Fast finite field arithmetic

Robin Larrieu

université PARIS-SACLAY

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

NNT : 2019SACLX073

Thèse de doctorat

Arithmétique rapide pour des corps finis

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'École Polytechnique

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 10 décembre 2019, par

M. ROBIN LARRIEU

Composition du Jury :

Lucia Di Vizio Directrice de Recherche, CNRS (LMV, UMR 8100)	Présidente
Pierrick Gaudry Directeur de Recherche, CNRS (LORIA, UMR 7503)	Rapporteur
Gilles Villard Directeur de Recherche, CNRS (LIP, UMR 5668)	Rapporteur
Magali Bardet Maître de Conférences, Université de Rouen (LITIS)	Examinatrice
Alin Bostan Chargé de Recherche, INRIA (INRIA Saclay)	Examinateur
Jean-Charles Faugère Directeur de Recherche, INRIA (INRIA Paris)	Examinateur
Joris van der Hoeven Directeur de Recherche, CNRS (LIX, UMR 7161)	Directeur de Thèse
Luca De Feo Maître de Conférences, Université de Versailles Saint-Quentin-en-Yvelines (LMV)	Co-directeur de Thèse

Remerciements

Je tiens tout d'abord à remercier mes encadrants, Joris van der Hoeven et Luca De Feo, sans qui ce travail n'aurait jamais vu le jour. Cette formule, quoi qu'un peu convenue voire cliché, est parfaitement adaptée : ils ont su dès le départ me montrer la bonne direction en ciblant des problèmes intéressants, et surtout m'apprendre peu à peu les ficelles du métier de chercheur. Grégoire Lecerf m'a aussi beaucoup aidé ; en plus de m'avoir éclairé sur certains concepts théoriques, il m'a expliqué la philosophie du logiciel MATHEMAGIX ce qui a été précieux pour le code que j'ai pu écrire. Je remercie également les rapporteurs, Pierrick Gaudry et Gilles Villard, pour avoir accepté la lourde tâche de lire en détail les plus de 150 pages qui suivent.

Parce que l'argent est une question récurrente dans tout projet de recherche, je remercie l'École Polytechnique d'avoir financé ma thèse et de m'avoir fourni l'environnement de travail. Je remercie également le CNRS pour avoir pris en charge mes déplacements en conférences, ainsi que l'équipe administrative du laboratoire pour avoir aidé à les organiser.

On dit souvent que le laboratoire est comme une deuxième maison pendant la thèse, je remercie donc le Laboratoire d'Informatique de l'École Polytechnique en général, et l'équipe MAX en particulier, pour l'accueil qui m'a été fait. De même, je remercie les doctorants de l'équipe STREAM pour m'avoir adopté lorsque j'étais le seul thésard dans mon équipe. Heureusement, la vie en thèse ne se limite pas totalement au laboratoire, alors je remercie mes amis de l'association Doc'Union, car je leur dois une grande partie de ma vie sociale de ces trois dernières années. Enfin, je remercie ma famille qui m'a toujours soutenu, et notamment pendant ces derniers mois légèrement chargés (parce que oui, même si écrire quelques articles permet de préparer le terrain, rédiger une thèse reste compliqué).

Contents

Chapter 1

Introduction

Contents

In his novel *Cat's Cradle* (1963), Kurt Vonnegut wrote "Any scientist who can't explain to an eight-year old what he is doing is a charlatan.". If this sentence sounds familiar, it is probably because similar statements are sometimes attributed to Albert Einstein (but there is no evidence he said that). Anyway, based on this principle, the first challenge I will try to address in the next few paragraphs is how to explain computer arithmetic to an eight-year old.

Around that age, one learns how to multiply integers: multiply each digit of the first operand by each digit of the second, then apply appropriate shifts and add everything together. This procedure is sufficiently systematic to be executable by a machine as in the old mechanical calculators, or an electronic circuitry as in modern-day computers. I chose this example because computer algebra is precisely about finding systematic methods, or "algorithms", to perform mathematical operations (such as multiplying integers, solving equations, or manipulating mathematical expressions in general). This allows mathematicians to delegate such calculations to computers, which are much faster and less error-prone than human beings for repetitive tasks.

Closely related to the concept of algorithm is the notion of "complexity", that is the measure of how efficient a procedure is. For instance, the schoolbook version is only one of the possible methods to multiply integers, and some are more efficient than others when the numbers get large.

- By definition, $12 \times 34 = 12 + 12 + \cdots + 12$ is a very simple procedure that could be executed by a machine. But of course, nobody would ever use this method except maybe to multiply by 2 or 3, because this is too inefficient. Indeed, when the numbers to multiply get one digit larger, there are ten times more operations to do!

- The schoolbook method says $12 \times 34 = 12 \times 4 + (12 \times 3) \times 10$ and the subproducts are decomposed as $(1 \times 4) \times 10 + 2 \times 4 = 48$ and $(1 \times 3) \times 10 + 2 \times 3 = 36$ so the result is $48 + 360 = 408$. All in all, this example requires 4 products of 1-digit numbers. More generally, doubling the number of input digits multiplies the number of operations by 4. This is much more efficient than the method above, but one can still do better.

- A method due to Karatsuba [KO63] allows to perform the product 12×34 using only 3 single-digit products. Start by computing $1 \times 3 = 3$ (tens digits), $2 \times 4 = 8$ (units digits) and $(1 + 2) \times (3 + 4) = 3 \times 7 = 21$ (sum tens+units). The final result is then

$$3 \times 100 + (21 - 3 - 8) \times 10 + 8 = 3 \times 100 + 10 \times 10 + 8 = 408.$$

  More generally, to multiply two large numbers, split them in two, then multiply the upper halves together, the lower halves together, and the sums "upper half + lower half" together. Obtaining the final result now requires a few substractions, shifts and additions; which are much easier than multiplications. With this method, doubling the number of input digits only multiplies the number of operations by 3. This is not very interesting for 2-digit numbers because there are more additions, but for larger numbers, this method becomes faster than the schoolbook algorithm.

There are even better algorithms [SS71, Für09, HH19a], but the mathematics involved are way out of the scope of this introduction. As these few references show, even the simple question of integer multiplication does not have a definitive answer[1] because algorithms are constantly improved. Recently, a new algorithm has been proposed [HH19d], and it is believed to be optimal. Indeed, it matches the widely accepted conjecture from the 1971 paper [SS71]; however, nobody has rigorously proven yet that doing better is impossible. For more details on computer algebra in general, and to get an overview of the numerous aspects of this research area, one can refer to textbooks like [GG13, BCG+17].

After this very short introduction to computer arithmetic, let me explain more specifically what this work *Fast Finite Fields Arithmetic* is about. Of course, when designing algorithms, one expects them to be *fast*, in the sense of complexity analysis and/or when running the program and measuring the execution time. If an eight-year old were to say that my thesis is about multiplying large numbers, he or she

[1] See https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_computer_science#Other_algorithmic_problems. In May 2019, the first entry states "What is the fastest algorithm for multiplication of two n-digit numbers?"

would not be that far from the truth, hence the *arithmetic* part. In fact, I do not multiply integers but more complex mathematical objects, that are still "numbers" in some sense. More precisely, the numbers I consider are elements of *finite fields* and certain objects built on them called polynomials. For example, $(3 \mod 7)$ is an element of the finite field $\mathbb{Z}/7\mathbb{Z}$, and $(4X^3 + 2X + 5 \mod 7)$ is a polynomial in one variable X with coefficients in $\mathbb{Z}/7\mathbb{Z}$. Notice that polynomials over $\mathbb{Z}/p\mathbb{Z}$ (where p is a prime number) behave just like numbers written in base p without carry. It is also worth mentioning that more complex finite fields are typically constructed using polynomials over $\mathbb{Z}/p\mathbb{Z}$.

Applications of finite fields and polynomials over finite fields include notably cryptography and error correcting codes. In most cases, the security of an encryption scheme (if based on such objects) is linked to the size of the finite fields/polynomials being used; the same is also true for the capacity of a code to correct many errors. This is a motivation to design efficient algorithms to compute with finite fields: there is no point in securing a communication and making it reliable if doing so slows it down so much it becomes impractical.

1.1 Background

Taking into account that large integers with n digits behave essentially like polynomials of degree n, up to the additional complication of handling the carry, it is not surprising that the history of polynomial multiplication follows closely that of integer multiplication. Obviously, naive $O(n^2)$ and Karatsuba's $O(n^{\log_2 3})$ multiplication algorithms as above are also valid for polynomials over any ring. Similarly, the $O(n \log n \log\log n)$ algorithm by Schönhage and Strassen [SS71] extends quite straightforwardly to polynomials over a (commutative) field, with a slight adaptation required in characteristic 2 [Sch77]. For general rings (not necessarily commutative or associative), the same $O(n \log n \log\log n)$ bound was established by Cantor and Kaltofen in [CK91].

At the time of writing, the story stops there for polynomials over general rings, but it continues in the case of finite fields, and more generally fields of positive characteristic. Indeed, the technique known as Kronecker substitution (found in [Kro82, §4] for multivariate polynomials) shows an equivalence between the multiplication of integers and polynomials over finite fields; a direct application introduces however unwanted logarithmic factors. Nevertheless, techniques for integer multiplication often also work for polynomials over finite fields and conversely. Therefore, a new algorithm in one case can generally be translated (more or less easily) into an algorithm in the other case, with comparable complexity. For instance, Fürer [Für09] gave the bound $O(n \log n \, K^{\log^* n})$ for integer multiplication, for an unspecified constant K and where $\log^*$ denotes the iterated logarithm (number of times the logarithm must be applied until the result is < 1). A few years later, Harvey, van der Hoeven and Lecerf [HHL16b, HHL17] showed that Fürer's bound holds with $K = 8$, both for integers and polynomials over finite fields.

This result was the starting point of my thesis, with the goal to study the practical implications of this theoretical complexity bound. In particular, Harvey, van der Hoeven and Lecerf gave a practical application of their new ideas to multiply polynomials in $\mathbb{F}_2[X]$, by using the properties of the extension field $\mathbb{F}_{2^{60}}$ [HHL16a]. This last paper provided important cases of interest for my work.

The Fast Fourier Transform. Starting with the Schönhage-Strassen algorithm [SS71], fast multiplication algorithm are based on the evaluation-interpolation principle and the Fast Fourier Transform (FFT). Multiplying polynomials by evaluation-interpolation simply consists in evaluating both input polynomials at sufficiently many points, multiplying the evaluations term-by-term, and interpolating the result. This extends to integers by seeing large integers as polynomials with small integer coefficients (for example, $123 = P(10)$ where $P := X^2 + 2X + 3$). Given a n-th root of unity $\omega \in \mathbb{K}$ (with $\omega^n = 1$), the Discrete Fourier Transform (DFT) of P is the tuple $\big(P(1), P(\omega), \ldots, P(\omega^{n-1})\big)$. The expression *Fast Fourier Transform* denotes any algorithm to compute a DFT and its inverse efficiently. In other words, the Fast Fourier Transform provides an efficient evaluation-interpolation scheme.

The first contributions of my thesis are improved variants of the FFT to achieve a faster multiplication. Although those work in a general setting, they are initially motivated by the example of $\mathbb{F}_{2^{60}}$ from [HHL16a].

Polynomial system solving. As a natural follow-up when writing about polynomials, the second topic treated in this thesis is the resolution of systems of algebraic equations. In the everyday language, "solving" means finding all roots that are common to all polynomials in the system. In other words, given a system of polynomials $S^{(1)}, \ldots, S^{(\ell)} \in \mathbb{K}[X_1, \ldots, X_r]$, for which values of $X_1, \ldots, X_r$ (in the algebraic closure of $\mathbb{K}$) do we have $S^{(1)} = \cdots = S^{(\ell)} = 0$? Unfortunately, a formal resolution as in "the solutions of $aX^2 + bX + c = 0$ are $X := (-b \pm \sqrt{b^2 - 4ac})/2a$" is not always possible: even for just one univariate polynomial, such general formulas exist only up to degree 4 [Abe12]. In computer algebra, the definition is more permissive; in fact there are several ways to answer the question.

One may expect an approximate value within the convergence radius of Newton's iteration for each root (which can then be computed with arbitrary precision), or a description of the zero set as an algebraic variety (dimension of the hypersurface, number of irreducible components,...). Sometimes, one is only interested in knowing the number of solutions if it is finite (zero-dimensional case). Alternatively, one may want to know whether a given polynomial P cancels on each of the common roots. By Hilbert's *Nullstellensatz* [Hil90], this is equivalent to P^k being in the ideal $I := \langle S^{(1)}, \ldots, S^{(\ell)} \rangle$ generated by S (for some integer k). For this reason, designing algorithms to compute modulo I can be considered as a resolution of the system in some sense, or at least a first step towards a complete resolution.

In the second set of contributions, I consider the problem of polynomial system solving from the arithmetic point of view. More specifically, I am interested in the sub-problem of the computation modulo the ideal I.

1.2 State of the art

This section reviews some of the classical algorithms related to my work, in particular on the topics of polynomial arithmetic and system solving.

1.2.1 Integer and polynomial arithmetic

Let us first conclude the story of integer and polynomial multiplication where we left it in the previous section. Recall that since [HHL16b, HHL17]), multiplying integers or polynomials over finite fields requires $O(n \log n\, 8^{\log^* n})$ operations.

For integer multiplication, several authors [HHL16b, CT19, HH19a] improved the bound to $O(n \log n\, 4^{\log^* n})$ (thus improving $K = 8$ to $K = 4$ in Fürer's bound), assuming various unproved conjectures of number theory. Shortly after, the bound was obtained inconditionally in [HH19b]. In the same time, a bound $O(n \log n\, 4^{\log^* n})$ was similarly obtained for polynomials over finite fields [HH19c] (assuming a plausible number theoretic conjecture).

Finally, Harvey and van der Hoeven showed in two companion papers, that multiplication can be done in $O(n \log n)$ operations for integers [HH19d] and polynomials over finite fields [HH19e] (again under a conjecture of number theory for the latter). This closes the gap with the conjectured lower bound by Schönhage and Strassen almost 50 years earlier [SS71]. It remains to prove that one cannot go below $n \log n$, and this would be the end of the journey. The contrary would be very surprising, but a proof is really needed: Kolmogorov had conjectured in the early sixties that the complexity of multiplication was intrinsically quadratic, and Karatsuba who heard this talk published his algorithm shortly after, proving the conjecture wrong.

The Fast Fourier Transform. As mentioned earlier, fast multiplication algorithms rely on the Fast Fourier Transform. The most popular FFT algorithm is based on a formula known to Gauss around 1800 [Gau66] and rediscovered by Cooley and Tukey in 1965 [CT65]; see Chapter 2 for more details. This method allows us to decompose a DFT of size $n = n_1 n_2$ into n_1 DFTs of size n_2 followed by n_2 DFTs of size n_1, in a divide-and-conquer fashion. It is typically used if n is a power of two, in this case its complexity is $O(n \log n)$ operations. An earlier version known as the Good-Thomas FFT [Goo58, Tho63] is also used in some situations, but it is limited to the case where n_1 and n_2 are coprime so it cannot be used when n is a power of 2.

If n is even, the most natural approach in the divide-and-conquer decomposition is to choose $n_1 := n/2$ and $n_2 := 2$. In general, one will try to set n_1 large and n_2 small or conversely. However, it was shown in [Bai89] that choosing $n_1 \approx n_2 \approx \sqrt{n}$ leads to better cache efficiency. Notice that this applies for both the Cooley-Tukey and the Good-Thomas FFT.

For coefficients in a finite field, there are restrictions on the admissible sizes for a FFT: in a field with q elements, the size n must divide $q - 1$. Then, sizes cannot always be chosen to be a large prime power, but there may be a large smooth mixed-

radix size (for example with $q = 2^{60}$, the size $3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \approx 17 \times 10^9$ is a divisor of $q - 1$). Alternatively, there is an additive variant of the FFT that exploits the structure of a finite field as an additive group [WZ88, Can89, GM10]; this technique is especially useful in small characteristic.

Using only FFT sizes that are powers of 2 causes a jump phenomenon in multiplication algorithms: whenever the degree gets above a power of two, the runtime increases abruptly, and remains essentially the same for degrees below the next power of two. The Truncated Fourier Transform (TFT) [Hoe04] was designed to mitigate this drawback. The idea is to discard some of the evaluation points (to keep only as many as needed), which leads to a smoother behavior. Further versions of this algorithm have been proposed to improve cache efficiency [Har09] or decrease memory requirement [HR10]. Also an additive version of the TFT has been given in [Mat08, Chapter 6].

Another family of improved variants tries to exploit symmetries in the input to speed-up the computation. For example the FFT for real coefficients is about twice as fast as a FFT of the same size with complex coefficients [Ber68, SJHB87]. Other types of symmetries have been studied in [Ten73, AJJ96, KRO07, VZ07, Ber13].

An efficient implementation of the FFT (more precisely Schönhage-Strassen algorithm) is used for long integer multiplication in GMP [Gra91]. The FFT and the TFT over a prime field are typically used for their polynomial arithmetic in software like FLINT [Har10], MATHEMAGIX [HLM+02] and NTL [Sho01]. Also, the FFTW library [FJ05] provides an efficient implementation of the FFT for real and complex coefficients, typically for applications in physics and signal processing.

Multiplication in $\mathbb{F}_2[X]$. The case of polynomials over the field with 2 elements is of particular interest, with several applications to geometric error correcting codes and algebraic crypto-systems. However, this case raises a specific difficulty: over such a small field, FFT techniques cannot be applied directly because of the lack of evaluation points. Solutions include the triadic version of Schönhage-Strassen algorithm [Sch77], or to use a well-chosen field extension with sufficiently large size. The reference software GF2X [BGTZ08] implements the first strategy by default,[2] and implementations of the second can be found in [HHL16a] (FFT over $\mathbb{F}_{2^{60}}$) and [CCK+17] (additive FFT over $\mathbb{F}_{2^{128}}$ or $\mathbb{F}_{2^{256}}$). Later, the authors of [CCK+17] improved their work by adapting the results of Chapter 4 to the setting of additive FFT [LCK+18, CCK+18].

Applications of fast multiplication. The multiplication of univariate polynomials is a fundamental operation in the sense that many higher-level arithmetic tasks depend directly on it. Let us review quickly a few examples of problems whose complexity is usually expressed as a function of the complexity of multiplication; more details are given in section 2.3. First we need the notation $\mathsf{M}(n)$ for the cost of multiplying polynomials of degree n, and for technical reasons we need to assume that this function increases faster than linearly but slower than quadratically.

[2] GF2X also implements the additive FFT (over $\mathbb{F}_{2^{128}}$) for certain applications.

After multiplication, the next nontrivial operation is the Euclidean division. Cook [CA69] gave a $O(\mathsf{M}(n))$ algorithm for integers, and Strassen [Str73, Lemma 3.5] adapted it for polynomials, using a result from Sieveking [Sie72]. In other words, Euclidean division costs the same as a multiplication, up to a constant factor. The big-Oh constant can be refined by reusing some intermediate results [Hoe10, Har11].

Another classical operation is the multi-point evaluation and interpolation. In the special case of Discrete Fourier Transform, both require $\mathsf{M}(n)+O(n)$ operations by [Blu70]. Notice that this is only helpful if n is *not* smooth, otherwise the FFT is already efficient enough. In the extreme case where n is prime, Rader gave an alternative method [Rad68] with the same complexity. For general sets of points, this requires $O(\mathsf{M}(n)\log n)$ operations (see [GG13, section 10.1] for a classical algorithm); the big-Oh constant has been improved in [BLS03, Ber04] using duality techniques. If the evaluation points are known beforehand and some precomputation on them is allowed, then one may also gain a factor $\log\log n$ [Hoe16]. If the points form a geometric progression $\alpha, \alpha^2, \ldots$ (this is slightly more general than a DFT), then evaluation and interpolation are possible in $O(\mathsf{M}(n))$ operations, by [RSR69, Mer74]. Improved algorithms [BS05] give $\mathsf{M}(n)+O(n)$ and $2\mathsf{M}(n)+O(n)$ respectively.

A third interesting observation is that the arithmetic of multivariate polynomials can be reduced to the univariate case. If $A, B \in \mathbb{K}[X_1, \ldots, X_r]$ have degree d_i in the variable X_i, then their product can be computed in $\mathsf{M}(2^{r-1}\prod d_i)$ operations by the technique of Kronecker substitution [Kro82, §4]. Notice that the product has degree $2d_i$ in the variable X_i, hence essentially $2^r \prod d_i$ coefficients. Alternatively, such polynomials can be multiplied using a multidimensional generalization of the Discrete Fourier Transform, see for example [GM86, Pan94].

1.2.2 Polynomial system solving

The next paragraphs present some of the techniques used for polynomial system solving, and complexity bounds when available. A particular attention is given to Gröbner bases because they are used more specifically in this work, but other common tools are mentioned for completeness. For simplicity, the complexity bounds are given assuming a well-determined system ($\ell = r$), with a finite number of solutions N, and the polynomials $S^{(i)}$ having degree d (Bézout's theorem states that $N \leqslant d^r$ with equality in general). The bounds are valid under the technical assumption that the homogenized polynomials $S^{(i)}$ are in Noether position and form a regular sequence (see e.g. [BCG+17, sections 26.4 and 26.5] for the definition). Although the problem is EXPSPACE-complete in the worst case [May89], such regularity assumptions give meaningful complexity bounds to compare the algorithms.

Gröbner bases. A classical tool for polynomial system solving is given by *Gröbner bases*; the precise definition is recalled in Chapter 6. The modern definition is due to Buchberger [Buc65] who also implemented the first algorithm (but its complexity is hard to estimate). A very similar definition can be found in [Ros59]. Also, standard bases were independently introduced by Hironaka [Hir64] for formal power series. Currently, the state-of-the art algorithms are Faugère's F4 and F5 [Fau99, Fau02].

Actually, partial results from the Gröbner basis theory also appeared in the prior work of Riquier [Riq10], Janet [Jan20], and Thomas [Tho37], in the context of partial differential equations. More recently, Pommaret bases [PH91] and involutive bases [ZB96, GB98] are extensions of these works. Let us also mention that Ritt's characteristic sets [Rit32, Rit50] present some similarities as well; they initiated the theory of triangular sets presented below.

Initially, Gröbner bases are designed to allow effective computation modulo I, but they also give information about the dimension of the variety or the number of solutions if it is finite. A given ideal admits several Gröbner bases: one reduced basis for each *monomial ordering*. The *grevlex* basis is the easiest to compute [BS88]; the *lex* basis is harder to obtain but provides more information, including a way to effectively compute the solutions. In practice, a complete resolution using Gröbner bases starts by computing the grevlex basis, which is then converted into the lex basis using for example the FGLM algorithm [FGLM93] or a Gröbner walk [CKM97].

