, even if they are conjugate under the Galois action.

We therefore try to find one of the k special-$\mathfrak{Q}$s that covers the most interesting searching space in term of polynomial a. To identify the one which looks the most interesting, we compute for all the special-$\mathfrak{Q}$s of the same orbit, the largest norm reached by one of the vertices of the sieving region $\mathcal{H}$. The special-$\mathfrak{Q}$ that reaches the smallest largest norm is selected as the special-$\mathfrak{Q}$ that will be used as the representative of the orbit, because we can hope that the norms of the elements in the whole sieving region are the smallest. Using the parameters of the computation of the 300 bits $\mathbb{F}_{p^6}$, see Section 8.2.3, for the first 991 orbits of special-$\mathfrak{Q}$s whose norms are larger than 2^{20}, this heuristic selects in 16% of the cases the best special-$\mathfrak{Q}$ (as if we select randomly the special-$\mathfrak{Q}$ in an orbit), and in 0.1% of the cases the worst special-$\mathfrak{Q}$: on average, we get 265 relations and miss 37.5 relations in an orbit.

A way to improve the selection of the best special-$\mathfrak{Q}$ would be to simulate the Murphy E function, by computing some additional norms on the shape of the sieving region $\mathcal{H}$ and computing the sum given in Equation (4.2) with these elements.

7.2.2 The sieving algorithms

All the previous steps have installed the preliminaries of the sieving step: we have initialized the norms of the elements that are in the special-$\mathfrak{Q}$-lattice, we know how to build the $\mathfrak{R}$-lattice inside the special-$\mathfrak{Q}$-lattice. We can now remove the contribution of the ideals of inertia degree equal to 1 in the corresponding norms. Let I_i bet the length of the interval i of a t-sieving region $\mathcal{H}$, for i in $[0, t[$. If the norm r of $\mathfrak{R}$ is less than I_0, we use the line sieve, because the level with respect to $\mathcal{H}$ and the lattice is equal to 0. When r is less $I_0 I_1$, we use the plane sieve, because of a level equal to 1 and otherwise, we use the space sieve (the level is equal to 2).

The implementation we propose of the three sieve algorithms seems to not follow the recursive general algorithms given in Chapter 6, because we have essentially unroll the recursive calls. For the line and plane sieves, we provide an implementation that works in any dimension, as we proposed a general line and plane sieves in Section 6.4.4. However, for the space sieve, we have specify our implementation in three dimensions, as the description in Section 6.4.6. We detail here the implementation choice we did.

The line sieve

Let consider Equation (4.6). We look for all the elements $\boldsymbol{c}$ such that $\boldsymbol{c}T_{\mathfrak{Q},\mathfrak{R}} \equiv 0 \bmod r$. Let i be the index of the first non-zero coordinate of $T_{\mathfrak{Q},\mathfrak{R}}$, which is equal to 1 following what we impose in the preliminaries. We therefore have the equality $\boldsymbol{c}[i] \equiv -\sum_{k=i+1}^{t-1} \boldsymbol{c}[k] T_{\mathfrak{Q},\mathfrak{R}} \bmod r$. As the norm of $\mathfrak{R}$ is less than $H_i^M - H_i^m$, we know that there always exists a solution of this equation such that $\boldsymbol{c}[i]$ is in I_i, given $\boldsymbol{c}[i+1], \boldsymbol{c}[i+2], \ldots, \boldsymbol{c}[t-1]$. To enumerate all the possible $\boldsymbol{c}$ in the intersection of the lattice formed by $M_{\mathfrak{Q},\mathfrak{R}}$ and the sieving region, we performed the following steps:

1. For the $(\prod_{k=i+1}^{t-1} I_k)$ possible values for $(c_{i+1}, c_{i+2}, \ldots, c_{t-1})$ in $[H_{i+1}^m, H_{i+1}^M[\times [H_{i+2}^m, H_{i+2}^M[\times \cdots \times [H_{t-1}^m, H_{t-1}^M[$
   (a) Find a value c_i in $[H_i^m, H_i^M[$, such that $c_i \equiv -\sum_{k=i+1}^{t-1} c_k T_{\mathfrak{Q},\mathfrak{R}} \bmod r$.
   (b) For all the value $c_i + \lambda r$ in $[H_i^m, H_i^M[$, mark in the array of norms that the value at index $(\cdot, \cdot, \cdots, \cdot, c_i + \lambda r, c_{i+1}, \ldots, c_{t-1})$ is divisible by r, where λ is in $\mathbb{Z}$.

For Item 1, we use the following procedure, that is in brief, adding one to the first possible coordinate of $(c_{i+1}, c_{i+2}, \ldots, c_{t-1})$ to stay in $[H_{i+1}^m, H_{i+1}^M[\times [H_{i+2}^m, H_{i+2}^M[\times \cdots \times [H_{t-1}^m, H_{t-1}^M[$.

1. Set k to $i+1$ and increment c_k.
2. While $c_k = H_k^M$
   (a) Set c_k to H_k^m.
   (b) Increment k.
   (c) If $k < t$, increment c_k.
   (d) Otherwise, break the loop.

Typically, the enumeration begins with $(c_{i+1}, c_{i+2}, \ldots, c_{t-1})$ equals to $(H^m_{i+1}, H^m_{i+2}, \ldots, H^m_{t-1})$ and goes to $(H^M_{i+1} - 1, H^M_{i+2} - 1, \ldots, H^M_{t-1} - 1)$, therefore, if we use the previous procedure only the number of required times to enumerate all the needed elements, that is $(\prod_{k=i+1}^{t-1} H^M_k - H^m_k)$, the directive in Item 2c is always performed and so, the one in Item 2d is never performed. A better enumeration will be to use a procedure close to the one to enumerate n-ary Gray code to perform less multiplication and reduction than we need with our proposed line sieve.

When the norm of $\mathfrak{R}$ becomes larger than the length I_0 of the interval $[H^m_0, H^M_0[$, we can not ensure that, for all $(\cdot, \cdot, \cdots, \cdot, c_{i+1}, c_{i+2}, \ldots, c_{t-1})$, there always exists such an element in the sieving region $\mathcal{H}$. This is why we use the plane sieve.

The plane sieve

Preliminaries. Let consider the lattice described by the rows of $M_{\mathfrak{Q},\mathfrak{R}}$ defined in Equation (7.1). This description below follows the initialization procedure of the `localntvgen`. We first look for the 1–nearly-transition-vectors thanks to the Franke–Kleinjung algorithm, and the look for 2–nearly-transition-vectors, which have their coordinate 2 equal to 1 and small coordinates 0 and 1 (thanks to the `CVA` function we explicit in the context of the plane sieve).

The first step of the plane sieve algorithm is to have a pleasant basis of the plane defined by the first two vectors of $M_{\mathfrak{Q},\mathfrak{R}}$: we therefore use a basis reduction, for example Function `reduce-qlattice`, to get $\boldsymbol{u}$ and $\boldsymbol{v}$ that reach the bounds given by Franke and Kleinjung (Proposition 6.2).

Then, we look for vectors allowing us to modify only the first two coordinates and an other coordinate of an element in the lattice to produce an other element in the lattice. Let i be the index of the $(i+1)$th vector $\boldsymbol{w_i}$ of the matrix $M_{\mathfrak{Q},\mathfrak{R}}$. If $i > 1$, the vector $\boldsymbol{w_i}$ have two non-zero values: one at index 0, denoted by R_i and one at index i, equals to 1. We want to reduce the two first coefficients of $\boldsymbol{w_i}$: this is equivalent to remove to $(R, 0)$ its closest vector in the lattice described by $\{(r, 0), (-T_{\mathfrak{Q},\mathfrak{R}}[1] + r, 1)\} = \{(r, 0), (T, 1)\}$. To find the closest vector and some closest vectors to $(R_i, 0)$, we proceed as follow:

1. Let $\boldsymbol{u'}$ and $\boldsymbol{v'}$ be the two vectors output by the Gauss reduction on $\{(r, 0), (T, 1)\}$, or $\{\boldsymbol{u}, \boldsymbol{v}\}$.
2. Let G be the matrix whose rows are the coefficients of $\boldsymbol{u'}$ and $\boldsymbol{v'}$
3. For all $1 < i < t$
   (a) Compute over the rational number $(x_i, y_i) = (R_i, 0)G^{-1}$.
   (b) Using the element $\lfloor x_i \rceil \boldsymbol{u'} + \lfloor y_i \rceil \boldsymbol{v'}$, build the triangle around $(R_i, 0)$ which the edges are formed by $\boldsymbol{u'}$, $\boldsymbol{v'}$ and $\boldsymbol{u'} \pm \boldsymbol{v'}$.
   (c) Returns the three differences between $(R_i, 0)$ and a vertex of the triangle.

The following procedure is summarized in Figure 7.2. Item 3a does not need a full inversion of G, which is a 2×2 matrix: indeed, the computation of x_i is equal to $R_i \boldsymbol{v}[1]/(\boldsymbol{u}[0]\boldsymbol{v}[1] - \boldsymbol{u}[1]\boldsymbol{v}[0])$ and y_i is equal to $-R_i \boldsymbol{u}[1]/(\boldsymbol{u}[0]\boldsymbol{v}[1] - \boldsymbol{u}[1]\boldsymbol{v}[0])$.

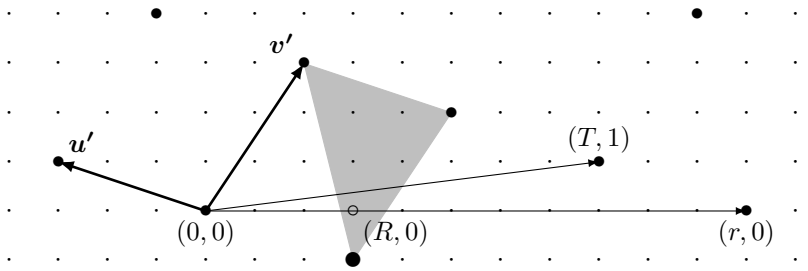


Figure 7.2 – Triangle around the target $(R, 0)$ composed by close vectors.

By extending all the vectors to the dimension t in the previous procedure, that is replacing $(R_i, 0)$ by $\boldsymbol{w_i}$, $(r, 0)$ and $(T, 1)$ by the two first vectors of $M_{\mathfrak{Q},\mathfrak{R}}$, we get three closest vectors, stored in a set W_i, of $\boldsymbol{w_i}$.

Algorithm. In our arsenal, we therefore have at this point:

- the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ that reach the conditions of Proposition 6.2;
- for $i > 1$, the set of three vectors W_i where the first two coordinates are relatively small and the coordinate i is equal to 1.

We now can describe the algorithm in three dimensions, and give a quick overview of the modification we need to apply for larger dimensions.

1. Set $\boldsymbol{c}$ to $(0, 0, 0)$, the starting point of our algorithm.
2. While $\boldsymbol{c}[2] < H_2^M$, do
   (a) Enumerate, with $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{u} + \boldsymbol{v}$, all the element in the intersection of the lattice and the sieving region of shape $(\cdot, \cdot, \boldsymbol{c}[2])$.
   (b) Add to $\boldsymbol{c}$ one of the vectors W_2, such that the new vector $\boldsymbol{c}$ is in the sieving region.
   (c) If it is not possible,
      i. Add to $\boldsymbol{c}$ one of the vectors W_2, such that the new vector $\boldsymbol{c}$ has its first coordinate coordinate in $[H_0^m, H_0^M[$ or the closest possible to an element of the sieving region if it is not possible.
      ii. If $\boldsymbol{c}$ is not in the sieving region, use a multiple of the vector $\boldsymbol{u}$ or $\boldsymbol{v}$ with the largest first coordinate to have the first coordinate of $\boldsymbol{c}$ in $[H_0^m, H_0^M[$.

In dimension t larger than 3, we do the following modification:

- The vector $\boldsymbol{c}$ is set to an element $\boldsymbol{C} = (\cdot, \cdot, H_2^m, H_3^m, \ldots, H_{t-1}^m)$ in the lattice but not necessarily in the sieving region at Item 1.
- The while loop at Item 2 is modified to a for loop for an element k in 0 to $(\prod_{i=2}^{t-1} H_i^M - H_i^m) - 1$, the control on the bounds of the vector $\boldsymbol{c}$ will be carry out by our last modification.

- To enumerate all the possible $(\prod_{i=2}^{t-1} H_i^M - H_i^m) - 1$ plane, we use a procedure, close to the one described for the line sieve, described below.

To enumerate all the possible planes, we store in a list S indexed from 0 to t, a current starting point at index i that have the same $(i+1)$th first coordinates as the one of the vector $\boldsymbol{c}$ used in our enumeration. Once we reach a new plane, we update S. At the beginning of the algorithm, the list S contains t times the vector $\boldsymbol{C}$ defined above, even if it is not necessary to define $S[0]$ and $S[1]$. The procedure is the following, given in input the list S, the $t-2$ lists W_i, the Franke–Kleinjung vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ and the t-sieving region $\mathcal{H}$:

1. Set k to 2 and $\boldsymbol{c}$ to an element of a new plane starting from $S[2]$, using the vectors of W_2 and eventually the Franke–Kleinjung vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ to have $\boldsymbol{c}[0]$ in $[H_0^m, H_0^M[$.
2. While $\boldsymbol{c}[k] = H_k^M$
   (a) Increment k
   (b) If $k < t$, set $\boldsymbol{c}$ to an element of a new plane starting from $S[k]$, using the vectors of W_k and eventually the Franke–Kleinjung vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ to have $\boldsymbol{c}[0]$ in $[H_0^m, H_0^M[$.
   (c) Othewise, break the loop.
3. For i in $[2, k]$, set $S[k]$ to $\boldsymbol{c}$.

The output of this algorithm gives a new starting point in $\mathcal{H}$ to enumerate a plane and the updated list S.

The plane sieve is the last of our enumeration algorithm that guarantees to discover all the elements in the intersection of the lattice and the sieving region. But, the plane sieve becomes inefficient when the volume of the lattice is larger than $I_0 I_1$, this is why we use the space sieve.

The space sieve

The space sieve is the last sieve algorithm we use in the three-dimensional case. As this algorithm has some heuristic parts, we will describe them and explain the choices behind.

Remark 7.2. Unlike the line and plane sieve, we do not have yet implemented a version of the space sieve in dimension higher than 3 fully functional, we then stick to the three-dimensional case.

A very simplified sketch of the plane sieve is given in the following:

Initialization. Look for the 1–nearly-transition-vectors and the 2–nearly-transition-vectors. Set $\boldsymbol{c}$ to (0, 0, 0).

Enumeration. While $\boldsymbol{c}[2] < H_2^M$

1. Enumerate the plane $(\cdot, \cdot, \boldsymbol{c}[2])$.
2. Add to $\boldsymbol{c}$ one of the 2–nearly-transition-vectors with the smallest third coordinate to have a new element in the sieving region.
3. If it is not possible

(a) Use the plane sieve to enumerate all the plane $(\cdot, \cdot, d)$, where $H_2^M > d > \boldsymbol{c}[2]$ and find a new element $\boldsymbol{c_n}$ in the sieving region.
(b) Add $\boldsymbol{c_n} - \boldsymbol{c}$ to the 2–nearly-transition-vectors and set $\boldsymbol{c}$ to $\boldsymbol{c_n}$.

Enumeration. The enumeration part is not really more complicated that it seems. In Item 2, we try all the 2–nearly-transition-vectors we have precomputed, sorted by increasing coordinate 2. We now describe the initialization step, that is how we find these 2–nearly-transition-vectors.

Initialization. A way to compute the sets of 1–nearly-transition-vectors and 2–nearly-transition-vectors is to perform small linear combination of an adapted basis. To find such a basis, we can use a skew basis reduction, described in Section A.1 to have three basis vectors $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$ such that:

- the coordinate $\boldsymbol{v_i}[0]$ must be less or not too large compare to I_0.
- the coordinate $\boldsymbol{v_i}[1]$ must be less or not too large compare to I_1.
- the coordinate $\boldsymbol{v_i}[2]$ is small.

To obtain such a basis, we perform a skew lattice reduction with the weights $w = \lambda(1/I_0, 1/I_1, I_0 I_1/r)$, where λ is a real number. To perform the LLL algorithm, we have an implementation of LLL over the `mpz_t` and over the `int64_t`: the implementation uses temporarily the type `__int128_t` and have a fallback to the implementation on the `mpz_t`.

Choice of λ. The parameter λ can be set to rI_0I_1: this implies that w is equal to $(rI_1, rI_0, I_0^2 I_1^2)$. The weight can therefore be encoded on `int64_t`. It seems reasonable because r is often less than 2^{32} (much less in our computations), and I_0 and I_1 are less than 2^{15} (less than 2^{11} in our computations). However, if we use this definition of the weight in our implementation, the weighted scalar product will probably produce temporary integers larger than 2^{127}, and then, the LLL computation of `int64_t` will probably backtrack almost every time to the LLL on `mpz_t`. A quite natural workaround is to consider that $I_0 = I_1$ (it is always the case in all of our computation): the weight w can now be written as (r, r, I_0^3), but the weighted scalar product can produce integer of size larger than 64-bit, which are not allowed in our implementation. We need to stick to the weight $w = (1, 1, I_0^3/r)$, which is a little bit less precise than what we can expect, due to the storage of I_0^3/r on `int64_t`.

Small linear combinations. Once the skew basis $\mathcal{B}$ composed by $\boldsymbol{v_0}$, $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ is computed, we compute 27 linear combinations $\lambda_i \boldsymbol{v_i}$, where the λ_i are in $[-1, 2[$. As a k–nearly-transition-vector has its coordinate k positive and $\mathbf{0}$ is not a nearly-transition-vector, we can restrict our computation to 13 linear combinations. To be useful, a large proportion of these linear combinations must give 2–nearly-transition-vectors or 1–nearly-transition-vectors.

Modification of the weights. During the small linear combinations, we want to produce the largest number of nearly-transition-vectors. If the skew basis given as input of the small linear combinations step is too stretch on the two

first coordinates, it will be maybe too difficult to find nearly-transition-vectors with the strategy we proposed. It is why we modify the theoretical weight w to have a bit smaller coordinates than expected on the two first coordinates and a bit larger last coordinate. It allows us to have better chance that a skew-small-vector produced by the small linear combinations will be a nearly-transition-vector. Experimentally, we choose to have the weight $(1, 1, 2I_0^3/r)$ to take into account this modification, that gives in our experiment, the best number of nearly-transition-vectors with a good enumeration of all the elements in the intersection of the lattice and the sieving region.

The space sieve in practice. In this last paragraph, we show the efficiency of the space sieve in term of running time and accuracy. We also provide some data about the number of nearly-transition-vectors (generated initially, used, computed on the fly).

Timings. We first tried to use an enumeration algorithm following [92, Algorithm 10]; the space sieve turned out to be about 120 times faster than our implementation of this enumeration algorithm.

We also compared the efficiency of the space sieve and the plane sieve. We sampled 1600 special-$\mathfrak{Q}$s among those that we used during the computation of a 300-bit $\mathbb{F}_{p^6}$ and sieved about $2^{16.7}$ ideals of norm larger than $I_0 I_1$ on each side. On average for each side, the plane sieve takes 5.20s in a single core `i5-4570 CPU @ 3.20GHz`, against 1.38s for the space sieve.

Accuracy. We know that the line and plane sieves report all the elements contained in the intersection of the sieving region and the lattice: it gives us a reference to compare the number of elements enumerated using the space sieve, because these elements must also be found using the line and plane sieves. With the same 1600 special-$\mathfrak{Q}$s used previously, we miss on average 5.7% of the elements in the intersections per special-$\mathfrak{Q}$s by using the space sieve: compared to the acceleration factor due to the space sieve, it seems reasonable to use the space sieve.

Number of nearly-transition-vectors. The space sieve algorithm relies heavily on the notion of nearly-transition-vectors, but the number of them is not easily controlled and we hope to find transition-vectors. For a given lattice which have no 1–transition-vector, we can define the 2–transition-vector associated to a point in the plane $[H_0^m, H_0^M[\times[H_1^m, H_1^M[$. In Figure 7.3, pictures are shown where a different color is associated to each transition vector, as we do in Figure 6.2 for the Franke–Kleinjung vectors. The example on the right is highly degenerate, in the sense that there are many different transition vectors; however, in this example, most of the area is covered by the vector with the smallest coordinate 2. We also expect that, if we need all the 2–transition-vectors to enumerate all the elements in the intersection of a lattice and the cylinder of square basis $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\mathbb{Z}$, it is not necessary to have found all the 2–transition-vectors to enumerate the elements in the intersection of a lattice and a sieving region.

On average for 2^{24} different lattices coming from our 300-bit $\mathbb{F}_{p^6}$ example for which we can use the space sieve, our initialization to find 2–nearly-transition-

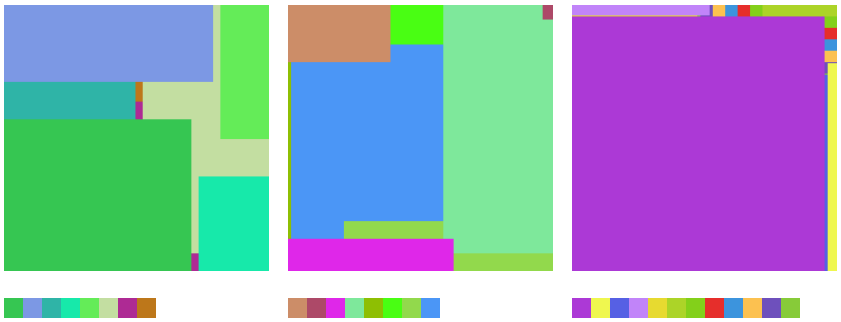


Figure 7.3 – Three examples of how transition vectors cover the plane $[H_0^m, H_0^M[\times[H_1^m, H_1^M[$. Each color corresponds to a transition vector; below each picture, the colors are listed in the increasing order of the coordinate 2 of the corresponding vector.

vectors procedure generates 10 vectors, at least 1 and at most 13. On average, only 4 are necessary to perform the enumeration, at least 1 and at most 13. During this enumeration, the plane sieve is called as a fall-back strategy on average 0.08 times, and at most 7 times. But, in more than 40% of the case, this fall-back strategy was just called to reach the bound of the coordinate 2, and then, do not enumerate any new element.

7.3 Post-processing variants

7.3.1 The multiple number field sieve variants

As described in Section 4.5.2, there exist two different versions of the multiple number field sieve algorithm, depending on the polynomial selection we use. These two versions implies a different implementation of the cofactorization step.

Asymmetric version

The asymmetric version is the simplest to implement. Recall that, at the beginning of Item 3 of the description of the relation collection at the beginning of this chapter, we have V arrays A_0 to A_{V-1} that contain, in a certain order (but the same for each A_i), the polynomials a that give a relation. In the asymmetric version, we have a main side, say the side 0: we enumerate the polynomials a in A_0, look for each polynomial a if we can find it in A_i, where i is in $[1, V[$, and for each side i, do the cofactorization in each sides and if the norms are smooth in at least two sides, keep the relation and give the factorization for each smooth norm.

Symmetric version

The symmetric version is more complicated. Indeed, there does not exist a main side, then we need to look, at each time, if a polynomial in an array is in at least one other array. To do that efficiently, we recall that the arrays A_i contain

the indices of the vector $\boldsymbol{c}$ in the one-dimensional array A and the arrays A_i are constructed to store increasing values. We use a similar algorithm to the one described below to find all the polynomials a that give a relation

1. Set k to the minimal value of all the minimal values of the arrays.
2. Set K to the maximal value of all the maximal values of the arrays.
3. While $k < K$
   (a) If k occurs at least in two arrays
      i. Find the corresponding vector $\boldsymbol{c}$ of the index k.
      ii. Compute the polynomial $a = \boldsymbol{c}M_{\mathfrak{Q}}$.
      iii. Perform the cofactorization in each needed side. If there is at least two smooth norms, report a and the smooth factorizations of the corresponding sides.
   (b) Set k to the next minimal values of the arrays.

7.3.2 Using Galois automorphism

We will describe in this section how we implement the Galois action, and specifically the Galois of order 6 we use in our computation, that is given by $\sigma : x \mapsto -(2x+1)/(x-1)$. We run the relation collection with polynomials a of degree 2, therefore let a be equal to $a_0+a_1x+a_2x^2$ a polynomial that gives a valid relation. Let y be equal to $-(2x+1)/(x-1)$, therefore, because $y \neq -2$, $x = (y-1)/(y+2)$. By replacing x by $(y-1)/(y+2)$ and by multiplying by $(y+2)^2$, we finfindt $a_0 + a_1x + a_2x^2 = (-4a_0 - 2a_1 + a_2) + (4a_0 + a_1 - 2a_2)y + (a_0 + a_1 + a_2)y^2$. Therefore, $\sigma(a) = (-4a_0 - 2a_1 + a_2) + (4a_0 + a_1 - 2a_2)x + (a_0 + a_1 + a_2)x^2$. In other words, the Galois action σ acting on a gives the polynomial a' whose coefficients are given by the matrix-vector product

$$(a'_0, a'_1, a'_2) = (a_0, a_1, a_2) \begin{pmatrix} 4 & 4 & 1 \\ -2 & 1 & 1 \\ 1 & -2 & 1 \end{pmatrix}. \tag{7.2}$$

Let f_0 and f_1 be two polynomials as defined in Chapter 4 that share the same Galois action σ. Let a be a polynomial that gives a relation, that is $\mathrm{Res}(f_0, a) = N_0$ and $\mathrm{Res}(f_1, a) = N_1$ are smooth. Let σ be acting on a to produce a': following the description on how to deal with the Galois action in Section 4.1.1, we know that $\mathrm{Res}(f_0, a') = N_0 \times \mathrm{Res}(f_0, x-1)/\operatorname{lc} f_0$ and $\mathrm{Res}(f_1, a') = N_1 \times \mathrm{Res}(f_1, x-1)/\operatorname{lc} f_1$, where $\mathrm{Res}(f_i, x-1)/\operatorname{lc} f_i$ is the norm in the appropriate number field of the denominator of the rational function that defines σ. If the polynomial a' is not content free, we must divided it by this content, that implies a division of the resultants by the content raises to the degree of the corresponding polynomial.

7.4 Integration into CADO-NFS

In this section, we will describe how the implementation is integrated into CADO-NFS, and what we would need to have an automatic tool like the script

`cado-nfs.py` to perform the complete computation of a discrete logarithm, given only the field $\mathbb{F}_{p^n}$ and the element from which we want to compute the discrete logarithm.

7.4.1 Using parts of CADO-NFS and NTL

Most of the code of CADO-NFS is written in C and our implementation of the relation collection in higher dimension is also written in C. There exists an implementation of polynomials over `mpz_t`, called `mpz_poly`, in which we mainly had to add the support of the computation of the resultant between two univariate polynomials, following Algorithm F.1. There exists also an implementation of polynomials over `double`, called `double_poly`, in which we also added an implementation of the resultant: this implementation is not robust against loss of precision, but has a fall-back to the computation of the resultant using `mpz_poly` in some cases.

We also use the implementation of the LLL algorithm over the `mpz_t`, and based our implementation on the LLL over `int64_t` on this code, itself based on the code provided in the NTL library [168].

For license compatibility, NTL (under GPL) could not be included in CADO-NFS (under LGPL), requiring to implement some needed algorithms. But, with the change in the license used by NTL, now under (L)GPL, it will be possible to use it: an interesting algorithm for the relation collection in higher dimension will be the irreducibility test over $\mathbb{Z}$ of a polynomial. Verifying if a degree 2 polynomial is irreducible over $\mathbb{Z}$ is not a difficult task, but for higher degree, it is more complicated. Then, in our implementation of the relation collection, if a polynomial of degree higher than 2 gives a relation, we do not check if this polynomial is irreducible or not, but we provide a tool written in C++ using NTL that takes as input a file containing the output relations and verify if the polynomial that gives the relation is irreducible.