The F4 algorithm relies on linear algebra on Macaulay matrices [Mac02], then its complexity can be estimated as

$$O\left(rD\binom{r+D}{D}^{\Omega}\right) \tag{1.1}$$

where $D := 1 + \sum(\deg S^{(i)} - 1) = 1 + r \times (d-1)$ is the Macaulay bound [BFS14, Proposition 1]. Here Ω denotes the exponent of matrix multiplication: two $n \times n$ matrices can be multiplied in $O(n^\Omega)$ operations (the current record is $\Omega \approx 2.37$ [LG14], but in practice one rather has $\Omega = \log_2(7)$ [Str69], or even $\Omega = 3$ by the naive algorithm). For the F5 algorithm, we have by [BFS14, Theorem 2] the following complexity estimate:

$$O\left(\frac{r(3d^3)^r}{d}\right). \tag{1.2}$$

Finally, the FGLM algorithm requires $O(rN^3)$ operations, and the bound was recently refined to $O(rN^\Omega)$ [FGHR14]. Notice that, under the aforementioned assumptions, the complexity of F4, F5 and FGLM is polynomial (in fact essentially cubic) in the number N of solutions. Let us mention that fast Gröbner basis algorithms have been used to break certain algebraic cryptosystems [FJ03, FHK+17].

Triangular sets. Characteristic or triangular sets constitute another popular tool with a vast dedicated literature. The historical introduction is due to Ritt in differential algebra [Rit32, Rit50] and Wu in geometry [Wu78].[3] More recent references with a particular focus on polynomial equations can be found in the work of Lazard, Moreno Maza and Schost, see for example [Laz92, ALM99, Sch03].

The complexity of Wu-Ritt algorithm has been estimated to $O(r^{\Omega+1}(d+1)^{O(r^2)})$ in the zero-dimensional case, and $O(\ell^{O(r)}(d+1)^{O(r^3)})$ in general [GM91a, GM91b] (here the regularity assumptions are a bit different from the usual Noether position

[3] Similar ideas appear already in the ancient Chinese monograph *Siyuan yujian* (1303, also known as the *Jade Mirror of the Four Unknowns*) by Zhu Shijie [Hoe77].

/ regular sequence). Also the algorithm from [Laz92] computes a triangular set from a lex Gröbner basis (in the zero-dimensional case); this algorithm is polynomial in the input/output size but its complexity is not precisely analyzed. Apart from these, there are few complexity results about the computation of triangular sets, but they do perform well in practice. For instance, [AM99] compares four algorithms for direct computation of triangular sets, The methods that are considered there are from [Wu87, Laz91, Kal91, Wan93], or in some cases an improved variant (see [AM99] and references therein for more details). Unlike [Laz92], these algorithms are valid for any system of polynomials (including non zero-dimensional case) and do not compute a Gröbner basis first. Depending on the system, one or the other method is most adapted; in some cases it is even faster than computing a Gröbner basis.

Triangular sets are particularly attractive to represent the zero set of a polynomial systems, because they reduce the resolution of a multivariate system to a succession of univariate problems. Moreover, the special case of *triangular systems* gives a simple representation of the quotient algebra $\mathbb{K}[X_1,\dots,X_r]/I$. A triangular system is a family $T = T^{(1)},\dots,T^{(r)}$ such that $T^{(i)} \in \mathbb{K}[X_1,\dots,X_i]$, is monic as a polynomial in X_i, and $I = \langle T^{(1)},\dots,T^{(r)}\rangle$ (such a system is also a lex Gröbner basis). Triangular systems are often used to represent towers of ring extensions [HL18c, HL19b]. It was also shown that any zero-dimensional system is equivalent to a union of triangular systems [Laz92, Proposition 2].

Unlike the computation of the triangular set itself, computation *modulo a triangular system* is well-studied: for $\Delta := \prod \deg T^{(i)}$, multiplication modulo T has $\tilde{O}(3^r\Delta)$ complexity [Leb15]. The complexity decreases to $O(K_\epsilon\Delta^{1+\epsilon})$ for any $\epsilon > 0$ (for some constant K_ϵ depending on ϵ) if each $T^{(i)}$ is in fact univariate in X_i [LMS09]. If the system defines a separable tower over a sufficiently large field, then the complexity decreases to $\Delta^{1+O(1/\sqrt{\log\Delta})}$, by [HL19b, Corollary 2] for towers of fields, and [HL18c, Theorem 7.13] for towers of rings.

Homotopy continuation. A third important family of solvers is based on homotopy methods; the idea is to deform continuously a system with known solutions into the target system. Historically, Gauss used such arguments to prove the fundamental theorem of algebra [Gau99]. Later Drexler [Dre77] and Smale [Sma81, Sma86] applied this principle to solve multivariate systems algorithmically. Unlike the other methods listed in this section, homotopy techniques are purely numerical and therefore a bit far from the topic of the present work; the reader should refer to classical books such as [AG90, Mor09] for more details. Nevertheless, this method is very efficient in practice when the goal is to obtain numerical approximations of the solutions. In fact, software such as PHCpack [Ver99] and Bertini [BHSW06] achieve unmatched performances for many systems.

Geometric resolution. Using an algebraic geometry point of view, Giusti and Heintz introduced a new technique of *geometric resolution* [GH91, GHM+98]. Again, the focus is on the effective computation of the solutions: the output of the method is a parametric representation of the zero set. This was formalized into the Kro-

necker solver [GLS01], with a complexity of $(rd^r + r^4)\tilde{O}(N^2)$. An improved variant over finite fields [HL18b] reduces the complexity to $\tilde{O}(d^{(r-1)(1+\epsilon)}N)$. An important remark on these complexity bounds is that the algorithms for geometric resolution are probabilistic (Las Vegas type), while methods based on Gröbner bases or triangular sets are deterministic. For the last bound from [HL18b], it is important to mention that the gain is only theoretical as it assumes fast modular composition [KU11], of which no efficient implementation is known.

Resultants. Another method to solve a system is to eliminate variables one after the other until reduction to a univariate problem. This is typically done using the resultant, as in [AS88] for example. It is worth mentioning that geometric resolution also uses the resultant, and more specifically the bivariate resultant. Given two bivariate polynomials of degree d, classical algorithms to compute their resultant have cubic complexity $\tilde{O}(d^3)$ (see [GG13, Section 11.2] for more details and references). Recently, Villard [Vil18] gave a sub-cubic algorithm for generic polynomials, with complexity $O(d^{(3-1/\Omega)(1+\epsilon)})$ where Ω is again the exponent of matrix multiplication. Villard's algorithm was implemented in [HNS19]. Finally, as an application of the result from Chapter 8, van der Hoeven and Lecerf gave a $d^{2+o(1)}$ algorithm for the generic bivariate resultant over finite fields [HL19c], but again this complexity bound requires fast modular composition.

Rational univariate representation. In the case of zero-dimensional systems, the solutions admit a *rational univariate representation (RUR)* [Rou99]: each root t of some univariate polynomial G defines a solution of the system given by $X_i := r_i(t)$, where $r_i \in \mathbb{K}(X)$ is a rational function. Such a representation can be computed in polynomial time (with respect to N) assuming the multiplicative structure of $\mathbb{K}[X_1, \ldots, X_r]/I$ is known. A variant of the RUR based on the transposition principle achieves $O(r2^r N^{5/2})$ complexity [BSS03], also taking the multiplicative structure of the quotient algebra as input.

1.3 Contributions

The results of this thesis can be sorted in two main topics, namely polynomial multiplication and the resolution of polynomial systems.

The first part studies the multiplication of univariate polynomials from a practical point of view. The purpose is not to find better asymptotic complexity bounds, but rather to improve the big-Oh constant, which could lead to faster implementations. Recall that the complexity of many other operations depends directly on the efficiency of the multiplication, then any practical improvement here directly improves the other operations as well.

The second part is dedicated to the arithmetic of multivariate polynomials modulo an ideal (as a subproblem of system solving). For simplicity, the study is limited to the case of bivariate polynomials and under certain genericity assumptions on the ideal. In this case, there are optimal (up to logarithmic factors) reduction algorithms; this greatly improves upon the more general classical results.

1.3.1 Multiplication of univariate polynomials

As mentioned earlier, fast multiplication algorithms are based on the Fast Fourier Transform (FFT). For this reason, the improved algorithms take the form of new variants for the FFT. Before presenting the results, it is important to recall the basics on evaluation-interpolation and FFT techniques; Chapter 2 gives some reminders on these classical topics.

The Truncated Fourier Transform. FFT multiplication is well-known for its staircase effect: there are typically jumps in the complexity when the size is a power of two, and little increase between jumps. The Truncated Fourier Transform (TFT) attempts to mitigate this phenomenon; if a FFT of size n can be computed in $\mathsf{F}(n)$ operations for some highly composite n, then one can evaluate and interpolate at $\ell < n$ well-chosen points in time

$$\frac{\ell}{n}\mathsf{F}(n) + O(n)\,.$$

This fact was already known when n is a prime power [Hoe04, Mat08]. Chapter 3 gives a generalization for any (composite) n, such mixed radices being common in a finite field setting: for example over $\mathbb{F}_{2^{60}}$ (as in [HHL16a]), the size n must be a divisor of $2^{60}-1 = 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$. This result was published in [Lar17] and presented at the international conference ISSAC in 2017.

The Frobenius FFT. The second contribution is an algorithm named the *Frobenius FFT* to speed-up FFT computations in extensions of finite fields. To multiply polynomials over a small finite field, it may be necessary to compute FFTs over a larger extension field, which causes an overhead. However, using the symmetries provided by the Frobenius automorphism, it is in theory possible to eliminate this overhead. For instance, assume that P is a polynomial with coefficients in $\mathbb{F}_q$ and one needs to compute its Fourier transform in the extension field $\mathbb{F}_{q^d}$. Then this operation can be done essentially d times faster than if P had coefficients in $\mathbb{F}_{q^d}$ (see Chapter 4). This result was published in [HL17] and presented at the international conference ISSAC in 2017.

Multiplication in $\mathbb{F}_2[X]$. A nice application of the Frobenius FFT is for the multiplication of polynomials over $\mathbb{F}_2$. It turns out that this technique works particularly well for the extension $\mathbb{F}_{2^{60}}$ of the field $\mathbb{F}_2$ and can be implemented efficiently in this case. Combining this with the existing implementation from [HHL16a] led to a speedup by a factor of 2 with respect to previous software. More details are given in Chapter 5. This result was published in [HLL17] and presented at the international conference MACIS in 2017. The implementation can be found in the package JUSTINLINE of the MATHEMAGIX software [HLM+02].

1.3.2 Reduction of polynomials in two variables

Consider the problem of computing with polynomials modulo an ideal; this is typically used to solve systems of polynomial equations or to construct rings with certain algebraic properties.

The main difficulty is: given a polynomial, compute a normal form, that is a canonical representative of its residue class. In one variable (over a field), this is easy because every ideal is principal, so that finding the generator is a GCD computation and the normal form is simply a Euclidean division; all these operations can be done in softly-linear time (i.e. linear up to logarithmic factors). However, this is no longer true for polynomials with several variables, and no optimal algorithm is known in general. Chapters 6-8 present a solution for a simplified situation: we consider polynomials in two variables, zero-dimensional ideals (there is a finite set of isolated common roots), and some regularity assumptions are made.

Reduction techniques. Let $A, B \in \mathbb{K}[X, Y]$ be degree n polynomials in two variables. They have generically n^2 coefficients and n^2 common roots; also the algebra $\mathbb{K}[X, Y]/I$ of polynomials modulo the ideal $I := \langle A, B \rangle$ generated by A, B has dimension n^2 (as a $\mathbb{K}$-vector space). Therefore, one could design algorithms to manipulate these structures whose complexity is quadratic in n, but classical algorithms are at least cubic. Chapter 6 presents the major new ingredients (common to Chapters 7 and 8) to achieve (softly-)quadratic complexity.

Vanilla Gröbner bases Chapter 7 deals with a setting of so-called *vanilla Gröbner bases*. In this situation, the structure of $\mathbb{K}[X, Y]/I$ can be precomputed and once it is done, any polynomial P can be reduced in normal form within quasi-quadratic complexity. In particular, operations like multiplication in $\mathbb{K}[X, Y]/I$ have quasi-optimal algorithms. This result was published in [HL18a] and presented at the international conference ISSAC in 2018. A proof-of concept implementation for SageMath is also available here:

`https://hal.archives-ouvertes.fr/hal-01770408/file/implementation.zip`

Case of the Grevlex order. Chapter 8 considers generating polynomials with a specific shape, that follows the usual degree-lexicographic order. This case is not covered by the previous vanilla situation, but a similar result does hold. Like before, once the structure of $\mathbb{K}[X, Y]/I$ has been precomputed (notice that its representation is different from that of the previous case), any polynomial can be reduced in normal form within quasi-quadratic complexity. But unlike the vanilla case, the structure itself can be computed efficiently: there is a softly-quadratic algorithm to compute the *concise Gröbner basis* that defines the structure of $\mathbb{K}[X, Y]/I$. Notice that it corresponds (at least in some sense) to a resolution of the bivariate system.

This result was presented at the international conference ACA in 2018 and published in the journal "Applicable Algebra in Engineering, Communication and Computing" [HL19a]. Also, the algorithm was implemented in the larrix package of

the MATHEMAGIX software [HLM+02]. The package was presented as a software demonstration in ISSAC 19, and the report will appear in the "ACM SIGSAM Communications in Computer Algebra" [Lar19].

1.3.3 List of publications

This section gives a list of the publications produced while preparing this thesis. All research articles were published in peer-reviewed conference proceedings or scientific journals. All software packages are freely available under the terms of the GNU General Public License (version 2 or later).

Research papers

[Lar17] Robin Larrieu. The Truncated Fourier Transform for mixed radices. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 261–268, New York, NY, USA, 2017. ACM

[HL17] Joris van der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 437–444, New York, NY, USA, 2017. ACM

[HLL17] Joris van der Hoeven, Robin Larrieu, and Grégoire Lecerf. Implementing fast carryless multiplication. In J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos, editors, *Proceedings of Mathematical Aspects of Computer and Information Sciences*, pages 121–136, Cham, 2017. Springer

[HL18a] Joris van der Hoeven and Robin Larrieu. Fast reduction of bivariate polynomials with respect to sufficiently regular Gröbner bases. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 199–206, New York, NY, USA, 2018. ACM

[HL19a] Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, June 2019

[Lar19] Robin Larrieu. Computing generic bivariate Gröbner bases with Mathemagix, 2019. ISSAC software demonstration, to appear in ACM SIGSAM Communications in Computer Algebra

Software

- Routines for the multiplication of polynomials in $\mathbb{F}_2[X]$ (see [HLL17]), in collaboration with Grégoire Lecerf and Joris van der Hoeven. Part of the JUSTINLINE package of MATHEMAGIX [HLM+02]. `http://www.mathemagix.org`
- A proof of concept implementation in SAGEMATH [Sag17] of the algorithms for generic bivariate Gröbner bases [HL18a, HL19a]. Available at `https://hal.archives-ouvertes.fr/hal-01770408/file/implementation.zip`

- An efficient library for computations with generic bivariate Gröbner bases in the setting of [HL19a]. Included in MATHEMAGIX [HLM+02] as the package LARRIX. http://www.mathemagix.org
- An experimental implementation for finite field embeddings, in collaboration with Luca De Feo. Available at https://gitlab.inria.fr/rlarrieu/tower. The method and partial results are given in Appendix A.

1.4 Notations and terminology

To avoid any ambiguity, let us review briefly the most useful mathematical concepts for this thesis and the related notations. Let us also define the setting regarding the representation of objects and the complexity model. Table 1.1 summarizes the notations for reference.

1.4.1 Elementary algebra

The reader should be familiar with the usual algebraic notions such as rings, ideals, polynomials, linear algebra and so on; these classical concepts are recalled for example in [GG13, Chapter 25] and references therein (or in fact any algebra textbook).

A ring or field is said to be *effective* if elements can be represented on a computer or Turing machine, and if there are algorithms to perform the ring operations (including exact and Euclidean division when it makes sense) and the zero test. In other words, this means that computations with effective rings or fields can be carried exactly. For example the ring $\mathbb{Z}$ of integers and the field $\mathbb{Q}$ of rational numbers are effective, but the fields $\mathbb{R}$ and $\mathbb{C}$ of real and complex numbers are not (a Turing machine has only countably many configurations). This does not mean that computing over $\mathbb{R}$ or $\mathbb{C}$ is impossible, but it requires a specific analysis (see e.g. [FHL+07, Moo66] for more details). In the following, all rings and fields are supposed to be effective, and the notation $\mathbb{K}$ denotes an arbitrary effective field.

For finite fields, recall that the number q of elements must be the power of some prime p called the characteristic. The finite field with q elements is unique up to isomorphism and it is usually designated by the notation $\mathbb{F}_q$. Recall that $\mathbb{F}_{p^d}$ contains (a subfield isomorphic to) $\mathbb{F}_{p^e}$ if and only if e divides d. The prime field $\mathbb{F}_p$ is the field $\mathbb{Z}/p\mathbb{Z}$ of integers modulo p, and the extension $\mathbb{F}_{p^d}$ can be represented as the field $\mathbb{F}_p[X]/\langle \mu(X)\rangle$ of polynomials over $\mathbb{F}_p$ modulo some degree d irreducible polynomial μ. In particular, this means that finite fields are effective.

To avoid confusions, we use parentheses for tuples of elements, and angle brackets for generating elements. For example, (a, b) denotes the pair composed of a and b, while $\langle a, b\rangle$ denotes the group or ideal (will be clear from the context) generated by a and b. For the same reason, $(a \mod b)$ denotes the residue class of a modulo b, while $(a \operatorname{rem} b)$ is the remainder in the Euclidean division of a by b. Also, the quotient in the Euclidean division will be denoted by $(a \operatorname{quo} b)$. There are non-Euclidean rings (for example multivariate polynomials) that have standard reduction procedures; in this case we will reuse the rem notation for the result of this reduction operation.

1.4.2 Computer representation of mathematical objects

There are several ways to represent polynomials, by giving explicitly their coefficients or by a straight-line program to evaluate the associated polynomial function. The latter can be interesting for polynomials like $(X+1)^{100}$, but in general the former is easier to manipulate. An explicit description with the coefficients can be *dense* when giving all coefficients, or *sparse* when giving only the nonzero coefficients (but an additional information on the position is then required). Similarly, matrices can be given either in dense or sparse representation.

In this thesis, we always assume dense representation; then the complexity of algorithms will depend on the degree of the polynomials and the size of the matrices. The set of univariate polynomials over $\mathbb{K}$ of degree less than d is denoted by $\mathbb{K}[X]^{<d}$. Similarly, the set of matrices with a rows and b columns is denoted by $\mathbb{K}^{a\times b}$. For polynomials in several variables, the degree is by default the total degree; when something else is meant, the notation will be self-explaining, for example $\mathbb{K}[X,Y]^{\deg_X<u,\deg_Y<v}$.

As usual, coefficients of polynomials or matrices are accessed with subscript indices, for example P_i is the coefficient of X^i in P. For families of polynomials or matrices, a specific element is denoted by a parenthesized superscript, as in $P^{(i)}$. For other families, typically arrays of integers, a specific element is designated by a subscript as usual.

1.4.3 Complexity model

In this work, we assume the *algebraic complexity model* [GG13]: operations in the base field have unit cost, also tasks like manipulation of indices and memory management are neglected. Indeed, this simplifies the analysis (compared for example to multi-tapes Turing machines [Pap94]) and this is especially pertinent in the context of finite fields because elements have a fixed size.

If we were to work with integers or rational numbers, we should additionally take coefficient growth into account to get an accurate estimate of the running time. Another situation where the algebraic complexity model is less pertinent is when we work in different fields simultaneously, because operations in larger fields are more expensive. In particular, the whole point of Chapter 4 is to exploit this fact to achieve a speedup for FFTs over $\mathbb{F}_{q^d}$ if the input is in $\mathbb{F}_q$. For this reason, the analysis there will exceptionally use the *Turing complexity model* for Turing machines with a finite number of tapes [Pap94].

Complexity functions will be typeset using sans-serif font and mention the base field, for example $\mathsf{M}_{\mathbb{K}}(n)$ is the complexity of multiplying two polynomials of degree n in $\mathbb{K}[X]$ (counting the number of operations in $\mathbb{K}$, or steps of Turing machines in Chapter 4). When $\mathbb{K}$ is clear from the context, the subscript can be dropped; also to avoid nested subscripts, M_q is used as a shorthand for $\mathsf{M}_{\mathbb{F}_q}$. The complexity of operations other than multiplication is denoted similarly by replacing the letter M (for example F for the Fourier Transform, C for modular composition, and so on).

Finally, recall the Landau notations $f(n) = O(g(n))$ and $f(n) = o(g(n))$ for $|f(n)| \leqslant C|g(n)|$ and $|f(n)| \leqslant c(n)|g(n)|$ respectively for large enough n, where C is some constant and $c(n) \to 0$. The notation $f(n) = \Theta(g(n))$ (for $f(n) = O(g(n))$ and $g(n) = O(f(n))$) is also useful to outline that f and g are of the same order. Moreover, the soft-Oh notation $f(n) = \tilde{O}(g(n))$ is commonly used in computer algebra [GG13, Chapter 25]. It means essentially the same as the big-Oh notation, up to logarithmic factors: more precisely, it means $f(n) = g(n)\log^{O(1)} g(n)$.

Table 1.1: Summary of notations

Typesetting conventions		
Font	**Example**	**Meaning**
Blackboard bold	$\mathbb{K}$	Rings and fields
Bold	$\mathbf{M}$	Matrices and vectors
Calligraphic	$\mathcal{I}$	Sets of indices or ring elements
Typewriter	$\texttt{foo}(a, b)$	Call to a function in an algorithm
Sans-serif	$\mathsf{M}(n)$	Complexity functions

Algebraic structures	
$\mathbb{A}$, $\mathbb{K}$	An arbitrary effective ring or field respectively
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}$	Set of natural integers, ring of integers, field of rational numbers
$\mathbb{R}, \mathbb{C}$	Fields of real and complex numbers
$\mathbb{F}_q$	Finite field with q elements
$\mathbb{A}/I$	Quotient structure (typically I is an ideal of the ring $\mathbb{A}$)
Mathematical operations	
$\langle a, b\rangle$	Group or ideal (clear from the context) generated by a, b.
quo, rem	Quotient and remainder in the Euclidean division
$a \mod I$	Residue class of a in the quotient $\mathbb{A}/I$
Polynomials and matrices	
Note. Assume dense representation (i.e. a vector of coefficients).	
$\mathbb{K}[X]^{<d}$	The set of polynomials (in 1 variable X) over $\mathbb{K}$ with degree less than d
$\mathbb{K}^{a\times b}$	The set of matrices over $\mathbb{K}$ with a rows and b columns
$P_i, \mathbf{M}_{i,j}$	The coefficient of X^i in P, the coefficient in row i and column j in $\mathbf{M}$
$P^{(i)}, \mathbf{M}^{(i)}$	The i-th element in the family of polynomials P or matrices $\mathbf{M}$
Complexity	
Note. Assume the algebraic complexity model (number of operations in $\mathbb{K}$), except for Chapter 4 where the Turing complexity model is used instead.	
$\mathsf{M}_{\mathbb{K}}(n)$	Cost for the multiplication in $\mathbb{K}[X]^{\leqslant n}$
M, M_q	When $\mathbb{K}$ is clear from the context, shorthand for $\mathsf{M}_{\mathbb{F}_q}$
$\Theta(\cdot)$	The same order: $f = \Theta(g) \Leftrightarrow f = O(g)$ and $g = O(f)$
$\tilde{O}(\cdot)$	Big-Oh up to logarithmic factors: $f = \tilde{O}(g) \Leftrightarrow f = g\log^{O(1)} g$

Part I

Multiplication of univariate polynomials

Chapter 2

Generalities

Contents

This chapter aims to recall classical techniques related to polynomial multiplication. This material can be found in most computer algebra textbooks, such as [GG13, BCG+17]; it does not contain new contributions but is useful to understand the next chapters. In particular, the definitions and notations from section 2.2 are essential to Chapters 3 and 4.