ECM chain

In addition to the use of some libraries available in CADO-NFS, we use the whole mechanism of the ECM chain to perform the cofactorization step of the relation collection. We describe here a simplified version: the input of this description is one norm N_0, without the small factors, coming from the relation collection and the corresponding smoothness bounds B_0, the output indicates if the norm is B_0-smooth, or if it is with high probability not B_0-smooth.

1. While the loop is not broken
   (a) Set B_1, the usual parameters of the ECM algorithm, to a small value and A, the possible factor found by ECM, to 0.
   (b) For i in $[0, k[$
      i. randomly choose one curve and perform ECM on it with B_1
      ii. if a factor is found, set A to this factor and break the for loop.
      iii. increase B_1.
   (c) If $A = 0$ or $A > B_0$, the input integer N_0 is not B_0-smooth and break the loop.

(d) Else,

i. divide N_0 by A,

ii. if $N_0 < B_0$, the input integer N_0 is B_0-smooth and break the loop.

The complete implementation of this simple description can be found in `sieve/ecm/facul.cpp`. The integer k of Item 1b, namely the number of curves we use, is chosen according to the target probability (90%, 95%, 99%) of finding a factor of size $\log B_0$. The increasing of B_1, from the value 105 set in Item 1a in CADO-NFS, in Item 1(b)iii is to add $\sqrt{B_1}$ to B_1. Continuing or not the cofactorization procedure in Item 1(d)ii is a more complicated test than the one written here.

The simple described strategy is not optimal for the two-dimensional relation collection: the first calls to ECM are probably useless, because the factors up to the sieving bound are removed from the norm. But, in our implementation of the relation collection, we do not remove from the norms all the primes up to the sieving bound, and we only perform a trial division with a smaller bound than the sieving bound: the first calls to ECM in our context is therefore useful. In CADO-NFS, ECM uses two types of curves: the Brent–Suyama one and the Montgomery one. Using Edward curves [31, 16] will probably give better result and could also be used in CADO-NFS.

Let us consider that all the primes, and their powers, up to the sieve bound are removed from the norms we test for smoothness. Let us also consider that we want to know if the two norms (N_0, N_1) are doubly smooth. Then, if the algorithm we described before can be applied, we can use clever strategies, as proposed by Kleinjung [62]. Some of them are implemented and available on demand in CADO-NFS.

The cofactorization on one side can be also done by using a factorization tree, as proposed in [30] and used practically for the computation in a 768-bit prime field [122]. This strategy still applies in our context, even if we remove only a few small primes.

Finally, Miele, Bos, Kleinjung and Lenstra evaluate in [137] the feasibility of the cofactorization on GPU, instead of the classical one on CPU.

7.4.2 Road map to an automatic tool

In this last section, we will describe some theoretical and practical challenges to solve in order to have an automatic tool to compute discrete logarithm using our implementation of the relation collection in three dimension and higher, as the `cado-nfs.py` script does for both discrete logarithm in prime fields and factorization. We do not discuss on how to make the input and output of the different steps understandable by the top-level script, but which programs we need at all the steps of the computation.

Polynomial selection

In Chapter 4, we have shown that there exist 6 different polynomial selections: even if two of them are now surpassed and the polynomial selection $\mathcal{A}$ generalizes two others, it remains two different polynomial selection, the JLSV_1 and the $\mathcal{A}$

one, the last one parametrized by two parameters, that gives 4 different types of polynomial pairs for a $\mathbb{F}_{p^6}$.

It is possible, as shown in the last different practical computations in $\mathbb{F}_{p^n}$ [72, 91, 85], to distinguish by estimating the size of norms, and therefore the Murphy E value, for a typical but not optimized polynomial pair, which polynomial selections are better than others, and then consider only two or three types of polynomial pairs. We must also take into account the Galois action and the ability to have a large negative α value.

Once we have found the probably best polynomial selection, we need to perform the polynomial selection. In dimension 3, we have implemented a way to compute the α and Murphy-E values, according to the description of Section 4.1. If all the needed functions for the JLSV_1 polynomial selection are available in the `mpz_poly`, it is not the case to perform the $\mathcal{A}$ polynomial selection. When this polynomial selection drops to the conjugation polynomial selection, we need to compute a bivariate resultant, that is why we implemented one in the new type `mpz_poly_bivariate`: the algorithm we use is described in Algorithm F.2. The other needed function are already implemented in `mpz_poly`. But, when the $\mathcal{A}$ polynomial selection does not correspond to the conjugation one, we need to perform at least a test of irreducibility over $\mathbb{Z}[x]$, which is not implemented in CADO-NFS. Because this test is implemented in NTL and its license is now compatible with the one of CADO-NFS, it may be possible to implement the $\mathcal{A}$ polynomial selection in all the cases.

However, this task is not easy, and needs to be carefully implemented to consider the largest number of polynomial pairs in the shortest time. A brief overview of how to perform such a task will be presented in Section 8.1.1.

Parameters for the relation collection

As the complete computations of a 240, 300, 389 and 422-bit $\mathbb{F}_{p^6}$ [85] and a 324-bit $\mathbb{F}_{p^5}$ [84] were performed using a three-dimensional relation collection, we can provide parameters that have a good chance to work for fields of close size with the same extension. However, for intermediate or larger sizes, or for different extension, it is difficult to predict which parameters can be useful, see Section 8.1.2 for details. This is the choice of the discrete logarithm implementation in CADO-NFS: parameter files are available to compute discrete logarithm in prime fields of size 100, 200, and 512-bit.

Linear algebra

Concerning the linear algebra, the main modifications concern the conditioning of the matrix, since the Wiedemann algorithm deal with matrices, and is agnostic of the way the matrix was produced.

In CADO-NFS, all the processing to build the matrix highly relies on the fact that the relation collection is done with polynomial of degree one, and then involves ideals of degree one. Before the filtering, the relations and the ideals are labeled in a unique way, taking advantage of the degree one of both ideals and relations. To reuse this system without entirely modifying all the mechanism, we encode a relation given by a triple (a_0, a_1, a_2) in a unique way that makes believe to CADO-NFS that the relation comes from a (A_0, A_1) pair. If these

changes work for practical computations, these features are not robust and not ready to be pushed into production.

A nice feature when a Galois action of order k is shared by the polynomials that define the number fields is the ability to reduce the number of column by a factor around k. This feature, available for a computation where the relation collection is performed with polynomial of degree 1 in $\mathbb{F}_{p^2}$, is currently at work.

Descent

This is probably the most difficult step to automate at this point, because we have only few experience about the initialization step with Guillevic's work [86, 88] and with the descent using a three-dimensional relation collection. For now, this task still requires human effort to select which parameters are the best, given a size of special-$\mathfrak{Q}$ to be descended and a targeted smoothness bound. The initialization step is coded in Magma and the descent step relies on our implementation of the relation collection. Once the choice of the parameters is stable, we can hope to implement an automatic tool.

The initialization procedure expresses the target as a product of ideals in one number field (say the number field 0). All the ideals $\mathfrak{Q} = (q, g)$ of norm larger than the corresponding smoothness bound are inserted in a stack S as $(0, q, g)$. This stack is the input of this simplified descent algorithm (in particular, we do not take care of the building of the tree):

1. While S is not empty:
   (a) pop out the first element of the stack and store it as (s, q, g).
   (b) perform the relation collection by setting the ideal (q, g) on side s
   (c) select the best relation and add to S, for each side i, the element (i, q, g), where (q, g) is an ideal of norm larger than the smoothness bound on side i.
   (d) if there is no relation, use a fall-back strategy.

In Item 1c, we call *best relation* the relation that will require less effort to descend the ideals of this relation whose norms are larger than the smoothness bound of the corresponding number field. This effort depends on the number of ideals that is needed to descend and the size of these ideals. The effort to descend one ideal of a given size is known, thanks to the parameters we have previously identified: indeed, for a given size of a special-$\mathfrak{Q}$, the time to descend a specific special-$\mathfrak{Q}$ is known.

The fall-back strategy in Item 1d is used when no relation was found. In this case, the first things to do is to modify the parameters: increasing the smoothness bounds, the searching space, the thresholds, ... If this still gives nothing, an extreme choice is to discard completely the relation that have given the special-$\mathfrak{Q}$ which is impossible to descend.

Chapter 8

Experimental results

To validate the practical impact of the improvement and new algorithms proposed theoretically, we try to run computations in a finite field of the largest characteristic we can reach, given an extension degree. It also helps cryptographers to (re)evaluate the security of a cryptosystem based on a certain type of fields.

Before running a computation or a part of the computation, especially the relation collection, we need to find *good* parameters. By good, we mean that the computation can be performed using a reasonable amount of resources in a not so large running time. As we have shown in Chapter 3, Chapter 4 and Chapter 5, NFS is composed of multiple algorithms, each of them having different parameters. We will focus on the relation collection, describing how we can evaluate if parameters have a chance to give a complete set of relations and how we perform the computation on a cluster. We will also summarize the results of the computations we did.

8.1 Looking for good parameters for the relation collection

We recall first that the input of the relation collection is the following:

- the polynomials that define the number fields;
- the sieving intervals that define the sieving region;
- the sieving bounds;
- the thresholds;
- the smoothness bounds;
- the range of special-$\mathfrak{Q}$s;
- the side of the special-$\mathfrak{Q}$s.

Remark 8.1. There exist also other parameters, which have less impact on the computation, for instance the number of curves used to perform the cofactorization step using ECM chain (see Section 7.4.1), if we do not use the default parameters.

In all these parameters, we can distinguish the one that can be theoretically defined to be the best, and the others that are more linked to the implementation. We will justify in the appropriate section this assertion, but we can from now give a scheme of the procedure to find good parameters:

1. select the polynomials;
2. select the smoothness bound, the original search space and the side of the special-$\mathfrak{Q}$s;
3. select the sieving region per special-$\mathfrak{Q}$, the sieving bounds, the thresholds and the range of special-$\mathfrak{Q}$s.

We will describe briefly how the polynomial selection can be performed for NFS-HD and how we can select good parameters for the relation collection step.

8.1.1 Polynomial selection

We begin first by the selection of the polynomials because we have the Murphy functions in our toolbag to distinguish some of the best polynomial pairs we can produce, using one of the four polynomial selections described in Section 4.1.

The first step is to distinguish which are the best polynomial pairs using the Murphy-E function, given a searching space and smoothness bounds, as in [91, Figure 1], in [84, Figure 1] or in [85, Figure 2]. We now know on which polynomial selection we can concentrate our effort. We recall that the JLSV_0 and JLSV_2 are not taken into account as shown in Section 4.1.3

Let us consider that we focus on one polynomial selection. Concerning the α function, the main part considers the number of roots modulo small primes, therefore, having a large negative value does not depend on the size of the coefficients of the polynomials. However, the Murphy-E value is highly dependent on the size of the coefficient of the polynomials. Then, we must control the size of the coefficients, allowing us to reduce the set of possible good polynomials. By looking at Table 4.1, all the polynomial selections, except the JLSV_1 one, involve a polynomial with small coefficients. Polynomials of tiny coefficients are not numerous, we have therefore a small room to find a polynomial with a good α value. The second polynomial is, except for the conjugation polynomial selection, depends on the shortest vector of a lattice, and the second shortest vector of this lattice has often larger coefficients. We have therefore not a large choice to define the polynomials.

With the JLSV_1 polynomial selection, even if the polynomial g_0 and g_1 of the description in Section 4.1.2 have small coefficients, we have more freedom by choosing the parameters c_0, c_1, c_2 of size about $\log \sqrt{p}$. If we use the asymmetric variant of the JLSV_1 polynomial selection, we can subtract a small multiple of the polynomial with the smallest coefficients to the polynomial of the largest coefficients, to obtain a better α value without increasing a lot the Murphy-E value.

About the conjugation polynomial selection, we have only a few choices to define the polynomial of degree $2n$, because it depends on the roots modulo p of the quadratic polynomial μ. But, there is a bit more room to define the polynomial of degree n, thanks to the rational reconstruction: it is possible to perform very small linear combinations between two polynomials obtained by two different rational reconstructions to increase the Murphy E value.

8.1.2 Parameters for the relation collection

To select the best parameters in term of running time and number of relations found, we first recall some obvious bounds on the parameters, then give first approaches to find these parameters and finally how we estimate the cost of the computation and the number of found relations.

Bounds on the parameters

The first obvious bound is the sieving intervals. Indeed, the sieving region defined by the sieving intervals must fit in memory if we use the standard description of the sieving algorithm (it exists the possibility to divide a search space that does not fit in memory in subregions that fit in memory, as in [122] where the original region per special-$\mathfrak{Q}$ contains 2^{40} elements: this works is under development in the implementation of the two-dimensional sieving in CADO-NFS, and this is not clear how to do such a task in an higher-dimensional sieving). Practically, the sieving region does not exceed 2^{31} elements for the practical computation we can achieve, according to the majority of the parameter files available in CADO-NFS.

The range of special-$\mathfrak{Q}$s has its largest value less than the smoothness bound of the corresponding side. For the lowest value, there is no theoretical reason to fix it above the sieving bound: indeed, if the special-$\mathfrak{Q}$ is an ideal we sieve, it suffices to not consider this ideal during the sieving step, to avoid to remove two times the contribution of the same ideal. But, the smallest the special-$\mathfrak{Q}$s are, the largest the number of duplicates is. The range of special-$\mathfrak{Q}$s begins therefore just above the sieving bound, and must finish before the smoothness bound.

The largest ideal we sieve is often an ideal that has a chance to have at least one element in the intersection of the lattice of this ideal in the $\mathfrak{Q}$-lattice and the sieving region. If the norm of the largest ideal we sieve is r, and the length of the intervals that define the t-sieving region are I_i, where i is in $[0, t[$, the norm of the ideal r is less than $I_0 I_1 \cdots I_{t-1}$.

About the smoothness bounds, an obvious bound is imposed by the linear algebra step. In our computations, we consider that the large prime bounds cannot exceed 2^{29}, but the implementation available in CADO-NFS can deal with larger smoothness bounds. This is dependent on the number of relations we can obtain to help the filtering step to reduce the size of the matrix.

Finding a first set of parameters

We assume from now on that we have selected one polynomial pair and want to find the other parameters.

Trade-offs. A first, non-surprising, remark, is that, if we increase the parameters, we increase the number of relations we find. But this remark must be nuanced.

Let us consider for example that we increase by one bit the length of all the intervals that define the sieving region. It is probable that we increase the number of relations. But, by always increasing the length of the sieving intervals, we reach values where the increasings do not allow us to discover new relations, or very few. Concerning the running time, it obviously increases when a parameter increases. We then can have a trade-off between the number of found relations and the running time.

This consideration is about the same for all the parameters. It exists therefore one or several sets of parameters that allow to have the best trade-off between the running time and the number of relation. It seems however that there exist parameters that can be chosen before the others.

Using Murphy-E. We have claimed earlier that the polynomial selection can be done without considering the other parameters. Once we have the polynomials that define the number fields, the smoothness bounds and the original search space can be approximated by considering the Murphy-E value, defined in Chapter 3 and Chapter 4.

Since only the smoothness bounds and the original search space are needed to compute the Murphy E value, we can approximate them. The rationale is to evaluate the number of relations we can obtain by increasing or decreasing the smoothness bounds to have a complete set of relation. For the smoothness bounds we get, we modify the search space to be consistent with the estimated cost of the linear algebra. In the same time, we can chose the side of the special-$\mathfrak{Q}$s: it is often the side where the norms are the largest.

From the original search space, we can almost define the sieving region per special-$\mathfrak{Q}$ and the range of special-$\mathfrak{Q}$s by considering that the number of elements in the original search space must be equal to the number of elements considered by the largest special-$\mathfrak{Q}$ multiplied by the volume of this special-$\mathfrak{Q}$. If all these parameters are fixed, we have three free parameters on the seven we have listed. It is more difficult to fix these parameters: we therefore use the algorithm to estimate the number of found relations and the timings described in the following section, to fix it after some experiments using different possible parameters.

Using previous computations. An alternative or a complementary approach to the one we have described previously is to use the parameters of previous computations. It is obviously easier to use and infer if the computations we use are performed in the same type of field and with a similar approach (with or without special-$\mathfrak{Q}$-method, with three-dimensional sieving algorithms, ...). Before our computations, there existed two reports of computations in $\mathbb{F}_{p^6}$, the one of Zajac [185] and the one of Hayasaka, Aoki, Kobayashi and Takagi [94]. These give to us first references we can try to improve.

But, if we perform from scratch a computation, or perform a significant record in a given field, a solution is to use parameters of computations done in different contexts to approximate the good parameters for our computations. The informations and parameters we have before trying to run a relation collection are about the size of the field and the polynomials that define the number

fields, and maybe the smoothness bounds and the searching space. We look for the parameters of a previous computation that target the same size, concerning the field or the length of a large integer to be factored, as we can found in CADO-NFS for example: this computation is called our *reference*.

Using these parameters, we can obtain some essential informations, a crucial one is the size of the norms in both sides. If these sizes are close to the one we can obtain with the reference computation, the parameters can be doubtless used as a first approximation, and refined locally. But if the sizes of the norms are different, smaller or larger, we need to change our reference: now, we look for parameters that allow to reach the same size of norms: this computation will be our new reference.

With this strategy, we quickly reach a reasonable first set of parameters.

Estimations

Let us consider that we have a first set of parameters and want to estimate if the number of relations allows us to have a complete set of relations and if the running time is achievable. The estimation we propose is relatively simple, but quite powerful. The rationale is to sample N special-$\mathfrak{Q}$s almost equally distributed along the range of special-$\mathfrak{Q}$s to have an idea of the distribution of the number of relations we hope to obtain given a size of a special-$\mathfrak{Q}$. We describe in the following the different steps of the estimation, in a simplified way but close enough to reality: we do not take care in particular of the non-equidistribution of the special-$\mathfrak{Q}$s. As input of the estimations, we have the parameters we want to estimate, especially the range of special-$\mathfrak{Q}$s $[q_m, q_M]$. The function π is the prime counting function.

1. Set s to $(q_M - q_m)/N$, and L to an empty list.
2. For i from q_m to q_M with step s
   (a) Set q to the next or previous prime to i that allows to build a special-$\mathfrak{Q}$ of inertia degree 1.
   (b) Perform the special-$\mathfrak{Q}$-method with the special-$\mathfrak{Q}$ and the other parameters, extract the number of found relations r and the timing t of this computation.
   (c) If this computation will be done with a Galois action of order k and the output relation are the conjugated relation, divide by k the timing and the number of relations.
   (d) Add to the list L the triple (q, t, r).
3. Set the estimated number of relations r and timing t to 0.
4. For all the two consecutive elements (q_0, t_0, r_0) and (q_1, t_1, r_1) in L
   (a) Add to r the quantity $(\pi(q_1) - \pi(q_0))(r_0 + r_1)/2$.
   (b) Add to t the quantity $(\pi(q_1) - \pi(q_0))(t_0 + t_1)/2$.
5. Output the estimated number of relations r and timing t.

In Item 4a, we compute the average number of relations we hope to find in the interval $[q_0, q_1]$ by considering that the average number of relations between q_0 and q_1 is almost the number of relations per special-$\mathfrak{Q}$ we have. An other approach is to compute the linear regression in $[q_0, q_1]$ of the number of relations, that is $(\pi(q_1)-\pi(q_0))(r_1+(r_0-r_1)/2)$ (we assume that $r_1 \geq r_0$). The same thing can be applied in Item 4b. The trend of our estimations is not modified, but there can be a little bit overvalued, on the contrary to the original description.

In this estimation, we however forget to take into account the number of duplicates, which can vary between 10% and 30% in our computations, and can exceed 50%, if we refer to the computation of the 1024-bit SNFS [65]. This must be taken into account before doing a computation.

Refinements

In the previous section, we have considered that the polynomial selection is fixed, in order to select the other parameters. But, if we consider the JLSV_1 asymmetric polynomial selection described in Section 4.1, this polynomial selection has a dependency in the size of the special-$\mathfrak{Q}$s, that is a parameter fixed at the end of the selection of the parameters. The strategy we propose and use in our computations is to start from a symmetric polynomial selection, find the parameters and then, perform an asymmetric polynomial selection, taking into account the size of the special-$\mathfrak{Q}$s. There is no reason that the older parameters are not suitable with this new polynomial selection.

There are two parameters that are almost only dependent on the implementation of the sieving algorithms: the sieving bounds and the thresholds. Let us consider that we have found parameters to have a complete set of relations. We can maybe improve the running time by modifying these two parameters: this is highly dependent of the implementations of the sieving algorithms and the ECM and the strategies to perform the cofactorization step. This local improvement must not modify drastically the number of found relations but can decrease the running time.

8.2 Computations

8.2.1 Using a cluster

Modern computations of discrete logarithms over finite field use a lot of core hours, see Table 1.1. Many steps of NFS are parallelizable (see [6, Figure 1]), especially the relation collection. In this phase, the special-$\mathfrak{Q}$-method allows us to run several instances for different special-$\mathfrak{Q}$ without the need of communication between instances. The relations given by each instance are aggregated and, when all the instances are finished, we only need to remove the duplicates (in our computations, a simple call to the shell command `sort -u` was enough).

The scheduler available on the clusters we used is OAR (`https://oar.imag.fr/`). Dealing with a cluster involves to deal with many other users and so, using exclusively all the machines of a cluster during more than a few hours is not allowed. A nice feature of the OAR scheduler is the ability to submit a job with the options `besteffort` and `idempotent`: the first one allows to submit a job which can be interrupted at any times by a higher priority job, and the second

one allows to reload automatically an interrupted job when the resources are anew available.

To maximize the chance to always have a job that runs on a node of a cluster, it is better to divide the tasks in the most atomic way as possible. In our computations, it means that the range of special-$\mathfrak{Q}$s per task should be relatively small (around a dozen). This has many advantages:

- it is easier to estimate the running time of a task,
- if a task is killed by someone, it is easier to identify it because its running time is much lower than expected, and rerun it (by hand or thanks to `idempotent`),
- if a problem occurs on a machine, we lose a few hours of computations, it is easier to identify the missing special-$\mathfrak{Q}$s and rerun the small range.

Indeed, for a range of special-$\mathfrak{Q}$s of norms close to each other, it is relatively easy to have a good idea of the running time of each special-$\mathfrak{Q}$, and then the whole running time of the small range. As the range is small, an interruption of the jobs is easily detectable and a workaround can be quickly performed. But, there are also some drawbacks:

- it can be difficult to manage all the tasks without the proper dedicated scripts and can become like a "baby-sitting" of jobs,
- the sieve basis is read by each thread at the start of a special-$\mathfrak{Q}$ range.

These drawbacks can fortunately be minimized. Reading the sieve basis is almost free (less than 2 seconds for the whole range of special-$\mathfrak{Q}$s), compared to the time to perform the relation collection (more than 200 seconds per special-$\mathfrak{Q}$ for the $\mathbb{F}_{p^6}$ of size 422 bits). About the "baby-sitting" of jobs, writing automatic scripts that send to a node a range of special-$\mathfrak{Q}$s not previously or currently used is not an easy task and we propose a quick overview of how this can be done.

First, we need to identify on the machines the largest number of thread we can use on a node to fit into memory. We need also to divide the targeted range of special-$\mathfrak{Q}$s in smaller ranges that contain an almost constant number of special-$\mathfrak{Q}$: this must be done by computing the number of primes in a range or by counting how many special-$\mathfrak{Q}$s there exist in the range. To launch a job, the process is essentially:

1. While there exists ranges of special-$\mathfrak{Q}$s that has not yet be sieved:
   (a) look for available node
   (b) for a given node
      i. select as many ranges of special-$\mathfrak{Q}$s as the node can accept
      ii. launch the relation collection on the different ranges of special-$\mathfrak{Q}$s

When we launch a range of special-$\mathfrak{Q}$s, we tag it to indicate that we process it. When the output is complete, the special-$\mathfrak{Q}$s range is tag to be finished, otherwise, if the output keeps unchanged for a lot of time, we reschedule it or wait if `idempotent` is available.

We now give some details about the computation we perform. The computation in extension degree 5 was performed with Guillevic and Morain [84]. The computations in extension degree 6 was split in two works: the first one concerning the polynomial selection and the relation collection in three finite fields of size 240, 300 and 389-bit with Gaudry and Videau, with the help from Guillevic for polynomial selection [72], and a second one completing these computations with the linear algebra and individual logarithm steps and computing a discrete logarithm in a 422-bit size field, performed with Guillevic, Morain and Thomé [85].

8.2.2 Extension of degree 5

We select the 65-bit prime $p = 31415926535897932429 = \lfloor 10^{19}\pi \rfloor + 45$, and consider the finite field $\mathbb{F}_{p^5}$ where $p^5 - 1 = (p-1) \cdot 11 \cdot 101 \cdot 191 \cdot 7363691 \cdot 33031656232204364259865845615041 \cdot \ell$ where ℓ is the 113-bit prime equal to 18872357657025660688767070155926911. Since the extension degree is prime, exTNFS [114] algorithm is restrained to its TNFS original form [22], with R a degree-5 number field above $\mathbb{Q}$ and sieving in dimension 10, or to NFS-HD, with $R = \mathbb{Q}$ (no tower), to compute discrete logarithms in the prime order subgroup of $\mathbb{F}_{p^5}^*$ of cardinality ℓ.

Our computations were done using `Xeon CPU E5520 @ 2.27GHz` cores.

Polynomial selection

The JLSV_1 and generalized Joux–Lercier (gJL) are expected to be the best polynomial selections, as shown in [84, Figure 1]. The best pair of polynomials (f_0, f_1) we get was found using the JLSV_1 method:

$$\begin{aligned} f_0 &= x^5 - 5x^4 - 5368736472x^3 + 10737472959x^2 - 5368736477x - 2, \\ f_1 &= 5851642500x^5 - 29258212500x^4 + 25042672429x^3 + 37689292642x^2 \\ &\quad - 4215540071x - 11703285000. \end{aligned}$$

We also provide two other polynomial pairs:

- one using the JLSV_0 polynomial selection

$$\begin{aligned} f_0 &= x^5 + 14x^4 - 7x^3 - 4x^2 - 4x + 15, \\ f_1 &= 2^8 f_0 + p \\ &= 256x^5 + 3584x^4 - 1792x^3 - 1024x^2 - 1024x + 31415926535897936269. \end{aligned}$$

- one using the gJL polynomial selection

$$\begin{aligned} f_0 &= 2x^6 + 3x^5 - x^4 + 2x^3 - 3x^2 - 2x - 3, \\ f_1 &= 4682288594364150x^5 + 1052001614041581 7x^4 - 17832477142237943x^3 \\ &\quad - 15171722661935206x^2 + 1592160578567340x + 1708993376270808. \end{aligned}$$

The α-values of these polynomial are reported in Table 8.1.

	$\alpha(f_0)$	$\alpha(f_1)$
JLSV_0	-2.0	-3.5
JLSV_1	-4.0	-8.3
gJL	-0.4	-4.5

Table 8.1 – α-values for the polynomial selection in $\mathbb{F}_{p^5}$.

Relation collection

Three-dimensional relation collection. The relation collection was performed using the special-$\mathfrak{Q}$ sieve [93] and the three-dimensional sieving algorithms described in Chapter 6. The smoothness bounds are set to 2^{25}, and the cofactorization is performed if on both sides, the remaining norms are smaller than 2^{80}, that is slightly more than three times the size of the large primes involved in the factorization of the norms. The special-$\mathfrak{Q}$s are set on side 1 and have norms in $]2^{21}, 2^{23.75}[$: inside a special-$\mathfrak{Q}$-lattice, we sieve on both sides the ideals of inertia degree 1 that have a norm bellow 2^{21}. There are 156,186 such ideals on side 0 and 155,192 on side 1. There are fewer ideals of inertia degree 2 of norm below than the smoothness bounds (759 on side 0 and 778 on side 1), and rare projective ideals (6 on side 1, which is coherent the factorization of the leading coefficient of f_1, that is $5851642500 = 2^2 \cdot 3^4 \cdot 5^4 \cdot 11 \cdot 37 \cdot 71$). We did not sieve these ideals.