2.1 The evaluation-interpolation principle

The naive quadratic algorithm and Karatsuba's method are based on symbolic manipulation of the coefficients. The technique used in asymptotically fast algorithms (i.e. with quasi-linear $\tilde{O}(n)$ complexity) is slightly different. The idea is to manipulate values of the polynomial functions (at various well-chosen points) instead of the coefficients of the polynomials. Indeed:

- A polynomial of degree less than n is uniquely determined by its values at any n distinct points. (If $\mathbb{A}$ is not an integral domain, the difference of any two of those points must not be a zero divisor.)
- If the ring $\mathbb{A}$ is commutative, operations on polynomials translate directly into operations on the values:

  $$(P \times_{\mathbb{A}[X]} Q)(x) = P(x) \times_{\mathbb{A}} Q(x).$$

  (If $\mathbb{A}$ is not commutative, the above equality is only valid for certain values of x: those that commute with every element of $\mathbb{A}$.)

- Collections of values are easier to manipulate than polynomials: they can be added or (more importantly) multiplied term-by-term, therefore with linear complexity.

Because of these elementary mathematical properties, polynomials can be multiplied using the so-called *evaluation-interpolation* principle: evaluate both input polynomials P, Q at a common sufficiently large set of points (if $\deg PQ < n$, then n points are enough), multiply term-by-term these evaluations to get the values of the product, and finally reconstruct the result by interpolation.

Remark 2.1. We need to assume that $\mathbb{A}$ contains at least n distinct elements.

The challenge is now to perform the evaluation and the interpolation efficiently. Evaluation by Horner's rule $(\cdots((P_{n-1}x + P_{n-2})x + P_{n-3})x + \cdots)x + P_0$ requires $O(n)$ operations, so repeating this n times would take quadratic time. Similarly, interpolation using the Lagrange polynomials

$$L^{(j)}(X) := \frac{\prod_{i \neq j}(X - x_i)}{\prod_{i \neq j}(x_j - x_i)}$$

is at least quadratic because each each $L^{(j)}$ is a degree $n-1$ polynomial.

Remark 2.2. In fact, interpolation using the Lagrange polynomials is quadratic only if the $L^{(j)}$ have been precomputed. Indeed, computing just one of them naively would already be quadratic so the overall complexity would be cubic. Even with fast multiplication (which is precisely what we want to build!) and reusing the intermediate results, the complexity could be made close to quadratic but there would be extra logarithmic factors.

Apparently, this evaluation-interpolation technique is therefore even slower than the naive multiplication algorithm. The solution is to choose carefully the set of evaluation points, to perform both operations more efficiently.

2.2 The Fast Fourier Transform (FFT)

The Discrete Fourier Transform (DFT) is the operation of evaluating a polynomial at a special set of points, namely the n-th roots of unity.

Recall that for a ring $\mathbb{A}$, an element $\omega \in \mathbb{A}$ is a principal n-th root of unity if $\omega^n = 1$ and

$$\sum_{k=0}^{n-1} \omega^{jk} = 0 \text{ for all } j \in \{1, \ldots, n-1\}.$$

If moreover n is invertible in $\mathbb{A}$, then in particular $1, \omega, \ldots, \omega^{n-1}$ are all distinct and the DFT

$$P \in \mathbb{A}[X]^{<n} \to \big(P(1), P(\omega), \ldots, P(\omega^{n-1})\big) \tag{2.1}$$

can be inverted. Indeed, let $\mathbf{F}^{(\omega)}$ be the DFT matrix given by $\mathbf{F}^{(\omega)}_{i,j} := \omega^{ij}$. Since ω is principal, we have

$$\left(\mathbf{F}^{(\omega^{-1})}\mathbf{F}^{(\omega)}\right)_{i,j} = \sum_{k=0}^{n-1} \omega^{(j-i)k} = \begin{cases} 0 & \text{if } i \neq j \\ n & \text{if } i = j \end{cases}$$

so that

$$\mathbf{F}^{(\omega^{-1})} = n(\mathbf{F}^{(\omega)})^{-1}\,. \tag{2.2}$$

In particular, inverting the discrete Fourier transform with respect to some root ω reduces to the computation of the direct DFT with respect to the root ω^{-1}.

Remark 2.3. If $\mathbb{A}$ is a field, then it is sufficient to have a primitive root of unity ($\omega^n = 1$ and $\omega^j \neq 1$ for all $j \in \{1, \ldots, n-1\}$). Indeed, any primitive root of unity in an integral ring is principal, and the characteristic cannot divide n (in characteristic p, we have $X^p - 1 = (X-1)^p$ so there is no primitive p-th root).

2.2.1 The Cooley-Tukey FFT

The Fast Fourier Transform (FFT) denotes an algorithm to compute DFTs efficiently. It is based on the divide-and-conquer principle and the following formula:

$$P(\omega^{u+n_1 v}) = \sum_{i=0}^{n_2-1} \omega^{iu} \cdot \left(\sum_{j=0}^{n_1-1} P_{i+n_2 j}(\omega^{n_2})^{ju} \right) \cdot (\omega^{n_1})^{iv} \quad (\text{if } n = n_1 n_2) \tag{2.3}$$

for all $u < n_1$ and $v < n_2$. This formula was already known to Gauss around 1805 [Gau66], and it was later rediscovered by Cooley and Tukey [CT65]. In formula (2.3), notice that the inner sum corresponds to a DFT of size n_1 and the outer sum to a DFT of size n_2. In other words, a DFT of size $n = n_1 n_2$ can be decomposed into n_2 DFTs of size n_1 followed by n_1 DFTs of size n_2; an example is given in Figure 2.1 for $n_1 = 3$ and $n_2 = 5$. The reindexing around the inner DFT and the order of the output are technical details; this is of practical interest in some implementations, see for example [Bai89, FJ05, HHL16a], and it also simplifies the design of variants in Chapters 3 and 4. We will analyze this more precisely in the next subsections.

Moreover, the smaller DFTs can be decomposed recursively if n_1 and/or n_2 are composite. Then, computing the DFT ultimately boils down to the computation of many DFTs of much smaller order (n/p DFTs of order p for each prime factor p of n). If n decomposes into many small prime factors, then this FFT algorithm is very efficient. Typically, when n is a power of 2, the FFT has $O(n \log n)$ complexity by the classical analysis of divide-and-conquer algorithms. This is also visible when representing on an arithmetic circuit (obtained by fully expanding the recursion) as in Figure 2.2: each elementary DFT of size 2 is represented by a butterfly-like graph ($\bowtie$), each row of butterflies then takes $O(n)$ operations and there are $\log n$ rows. Notice that the butterflies in the top rows are interlaced as a consequence of the reindexing (as in Figure 2.1); also, notice again the permutation of the output that will be made explicit in the next subsection. Clearly, the $O(n \log n)$ complexity

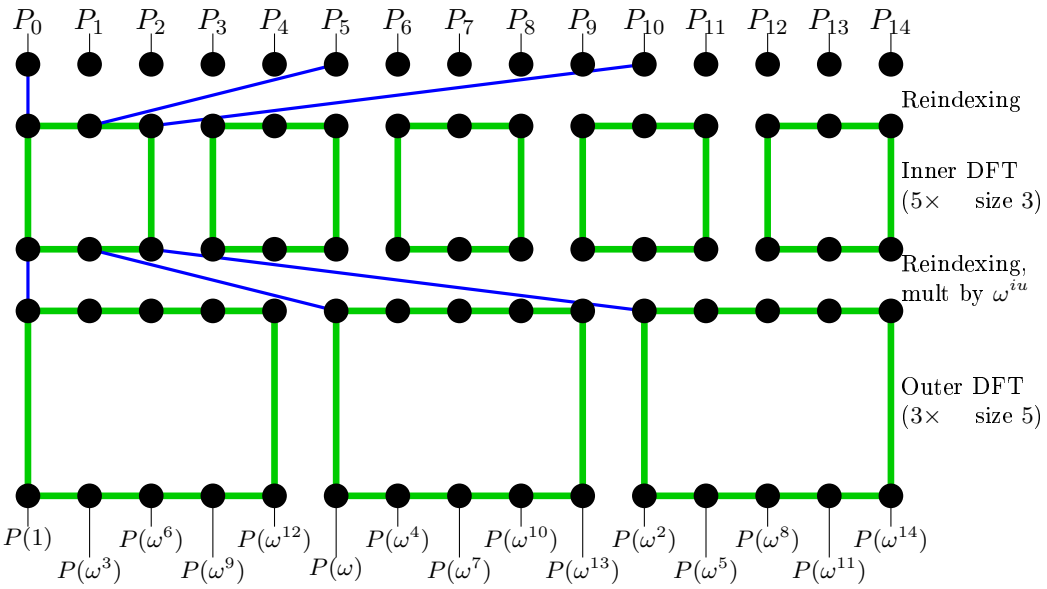


Figure 2.1: Example of Fast Fourier Transform of size $15 = 3 \times 5$.

is also valid when n is any prime power, or more generally if its prime factors are bounded, although larger prime factors increase the big-Oh constant. For more details on the complexity analysis, see for example [CT65] or [GG13, Theorem 8.15].

2.2.2 Generalized bitwise mirrors

When the size of the FFT is a power of 2, it is classical to order the output according to bit-reversed indices, as in Figure 2.2. This definition can be generalized to any composite size and this subsection makes it explicit because the FFT variants in Chapters 3 and 4 rely on this ordering.

In this subsection and in the next, we identify the polynomial $P \in \mathbb{K}[X]^{<n}$ with its vector of coefficients $\mathbf{P} \in \mathbb{K}^n$. Also, we denote by $\hat{\mathbf{P}}$ its DFT: $\hat{\mathbf{P}}_i := P(\omega^i)$. The purpose of this subsection is to define a permutation $i \to [i]_{\mathbf{v}}$ called the $\mathbf{v}$-mirror, that generalizes the bit-reversed indexation and such that $\mathrm{FFT}(\mathbf{P})_i = \hat{\mathbf{P}}_{[i]_{\mathbf{v}}}$.

Let the vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$ be a decomposition of n (i.e. $n = \prod_{j<d} p_j$), with non necessarily prime factors p_j. Then, any index $i < n$ can be uniquely written in the form:

$$i = i_0 \cdot p_1 \cdots p_{d-1} + i_1 \cdot p_2 \cdots p_{d-1} + \cdots + i_{d-2} \cdot p_{d-1} + i_{d-1}$$

with $i_j < p_j$ for all j. With these notations, the $\mathbf{v}$-mirror of i is defined as

$$[i]_{\mathbf{v}} = i_0 + i_1 \cdot p_0 + i_2 \cdot p_0 p_1 + \cdots + i_{d-1} \cdot p_0 \cdots p_{d-2}\,.$$

Notice that when all p_j are 2, this definition coincides with the usual bitwise mirror for the radix-2 FFT. For $h \leqslant d$, let $\mathbf{v}^{(\mathrm{L})} = (p_0, \ldots, p_{h-1})$, $\mathbf{v}^{(\mathrm{R})} = (p_h, \ldots, p_{d-1})$,

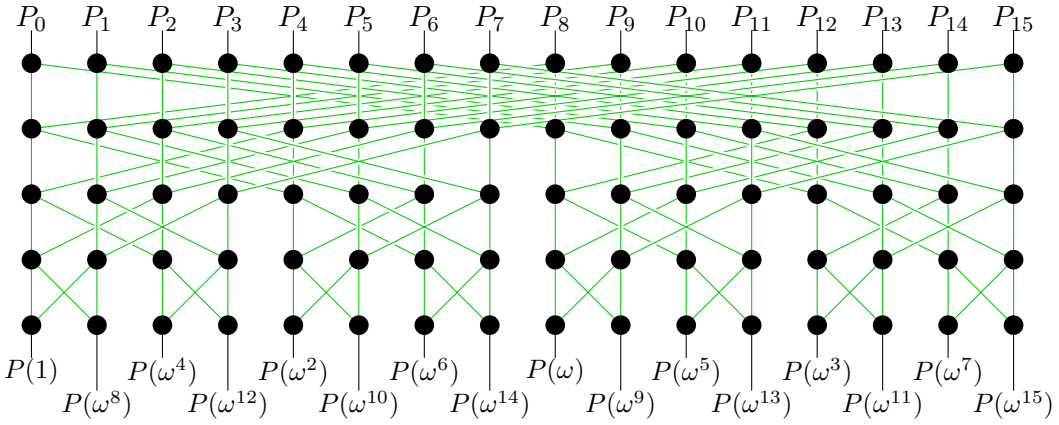


Figure 2.2: A Fast Fourier Transform of size 16 as an arithmetic circuit.

$n^{(\mathrm{L})} = \prod_{j=0}^{h-1} p_j$ and $n^{(\mathrm{R})} = \prod_{j=h}^{d-1} p_j$ (so that $n = n^{(\mathrm{L})} n^{(\mathrm{R})}$). It is easy to show the following basic properties of the $\mathbf{v}$-mirror:

$$[[i]_{\mathbf{v}}]_{\bar{\mathbf{v}}} = i \qquad \text{with } \bar{\mathbf{v}} = (p_{d-1}, \ldots, p_0), \tag{2.4}$$

$$[i]_{\mathbf{v}} = i \qquad \text{if } d = 1 \text{ i.e. } \mathbf{v} = (n), \tag{2.5}$$

$$[I + n^{(\mathrm{R})} J]_{\mathbf{v}} = [J]_{\mathbf{v}^{(\mathrm{L})}} + n^{(\mathrm{L})} [I]_{\mathbf{v}^{(\mathrm{R})}} \qquad \text{for } I < n^{(\mathrm{R})},\ J < n^{(\mathrm{L})}. \tag{2.6}$$

2.2.3 Further notations for the steps of the FFT

Now consider an execution of the FFT algorithm as in section 2.2.1. The recursive calls of the algorithm define a decomposition $n = p_0 \cdots p_{d-1}$ of n, with

$$n_1 = n^{(\mathrm{L})} := p_0 \cdots p_{h-1} \text{ and } n_2 = n^{(\mathrm{R})} := p_h \cdots p_{d-1}.$$

As in the previous subsection, set

$$\mathbf{v} := (p_0, \ldots, p_{d-1}),\ \mathbf{v}^{(\mathrm{L})} := (p_0, \ldots, p_{h-1}) \text{ and } \mathbf{v}^{(\mathrm{R})} = (p_h, \ldots, p_{d-1}).$$

Assume also that the output verifies $\mathrm{FFT}(P)_i = \hat{\mathbf{P}}_{[i]_{\mathbf{v}}}$ (that is, the Discrete Fourier Transform that is returned is reordered according to the $\mathbf{v}$-mirror).

For clarity, we will introduce different input and output vectors. These vectors are actually just names to represent specific parts of the working vector at different steps of the algorithm, but no duplication of data should occur.

- Let $\mathbf{Q}$ (of size n) be the output vector of the algorithm:

$$\mathbf{Q}_i := \mathrm{FFT}(P)_i = \hat{\mathbf{P}}_{[i]_{\mathbf{v}}}.$$

- For each $i < n_2$, let $\boldsymbol{\alpha}^{(i)}$ and $\boldsymbol{\beta}^{(i)}$ be the input and output vectors of the i-th inner DFT (recursive call of the FFT algorithm, each vector has size n_1).

- For each $j < n_1$, let $\boldsymbol{\gamma}^{(j)}$ and $\boldsymbol{\delta}^{(j)}$ be the input and output vectors of the j-th outer DFT (each vector has size n_2).

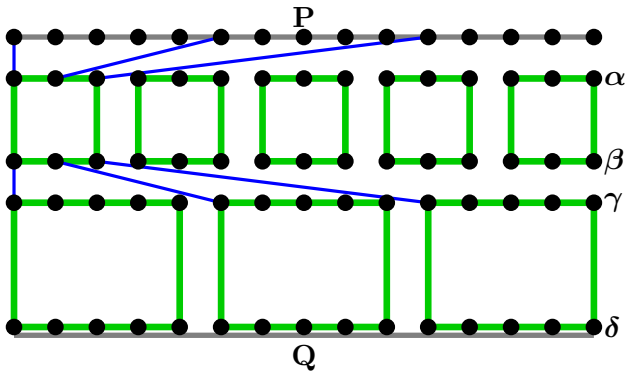


Figure 2.3: Notations for the steps of the FFT.

These notations are illustrated in Figure 2.3. By definition, we have the following properties for all i, j:

$$\boldsymbol{\alpha}^{(i)}_j = \mathbf{P}_{i+n_2 j}\,, \tag{2.7}$$

$$\boldsymbol{\beta}^{(i)}_j = \mathrm{FFT}(\boldsymbol{\alpha}^{(i)})_j = (\widehat{\boldsymbol{\alpha}^{(i)}})_{[j]_{\mathbf{v}^{(\mathrm{L})}}}\,, \tag{2.8}$$

$$\boldsymbol{\gamma}^{(j)}_i = \omega^{i[j]_{\mathbf{v}^{(\mathrm{L})}}} \times \boldsymbol{\beta}^{(i)}_j\,, \tag{2.9}$$

$$\boldsymbol{\delta}^{(j)}_i = \mathrm{FFT}(\boldsymbol{\gamma}^{(j)})_i = (\widehat{\boldsymbol{\gamma}^{(j)}})_{[i]_{\mathbf{v}^{(\mathrm{R})}}}\,, \tag{2.10}$$

$$\mathbf{Q}_{i+n_2 j} = \boldsymbol{\delta}^{(j)}_i\,. \tag{2.11}$$

To verify that $\mathbf{Q}_{i+n_2 j} = \hat{\mathbf{P}}_{[i+n_2 j]_{\mathbf{v}}}$ as expected, let us rewrite the above equations so that the indices match section 2.2.1 and equation (2.3). Let $U, u < n_1$ and $V, v < n_2$ such that $U = [u]_{\mathbf{v}^{(\mathrm{L})}}$ and $V = [v]_{\mathbf{v}^{(\mathrm{R})}}$. Then, we have

$$\begin{aligned}
\boldsymbol{\alpha}^{(i)}_j &= \mathbf{P}_{i+n_2 j}\,,\\
\boldsymbol{\beta}^{(i)}_u &= \mathrm{FFT}(\boldsymbol{\alpha}^{(i)})_u = (\widehat{\boldsymbol{\alpha}^{(i)}})_U\,,\\
\boldsymbol{\gamma}^{(u)}_i &= \omega^{iU} \times \boldsymbol{\beta}^{(i)}_u\,,\\
\boldsymbol{\delta}^{(u)}_v &= \mathrm{FFT}(\boldsymbol{\gamma}^{(u)})_v = (\widehat{\boldsymbol{\gamma}^{(u)}})_V\,,
\end{aligned}$$

This means $\mathbf{Q}_{v+n_2 u} = \hat{\mathbf{P}}_{U+n_1 V}$ by formula (2.3), and we conclude by rewriting (2.6) as $U + n_1 V = [v + n_2 u]_{\mathbf{v}}$.

Remark 2.4. In practice, the inverse Fourier transform is computed by reverting the FFT algorithm rather than using equation (2.2). More precisely, for a composite n, the FFT steps are performed in reverse order (i.e. bottom-up in Figure 2.1 or 2.2), and each base case (prime size) is inverted with equation (2.2), or even by direct computation. Notice that doing so tackles directly the issue of the mirrored indexation.

To design variants of the FFT in the next chapters, we assume that we have at our disposal the algorithms for the plain FFT and its inverse:

Algorithm 2.1. Fast Fourier Transform

Prototype: $\texttt{FFT}(\mathbf{v}, \mathbf{P}, \omega)$
Input: A vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, a vector $\mathbf{P} \in \mathbb{K}^n$, and a primitive n-th root of unity ω (with $n = \prod_{i<d} p_i$).
Output: The vector $\mathbf{Q}$ with $\mathbf{Q}_i = \hat{\mathbf{P}}_{[i]_{\mathbf{v}}}$.

Algorithm 2.2. Inverse Fast Fourier Transform

Prototype: $\texttt{IFFT}(\mathbf{v}, \mathbf{Q}, \omega)$
Input: A vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, a vector $\mathbf{Q} \in \mathbb{K}^n$, and a primitive n-th root of unity ω (with $n = \prod_{i<d} p_i$).
Output: The vector $\mathbf{P}$ such that $\mathbf{Q}_i = \hat{\mathbf{P}}_{[i]_{\mathbf{v}}}$.

2.2.4 The case of finite fields

The field of complex numbers $\mathbb{C}$ contains suitable roots of unity for any order n ($\exp(2\mathrm{i}\pi/n) \in \mathbb{C}$ is a primitive n-th root). However, the situation is more complicated for finite fields: they are *finite*, so there can be only finitely many $n \in \mathbb{N}$ such that there is a primitive n-th root! Also, as we have already mentioned, there is no primitive p-th root in characteristic p. Actually, the roots of unity present in a given finite field are easy to describe: $\mathbb{F}_q$ contains a primitive n-th root if and only if n divides $q-1$. Indeed, it is classical that the multiplicative group $\mathbb{F}_q^\times$ is cyclic of order $q-1$.

Remark 2.5. Using the fact that $\mathbb{F}_q^\times$ is cyclic, it is easy to compute primitive roots for a given order. First, one precomputes $g \in \mathbb{F}_q$ that generates the group $\mathbb{F}_q^\times$. Then, for any n that divides $q-1$, the root $\omega := g^{(q-1)/n}$ is a primitive n-th root of unity.

Since the order n must divide $q-1$, some finite fields are more suitable than others for the FFT, depending on whether $q-1$ is smooth (meaning it has many small prime factors) or not. This brings another complication: the degree of the polynomials to multiply is now bounded by the order of available roots. When the degrees become too large, the solution proposed by Schönhage and Strassen [SS71] (valid for any field $\mathbb{K}$, then generalized to arbitrary rings by Cantor and Kaltofen [CK91]) is to add "virtual" roots of unity. They consider the ring $\mathbb{K}[\xi]/\langle \xi^n + 1\rangle$, so that $\omega := \xi \mod (\xi^n + 1)$ is a primitive $2n$-th root of unity, with an adaptation if $\mathbb{K}$ has characteristic 2 [Sch77].

Another solution is to embed $\mathbb{F}_q$ into a well-chosen extension $\mathbb{F}_{q^d}$ as in [HHL17]. On the one hand, if d is smooth, then there is a root $\omega \in \mathbb{F}_{q^d}$ of smooth order $q^d - 1$. On the other hand, if d is small (say $d = \Theta(\log n)$), then the overhead of computing in $\mathbb{F}_{q^d}$ instead of $\mathbb{F}_q$ is not too large. By the combination of these two ideas, we get a recursive algorithm where the size decreases exponentially with each call; this leads to a complexity of the type $O(n \log n\, K^{\log^* n})$ operations in $\mathbb{F}_q$. The

main difficulty is to find a d that is both sufficiently small and such that $q^d - 1$ is sufficiently smooth. In fact, the constant K depends on how small d can be chosen with the right properties, which gives different bounds assuming different number theoretic conjectures [HHL17, HH19c]. Even more work is needed to achieve the bound $O(n \log n)$ as in [HH19e].

In practice, the extension $\mathbb{F}_{q^d}$ can be chosen and fixed once and for all as long as it is large enough to cover any realistic input. For example, to multiply polynomials in $\mathbb{F}_2[X]$, the paper [HHL16a] suggests to work with the extension field $\mathbb{F}_{2^{60}}$. This field was chosen because

- an element fits in 60 bits (that is almost a machine word),
- the polynomial $\mu := (X^{61} - 1)/(X - 1)$ is irreducible over $\mathbb{F}_2$ of degree 60 (one can represent $\mathbb{F}_{2^{60}}$ as $\mathbb{F}_2[Z]/\langle \mu(Z) \rangle$ and the special form of μ gives efficient modular arithmetic),
- the multiplicative order $2^{60} - 1$ is smooth:

$$2^{60} - 1 = 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$$

  (so that FFT in $\mathbb{F}_{2^{60}}$ is efficient).

Notice that $2^{60} \simeq 10^{18}$ is so large it allows for inputs of billions of gigabytes, which is more than enough for any practical application.

Remark 2.6. It is also possible to use a redundant representation $\mathbb{F}_2[Z]/\langle Z^{61} - 1 \rangle$, where α and $\alpha + \mu(Z)$ represent the same element. This has the advantage that arithmetic modulo $Z^{61} - 1$ is even more efficient; in practice for sequences of additions and multiplications in $\mathbb{F}_{2^{60}}$, one will use the redundant representation for the intermediate results and the non-ambiguous representation for the return value. Notice that the representation modulo $Z^{61} - 1$ requires 61 bits, which also fits in a machine word.

For finite fields, there is a variant of the fast Fourier transform called the *additive FFT* [GM10] that is also used in practice. Without giving much details (only the Cooley-Tukey FFT from section 2.2.1 is used in the next chapters), let us briefly review how it works. Instead of evaluating at the set of roots of unity, the additive FFT considers a $\mathbb{F}_p$-vector-space $\mathcal{V} \subset \mathbb{F}_q$ and evaluates the polynomial at each point of $\mathcal{V}$. If $\mathcal{V}$ has dimension d (hence p^d elements), then the additive FFT with respect to $\mathcal{V}$ decomposes in a divide-and-conquer fashion into p additive FFTs with respect to some $\mathcal{W}$ of dimension $d - 1$. The complexity of the additive FFT is then again $O(n \log n)$, with $n = p^d$. Of course, it is still necessary to work in a field extension (introducing an overhead) if the degrees of the polynomials get too large.

2.3 Some applications of fast multiplication

Multiplication is a fundamental operation that is often used as a basic building block for more complex tasks. In fact, complexity bounds are generally given in terms of the complexity of multiplication. This section gives a few classical examples.

Definition 2.1. A *multiplication time* for a ring $\mathbb{A}$ is a function $\mathsf{M}_{\mathbb{A}} : \mathbb{N} \to \mathbb{R}$ such that polynomials of degree less than n can be multiplied in at most $\mathsf{M}_{\mathbb{A}}(n)$ operations in $\mathbb{A}$.

Moreover, we make the classical assumption that $\mathsf{M}_{\mathbb{A}}$ is somewhere between linear and quadratic; more precisely

$$\mathsf{M}_{\mathbb{A}}(n)/n \leqslant \mathsf{M}_{\mathbb{A}}(m)/m \text{ if } n \leqslant m, \quad \mathsf{M}_{\mathbb{A}}(mn) \leqslant m^2\mathsf{M}_{\mathbb{A}}(n) \tag{2.12}$$

which implies in particular $\mathsf{M}_{\mathbb{A}}(m+n) \geqslant \mathsf{M}_{\mathbb{A}}(n)+\mathsf{M}_{\mathbb{A}}(m)$, and $\mathsf{M}_{\mathbb{A}}(cn) = O(\mathsf{M}_{\mathbb{A}}(n))$ for any positive constant c.

Remark 2.7. To simplify notations, the subscript $\mathbb{A}$ will be dropped when $\mathbb{A}$ is clear from the context, also M_q will be used as a shorthand for $\mathsf{M}_{\mathbb{F}_q}$. Recall that for a sufficiently large C, one may take $\mathsf{M}_{\mathbb{A}}(n) = Cn^2$ by the naive algorithm, or $\mathsf{M}_{\mathbb{A}}(n) = Cn\log n\log\log n$ by [SS71, CK91], or $\mathsf{M}_{\mathbb{F}_q}(n) = Cn\log n$ for finite fields by [HH19e]. Notice that all these functions verify the hypotheses (2.12).

2.3.1 Euclidean division

Let $A, B \in \mathbb{K}[X]$ of degrees $n \geqslant m$ respectively. Let $\bar{A}$, and $\bar{B}$ be the polynomials obtained by reversing the order of the coefficients ($\bar{A} := X^n A(1/X)$ and similarly for $\bar{B}$). By Newton's iteration

$$\begin{aligned} P^{(0)} &:= 1/\bar{B}_0, \\ P^{(i+1)} &:= (2P^{(i)} - \bar{B}(P^{(i)})^2) \operatorname{rem} X^{2^{i+1}}, \end{aligned}$$

we find a polynomial $P^{\#} := P^{(\lceil \log_2(n-m)\rceil)}$ such that $P^{\#}\bar{B} \equiv 1 \mod X^{n-m}$.

Then, $\bar{Q} := \bar{A}P^{\#} \operatorname{rem} X^{n-m}$ is a degree $n-m$ polynomial that verifies the relation $\bar{Q}\bar{B} \equiv \bar{A} \mod X^{n-m}$. Reversing the order of coefficients, we get that A and QB (with $Q := X^{n-m}\bar{Q}(1/X)$) have the same $n-m$ leading terms. In other words, $A \operatorname{quo} B = Q$, and therefore $A \operatorname{rem} B = A - BQ$.

This algorithm is sometimes known as the Cook-Sieveking-Kung algorithm. Cook [CA69] first described it for integers, followed by Sieveking [Sie72] and Kung [Kun74] for the inversion of power series. Strassen [Str73, Lemma 3.5] then reduced the Euclidean division of polynomials to the inversion of power series as before. We can check that this algorithm has complexity $O(\mathsf{M}(n))$. For a more precise description of the algorithm and a complete complexity analysis, see for example [GG13, section 9.1].

Refinements are also possible in the so-called *FFT model*: assuming FFT multiplication, if there is a common operand in several multiplication, then its Fourier transform is computed only once. Using this principle in the above Newton iteration leads to an improvement in the big-Oh constant [Hoe10, Har11].

2.3.2 Multi-point evaluation and interpolation

Consider a polynomial $A \in \mathbb{K}[X]^{<n}$ and n points $x_0, \ldots, x_{n-1}$. It is possible to compute the evaluations $(A(x_i))_{i<n}$ efficiently with a divide-and-conquer algorithm.

Indeed, let $M^{(l)} := \prod_{i<n/2}(X - x_i)$ and $M^{(r)} := \prod_{n/2\leqslant i}(X - x_i)$; then $A \operatorname{rem} M^{(l)}$ has the same values on $x_0, \ldots, x_{(n+1)\operatorname{quo}2-1}$ and has half the degree (and similarly for $A \operatorname{rem} M^{(r)}$).

Conversely, given n (distinct) points $x_0, \ldots, x_{n-1}$ and n values $y_0, \ldots, y_{n-1}$, there is a divide-and-conquer algorithm to find $A \in \mathbb{K}[X]^{<n}$ such that $A(x_i) = y_i$ for all i. Indeed, if $A^{(l)}, A^{(r)} \in \mathbb{K}[X]^{<n/2}$ are such that

$$A^{(l)}(x_i) = y_i \text{ for } i < n/2 \quad \text{and} \quad A^{(r)}(x_i) = y_i \text{ for } n/2 \leqslant i,$$

then $A := M^{(l)}A^{(r)} + M^{(r)}A^{(l)}$ is a solution to the problem.

Assuming the results are suitably reused, both the evaluation and interpolation can be done using $O(\mathsf{M}(n)\log n)$ operations. For more details, see for example [GG13, Chapter 10].

A general algorithmic theorem known as the *transposition principle* [Bor57, Fid72] states that given an algorithm to compute $\mathbf{x} \mapsto \mathbf{Mx}$ for a matrix $\mathbf{M}$, one may find an algorithm for the transposed map $\mathbf{y} \mapsto \mathbf{M}^\top\mathbf{y}$ with essentially the same complexity. Applied to the problem of multi-point evaluation and interpolation, this result led to improvements in the complexity bound by a constant factor [BLS03, Ber04].

Alternatively, the FFT model is helpful if the set of evaluation points is fixed beforehand [Hoe16]: assuming precomputation of the Fourier transforms for $M^{(l)}, M^{(r)}$ (and their recursive splitting, all of which only depend on the evaluation points), it is possible to gain a factor $\log\log n$.

Special sets of points. The above method is valid for any set of points, but there are more efficient algorithms if the evaluation points have a special structure. A typical example is the case of a geometric progression $x_i = \alpha^i$ for some α. In this case, there are $O(\mathsf{M}(n))$ complexity bounds, that is better by a logarithmic factor than the general case.

If α is a primitive n-th root (that is, we need to compute a DFT), then Bluestein's transform [Blu70] allows to perform the evaluation in $\mathsf{M}(n)+O(n)$ operations. Also, the interpolation has the same complexity by formula (2.2). Notice that this is useful if n is *not* smooth, otherwise the FFT algorithm is efficient enough. When n is prime, Rader's reduction [Rad68] is another way to compute DFTs with this complexity.

More generally for any geometric progression, evaluation and interpolation can be done using $O(\mathsf{M}(n))$ operations. Such algorithms are found in [RSR69, Blu70] (evaluation) and [Mer74] (interpolation), and recalled in [BCG+17, section 5.4]. The current best bounds are due to Bostan and Schost [BS05], who gave algorithms of complexity $\mathsf{M}(n)+O(n)$ for the evaluation and $2\mathsf{M}(n)+O(n)$ for the interpolation.

Comparison of evaluation and interpolation. It was shown that, over a field of characteristic 0, evaluation and interpolation are essentially equivalent [BS04]. More precisely, let $\mathsf{E}(n)$ and $\mathsf{I}(n)$ denote the complexity of multipoint evaluation and interpolation respectively (possibly assuming special sets of points). Then we have $\mathsf{E}(n) \leqslant 2\mathsf{I}(n) + O(\mathsf{M}(n))$ and conversely $\mathsf{I}(n) \leqslant 3\mathsf{E}(n) + O(\mathsf{M}(n))$.

2.3.3 Multiplication in several variables

Let $A, B \in \mathbb{K}[X_1, \ldots, X_r]$ be polynomials in r variables, with degree less than d_i in each variable X_i. By the technique of Kronecker substitution [Kro82], it is possible to compute the product AB as one product $\tilde{A}\tilde{B}$ in one variable Y and degrees less than $2^{r-1}d_1 \cdots d_r$. The product AB then requires $\mathsf{M}(2^{r-1}d_1 \cdots d_r)$ operations in $\mathbb{K}$. Notice that the polynomial $\tilde{A}\tilde{B}$ has degree less than $2^r d_1 \cdots d_r$, which is essentially the number of coefficients of AB; therefore the factor 2^{r-1} should not be considered as an overhead.

The technique consists in rewriting X_r as $X_{r-1}^{2d_{r-1}}$ to eliminate the variable X_r, and repeating until just one variable remains. In other words, one sets

$$\tilde{A}(Y) := A(Y, Y^{2d_1}, Y^{4d_1d_2}, \ldots, Y^{2^{r-1}d_1 \cdots d_{r-1}}),$$

of degree less than $2^{r-1}d_1 \cdots d_r$, and similarly for $\tilde{B}$. Clearly, the multivariate product AB verifies

$$AB(Y, Y^{2d_1}, Y^{4d_1d_2}, \ldots, Y^{2^{r-1}d_1 \cdots d_{r-1}}) = \tilde{A}(Y)\tilde{B}(Y).$$

Since AB has degree less than $2d_i$ in each variable X_i, the result AB can be reconstructed from the univariate product $\tilde{A}\tilde{B}$.

Remark 2.8. As an alternative, one could consider $\mathbb{A} = \mathbb{K}[X_1, \ldots, X_{r-1}]$, use the canonical isomorphism $\mathbb{K}[X_1, \ldots, X_r] \cong \mathbb{A}[X_r]$, and reduce recursively to perform operations in $\mathbb{A}$. However, most computer algebra systems feature highly-optimized univariate polynomial multiplication for the basic rings (such as $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{F}_q$, ...), therefore Kronecker substitution is generally more efficient in practice.

Let us mention that the evaluation-interpolation principle extends to the multivariate setting [Pan94, HS13]. Applying this method directly can also be useful in some situations.

Chapter 3

The Truncated Fourier Transform

Contents

Note. *The results in this chapter were published in* [Lar17].

The standard version of FFT-based multiplication algorithms uses transforms whose size n is a power of two. This requirement causes the following drawback: when a polynomial of degree less than d is considered (or when d evaluation points are needed) with d slightly larger than 2^k, one must perform a FFT of order 2^{k+1}. This causes a significant overhead since up to twice as many values are computed as what is actually needed.

This jump phenomenon can be mitigated by allowing a more precise choice of $n \geqslant d$. For example, instead of requiring $n = 2^k$, one can allow more general products of small primes such as $n = 2^k 3^\ell 5^m$. FFTs of such sizes reduce to DFTs of sizes 2, 3 and 5, for which efficient codelets are implemented e.g. in the FFTW3 library [FJ05]. Alternatively, optimized radix-2 methods may be preferred because of their simplicity (fewer base cases to handle). For example, the *FFT pruning* [Mar71] aims to reduce the overhead for a zero-padded sequence. Another example is Crandall and Fagin's *Discrete Weighted Transform* [CF94], which reduces a problem of size $d < 2^k$ to two problems of size 2^ℓ and 2^m with $d < 2^\ell + 2^m < 2^k$.

Another elegant solution to this jump phenomenon is to use the *Truncated Fourier Transform* [Hoe04]. The TFT behaves as a usual FFT of order 2^k, but it performs a multipoint evaluation with exactly the desired length, while avoiding the computation of all intermediate values that are not needed for obtaining the output. Moreover, the interpolation can be performed with the same complexity using the inverse TFT. Improvements to this algorithm were made to reduce memory usage [HR10], and improve cache friendliness [Har09]. Mateer [Mat08, Chapter 6] also proposed a different formulation of the TFT inspired by Crandall and Fagin's reduction. He mentions that this alternative formulation can be used with a few adaptations when $n = 3^k$, or another prime power; he also shows that the additive FFT (as in [GM10]) admits a truncated variant as well.

The methods discussed above rely on a choice of n with a very specific form. This requires the base field $\mathbb{K}$ to contain primitive n-th roots of unity for all such n. This is true for $\mathbb{K} = \mathbb{C}$, but in general, the choice of roots of unity is restricted. It is always possible to add virtual roots of unity (with certain restrictions on their order if the field has non-zero characteristic) as in the Schönhage-Strassen algorithm [SS71], but this extension causes computational overhead.

Assume that the choice of roots of unity is restricted by both our base field $\mathbb{K}$ and practical considerations. Let $\mathcal{S} \subset \mathbb{N}$ denote the set of orders n for the roots of unity that can be used in FFTs. For example, the use cases mentioned above assume $\mathbb{K} = \mathbb{C}$, and the only criterion is n having small prime factors. This leads to sets of the form $\mathcal{S} = \{2^k | k \in \mathbb{N}\}$, or $\mathcal{S} = \{2^k 3^\ell 5^m | k, \ell, m \in \mathbb{N}\}$. In a finite field $\mathbb{F}_q$, there are roots of unity only for specific orders, so one would have $\mathcal{S} = \{n | n \text{ divides } q - 1\}$. As discussed previously, there is a jump phenomenon at elements of $\mathcal{S}$; the present chapter aims to mitigate this drawback.

Overview of the results: This chapter presents a generalization of van der Hoeven's algorithm [Hoe04] for any FFT size n. Recall that [Hoe04] requires $n = 2^k$, and Mateer [Mat08, Chapter 6] extended it to $n = p^k$ for all primes p, but the case of mixed radices remained open.

Here we focus on the usual Cooley-Tukey FFT as in section 2.2; we do not consider the case of additive FFTs. Notice that this is not an issue since the size of an additive FFT is by definition a power of the characteristic, so that Mateer's work [Mat08, Chapter 6] is already complete in this case.

3.1 The Truncated Fourier Transform for mixed radices

This section generalizes the Truncated Fourier Transform (TFT) [Hoe04] for an arbitrary order $n = p_0 p_1 \cdots p_{d-1}$. Given a vector $\mathbf{P}$ of length n, the TFT computes $\ell \leqslant n$ well chosen values of the Discrete Fourier Transform of $\mathbf{P}$. Note that these are not necessarily the first ℓ values of $\hat{\mathbf{P}}$: actually, we consider the mirrored indexation (section 2.2.2) with respect to $\mathbf{v} = (p_0, p_1, \ldots, p_{d-1})$. The Truncated Fourier Transform of $\mathbf{P}$ with size n and length ℓ is the vector

$$\mathbf{T} = (\mathbf{T}_i)_{i<\ell} = (\hat{\mathbf{P}}_{[0]_\mathbf{v}}, \ldots, \hat{\mathbf{P}}_{[\ell-1]_\mathbf{v}}) .$$

The algorithm presented in this section aims to perform less computation than by simply computing the DFT of $\mathbf{P}$ and discarding the unused values.

3.1.1 Atomic transforms

At first we consider the following base case: given $\mathbf{P} := (P_0, \ldots, P_{p-1})$ for p prime or reasonably small, we want to compute directly the TFT $(\hat{P}_0, \ldots, \hat{P}_{\ell-1})$. To do so, one can naively apply Horner's rule for each value as in formula (2.1), which is especially efficient for small p (principle of a specialized *codelet*). For larger p, it becomes more interesting to compute the full DFT, then discard unused values. A full DFT of a such size can be computed using efficient transformations such as Rader's algorithm [Rad68] and Bluestein's transform [Blu70].

Remark 3.1. We do not use a mirrored indexation in the base case because of property (2.5), which states that the mirror operation with respect to a vector of length 1 is the identity.

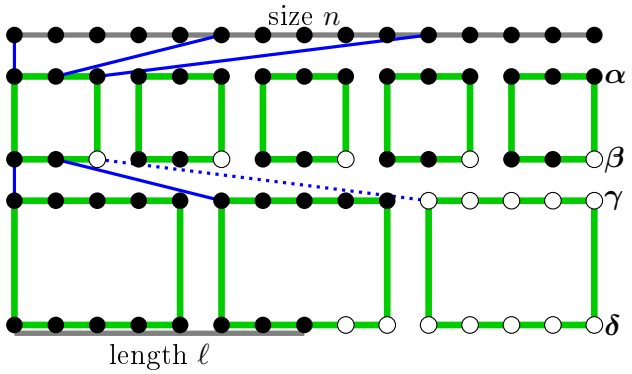
We assume that these considerations translate into the following algorithm:

Algorithm 3.1. Truncated Fourier Transform (base case)

Prototype: `atomicTFT`$(n, \ell, \mathbf{P}, \omega)$
Input: Integers $n \in \mathbb{N}$ and $\ell < n$, a vector $\mathbf{P} \in \mathbb{K}^n$, and a primitive n-th root of unity ω.
Output: The vector $\mathbf{T} = (\hat{\mathbf{P}}_i)_{i<\ell}$.

3.1.2 General idea

Consider the Cooley-Tukey FFT and its recursive formulation as in sction 2.2.1 for a composite $n = n_1 n_2$. Assume that only $\ell \leqslant n$ values of the output are actually needed. Then the plain FFT algorithm can be modified to avoid computation of irrelevant intermediate values.



Values marked by a white dot are not needed in the case of a TFT.

Figure 3.1: Overview of the Truncated Fourier Transform (TFT)

Recall the notations from section 2.2.3: $\boldsymbol{\alpha}^{(i)}$ and $\boldsymbol{\beta}^{(i)}$ are the input/output of the i-th inner FFT; similarly $\boldsymbol{\gamma}^{(j)}$ and $\boldsymbol{\delta}^{(j)}$ are the input/output of the j-th outer FFT. If we want to return the vector $(\hat{\mathbf{P}}_{[i]_v})_{0\leqslant i<\ell}$, then according to relation (2.11), the vectors $\boldsymbol{\delta}^{(j)}$ need to be computed only for $j < m := \lceil \ell/n_2 \rceil$. This means by definition (2.10) that only the $\boldsymbol{\gamma}^{(j)}$ with $j < m$ are needed. From formula (2.9), we conclude that for every $i < n_2$, only the first m values of $\boldsymbol{\beta}^{(i)}$ need to be computed. Moreover, if $q := (\ell \operatorname{quo} n_2) < m$, only the $r = (\ell \operatorname{rem} n_2)$ first values of $\boldsymbol{\delta}^{(q)}$ are needed. Figure 3.1 gives a visual representation of which values are actually needed.

3.1.3 Presentation of the algorithm

The previous discussion suggests that a TFT of order $n = n_1 n_2$ can be decomposed into n_2 TFTs of order n_1 followed by m TFTs of order n_2 (as for the usual FFT). If the top-level TFT has length ℓ, then the inner TFTs have length $m = \lceil \ell/n_2 \rceil$. Most of the outer TFTs are actually usual FFTs; only the last one may be a TFT of length $r = (\ell \operatorname{rem} n_2)$ (unless $r = 0$). This leads to the following recursive algorithm:

Algorithm 3.2. Truncated Fourier Transform

Prototype: $\texttt{TFT}(\mathbf{v}, \ell, \mathbf{P}, \omega)$

Input: a vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, an integer $\ell \leqslant n$, a vector $\mathbf{P} = (P_i)_{i<n}$, and a primitive n-th root of unity ω (with $n := \prod_{i<d} p_i$).