In each special-$\mathfrak{Q}$-lattice, we consider a sieving region that contains 2^{25} elements $\boldsymbol{c}$ of the lattice, where the coordinates $(\boldsymbol{c}[0], \boldsymbol{c}[1], \boldsymbol{c}[2])$ are in the sieving region $[-2^8, 2^8[\times[-2^8, 2^8[\times[0, 2^7[$. The time per special-$\mathfrak{Q}$ during the computation was between 15.37 seconds and 93.87 seconds, and the largest number of relations per special-$\mathfrak{Q}$ is 34. The cost to find the 6,171,924 relations was about 359 CPU days.

Two-dimensional relation collection. For comparison, the relation collection was also performed with a two-dimensional sieving using the CADO-NFS implementation. For this computation, we use a polynomial pair coming from the JLSV_0 polynomial selection.

The special-$\mathfrak{Q}$s are set on side 0 and have norms in $]2^{24.25}, 2^{26}[$: inside a special-$\mathfrak{Q}$-lattice, we sieve on both sides the ideals of inertia degree 1 that have a norm bellow $2^{24.25}$. The smoothness bounds are set to 2^{26} on side 0 and 2^{27} on side 1, and the cofactorization is performed if on both sides, the remaining norms is less than 2^{52} on side 0 and 2^{54} on side 1, that is 2 large primes on both sides.

In each special-$\mathfrak{Q}$-lattice, we consider a sieving region that contains 2^{29} elements $\boldsymbol{c}$ of the lattice, where the coordinate $(\boldsymbol{c}[0], \boldsymbol{c}[1])$ are in $[-2^{14}, 2^{14}[\times[0, 2^{14}[$. The time per special-$\mathfrak{Q}$ during the computation was between 0.54 seconds and 18.14 seconds, and the largest number of relations special-$\mathfrak{Q}$ is 21. The cost to find the $10,458,616$ relations was about 375 CPU days. It is smaller than the $11,561,362 = \pi(2^{26}) + \pi(2^{27})$ that are almost needed, where π is the prime counting function.

We also provide an estimation using other parameters we expect to give a complete set of relations using a two-dimensional relation collection. Finally, the best running time for the relation collection and the smallest matrix are

reached by using the three-dimensional relation collection.

Summary of parameters for the relation collection. In Table 8.2, we summarize the parameters we use for the real computation and the results of some experiments with other parameters without performing the whole computation, but by inferring the results using a sample of special-$\mathfrak{Q}$s.

Dimension	2	2	2	3
Polynomials	JLSV_0	JLSV_0	JLSV_1	JLSV_1
Sieving bounds	$2^{24.25}$, $2^{24.25}$	$2^{24.25}$, $2^{24.25}$	$2^{24.25}$, $2^{24.25}$	2^{21}, 2^{21}
Smoothness bounds	2^{26}, 2^{27}	2^{26}, 2^{27}	2^{26}, 2^{26}	2^{25}, 2^{25}
Thresholds	2^{52}, 2^{54}	2^{52}, 2^{54}	2^{52}, 2^{52}	2^{80}, 2^{80}
Special-$\mathfrak{q}$ side	0	1	1	1
Special-$\mathfrak{q}$-range	$]2^{24.25}, 2^{26}[$	$]2^{24.25}, 2^{27}[$	$]2^{24.25}, 2^{26}[$	$]2^{21}, 2^{23.75}[$
Sieving region	2^{29}	2^{29}	2^{29}	2^{25}
Mean of norms	$2^{118}, 2^{203}$	—	—	$2^{144}, 2^{128}$
Raw relations	$10,458,616$	—	—	$6,171,924$
Unique relations	$8,256,215$	$\approx 16,000,000$	$\approx 9,900,000$	$4,999,773$
Needed relations	$\approx 11,561,362$	$\approx 11,561,362$	$\approx 7,915,618$	$\approx 4,127,378$
Set of relations	Incomplete	Complete (probably)	Complete (probably)	Complete
Time (CPU days)	375	≈ 900	≈ 400	359

Table 8.2 – Data for the relation collections in $\mathbb{F}_{p^5}$ of 324 bits.

Filtering

On the 6,171,924 relations produced with the relation collection, 4,999,773 were unique, and this led to a 1,489,631×1,489,625 matrix after singleton removal, reduced to a final 490,307×490,301 matrix after more intensive filtering.

Linear algebra

The linear algebra step is performed using the block-Wiedemann algorithm. The parameters used were $m = 12$ and $n = 6$. Then 6 parallel jobs were run, one for each of the 6 sequences. Each parallel job used a 2×2 node topology, each node having 8 cores.

The time to compute the Krylov subspaces was 237 hours, then 4 hours for the linear generator and 35 hours for the creation the solutions from the generator. 3,787,509 logs were reconstructed (out of at most 4,128,343 possible logs).

Individual logarithms

We finally ran the computation of an individual logarithm. First note that $h = X+1$ generates the whole multiplicative group $\mathbb{F}_{p^5}^*$ where $\mathbb{F}_{p^5}$ is represented using f_0 as defining polynomial. We find that h lifts to K_0 as $z + 1$ of norm $2^2 \cdot 3^2 \cdot 5^2 \cdot 23 \cdot 1037437$, corresponding to the factorization in O_0

$$(z+1) = \langle 3, x+4\rangle^2 \langle 2, x^3+x^2+x+3\rangle \langle 5, x+6\rangle^2 \langle 23, x+1\rangle \langle 1037437, x+1\rangle.$$

All logs were known from the first phase (including that of the degree-two ideal above 2), but that of norm 1037437 that we needed to descend. Finally,

$$\text{vlog}(h) = 69480237664316728325370489421116 17 \bmod \ell.$$

Now, consider the target made of the decimals of π

$$t = 3141592653589793238X^4 + 4626433832795028841X^3 + 9716939937510582097X^2 + 4944592307816406286X + 2089986280348253421.$$

After 20,000 seconds we find that the lift of $h^{9002259}\,t$ has a smooth norm and corresponding ideal factorization

$$\langle 2, x^3+2\rangle^2\langle 41, x+11\rangle\langle 43, x+21\rangle\langle 3471899, x+3245828\rangle$$
$$\langle 37276061, x+17122378\rangle\langle 3115088134901, x+1265257252254\rangle$$
$$\langle 366996697855783, x+268803256185002\rangle$$
$$\langle 377568478750783, x+9644708969240\rangle$$
$$\langle 4811620104558151, x+2380670555180752\rangle$$
$$\langle 120866356812660071, x+98064663938425303\rangle$$
$$\langle 4133950459282418267, x+1195413435698177697\rangle.$$

All ideals of norm > 37276061 had to be re-expressed in terms of prime ideals of smaller norm. Contrary to the relation collection step, we can re-express the elements by looking for a relation given by a degree 1 polynomial, and therefore use the program `las_descent` of the CADO-NFS package [176], which took 11,958 seconds, finally leading to

$$\mathrm{vlog}(t) = 284270745040684398905938148353673 8 \bmod \ell.$$

Note that we could use ideals of inertia degree larger than 1 whose logarithms would be known from the first step, though they rarely pop up at this stage, except for the smallest ones. Re-expressing these ideals would require to find a relation given by a polynomial of degree at most 2.

Summary of the computation

We summarize in Table 8.3 the running time of the four main steps of our computation using a three-dimensional relation collection.

Part	Time (CPU days)
Polynomial selection	15
Relation collection	359
Linear algebra	11.5
Individual logarithm	0.37
Total	386

Table 8.3 – Timing for each part of the computation of the discrete logarithm in $\mathbb{F}_{p^5}$ of 324 bits.

8.2.3 Extension of degree 6

The experiments in this section have been done with our C implementation of the three-dimensional sieving that we made available in the CADO-NFS official repository [176] (commit `089d552...`). In the file `README.nfs-hd`, instructions are given to reproduce all our experiments. The polynomial selection based on the criteria and constructions that we explained in Section 4.1 was carried out by Aurore Guillevic. All the running times are given after normalization for a single core at 2 GHz.

Computation with a 240-bit example of the literature

Polynomial selection and relation collection. Our first experiment follows the two previous computations made by Zajac [185] and Hayasaka et al. [94] for a 40-bit prime $p = 1081034284409$, yielding $\ell = 389545041355532555398291$. They used different sieving algorithms but the same polynomial pair, the same smoothness bounds and the same sieving region for the polynomials a. With these parameters, the Murphy E value computed with our description is about $2^{-24.5}$ with Zajac's parameters and $2^{-21.6}$ for Hayasaka's parameters. The difference between the two is due to the special-$\mathfrak{Q}$ sieve.

We selected our own polynomials. In this small field, the best polynomial selection appears to be the asymmetric JLSV_1 method with the explicit Galois action of order 6 given by $x \mapsto -(2x+1)/(x-1)$. We chose $f_0 = x^6 + 91354x^5 + 228370x^4 - 20x^3 - 228385x^2 - 91348x + 1$ and $f_1 = 23667000x^6 + 6549182x^5 - 338632045x^4 - 473340000x^3 - 16372955x^2 + 135452818x + 23667000$. We selected the smoothness bounds and the sieving region in order to reduce the total sieving time. The sizes of norms are about 115 bits on the f_0-side and 117 bits on the f_1-side, after subtracting the contribution of the special-$\mathfrak{Q}$. We sieved all the prime ideals $\mathfrak{r}$ of inertia degree 1 less than 2^{19}. The thresholds are set to 2^{65}. We obtained 1312416 raw relations with 12.3% duplicates. These results are summarized in Table 8.4.

	Zajac [185]	Hayasaka et al. [94]	Our work
Polynomials	JLSV_0	JLSV_0	JLSV_1
Special-$\mathfrak{q}$ sieve	No	Yes	Yes
Enumeration algorithm	Line sieve	Line sieve and FK method in 3D	Line, plane and space sieves
Sieving region (global or per $\mathfrak{q}$)	$2^{19} \times 2^{14} \times 1149$	$2^8 \times 2^8 \times 2^6$	$2^7 \times 2^7 \times 2^6$
Smoothness bounds	$2^{22.64}$, $2^{22.64}$	$2^{22.64}$, $2^{22.64}$	2^{23}, 2^{23}
α values	1.7, 0	1.7, 0	-1.8, -11.5
Murphy-E	$2^{-24.5}$	$2^{-21.6}$	$2^{-20.1}$
Number of special-$\mathfrak{Q}$s	—	$2^{17.77}$	$2^{14.44}$
Order of the Galois action	—	—	6
Number of relations	1077984	937575	1151099
Number of needed relations	854833	893773	1128604
Timing (days)	24.13	21.94	0.90

Table 8.4 – Relation collections in $\mathbb{F}_{p^6}$ with $p = 1081034284409$.

Individual logarithm. We aim to compute the discrete logarithm of $c = x^5 + 314159265358979323 8x^4 + 462643383279502884 1x^3 + 971693993751058209 7x^2 + 494459230781640628 6x + 208998628034825342 1$ obtained from the decimals of π, in basis $g = x + 4$, in $\mathbb{F}_{p^6} = \mathbb{F}_p[x]/(\varphi(x))$, where $\varphi = f_0$ in this case. We found

$$\mathrm{vlog}(g) = 129187912983303781856450 \text{ and}$$

$$\mathrm{vlog}(c) = 284315950357331821900688,$$

so that $g^{h\,\mathrm{vlog}(c)} = c^{h\,\mathrm{vlog}(g)}$ in $\mathbb{F}_{p^6}$, where $h = (p^6 - 1)/\ell$ is the cofactor.

Computation for a 300-bit finite field

We choose $p = 1043035802846857$, a 50-bit prime, and a large factor of $p^6 - 1$ $\ell = 1087923686020386502029991931593$.

Polynomial selections. We report here the three different pairs of polynomials we used to build Table 4.3:

- Conjugation:
  - $f_0 = x^{12} + 12x^{11} + 40x^{10} - 20x^9 - 245x^8 - 200x^7 + 344x^6 + 592x^5 + 250x^4 - 20x^3 - 26x^2 + 1$;
  - $f_1 = 31943784x^6 + 201177002x^5 + 23785745x^4 - 638875680x^3 - 502942505x^2 - 9514298x + 31943784$.
- Sarkar–Singh with $d = 2$, yielding degrees $(8, 6)$:
  - $f_0 = 2x^8 - 2x^7 + 6x^6 - 4x^5 + 9x^4 - 4x^3 + 6x^2 - 2x + 2$;
  - $f_1 = 13305451020x^6 + 13068452527x^5 - 122274520263x^4 + 74260869388x^3 - 122274520263x^2 + 13068452527x + 13305451020$.
- Sarkar–Singh with $d = 3$, yielding degrees $(9, 6)$:
  - $f_0 = x^9 + 2x^8 - 5x^7 - 9x^6 + 13x^5 + 24x^4 - 2x^3 - 15x^2 - 7x - 1$;
  - $f_1 = 10266423024x^6 - 6028238612x^5 - 67420797690x^4 - 2036172080x^3 + 116716740730x^2 + 67626776756x + 10266423024$.

The JLSV_1 polynomial pair we used is $f_0 = x^6 - 867578x^5 - 2168960x^4 - 20x^3 + 2168945x^2 + 867584x + 1$ and $f_1 = 2404471680x^6 + 4874502674x^5 - 23880818515x^4 - 48089433600x^3 - 12186256685x^2 + 9552327406x + 2404471680$, with the same Galois action as for the smaller example.

Relation collection. Before performing this computation, we have compared four different polynomial selections, summarized in Section 4.1.3.

For the relation collection, we kept the sieving region and the smoothness bounds of the experiments and used the polynomials given by the asymmetric JLSV_1 polynomial selection. The $2^{14.7}$ special-$\mathfrak{Q}$s are set on the f_1-side. The sieving bounds equal to $2^{20.5}$ and the thresholds were set to 2^{80}. We obtained 4637772 raw relations that gave 4231562 unique relations after duplicate removal; there are 4129438 ideals in the factor bases. The relation collection time is 6.84 days.

Linear algebra. The block Wiedemann algorithm was used with parameters $m = 30$ and $n = 10$. The cumulated running times for the various steps of the algorithm were 32 core hours for the computation of the Krylov sequences, 3 core hours for the computation of the linear generator, and 4.5 core hours for the computation of the solution vector (on a `Xeon CPU E5520 @ 2.27GHz`). We got 3,650,023 logarithms of the factor bases.

Individual logarithm. Keeping the notations of the previous computation, $g = x + 5$, $c = x^5 + 3141592653589793238x^4 + 46264338327950288 41x^3 + 97169399375105820 97x^2 + 4944592307816406286x + 2089986280348253421$. We find

$$\text{vlog}(g) = 732699947206837604640731573271 \text{ and}$$

$$\text{vlog}(c) = 766651238054992225393286911609.$$

Computation for a 389-bit finite field

Polynomial selection and relation collection. We have selected $p = 31415926535897942161$, a 65-bit prime, yielding a 130-bit large prime factor of $p^6 - 1$ to be $\ell = 986960440108936476119700657858603407761$. For this computation, the asymmetric JLSV_1 polynomial selection with the same Galois action of order 6 seems to give anew the best polynomials in terms of Murphy-E value. The polynomial pair is chosen to be $f_0 = x^6 - 218117072x^5 - 545292695x^4 - 20x^3 + 545292680x^2 + 218117078x + 1$ and $f_1 = 288064804440x^6 + 1381090484642x^5 - 868245854995x^4 - 5761296088800x^3 - 3452726211605x^2 + 347298341998x + 288064804440$. We selected the sieving region to be $2^{10} \times 2^{10} \times 2^8$ and two smoothness bounds equal to 2^{28}. The $2^{18.7}$ special-$\mathfrak{Q}$s are set on the f_1 side and the average value of the norms are 2^{160} on the f_0-side and 2^{173} on the f_1-side. The sieving bounds are equal to 2^{21} and the thresholds are set to 2^{90}. The relation collection required 790 days to find 29428326 unique relations after the removal of less than 20.3% duplicates; this is greater than the 29261526 ideals in the factor bases. This computation was done with commit `da20cf...`.

Linear algebra. We used parameters $n = 10$ and $m = 20$ in the Block-Wiedemann implementation in CADO-NFS. The cumulated numbers of core-years for the various steps of the algorithm are 80 days for the Krylov sequences, 6 days for the linear generator computation, and 14 days for the final computation of the solution, which yielded the values of 19,805,202 logarithms of the factor bases.

Individual logarithm. Keeping the notations of the previous computation, $g = x + 3$, $c = x^5 + 3141592653589793238x^4 + 46264338327950288 41x^3 + 97169399375105820 97x^2 + 4944592307816406286x + 2089986280348253421$. We find

$$\text{vlog}(g) = 907665820983150820551985406251606874974 \text{ and}$$

$$\text{vlog}(c) = 594727449023976898713456336273989724540.$$

Computation for a 422-bit finite field

The prime p is made of the first 22 decimals of the RSA1024 challenge. We have $p = 1350664108659952233509$ and a large factor of $p^2 - p + 1$, a factor of $p^6 - 1$, is $\ell = 2802294215702278424000412713285495714623$. This example comes from the pairing context.

Polynomial selection and relation collection. For this computation, we select the sieving region to be $2^{10} \times 2^{10} \times 2^8$ for each special-$\mathfrak{q}$. Both smoothness bounds are equal to 2^{29} and sieving bounds are equal to 2^{21}. We set the $2^{23.6}$ special-$\mathfrak{Q}$s on the f_0-side whose norm are larger than the corresponding sieving bound. However, the Galois action of order 6 allows us to only consider $2^{21.1}$ special-$\mathfrak{Q}$s and deduce the relations possibly given by the 5 other special-$\mathfrak{Q}$s in the orbit for free. The average of the maximal norms is about 2^{151} on side 0 (the contribution of the special-$\mathfrak{Q}$s are removed) and 2^{203} on side 1. We found about 72 M unique relations in about 8400 days on a single core (see Table 8.5), after removing the 28.8% duplicates. The computation was ran using clusters of Grid'5000 (`https://www.grid5000.fr`) with a method close to the one described in Section 8.2.1.

We experimented a non-conventional trick. We designed two polynomials with balanced coefficient size but unbalanced α: we were lucky and got $\alpha(f_1) = -14.4$, but $\alpha(f_0) = -2,2$ only. We put the special-$\mathfrak{Q}$ on the side 0, so that the norm after removing the contribution of the special-$\mathfrak{Q}$ was of 142 to 191 bits. On side 1, the norm growed from 175 to 245 bits. The two sides are unbalanced, but because of the high effect of α on side 1 (equivalent to removing $\alpha/\log(2) = 48$ bits), we got enough relations. We increased the threshold on side 1 from 110 to 115 then 121.

Cluster	Threads/node	Time (days)
Xeon E5-2650 2.00 GHz, 8 cores/CPU, 2 CPUs/node	31	4803
Xeon E5-2630 v3 2.40 GHz, 8 cores/CPU, 2 CPUs/node	31	1981
Opteron 6164 HE 1.70 GHz, 12 cores/CPU, 2 CPUs/node	23	588
Xeon X3440 2.53GHz, 4 cores/CPU, 1 CPU/node	4	518
Xeon L5420 2.50 GHz, 4 cores/CPU, 2 CPUs/node	7	312
Xeon X5570 2.93 GHz, 4 cores/CPU, 2 CPUs/node	8	198

Table 8.5 – Time per machine.

Linear algebra. We used a combination of `Xeon E5-2630v3`, `E5-2650` and `E7-4850 v3 CPUs`, connected with `Infiniband FDR fabric`. The block Wiedemann algorithm was used with parameters $m = 30$ and $n = 10$. The cumulated running times for the various steps of the algorithm were 2.67 core-years for the computation of the Krylov sequences, 0.1 core-year for the computation of the linear generator, and 0.3 core-year for the computation of the solution vector.

Individual computation. Define $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+2)$. The curve $E/\mathbb{F}_{p^2} : y^2 = x^3 + b,\ b = i + 2$ is supersingular of trace p, hence of order $p^2 - p + 1$. Define $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[j]/(j^3 - b)$. The embedding field of the curve E is $\mathbb{F}_{p^6}$. We take $G_0 = (6, 875904596857578874580 + 221098138973401953062i)$ as a generator of $E(\mathbb{F}_{p^2})$, and $G_1 = [651]G_0$ is a generator of $E(\mathbb{F}_{p^2})[\ell]$. The distorsion map $\phi : (x, y) \mapsto (x^p/(jb^{(p-2)/3}), y^p/(b^{(p-1)/2}))$ gives a generator $G_2 = \phi(G_1)$ of the second dimension of the ℓ-torsion, $j \in \mathbb{F}_{p^6}$ is a cube root of b. We take the point $P_0 = (314159265358979323847 + 264338327950288419716i, 935658401868915145130 + 643077111364229171931i) \in E(\mathbb{F}_{p^2})$ from the decimals of π, and $P = 651P_0 \in E(\mathbb{F}_{p^2})[\ell]$ will be our challenge. We aim to compute the discrete logarithm of P in basis G_1. For doing so, we transfer the generator G_1 and the point P to $\mathbb{F}_{p^6}$, as $g = e_{\text{Tate}}(G_1, \phi(G_1))$ and $t = e_{\text{Tate}}(P_1, \phi(G_1))$. The initial splitting with Guillevic's algorithms [86, 88] gave a 40-bit smooth generator

$g^{545513} = uvw\ (-141849807327922 - 5453622801413x + 541464063196 59x^2)$ where $u \in \mathbb{F}_{p^2}, v \in \mathbb{F}_{p^3}, w \in \mathbb{F}_p$ so that their logarithm modulo ℓ is zero. The norm of the latter term is 40-bit smooth and its factorization is equal to $3^3 \cdot 7^2 \cdot 11^2 \cdot 17 \cdot 317 \cdot 35812537 \cdot 16941885101 \cdot 17450874689 \cdot 22088674079 \cdot 35134635829 \cdot 85053580259 \cdot 144278841431 \cdot 1128022180423 \cdot 2178186439939$.

We also got a 44-bit smooth challenge: $g^{58779}t = uvw(-137392843659670 - 34918302724509x + 13401171220212x^2)$. The norm of the latter term is 44-bit smooth: $821 \cdot 3877 \cdot 6788447 \cdot 75032879 \cdot 292064093 \cdot 257269999897 \cdot 456432316517 \cdot 1029313376969 \cdot 3142696252889 \cdot 4321280585357 \cdot 18415984442663$.

We obtained that $\mathrm{vlog}(g) = 14636111560202813908403410352551744199 92$ and $\mathrm{vlog}(t) = 18004302008056970405325216125240295266 11$, so that $\log_g(t) = \mathrm{vlog}(t)/\mathrm{vlog}(g) \mod \ell = 75207848026896577063286973539798946459 2$.

8.2.4 Summary of the computations

With these five computations, we can add some rows to Table 1.1.

Finite field	Date	Bit size	Algorithm	Cost: CPU days	Authors
$\mathbb{F}^*_{p^6}$	2017	422	NFS	$9.52 \cdot 10^3$	Grémy, Guillevic, Morain and Thomé [85]
	2017	389	NFS	890	Grémy, Guillevic, Morain and Thomé [85]
$\mathbb{F}^*_{p^5}$	2017	324	NFS	386	Grémy, Guillevic and Morain [84]

Table 8.6 – Discrete logarithm records on finite fields (complement to Table 1.1).

Conclusion

In this work, we have presented and studied variants of the number field sieve algorithm that compute efficiently discrete logarithms in medium characteristic finite fields. Among the four main steps of these algorithms, we focused on the relation collection by presenting sieve algorithms in small dimensions, thus providing an efficient way to perform this step. Even if the discrete logarithm problem was at the heart of the emergence of the public-key cryptography, the attention has been more focused on the factorization algorithms for a long time. But this situation changed in the last few years, increasing the demand for collecting relations in dimension larger than two.

For the classical version of NFS in medium characteristic, we have explained how to compute the quality criteria of Murphy, the α and Murphy E quantities, to take into account the specificities of the three-dimensional relation collection. We also described a modification of the JLSV_1 polynomial selection to take into account the special-$\mathfrak{Q}$-method by unbalancing the sizes of the coefficients with respect to the size of the special-$\mathfrak{Q}$s.

We have also described our implementation of the relation collection for the classical NFS algorithm, especially how the norms are initialized and how we implemented the three sieve algorithms (line, plane and space sieves) we need in three dimensions. Two of them, the generalized line and plane sieves, are furthermore described and implemented to enumerate elements in lattices of any dimensions.

Our implementation, combined with the quality criteria and the unbalanced JLSV_1 polynomial selection, allowed us to perform the relation collection for five computations of discrete logarithms. A first computation in $\mathbb{F}_{p^6}$ redo the record of the literature in less time, and the three others establish new records, the largest one a 422-bit $\mathbb{F}_{p^6}$. The implementation was also used to compute a discrete logarithm in a 324-bit $\mathbb{F}_{p^5}$, the first computation in this extension.

Finally, we proposed a general framework to sieve in any small dimensions, where the three three-dimensional sieves are particular cases. In this general framework, we described and analyzed two algorithms. We introduce the notion of transition vector to generalize the vectors produced in Franke-Kleinjung's algorithm, and a weaker notion called nearly-transition vector. The major difference between the two proposed algorithm is the building of the nearly-transition-vectors. Because of the pattern of nearly-transition-vector, this implies a modification of the enumeration step, when no nearly-transition-vector allows to reach a new valid point, called fall-back strategy. The `globalntvgen` algorithm generates nearly-transition-vectors thanks to a skew basis reduction on the whole basis of the lattice and always calls the fall-back strategy. The `localntvgen` algorithm generates nearly-transition-vectors by mixing a skew

basis reduction and some closest vector computations, and rarely calls an aggressive fall-back strategy. The generalized line sieve is a particular case of both algorithms, the generalized plane sieve and the space sieve are particular case of `localntvgen`.

Perspectives

Sieve algorithms and relation collection. We have proposed two generic algorithms to sieve in any small dimensions. Even though we propose a Sage implementation to do some experiments on these algorithms, challenges still remain which are:

- the initialization step with a Graver basis: computing a Graver basis of a lattice is costly, but our problem is bounded and we can maybe modify the algorithm to take into account these bounds and then have a faster than expected initialization step to have the best nearly-transition-vectors,
- the initialization step: we have described initialization steps for both algorithms, highly dependent on the results of a skew basis reduction, but there exist maybe other mechanisms to produce nearly-transition-vectors of better quality,
- the fall-back strategies: in `globalntvgen`, the strategy seems unnecessary, which is not expected and we wonder if this strategy is really needed theoretically,
- the running time: there does not exist an efficient implementation of both algorithms to compare the running time to an existing algorithm (for example, the enumeration of element of a lattice in a sphere, as in [92, Algorithm 10], or the generalized plane sieve),
- the correctness: in our Sage implementation, we only focus on the correctness, and therefore use a probably too large number of calls to LLL or skew LLL (especially for `globalntvgen`), but it impacts necessarily the running time.

Besides the questions about the enumeration part itself, a problem occurs in the relation collection with the initialization of the norms. If the algorithm we proposed in Chapter 7 works for any dimension, its efficiency and accuracy are not guaranteed for dimensions higher than 4. It is highly probable that, in dimension 6 and higher, the cost of the initialization of norms will outmatch the cost of sieving, whereas, classically, the initialization of the norms and the cofactorization steps are negligible.