Output: the vector $\mathbf{T} = (\hat{\mathbf{P}}_{[i]_v})_{i<\ell}$.

if $d = 1$ **then return** $\texttt{atomicTFT}(p_0, \ell, \mathbf{P}, \omega)$. ▷ Algorithm 3.1
else if $\ell = n$ **then return** $\texttt{FFT}(\mathbf{v}, \mathbf{P}, \omega)$. ▷ Algorithm 2.1
else
    Choose $h \in \{1, \ldots, d-1\}$.
    Set $n_1 := n^{(\mathrm{L})} = \prod_{i<h} p_i$ and $\mathbf{v}^{(\mathrm{L})} := (p_i)_{i<h}$.
    Set $n_2 := n^{(\mathrm{R})} = \prod_{h\leqslant i} p_i$ and $\mathbf{v}^{(\mathrm{R})} := (p_i)_{h\leqslant i}$.
    Set $m := \lceil \ell/n_2 \rceil$, $q := \ell \operatorname{quo} n_2$ and $r := \ell \operatorname{rem} n_2$.
    for $0 \leqslant i < n_2$ **do** ▷ n_2 TFTs of size n_1
        Set $\boldsymbol{\alpha}^{(i)}_j := P_{i+n_2 j}$ for all $j < n_1$.
        Compute $\boldsymbol{\beta}^{(i)} := \texttt{TFT}(\mathbf{v}^{(\mathrm{L})}, m, \boldsymbol{\alpha}^{(i)}, \omega^{n_2})$.
        Set $(\boldsymbol{\gamma}^{(j)})_i := \omega^{i[j]_{\mathbf{v}^{(\mathrm{L})}}} \times (\boldsymbol{\beta}^{(i)})_j$ for all $j < m$.
    end for
    for $0 \leqslant j < m$ **do** ▷ m TFTs of size n_2
        if $j < q$ **then**
            Set $\boldsymbol{\delta}^{(j)} := \texttt{FFT}(\mathbf{v}^{(\mathrm{R})}, \boldsymbol{\gamma}^{(j)}, \omega^{n_1})$. ▷ Algorithm 2.1
        else
            Compute $\boldsymbol{\delta}^{(j)} := \texttt{TFT}(\mathbf{v}^{(\mathrm{R})}, r, \boldsymbol{\gamma}^{(j)}, \omega^{n_1})$.
        end if
    end for
    return $\mathbf{T} := (\boldsymbol{\delta}^{(0)}, \ldots, \boldsymbol{\delta}^{(m-1)})$.
end if

Theorem 3.1. *Assuming the correctness of Algorithms 2.1 and 3.1, Algorithm 3.2 is correct.*

Proof. Case $\ell = n$ corresponds to a full FFT, then Algorithm 2.1 returns the expected result. If $d = 1$ (typically when n is prime), the Discrete Fourier Transform is computed directly and unnecessary outputs are discarded. The results are ordered as expected because of property (2.5), as in Remark 3.1. In the other cases, the algorithm is called recursively, and its correctness results by induction from the discussion in the previous subsection (section 3.1.2). □

Remark 3.2. As for the in-place Cooley-Tukey FFT presented in [HHL16a], the result depends on the vector $\mathbf{v}$, but not on the choice of h. For this reason, h can be chosen to optimize cache access (typically $h \sim d/2$), as it was done in [Har09] for the radix-2 TFT. This strategy may lead to better performance since it spares frequent data exchanges with the RAM.

3.2 The inverse TFT for mixed radices

The formula (2.2) to invert the usual FFT cannot be used in the case of the TFT because not all values of the transform are known. As for the standard radix-2 TFT [Hoe04], we revert the algorithm computing the TFT instead.

The inversion of a TFT is to be understood as follows: assume that the values $(\mathbf{T}_i)_{0\leqslant i<\ell}$ of the output, and $(\mathbf{P}_i)_{\ell\leqslant i<n}$ of the input are known. Then, the goal is to retrieve the missing values $(\mathbf{P}_i)_{0\leqslant i<\ell}$ of the input. Typically, the values $\mathbf{P}_i$ (for $i \geqslant \ell$) are known to be 0 because of a simple analysis regarding the degree, but the coefficients of highest degree of a polynomial can also be deduced from a limit analysis.

At first, we provide a method to solve the base case of size p by direct computation (here p is not necessarily prime, but it should be reasonably small, so the problem can be solved without further decomposition). Then, we present a recursive algorithm that reduces the TFT inversion to such a base case.

3.2.1 Atomic inverse transforms

In this section, we consider the following *skew butterfly* problem: given the ℓ first values of the transform $\mathbf{T}$, and the $p-\ell$ last coefficients of the vector $\mathbf{P}$ (representing a polynomial P), how can we compute the missing values ? More precisely, given

$$\mathbf{T}^{(1)} := (\hat{\mathbf{P}}_0, \ldots, \hat{\mathbf{P}}_{\ell-1}) \quad \text{and} \quad \mathbf{P}^{(2)} := (\mathbf{P}_\ell, \ldots, \mathbf{P}_{p-1}),$$

we wish to compute

$$\mathbf{T}^{(2)} := (\hat{\mathbf{P}}_\ell, \ldots, \hat{\mathbf{P}}_{p-1}) \quad \text{and} \quad \mathbf{P}^{(1)} := (\mathbf{P}_0, \ldots, \mathbf{P}_{\ell-1}).$$

Recall that the Discrete Fourier Transform is given by the following matrix-vector product:

$$\begin{pmatrix} \hat{\mathbf{P}}_0 \\ \hat{\mathbf{P}}_1 \\ \vdots \\ \hat{\mathbf{P}}_{p-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{p-1} & \cdots & \omega^{(p-1)(p-1)} \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \vdots \\ \mathbf{P}_{p-1} \end{pmatrix},$$

or, in a more compact form,

$$\hat{\mathbf{P}} = \mathbf{F}^{(\omega)} \cdot \mathbf{P}\,.$$

For any $m \leqslant p$, we define the submatrices

$$\begin{aligned} \mathbf{V}^{(\omega,m)} &:= (\omega^{ij})_{0\leqslant i,j<m}\,, \\ \tilde{\mathbf{V}}^{(\omega,m)} &:= (\omega^{(i+m)(j+m)})_{0\leqslant i,j<p-m}\,, \\ \mathbf{W}^{(\omega,m)} &:= (\omega^{i(j+m)})_{i<m\,;\,j<p-m}\,. \end{aligned}$$

Notice that $\mathbf{V}^{(\omega,m)}$ has size $m \times m$, also $\tilde{\mathbf{V}}^{(\omega,m)}$ has size $(p-m) \times (p-m)$ and $\mathbf{W}^{(\omega,m)}$ has size $m \times (p-m)$. Then by definition, we have

$$\mathbf{F}^{(\omega)} = \begin{pmatrix} \mathbf{V}^{(\omega,m)} & \mathbf{W}^{(\omega,m)} \\ (\mathbf{W}^{(\omega,m)})^\top & \tilde{\mathbf{V}}^{(\omega,m)} \end{pmatrix}. \tag{3.1}$$

The considered *skew butterfly* problem is equivalent to the resolution of the following matrix equation with parameters $\mathbf{P}^{(2)}, \mathbf{T}^{(1)}$ and unknowns $\mathbf{P}^{(1)}, \mathbf{T}^{(2)}$:

$$\begin{pmatrix} \mathbf{T}^{(1)} \\ \mathbf{T}^{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{V}^{(\omega,\ell)} & \mathbf{W}^{(\omega,\ell)} \\ (\mathbf{W}^{(\omega,\ell)})^\top & \tilde{\mathbf{V}}^{(\omega,\ell)} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{P}^{(1)} \\ \mathbf{P}^{(2)} \end{pmatrix} \tag{3.2}$$

The matrix $\mathbf{V}^{(\omega,\ell)}$ has determinant $\prod_{0\leqslant i<j<\ell}(\omega^i - \omega^j) \neq 0$ (Vandermonde matrix), hence it is invertible. Therefore,

$$\mathbf{P}^{(1)} = (\mathbf{V}^{(\omega,\ell)})^{-1}(\mathbf{T}^{(1)} - \mathbf{W}^{(\omega,\ell)} \cdot \mathbf{P}^{(2)})\,. \tag{3.3}$$

Once $\mathbf{P}^{(1)}$ is known, it is easy to compute $\mathbf{T}^{(2)}$ as

$$\mathbf{T}^{(2)} = (\mathbf{W}^{(\omega,\ell)})^\top \cdot \mathbf{P}^{(1)} + \tilde{\mathbf{V}}^{(\omega,\ell)} \cdot \mathbf{P}^{(2)}\,. \tag{3.4}$$

Often, it is not necessary to compute $\mathbf{T}^{(2)}$ entirely, but only specific values. In this case, equation (3.4) reduces to a much smaller computation. For the inverse TFT, we are interested only in the computation of $\mathbf{P}^{(1)}$. The computation of $\mathbf{T}^{(2)}$ is only needed to propagate information about $\mathbf{P}^{(2)}$ for the recursive calls. It turns out that returning the first value of $\mathbf{T}^{(2)}$ is actually sufficient for this purpose. For our usage, we assume that the results from this section translate into the following algorithm:

Algorithm 3.3. Inverse Truncated Fourier Transform (base case)

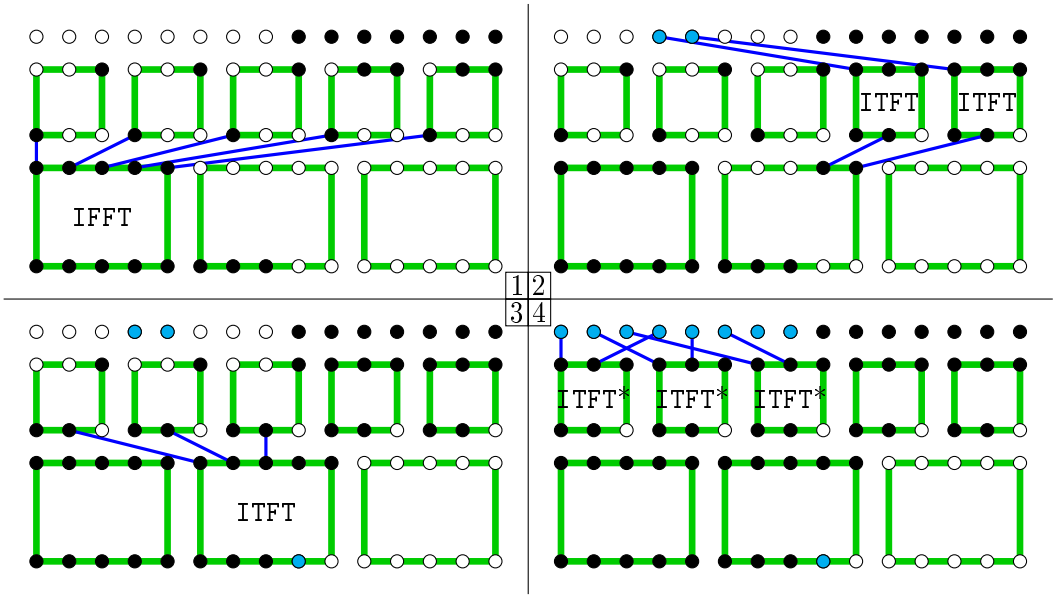
Prototype: `atomicITFT`$(n, \ell, \mathbf{P}^{(2)}, \mathbf{T}^{(1)}, \omega)$

Input: Integers $n \in \mathbb{N}$ and $\ell < n$, vectors $\mathbf{P}^{(2)} \in \mathbb{K}^{n-\ell}$ and $\mathbf{T}^{(1)} \in \mathbb{K}^{\ell}$, and a primitive n-th root of unity ω.

Output: The vector $\mathbf{P}^{(1)}$ and the value $t_\ell = \mathbf{T}_0^{(2)}$, where $\mathbf{P}^{(1)}, \mathbf{T}^{(2)}$ are the solutions of equation (3.2)

3.2.2 Recursive algorithm

In a similar way as in the case $n = 2^k$ [Hoe04], intermediate results are not always computed in an order corresponding to the recursion depth. In a usual FFT, all outer FFTs are inverted, then the inner FFTs are inverted. On the contrary for the TFT, some of the inner TFTs must be inverted first, and these inversions provide additional values that allow the outer TFT to be inverted. Therefore, the algorithm will return some of the missing output values $(\mathbf{T}_i)_{\ell \leqslant i < n}$ in addition to the desired input values $(\mathbf{P}_i)_{0 \leqslant i < \ell}$. It turns out that the outer TFT needs only one value from each inner TFT before it can be inverted, so returning $\mathbf{T}_\ell$ is actually sufficient for the functioning of the recursive algorithm. Algorithm 3.4 describes more precisely this behavior, and the overall idea is illustrated in Figure 3.2.



Black (resp. white) dots represent known (resp. unknown) values at each step; cyan dots represent the output of the algorithm. `ITFT`* does not return additional output values and may be a usual `IFFT`.

Figure 3.2: The four steps of the inverse TFT.

Algorithm 3.4. Inverse Truncated Fourier Transform

Prototype: $\texttt{ITFT}(\mathbf{v}, \ell, \mathbf{P}^{(2)}, \mathbf{T}, \omega)$

Input: A vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, an integer $\ell < n$, vectors $\mathbf{P}^{(2)} = (P_i)_{\ell \leqslant i < n}$ and $\mathbf{T} = (t_i)_{i<\ell}$, and a primitive n-th root of unity ω (with $n := \prod_{i<d} p_i$).

Output: The vector $\mathbf{P}^{(1)} = (P_i)_{i<\ell}$ and the value t_ℓ such that

- $\mathbf{T} = \texttt{TFT}(\mathbf{v}, \ell, \mathbf{P}, \omega)$, with $\mathbf{P} := (P_i)_{i<n}$ and TFT as in Algorithm 3.2,
- $t_\ell = \hat{\mathbf{P}}_{[\ell]_\mathbf{v}}$, again with $\mathbf{P} := (P_i)_{i<n}$.

1: **if** $\ell = 0$ **then return** $\varnothing, \sum_{i=0}^{n-1} P_i$.
2: **else if** $d = 1$ **then return** $\texttt{atomicITFT}(p_0, \ell, \mathbf{P}^{(2)}, \mathbf{T}, \omega)$. ▷ Algorithm 3.3
3: **else**
4: Choose $h \in \{1, \ldots, d-1\}$.
5: Set $n_1 := n^{(\mathrm{L})} = \prod_{i<h} p_i$ and $\mathbf{v}^{(\mathrm{L})} := (p_i)_{i<h}$.
6: Set $n_2 := n^{(\mathrm{R})} = \prod_{h \leqslant i} p_i$ and $\mathbf{v}^{(\mathrm{R})} := (p_i)_{h \leqslant i}$.
7: Set $m := \lceil \ell / n_2 \rceil$, $q := \ell \operatorname{quo} n_2$ and $r := \ell \operatorname{rem} n_2$.
8: **for** $j < q$ **do** ▷ **"Step 1"**
9: Set $\boldsymbol{\delta}_i^{(j)} := \mathbf{T}_{i+n_2 j}$ for all $i < n_1$.
10: Compute $\boldsymbol{\gamma}^{(j)} := \texttt{IFFT}(\mathbf{v}^{(\mathrm{R})}, \boldsymbol{\delta}^{(j)}, \omega^{n_1})$. ▷ Algorithm 2.2
11: Set $\boldsymbol{\beta}_j^{(i,1)} := \omega^{-i[j]_\mathbf{v}^{(\mathrm{L})}} \times \boldsymbol{\gamma}_i^{(j)}$ for all $i < n_1$.
12: **end for**
13: **for** $r \leqslant i < n_2$ **do** ▷ **"Step 2"**
14: Set $\boldsymbol{\alpha}_j^{(i,2)} := P_{i+n_2 j}$ for all $j \geqslant q$. ▷ $i + n_2 j \geqslant \ell$
15: Compute $(\boldsymbol{\alpha}^{(i,1)}, \beta_{i,q}) := \texttt{ITFT}(\mathbf{v}^{(\mathrm{L})}, q, \boldsymbol{\alpha}^{(i,2)}, \boldsymbol{\beta}^{(i,1)}, \omega^{n_2})$. ▷ $q < n_1$
16: Set $\boldsymbol{\gamma}_i^{(q)} := \omega^{i[q]_{\mathbf{v}^{(\mathrm{R})}}} \times \beta_{i,q}$.
17: Set $\mathbf{P}_{i+n_2 j}^{(1)} := \boldsymbol{\alpha}_j^{(i,1)}$ for all $j < q$.
18: **end for**
19: Set $\boldsymbol{\delta}^{(q,1)} := (\mathbf{T}_{i+n_2 q})_{i<r}$. ▷ **"Step 3"**
20: Set $\boldsymbol{\gamma}^{(q,2)} := (\boldsymbol{\gamma}_i^{(q)})_{r \leqslant i < n_2}$.
21: Compute $(\boldsymbol{\gamma}^{(q,1)}, \delta_{q,r}) := \texttt{ITFT}(\mathbf{v}^{(\mathrm{R})}, r, \boldsymbol{\gamma}^{(q,2)}, \boldsymbol{\delta}^{(q,1)}, \omega^{n_1})$.
22: **for** $i < r$ **do** ▷ **"Step 4"**
23: Set $\boldsymbol{\beta}_q^{(i,1)} := \omega^{-i[q]_{w_1}} \times \boldsymbol{\gamma}_i^{(q,1)}$.
24: **if** $q + 1 < n_1$ **then**
25: Set $\boldsymbol{\alpha}_j^{(i,2)} := P_{i+n_2 j}$ for all $j > q$. ▷ $i + n_2 j \geqslant \ell$
26: Compute $(\boldsymbol{\alpha}^{(i,1)}, \text{dummy}) := \texttt{ITFT}(\mathbf{v}^{(\mathrm{L})}, q+1, \boldsymbol{\alpha}^{(i,2)}, \boldsymbol{\beta}^{(i,1)}, \omega^{n_2})$.
27: **else**
28: Compute $\boldsymbol{\alpha}^{(i,1)} := \texttt{IFFT}(\mathbf{v}^{(\mathrm{L})}, \boldsymbol{\beta}^{(i,1)}, \omega^{n_2})$. ▷ Algorithm 2.2
29: **end if**
30: Set $\mathbf{P}_{i+n_2 j}^{(1)} := \boldsymbol{\alpha}_j^{(i,1)}$ for all $j \leqslant q$.
31: **end for**
32: **return** $(\mathbf{P}^{(1)}, \delta_{q,r})$.
33: **end if**

Theorem 3.2. *Assuming the correctness of Algorithms 2.2 and 3.3, Algorithm 3.4 is correct.*

Proof. We show that equations (2.7-2.11) are satisfied, by induction over d. The case $\ell = 0$ is clearly correct. For $d = 1$, the result is computed directly using Algorithm 3.3, which is supposed to be correct. As for Algorithm 3.2, the result is ordered as expected (Remark 3.1).

In *Step 1*, the $\boldsymbol{\gamma}^{(j)}$ are computed for all $j < q$ using a full reverse FFT. This means $\boldsymbol{\gamma}^{(j)}$ and $\boldsymbol{\delta}^{(j)}$ verify equation (2.10) for $j < q$. Then, the first part of every vector $\boldsymbol{\beta}^{(i)}$ is computed according to equation (2.9).

At this point, the vectors $\boldsymbol{\alpha}^{(i)}$ and $\boldsymbol{\beta}^{(i)}$ are partially known. More precisely, we know the values $\boldsymbol{\alpha}_j^{(i)}$ with $j > q$, and the values $\boldsymbol{\beta}_j^{(i)}$ with $j < q$. Moreover, $\boldsymbol{\alpha}_q^{(i)}$ is known for $i \geqslant r$ (by definition, $\ell = n_2\, q + r$). In *Step 2*, a recursive call computes the missing part of $\boldsymbol{\alpha}^{(i)}$ for these i, as well as the value $\beta_{i,q} = \boldsymbol{\beta}_q^{(i)}$. This means $\boldsymbol{\alpha}^{(i)}$ and $(\boldsymbol{\beta}_j^{(i)})_{j \leqslant q}$ verify equation (2.8) for all $i \geqslant r$ (the recursive call is correct by the induction hypothesis).

At the end of *Step 2*, the second part of $\boldsymbol{\gamma}^{(q,2)} := (\boldsymbol{\gamma}_i^{(q)})_{i \geqslant r}$ is computed according to equation (2.9). The first part $\boldsymbol{\delta}^{(q,1)} := (\boldsymbol{\delta}_i^{(q)})_{i<r}$ is also known from the input ($\mathbf{T}$). In *Step 3*, the missing part $\boldsymbol{\gamma}^{(q,1)} := (\boldsymbol{\gamma}_i^{(q)})_{i<r}$ of $\boldsymbol{\gamma}^{(q)}$ is computed, as well as $\boldsymbol{\delta}_r^{(q)}$, through a recursive call. Then, $\boldsymbol{\gamma}^{(q)}$ and $(\boldsymbol{\delta}_i^{(q)})_{i \leqslant r}$ verify equation (2.10).

Finally in *Step 4*, the values $\boldsymbol{\beta}_q^{(i)}$ are computed for $i < r$. The values $(\boldsymbol{\alpha}_j^{(i)})_{j>q}$ being given on input, and the $(\boldsymbol{\beta}_j^{(i)})_{j<q}$ being known from *Step 1*, the missing $(\boldsymbol{\alpha}_j^{(i)})_{j \leqslant q})$ can be computed using a recursive call. Since this call is correct by the induction hypothesis, $\boldsymbol{\alpha}^{(i)}$ and $(\boldsymbol{\beta}_j^{(i)})_{j \leqslant q}$ verify equation (2.8) for $i < r$ (hence for all i because of *Step 2*.

All in all, the vectors (some truncated) $\boldsymbol{\alpha}^{(i)}$, $(\boldsymbol{\beta}_j^{(i)})_{j \leqslant q}$, $\boldsymbol{\gamma}^{(j)}$ (for $j \leqslant q$), $\boldsymbol{\delta}^{(j)}$ (for $j < q$) and $(\boldsymbol{\delta}_i^{(q)})_{i \leqslant r}$ verify equations (2.7-2.11). This is sufficient to prove correctness as seen in section 3.1 (where ℓ is replaced by $\ell' = \ell + 1$) □

3.2.3 Practical remarks

Remark 3.3. Algorithm 3.4 can be used to compute the unique polynomial P of degree less than n such that ℓ evaluation points are given by the vector $\mathbf{T}$ and the coefficients of degree at least ℓ are given by $\mathbf{P}^{(2)}$:

$$\forall i < \ell, P(\omega^{[i]_v}) = \mathbf{T}_i \quad \text{and} \quad \mathbf{P}^{(2)} = (P_i)_{\ell \leqslant i < n}\,.$$

In particular, it can be used to interpolate a polynomial of degree less than ℓ by setting $\mathbf{P}^{(2)} = (0, \ldots, 0)$.