Implementation of the three-dimensional sieves. Even if our implementation allowed us to perform some record computations, it is not highly optimized, and there exist many ways to improve our work:

- some routines can be implemented differently to provide a speed-up, such as the computation of the Franke–Kleinjung vectors, which can be written in assembly language, or the three-dimensional LLL, which can be especially implemented for this dimension,

- marking the hits can be done by using bucket sieving, which is more efficient than updating each hit in the array, due to memory accesses that are more cache-friendly,
- the storage of the small factors that divide a norm may allow us to decrease the time spent in the cofactorization step.

ExTNFS. The exTNFS algorithm is for now the algorithm that provides the best complexity to compute discrete logarithms in finite fields of medium characteristics. Even if we omit the questions about the relation collection listed above, proposing an implementation of this algorithm does not seem to be easy. In addition to the challenges we have listed in Chapter 5.

First, the number of polynomial selection methods available to define the number field is large, and we need to have at least a Murphy-E like function to distinguish some of the best pairs. As the coefficients of the polynomials are algebraic integers, it is also needed to develop algorithms that enumerate and produce good polynomials: the description of a generic α-value is therefore necessary.

Everything seems under control in the linear algebra and in the computation of an individual logarithm. But, it is also what was said about these steps for NFS-HD and we discovered during our computations several difficulties.

Special-$\mathfrak{Q}$ method. In all dimensions, the special-$\mathfrak{Q}$ method can be refined, due to the quantity of parameters we need to tune. For example, it is widely believed that the special-$\mathfrak{Q}$s must be larger than the sieving bound, not only for practical reasons, but also because the number of duplicates using such special-$\mathfrak{Q}$s was expected to be large: the last experiment using the CADO-NFS software seems to invalidate this intuition (see for example commit `f8350cb...`).

An idea we found interesting to explore, especially in the context of (M)NFS in higher dimension, is the possibility to force special-$\mathfrak{Q}$s in several sides, instead of one side. The advantage is expected to be more visible when the norms in some sides are balanced: instead of strongly decreasing the norm in one side only, we will decrease the norms in all the sides, keeping the balancedness. To build the equivalent of the special-$\mathfrak{Q}$-lattice where there is only one special-$\mathfrak{Q}$, it suffices to compute the intersection of all the special-$\mathfrak{Q}$-lattices, which can be done by solving a system of congruences thanks to the Chinese remainder theorem.

Another perspective is to put two, or more, special-$\mathfrak{Q}$s on the same side [38]: it may help to build a matrix with the left columns as sparse as possible, implying that the filtering step can produce smaller matrices than expected.

It seems anyway that the relation collection step, and especially the special-$\mathfrak{Q}$ method, have more freedom than what we consider in this thesis.

MNFS. Despite its theoretical advantage to compute discrete logarithms, there is no reported record using MNFS. This can be explained by some reasons:

- the constraints for the polynomial selection about the size of the coefficients, a common Galois action, a negative α-value are difficult to reach with two polynomials, and even more difficult for more than two polynomials,

- the sizes of the actual records are maybe too small to observe a practical gain by using more than two number fields,
- most of the implementations of NFS use two number fields, with some optimizations, especially during the cofactorization step, difficult to translate to multiple number fields.

Even if the practical gain of using exTNFS seems to be more promising than using MNFS, it can be still attractive to deal with MNFS, if the challenge about the polynomial selection is solved, that is finding three or four good polynomials instead of two. First, the effort to have an implementation of MNFS seems less significant than for exTNFS. Starting from an implementation of NFS, the sieve algorithms are already available, the main change arise in the cofactorization step. Packing the matrix needs certainly some work to take into account the columns coming from the new number fields and the related Schirokauer maps. The descent can be done by considering only two number fields, at least for a first computation. Secondly, some records, as the one of Gaudry, Guillevic and Morain in $\mathbb{F}_{p^3}$ [73], use almost all the possible special-$\mathfrak{Q}$s range in both sides: a larger computation will need certainly to increase the smoothness bounds, if two number fields are used, but keeping the same smoothness bounds and using a third number field can be considered. Finally, the exTNFS algorithm have also multiple variant: understanding the behavior of MNFS in a simpler context can be a first step to understand the impact of the multiple variant of exTNFS.

Bibliography

[1] Adj, G., Canales-Martínez, I., Cruz-Cortés, N., Menezes, A., Oliveira, T., Rivera-Zamarripa, L., Rodríguez-Henríquez, F.: Computing discrete logarithms in cryptographically-interesting characteristic-three finite fields. Cryptology ePrint Archive, Report 2016/914 (2016), announcement available at the NMBRTHRY archives, item 004923. Cited page 21.

[2] Adleman, L.: A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In: 20th Annual Symposium on Foundations of Computer Science. pp. 55–60. IEEE (1979). Cited page 16.

[3] Adleman, L.: The function field sieve. In: Adleman, L., Huang, M.D. (eds.) ANTS-I. LNCS, vol. 877, pp. 108–121. Springer (1994). Cited page 19.

[4] Adleman, L., DeMarrais, J.: A subexponential algorithm for discrete logarithms over all finite fields. Mathematics of Computation 61(203), 1–15 (1993). Cited pages 61, 204.

[5] Adleman, L., Huang, M.D.: Function Field Sieve Method for Discrete Logarithms over Finite Fields. Information and Computation 151(1), 5–16 (1999). Cited page 19.

[6] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In: 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 5–17. Computer and Communications Security 2015, ACM (2015). Cited pages 2, 23, 150.

[7] Agence nationale de la sécurité des systèmes d'information: Référentiel général de sécurité. Tech. Rep. version 2.0, Premier ministre, République Française (February 2014). Cited page 2.

[8] ANSI X9.42-2003 (R2013): Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. Tech. rep., American Bankers Association (2003). Cited page 9.

[9] Atkin, A., Bernstein, D.: Prime sieves using binary quadratic forms. Mathematics of Computation 73(246), 1023–1030 (2004). Cited pages 24, 30, 31.

[10] Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica 6(1), 1–13 (1986). Cited page 120.

[11] Bai, S.: Polynomial Selection for the Number Field Sieve. Ph.D. thesis, The Australian National University (2011). Cited page 188.

[12] Bai, S., Bouvier, C., Kruppa, A., Zimmermann, P.: Better polynomials for GNFS. Mathematics of Computation 85(298), 861–873 (2016). Cited page 42.

[13] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H.: Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. Society for Industrial and Applied Mathematics (2000). Cited page 50.

[14] Barbulescu, R.: Algorithmes de logarithmes discrets dans les corps finis. Ph.D. thesis, Université de Lorraine (2013). Cited page 54.

[15] Barbulescu, R.: New record in $\mathbb{F}_{p^3}$. Slides presented at the CATREL-workshop (2015), `https://webusers.imj-prg.fr/~razvan.barbaud/p3dd52.pdf`. Cited page 70.

[16] Barbulescu, R., Bos, J., Bouvier, C., Kleinjung, T., Montgomery, P.: Finding ECM-friendly curves through a study of Galois properties. The Open Book Series 1(1), 63–86 (2013). Cited page 142.

[17] Barbulescu, R., Bouvier, C., Detrey, J., Gaudry, P., Jeljeli, H., Thomé, E., Videau, M., Zimmermann, P.: Discrete Logarithm in GF(2^{809}) with FFS. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 221–238. Springer (2014), announcement available at the NMBRTHRY archives, item 004534. Cited page 21.

[18] Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. Cryptology ePrint Archive, Report 2017/334 (2017). Cited page 92.

[19] Barbulescu, R., Gaudry, P., Guillevic, A., Morain, F.: Improvements to the number field sieve for non-prime finite fields (2014), `https://hal.inria.fr/hal-01052449`, preprint. Cited pages 21, 67, 70, 72, 81.

[20] Barbulescu, R., Gaudry, P., Guillevic, A., Morain, F.: Improving NFS for the discrete logarithm problem in non-prime finite fields. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 129–155. Springer (2015). Cited pages 3, 20, 61, 62, 66, 68, 69, 70.

[21] Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic. In: Nguyen, P., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 1–16. Springer (2014). Cited pages 3, 19.

[22] Barbulescu, R., Gaudry, P., Kleinjung, T.: The Tower Number Field Sieve. In: Iwata, T., Cheon, J. (eds.) ASIACRYPT 2015. LNCS, vol. 9453, pp. 31–55. Springer (2015). Cited pages 5, 20, 83, 84, 85, 88, 152, 208.

[23] Barbulescu, R., Lachand, A.: Some mathematical remarks on the polynomial selection in NFS. Mathematics of Computation 86(303), 397–418 (2017). Cited pages 40, 63.

[24] Barbulescu, R., Pierrot, C.: The Multiple Number Field Sieve for Medium and High Characteristic Finite Fields. LMS Journal of Computation and Mathematics 17, 230–246 (2014). Cited pages 5, 20, 58, 79, 80, 86, 191, 192.

[25] Barreto, P., Lynn, B., Scott, M.: Constructing Elliptic Curves with Prescribed Embedding Degrees. In: Cimato, S., Persiano, G., Galdi, C. (eds.) Security in Communication Networks 2002. LNCS, vol. 2576, pp. 257–267. Springer (2003). Cited page 91.

[26] Barreto, P., Naehrig, M.: Pairing-Friendly Elliptic Curves of Prime Order. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 319–331. Springer (2006). Cited page 91.

[27] Bays, C., Hudson, R.: The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}. BIT Numerical Mathematics 17(2), 121–127 (1977). Cited page 26.

[28] Berlekamp, E.: Algebraic Coding Theory. McGraw-Hill series in systems science, World Scientific (2015). Cited page 51.

[29] Bernstein, D.: primegen, `http://cr.yp.to/primegen.html`. Cited page 31.

[30] Bernstein, D.: How to find small factors of integers (2002), `https://cr.yp.to/papers.html`. Cited page 142.

[31] Bernstein, D., Birkner, P., Lange, T., Peters, C.: ECM using Edwards curves. Mathematics of Computation 82(282), 1139–1179 (2013). Cited page 142.

[32] Bernstein, D., Lange, T.: Two grumpy giants and a baby. The Open Book Series 1(1), 87–111 (2013). Cited page 14.

[33] Bernstein, D., Lange, T., Niederhagen, R.: Dual EC: A Standardized Back Door. In: Ryan, P., Naccache, D., Quisquater, J.J. (eds.) The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday. LNCS, vol. 9100, pp. 256–281. Springer (2016). Cited page 23.

[34] Bistritz, Y., Lifshitz, A.: Bounds for resultants of univariate and bivariate polynomials. Linear Algebra and its Applications 432(8), 1995–2005 (2010). Cited pages 39, 43, 58, 72, 128, 129.

[35] Blake, I., Seroussi, G., Smart, N.: Advances in Elliptic Curve Cryptography, vol. 317. Cambridge University Press (2005). Cited page 11.

[36] Boneh, D., Franklin, M.: Identity-Based Encryption from the Weil Pairing. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 213–229. Springer (2001). Cited page 12.

[37] Boneh, D., Lynn, B., Shacham, H.: Short Signatures from the Weil Pairing. Journal of Cryptology 17(4), 297–319 (2004). Cited page 12.

[38] Boudot, F.: On Improving Integer Factorization and Discrete Logarithm Computation using Partial Triangulation. Cryptology ePrint Archive, Report 2017/758 (2017). Cited page 163.

[39] Bouillaguet, C., Delaplace, C.: Sparse Gaussian Elimination modulo p: An Update. In: Gerdt, V., Koepf, W., Seiler, W., Vorozhtsov, E. (eds.) CASC 2016. LNCS, vol. 9890, pp. 101–116. Springer (2016). Cited page 50.

[40] Bouvier, C., Gaudry, P., Imbert, L., Jeljeli, H., Thomé, E.: Discrete logarithms in GF(p) — 180 digits (June 2014), announcement available at the NMBRTHRY archives, item 004703. Cited pages 21, 42, 46, 47, 48.

[41] Bouvier, C.: Algorithmes pour la factorisation d'entiers et le calcul de logarithme discret. Ph.D. thesis, Université de Lorraine (2015). Cited page 49.

[42] Brent, R.: The first occurrence of large gaps between successive primes. Mathematics of Computation 27(124), 959–963 (1973). Cited page 26.

[43] Buhler, J., Lenstra, H., Pomerance, C.: Factoring Integers with the Number Field Sieve. In: Lenstra, A., Lenstra, H. (eds.) The Development of the Number Field Sieve. Lecture Notes in Mathematics, vol. 1554, pp. 50–94. Springer (1993). Cited page 44.

[44] Canfield, E.R., Erdös, P., Pomerance, C.: On a problem of Oppenheim concerning "factorisatio numerorum". Journal of Number Theory 17(1), 1–28 (1983). Cited page 16.

[45] Cavallar, S.: On the Number Field Sieve Integer Factorisation Algorithm. Ph.D. thesis, University of Leiden (2002). Cited pages 34, 49.

[46] Cohen, H.: A Course in Computational Algebraic Number Theory, Graduate Texts in Mathematics, vol. 138. Springer (2000). Cited pages 189, 201.

[47] Commeine, A., Semaev, I.: An Algorithm to Solve the Discrete Logarithm Problem with the Number Field Sieve. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 174–190. Springer (2006). Cited pages 20, 43, 52, 58, 206.

[48] Coppersmith, D.: Modifications to the number field sieve. Journal of Cryptology 6(3), 169–180 (1993). Cited pages 20, 51, 58, 206.

[49] Coppersmith, D.: Fast evaluation of logarithms in fields of characteristic two. IEEE Transactions on Information Theory 30(4), 587–594 (1984). Cited pages 3, 19.

[50] Coppersmith, D.: Solving linear equations over $GF(2)$: block Lanczos algorithm. Linear Algebra and its Applications 192, 33–60 (1993). Cited page 50.

[51] Coppersmith, D.: Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. Mathematics of Computation 62(205), 333–350 (1994). Cited pages 50, 52.

[52] Coppersmith, D., Odlzyko, A., Schroeppel, R.: Discrete logarithms in $GF(p)$. Algorithmica 1(1), 1–15 (1986). Cited page 42.

[53] Davis, J., Holdridge, D.: Factorization Using the Quadratic Sieve Algorithm. In: Chaum, D. (ed.) CRYPTO 1983. pp. 103–113. LNCS, Springer (1984). Cited page 46.

[54] Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor (2008), `http://www.rfc-editor.org/rfc/rfc1654.txt`. Cited page 9.

[55] Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976). Cited pages 1, 9, 204.

[56] Dunten, B., Jones, J., Sorenson, J.: A space-efficient fast prime number sieve. Information Processing Letters 59(2), 79–84 (1996). Cited page 31.

[57] Eberly, W., Kaltofen, E.: On Randomized Lanczos Algorithms. In: ISSAC 1997. pp. 176–183. ACM (1997). Cited page 50.

[58] Ekkelkamp, W.: On the Amount of Sieving in Factorization Methods. Ph.D. thesis, University of Leiden (2010). Cited page 34.

[59] ElGamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: Blakley, G., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer (1985). Cited pages 1, 9.

[60] Elkenbracht-Huizing, M.: An implementation of the number field sieve. Experimental Mathematics 5(3), 231–253 (1996). Cited page 38.

[61] Franke, J., Kleinjung, T.: Continued fractions and lattice sieving. In: SHARCS 2005 (2005), `http://www.sharcs.org/`. Cited pages 45, 96, 97, 98, 208, 209.

[62] Franke, J., Kleinjung, T.: Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers. In: SHARCS 2006 (2006), `http://www.sharcs.org/`. Cited page 142.

[63] Freeman, D., Scott, M., Teske, E.: A Taxonomy of Pairing-Friendly Elliptic Curves. Journal of Cryptology 23, 224–280 (2010). Cited page 80.

[64] Frey, G., Rück, H.G.: A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. Mathematics of Computation 62(206), 865–874 (1994). Cited page 11.

[65] Fried, J., Gaudry, P., Heninger, N., Thomé, E.: A Kilobit Hidden SNFS Discrete Logarithm Computation. In: Coron, J.S., Nielsen, J. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 202–231. Springer (2017), announcement available at the NMBRTHRY archives, item 004934. Cited pages 21, 23, 52, 55, 58, 150.

[66] Galbraith, S.: Mathematics of Public Key Cryptography. Cambridge University Press (2012). Cited pages 12, 15, 180.

[67] Galbraith, S., Pollard, J., Ruprai, R.: Computing discrete logarithms in an interval. Mathematics of Computation 82(282), 1181–1195 (2013). Cited page 15.

[68] Galbraith, S., Wang, P., Zhang, F.: Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. Cryptology ePrint Archive, Report 2015/605 (2015). Cited page 14.

[69] Galway, W.: Robert Bennion's "hopping sieve". In: Buhler, J. (ed.) ANTS-III. LNCS, vol. 1423, pp. 169–178. Springer (1998). Cited page 31.

[70] Galway, W.: Dissecting a Sieve to Cut Its Need for Space. In: Bosma, W. (ed.) ANTS-IV. LNCS, vol. 1838, pp. 297–312. Springer (2000). Cited page 31.

[71] Galway, W.F.: Analytic Computation of the Prime-Counting Function. Ph.D. thesis, University of Illinois (2004). Cited page 31.

[72] Gaudry, P., Grémy, L., Videau, M.: Collecting relations for the Number Field Sieve in $GF(p^6)$. LMS Journal of Computation and Mathematics 19, 332–350 (2016). Cited pages 4, 63, 67, 73, 104, 123, 124, 128, 143, 152, 188, 207, 208, 210, 212.

[73] Gaudry, P., Guillevic, A., Morain, F.: Discrete logarithm record in GF(p^3) of 592 bits (180 decimal digits) (August 2016), announcement available at the NMBRTHRY archives, item 004930. Cited pages 21, 70, 164.

[74] Golub, G., Van Loan, C.: Matrix Computations (3rd Ed.). Johns Hopkins University Press, Baltimore (1996). Cited page 50.

[75] González, Á.: Measurement of Areas on a Sphere Using Fibonacci and Latitude–Longitude Lattices. Mathematical Geosciences pp. 42–49 (2010). Cited page 64.

[76] Gordon, D.: Designing and Detecting Trapdoors for Discrete Log Cryptosystems. In: Brickell, E. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 66–75. Springer (1993). Cited pages 20, 23, 57, 58, 86.

[77] Gordon, D.: Discrete Logarithms in GF(p) Using the Number Field Sieve. SIAM Journal on Discrete Mathematics 6(1), 124–138 (1993). Cited pages 2, 20, 205.

[78] Gordon, D.M., McCurley, K.S.: Massively parallel computation of discrete logarithms. In: Brickell, E.F. (ed.) CRYPTO 1992. pp. 312–323. Springer (1993). Cited page 47.

[79] Granger, R., Vercauteren, F.: On the discrete logarithm problem on algebraic tori. In: Shoup, V. (ed.) CRYPTO 2005. pp. 66–85. LNCS, Springer (2005). Cited page 91.

[80] Granger, R., Kleinjung, T., Zumbrägel, J.: On the discrete logarithm problem in finite fields of fixed characteristic. Transactions of the American Mathematical Society Accepted. Cited pages 3, 19.

[81] Granger, R., Kleinjung, T., Zumbrägel, J.: Discrete Logarithms in GF(2ˆ9234) (January 2014), announcement available at the NMBRTHRY archives, item 004666. Cited page 21.

[82] Graver, J.: On the foundations of linear and integer linear programming I. Mathematical Programming 9(1), 207–226 (1975). Cited page 101.

[83] Grémy, L., Guillevic, A.: DiscreteLogDB, a database of computations of discrete logarithms (2017), `https://gitlab.inria.fr/dldb/discretelogdb`. Cited pages 5, 21, 205.

[84] Grémy, L., Guillevic, A., Morain, F.: Breaking DLP in $GF(p^5)$ using 3-dimensional sieving (2017), preprint, 7 pages. Cited pages 4, 143, 146, 152, 160, 211, 213.

[85] Grémy, L., Guillevic, A., Morain, F., Thomé, E.: Computing discrete logarithms in $GF(p^6)$. In: SAC 2017. LNCS, Springer, to appear. Cited pages 4, 143, 146, 152, 160, 211, 213.

[86] Guillevic, A.: Computing individual discrete logarithms faster in GF(p^n) with the NFS-DL algorithm. In: Iwata, T., Cheon, J. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 149–173. Springer (2015). Cited pages 62, 76, 144, 159, 212.

[87] Guillevic, A.: Individual Discrete Logarithm in GF(p^k). Slides presented at the CATREL-workshop (2015), `http://www.lix.polytechnique.fr/Labo/Aurore.Guillevic/catrel-workshop/Aurore_Guillevic_CATRELworkshop.pdf`. Cited pages 21, 67.

[88] Guillevic, A.: Faster individual discrete logarithms with the QPA and NFS variants. Cryptology ePrint Archive, Report 2016/684 (2017). Cited pages 54, 62, 76, 77, 87, 144, 159, 212.

[89] Guillevic, A., Morain, F.: Discrete Logarithms. In: El Mrabet, N., Joye, M. (eds.) Guide to pairing-based cryptography, p. 42. CRC Press - Taylor and Francis Group (2016). Cited page 3.

[90] Guillevic, A., Morain, F.: Breaking a dlp on a 170-bit n=3 mnt curve. (April 2016), announcement available at the NMBRTHRY archives, item 004900. Cited page 67.

[91] Guillevic, A., Morain, F., Thomé, E.: Solving discrete logarithms on a 170-bit MNT curve by pairing reduction. In: SAC 2016. LNCS, Springer, to appear. Cited pages 70, 73, 143, 146.

[92] Hanrot, G., Pujol, X., Stehlé, D.: Algorithms for the Shortest and Closest Lattice Vector Problems. In: Chee, Y., Guo, Z., Ling, S., Shao, F., Tang, Y., Wang, H., Xing, C. (eds.) International Workshop Coding and Cryptology 2011. LNCS, vol. 6639, pp. 159–190. Springer (2011). Cited pages 138, 162.

[93] Hayasaka, K., Aoki, K., Kobayashi, T., Takagi, T.: An Experiment of Number Field Sieve for Discrete Logarithm Problem over $GF(p^{12})$. In: Fischlin, M., Katzenbeisser, S. (eds.) Number Theory and Cryptography. LNCS, vol. 8260, pp. 108–120. Springer (2013). Cited pages 4, 21, 66, 75, 153, 208.

[94] Hayasaka, K., Aoki, K., Kobayashi, T., Takagi, T.: A construction of 3-dimensional lattice sieve for number field sieve over $\mathbb{F}_{p^n}$. Cryptology ePrint Archive, 2015/1179 (2015). Cited pages 75, 104, 113, 124, 148, 156, 208.

[95] Hellman, M.: An overview of public key cryptography. IEEE Communications Magazine 40(5), 42–49 (2002). Cited pages 1, 204.

[96] Hildebrand, A., Tenenbaum, G.: Integers without large prime factors. Journal de théorie des nombres de Bordeaux 5(2), 411–484 (1993). Cited page 16.

[97] Jeljeli, H.: Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis. Ph.D. thesis, Université de Lorraine (2015). Cited pages 47, 50.

[98] Joux, A., Pierrot, C.: Nearly sparse linear algebra and application to discrete logarithms computations. In: Canteaut, A., Effinger, G., Huczynska, S., Panario, D., Storme, L. (eds.) Contemporary Developments in Finite Fields and Applications, pp. 119–144. World Scientific Publishing Company (2016). Cited page 52.

[99] Joux, A.: A One Round Protocol for Tripartite Diffie–Hellman. Journal of Cryptology 17(4), 263–276 (2004). Cited page 12.

[100] Joux, A.: Discrete Logarithms in GF(2^6168) [=GF((2^257)^24)] (May 2013), announcement available at the NMBRTHRY archives, item 004544. Cited page 21.

[101] Joux, A.: Faster Index Calculus for the Medium Prime Case Application to 1175-bit and 1425-bit Finite Fields. In: Johansson, T., Nguyen, P. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 177–193. Springer (2013), announcements available at the NMBRTHRY archives, item 004451 and item 004458. Cited page 21.

[102] Joux, A.: A New Index Calculus Algorithm with Complexity $L(1/4+o(1))$ in Small Characteristic. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. vol. 8282, pp. 355–379. Springer (2014). Cited page 19.

[103] Joux, A., Lercier, R.: Improvements to the General Number Field Sieve for discrete logarithms in prime fields. Mathematics of Computation 72(242), 953–967 (2003). Cited pages 42, 54.

[104] Joux, A., Lercier, R.: Discrete logarithms in GF(p) — 130 digits (June 2005), announcement available at the NMBRTHRY archives, item 002778. Cited page 66.

[105] Joux, A., Lercier, R.: The Function Field Sieve in the Medium Prime Case. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 254–270. Springer (2006). Cited page 19.

[106] Joux, A., Lercier, R., Smart, N., Vercauteren, F.: The Number Field Sieve in the Medium Prime Case. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 326–344. Springer (2006). Cited pages 3, 20, 61, 66, 67, 68, 91, 206.

[107] Joux, A., Odlyzko, A., Pierrot, C.: The Past, Evolving Present, and Future of the Discrete Logarithm. In: Koç, Ç. (ed.) Open Problems in Mathematics and Computational Science, pp. 5–36. Springer (2014). Cited page 3.

[108] Joux, A., Pierrot, C.: The Special Number Field Sieve in $\mathbb{F}_{p^n}$. In: Cao, Z., Zhang, F. (eds.) Pairing 2013. LNCS, vol. 8365, pp. 45–61. Springer (2013). Cited pages 20, 80.

[109] Joux, A., Pierrot, C.: Improving the Polynomial time Precomputation of Frobenius Representation Discrete Logarithm Algorithms. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 378–397. Springer (2014). Cited page 19.

[110] Joux, A., Pierrot, C.: Discrete logarithm record in characteristic 3, GF(3^(5*479)) a 3796-bit field (September 2014), announcement available at the NMBRTHRY archives, item 004745. Cited page 21.

[111] Kachisa, E., Schaefer, E., Scott, M.: Constructing Brezing-Weng Pairing-Friendly Elliptic Curves Using Elements in the Cyclotomic Field. In: Galbraith, S., Paterson, K. (eds.) Pairing 2008. LNCS, vol. 5209, pp. 126–135. Springer (2008). Cited page 91.

[112] Kaltofen, E.: Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation 64(210), 777–806 (1995). Cited pages 50, 51.

[113] Kerry, C., Gallagher, P.: FIPS PUB 186-4: Digital Signature Standard (DSS). Tech. rep., NIST (2013). Cited pages 11, 22, 23.

[114] Kim, T., Barbulescu, R.: Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9814, pp. 543–571. Springer (2016). Cited pages 3, 20, 83, 87, 152, 208, 211.

[115] Kim, T., Jeong, J.: Extended Tower Number Field Sieve with Application to Finite Fields of Arbitrary Composite Extension Degree. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 388–408. Springer (2017). Cited pages 87, 211.

[116] Kleinjung, T.: On polynomial selection for the general number field sieve. Mathematics of Computation 75(256), 2037–2047 (2006). Cited page 42.

[117] Kleinjung, T.: Polynomial Selection. Slides presented at the CADO workshop on integer factorization (2008), `http://cado.gforge.inria.fr/workshop/slides/kleinjung.pdf`. Cited page 42.

[118] Kleinjung, T.: Filtering and the matrix step in NFS. Slides presented at the Workshop on Computational Number Theory on the occasion of Herman te Riele's retirement from CWI Amsterdam (2011), `https://event.cwi.nl/wcnt2011/slides/kleinjung.pdf`. Cited page 51.

[119] Kleinjung, T.: Discrete Logarithms in GF(2^1279) (October 2014), announcement available at the NMBRTHRY archives, item 004751. Cited page 21.

[120] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A., Thomé, E., Bos, J., Gaudry, P., Kruppa, A., Montgomery, P., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit rsa modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer (2010). Cited pages 20, 49.

[121] Kleinjung, T., Bos, J., Lenstra, A.: Mersenne factorization factory. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 358–377. Springer (2014). Cited page 51.

[122] Kleinjung, T., Diem, C., Lenstra, A., Priplata, C., Stahlke, C.: Computation of a 768-Bit Prime Field Discrete Logarithm. In: Coron, J.S., Nielsen, J. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 185–201. Springer (2017). Cited pages 20, 21, 41, 42, 142, 147.

[123] Kraitchik, M.: Théorie des nombres, vol. 1. Gauthier-Villars Paris (1922). Cited pages 3, 204.

[124] LaMacchia, B., Odlyzko, A.: Solving Large Sparse Linear Systems Over Finite Fields. In: Menezes, A., Vanstone, S. (eds.) CRYPT0 1990. LNCS, vol. 537, pp. 109–133. Springer (1991). Cited page 49.

[125] Lanczos, C.: Solution of systems of linear equations by minimized iterations. Journal of Research of the National Bureau of Standards 49(1), 33–53 (1952). Cited page 50.

[126] Lenstra, A., Lenstra, H.: Algorithms in number theory. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, chap. 12, pp. 675–715. Elsevier (1990). Cited pages 2, 204.

[127] Lenstra, A., Hendrik, W.: The Development of the Number Field Sieve, Lecture Notes in Mathematics, vol. 1554. Springer (1993). Cited pages 2, 37, 205.

[128] Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982). Cited pages 181, 182.

[129] Lenstra, A., Verheul, E.: The XTR public key system. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 1–19. Springer (2000). Cited pages 12, 91.

[130] Lenstra, H.W.: Factoring Integers with Elliptic Curves. Annals of Mathematics 126(3), 649–673 (1987). Cited pages 18, 34.

[131] Massey, J.: Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory 15(1), 122–127 (1969). Cited page 51.

[132] Matyukhin, D.: On asymptotic complexity of computing discrete logarithms over $GF(p)$. Discrete Mathematics and Applications 13(1), 27–50 (2003). Cited page 58.

[133] Maurer, U.: Towards the Equivalence of Breaking the Diffie–Hellman Protocol and Computing Discrete Logarithms. In: Desmedt, Y. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 271–281. Springer (1994). Cited page 9.

[134] Menezes, A., Sarkar, P., Singh, S.: Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography. In: Phan, R., Yung, M. (eds.) Mycrypt 2016. LNCS, vol. 10311, pp. 83–108. Springer (2017). Cited pages 88, 92.

[135] Menezes, A.J., Okamoto, T., Vanstone, S.: Reducing elliptic curve logarithms to logarithms in a finite field. IEEE Transactions on Information Theory 39(5), 1639–1646 (1993). Cited page 11.

[136] Mertens, F.: Ein beitrag zur analytischen zahlentheorie. Journal für die reine und angewandte Mathematik 78, 46–62 (1874). Cited page 25.

[137] Miele, A., Bos, J., Kleinjung, T., Lenstra, A.: Cofactorization on Graphics Processing Units. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 335–352. Springer (2014). Cited page 142.

[138] Mitsunari, S., Sakai, R., Kasahara, M.: A New Traitor Tracing. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 85(2), 481–484 (2002). Cited page 12.

[139] Montgomery, P.: A Block Lanczos Algorithm for Finding Dependencies over $GF(2)$. In: Guillou, L., Quisquater, J.J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 106–120. Springer (1995). Cited page 50.

[140] Murphy, B.: Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm. Ph.D. thesis, The Australian National University (1999). Cited pages 40, 41, 42, 207.

[141] Nguyen, P., Stehlé, D.: An LLL algorithm with quadratic complexity. SIAM Journal on Computing 39(3), 874–903 (2009). Cited page 182.

[142] Onn, S.: Theory and Applications of n-Fold Integer Programming. In: Lee, J., Leyffer, S. (eds.) Mixed Integer Nonlinear Programming, IMA, vol. 154, pp. 559–593. Springer (2012). Cited page 102.

[143] Pierrot, C.: The Multiple Number Field Sieve with Conjugation and Generalized Joux–Lercier Methods. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 156–170. Springer (2015). Cited pages 3, 20, 61, 79, 80, 81, 84, 191.

[144] Pohlig, S., Hellman, M.: An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. IEEE Transactions on Information Theory 24(1), 106–110 (1978). Cited page 13.

[145] Pollard, J.: The lattice sieve. In: Lenstra, A., Lenstra, H. (eds.) The Development of the Number Field Sieve, Lecture Notes in Mathematics, vol. 1554, pp. 43–49. Springer (1993). Cited pages 45, 46, 206.

[146] Pollard, J.: Monte Carlo methods for index computation (mod p). Mathematics of Computation 32(143), 918–924 (1978). Cited page 14.

[147] Pomerance, C.: Analysis and comparison of some integer factoring algorithms. In: Lenstra, H., Tijdeman, R. (eds.) Computational Methods in Number Theory. Mathematical Centre Tracts, vol. 154, pp. 89–139. Mathematish Centrum (1982), `http://oai.cwi.nl/oai/asset/19571/19571A.pdf`. Cited pages 2, 54, 204.

[148] Pomerance, C.: A Tale of Two Sieves. Notices of the American Mathematical Society 43, 1473–1485 (1996). Cited pages 3, 93, 206.

[149] Pomerance, C., Selfridge, J., Wagstaff, S.: The Pseudoprime to $25 \cdot 10^9$. Mathematics of Computation 35(151), 1003–1026 (1980). Cited page 12.

[150] Pomerance, C., Smith, J.: Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes. Experimental Mathematics 1(2), 89–94 (1992). Cited page 49.

[151] Pritchard, P.: A sublinear additive sieve for finding prime number. Communications of the ACM 24(1), 18–23 (1981). Cited page 31.

[152] Pritchard, P.: Explaining the wheel sieve. Acta Informatica 17(4), 477–485 (1982). Cited page 31.

[153] Pritchard, P.: Fast compact prime number sieves (among others). Journal of Algorithms 4(4), 332–344 (1983). Cited page 29.

[154] Pritchard, P.: Linear prime-number sieves: A family tree. Science of Computer Programming 9(1), 17–35 (1987). Cited page 29.

[155] Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978). Cited page 1.

[156] Robert, B., Wagstaff, S.: Lucas Pseudoprimes. Mathematics of Computation 35(152), 1391–1417 (1980). Cited page 12.

[157] Rubin, K., Silverberg, A.: Torus-based cryptography. In: Boneh, D. (ed.) CRYPTO 2003, LNCS, vol. 2729, pp. 349–365. Springer (2003). Cited page 12.

[158] Sarkar, P., Singh, S.: A General Polynomial Selection Method and New Asymptotic Complexities for the Tower Number Field Sieve Algorithm. In: Cheon, J., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 37–62. Springer (2016). Cited pages 87, 211.

[159] Sarkar, P., Singh, S.: A Generalisation of the Conjugation Method for Polynomial Selection for the Extended Tower Number Field Sieve Algorithm. Cryptology ePrint Archive, Report 2016/537 (2016). Cited pages 87, 211.

[160] Sarkar, P., Singh, S.: New Complexity Trade-Offs for the (Multiple) Number Field Sieve Algorithm in Non-Prime Fields. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 429–458. Springer (2016). Cited pages 70, 80, 82, 206.

[161] Sarkar, P., Singh, S.: Tower number field sieve variant of a recent polynomial selection method. Cryptology ePrint Archive, Report 2016/401 (2016). Cited pages 87, 211.

[162] Schirokauer, O.: Using number fields to compute logarithms in finite fields. Mathematics of Computation 69(231), 1267–1283 (2000). Cited pages 20, 83, 208.

[163] Schirokauer, O.: Virtual logarithms. Journal of Algorithms 57, 140–147 (2005). Cited page 38.

[164] Schirokauer, O.: Discrete logarithms and local units. Philosophical Transactions of the Royal Society A 345(1676), 409–423 (1993). Cited pages 38, 43.

[165] Schirokauer, O.A.: On pro-finite groups and on discrete logarithms. Ph.D. thesis, University of California (1992). Cited page 20.

[166] Schnorr, C., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: Budach, L. (ed.) Fundamentals of Computation Theory 1991. LNCS, vol. 529, pp. 68–85. Springer (1991). Cited page 182.

[167] Shanks, D.: Class Number, a Theory of Factorization and Genera. In: Symposia in Pure Mathematics 1971. vol. 20, pp. 415–440 (1971). Cited page 13.

[168] Shoup, V.: NTL: A library for doing Number Theory. Available at www.shoup.net/ntl. Cited page 141.

[169] Shoup, V.: Lower Bounds for Discrete Logarithms and Related Problems. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 256–266. Springer (1997). Cited page 13.

[170] Shoup, V.: A Computational Introduction to Number Theory and Algebra. Cambridge University Press (2009). Cited page 21.

[171] Singleton, R.: Algorithm 357: An Efficient Prime Number Generator. Communications of the ACM 12(10), 563–564 (1969). Cited page 26.

[172] Smart, N.: Cryptography Made Simple. Information Security and Cryptography, Springer (2016). Cited page 180.

[173] Sorenson, J.: Trading time for space in prime number sieves. In: Buhler, J. (ed.) ANTS-III. LNCS, vol. 1423, pp. 179–195. Springer (1998). Cited pages 29, 31.

[174] Sorenson, J.: The Pseudosquares Prime Sieve. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS-VII. LNCS, vol. 4076, pp. 193–207. Springer (2006). Cited page 31.

[175] The 4ti2 team: 4ti2—a software package for algebraic, geometric and combinatorial problems on linear spaces. Available at www.4ti2.de. Cited page 102.

[176] The CADO-NFS Development Team: CADO-NFS, an implementation of the number field sieve algorithm (2017), `http://cado-nfs.gforge.inria.fr/`, development version. Cited pages 23, 49, 55, 126, 128, 155, 208, 210.

[177] The Sage Developers: SageMath, the Sage Mathematics Software System (Version 7.5.1) (2017), `http://www.sagemath.org`. Cited page 56.

[178] Thomé, E.: A modified block Lanczos algorithm with fewer vectors. In: Topics in Computational Number Theory inspired by Peter L. Montgomery. Cambridge University Press (2016). Cited page 50.

[179] Valenta, L., Adrian, D., Sanso, A., Cohney, S., Fried, J., Hastings, M., Halderman, J.A., Heninger, N.: Measuring small subgroup attacks against Diffie–Hellman. In: Network and Distributed System Security Symposium 2017. Internet Society (2017). Cited page 22.

[180] Valenta, L., Cohney, S., Liao, A., Fried, J., Bodduluri, S., Heninger, N.: Factoring as a service. In: Grossklags, J., Preneel, B. (eds.) Financial Cryptography and Data Security 2016. LNCS, vol. 9603, pp. 321–338. Springer (2017). Cited page 50.

[181] Vialla, B.: Contributions à l'Algèbre Linéaire Exacte sur Corps Finis et au Chiffrement Homomorphe. Ph.D. thesis, Université de Montpellier (2015). Cited page 50.

[182] Walisch, K.: primesieve, `http://primesieve.org`. Cited page 27.

[183] Wiedemann, D.: Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory 32(1), 54–62 (1986). Cited page 50.

[184] Williams, H.: A $p+1$ method of factoring. Mathematics of Computation 39(159), 225–234 (1982). Cited page 12.

[185] Zajac, P.: Discrete Logarithm Problem in Degree Six Finite Fields. Ph.D. thesis, Slovak University of Technology (2008), `http://www.kaivt.elf.stuba.sk/kaivt/Vyskum/XTRDL`. Cited pages 3, 4, 5, 21, 66, 71, 80, 118, 126, 148, 156, 191, 206, 208.

[186] Zajac, P.: On the use of the lattice sieve in the 3D NFS. Tatra Mountains Mathematical Publications 45, 161–172 (2010). Cited page 75.

[187] Zhu, Y., Zhuang, J., Lv, C., Lin, D.: Improvements on the individual logarithm step in extended tower number field sieve. Cryptology ePrint Archive, Report 2016/727 (2016). Cited page 87.

Appendix A

Background on lattices

Lattices are not the main interest of this thesis, but we use results on lattices in few chapters. We therefore recall a selection of results that will be useful in this thesis, especially in Chapter 3, Chapter 4, Chapter 5 and Chapter 6.

A.1 Lattice and basis reduction

In this section, we introduce some notions on lattices and basis reduction we used in the following section, and in some chapters of this thesis. We begin by defining a restricted definition of a lattice. We follow here the definitions given in [66, Chapter 16] and in [172, Chapter 5]. In the following, if a vector $\boldsymbol{u}$ has size n, we access to its coordinates are $(u_0, u_1, \ldots, u_{n-1})$ by $\boldsymbol{u}[i] = u_i$, where i is in $[0, n[$.

Definition A.1 (Lattice)**.** Let $\mathcal{B} = \{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{n-1}}\}$ be a linearly independent set of vectors in $\mathbb{Z}^m$ for some integers n and $m \geq n$. The lattice generated by $\mathcal{B}$ is the set Λ of integer linear combination of the $\boldsymbol{b_i}$. The set $\mathcal{B}$ is a basis of the lattice, its rank is n and if $m = n$, the lattice is full rank.

The basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{n-1}}\}$, whose elements are in $\mathbb{Z}^m$, of a lattice can be written as a $n \times m$ matrix M with each row i represents the vector $\boldsymbol{b_i}$. In the following, all the matrices representing a lattice will be written with this convention.

Definition A.2 (Volume of a lattice)**.** Let Λ be a lattice in $\mathbb{Z}^m$. The volume of Λ is the volume of the fundamental parallelepiped generated by of any basis of Λ. If Λ is full-rank and if M is a matrix of a basis of Λ, the volume of Λ is equal to $\det M$, denoted $\det \Lambda$.

Lemma A.1 (Basis of a lattice)**.** *Let M and M' be two $n \times m$ matrices representing the basis vector of a lattice, for some integers m and n. These two matrices represent the same lattice if and only if there exist a $n \times n$ unimodular matrix U such that $M' = UM$.*

A common way to study lattices is to deal with a short and nearly orthogonal lattice basis. An algorithm to find such a basis is the LLL algorithm, that computes an LLL reduced basis, defined in Definition A.3, that is close to reach

these two requirements. Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be two vectors of $\mathbb{Q}^m$. We denote by $\langle\cdot|\cdot\rangle$ the scalar product between two elements of $\mathbb{Q}^m$ defined by $\langle \boldsymbol{u}|\boldsymbol{v}\rangle = \boldsymbol{u}[0]\boldsymbol{v}[0] + \boldsymbol{u}[1]\boldsymbol{v}[1] + \cdots + \boldsymbol{u}[n-1]\boldsymbol{v}[n-1]$. The classical norm of $\boldsymbol{u}$ is defined as $\|\boldsymbol{u}\| = \sqrt{\langle \boldsymbol{u}|\boldsymbol{u}\rangle}$. The infinity norm of an element $\boldsymbol{u}$ is defined as the maximum of the magnitude of its coefficient, say $\|\boldsymbol{u}\|_\infty = \max_{0\leq i<n}(|\boldsymbol{u}[i]|)$.

Definition A.3 (LLL reduction)**.** Let $\mathcal{B} = \{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{n-1}}\}$ be a basis of a lattice in $\mathbb{Z}^m$. The Gram-Schmidt orthogonalization of $\mathcal{B}$ is the orthogonal family $\boldsymbol{b_0^*}, \boldsymbol{b_1^*}, \ldots, \boldsymbol{b_{n-1}^*}$ of $\mathbb{Q}^m$ defined as $\boldsymbol{b_i^*} = \boldsymbol{b_i} - \sum_{j=0}^{i-1} \mu_{i,j}\boldsymbol{b_j^*}$ where $\mu_{i,j} = \langle \boldsymbol{b_i}|\boldsymbol{b_j^*}\rangle / \|\boldsymbol{b_j^*}\|^2$. The basis $\mathcal{B}$ is LLL reduced with factor (δ, η), δ in $]1/4, 1[$ and η in $[1/2, \sqrt{\delta}[$, if:

Size reduction for $0 \leq j < i < n$, $|\mu_{i,j}| < \eta$,

Lovász condition for $0 < i < n$, $\|\boldsymbol{b_i^*}\|^2 \geq (\delta - \mu_{i,i-1}^2)\|\boldsymbol{b_{i-1}^*}\|^2$.

We can compute a LLL-reduced basis with factor $(\delta, 1/2)$ in polynomial time thanks to the LLL algorithm [128].

For a n-tuple $(w_0, w_1, \ldots, w_{n-1})$, the *weighted* scalar product is equal to $\langle \boldsymbol{u}|\boldsymbol{v}\rangle_w = w_0^2\boldsymbol{u}[0]\boldsymbol{v}[0] + w_1^2\boldsymbol{u}[1]\boldsymbol{v}[1] + \cdots + w_{n-1}^2\boldsymbol{u}[n-1]\boldsymbol{v}[n-1]$. A basis $\mathcal{B}$ is *skew-LLL reduced with factor* (δ, η) *and weight* w the LLL reduction with factor (δ, η) which uses instead of the classical scalar product the weighted scalar product.

Remark A.1. To simplify our notation, we say that a basis is LLL reduced if the basis is LLL reduced with factor $(99/100, 1/2)$. We also indistinctly denote a skew-LLL reduced basis by the term weighted reduced basis or skew reduced basis.

We now recall bounds on some specific vectors of a lattice.

Theorem A.1 (Minkowski first theorem)**.** *Let Λ be a full-rank lattice of rank n, the norm λ_1 of the shortest vector, with respect to the classical norm, is less or equal to $\sqrt{n}(|\det\Lambda|)^{1/n}$. The norm λ_1^∞ of the shortest vector, with respect to the infinity norm, is less or equal to $|\det\Lambda|^{1/n}$.*

Definition A.4 (Succesive minima)**.** Let Λ be a lattice of rank n. The successive minima of this lattice are the n real λ_i such that, for i in $[1, n]$, there exist i linearly independent vectors of the lattice whose norm is less or equal to λ_i.

Theorem A.2 (Minkowski second theorem)**.** *Let Λ a full-rank lattice of rank n, the product of the n successive minima is less or equal to $n^{n/2}|\det\Lambda|$.*

A.2 Sublattices and translate

Definition A.5 (Sublattice)**.** Let Λ_0, Λ_1 be lattices in $\mathbb{Z}^m$ of the same rank n. The lattice Λ_1 is a sublattice of Λ_0 if, for all $\boldsymbol{x}$ in Λ_1, $\boldsymbol{x}$ is also in Λ_0.

Let M_0, M_1 be $n \times m$ matrices that represent respectively a basis of the lattices Λ_0 and Λ_1. If Λ_1 is a sublattice of Λ_0 then there exists a $n \times n$ matrices B such that $M_1 = BM_0$.

Definition A.6 (Translate of a lattice)**.** Let Λ be lattices in $\mathbb{Z}^m$ of rank n generated by a basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{n-1}}\}$. A translate of Λ by a vector $\boldsymbol{x}$ in $\mathbb{Z}^m$ is the set $\boldsymbol{x}$ plus linear integer combinations of the $\boldsymbol{b_i}$.

Let M be a $n \times m$ matrices that reprensents a basis of a lattice Λ. The elements of a translate of Λ can be respresented as $(a_0, a_1, \ldots, a_{m-1})M + \boldsymbol{x}$, with $a_0, a_1, \ldots, a_{m-1}$ integers.

A.2.1 Hard problems in lattices

Definition A.7 (Shortest vector)**.** Let Λ be an lattice in $\mathbb{Z}^m$. The shortest vector is the non-zero vector $\boldsymbol{v}$ of Λ such that $\|\boldsymbol{v}\|$ is minimal.

A good approximation of the shortest vector in a lattice can be computed using LLL: with an LLL-reduced basis with factor (δ, η) of a lattice Λ of rank n, the norm of the shortest vector basis is not larger than $(1/(\delta - \eta^2))^{(n-1)/4}(\det \Lambda)^{1/n}$ following [141, Theorem 1]. The squared norm of the first non-zero basis vector given by a LLL reduction of factor $(\delta, 1/2)$ is no more than $1/(\delta - 1/4)^{n-1}$ times that of the shortest vector in the lattice [166, Theorem 1] (the case $\delta = 3/4$ has been proved in the original article about LLL [128, Proposition 1.6]).

Definition A.8 (Closest vector)**.** Let Λ be an lattice in $\mathbb{Z}^m$. Let $\boldsymbol{t}$ be a vector in $\mathbb{Q}^m$. The closest vector of $\boldsymbol{t}$ is the vector $\boldsymbol{v}$ of Λ such that $\|\boldsymbol{t} - \boldsymbol{v}\|$ is minimal.

Appendix B

A small NFS implementation

This implementation can be found at `https://github.com/lgremy/smallnfs`.

```
P.<x> = ZZ[]; l = 3141592653589793238462773; p = 2 * l + 1; d = 3;
B = [4096, 4096]; I =[2^11, 2^11]; H = [2^7, 2^7]; fbb = [B[0] // 4, B[1] // 4];
thresh = [B[0]^3, B[1]^3]

# ---------- Utilities ----------
# Pseudo ideal factorization
def pseudo_ideal_facto(a, rel):
    facto = []
    for i in rel:
        if a[1] % i[0] == 0:
            facto.append(((i[0], P(0)), i[1]))
        else:
            Q.<y> = GF(i[0])[]
            for fac in Q(a).factor():
                if fac[0].degree() == 1:
                    facto.append(((i[0], P(fac[0])), i[1]))
    return facto

# Square and multiply
def pow_mod(a, n, f):
    if n == 0:
        return  1
    elif n == 1:
        return  a
    elif n % 2 == 0:
        return pow_mod((a * a) % f,  n / 2, f)
    else:
        return (a * pow_mod((a * a) % f, (n - 1) / 2, f)) % f

# a^(lb*(p-1)/l) == b^(la*(p-1)/l)
def assert_rat_side(a, la, b, lb, p, l):
    assert(power_mod(a, lb *  (p - 1) // l, p) == power_mod(b, la *  (p - 1) //
        l, p))

# Build the matrix associated to an ideal
def ideal_matrix(ideal):
    return matrix([[ideal[0], 0], ideal[1].coefficients(sparse=False)])

# ---------- Polynomial selection ----------
# Base m with f0 of degree 1 and lc(f0) == 1
def pol_sel(m, d, p):
    f0 = x - m
    f1 = [0] * (d + 1)
    f1[d] = p // m^d
    r = f1[d] * m^d
    for i in range(d - 1, -1, -1):
        f1[i] = (p - r) // m^i
        r = r + f1[i] * m^i
    f1 = P(f1)
    return (f0, P(f1))

def build_ideal(poly, q, lc, lcp):
    ideal = []
    avoid = []
    if poly.discriminant() % q != 0:
        if lc % q == 0 and lcp % q != 0:
            ideal.append((q, P(0))) # q is purely projective
        if lc % q == 0 and lcp % q == 0:
            avoid.append(q)
            return (ideal, avoid)
        Q.<y> = GF(q)[]
        for fac in Q(poly).factor():
```

```
            if fac[0].degree() == 1 and fac[1] == 1:
                ideal.append((q, P(fac[0])))
    else:
        avoid.append(q)
    return (ideal, avoid)

# Build factor basis
def build_fb(f, B):
    avoid = []; F = []
    for i in range(len(f)):
        avoidtmp = []
        Ftmp = []
        poly = f[i]
        lc = poly.leading_coefficient()
        lcp = poly.coefficients(sparse=False)[poly.degree() - 1]
        for q in primes(B[i]):
            (ideals, avoids) = build_ideal(poly, q, lc, lcp)
            for k in ideals:
                Ftmp.append(k)
            for k in avoids:
                avoidtmp.append(k)
        avoid.append(avoidtmp)
        F.append(Ftmp)
    return (F, avoid)

# ---------- Relation collection ----------
# Return True if F is B-smooth and do not have factor in avoid
def is_smooth_and_avoid(N, B, avoid):
    if N == 0:
        return (False, 0)
    if N == 1:
        return (False, 0)
    fac = N.factor()
    if fac[len(fac) - 1][0] < B:
        for f in fac:
            if f[0] in avoid:
                return (False, 0)
        return (True, fac)
    return (False, 0)

# Compute norm
def norm(a, f):
    return abs(a.resultant(f))

# Line sieve
def line_sieve(p, r, H, L, side):
    x0 = 0
    for e1 in range(0, H[1]):
        e0 = x0
        while e0 < H[0]:
            if e0 >= -H[0]:
                L[e0 + H[0]][e1][side] = L[e0 + H[0]][e1][side] / p
            e0 = e0 + p
        e0 = x0 - p
        while e0 >= -H[0]:
            if e0 < H[0]:
                L[e0 + H[0]][e1][side] = L[e0 + H[0]][e1][side] / p
            e0 = e0 - p
        x0 = x0 + r
        if x0 >= H[0]:
            x0 = x0 - p


# Arrange matrix special-q
def arrange_matrix_spq(M):
    coeff = []
    if M[0][1] < 0:
        coeff.append([-1, 0])
    else:
        coeff.append([1, 0])
    if M[1][1] < 0:
        coeff.append([0, -1])
    else:
        coeff.append([0, 1])
    M = matrix(coeff) * M
    if M[0][1] > M[1][1]:
        M.swap_rows(0, 1)
    return M

# Verify smoothness of a0 + a1 * x
def good_rel_spq(a0, a1, f, q, qside, avoid):
    n = norm(a0 + a1 * x, f[0])
    if qside == 0:
        n = n / q
    fac0 = is_smooth_and_avoid(n, B[0], avoid[0])
    n = norm(a0 + a1 * x, f[1])
    if qside == 1:
        n = n / q
    fac1 = is_smooth_and_avoid(n, B[1], avoid[1])
    return fac0[0] and fac1[0]

# Root in q-lattice
def root_qlattice(M, i):
    inv =  M[0][0] - M[0][1] * i[1][0]
    if inv % i[0] == 0:
        return None
    return ((i[1][0] * M[1][1] - M[1][0]) * (M[0][0] - M[0][1] *
        i[1][0]).inverse_mod(i[0])) % i[0]
```

```
# Special-q + line sieve
def spq_sieve(ideal, qside, f, B, H, F, avoid, fbb, thresh, nb_rel):
    Q.<y> = GF(ideal[0])[]
    assert(Q(ideal[1]) in [fac[0] for fac in Q(f[qside]).factor()])
    M = ideal_matrix(ideal).LLL()
    M = arrange_matrix_spq(M)
    R = []
    L = [[[norm(P(list(vector((i0, i1)) * M)), f[0]), norm(P(list(vector((i0,
        i1)) * M)), f[1])] for i1 in range(0, H[1])] for i0 in range(-H[0],
            H[0])]
    for i in F[0]:
        if i[0] > fbb[0]:
            break
        r = root_qlattice(M, i)
        if r != None:
            line_sieve(i[0], r, H, L, 0)
    for i in F[1]:
        if i[0] > fbb[1]:
            break
        if (i[1].degree() == 1):
            r = root_qlattice(M, i)
            if r != None:
                line_sieve(i[0], r, H, L, 1)
    for i0 in range(-H[0], H[0]):
        for i1 in range(0, H[1]):
            if (L[i0 + H[0]][i1][0] < thresh[0] and L[i0 + H[0]][i1][1] <
                    thresh[1]):
                [a0, a1] = list(vector((i0, i1)) * M)
                if gcd(a0, a1) == 1 and a1 >= 0:
                    if good_rel_spq(a0, a1, f, ideal[0], qside, avoid):
                        R.append((a0 + a1 * x, norm(a0 + a1 * x, f[0]).factor(),
                                norm(a0 + a1 * x, f[1]).factor()))
                    if len(R) == nb_rel:
                        return R
    return R

# Remove duplicate relations
def dup(L):
    D = {}
    for i in L:
        D[i[0]] = (i[1], i[2])
    L = []
    for i in D:
        L.append((i, D[i][0], D[i][1]))
    return L


# High level function to perform relation collection
def find_rel_spq_sieve(f, B, H, F, avoid, fbb, thresh):
    R = []
    for i in F[1]:
        if i[0] > fbb[1] and i[1].degree() == 1:
            R = R + spq_sieve(i, 1, f, B, H, F, avoid, fbb, thresh, -1)
    return dup(R)

# ---------- Linear algebra ----------
# Number of SMs
def nb_SM(f):
    return len(f.real_roots()) + (len(f.complex_roots()) -
            len(f.real_roots())) / 2 - 1

# Compute SM exponent
def sm_exp(f, l):
    Q.<y> = GF(l)[]
    return lcm([l^i[0].degree() - 1 for i in Q(f).factor()])

# Compute SM
def compute_SM(a, sm_l_exp, nb_sm_l, l, f):
    Q.<y> = IntegerModRing(l^2)[]
    aq = Q(a); fq = Q(f); L = []
    tmp = list(P(pow_mod(aq, sm_l_exp, fq)) - 1)
    i = len(tmp) - 1
    while len(L) < nb_sm_l:
        L.append(Integer(tmp[i] / l))
        i = i - 1
    return L

# Row transformation
def row_trans(r, F, column_l, nb_sm_l, sm_l_exp, l, fl):
    L = [0 for i in range(len(F[0]) + len(F[1]) + column_l)]
    for ideal in pseudo_ideal_facto(r[0], r[1]):
        L[F[0].index(ideal[0])] = ideal[1]
    for ideal in pseudo_ideal_facto(r[0], r[2]):
        L[len(F[0]) + F[1].index(ideal[0])] = -ideal[1]
    if column_l == 1:
        L[len(F[0]) + len(F[1])] = 1
    if nb_sm_l != 0:
        L = L + compute_SM(r[0], sm_l_exp, nb_sm_l, l, fl)
    return L

# Build the matrix of relations
def build_mat(R, F, fl, l, column_l, nb_sm_l, sm_l_exp):
    M = []
    for r in R:
        L = row_trans(r, F, column_l, nb_sm_l, sm_l_exp, l, fl)
        M.append(L)
    return matrix(GF(l), M)

# Virtual logarithms
```

```
# Return all the indices such that L[i] == k
def index(L, k):
    return [i for i in range(len(L)) if L[i] == k]

# Find not known virtual log
def not_known(K):
    nk = []
    for i in range(1, K.dimensions()[0]):
        L = list(set(list(K[i])))
        for j in L:
            if j != 0:
                nk = nk + index(list(K[i]), j)
    return nk

# Associate virtual logarithm to ideal
def associate(F, K, nk, column_1, nb_sm_1):
    V = [{}, {}]; col1 = -1; SM1 = [-1 for i in range(nb_sm_1)]
    for i in range(len(F[0])):
        if i not in nk:
            V[0][F[0][i]] = K[i]
    for i in range(len(F[1])):
        if i + len(F[0]) not in nk:
            V[1][F[1][i]] = K[len(F[0]) + i]
    if column_1 == 1:
        col1 = K[len(F[0]) + len(F[1])]
    for i in range(nb_sm_1):
        SM1[i] = K[i + len(F[0]) + len(F[1]) + column_1]
    return (V, col1, SM1)

# ---------- Individual logarithm ----------
# Compute individul logarithm of an ideal above next_prime(B[0]) in K0 (always exists)
def ind_log_0(f, B, H, F, avoid, V, col1, fbb, thresh, SM1, sm_1_exp, l):
    q = next_prime(B[0])
    # Assume we can have a relation with the previous setting
    spq = build_ideal(f[0], q, f[0].leading_coefficient(),
            f[0].coefficients(sparse=False)[f[0].degree() - 1])[0][0]
    # Take the first relation
    r = spq_sieve(spq, 0, f, B, H, F, avoid, fbb, thresh, -1)[0]
    pseudo_ideal_facto_0 = pseudo_ideal_facto(r[0], r[1])
    coeff_spq = Integer(pseudo_ideal_facto_0[[i[0] for i in
        pseudo_ideal_facto_0].index(spq)][1])
    vlog = (-(sum([V[0][i[0]] * i[1] for i in pseudo_ideal_facto_0 if i[0] in
        V[0].keys()])) + (sum([V[1][i[0]] * i[1] for i in
            pseudo_ideal_facto(r[0], r[2])])) - col1)
    if len(SM1) != 0:
        sm = compute_SM(r[0], sm_1_exp, len(SM1), 1, f[1])
        for i in range(len(SM1)):
            vlog = vlog - sm[i] * SM1[i]
    vlog = vlog % l
    vlog = (vlog * coeff_spq.inverse_mod(l)) % l
    V[0][spq] = vlog

# ---------- Main ----------
def main(d, p, B, H, l, fbb, thresh):
    print("Polynomial selection")
    m = floor(p^(1/(d + 1)))
    f = pol_sel(m, d, p)

    # Build factor basis
    (F, avoid) = build_fb(f, B)

    print("Relation collection")
    R = find_rel_spq_sieve(f, B, H, F, avoid, fbb, thresh)
    assert(len(R) >= len(F[0]) + len(F[1]))

    print("Linear algebra")
    sm_1_exp = sm_exp(f[1], l)
    nb_sm_1 = nb_SM(f[1])
    column_1 = 0
    # Add a column of 1 that resprents the ideals that divide the leading
    # coefficient
    if f[1].leading_coefficient() != 1:
        column_1 = 1
    M = build_mat(R, F, f[1], l, column_1, nb_sm_1, sm_1_exp)
    K = M.right_kernel().basis_matrix()

    # Virtual logarithm
    nk = not_known(K)
    K = K[0]
    (V, col1, SM1) = associate(F, K, nk, column_1, nb_sm_1)


    print("Individual logarithm")
    ind_log_0(f, B, H, F, avoid, V, col1, fbb, thresh, SM1, sm_1_exp, l)
    # Assert on rational side
    for i in range(len(V[0].keys())):
        for j in range(i + 1, len(V[0].keys())):
            assert_rat_side(Integer(V[0].keys()[i][0]),
                    Integer(V[0][V[0].keys()[i]]), Integer(V[0].keys()[j][0]),
                    Integer(V[0][V[0].keys()[j]]), p, l)

    return (V, col1, SM1)
```