Remark 3.4. In section 3.2.1, the order p may be composite (which can happen if an element of the vector $\mathbf{v}$ from Algorithm 3.4 is composite), but the resolution of equation (3.3) is not efficient if p is large. Algorithm 3.4 shows that the problem can be reduced to smaller sizes as long as its order is composite. However, it seems difficult to solve a skew butterfly problem if its size is a large prime.

For example, in $\mathbb{F}_{2^{60}}$, we have primitive roots of unity of order

$$2^{60} - 1 = 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321\,.$$

It is feasible to perform the inversion using linear algebra for size up to 13 efficiently, but direct computation may become too costly for $p = 331$ or 1321 for example.

Remark 3.5. An inverse TFT of length 0 is actually very simple: it reduces to the computation of t_0. For this reason, if $(\ell \operatorname{rem} n_2) = 0$, then the recursive call in Algorithm 3.4 at Step 3 becomes trivial. This partially solves the problem mentioned in the previous remark: assume the prime factors of n are sorted in the vector $\mathbf{v}$ (in increasing order). If for example an inversion by direct computation is possible for $p_0, \ldots, p_{k-1}$ but not for $p_k, \ldots, p_{d-1}$ (because these primes are too large), then a TFT of length ℓ can still be reverted if $\tilde{n} = p_k \times \cdots \times p_{d-1}$ divides ℓ.

Remark 3.6. It is important that the recursive calls in *Step 2* return the additional output value $\beta_{i,q} = \boldsymbol{\beta}_q^{(i)}$. However, it is not necessary that Algorithm 3.4 always returns the additional output value t_ℓ. For example, this value is simply discarded in the recursive calls from *Step 4*. Another typical case where this value is not needed is for the interpolation a polynomial of degree less than ℓ from the ℓ values $(\hat{\mathbf{P}}_{[i]_v})_{i<\ell}$. It is possible to adapt Algorithm 3.4 to avoid this unnecessary computation when the value t_ℓ is not needed.

3.2.4 A remarkable duality for atomic inverse transforms

Direct resolution of the *skew butterfly* problem from section 3.2.1 requires the inversion of the matrix $\mathbf{V}^{(\omega,\ell)}$ of size $\ell \times \ell$, which becomes expensive if ℓ is large. In this section, we present a dual problem that can be solved through the inversion of matrix $\mathbf{V}^{(\omega^{-1},p-\ell)}$, that has size $(p-\ell)\times(p-\ell)$. This duality ensures that the *skew butterfly* problem can always be solved through the inversion of a matrix $\mathbf{V}^{(\phi,m)}$ (and a few matrix-vector products), where $m \leqslant p/2$ and ϕ is either ω or ω^{-1}. In the upcoming complexity analysis, we will then be able to assume that $\ell \leqslant p/2$.

Property (2.2) gives the inverse matrix $(\mathbf{F}^{(\omega)})^{-1} = (1/p)\cdot\mathbf{F}^{(\omega^{-1})}$. Then, with a decomposition of $(\mathbf{F}^{(\omega)})^{-1}$ as in formula (3.1), equation (3.2) can be rewritten as follows:

$$\begin{pmatrix} \mathbf{P}^{(1)} \\ \mathbf{P}^{(2)} \end{pmatrix} = \frac{1}{p} \cdot \begin{pmatrix} \mathbf{V}^{(\omega^{-1},\ell)} & \mathbf{W}^{(\omega^{-1},\ell)} \\ (\mathbf{W}^{(\omega^{-1},\ell)})^\top & \tilde{\mathbf{V}}^{(\omega^{-1},\ell)} \end{pmatrix} \begin{pmatrix} \mathbf{T}^{(1)} \\ \mathbf{T}^{(2)} \end{pmatrix}.$$

We want to solve the equation above with parameters $\mathbf{P}^{(2)}, \mathbf{T}^{(1)}$ and unknowns $\mathbf{P}^{(1)}, \mathbf{T}^{(2)}$. We have

$$\mathbf{T}^{(2)} = (\tilde{\mathbf{V}}^{(\omega^{-1},\ell)})^{-1} \cdot (p\mathbf{P}^{(2)} - (\mathbf{W}^{(\omega^{-1},\ell)})^\top \mathbf{T}^{(1)})\,. \tag{3.5}$$

Once $\mathbf{T}^{(2)}$ is known, it is easy to compute $\mathbf{P}^{(1)}$ since

$$\mathbf{P}^{(1)} = \frac{1}{p}(\mathbf{V}^{(\omega^{-1},\ell)}\mathbf{T}^{(1)} + \mathbf{W}^{(\omega^{-1},\ell)}\mathbf{T}^{(2)})\,. \tag{3.6}$$

Except for the factor $1/p$, these formulas are symmetric to the formulas (3.3) and (3.4). This shows the duality between these two problems.

Proposition 3.3. *The dual problem with parameter ℓ is equivalent to the direct problem from section 3.1.1 with parameter $p-\ell$, up to $O(p)$ additional field operations.*

Proof. The matrix $\mathbf{V}^{(\omega^{-1})}$ is obtained from $\mathbf{V}^{(\omega)}$ by a simple row (or column) permutation, so no field operation is needed to compute the different sub-matrices. To reduce notations, let $\phi = \omega^{-1}$.

The main difference between the two problems is therefore that the dual problem involves the inversion of $\tilde{\mathbf{V}}^{(\phi,\ell)}$ in equation (3.5), instead of $\mathbf{V}^{(\omega,\ell)}$ as in equation (3.3). Here $\tilde{\mathbf{V}}^{(\phi,\ell)}$ has size $(p-\ell)\times(p-\ell)$ and $\mathbf{V}^{(\omega,\ell)}$ has size $\ell\times\ell$. It appears however that the inversion of $\tilde{\mathbf{V}}^{(\phi,\ell)}$ is not harder than the inversion of $\mathbf{V}^{(\phi,p-\ell)}$. Indeed, we have the following property:

$$(\tilde{\mathbf{V}}^{(\phi,\ell)})_{i,j} = \phi^{(\ell+i)(\ell+j)} = \phi^{\ell(\ell+j)} \cdot \phi^{ij} \cdot \phi^{\ell i} \quad \text{for all } i,j\,,$$

hence,

$$\tilde{\mathbf{V}}^{(\phi,\ell)} = \mathbf{D}^{(\phi,\ell)} \cdot \mathbf{V}^{(\phi,p-\ell)} \cdot \tilde{\mathbf{D}}^{(\phi,\ell)}\,, \tag{3.7}$$

where $\mathbf{D}^{(\phi,\ell)}$ and $\tilde{\mathbf{D}}^{(\phi,\ell)}$ are diagonal matrices of size $(p-\ell)\times(p-\ell)$. □

3.3 Complexity analysis

This section aims to evaluate the field operation count for a TFT of size n and length $\ell \leqslant n$, and to compare it with the cost for a full FFT of size n. In the following, we note these costs $\mathsf{T}(\ell,n)$ and $\mathsf{F}(n)$ respectively. Asymptotic bounds involve the field operation count for common arithmetic operations on polynomials of degree n: $\mathsf{M}(n)$ for the multiplication and $\mathsf{C}(n)$ for the cyclic convolution (multiplication modulo X^n-1).

Let $n = p_0 \cdots p_{d-1}$ be the size of the Discrete Fourier Transform (FFT or TFT) that is considered. If we develop completely the recursive calls of the FFT algorithm as in section 2.2, the execution decomposes into d successive transformations of a vector of length n (see Figure 3.3). At each row, the working vector is transformed using n/p_i independent DFTs of size p_i, which are computed directly.

3.3.1 Complexity of a full FFT

In a full FFT, all n/p_i atomic DFTs are computed at each row. If we note $\mathsf{f}(p) := \mathsf{F}(p)/p$ the normalized operation count (per intermediate value), then each of the atomic DFTs has an operation count of $p_i\mathsf{f}(p_i)$. Summing for all i, we get the following result:

Theorem 3.4 (Complexity of the usual Cooley-Tukey FFT)**.** *We have*

$$\mathsf{F}(n) = n\cdot(\mathsf{f}(p_0)+\mathsf{f}(p_1)+\cdots+\mathsf{f}(p_{d-1})) + O(nd)\,. \tag{3.8}$$

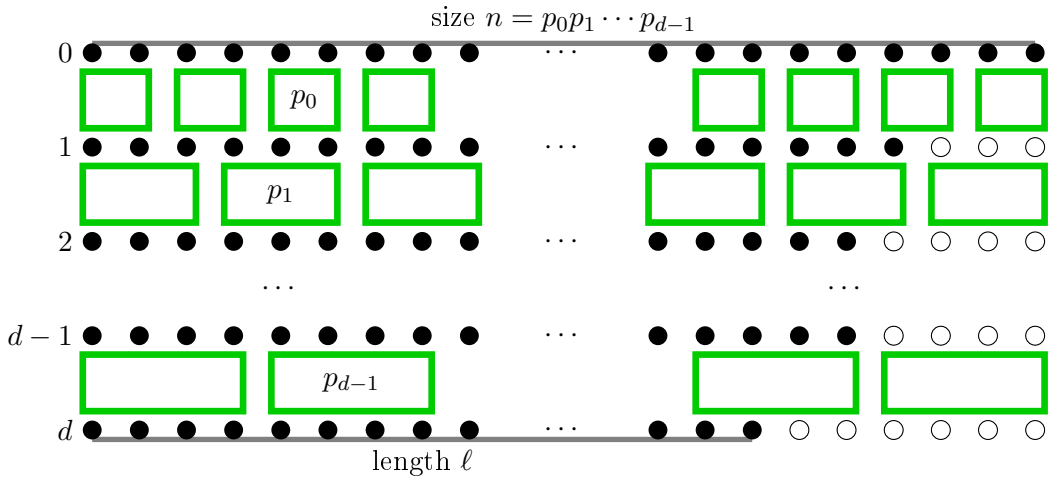


Figure 3.3: The TFT algorithm with unfolded recursive calls

Remark 3.7. The term $O(nd)$ is the cost for the operations between rows of atomic DFTs, that is the multiplications by twiddle factors. The multiplicative constant is actually small; in fact, with a precomputed table of twiddle factors, this represents $n(d-1)$ multiplications, and even less if we take into account that some of the twiddle factors are equal to 1.

Remark 3.8. For small p, it is most efficient to compute the atomic DFTs using specialized codelets that perform naive matrix-vector products. This yields $\mathsf{F}(p) \sim p^2$, or $\mathsf{f}(p) \sim p$. For larger p, methods like Rader's or Bluestein's algorithms are more efficient. In this case, $\mathsf{F}(p) \sim p \log p$, or $\mathsf{f}(p) \sim \log p$ (with a larger constant factor than for the naive method).

3.3.2 Complexity of atomic TFTs

Recall the notations F and T for the costs of the FFT and TFT respectively (while f and t represent the corresponding normalized costs). As discussed in section 3.1.1, there are two simple methods to compute an atomic TFT:

- for very small ℓ, one can naively compute the ℓ first values of the Fourier transform using Horner's rule (p additions and p multiplications for each evaluation point). This method has a cost of $\mathsf{T}(\ell, p) = 2\ell p$.
- for ℓ near p, it is interesting to compute the full atomic DFT and discard the last $p - \ell$ values, at a cost of $\mathsf{T}(\ell, p) = \mathsf{F}(p)$.

For intermediate values of ℓ, notice that the TFT points form a geometric progression. Then, a polynomial of degree less than p can be evaluated on these ℓ points using $p/\ell\,\mathsf{M}(\ell) + O(p)$ field operations. This is a consequence of the following result, already mentioned in Section 2.3.2:

Lemma 3.5 (Bostan, Schost [BS05, Section 5]). *Let $(1, \omega, \dots, \omega^{\ell-1})$ be a geometric progression such that the points ω^i are pairwise distinct. Then, a polynomial of degree less than ℓ can be evaluated on these points in $\mathsf{M}(\ell) + O(\ell)$ field operations.*

As in the previous subsection, we normalize the operation count per intermediate value: $\mathsf{t}(\ell, p) = \mathsf{T}(\ell, p)/\ell$. We introduce the overhead for the atomic TFT as $\mathsf{k}(\ell, p) := \mathsf{t}(\ell, p)/\mathsf{f}(p)$. Since $\mathsf{k}(\ell, p)$ decreases with ℓ, it is meaningful to also introduce

$$\mathsf{K}(p) := \sup_{\ell \leqslant p} \ell \cdot \mathsf{k}(\ell, p)\,.$$

By definition, we have:

$$\begin{aligned} \mathsf{t}(\ell, p) &= \mathsf{k}(\ell, p)\mathsf{f}(p)\,, \\ \mathsf{T}(\ell, p) = \ell \cdot \mathsf{t}(\ell, p) &\leqslant \mathsf{K}(p)\mathsf{f}(p)\,. \end{aligned} \tag{3.9}$$

Remark 3.9. Consider to simplify that all evaluations are equally hard (though evaluation at 1 is slightly easier). We get $\mathsf{F}(p) \leqslant p/\ell\,\mathsf{T}(\ell, p)$, so that $\mathsf{k}(\ell, p) \geqslant 1$. On the other hand, a TFT is immediately obtained from a full FFT, which gives $\mathsf{T}(\ell, p) \leqslant \mathsf{F}(p)$ hence $\mathsf{K}(p) \leqslant p$. The extreme case $\ell = p$ gives actually $\mathsf{K}(p) = p$. However, we keep the notation $\mathsf{K}(p)$ to respect the symmetry with the inverse TFT.

3.3.3 Complexity of atomic inverse TFTs

Similarly, we introduce the corresponding costs for the inverse TFT: let $\mathsf{T}^*(\ell, p)$ be the cost for the inverse TFT, and $\mathsf{t}^*(\ell, p) := \mathsf{T}^*(\ell, p)/\ell$ the normalized cost. The corresponding overhead is $\mathsf{k}^*(\ell, p) := \mathsf{t}^*(\ell, p)/\mathsf{f}(p)$ and $\mathsf{K}^*(p) := \sup_{\ell \leqslant p} \ell \cdot \mathsf{k}^*(\ell, p)$.

Without loss of generality, we consider only the direct resolution from section 3.2.1 and we assume $\ell \leqslant p/2$. Indeed, if $\ell > p/2$, then we can reduce to the dual problem, which causes only $O(p)$ additional operations because of Proposition 3.3.

We first need to evaluate the cost of the matrix-vector products:

Lemma 3.6. *The products $\mathbf{W}^{(\omega,\ell)} \cdot \mathbf{P}^{(2)}$ and $(\mathbf{W}^{(\omega,\ell)})^\top \cdot \mathbf{P}^{(1)} + \tilde{\mathbf{V}}^{(\omega,\ell)} \cdot \mathbf{P}^{(1)}$ can each be done in $\mathsf{F}(p)$ field operations.*

Proof. Computing the function $\mathbf{X} \mapsto \mathbf{F}^{(\omega)} \cdot \mathbf{X}$ correspond to a FFT of size p, which needs by definition $\mathsf{F}(p)$ field operations. Notice that the desired products are respectively the upper and lower parts of

$$\mathbf{F}^{(\omega)} \cdot \begin{pmatrix} 0 \\ \mathbf{P}^{(2)} \end{pmatrix} \quad \text{and} \quad \mathbf{F}^{(\omega)} \cdot \begin{pmatrix} \mathbf{P}^{(1)} \\ \mathbf{P}^{(2)} \end{pmatrix}. \qquad \square$$

Let us now examine the cost of the inversion of the matrix $\mathbf{V}^{(\omega,\ell)}$. Actually, it is not necessary to compute $(\mathbf{V}^{(\omega,\ell)})^{-1}$; it is sufficient to compute the function $\mathbf{Y} \mapsto (\mathbf{V}^{(\omega,\ell)})^{-1}\mathbf{Y}$, which is a polynomial interpolation on the points $1, \omega, \dots, \omega^{\ell-1}$. Recall the following upper bound mentioned in section 2.3.2:

Lemma 3.7 (Bostan, Schost [BS05, Section 5]). *Let $(1, \omega, \dots, \omega^{\ell-1})$ be a geometric progression such that the points ω^i are pairwise distinct. Then, the interpolation of a polynomial of degree less than ℓ on these points can be performed in $2\mathsf{M}(\ell) + O(\ell)$ operations.*

We are now able to bound the cost of an atomic inverse TFT:

Proposition 3.8. *We have $\mathsf{T}^*(\ell, p) \leqslant 4\mathsf{F}(p) + O(p)$ for large p. Hence,*

$$\mathsf{K}^*(p) \leqslant 4p(1 + o(1)) .$$

Proof. At first, we compute $\mathbf{Y} := \mathbf{T}^{(1)} - \mathbf{W}^{(\omega,\ell)}\mathbf{P}^{(2)}$, which costs $F(p) + O(p)$ by Lemma 3.6.

Then, the computation of $\mathbf{P}^{(1)} = (\mathbf{V}^{(\omega,\ell)})^{-1}\mathbf{Y}$ can be done using $2\mathsf{M}(\ell) + O(\ell)$ base field operations by Lemma 3.7. This is not more than $2\mathsf{F}(p) + O(p)$ for large p. Indeed, since $\ell \leqslant p/2$, a multiplication of size ℓ can be seen as a cyclic convolution of size p, that is $\mathsf{M}(\ell) \leqslant \mathsf{C}(p)$. Moreover, Bluestein's transform [Blu70] is an efficient method to compute the DFT for a large p, which gives $\mathsf{F}(p) = \mathsf{C}(p) + O(p)$ and in particular $\mathsf{C}(p) \leqslant \mathsf{F}(p)$ asymptotically.

Finally, $\mathbf{T}^{(2)} = (\mathbf{W}^{(\omega,\ell)})^\top \cdot \mathbf{P}^{(1)} + \tilde{\mathbf{V}}^{(\omega,\ell)} \cdot \mathbf{P}^{(2)}$ can be computed in $\mathsf{F}(p)$ operations because of Lemma 3.6. □

3.3.4 Complexity of the TFT and its inverse

When computing row $i+1$ from row i in the TFT, some of the atomic DFTs are not performed, most of the remaining are full DFTs and only a few are TFTs. More precisely, we can isolate the $i \to i+1$ transform as follows: split the TFT as in section 3.1.2, first at $h = i+1$ ($n_1 = p_0 \cdots p_i$), then each of the n_2 inner TFTs is split at $h' = i$ ($n'_1 = p_0 \cdots p_{i-1}$). This shows that the $i \to i+1$ transform consists in $\pi_i \lfloor \ell'/p_i \rfloor$ atomic (full) DFTs and π_i atomic TFTs, where $\pi_i = p_{i+1} \cdots p_{d-1} (= n_2)$ and $\ell' = \lceil \ell/\pi_i \rceil$. The atomic TFTs have length $r_i = (\ell' \operatorname{rem} p_i)$.

As a consequence, the complexity of the complete TFT is given by

$$\mathsf{T}(\ell, n) = \sum_{i=0}^{d-1} \left(\pi_i \left\lfloor \frac{\lceil \ell/\pi_i \rceil}{p_i} \right\rfloor p_i \mathsf{f}(p_i) + \pi_i r_i \mathsf{t}(r_i, p_i) + O(\pi_i \lceil \ell/\pi_i \rceil) \right) ,$$

where f and t represent the corresponding normalized costs of the TFT and FFT as before. Recall also the notations $\mathsf{k}(r, p) := \mathsf{t}(t, p)/\mathsf{f}(p)$ and $\mathsf{K}(p) := \sup r\mathsf{k}(r, p)$ for the overhead of an atomic TFT. Since $\lfloor \ell'/p_i \rfloor p_i + r_i = \ell'$ (Euclidean division), the above formula rewrites as

$$\begin{aligned} \mathsf{T}(\ell, n) \leqslant & \sum_{i=0}^{d-1} \left(\pi_i \left\lceil \frac{\ell}{\pi_i} \right\rceil \mathsf{f}(p_i) + O(\pi_i \lceil \ell/\pi_i \rceil) \right) \\ & + \sum_{i=0}^{d-1} \pi_i r_i (\mathsf{k}(r_i, p_i) - 1) \mathsf{f}(p_i) . \end{aligned}$$

We have $\lceil \ell/\pi_i \rceil \leqslant (\ell/\pi_i) + 1$. Moreover, the term $O(\pi_i \lceil \ell/\pi_i \rceil) = O(\ell) + O(\pi_i)$ corresponds to the multiplication by twiddle factors at each step, as in Theorem 3.4. We then have the (otherwise abusive) simplification

$$\sum_{i=0}^{d-1} \big(\ell\, \mathsf{f}(p_i) + O(\ell)\big) = \frac{\ell}{n}\mathsf{F}(n)\,.$$

This yields

$$\mathsf{T}(\ell, n) \leqslant \frac{\ell}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1}(\pi_i \mathsf{f}(p_i) + O(\pi_i)) + \sum_{i=0}^{d-1} \pi_i r_i (\mathsf{k}(r_i, p_i) - 1)\mathsf{f}(p_i)\,.$$

By definition, we have $r_i \mathsf{k}(r_i, p_i) \leqslant \mathsf{K}(p_i)$, and it is clear that $\mathsf{K}(p_i) \geqslant \mathsf{k}(1, p_i) \geqslant 1$. Then, handling the cases $r_i = 0$ and $1 \leqslant r_i \leqslant p_i - 1$ separately shows easily that $r_i(\mathsf{k}(r_i, p_i) - 1) + 1 \leqslant \mathsf{K}(p_i)$. Finally, using equation (3.8), we get the bound:

Theorem 3.9 (Complexity of the TFT)**.** *The Truncated Fourier Transform can be performed using*

$$\mathsf{T}(\ell, n) \leqslant \frac{\ell}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1} (p_{i+1} \cdots p_{d-1}) \Big(\mathsf{K}(p_i)\mathsf{f}(p_i) + O(1)\Big) \tag{3.10}$$

field operations.

For the inverse TFT, all atomic inverse TFTs (at a given row of atomic transform) do not necessarily have the same length. However the above reasoning still applies, and we get a similar bound (though with a larger overhead $\mathsf{K}^*(p) \geqslant \mathsf{K}(p)$ than for the direct transform):

Theorem 3.10 (Complexity of the Inverse TFT)**.** *The Inverse Truncated Fourier Transform can be performed using*

$$\mathsf{T}^*(\ell, n) \leqslant \frac{\ell}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1} (p_{i+1} \cdots p_{d-1}) \Big(\mathsf{K}^*(p_i)\mathsf{f}(p_i) + O(1)\Big) \tag{3.11}$$

field operations.