Appendix C

Polynomial selection for the NFS

In this chapter, we try to investigate how the α quantity is computed, according to Definition 3.1. We just recall that, given a irreducible polynomial f over $\mathbb{Z}$ and an integer $t > 1$, $\alpha(f) = \sum_{\ell \text{ prime}} \alpha_l(f)$, with for all prime ℓ,

$$\alpha_\ell(f) = \ln(\ell)\big[\,\mathbb{A}(\mathrm{val}_\ell(n), n \in \mathbb{Z}) \\ - \mathbb{A}(\mathrm{val}_\ell(\mathrm{Res}_x(f(x), a(x)), \text{ where } a \in \mathbb{Z}[x], \deg a = t-1, a \text{ irreducible}))\big],$$

where $\mathbb{A}(\cdot)$ is the average value and val_ℓ the ℓ-adic valuation.

C.1 First term

Let us investigate the first term of $\alpha_\ell(f)$ which is $\mathbb{A}(\mathrm{val}_\ell(n), n \in \mathbb{Z})$. This term is equal to $1/(\ell-1)$ and the sketch of the proof is the following. A random number n is divisible by ℓ with probability $1/\ell$, is divisible by ℓ^2 with probability $1/\ell^2$ and so on. The average valuation of ℓ is therefore $\sum_{i=1}^{\infty} 1/\ell^i = 1/(1-1/\ell) - 1 = 1/(\ell-1)$.

C.2 Second term

C.2.1 Two dimensional case

Simplified case

Let us now investigate the second term, firstly on a monic polynomial of degree one, say $f(x) = x - m$. We know that $\mathrm{Res}_x(f(x), a(x)) = a_0 + m a_1$. In this case, the condition to have a irreducible is translated into $\gcd(a_0, a_1) = 1$, which can be written as ℓ does not divide $\gcd(a_0, a_1)$.

If a_0 and a_1 are in $[0, \ell[$, there exists $\ell^2 - 1$ pairs (a_0, a_1) that fit in the condition on the polynomial a. For any a_1, if $a_0 \equiv -a_1 m \bmod \ell$, then ℓ divides the resultant between f and a, otherwise it does not. If $a_1 = 0$, the previous modular equation does not have a solution. If $a_1 \neq 0$, there exists only one solution for the modular equation. Then, there exist $\ell - 1$ polynomial $a_0 + a_1 x$

such that their resultant with f is divisible by ℓ: the average value is therefore equal to $(\ell-1)/(\ell^2-1)$.

If a_0 and a_1 are in $[0,\ell^2[$, there exists $\ell^4-\ell^2$ pairs (a_0,a_1) that fit in the condition on the polynomial a. The divisibility of the resultant between a and f by ℓ^2 occurs when $a_0 \equiv -ma_1 \bmod \ell^2$. If ℓ does not divides a_1, there exists one value for a_0 such that $a_0 \equiv -ma_1 \bmod \ell^2$. If ℓ divides a_1, then ℓ does not divide a_0 and $a \not\equiv -a_1 m \bmod \ell$ and there is no valuation. The number of valid polynomial a such that their resultant with f is divisible by ℓ^2 is therefore $\ell^2-\ell$: the average value is then $(\ell^2-\ell)/(\ell^4-\ell^2) = (\ell-1)/(\ell(\ell^2-1))$.

By continuing to consider that a_0 and a_1 are in $[0,\ell^k[$, we can show that the average value for this set is $(\ell-1)/(\ell^k(\ell^2-1))$. To have the complete value of the second term, we need to compute the sum of all the average value of the different sets $[0,\ell^k[^2$, that is $\sum_{k=1}^{\infty}(\ell-1)/(\ell^k(\ell^2-1)) = \ell/(\ell^2-1)$.

General case

If ℓ does not divide the discriminant of f nor its leading coefficient, the previous result can be extended to any non linear polynomial, by lifting all the unique roots thanks the Hensel's lemma: the second term is equal to $\ell/(\ell^2-1)$. If the roots are multiple, the formula change a bit, see for example [11, Section 3.2.3]: an α function that covers all the case is available in CADO-NFS.

Another strategy is to evaluate this term thanks to a Monte-Carlo approach: this is the choice of the implementation available in Magma.

C.2.2 Three dimensional case

We now reproduce the proof of formula to compute $\alpha_l(f)$ in some situation.

General case

Proposition C.1 ([72, Proposition 2])**.** *Let f be an irreducible polynomial over $\mathbb{Z}$ and ℓ be a prime not dividing the leading coefficient of f or its discriminant. Then, in the case of sieving in dimension $t = 3$,*

$$\alpha_l(f) = \frac{\ln(\ell)}{\ell-1}\left(1 - n_1\frac{\ell(\ell+1)}{\ell^2+\ell+1} - 2n_2\frac{\ell^2}{(\ell+1)(\ell^2+\ell+1)}\right),$$

where n_1 and n_2 are the number of linear (respectively, degree-2) irreducible factors of f modulo ℓ.

Proof. The condition on the leading coefficient allows us to avoid questions about projective roots, and the condition on the discriminant implies that any irreducible factor of f modulo ℓ can be lifted to an irreducible factor of f of the same degree over the ℓ-adic ring $\mathbb{Z}_\ell$.

Let φ be a quadratic irreducible factor of f over $\mathbb{Z}_\ell$. Let a be a quadratic polynomial with coefficients in $\mathbb{Z}$ whose content is not divisible by ℓ. Then the ℓ-adic valuation of the resultant of φ and a is $2k$, where k is the largest integer such that a is proportional to φ modulo ℓ^k. The number of a with coefficients in $[0,\ell^k-1]$ that satisfy this condition is $\ell^k-\ell^{k-1}$ since they are the polynomials of the form $\gamma\varphi$, where γ is not divisible by ℓ. Furthermore, the number of polynomials a with coefficients in $[0,\ell^k-1]$ whose content is not divisible by ℓ is

$\ell^{3k} - \ell^{3k-3}$. Hence the proportion of those polynomials for which the valuation of its resultant with φ is at least $2k$ is $(\ell^k - \ell^{k-1})/(\ell^{3k} - \ell^{3k-3})$. Finally, the contribution due to φ in the expected valuation of the resultant of f and a is $\sum_{k\geq 1} 2(\ell^k - \ell^{k-1})/(\ell^{3k} - \ell^{3k-3}) = 2\ell^2/((\ell^2-1)(\ell^2+\ell+1))$.

The case of the contributions of roots of f is handled similarly: the number of polynomials a with coefficients in $[0, \ell^k - 1]$ whose content is not divisible by ℓ and that give a value divisible by ℓ^k when evaluated at an ℓ-adic root ρ of f is $\ell^{2k} - \ell^{2k-2}$, since they are all of the form $(x - \rho)(\alpha x - \beta)$, with α and β in $[0, \ell^k - 1]$ and not simultaneously divisible by ℓ. Therefore the contribution of a root ρ in the expected valuation of the resultant of f and a is $\sum_{k\geq 1}(\ell^{2k} - \ell^{2k-2})/(\ell^{3k} - \ell^{3k-3}) = (\ell^2 + \ell)/(\ell^3 - 1)$.

□

Workaround in particular cases

When the proposition does not apply, the natural workaround is to compute the factorization of f over the ℓ-adic field (see for instance [46, Chapter 6.1]) and for each factor do the same kind of study as in the proof of the proposition. One could argue that, since computing the ℓ-maximal order is required for converting relations to rows of the matrix, this is appropriate. However, computing α must be as fast as possible because we might want to investigate billions of polynomials. In the classical two-dimensional case, a very simple lifting is enough to deduce the average ℓ-adic valuation. In the following section, we sketch a similar approach that, in many cases, will give the average valuation without having to perform a full ℓ-adic factorization.

Finally, the case where ℓ divides the leading coefficient of f is dealt with by adding the contribution of the (possibly multiple) root 0 in the reverted polynomial $f(1/x)x^{\deg f}$.

Computing the average ℓ-adic valuation

In the case of two-dimensional sieving, the average ℓ-adic valuation can be computed with the small recursive lifting function given as Algorithm C.1. This is what is done, for instance, in the CADO-NFS implementation. This is admittedly much simpler and faster that running a full factorization of f over the ℓ-adics, which is advantageous when many polynomials have to be tested.

The equivalent for three-dimensional sieving is not as simple, but can still be faster in many cases than a full ℓ-adic factorization. Let us give first the modifications to be made for computing the contribution of irreducible factors of degree 2 modulo ℓ. The normalization factor C must be changed to $\ell^2/(\ell^2+\ell+1)$. Then all the computations must no longer be done over the integers, but over the unramified extension $\mathbb{Q}_{\ell^2}$ of $\mathbb{Q}_\ell$ of degree 2. Only the roots genuinely over this extension are considered in the `for` loop, but their computation is still done modulo ℓ: in step 4, we are now doing polynomial factorization over $\mathbb{F}_{\ell^2}$. Finally, in step 6, the contribution to add is $1/(\ell^2-1)$, and in step 8, the result of the recursive call must be multiplied by C/ℓ^2 instead of C/ℓ. We obtain Algorithm C.2.

We emphasize that this algorithm returns the correct answer, even in the case where the degree-2 factor is a multiple factor of f modulo ℓ; knowing the nature of the ℓ-adic factorization above this multiple factor is not required.

The case of a multiple factor of degree 1 is less straightforward, because the contribution to the average valuation will not be the same for a split, inert or ramified factor above this root. Still, in most cases, and in particular when the root is only a double root, it is possible to adapt Algorithm C.1 and get the correct answer. We skip the details; the corresponding code is given in the `nfs-hd` directory of the CADO-NFS repository.

Algorithm C.1: `av_val_dim2`
Average ℓ-adic valuation, two-dimensional case.

input : a polynomial f, a prime ℓ
output: the average ℓ-adic valuation of $\mathrm{Res}_x(f(x), a - bx)$

$v \leftarrow \ell$-valuation of the content of f;
$f \leftarrow f/\ell^v$;
$C \leftarrow \ell/(\ell+1)$; $v \leftarrow Cv$;
for *r in Roots(f) modulo* ℓ **do**
  if *r is a simple root* **then**
    $v \leftarrow v + C/(\ell - 1)$;
  else
    $v \leftarrow v + C/\ell \times$ `av_val_dim2`$(f(r + \ell x), \ell)$;
  end
end
return v;

Algorithm C.2: `av_val_dim3_deg2`
Average ℓ-adic valuation, three-dimensional case; contribution of irreducible factors of degree 2.

input : a polynomial f over $\mathbb{Q}_{\ell^2}$, a prime ℓ
output: the contribution to the average ℓ-adic valuation of $\mathrm{Res}_x(f(x), ax^2 + bx + c)$, coming from irreducible factors of degree 2 mod ℓ

$v \leftarrow$ minimum ℓ-valuation of the coefficients of f;
$f \leftarrow f/\ell^v$;
$C \leftarrow \ell^2/(\ell^2 + \ell + 1)$; $v \leftarrow Cv$;
for *r in Roots(f) modulo* ℓ **do**
  at top-level of recursion, if r is in $\mathbb{Q}_\ell$, skip it;
  if *r is a simple root* **then**
    $v \leftarrow v + C/(\ell^2 - 1)$;
  else
    $v \leftarrow v + C/\ell^2 \times$ `av_val_dim3_deg2`$(f(r + \ell x), \ell)$;
  end
end
return v;

Appendix D

Complexity analysis of Zajac's MNFS

In this appendix, we analyze the MNFS variant described by Zajac in [185, Section 6.4] to compute discrete logarithms in fields $\mathbb{F}_{p^n} = \mathbb{F}_Q$ of medium characteristic. We recall that the number of number fields is denoted by V, the polynomials used in this variant are defined by:

- the polynomial f_0 is an irreducible polynomial with small coefficients, say in $O(1)$, of degree n,
- the polynomials f_i, where i is in $[1, V[$, are defined by $f_i = f_0 + ph_i$, where the h_i are polynomials of degree less than n such that the f_i are irreducible.

We essentially follow the analyses given by Barbulescu–Pierrot in [24] and Pierrot in [143]. In medium characteristic, we have $p = L_Q(l_p, c_p)$, where l_p is in $]1/3, 2/3[$, then, the extension degree n is equal to $1/c_p(\log Q/\log\log Q)^{1-l_p})$. Let B_i the smoothness bound on side i, E be the bound on the coefficients of the polynomial a of degree $t-1$ mapped in the number fields. We assume that we can express the parameters $V = L_Q(1/3, c_v)$, $B_0 = L_Q(1/3, c_b)$, $B_{i\neq 0} = L_Q(1/3, c'_b)$, $E = L_Q(l_p - 1/3, c_e c_p)$ and $t = c_t/c_p(\log Q/\log\log Q)^{2/3-l_p}$.

Let first consider the high-level point of view of the complexity of an index calculus, assuming that the individual logarithm step is negligible. The number of ideal involved in a relation on side i is equal to $\pi(B_i) \leq B_i$, where π is the prime counting function. The cost of the linear algebra is asymptotically equal to $(B_0 + (V-1)B_i)^2 \leq (B_0 + VB_i)^2$ using the Wiedemann algorithm and the cost of the relation collection is bounded, using the ECM algorithm to perform the cofactorization step, by $E^t(L_{B_0}(1/2, \sqrt{2}) + (V-1)L_{B_i}(1/2, \sqrt{2})) = O(E^t)$ following Proposition 1.1. In a first approximation which can be suboptimal, see for example the analysis in Section 1.3.2, these two costs are balanced, that is $E^t = (B_0 + VB_i)^2$. We require in addition that $VB_i = B_0$: it seems anew suboptimal to have the bounds B_i smaller than B_0 while the norms on the side 0 are lower than the norms on other sides (but it allows us to perform a simple complexity analysis), we therefore get that $B_i = L_Q(1/3, c_b - c_v)$ and asymptotically, we have

$$E^t = B_0^2, \tag{D.1}$$

which can be translated by $c_e c_t = 2c_b$.

In addition, during the relation collection, we must produce as many relations as the number of unknown ideals, that is around $2B_0 = O(B_0)$ relations. Let P be the probability of getting a relation, that is having a smooth norm on side 0 and one in another side i where i is in $[1, V[$. We must have $E^t P = B_0$, that is, using Equation (D.1),

$$B = 1/P. \tag{D.2}$$

With these parameters, we can give an upper bound on the norms in each number field. On side 0, the upper bound N_0 is equal to $(\deg a + \deg f_0)! \|f_0\|_\infty^{\deg a} \|a\|_\infty^{\deg f_0} \approx \|f_0\|_\infty^t \|a\|_\infty^n = E^n$. On the other side, the upper bound N_i is equal to $E^n p^t$. Expressed with the L function, we have $N_0 = L_Q(2/3, c_e)$ and $N_i = L_Q(2/3, c_e + c_t)$. At this step, we can try to minimize the product of the norms, but this strategy seems suboptimal and we continue our computation. Using the bounds on the norms, we can estimate the probability P of getting a relation: this probability is equal to the product of the probability that the norm of a polynomial a on side 0 is B_0-smooth and the probability that the norm of a in at least one other side is B_i-smooth. Using Corollary 1.1, the probability of smoothness on side 0 is equal to $L_Q(1/3, -1/3 \cdot c_e/c_t)$. The probability of smoothness in at least one other side is equal to $1 - (1 - P_i)^V \approx VP_i$, where P_i is the probability of a norm to be B_i-smooth, that is $VP_i = L_Q(1/3, c_v - 1/3 \cdot (c_e + c_t)/(c_b - c_v))$. Putting together, the probability P is equal to $L_Q(1/3, -1/3 \cdot c_e/c_t + c_v - 1/3 \cdot (c_e + c_t)/(c_b - c_v)$. Using Equation (D.1) and Equation (D.2), we get $c_b = 2/(3c_t) - c_v + (2c_b + c_t^2)/(3c_t(c_b - c_v))$. By expanding this equation, we get

$$g(c_t, c_b, c_v) = 3c_t c_b^2 - 4c_b + 2c_v - 3c_t c_v^2 - c_t^2 = 0. \tag{D.3}$$

The cost of the MNFS algorithm is equal to $L_Q(1/3, 2c_b)$, that is the cost of the linear algebra since it is equal to the cost of the relation collection. We want to minimize this cost, that is minimize $f(c_t, c_b, c_v) = 2c_b$, under the constraint in Equation (D.3). Using the method of Lagrange multipliers, we get the following system, where λ is non-zero real number.

$$\begin{cases} \dfrac{\partial f}{\partial c_v} + \lambda \dfrac{\partial g}{\partial c_v} = \lambda(2 - 6c_t c_v) = 0 \\ \dfrac{\partial f}{\partial c_b} + \lambda \dfrac{\partial g}{\partial c_b} = 2 + \lambda(6c_t c_b - 4) = 0 \\ \dfrac{\partial f}{\partial c_t} + \lambda \dfrac{\partial g}{\partial c_t} = \lambda(3c_b^2 - 3c_v^2 - 2c_t) = 0 \end{cases}.$$

From the first and third rows of this system, we deduce that $c_t = 2/(6c_v)$ and $c_b = \sqrt{c_v^2 + 4/(18c_v)}$. Putting the value of these two variables, in Equation (D.3), we get $-972c_v^6 - 252c_v^3 + 1 = 0$. We deduce that $c_v = ((2\sqrt{13} - 7)/54)^{1/3}$, then $c_b = ((46 + 13\sqrt{13})/54)^{1/3}$. The complexity of this MNFS variant is therefore in

$$L_Q\left(1/3, \left(\frac{4(46 + 13\sqrt{13})}{27}\right)^{1/3} \approx 2.40\right).$$

This is exactly the complexity announced in [24], recalled in Section 4.5.2, using the JLSV_1 polynomial selection instead of the JLSV_0 one.

Appendix E

Sieving algorithms

E.1 Two-dimensional sieve algorithms

In this section, we give the algorithms of the line sieve and the lattice sieve. Let Λ be a lattice whose a basis follows the one given in Chapter 6. Let $\mathcal{H}$ be a sieving region of shape $[H_0^m, H_0^M[\times[H_1^m, H_1^M[$.

E.1.1 Line sieve

We give, in Algorithm E.1, the algorithm of the line sieve in the case of the volume of Λ is less or larger than $H_0^M - H_0^m$. When the volume of the lattice Λ is larger than $H_0^M - H_0^m$, the only difference with Algorithm E.1 is the append of $\boldsymbol{c}$ to the list L: we need to verify before this append if $\boldsymbol{c}$ is in $\mathcal{H}$. This is because the transition-vectors in this case are not the same as the case when the volume of Λ is less than $H_0^M - H_0^m$.

E.1.2 Lattice sieve

Let the volume of Λ be larger than $H_0^M - H_0^m$. In this case, we apply the sieve of Franke–Kleinjung, described in Algorithm E.2. Function `reduce-qlattice` is described in Section 6.2.2.

E.2 General algorithm

In this section, we will give the algorithm of the functions we have not describe in Section 6.4. We recall the general structure of the sieve algorithm:

Initialization.

1. Given $\mathcal{H}$ and Λ, the procedure `findV` returns nearly-transition-vectors and skew-small-vectors.
2. Set $\boldsymbol{c}$ to $\mathbf{0}$ and k to $t-1$

Enumeration.

3. While $\boldsymbol{c}[k] < H_k^M$:

Algorithm E.1: A line sieve algorithm when the volume of Λ is less than $H_0^M - H_0^m$.

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}\}$ of Λ, the sieving region $\mathcal{H}$
output: list of elements in $\Lambda \cap \mathcal{H}$
$\boldsymbol{c} \leftarrow \boldsymbol{0}$; $L \leftarrow \emptyset$;
while $\boldsymbol{c}[1] < H_1^M$ **do**
  $\boldsymbol{c_t} \leftarrow \boldsymbol{c}$;
  while $\boldsymbol{c}[0] < H_0^M$ **do** $L \leftarrow L \cup \boldsymbol{c}$; $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{b_0}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{c_t} - \boldsymbol{b_0}$; `// Avoid to append two times` $\boldsymbol{c_t}$ `to` L
  while $\boldsymbol{c}[0] \geq H_0^m$ **do** $L \leftarrow L \cup \boldsymbol{c}$; $\boldsymbol{c} \leftarrow \boldsymbol{c} - \boldsymbol{b_0}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{b_0}$; `//` $\boldsymbol{c}$ `is again in the sieving region`
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{b_1}$;
end
$\boldsymbol{c} \leftarrow -\boldsymbol{b_1}$;
while $\boldsymbol{c}[1] \leq H_1^m$ **do**
  $\boldsymbol{c_t} \leftarrow \boldsymbol{c}$;
  while $\boldsymbol{c}[0] < H_0^M$ **do** $L \leftarrow L \cup \boldsymbol{c}$; $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{b_0}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{c_t} - \boldsymbol{b_0}$; `// Avoid to append two times` $\boldsymbol{c_t}$ `to` L
  while $\boldsymbol{c}[0] \geq H_0^m$ **do** $L \leftarrow L \cup \boldsymbol{c}$; $\boldsymbol{c} \leftarrow \boldsymbol{c} - \boldsymbol{b_0}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{b_0}$; `//` $\boldsymbol{c}$ `is again in the sieving region`
  $\boldsymbol{c} \leftarrow \boldsymbol{c} - (\boldsymbol{b_1} - \boldsymbol{b_0})$;
end
return L;

   (a) Report $\boldsymbol{c}$.
   (b) If $k > 0$, call recursively this enumeration procedure (`sieve`) with input $\boldsymbol{c}$ and $k - 1$.
   (c) Add $\boldsymbol{v}$ to $\boldsymbol{c}$, where $\boldsymbol{v}$ is a k–nearly-transition-vector, such that $\boldsymbol{c}$ lands in $\mathcal{H}_{k-1}$ (`add`)
   (d) If not possible, the fall-back strategy (`fbAdd`) try to produce a new element in $\mathcal{H}$, and then a new k–nearly-transition-vector, by using k–skew-small-vectors.
4. Recover $\boldsymbol{c}$ as it was when the procedure was called.
5. While $\boldsymbol{c}[k] \geq H_k^m$:
   (a) Perform Item 3a, Item 3b, Item 3c and Item 3d by considering $\boldsymbol{c} - \boldsymbol{v}$ instead of $\boldsymbol{c} + \boldsymbol{v}$.