Remark 3.10. The bound (3.10) can be rewritten as

$$\mathsf{T}(\ell, n) \leqslant \frac{\ell}{n}\mathsf{F}(n) + n \cdot \left(\sum_{i=0}^{d-1} \frac{\mathsf{K}(p_i)\mathsf{f}(p_i)}{p_0 \cdots p_i}\right) + O(n)\,.$$

Since $\mathsf{K}(p)\mathsf{f}(p)$ increases with p, this shows that it is best to sort the prime factors of n in increasing order ($p_0 \leqslant p_1 \leqslant \cdots \leqslant p_{d-1}$) to minimize the overhead. Moreover, assuming n is highly composite, $p_0 \cdots p_i$ grows much faster than $\mathsf{K}(p_i)\mathsf{f}(p_i)$, so that the predominant term in the linear factor is $\mathsf{K}(p_0)\mathsf{f}(p_0)/p_0$. In the simple case where all p_i are equal, the bound can be simplified as

$$\mathsf{T}(\ell, n) \leqslant \frac{\ell}{n}\mathsf{F}(n) + \frac{n\mathsf{K}(p)\mathsf{f}(p)}{p-1} + O(n)\,.$$

Implementation issues

The original algorithm [Hoe04] for the radix-2 TFT is well understood and actually implemented in practice. For example, software like FLINT [Har10], MATHEMAGIX [HLM+02] and NTL [Sho01] use the truncated Fourier transform in their polynomial arithmetic over $\mathbb{Z}$ and $\mathbb{F}_p$ ($p > 2$), achieving a noticeable gain with respect to plain FFT-multiplication. However, it seems unlikely to observe a similar improvement in the general case with mixed-radix TFT.

Indeed, the complex structure of atomic transforms would often lead to low-level code that is less efficient than the highly optimized codelets used in plain FFT. This is particularly true for the skew butterfly problem of section 3.2.1, that involves nontrivial linear algebra. Additionally, we remark that the complexity analysis in sections 3.3.2-3.3.3 relies on artificial reductions to other arithmetic problems, which is another illustration of the difficulty of these low-level tasks. For this reason, there was little interest in trying to implement a generic version of the algorithm from the present chapter: it would certainly not be competitive with traditional fast Fourier transform.

Nevertheless, it is possible that a partial implementation with a specific application in mind could lead to practical improvements in certain situations. It was initially planned to implement the TFT in $\mathbb{F}_{2^{60}}$ (motivated by the paper [HHL16a]), to get a smoother behavior for the multiplication in $\mathbb{F}_2[X]$. This idea was abandoned because the *Frobenius FFT* presented in the next chapter was more promising.

Chapter 4

The Frobenius FFT

Contents

Note. *The results in this chapter were published in* [HL17].

Consider a FFT-based multiplication algorithm over a finite field. If the field is small but the polynomials have large degree, then there are simply not enough points to use the evaluation-interpolation strategy. As mentioned earlier, one solution is to embed the base field into a suitable extension; however computations in the larger field are more expensive. The goal of this chapter is to show that this induced overhead can be compensated using the symmetries provided by the Frobenius map.

To help with the intuition, let us start by an analogy with the complex numbers. Let $n = 2^\ell$ and let $\omega = \exp(2i\pi/n)$ be a primitive n-th root of unity in $\mathbb{C}$. The traditional algorithm for computing discrete Fourier transforms [CT65] takes a polynomial $P \in \mathbb{C}[X]$ of degree $< n$ on input, and returns the vector $(P(\omega^0), \ldots, P(\omega^{n-1}))$. If P actually admits real coefficients, then we have $P(\omega^{-i}) = \overline{P(\omega^i)}$ for all i, so roughly $n/2$ of the output values are superfluous. Because of this symmetry, it is well known that such "real FFTs" can be computed roughly twice as fast as their complex counterparts [Ber68, SJHB87].

More generally, there exists an abundant literature on the computation of FFTs of polynomials with various types of symmetry. *Crystallographic* FFT algorithms

date back to [Ten73], with contributions as recent as [KRO07], but are dedicated to crystallographic symmetries. So called *lattice FFTs* have been studied in a series of papers initiated by [GM86] and continued in [VZ07, Ber13]. A more general framework due to [AJJ96] was recently revisited from the point of view of high-performance code generation [JX07]. Further symmetric FFT algorithms and their truncated versions were developed in [HLS13].

In this chapter, we focus on discrete Fourier transforms of polynomials $P \in \mathbb{F}_q[X]$ with coefficients in the finite field with q elements. In general, primitive n-th roots of unity ω only exist in extension fields of $\mathbb{F}_q$, say $\omega \in \mathbb{F}_{q^d}$. This puts us in a similar situation as in the case of real FFTs: our primitive root of unity ω lies in an larger field than the coefficients of the polynomial. This time, the degree of the extension is $d = [\mathbb{F}_{q^d} : \mathbb{F}_q]$ instead of $2 = [\mathbb{C} : \mathbb{R}]$. As in the case of real FFTs, it is natural to search for algorithms that allow us to compute the DFT of a polynomial in $\mathbb{F}_q[X]$ approximately d times faster than the DFT of a polynomial in $\mathbb{F}_{q^d}[X]$. The idea is to use the Frobenius automorphism

$$\begin{aligned} \phi_q : \mathbb{F}_{q^d} &\to \mathbb{F}_{q^d} \\ x &\mapsto x^q \end{aligned}$$

as the analogue of complex conjugation.

If $P \in \mathbb{F}_q[X]$, then we will exploit the fact that $P(\phi_q^k(\omega^i)) = \phi_q^k(P(\omega^i))$ for all $0 \leqslant i < n$ and $0 \leqslant k < d$. The set of evaluation points (that is the cyclic group $\langle\omega\rangle$ generated by ω) can be partitioned into roughly n/d orbits for the action of the Frobenius map. Because of the above fact, it suffices to keep one element in each of these orbits, and still have all the information. Moreover, as for DFTs over $\mathbb{R}$, the operation is invertible directly, without computing the "missing" values. We will design an efficient algorithm to do this, called the *Frobenius FFT* (section 4.3).

Alternatively, it is possible to reduce the computation of d discrete Fourier transforms of polynomials $P^{(0)}, \ldots, P^{(d-1)} \in \mathbb{F}_q[X]^{<n}$ to a single discrete Fourier transform of a polynomial $\tilde{P} \in \mathbb{F}_{q^d}[X]^{<n}$. This technique is useful if many transforms have to be computed, and it is detailed in section 4.4.

4.1 Finite field arithmetic

In this section, we first recall some basic operations in finite fields and the associated complexity functions. Throughout this chapter, we need to work with polynomials over various fields: a base field $\mathbb{F}_q$ and several extensions $\mathbb{F}_{q^d}$, $\mathbb{F}_{q^e}, \ldots$ Unlike the other chapters that use the algebraic complexity model, we assume in this chapter the *Turing complexity model* with a finite number of tapes [Pap94]. We do this to ensure a common framework in the complexity analysis and to take into account that operations in $\mathbb{F}_{q^d}$ are more expensive than in $\mathbb{F}_q$ (counting operations in $\mathbb{F}_q$ is also possible, but more technical). Recall our notation M_q for the complexity of polynomial multiplication over $\mathbb{F}_q$, and similarly for other operations.

Remark 4.1. Bounds in the algebraic complexity model generally remain valid with Turing machines, up to the size of the coefficient that becomes explicit. For

example, the currently best known bound [HH19e] for polynomial multiplication is

$$\mathsf{M}_q = O(n \log q \log(n \log q))$$

in the Turing complexity model (under certain assumptions on the distribution of prime numbers). The counterpart in the algebraic complexity model is simply $O(n \log n)$. The Turing complexity model requires however a careful analysis to achieve a bound that is uniform in n and q.

Using the representation $\mathbb{F}_q \cong \mathbb{F}_p[Y]/\langle \mu(Y) \rangle$ where $q = p^\delta$ and μ is an irreducible polynomial of degree δ, a multiplication in $\mathbb{F}_q$ reduces to a constant number of multiplications in $\mathbb{F}_p[Y]^{<\delta}$. Similarly, a multiplication in $\mathbb{F}_q[X]^{<n}$ reduces to a multiplication in $\mathbb{F}_p[Y]^{<2n\delta}$ by Kronecker substitution, followed by $O(n)$ multiplications in $\mathbb{F}_p[Y]^{<\delta}$ to reduce each coefficient modulo μ. In particular, we get $\mathsf{M}_q(n) = O(\mathsf{M}_p(n\delta))$, uniformly in n and δ.

The computation of the Frobenius automorphism ϕ_q in $\mathbb{F}_{q^d}$ requires $O(\log q)$ multiplications in $\mathbb{F}_{q^d}$ when using binary powering, so this operation has complexity $\Phi_q(d) = O(\mathsf{M}_q(d) \log q)$. This cost can be lowered by representing elements of $\mathbb{F}_{q^d}$ with respect to so-called normal bases [Ble07]. However, multiplication in $\mathbb{F}_{q^d}$ becomes more expensive for this representation.

Another important operation in finite fields is *modular composition*: given polynomials $P, Q \in \mathbb{F}_q[X]^{<n}$ and a monic polynomial $R \in \mathbb{F}_q[X]$ of degree n, the aim is to compute $(P \circ Q) \operatorname{rem} R$. If R is the minimal polynomial of the finite field $\mathbb{F}_{q^n} \cong \mathbb{F}_q[Y]/\langle R(Y) \rangle$, then this operation also corresponds to the evaluation of the polynomial $P \in \mathbb{F}_q[X]^{<n}$ at a point in $\mathbb{F}_{q^n}$. If R and R' are two distinct monic irreducible polynomials in $\mathbb{F}_q[X]$ of degree n, one may wish to convert an element in $\mathbb{F}_q[Y]/\langle R(Y) \rangle$ to an element in $\mathbb{F}_q[Z]/\langle R'(Z) \rangle$. This operation also boils down to one modular composition of degree n over $\mathbb{F}_q$. Indeed, assume given as precomputation $Q(Z) \in \mathbb{F}_q[Z]/\langle R'(Z) \rangle$ being the image of Y. Then the image of $P(Y) \in \mathbb{F}_q[Y]/\langle R(Y) \rangle$ is $P(Q(Z)) \mod R'(Z)$.

We will denote by $\mathsf{C}_q(n)$ the cost of a modular composition as above. Theoretically speaking, Kedlaya and Umans proved the upper bound $\mathsf{C}_q(n) = (n \log q)^{1+o(1)}$ [KU11]. From a practical point of view, one rather has $\mathsf{C}_q(n) = O(n\mathsf{M}_q(n))$ using Horner's rule, or slightly better using fast linear algebra (see e.g. [BK78]). If n is smooth and one needs to compute several modular compositions for the same modulus, then algorithms of quasi-linear complexity do exist [HL18d].

Remark 4.2. The complexity of Kedlaya-Umans's algorithm has a rather theoretical flavor because no efficient implementation is known at the time of writing. To be aware of this issue, it seems important to highlight any dependency on fast modular composition of this type.

The discrete Frobenius Fourier transform potentially involves computations in all intermediate fields $\mathbb{F}_{q^e}$ where e divides d. The necessary minimal polynomials $\mu^{(e)}$ for these fields can be kept in a precomputed table. Notice that the size of this table is $\sum_{e \mid d} e$ (that is $O(d \log \log d)$ by [NR97]) coefficients in $\mathbb{F}_q$.

Now, given $\mu^{(e)}$ and $\mu^{(d)}$, we wish to compute the conversions (embedding or projection)

$$\mathbb{F}_{q^e} \cong \mathbb{F}_q[Y]/\langle \mu^{(e)} \rangle \quad \leftrightarrow \quad \mathbb{F}_{q^d} \cong \mathbb{F}_q[Y]/\langle \mu^{(d)} \rangle \,. \tag{4.1}$$

The paper [HL18d] also contains efficient algorithms for this task: using modular composition, we notice that this problem reduces in time

$$\mathsf{C}_q(d) + (d/e)\mathsf{C}_q(e) + \mathsf{C}_{q^e}(d/e)$$

to the case where we are free to choose $\mu^{(e)}$ and $\mu^{(d)}$ that suit us. We let $\mathsf{V}_q(d,e)$ be a cost function for conversions as in (4.1). We also denote

$$\mathsf{W}_q(d) := \max_{e \,|\, e' \,|\, d} (d/e')\mathsf{V}_q(e', e) \,.$$

The notations are summarized in Table 4.1 for future reference throughout this chapter. For convenience, the complexity of a twiddled DFFT defined in section 4.2.1 is also included in this table.

Table 4.1: Notations for the complexity of various operations.
Note: In this chapter, we use the Turing complexity model.

$\mathsf{M}_q(n)$	Polynomial multiplication in degree n over $\mathbb{F}_q$
$\mathsf{F}_q(n)$	Discrete Fourier Transform of size n over $\mathbb{F}_q$
$\mathsf{C}_q(n)$	Modular composition in degree n over $\mathbb{F}_q$
$\Psi_q(n,d)$	Twiddled DFFT of size n over $\mathbb{F}_q$, output in $\mathbb{F}_{q^d}$ (section 4.2.1).
$\Phi_q(d)$	Computation of one Frobenius map ϕ_q in $\mathbb{F}_{q^d}$
$\mathsf{V}_q(d,e)$	Conversion between $\mathbb{F}_{q^d}$ and $\mathbb{F}_{q^e}$ with $e \,|\, d$, as in (4.1)
$\mathsf{W}_q(d)$	Shortcut for $\max_{e \,|\, e' \,|\, d}(d/e')\mathsf{V}_q(e', e)$

4.2 Frobenius Fourier transforms

Let $\mathbb{F}_q \subseteq \mathbb{F}_{q^d}$ be an extension of finite fields. Let $\phi_q : \mathbb{F}_{q^d} \to \mathbb{F}_{q^d}; x \mapsto x^q$ denote the Frobenius automorphism of $\mathbb{F}_{q^d}$ over $\mathbb{F}_q$. We recall that the group $\langle \phi_q \rangle$ generated by ϕ_q has size d and that it coincides with the Galois group of $\mathbb{F}_{q^d}$ over $\mathbb{F}_q$.

Let $\omega \in \mathbb{F}_{q^d}$ be a primitive n-th root of unity. Recall that n must verify $n \,|\, q^d - 1$ and $\gcd(n, q) = 1$. The Frobenius automorphism ϕ_q naturally acts on $\langle \omega \rangle = (1, \omega, \ldots, \omega^{n-1})$, and we denote by $r_q(\omega^i)$ the order of an element ω^i under this action. Notice that $r_q(\omega^i) \,|\, d$. A subset $\mathcal{S} \subseteq \langle \omega \rangle$ is called a *cross section* if for every $\omega^j \in \langle \omega \rangle$, there exists exactly one $\omega^i \in \mathcal{S}$ such that $\omega^i = \phi_q^k(\omega^j)$ for some $k \in \mathbb{N}$. We will denote this element ω^i by $\pi_{\mathcal{S}}(\omega^j)$. We will also denote the corresponding set of indices by $\mathcal{I} := \{i < n : \omega^i \in \mathcal{S}\}$.

Now consider a polynomial $P \in \mathbb{F}_q[X]^{<n}$. Recall that its discrete Fourier transform with respect to ω is defined to be the vector

$$\mathrm{DFT}_\omega(P) := (P(1), P(\omega), \ldots, P(\omega^{n-1})) \,.$$

Since P admits coefficients in $\mathbb{F}_q$, we have

$$P(\phi_q^k(\omega^j)) = \phi_q^k(P(\omega^j)) \text{ for all } j,k \in \mathbb{N}.$$

For any $\omega^j \in \langle\omega\rangle$, this means that we may retrieve $P(\omega^j)$ from $P(\pi_{\mathcal{S}}(\omega^j))$. Indeed, setting $\omega^i = \pi_{\mathcal{S}}(\omega^j)$ with $\omega^i = \phi_q^k(\omega^j)$, we obtain $P(\omega^j) = \phi_q^{-k}(P(\omega^i))$. We call

$$\mathrm{DFFT}_{\omega,\mathcal{S}}(P) = (P(\omega^i))_{i\in\mathcal{I}}$$

the *discrete Frobenius Fourier transform* of P with respect to ω and the section $\mathcal{S}$.

Let $\omega^i \in \mathcal{S}$ and $r = r_q(\omega^i)$. Then ϕ_q^r leaves ω^i invariant, whence $\phi_q^r(P(\omega^i)) = P(\phi_q^r(\omega^i)) = P(\omega^i)$. Since $\mathbb{F}_{q^r}$ is the subfield of $\mathbb{F}_{q^d}$ that is fixed under the action of ϕ_q^r, this yields

$$P(\omega^i) \in \mathbb{F}_{q^{r_q(\omega^i)}}.$$

It follows that the number of coefficients in $\mathbb{F}_q$ needed to represent $\mathrm{DFFT}_{\omega,\mathcal{S}}(P)$ is given by

$$\sum_{i\in\mathcal{I}} r_q(\omega^i) = n.$$

In particular, the output and input size n of the discrete Frobenius Fourier transform coincide.

4.2.1 Twiddled transforms

For a transform of size $n = n_1 n_2$, let us rewrite the Cooley-Tukey formula (2.3) as follows:

$$\begin{aligned} P(\omega^{u+n_1 v}) &= \sum_{i=0}^{n_2-1} \omega^{iu} \cdot \left(\sum_{j=0}^{n_1-1} P_{i+n_2 j}(\omega^{n_2})^{ju} \right) \cdot (\omega^{n_1})^{iv} \\ &= \sum_{i=0}^{n_2-1} \left(\sum_{j=0}^{n_1-1} P_{i+n_2 j}(\omega^{n_2})^{ju} \right) \cdot (\omega^u(\omega^{n_1})^v)^i \\ &= P^{(\mathrm{R},u)}(\omega^u(\omega^{n_1})^v) \end{aligned}$$

One way to see this is to consider that the second wave of FFTs operates on "twiddled" polynomials $\tilde{P}^{(\mathrm{R},u)}(X) := P^{(\mathrm{R},u)}\big(\omega^u X\big)$. Notice that it corresponds to the presentation from section (2.2.1): ignoring the mirrored indexation in u, the coefficients of $\tilde{P}^{(\mathrm{R},u)}$ are given by the vector $\boldsymbol{\gamma}^{(u)}$. An alternative point of view is to directly consider the evaluation of $P^{(R,u)}$ at the points $\eta(\omega^{n_1})^v$ with $v < n_2$ and where $\eta = \omega^u$. We will call this operation a *twiddled DFT* (with respect to the n_2-th root ω^{n_1}).

The twiddled DFFT is defined in a similar manner. More precisely, assume that we are given an s-th root of unity $\eta \in \mathbb{F}_{q^d}$ such that the set $\eta\langle\omega\rangle$ is globally stable under the action of ϕ_q. Assume also that we have a cross section $\mathcal{S}$ of $\eta\langle\omega\rangle$ under this action and the corresponding set of indices $\mathcal{I} := \{i < n : \eta\omega^i \in \mathcal{S}\} \subseteq \{0,\dots,n-1\}$. Then the *twiddled DFFT* computes the family

$$\mathrm{DFFT}_{\eta,\omega,\mathcal{S}}(P) := (P(\eta\omega^i))_{i\in\mathcal{I}}.$$

Again, we notice that

$$\sum_{i\in\mathcal{I}} r_q(\eta\omega^i) = n\,. \tag{4.2}$$

This operation will serve as the base case for the Frobenius FFT algorithm in section 4.3; we assume the following prototype:

Algorithm 4.1. Twiddled Discrete Frobenius Fourier Transform

Prototype: `TDFFT`$(q, n, \eta, \omega, \mathcal{S}, P)$

Input: A prime power $q = p^\delta$ and an integer n, roots of unity $\eta, \omega \in \mathbb{F}_{q^d}$ (with ω of order n and $\eta\langle\omega\rangle$ globally stable under the action of ϕ_q), an index set $\mathcal{I} := \{i < n : \eta\omega^i \in \mathcal{S}\}$ (where $\mathcal{S}$ is a cross-section of $\eta\langle\omega\rangle$), and a polynomial $P \in \mathbb{F}_q[X]^{<n}$.

Output: The vector $(P(\eta\omega^i))_{i\in\mathcal{I}}$.

We will denote by $\Psi_q(n, d)$ the complexity of computing a twiddled DFFT of this kind. We expect $\Psi_q(n, d) \approx 1/d\,\mathsf{F}_{q^d}(n)$. The next subsections give examples of twiddled DFFTs in various situations and analyze the speed-up factor in each case.

4.2.2 The naive strategy

If n is a small number, it is most efficient to compute ordinary DFTs of order n in a naive fashion, by evaluating $P(\omega^i)$ for $i < n$ using Horner's method. From an implementation point of view, one may do this using specialized codelets for various small n.

The same naive strategy can also be used for the computation of DFFTs and twiddled DFFTs. Given a cross section $\mathcal{S} \subseteq \eta\langle\omega\rangle$ and the corresponding set of indices $\mathcal{I}$, it suffices to evaluate $P(\eta\omega^i)$ separately for each $i \in \mathcal{I}$. Let $r_i := r_q(\eta\omega^i) \,|\, d$ be the order of $\eta\omega^i$ under the action of ϕ_q. Then $\eta\omega^i$ actually belongs to $\mathbb{F}_{q^{r_i}}$, so the evaluation of $P(\eta\omega^i)$ can be done in time $n\mathsf{M}_{q^{r_i}}(1) + O(nr_i \log q)$. With the customary assumption that $\mathsf{M}_{q^n}(1)/n$ is increasing as a function of n, it follows using (4.2) that the complete twiddled DFFT can be computed in time

$$\begin{aligned}\Psi_q(n, d) &\leqslant n \sum_{i\in\mathcal{I}} \mathsf{M}_{q^{r_i}}(1) + O(n^2 \log q)\\ &\leqslant \frac{n^2}{d}\mathsf{M}_{q^d}(1) + O(n^2 \log q)\,.\end{aligned}$$

Assuming that full DFTs of order n over $\mathbb{F}_{q^d}$ are also computed using the naive strategy, this yields

$$\Psi_q(n, d) \leqslant \frac{1}{d}\mathsf{F}_{q^d}(n)(1 + o(1))\,. \tag{4.3}$$

In other words, for small n, we have achieved our goal to gain a factor d.

4.2.3 Full Frobenius action

The next interesting case for study is when n is prime and the action of ϕ_q on $\eta\langle\omega\rangle$ is either transitive, or has only two orbits and one of them is trivial. If $\eta \in \langle\omega\rangle$,

then $\omega^0 = 1$ always forms an orbit of its own in $\eta\langle\omega\rangle$, but we can have $|\mathcal{S}| = 2$. If $\eta \notin \langle\omega\rangle$, then it can happen that $\mathcal{S}$ is a singleton. This happens for instance for a 9-th primitive root of unity $\eta \in \mathbb{F}_{4^3}$, for $\omega = \eta^3 \in \mathbb{F}_4$, and $n = 3$.