With the functions described in Section 6.4 and the functions below, we instantiate this general description in Algorithm E.3 and Algorithm E.4, that is the first and second sieve algorithms we describe in Section 6.4.

Algorithm E.2: Lattice sieve.

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}\}$ of Λ, the sieving region $\mathcal{H}$
output: list of elements in $\Lambda \cap \mathcal{H}$
$(\boldsymbol{u}, \boldsymbol{v}) \leftarrow$ `reduce-qlattice`$(\boldsymbol{b_0}, \boldsymbol{b_1})$;
$\boldsymbol{c} \leftarrow \boldsymbol{0}$; $L \leftarrow \emptyset$;
`// increasing` $\boldsymbol{c}[1]$
while $\boldsymbol{c}[1] < H_1^M$ **do**
  $L \leftarrow L \cup \{\boldsymbol{c}\}$;
  if $H_0^m - \boldsymbol{u}[0] \leq \boldsymbol{c}[0] < H_0^M$ **then**
    $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{u}$;
  else if $H_0^m \leq \boldsymbol{c}[0] < H_0^M - \boldsymbol{v}[0]$ **then**
    $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{v}$;
  else
    $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{u} + \boldsymbol{v}$;
  end
end
`// avoid to report` **0** `two times`
if $H_0^m \leq 0 < H_0^M + \boldsymbol{u}[0]$ **then**
  $\boldsymbol{c} \leftarrow -\boldsymbol{u}$;
else if $H_0^m + \boldsymbol{v}[0] \leq 0 < H_0^M$ **then**
  $\boldsymbol{c} \leftarrow -\boldsymbol{v}$;
else
  $\boldsymbol{c} \leftarrow -(\boldsymbol{u} + \boldsymbol{v})$;
end
`// decreasing` $\boldsymbol{c}[1]$
while $\boldsymbol{c}[1] \leq H_1^m$ **do**
  $L \leftarrow L \cup \{\boldsymbol{c}\}$;
  if $H_0^m \leq \boldsymbol{c}[0] < H_0^M + \boldsymbol{u}[0]$ **then**
    $\boldsymbol{c} \leftarrow \boldsymbol{c} - \boldsymbol{u}$;
  else if $H_0^m + \boldsymbol{v}[0] \leq \boldsymbol{c}[0] < H_0^M$ **then**
    $\boldsymbol{c} \leftarrow \boldsymbol{c} - \boldsymbol{v}$;
  else
    $\boldsymbol{c} \leftarrow \boldsymbol{c} - (\boldsymbol{u} + \boldsymbol{v})$;
  end
end
return L;

Function `fbSub1`(k, $\boldsymbol{c}$, $\mathcal{H}$, S, Λ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ

output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

while $\boldsymbol{c}[k] \geq H_k^m$ **do**
  $L \leftarrow \emptyset$;
  for $\boldsymbol{v} \in S[k]$ **do**
    $D \leftarrow$ `CVA`$(\boldsymbol{c} - \boldsymbol{v}, \Lambda, k-1)$;
    for $\boldsymbol{d} \in D$ **do**
      if $\boldsymbol{c} - \boldsymbol{v} - \boldsymbol{d} \in \mathcal{H}$ **then return** $\boldsymbol{c} - \boldsymbol{v} - \boldsymbol{d}$;
      $L \leftarrow L \cup \{\boldsymbol{c} - \boldsymbol{v} - \boldsymbol{d}\}$;
    end
  end
  set $\boldsymbol{c}$ to an element of L;
end
return $\boldsymbol{c}$; `// c is out of` $\mathcal{H}$

Function `sub1`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ

output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

for $\boldsymbol{v} \in T[k]$ **do**
  if $\boldsymbol{c} - \boldsymbol{v} \in \mathcal{H}_{k-1}$ **then return** $\boldsymbol{c} - \boldsymbol{v}$;
end
if $k > 0$ **then**
  $\boldsymbol{d} \leftarrow$ `fbSub1`$(k, \boldsymbol{c}, \mathcal{H}, S, \Lambda)$;
  if $\boldsymbol{d} \in \mathcal{H}$ **then** $T[k] \leftarrow T[k] \cup \{\boldsymbol{c} - \boldsymbol{d}\}$; $S[k] \leftarrow S[k] \cup \{\boldsymbol{c} - \boldsymbol{d}\}$;
  $\boldsymbol{c} \leftarrow \boldsymbol{d}$;
else
  $\boldsymbol{c} \leftarrow (H_0^m - 1, H_1^m - 1, \ldots, H_{t-1}^m - 1)$; `// c is out of` $\mathcal{H}$
end
return $\boldsymbol{c}$;

Function `sieve1`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ, L)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the list L that contains the elements in $\mathcal{H} \cap \Lambda$

$\boldsymbol{c_t} \leftarrow \boldsymbol{c}$;
while $\boldsymbol{c_t}[k] < H_k^M$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve1`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L)$;
  $\boldsymbol{c_t} \leftarrow$ `add1`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda)$;
end
$\boldsymbol{c_t} \leftarrow$ `sub1`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda)$;
while $\boldsymbol{c_t}[k] \geq H_k^m$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve1`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L)$;
  $\boldsymbol{c_t} \leftarrow$ `sub1`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda)$;
end

Algorithm E.3: `globalntvgen`.

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the sieving region $\mathcal{H}$, the level ℓ with respect to $\mathcal{H}$ and Λ, the bounds on the small linear combinations $\mathcal{A}$
output: list of visited point in $\Lambda \cap \mathcal{H}$
$(T, S) =$ `findV1`$(\Lambda, \mathcal{H}, \ell, \mathcal{A})$; $L \leftarrow \emptyset$;
`sieve1`$(t-1, \boldsymbol{0}, \mathcal{H}, T, S, \Lambda, L)$;
remove duplicates of L;
return L;

Function `fbSub2`(k, $\boldsymbol{c}$, $\mathcal{H}$, S, Λ, ℓ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the level ℓ with respect to $\mathcal{H}$ and Λ
output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

```
while c[k] < H_k^M do
    L ← ∅;
    for v ∈ S[k] do
        if ℓ = t − 1 then
            D ← CVA(c − v, Λ, t − 2);
        else
            D ← CVA(c − v, Λ, ℓ);
        end
        for d ∈ D do
            if c − v − d ∈ H then return c − v − d;
            L ← L ∪ {c − v − d};
        end
    end
    set c to an element of L;
    if k − 1 > ℓ then
        d ← fbAdd2(k − 1, c, H, S, Λ, ℓ); if d ∈ H then return d;
        d ← fbSub2(k − 1, c, H, S, Λ, ℓ); if d ∈ H then return d;
    end
end
return c;                                          // c is out of H
```

Function `sub2`(k, $\boldsymbol{c}$, $\mathcal{H}$, T, S, Λ, ℓ)

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the level ℓ with respect to $\mathcal{H}$ and Λ
output: a new element in $\mathcal{H} \cap \Lambda$ or an element out of $\mathcal{H}$

```
1  for v ∈ T[k] do
2      if c − v ∈ H_{k−1} then return c − v;
3  end
4  if k > ℓ or k = t − 1 then
5      d ← fbSub2(k, c, H, S, Λ, ℓ); c ← c − d;
6      if c ∈ H then T[k] ← T[k] ∪ {c − d}; S[k] ← S[k] ∪ {c − d};
7      c ← d;
8  else
9      c ← (H_0^m − 1, H_1^m − 1, ..., H_{t−1}^m − 1);        // c is out of H
10 end
11 return c;
```

Function `sieve2`$(k,\ \boldsymbol{c},\ \mathcal{H},\ T,\ S,\ \Lambda,\ L,\ \ell)$

input : an integer k defining which nearly-transition-vectors are considered, the current element $\boldsymbol{c} \in \mathcal{H} \cap \Lambda$, the sieving region $\mathcal{H}$, the set T of nearly-transition-vectors, the set S of skew-small-vectors, the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the list L that contains the elements in $\mathcal{H} \cap \Lambda$, the level ℓ with respect to $\mathcal{H}$ and Λ

$\boldsymbol{c_t} \leftarrow \boldsymbol{c}$;
while $\boldsymbol{c_t}[k] < H_k^M$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve2`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L, \ell)$;
  $\boldsymbol{c_t} \leftarrow$ `add2`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda, \ell)$;
end
$\boldsymbol{c_t} \leftarrow$ `sub2`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda, \ell)$;
while $\boldsymbol{c_t}[k] \geq H_k^m$ **do**
  if $\boldsymbol{c_t} \in \mathcal{H}$ **then** $L \leftarrow L \cup \{\boldsymbol{c_t}\}$;
  if $k > 0$ **then** `sieve2`$(k-1, \boldsymbol{c_t}, \mathcal{H}, T, S, \Lambda, L, \ell)$;
  $\boldsymbol{c_t} \leftarrow$ `sub2`$(k, \boldsymbol{c}, \mathcal{H}, T, S, \Lambda, \ell)$;
end

Algorithm E.4: `localntvgen`.

input : the basis $\{\boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_{t-1}}\}$ of Λ, the sieving region $\mathcal{H}$, the level ℓ with respect to $\mathcal{H}$ and Λ, the bounds on the small linear combinations $\mathcal{A}$
output: list of visited point in $\Lambda \cap \mathcal{H}$
$(T, S) =$ `findV2`$(\Lambda, \mathcal{H}, \ell, \mathcal{A})$; $L \leftarrow \emptyset$;
`sieve2`$(t-1, \boldsymbol{0}, \mathcal{H}, T, S, \Lambda, L, \ell)$;
remove duplicates of L;
return L;

E.3 Suitability of Graver basis

In this section, we will proof that the transition vectors listed in Section 6.4.4 when the volume r of Λ is less than I_0 are in the Graver basis of Λ.

Let us consider $(r, 0, 0, \ldots, 0)$, the 0–transition-vector. The vector $(r, 0, 0, \ldots, 0)$ is obviously in the Graver basis. Indeed, in the orthants such that the first coordinate of the vectors is positive, there does not exist two non-zero vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of Λ in the orthant such that $\boldsymbol{u} + \boldsymbol{v} = (r, 0, 0, \ldots, 0)$.

Let us now consider the set of 1–transition-vectors described in Section 6.4.4 is equal to $\{\boldsymbol{b_1}, \boldsymbol{b_1} - \boldsymbol{b_0}\}$. We recall that $\boldsymbol{b_1}$ is equal to $(\lambda_0, 1, 0, 0, \ldots, 0)$, where λ_0 is in $[0, r[$. In the positive orthant, where $\boldsymbol{b_1}$ lies, we cannot express $\boldsymbol{b_1}$ as a sum of two non-zero vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of this orthant. Indeed, let $\boldsymbol{u}$ be equal to $(c_0, 0, 0, \ldots, 0)$ and $\boldsymbol{v} = (c_1, 1, 0, 0, \ldots, 0)$, where c_0 and c_1 are two non-negative integers. Due to the shape of our lattice, the integer c_0 is necessarily a non-zero multiple of r, and, because $0 < \lambda_0 < r$, the integer c_1 will be negative, which is not possible. Then, the element $\boldsymbol{b_1}$ is in the Graver basis of Λ. We can prove, for all $0 < i < t$, that $\boldsymbol{b_i}$ is in the Graver basis of Λ.

The element $\boldsymbol{b_1} - \boldsymbol{b_0} = (\lambda_0 - r, 1, 0)$ is also an element of this Graver basis. Indeed, by trying to express $(\lambda_0 - r, 1, 0)$ as the sum of two elements $(c_0, 0, 0, \ldots, 0)$ and $(c_1, 1, 0, 0, \ldots, 0)$, where c_0 and c_1 are negative, the integer c_0 is necessarily a positive multiple of $-r$, and then c_1 must be positive, which are not allowed.

Then, the vector $\boldsymbol{b_0}$, the vectors $\boldsymbol{b_i}$ and $\boldsymbol{b_i} - \boldsymbol{b_0}$, where i is in $[1, t[$, are in the Graver basis of Λ.

Appendix F

Resultants

F.1 Univariate polynomials

We recall here the algorithm to compute the resultant of univariate polynomials in Algorithm F.1, following the description given in [46, Algorithm 3.3.7]. Let a and b be two polynomials of $\mathbb{Z}[X]$ where $b \neq 0$ and $\deg a \geq \deg b$. The goal of the pseudo-division function (`pseudodiv`) is to compute $(\operatorname{lc} b)^{\deg a - \deg b + 1} a = bq + r$, with q and r two polynomials of $\mathbb{Z}[X]$ with $\deg r < \deg b$.

Function `pseudodiv`(a, b).

Input: two polynomials a and b of $\mathbb{Z}[X]$
Output: two polynomials q (quotient) and r (remainder) of $\mathbb{Z}[X]$
$r \leftarrow a$; $q \leftarrow 0$; $e = \deg a - \deg b + 1$;
while $\deg r \geq \deg b$ **do**
  $s \leftarrow (\operatorname{lc} r) x^{\deg r - \deg b}$;
  $q \leftarrow (\operatorname{lc} b) q + s$;
  $r \leftarrow (\operatorname{lc} b) r - sb$;
  $e = e - 1$;
end
$k \leftarrow (\operatorname{lc} b)^e$;
$q \leftarrow kq$; $r \leftarrow kr$;

F.2 Bivariate polynomials

Computing resultants between bivariate polynomials is one of the needed operations to perform the conjugation and $\mathcal{A}$ polynomial selection. Classically, the resultant between two bivariate integer polynomials is computed with an evaluation and interpolation strategy, as in Algorithm F.2.

Let a and b be two bivariate polynomials of $\mathbb{Z}[X][Y]$. Let d_a^X and d_b^X be the degree of a and b in the variable X, and d_a^Y and d_a^Y the degree in Y. The goal of the following algorithm is to compute $\operatorname{Res}_X(a, b)$. This resultant is a polynomial in Y of degree $\max(d_a^Y, d_b^Y)(d_a^X + d_b^X)$: to compute it, we need to

Algorithm F.1: Resultant between a and b.

Input: two polynomials a and b of $\mathbb{Z}[X]$.
Output: the resultant of a and b.
if $a = 0$ *or* $b = 0$ **then**
  return 0
end
$g \leftarrow 1$; $h \leftarrow 1$; $s \leftarrow 1$; $t \leftarrow (\operatorname{cont} a)^{\deg b}(\operatorname{cont} b)^{\deg a}$;
$a \leftarrow a/\operatorname{cont} a$; $b \leftarrow b/\operatorname{cont} b$;
if $\deg a < \deg b$ **then**
  swap a and b;
  if $\deg a \equiv 1 \bmod 2$ *and* $\deg b \equiv 1 \bmod 2$ **then**
    $s \leftarrow -1$;
  end
end
while $\deg b > 0$ **do**
  $d \leftarrow \deg a - \deg b$;
  if $\deg a \equiv 1 \bmod 2$ *and* $\deg b \equiv 1 \bmod 2$ **then**
    $s \leftarrow -s$;
  end
  compute r, the remainder of the pseudo division of a and b (see Function `pseudodiv`);
  $a \leftarrow b$;
  $b \leftarrow r/(gh^d)$;
  $g \leftarrow \operatorname{lc} a$;
  $h \leftarrow h^{(1-d)}g^d$;
end
if $b = 0$ **then**
  return 0
else
  $h \leftarrow h^{1-\deg a}(\operatorname{lc} b)^{\deg a}$;
  return $s \cdot t \cdot h$
end

compute $\max(d_a^Y, d_b^Y)(d_a^X + d_b^X)+1$ resultants between polynomials by specifying the variable Y in a and b.

Algorithm F.2: Resultant of bivariate polynomials $\mathrm{Res}_X(a, b)$.

Input: two polynomials a and b of $\mathbb{Z}[X][Y]$.
Output: the resultant, in Y, of a and b.
$N \leftarrow \max(d_a^Y, d_b^Y)(d_a^X + d_b^X) + 1$;
$y_{\mathrm{eval}} \leftarrow$ random value;
$L \leftarrow \emptyset$;
for *i in* $[0, N[$ **do**
  $a_{\mathrm{eval}}(X) \leftarrow a(X, y_{\mathrm{eval}})$;
  $b_{\mathrm{eval}}(X) \leftarrow b(X, y_{\mathrm{eval}})$;
  `// If` $\deg(a_{\mathtt{eval}}) \neq d_a^X$ `or` $\deg(b_{\mathtt{eval}}) \neq d_b^X$`, the value` $y_{\mathtt{eval}}$ `is not a valid point to perform the interpolation`
  while $\deg(a_{eval}) = d_a^X$ *or* $\deg(b_{eval}) = d_b^X$ **do**
    $y_{\mathrm{eval}} \leftarrow y_{\mathrm{eval}} +$ random value;
    $a_{\mathrm{eval}}(X) \leftarrow a(X, y_{\mathrm{eval}})$;
    $b_{\mathrm{eval}}(X) \leftarrow b(X, y_{\mathrm{eval}})$;
  end
  $L \leftarrow L \cup \{(y_{\mathrm{eval}}, \mathrm{Res}(a_{\mathrm{eval}}, b_{\mathrm{eval}}))\}$;
  $y_{\mathrm{eval}} \leftarrow y_{\mathrm{eval}} +$ random value;
end
`/* interpolate can be computed by any interpolation algorithm, as the Lagrange interpolation */`
return `interpolate`(L);

Résumé en français

La cryptologie est la science du secret. Cette science se divise en deux disciplines principales : la cryptographie, qui décrit des systèmes de chiffrement, et la cryptanalyse, qui étudie la sécurité de ces systèmes. Avant 1976, la cryptographie était uniquement symétrique, à savoir que la clef de chiffrement est la même que celle permettant le déchiffrement. Ce type de chiffrement a un inconvénient majeur, celui de la distribution des clefs. En effet, garantir que la clef commune entre deux parties ne soit connue que d'elles seules semble impossible sans une rencontre physique de celles-ci. En 1976, Diffie et Hellman, aidés de Merkle [95], décrivent l'échange de clef Diffie-Hellman [55], un protocole permettant de partager à travers un moyen de communication publique une clef commune. Ce protocole marque le début de la cryptographie asymétrique.

1 Le problème du logarithme discret

La sécurité de l'échange de clefs proposé par Diffie et Hellman repose sur un problème mathématique supposé difficile, le problème du logarithme discret. D'autres problèmes supposés difficiles peuvent être utilisés en cryptographie asymétrique, comme par exemple la factorisation d'entier ou la recherche de vecteurs courts dans un réseau. Dans cette thèse, nous nous concentrons sur le problème du logarithme discret défini dans les corps finis, dont l'énoncé est le suivant :

Définition 1 (Problème du logarithme discret)**.** Soient $\mathbb{F}_{p^n}$ un corps finis, g un générateur du groupe multiplicatif $\mathbb{F}_{p^n}^*$ et h un élément de $\mathbb{F}_{p^n}^*$. Le problème du logarithme discret réside dans le calcul de l'entier k tel que $g^k = h$, aussi exprimé $k = \log_g(h)$.

L'utilisation du problème du logarithme discret a amené le développement de plusieurs systèmes de chiffrement, comme le chiffrement et la signature ElGamal, la cryptographie basée sur les couplages et celle basée sur les tores. Pour que tous ces systèmes soient considérés comme sûr, une condition nécessaire est que le problème du logarithme discret soit difficile.

Depuis 1993 et le crible linéaire de Adleman et Demarrais [4], le problème du logarithme discret dans tous les corps finis peut être résolu grâce à un algorithme de complexité sous-exponentielle, qui s'exprime sous la forme $L_{p^n}(\alpha, c) = \exp((c + o(1))(\log p^n)^\alpha(\log\log p^n)^{1-\alpha})$ [147, 126]. Tous les algorithmes de complexité sous-exponentielle permettant le calcul d'un logarithme discret sont de la famille des algorithmes par calcul d'indice [123]. Dans ces algorithmes, le but est de trouver le logarithme discret d'un sous ensemble d'éléments dit *petits*,

puis d'exprimer le logarithme d'un *grand* élément à partir des petits éléments. La réalisation de ces deux parties de l'algorithme est divisée en trois étapes, quand une représentation aisée du corps a été définie, en effectuant :

1. la recherche de relations : l'objectif est de produire un nombre suffisant de relations du type $\sum_i e_i \log_g f_i = \sum_j e_j \log_g f_j$, où les f_i et f_j sont des petits éléments ;
2. l'algèbre linéaire : les logarithmes inconnus des petits éléments deviennent les inconnues d'un système d'équations linéaires, tirées des relations ;
3. le calcul d'un logarithme individuel : le logarithme du grand élément est exprimé en fonction de logarithmes de plus petits éléments jusqu'à n'utiliser que des éléments dont le logarithme est connu.

Si la représentation du corps fini est bien choisie, la complexité du meilleur algorithme pour calculer un logarithme discret dépend du type de corps finis ciblé. Elle est estimée :

— quasi-polynomiale pour les corps finis de petites caractéristiques, souvent $p = 2$ ou $p = 3$;
— en $L_{p^n}(1/3, (64/9)^{1/3})$ pour les corps finis de grandes caractéristiques, par exemple $n = 1$;
— en $L_{p^n}(1/3, (64/9)^{1/3})$ pour les corps finis de moyennes caractéristiques quand n est composé, et en $L_{p^n}(1/3, (96/9)^{1/3})$ quand n est premier.

Dans les cas de moyenne et grande caractéristiques, l'algorithme permettant d'atteindre la complexité en $L(1/3)$ est le crible algébrique, number field sieve (NFS) en anglais. Initialement proposé pour factoriser de grands entiers [127], NFS a été étendu au calcul de logarithmes discrets, tout d'abord pour les corps premiers [77] puis pour d'autres corps finis. L'algorithme NFS est un algorithme par calcul d'indice : cependant, l'étape de définition du corps finis devient une étape à elle seule, appelée sélection polynomiale.

Pour évaluer l'impact des algorithmes sous-exponentiels, les cryptographes essayent d'établir des records de calculs, pour indiquer quelles sont les tailles de clefs obsolètes, et réévaluer la sécurité des systèmes de chiffrement existants. Avec Guillevic, nous avons créé une base de donnée recensant tous ces records, et plus généralement, tous les calculs de logarithmes discrets [83].

2 L'algorithme NFS en moyenne caractéristique

L'une des spécificités de NFS est la façon dont sont collectées les relations. Cette recherche de relations se déroule par l'intermédiaire de corps de nombres K_0 et K_1. Soient f_0 et f_1 deux polynômes à coefficients entiers et irréductibles de degrés supérieurs ou égaux à n, définissant K_0 et K_1. Soit φ un facteur commun de degré n de f_0 et f_1 modulo p. Soit a un polynôme irréductible de degré $t-1$. Si la factorisation de a dans K_0 implique des idéaux de petites normes, et s'il en est de même dans K_1, alors une relation dans $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/\varphi(x)$ impliquant de petits éléments de $\mathbb{F}_{p^n}$ peut être déduite. Le choix de f_0 et f_1 est crucial pour assurer les meilleures probabilités de friabilité dans K_0 et K_1.

Cependant, utiliser des corps de nombres implique également quelques modifications dans la phase d'algèbre linéaire, par exemple l'ajout de colonnes denses

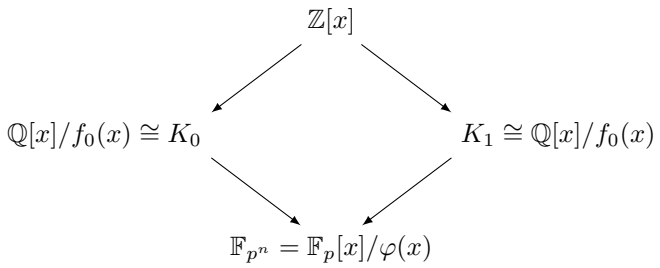


FIGURE 1 – Diagramme de NFS pour le calcul de logarithme discret dans $\mathbb{F}_{p^n}$.

dans la matrice pour l'algèbre linéaire provenant notamment des applications de Schirokauer.

La recherche de relation, représentée en Figure 1 est une étape importante de NFS, et depuis son utilisation dans le crible quadratique par Pomerance [148], elle est réalisée efficacement en utilisant des algorithmes de cribles. Un désavantage de ces algorithmes est la mémoire qu'ils occupent : en effet, ils doivent stocker la valeur des normes de tous les polynômes a qui ont une chance de produire une relation. Comme ce nombre est suffisamment important pour ne pas tenir en mémoire, Pollard a proposé la méthode par special-$\mathfrak{Q}$s [145], qui permet de diviser l'espace des polynômes a en ne considérant que ceux dont la factorisation dans un des corps de nombres fait intervenir l'idéal $\mathfrak{Q}$, ce qui représente un sous-réseau de l'ensemble initial des polynômes a et ainsi un sous-ensemble de plus petite taille des polynômes a pouvant produire une relation. En itérant sur plusieurs special-$\mathfrak{Q}$s, l'espace original des polynômes a est presque entièrement couvert.

Cette méthode par special-$\mathfrak{Q}$s permet également de simplifier le calcul d'un logarithme individuel. Après une première phase de réécriture de la cible, vue comme un élément d'un des corps de nombres, chaque idéal $\mathfrak{Q}$ dont le logarithme n'est pas connu est exprimé en fonction d'idéaux de taille plus petite en forçant $\mathfrak{Q}$ dans la factorisation en idéaux à la manière d'un special-$\mathfrak{Q}$, et ainsi de suite jusqu'à ce que la cible ne soit exprimée qu'en utilisant les idéaux de logarithmes connus.

En 2006, Joux, Lercier, Smart et Vercauteren décrivent la première version de NFS en moyenne caractéristique (NFS-HD) [106], d'une complexité $L_{p^n}(1/3, (128/9)^{1/3} \approx 2.43)$. En 2014, Barbulescu et Pierrot atteignent une complexité en $L_{p^n}(1/3, 2.40)$ avec l'algorithme MNFS, un algorithme proposé initialement pour la factorisation [48] et pour les corps premiers [47], qui exploite plusieurs corps de nombres au lieu de deux pour diminuer la complexité. Nous analysons une variante de MNFS, proposée par Zajac en 2008 [185] utilisant une autre sélection polynomiale que celle utilisée par Barbulescu et Pierrot, qui atteint cette même complexité. En 2015, Barbulescu, Gaudry, Guillevic et Morain proposent une nouvelle sélection polynomiale, permettant d'atteindre la complexité $L_{p^n}(1/3, (96/9)^{1/3} \approx 2.21)$. En combinant cette dernière sélection polynomiale et MNFS, Pierrot obtient la complexité $L_{p^n}(1/3, 2.16)$. Enfin, Sarkar et Singh [160] obtiennent dans le cas médian entre moyenne et large caractéristiques une meilleure complexité, grâce à une nouvelle sélection polynomiale. Leurs caractéristiques sont listées en Table 1.

Variant	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$
JLSV_0	n	petit	n	p
JLSV_1	n	$p^{1/2}$	n	$p^{1/2}$
JLSV_2	n	$p^{n/(d_2+1)}$	$d_2 \geqslant n$	$p^{n/(d_2+1)}$
GJL	$d_3 \geqslant n$	$p^{n/(d_3+1)}$	d_3+1	petit
Conjugation	n	$p^{1/2}$	$2n$	petit
$\mathcal{A}$	$d_5 r \geqslant n$	$p^{n/(d_5(r+1))}$	$d_5(r+1)$	petit

TABLE 1 – Sélections polynomiales pour NFS-HD dans $\mathbb{F}_{p^n}$, où d_5 est un facteur de n et r un entier supérieur ou égale à n/d_5.

L'une des spécificité de NFS-HD réside dans la recherche de relations. Dans le cas classique de la grande caractéristique, la complexité théorique est atteinte en utilisant des polynômes a de degré un : la recherche de relation est réalisée en dimension deux. Avec NFS-HD, le degré de ces polynômes peut être plus grand. Cela implique quelques différences avec le cas classique, dans la sélection polynomiale et la phase de crible.

Le meilleur type de sélection polynomiale se déduit en estimant la taille des normes des polynômes a dans chaque corps de nombres K_0 et K_1. Cependant, même si un seul type se détachait, le nombre de paires possibles est important, et une ou plusieurs offriront les meilleures probabilités de friabilité. En deux dimensions, Murphy [140] a décrit deux fonctions permettant de calculer la qualité d'une paire de polynômes. Avec Gaudry et Videau [72], nous avons étendu en trois dimensions ces critères notamment le critère α qui permet de quantifier le comportement d'un polynôme f modulo de petits premiers, ce qui implique que les idéaux de petites normes seront nombreux, ces idéaux étant très présents dans la factorisation en idéaux.

Définition 2 (Fonction α de Murphy)**.** Soit f un polynôme irréductible à coefficients entiers et un entier $t > 1$. La quantité α du polynôme f a pour valeur $\alpha(f) = \sum_{\ell \text{ premier}} \alpha_\ell(f)$, avec pour tous premiers ℓ,

$$\begin{aligned}\alpha_\ell(f) = \ln(\ell)\big[1/(\ell-1) \\ - \mathbb{A}(\text{val}_\ell(\text{Res}_x(f(x), a(x)), \text{ où } a \in \mathbb{Z}[x], \deg a = t-1, a \text{ irréductible}))\big],\end{aligned}$$

où $\mathbb{A}(\cdot)$ est la valeur moyenne et val_ℓ la valuation ℓ-adique.

Dans le cas de la dimension trois, nous explicitons le terme α_ℓ. Si ℓ est un premier qui ne divise ni le coefficient dominant de f ni son discriminant, alors le second terme vaut, où n_1 et n_2 sont le nombre de racines simples, respectivement multiples, de f modulo ℓ :

$$\alpha_\ell(f) = \frac{\ln(\ell)}{\ell-1}\left(1 - n_1\frac{\ell(\ell+1)}{\ell^2+\ell+1} - 2n_2\frac{\ell^2}{(\ell+1)(\ell^2+\ell+1)}\right),$$

Nous avons également décrit une modification de la sélection polynomiale JLSV_1 permettant de tenir compte de la méthode par special-$\mathfrak{Q}$s, forcé ici du côté 1, impliquant un déséquilibre contrôlé entre $\|f_0\|_\infty = p^{1/2-\varepsilon}$ et $\|f_1\|_\infty = p^{1/2+\varepsilon}$, où $\varepsilon \geqslant 0$.

Nous savons donc maintenant sélectionner les meilleurs polynômes pour la recherche de relations en dimension trois. Cependant, la recherche de relations effectuée avec des cribles adaptés en trois dimensions semble encore trop coûteuse.

Zajac propose un premier crible en ligne en trois dimensions, dont le temps de crible est presque deux fois supérieur à celui de l'algèbre linéaire [185] dans le calcul d'un logarithme discret dans un $\mathbb{F}_{p^6}$ de taille 240 bits. Hayasaka, Aoki, Kobayashi et Takagi, pour le même calcul, obtiennent un temps de recherche de relations similaire, malgré un nouveau crible adapté à la dimension trois [94] et l'utilisation de la méthode par special-$\mathfrak{Q}$s [93]. Pour améliorer ce temps de calcul, nous avons proposé deux nouveaux cribles adaptés en trois dimensions [72] : le plane sieve, une généralisation du crible de Franke et Kleinjung en deux dimensions [61], et le space sieve, un nouveau crible. Ces nouveaux algorithmes de crible, combinés à la sélection polynomiale adapté pour la méthode par special-$\mathfrak{Q}$s et réalisée par Guillevic, nous a permis de gagner un facteur supérieur à vingt sur le temps de crible de l'exemple de Zajac.

L'algorithme NFS-HD n'est pas la seule variante de NFS qui nécessite une recherche de relations en dimension plus grande que deux. En 2015, Barbulescu, Gaudry et Kleinjung analysent l'algorithme tower NFS [22] (TNFS), une variante proposé par Schirokauer [162], qui nécessite de chercher des relations en dimension paire à partir de quatre, particulièrement efficace lorsque p a une forme spéciale et pour la grande caractéristique. À ce propos, nous montrons qu'une autre sélection polynomiale que celle originalement envisagée par les auteurs permet d'atteindre la complexité attendue. La même année, Kim et Barbulescu décrivent l'algorithme extended TNFS [114] (exTNFS), qui s'attaque à la moyenne caractéristique, avec une recherche de relations du même type, à savoir de dimension paire. Le crible en ligne et le plane sieve s'étendent très naturellement à n'importe quelle dimension, comme le montre notre implémentation présente dans CADO-NFS [176]. Cependant, le space sieve a été conçu en dimension trois, et sa généralisation à toutes les dimensions semble plus difficile. De plus, le space sieve ne serait pas toujours le meilleur algorithme en dimension quatre. C'est pourquoi nous avons proposé un cadre général pour cribler dans toutes les petites dimensions.

3 Algorithmes de cribles généralisés

Pour simplifier notre explication, nous ne considérons pas la méthode par special-$\mathfrak{Q}$s dans le reste de notre explication. Soient a un polynôme de degré $t-1$ égal à $a_0 + a_1 x + \cdots + a_{t-1}x^{t-1}$ et $\boldsymbol{a}$ le vecteur des coefficients de a. Tous les polynômes a dont la factorisation en idéaux dans K_0 fait intervenir l'idéal $\mathfrak{R}$ de norme r ont leurs coordonnées décrites par un élément du réseau $\Lambda_{\mathfrak{R}}$ de dimension t et de base $\mathcal{B} = \{(r, 0, 0, \ldots, 0), (\alpha_0, 1, 0, 0, \ldots, 0), (\alpha_1, 0, 1, 0, 0, \ldots, 0), \ldots, (\alpha_{t-2}, 0, 0, \ldots, 0, 1)\}$, où les α_i sont déterminés par l'idéal $\mathfrak{R}$. L'ensemble des coefficients de a est borné par une région de crible $\mathcal{H}$, définie par les intervalles $[H_0^m, H_0^M[\times [H_1^m, H_1^M[\times \cdots \times [H_{t-1}^m, H_{t-1}^M[$. La longueur de ces intervalles sera notée $I_i = H_i^M - H_i^m$.

Nous allons nous appuyer sur la notion de densité des éléments du réseau dans $\mathcal{H}$. La région de crible a un volume fixe $V_{\mathcal{H}}$ égal à $I_0 I_1 \cdots I_{t-1}$. La densité du réseau $\Lambda_{\mathfrak{R}}$ de volume r correspond au nombre d'éléments estimés présents dans l'intersection de $\Lambda_{\mathfrak{R}}$ avec $\mathcal{H}$, soit $V_{\mathcal{H}}/r$. En dimension trois, quand le réseau est très dense, $r < I_0$, le crible en ligne est utilisé ; quand le réseau est moyennement dense, $r < I_0 I_1$, le plane sieve est utilisé ; dans les autres cas, le space sieve est utilisé. Le crible en ligne peut bien évidemment être utilisé dans tous les cas,

mais son efficacité sera réduite quand $r > I_0$. Un des désavantages du space sieve réside dans le caractère heuristique du crible : lors de l'énumération des éléments de l'intersection du réseau $\Lambda_{\mathfrak{R}}$ et de la région de crible $\mathcal{H}$, le space sieve peut ne pas rapporter tous les éléments présents, bien qu'il en rapporte une grande majorité. Pour ces trois cribles, nous définissons la notions de vecteurs de transition approchés.

Définition 3. Soit k un entier de $[0, t[$. Un *k-vecteur de transition approché* est un élément $\boldsymbol{v} \neq \mathbf{0}$ de $\Lambda_{\mathfrak{R}}$ qui permet d'atteindre, à partir de $\boldsymbol{a}$ dans l'intersection de $\Lambda_{\mathfrak{R}}$ et $[H_0^m, H_0^M[\times[H_1^m, H_1^M[\times\cdots\times[H_k^m, H_k^M[\times\mathbb{Z}^{t-(k+1)}$, un nouvel élément $\boldsymbol{a_n} = \boldsymbol{a} + \boldsymbol{v}$ dans cette intersection, tel que les $t-1-k$ dernières coordonnées de $\boldsymbol{a}$ et $\boldsymbol{a_n}$ soient égales et que la coordonnée $\boldsymbol{a_n}[k]$ soit plus grande que $\boldsymbol{a}[k]$.

Ces vecteurs de transition approchés sont l'analogue des vecteurs de Franke et Kleinjung [61] mais ne capturent pas la condition qu'il n'existe pas d'élément ayant une coordonné k entre celle de $\boldsymbol{a}$ et celle de $\boldsymbol{a_n}$. Cependant, vérifier qu'un vecteur est un vecteur de transition approché est aisé : essentiellement, si sa coordonnées j est plus petite que I_j, alors le vecteur est un vecteur de transition approché. La forme des vecteurs de transition approchés en dimension trois est la suivante, où $O(c)$ signifie proche de la valeur c :

— grande densité : les vecteurs de transition approchés ont une forme proche de $(O(r), O(1), O(1))$.

— moyenne densité : les vecteurs de transition approchés ont une forme proche de $(O(I_0), O(r/I_0), O(1))$.

— faible densité : les vecteurs de transition approchés ont une forme proche de $(O(I_0), O(I_1), O(r/(I_0 I_1)))$.

En plus grande dimension, cette forme se généralise, étant donné le niveau de densité ℓ : en dimension trois, la grande densité correspond à $\ell = 0$, la moyenne à $\ell = 1$ et la faible à $\ell = 2$. Étant donné un niveau de densité, la forme générale sera, en simplifiant les notations en n'écrivant plus le $O(\cdot)$, $(I_0, I_1, \ldots, I_{\ell-1}, r/(I_0 I_1 \cdots I_{\ell-1}), 1, 1, \ldots, 1)$.

Fort de cette forme générale, nous allons décrire deux algorithmes permettant de construire de tels vecteurs. Le premier, `globalntvgen`, effectue une réduction de réseau déséquilibrée dans le sens d'obtenir des vecteurs de bases ayant la forme recherchée. Il effectue ensuite de petites combinaisons linéaires entre ses vecteurs pour augmenter le nombre de potentiel de vecteurs de transition approchés. Le second, `localntvgen`, n'effectue la réduction de base déséquilibrée que sur les ℓ premier vecteurs. Après de petites combinaisons linéaires des vecteurs produits, il effectue une recherche de vecteurs proches de chacun des vecteurs n'ayant pas servi au calcul de la base réduite, dans cette même base de ℓ vecteurs. Les modèles de vecteurs de transition approchés pour ses deux différents algorithmes sont listés en Table 2.

Comme le space sieve, ces deux algorithmes ne permettent pas de garantir une énumération exhaustive. De plus, à partir d'un élément de $\mathcal{H} \cap \Lambda_{\mathfrak{R}}$, nous n'avons aucune garantie qu'un des vecteurs de transition approchés permettent de rester dans l'intersection. Dans ces cas, les deux algorithmes disposent de stratégies pour générer de nouveaux vecteurs de transition approchés ou de sortir « proprement » de l'intersection. Puisque les deux algorithmes génèrent leurs vecteurs de transition approchés différemment, la stratégie de générations

k	`globalntvgen`	`localntvgen`
0	$(>0,0,0,0,0,0)$	$(>0,0,0,0,0,0)$
1	$(\cdot,>0,0,0,0,0)$	$(\cdot,>0,0,0,0,0)$
2	$(\cdot,\cdot,>0,0,0,0)$	$(\cdot,\cdot,>0,0,0,0)$
3	$(\cdot,\cdot,\cdot,>0,0,0)$	$(\cdot,\cdot,\cdot,1,0,0)$
4	$(\cdot,\cdot,\cdot,\cdot,>0,0)$	$(\cdot,\cdot,\cdot,0,1,0)$
5	$(\cdot,\cdot,\cdot,\cdot,\cdot,>0)$	$(\cdot,\cdot,\cdot,0,0,1)$

TABLE 2 – Modèles des k-vecteurs de transition approchés quand $\ell = 2$.

de nouveaux vecteurs de transition approchés seront différentes. Pour pouvoir réaliser ces stratégies, les deux algorithmes vont conserver tous les vecteurs produits durant la phase de création des vecteurs de transition approchés, et non pas seulement les vecteurs de transition approchés. Tous ces vecteurs seront appelé vecteurs déséquilibrés.

Définition 4. Soit k un entier de $[0,t[$. Un *k-vecteur déséquilibré* est un élément $\boldsymbol{v} \neq \mathbf{0}$ de $\Lambda_{\mathfrak{R}}$ qui permet d'atteindre, à partir $\boldsymbol{a}$ de $\Lambda_{\mathfrak{R}}$, un nouvel élément $\boldsymbol{a_n} = \boldsymbol{a} + \boldsymbol{v}$ de $\Lambda_{\mathfrak{R}}$, tel que les $t-1-k$ dernières coordonnées de $\boldsymbol{a}$ et $\boldsymbol{a_n}$ soient égales et la coordonnée $\boldsymbol{a_n}[k]$ soit plus grande que $\boldsymbol{a}[k]$.

Soit $\boldsymbol{a}$ un élément de $\Lambda_{\mathfrak{R}} \cap \mathcal{H}$ tel qu'aucun k-vecteur de transition approché ne permette de rester dans $\Lambda_{\mathfrak{R}} \cap \mathcal{H}$. Soit $\boldsymbol{v}$ un k-vecteur de transition approché. L'élément $\boldsymbol{a}+\boldsymbol{v}$ est nécessairement en dehors de l'intersection de $\Lambda_{\mathfrak{R}}$ et $\mathcal{H}$. Pour essayer de trouver un nouveau k-vecteur de transition approché, l'élément $\boldsymbol{a}+\boldsymbol{v}$ est réduit par un vecteur $\boldsymbol{d}$, un élément proche de $\boldsymbol{a}+\boldsymbol{v}$ dans une certaine base, qui sera définie suivant l'algorithme de crible utilisé. Plusieurs vecteurs proches peuvent être utilisés pour tester différentes réductions.

Pour l'algorithme `globalntvgen`, la base pour chercher un vecteur proche est constituée des k premiers vecteurs de $\mathcal{B}$. Ainsi, les coordonnées 0 à $k-1$ de $\boldsymbol{a}+\boldsymbol{v}$ peuvent être modifiées, la coordonnée k étant modifiée par les k-vecteurs déséquilibrés. Cette stratégie est appelée à chaque fois qu'un k-vecteur de transition approché semble manquer.

Pour l'algorithme `localntvgen`, la base est composée des ℓ premiers vecteurs de $\mathcal{B}$. Ainsi, les coordonnées 0 à $\ell-1$ sont modifiées, la coordonnée k étant modifié par les k-vecteurs déséquilibrés. Comme les k-vecteurs déséquilibrés sont creux, la stratégie est appelée récursivement sur les $k-1$-vecteurs déséquilibrés, jusqu'à ce que $k \leqslant \ell$, quand les coordonnées des k-vecteurs déséquilibrés deviennent denses.

4 Implémentation et calculs records

Nous avons implémenté une recherche de relations en dimension trois, au sein du logiciel CADO-NFS [176]. Nous détaillons nos choix d'implémentation et expliquons comment les trois étapes de la recherche de relations sont effectuées : l'initialisation des normes ; les trois cribles (en ligne, plane et space) qui diffèrent dans leur implémentation de l'algorithme général des algorithmes `globalntvgen` et `localntvgen` ; la cofactorisation avec les chaînes d'ECM.

En partie grâce à cette implémentation, nous avons, avec Gaudry et Videau [72], et grâce à la sélection polynomiale réalisée par Guillevic, été capables

de réaliser la recherche de relations dans des corps $\mathbb{F}_{p^6}$ de taille 300 et 389 bits. Ces calculs ont été complétés par l'algèbre linéaire et le calcul d'un logarithme individuel dans un article coécrit avec Guillevic, Morain et Thomé et nous avons établi un nouveau record dans un corps $\mathbb{F}_{p^6}$ de taille 422 bits. De plus, avec Guillevic et Morain, nous avons réalisé le premier calcul complet dans un corps $\mathbb{F}_{p^5}$ de taille 324 bits. Ces calculs sont résumés en Table 3

Corps fini	Taille (bits)	Algorithme	Coût (jours CPU)	Auteurs
$\mathbb{F}_{p^6}^*$	422	NFS	$9,52 \cdot 10^3$	Grémy, Guillevic, Morain et Thomé [85]
	389	NFS	890	Grémy, Guillevic, Morain et Thomé [85]
$\mathbb{F}_{p^5}^*$	324	NFS	386	Grémy, Guillevic et Morain [84]

TABLE 3 – Records de calculs de logarithmes discrets.

5 Conclusion

Dans cette thèse, nous proposons une étude de la recherche de relations pour NFS en moyenne caractéristique. Pour ce type de corps fini, la recherche de relations doit se faire en dimension plus grande que deux pour atteindre la meilleure complexité, que ce soit avec les variantes classiques de NFS autour de sa description originale en 2006, ou avec l'algorithme plus récent exTNFS quand n s'écrit sous la forme d'un produit $\eta\kappa$, sous certaines conditions sur η et κ que nous ne détaillerons pas ici. L'algorithme exTNFS est relativement jeune, et les questions autour de cet algorithme, notamment pour avoir une implémentation de celui-ci, sont grandes. Très brièvement, l'algorithme exTNFS exploite un diagramme similaire à celui de la Figure 1, à ceci près que les ensembles $\mathbb{Z}$ et $\mathbb{Q}$ sont remplacés par un anneau quotient R défini par $\mathbb{Q}(t)/h(t)$, avec h un polynôme irréductible de degré η.

Nous avons mentionné que, pour NFS, la phase de description du corps devenait une étape à part entière, la sélection polynomiale. Il en est de même pour exTNFS, pour lequel trois polynômes sont à définir : h, f_0 et f_1. Pour cette étape, neuf sélections polynomiales, voir Table 4, ont été décrites [114, 161, 158, 115, 159], les plus générales d'entre elles définissant les coefficients de f_0 et f_1 dans R.

Cependant, au contraire de NFS, les critères de qualité, et essentiellement celui défini par la fonction α de Murphy, ne sont pas définis. En effet, si le choix de h semble indépendant de celui de f_0 et f_1, ce polynôme influence la qualité de f_0 et f_1 et les bonnes caractéristiques pour h ne sont pas encore déterminées. En effet, les idéaux des corps de nombres K_0 et K_1 définis par f_0 et f_1 sont eux-mêmes construits à partir d'idéaux dans R. Le critère de qualité α sera donc défini par au moins deux polynômes h et f.

Cet effort pour trouver les bonnes qualités des polynômes n'est pas vain. Fort de notre implémentation de NFS en dimension 3, nous avons implémenté une recherche de relations en dimension 4 pour exTNFS, qui n'utilise ni `globalntvgen`, ni `localntvgen`, mais un précurseur de ces algorithmes, et nous avons pu constater que les tailles des normes pour l'exemple du calcul de logarithme discret dans

	$\deg f_0$	$\|f_0\|_\infty$	$\deg f_1$	$\|f_1\|_\infty$
JLSV_1 (dans $\mathbb{Z}$)	κ	$p^{1/2}$	κ	$p^{1/2}$
JLSV_2 (dans $\mathbb{Z}$)	κ	$p^{\kappa/(d_2+1)}$	$d_2 \geqslant \kappa$	$p^{\kappa/(d_3+1)}$
GJL (dans $\mathbb{Z}$)	$d_3 \geqslant \kappa$	$p^{\kappa/(d_3+1)}$	d_3+1	petit
Conj (dans $\mathbb{Z}$)	κ	$p^{1/2}$	2κ	petit
$\mathcal{B}$ (dans $\mathbb{Z}$)	$d_5 r_5 \geqslant \kappa$	$p^{n/(d_5(r_5+1))}$	$d_5(r_5+1)$	petit
$\mathcal{C}$ (dans R)	$d_6 r_6 \geqslant \kappa$	$p^{(r_6(\eta+1)+\kappa/d_6)/(r_6\eta+1)}$	$d_6(r_6+1)$	petit
gJLSV (dans R)	κ	$p^{\kappa/(d_7+1)}$	$d_7 \geqslant \kappa$	$p^{\kappa/(d_7+1)}$
gConj (dans R)	κ	$p^{1/2}$	2κ	petit
$\mathcal{D}$ (dans R)	$d_9 r_9 \geqslant \kappa$	$p^{\kappa/(d_9(r_9+1))}$	$d_9(r_9+1)$	small

TABLE 4 – Sélections polynomiales pour exTNFS dans $\mathbb{F}_{p^n}$, où d_5, d_6 et d_9 sont des facteurs de κ et $r_i \geqslant \kappa/d_i$.

un corps $\mathbb{F}_{p^6}$ de 389 bits étaient du même ordre que celles obtenues avec NFS dans un corps $\mathbb{F}_{p^6}$ de 300 bits, pour lequel la recherche de relations avait pris une petite dizaine de jours-cœur. Cependant, du fait de la mauvaise qualité supposée des polynômes h, f_0 et f_1, le nombre de relations produites n'était pas celui escompté. De plus, le nombre de polynômes a différents mais ayant la même factorisation en idéaux pouvait dans certain cas être très élevé : cela vient probablement de critères trop peu précis pour définir les caractéristiques pour qu'un polynôme a donne une relation. La phase de crible de la recherche de relations peut être réalisée avec les deux algorithmes que nous proposons, `globalntvgen` et `localntvgen`. L'étape de la cofactorisation ne semble pas problématique puisque les normes des éléments sont toujours des entiers. Concernant l'initialisation des normes, la première phase de la recherche de relations, l'algorithme que nous proposons semble peiner, au delà de la dimension quatre, à tenir un temps raisonnable, par rapport à celui du crible, or plus la dimension est grande, plus les tailles de normes sont faibles, et donc plus le calcul semblerait facile.

Dans la phase d'algèbre linéaire, l'étape de préparation de la matrice est celle qui présente le plus d'inconnues, puisque le calcul du noyau à droite de la matrice ne dépend pas de la façon dont la matrice a été construite. Les deux inconnues que nous pouvons identifier à ce stade sont le passage de la factorisation des normes à la factorisation en idéaux ainsi que la prise en compte des applications de Schirokauer.

Concernant le calcul d'un logarithme individuel, Guillevic a proposé une extension de son travail pour NFS permettant d'améliorer le début de cette phase [88]. La descente par special-$\mathfrak{Q}$ semble elle aussi sous contrôle. Cette dernière étape du calcul semble donc être la moins ardue des quatre phases de l'algorithme exTNFS.

Au terme de cette étude sur le calcul de logarithmes discrets dans les corps finis de moyenne caractéristique, nous avons pu montrer par des expérimentations pratiques la validité des travaux théoriques menés au sujet de NFS depuis 2006. Les six sélections polynomiales ont toutes été considérées dans nos calculs, même si la sélection JLSV_1 était la plus intéressante dans $\mathbb{F}_{p^5}$ et $\mathbb{F}_{p^6}$, la recherche de relations en dimension trois a prouvé son efficacité, même en utilisant une implémentation moins compétitive que celle présente en dimension deux [72], et l'utilisation des techniques d'amorçage [86, 88] pour le calcul du logarithme

individuel a participé à la rapidité des calculs [84, 85]. Les problèmes posés par le nouvel algorithme exTNFS, tant théoriques que pratiques, ouvrent le champ à de nouvelles améliorations et une meilleure compréhension de l'algorithme NFS lui-même, puisque tous ces algorithmes partagent une même structure, celle des algorithmes à calcul d'indice, et une recherche de relations très similaire.

D'autres questions sont aussi ouvertes, notamment au sujet des deux algorithmes `globalntvgen` et `localntvgen` : une meilleure compréhension de leurs qualités et défauts semble nécessaire avant d'envisager une implémentation efficace, de l'un voire des deux algorithmes. L'utilisation de special-$\mathfrak{Q}$s dans plusieurs corps de nombres au lieu d'un seul, par exemple en vue de réaliser la recherche de relations dans MNFS, peut également être considérée.

Résumé

La sécurité des systèmes cryptographiques à clef publique repose sur la difficulté de résoudre certains problèmes mathématiques, parmi lesquels se trouve le problème du logarithme discret sur les corps finis $\mathbb{F}_{p^n}$. Dans cette thèse, nous étudions les variantes de l'algorithme de crible algébrique, number field sieve (NFS) en anglais, qui résolvent le plus rapidement ce problème, dans le cas où la caractéristique du corps est dite moyenne.

NFS peut être divisé en quatre étapes principales : la sélection polynomiale, la recherche de relations, l'algèbre linéaire et le calcul d'un logarithme individuel. Nous décrivons ces étapes, en insistant sur la recherche de relations, une des étapes les plus coûteuses. Une des manières efficaces de réaliser cette étape est d'utiliser des algorithmes de crible.

Contrairement au cas classique où la recherche de relations est réalisée dans un espace à deux dimensions, les corps finis que nous ciblons requièrent une énumération d'éléments dans un espace de plus grande dimension pour atteindre la meilleure complexité théorique. Il existe des algorithmes de crible efficaces en deux dimensions, mais peu pour de plus grandes dimensions. Nous proposons et analysons deux nouveaux algorithmes de crible permettant de traiter n'importe quelle dimension, en insistant particulièrement sur la dimension trois.

Nous avons réalisé une implémentation complète de la recherche de relations pour plusieurs variantes de NFS en dimensions trois. Cette implémentation, qui s'appuie sur nos nouveaux algorithmes de crible, est diffusée au sein du logiciel CADO-NFS. Nous avons pu valider ses performances en nous comparant à des exemples de la littérature. Nous avons également été en mesure d'établir deux nouveaux records de calcul de logarithmes discrets, l'un dans un corps $\mathbb{F}_{p^5}$ de taille 324 bits et l'autre dans un corps $\mathbb{F}_{p^6}$ de taille 422 bits.

Abstract

The security of public-key cryptography relies mainly on the difficulty to solve some mathematical problems, among which the discrete logarithm problem on finite fields $\mathbb{F}_{p^n}$. In this thesis, we study the variants of the number field sieve (NFS) algorithm, which solve the most efficiently this problem, in the case where the characteristic of the field is medium.

The NFS algorithm can be divided into four main steps: the polynomial selection, the relation collection, the linear algebra and the computation of an individual logarithm. We describe these steps and focus on the relation collection, one of the most costly steps. A way to perform it efficiently is to make use of sieve algorithms.

Contrary to the classical case for which the relation collection takes place in a two-dimensional space, the finite fields we target require the enumeration of elements in a higher-dimensional space to reach the best theoretical complexity. There exist efficient sieve algorithms in two dimensions, but only a few in higher dimensions. We propose and study two new sieve algorithms allowing us to treat any dimensions, with an emphasis on the three-dimensional case.

We have provided a complete implementation of the relation collection for some variants of the NFS in three dimensions. This implementation relies on our new sieve algorithms and is distributed in the CADO-NFS software. We validated its performances by comparing with examples from the literature. We also establish two new discrete logarithm record computations, one in a 324-bit $\mathbb{F}_{p^5}$ and one in a 422-bit $\mathbb{F}_{p^6}$.