If $\mathcal{S}$ is a singleton, say $\mathcal{S} = \{\eta\}$, then we necessarily have $r_q(\eta) = n$ and the DFFT reduces to a single evaluation $P(\eta)$ with $P \in \mathbb{F}_q[X]^{<n}$ and $\eta \in \mathbb{F}_{q^n}$. This is precisely the problem of modular composition that we encountered in section 4.1, whence $\Psi_q(n, n) = \mathsf{C}_q(n)$. The speed-up $\mathsf{F}_{q^n}(n)/\mathsf{C}_q(n)$ with respect to a full DFT is comprised between 1 and n, depending on the efficiency of our algorithm for modular composition. Theoretically speaking, we recall that $\mathsf{C}_q(n) = (n \log q)^{1+o(1)}$, which allows us to gain a factor $n/(n \log q)^\epsilon$ for any $\epsilon > 0$.

In a similar way, if $\eta \in \langle\omega\rangle$ and $|\mathcal{S}| = 2$, then the computation of one DFFT reduces to n additions (in order to compute $P(1)$) and one composition modulo a polynomial of degree $n - 1$ (instead of n).

Remark 4.3. Assume that n is prime, $\eta \in \langle\omega\rangle$ and $|\mathcal{S}| = 2$. If we are free to choose the representation of elements in $\mathbb{F}_{q^n}$, then the DFFT becomes particularly efficient if we take $\mathbb{F}_{q^n} = \mathbb{F}_q[\omega]$. In other words, if $\mu \in \mathbb{F}_q[Y]$ is the monic minimal polynomial of ω over $\mathbb{F}_q$ (so that $\deg \mu = d = n - 1$), then we represent elements of $\mathbb{F}_{q^n}$ as polynomials in $\mathbb{F}_q[Y]$ modulo μ, so that $\omega = Y \mod \mu$. Given $P \in \mathbb{F}_q[X]^{<n}$, just writing down $P(\omega) = (P - P_d\mu)(Y)$ then yields the evaluation of P at ω, whence

$$\Psi_q(n, n) = n\mathsf{M}_q(1) + O(n \log q)\,.$$

4.3 The Frobenius FFT

Let $n = p_0 \cdots p_{\ell-1}$ and $\mathbf{v} = (p_0, \ldots, p_{\ell-1})$ be as in section 2.2.3, and let $\omega \in \mathbb{F}_{q^d}$ be a primitive n-th root of unity. This section presents a variant of the FFT from section 2.2.1: the *Frobenius FFT* (or FFFT). This algorithm uses the same mirrored indexation as the Cooley-Tukey FFT: given $P \in \mathbb{F}_q[X]^{<n}$, it thus returns the family $(P(\omega^{[i]_\mathbf{v}}))_{i\in\mathcal{I}}$. We start with the description of the index set $\mathcal{I} = \{i < n : \omega^{[i]_\mathbf{v}} \in \mathcal{S}\}$ that corresponds to the so-called "privileged cross section" $\mathcal{S}$ of $\langle\omega\rangle$.

4.3.1 Privileged cross sections

Given $i < n$, consider the orbit $\mathcal{O}_q(\omega^{[i]_\mathbf{v}}) := \langle\phi_q\rangle(\omega^{[i]_\mathbf{v}})$ of $\omega^{[i]_\mathbf{v}}$ under the action of $\langle\phi_q\rangle$. In this orbit, we keep the leftmost element in mirrored indexation: we define

$$\pi_{q,\mathbf{v}}(\omega^{[i]_\mathbf{v}}) := \omega^{[j]_\mathbf{v}}\,,$$
$$\text{where } j := \min\{0 \leqslant k < n : \omega^{[k]_\mathbf{v}} \in \mathcal{O}_q(\omega^{[i]_\mathbf{v}})\}\,.$$

Then $\mathcal{S} := \operatorname{im}\pi_{q,\mathbf{v}}$ is a cross section of $\langle\omega\rangle$ under the action of ϕ_q; we call it the *privileged cross section* for ϕ_q (and $\mathbf{v}$).

As in the previous chapters, let $\mathbf{v}^{(\mathrm{L})} := (p_0, \ldots, p_{h-1})$ and $\mathbf{v}^{(\mathrm{R})} := (p_h, \ldots, p_{\ell-1})$ for some $h \leqslant d$; define also $n^{(\mathrm{L})} := p_0 \cdots p_{h-1}$ and $n^{(\mathrm{R})} := p_h \cdots p_{\ell-1}$. Then $\omega^{(\mathrm{L})} := \omega^{n^{(\mathrm{R})}}$ is a primitive $n^{(\mathrm{L})}$-th root of unity. Let $\mathcal{S}^{(\mathrm{L})} := \operatorname{im}\pi_{q,\mathbf{v}^{(\mathrm{L})}}$ be the corresponding privileged cross section of $\langle\omega^{(\mathrm{L})}\rangle$.

Lemma 4.1. *Let $i = an^{(\mathrm{R})} + b$ (with $a < n^{(\mathrm{L})}$ and $b < n^{(\mathrm{R})}$) and $\alpha < n^{(\mathrm{L})}$. To simplify notations, we set $\bar{\imath} := [i]_{\mathbf{v}}$, $\bar{a} := [a]_{\mathbf{v}^{(\mathrm{L})}}$, $\bar{b} := [b]_{\mathbf{v}^{(\mathrm{R})}}$, and $\bar{\alpha} := [\alpha]_{\mathbf{v}^{(\mathrm{L})}}$. We have the equivalence*

$$(\omega^{(\mathrm{L})})^{\bar{\alpha}} \in \mathcal{O}\big((\omega^{(\mathrm{L})})^{\bar{a}}\big) \quad \Leftrightarrow \quad \omega^{[\alpha n^{(\mathrm{R})}+\beta]_{\mathbf{v}}} \in \mathcal{O}\big(\omega^{\bar{\imath}}\big) \text{ for some } \beta < n^{(\mathrm{R})}\,.$$

Proof. Let $j := \alpha n^{(\mathrm{R})} + \beta$, and define $\bar{\jmath} := [j]_{\mathbf{v}}$ and $\bar{\beta} := [\beta]_{\mathbf{v}^{(\mathrm{R})}}$. Assume that $\omega^{\bar{\jmath}} = \phi_q^k(\omega^{\bar{\imath}})$ for some k. Since

$$(\omega^{\bar{\imath}})^{n^{(\mathrm{R})}} = \big(\omega^{\bar{b}\,n^{(\mathrm{L})}+\bar{a}}\big)^{n^{(\mathrm{R})}} = \big(\omega^{\bar{a}}\big)^{n^{(\mathrm{R})}} = (\omega^{(\mathrm{L})})^{\bar{a}}\,.$$

and similarly $(\omega^{\bar{\jmath}})^{n^{(\mathrm{R})}} = (\omega^{(\mathrm{L})})^{\bar{\alpha}}$, it follows that $(\omega^{(\mathrm{L})})^{\bar{\alpha}} = \phi_q^k\big((\omega^{(\mathrm{L})})^{\bar{a}}\big)$.

Inversely, given $\alpha < n^{(\mathrm{L})}$ with $(\omega^{(\mathrm{L})})^{\overline{\alpha}} = \phi_q^k\big((\omega^{(\mathrm{L})})^{\bar{a}}\big)$ for some k, we claim that there exists a $\beta < n^{(\mathrm{R})}$ such that $\omega^{\bar{\jmath}} = \phi_q^k(\omega^{\bar{\imath}})$ for $j = \alpha n^{(\mathrm{R})} + \beta$. Indeed, $(\omega^{(\mathrm{L})})^{\bar{\alpha}} = \phi_q^k\big((\omega^{(\mathrm{L})})^{\bar{a}}\big)$ implies that

$$\big(\omega^{\bar{\alpha}}\big)^{n^{(\mathrm{R})}} = \big(\phi_q^k(\omega^{\bar{a}})\big)^{n^{(\mathrm{R})}} = \big(\phi_q^k(\omega^{\bar{\imath}})\big)^{n^{(\mathrm{R})}}\,,$$

whence $\omega^{\bar{\alpha}}/\phi_q^k(\omega^{\bar{\imath}})$ is an $n^{(\mathrm{R})}$-th root of unity. This means that there exists a $\beta < n^{(\mathrm{R})}$ with $\omega^{\bar{\alpha}}/\phi_q^k(\omega^{\bar{\imath}}) = (\omega^{n^{(\mathrm{L})}})^{-\bar{\beta}}$, i.e.

$$\omega^{\bar{\jmath}} = \omega^{\bar{\alpha}+\bar{\beta}n^{(\mathrm{L})}} = \phi_q^k(\omega^{\bar{\imath}})\,.$$

□

This leads to an expression of $\mathcal{S}^{(\mathrm{L})}$ as a function of $\mathcal{S}$:

Proposition 4.2. *We have*

$$\mathcal{S}^{(\mathrm{L})} = \mathcal{S}^{\,n^{(\mathrm{R})}} := \{u^{\,n^{(\mathrm{R})}} : u \in \mathcal{S}\}.$$

Proof. Assume that j is minimal with $\omega^{\bar{\jmath}} = \phi_q^k(\omega^{\bar{\imath}})$ for some k. Given $\gamma < n^{(\mathrm{L})}$ such that $(\omega^{(\mathrm{L})})^{\bar{\gamma}} \in \mathcal{O}_q\big((\omega^{(\mathrm{L})})^{\bar{a}}\big)$, Lemma 4.1 shows that there exists a $\delta < n^{(\mathrm{R})}$ such that $\omega^{\bar{\ell}} \in \mathcal{O}_q(\omega^{\bar{\imath}})$ for $\ell = \gamma n^{(\mathrm{R})} + \delta$. But then $j \leqslant \ell$, whence $\alpha \leqslant \gamma$. This shows that $\omega^{\bar{\jmath}} \in \mathcal{S}$ implies $(\omega^{\bar{\jmath}})^{n^{(\mathrm{R})}} = (\omega^{(\mathrm{L})})^{\bar{\alpha}} \in \mathcal{S}^{(\mathrm{L})}$.

Inversely, assume that $\alpha < n^{(\mathrm{L})}$ is minimal such that $(\omega^{(\mathrm{L})})^{\bar{\alpha}} \in \mathcal{O}_q((\omega^{(\mathrm{L})})^{\bar{a}})$. Then Lemma 4.1 implies that there exists a $\beta < n^{(\mathrm{R})}$ such that $\omega^{\bar{\jmath}} \in \mathcal{O}_q(\omega^{\bar{\imath}})$ for $j = \alpha n^{(\mathrm{R})} + \beta$. Without loss of generality we may assume that β was chosen minimal while satisfying this property. Given $\gamma < n^{(\mathrm{L})}$, $\delta < n^{(\mathrm{R})}$ and $\ell = \gamma n^{(\mathrm{R})} + \delta$ with $\omega^{\bar{\ell}} \in \mathcal{O}_q(\omega^{\bar{\imath}})$, we have $(\omega^{(\mathrm{L})})^{\bar{\ell}} \in \mathcal{O}_q\big((\omega^{(\mathrm{L})})^{\bar{a}}\big)$. Consequently, $\alpha \leqslant \gamma$, and $\beta \leqslant \delta$ whenever $\alpha = \gamma$. In other words, $j \leqslant \ell$. This shows that $(\omega^{(\mathrm{L})})^{\bar{\alpha}} \in \mathcal{S}^{(\mathrm{L})}$ implies the existence of a $\beta < n^{(\mathrm{R})}$ with $\omega^{\bar{\jmath}} \in \mathcal{S}$ and $(\omega^{\bar{\jmath}})^{n^{(\mathrm{R})}} = (\omega^{(\mathrm{L})})^{\bar{\alpha}}$. □

4.3.2 The main algorithm

We are now in a position to adapt the Cooley-Tukey FFT. In the case when $\ell = 1$ (in particular if n is prime), we will use one of the algorithms from section 4.2 for the computation of twiddled DFFTs. This leads to Algorithm 4.2 below. Recall the notations $\boldsymbol{\alpha}^{(i)}$, $\boldsymbol{\beta}^{(i)}$, $\boldsymbol{\gamma}^{(j)}$, $\boldsymbol{\delta}^{(j)}$ from section 2.2.3 for the intermediate working vectors in the FFT algorithm.

Algorithm 4.2. Frobenius Fast Fourier Transform

Prototype: $\texttt{FFFT}(q, \mathbf{v}, \omega, P)$

Input: A prime power $q = p^\delta$, a vector $\mathbf{v} = (p_0, \ldots, p_{\ell-1})$, a n-th root of unity $\omega \in \mathbb{F}_{q^d}$ and a polynomial $P \in \mathbb{F}_q[X]^{<n}$.

Output: The vector $(P(\omega^{[i]_\mathbf{v}}))_{i \in \mathcal{I}}$, where $\mathcal{I}$ is the index set corresponding to the privileged cross-section $\mathcal{S}$ for ϕ_q.

1: **if** $\ell = 1$ **then return** $\texttt{TDFFT}(q, n, 1, \omega, \mathcal{I}, P)$. ▷ Algorithm 4.1
2: **end if**
3: Take $h = \ell - 1$. ▷ see Remark 4.4
4: Set $n_1 := n^{(\mathrm{L})} = \prod_{i<h} p_i$ and $\mathbf{v}^{(\mathrm{L})} := (p_i)_{i<h}$.
5: Set $n_2 := n^{(\mathrm{R})} = \prod_{h \leqslant i} p_i$ and $\mathbf{v}^{(\mathrm{R})} := (p_i)_{h \leqslant i}$.
6: Set $\omega^{(\mathrm{L})} := \omega^{n^{(\mathrm{R})}}$ and $\omega^{(\mathrm{R})} := \omega^{n^{(\mathrm{L})}}$
7: Let $\mathcal{I}^{(\mathrm{L})} := \left\{ a < n^{(\mathrm{R})} : (\omega^{(\mathrm{L})})^{\bar{a}} \in \mathcal{S}^{(\mathrm{L})} := \mathcal{S}^{n^{(\mathrm{R})}} \right\}$ ▷ cross-section for inner DFFTs
8: **for** $0 \leqslant i < n^{(\mathrm{R})}$ **do**
9: Set $P^{(\mathrm{L},i)}(X) := \sum_{j < n^{(\mathrm{L})}} P_{i+n^{(\mathrm{R})}j} X^j$. ▷ $\approx \boldsymbol{\alpha}^{(i)}$
10: Compute $\boldsymbol{\beta}^{(i)} := \texttt{FFFT}(q, \mathbf{v}^{(\mathrm{L})}, \omega^{(\mathrm{L})}, P^{(\mathrm{L},i)})$. ▷ $\left(P^{(\mathrm{L},i)}((\omega^{(\mathrm{L})})^{\bar{a}})\right)_{a \in \mathcal{I}^{(\mathrm{L})}}$
11: **end for**
12: **for** $a \in \mathcal{I}^{(\mathrm{L})}$ **do**
13: Set $r(a) := r_q((\omega^{(\mathrm{L})})^{\bar{a}})$.
14: Set $P^{(\mathrm{R},a)}(X) := \sum_{i < n^{(\mathrm{R})}} \tilde{\boldsymbol{\beta}}_a^{(i)} X^i$. ▷ $P^{(\mathrm{R},a)} \in \mathbb{F}_{q^{r(a)}}[X]$
15: Set $\mathcal{I}^{(\mathrm{R},a)} := \left\{ b < n^{(\mathrm{R})} : \bar{a}\, n^{(\mathrm{R})} + \bar{b} \in \mathcal{I} \right\}$. ▷ i.e. $\omega^{[i]_\mathbf{v}} \in \mathcal{S}$ for $i := a + n^{(\mathrm{L})} b$
16: **if** $\omega^{\bar{a}}, \omega^{(\mathrm{R})} \in \mathbb{F}_{q^{r(a)}}$ **then** ▷ $\mathcal{I}^{(\mathrm{R},a)} = \{0, \ldots, n^{(\mathrm{R})} - 1\}$
17: Set $\boldsymbol{\gamma}_i^{(a)} := (\omega^{\bar{a}})^i\, P_i^{(\mathrm{R},a)}$.
18: Compute $\boldsymbol{\delta}^{(a)} := \texttt{FFT}(\mathbf{v}^{(\mathrm{R})}, \boldsymbol{\gamma}^{(a)}, \omega^{(\mathrm{R})})$. ▷ Algorithm 2.1
19: **else**
20: Compute $\boldsymbol{\delta}^{(a)} := \texttt{TDFFT}\big(q^{r(a)}, n^{(\mathrm{R})}, \omega^{\bar{a}}, \mathcal{I}^{(\mathrm{R},a)}, P^{(\mathrm{R},a)}\big)$. ▷ Algorithm 4.1
21: **end if**
22: **end for**
23: **return** $(\boldsymbol{\delta}^{(a)})_{a \in \mathcal{I}^{(\mathrm{L})}}$.

Theorem 4.3. *Assuming the correctness of Algorithms 4.1 and 2.1, Algorithm 4.2 is correct.*

Proof. This is a consequence of section 4.2.1 and Proposition 4.2. □

Remark 4.4. For the usual FFT, it was possible to choose h in such a way that $n^{(\mathrm{L})} \approx n^{(\mathrm{R})}$, and we recall that this improves the cache efficiency of practical implementations. However, this optimization is more problematic in our setting since it would require the development of an efficient recursive algorithm for twiddled FFFTs. This is an interesting topic, but raises serious technical difficulties.

Let us now say a word about inverse transforms. For composite sizes, inverse FFFTs can be computed in a straightforward way by reversing the main Algo-

rithm 4.2. It remains to handle the base case, that is inverting the various algorithms from section 4.2. This requires a bit more effort, but does not raise any serious difficulties (as a fallback algorithm, one can always write the matrix of the direct transform and precompute its inverse). The same holds for the algorithms from section 4.4 below to compute multiple DFTs.

Remark 4.5. Following the publication of the original paper [HL17], Li et al. showed that the Frobenius Fourier Transform extends to the case of additive FFTs [LCK+18, CCK+18].

4.3.3 Complexity analysis

Let us now perform the complexity analysis of Algorithm 4.2. For $k \in \{0, \dots, \ell-1\}$, we first focus on all FFTs and twiddled DFFTs that are computed "at stage k" using a fallback algorithm. These FFTs and twiddled DFFTs are all of length p_k. Given $e \,|\, e' \,|\, d$, let $\nu_{k,e,e'}$ be the number of transforms of a polynomial in $\mathbb{F}_{q^e}[X]^{<p_k}$ over $\mathbb{F}_{q^{e'}}$. From (4.2), it follows that

$$\sum_{e \,|\, e' \,|\, d} p_k \nu_{k,e,e'} e = n.$$

Now a naive bound for the cost of an FFT or twiddled DFFT of a polynomial in $\mathbb{F}_{q^e}[X]^{<p_k}$ over $\mathbb{F}_{q^{e'}}$ is

$$\Psi_{q^e}(p_k, e'/e) \leqslant \mathsf{F}_{q^{e'}}(p_k) + p_k \mathsf{M}_{q^e}(1)\,. \tag{4.4}$$

This means that the cost of all FFTs and twiddled DFFTs at stage k is bounded by

$$\begin{aligned}\bar{\Psi}_k &:= \sum_{e \,|\, e' \,|\, d} \nu_{k,e,e'} \Psi_{q^e}(p_k, e'/e) \\ &\leqslant \sum_{e \,|\, e' \,|\, d} \nu_{k,e,e'} \mathsf{F}_{q^{e'}}(p_k) + n\frac{\mathsf{M}_{q^d}(1)}{d}.\end{aligned}$$

Now $\nu_{k,e,e'}$ can only be non zero if $e' \leqslant p_k e$. Consequently,

$$\begin{aligned}\bar{\Psi}_k &\leqslant \sum_{e \,|\, e' \,|\, d} \nu_{k,e,e'} \cdot e \cdot \frac{e'}{e} \cdot \frac{\mathsf{F}_{q^{e'}}(p_k)}{e'} + n\frac{\mathsf{M}_{q^d}(1)}{d} \\ &\leqslant \sum_{e \,|\, e' \,|\, d} \nu_{k,e,e'}\, e\, p_k \frac{\mathsf{F}_{q^d}(p_k)}{d} + n\frac{\mathsf{M}_{q^d}(1)}{d} \\ &\leqslant \frac{n}{d}\big(\mathsf{F}_{q^d}(p_k) + \mathsf{M}_{q^d}(1)\big).\end{aligned}$$

The total cost of all FFTs and twiddled DFFTs of prime length is therefore bounded by

$$\bar{\Psi}_0 + \cdots + \bar{\Psi}_{\ell-1} \leqslant \frac{n\ell}{d}\big(\mathsf{F}_{q^d}(\max p_i) + \mathsf{M}_{q^d}(1)\big)\,.$$

Notice that all twiddled DFFTs are not necessarily over the base field $\mathbb{F}_q$: some of them are applied to polynomials with coefficients in $\mathbb{F}_{q^e}$, over an extension $\mathbb{F}_{q^{e'}}$.

In this case, the output uses the representation $\mathbb{F}_{q^{e'}} \cong \mathbb{F}_{q^e}[X]/\langle \mu^{e',e} \rangle$, which is not the "standard" representation of $\mathbb{F}_{q^{e'}}$. This means the result needs to be converted; the cost of all such conversions at stage k is bounded by

$$\begin{aligned}\bar{\mathsf{V}}_k &:= \sum_{e \neq e' \mid d} \nu_{k,e,e'} \mathsf{V}_q(e', e) \\ &\leqslant \sum_{e \neq e' \mid d} p_k \nu_{k,e,e'} e \frac{\mathsf{W}_q(d)}{d}\end{aligned}$$

For a number a with prime factorization $a = p_1^{i_1} \cdots p_r^{i_r}$, let $\lambda(a) := i_1 + \cdots + i_r$. We also denote $N_{k,e} = \sum_{e' \in e\mathbb{Z}} p_k \nu_{k,e,e'} e$ for the total size of the transforms "at stage k" that take inputs in $\mathbb{F}_{q^e}$ (in terms of the number of coefficients in $\mathbb{F}_q$). We observe that $\sum_{e \mid e'} p_k \nu_{k,e,e'} e$ is the total size of the transforms "at stage k" that *have their outputs in* $\mathbb{F}_{q^{e'}}$, hence $N_{k+1,e'} = \sum_{e \mid e'} p_k \nu_{k,e,e'} e$. Let us show by induction over k that

$$\bar{\mathsf{V}}_0 + \cdots + \bar{\mathsf{V}}_{k-1} \leqslant \sum_{e \mid d} N_{k,e} \lambda(e) \frac{\mathsf{W}_q(d)}{d}.$$

This is clear for $k = 0$, since $N_{0,1} = n$ and $N_{0,e} = 0$ for $e > 1$. Assuming that the relation holds for a given k, we get

$$\begin{aligned}\bar{\mathsf{V}}_0 + \cdots + \bar{\mathsf{V}}_k &\leqslant \sum_{e \mid e' \mid d} p_k \nu_{k,e,e'} e \lambda(e) \frac{\mathsf{W}_q(d)}{d} + \sum_{e \neq e' \mid d} p_k \nu_{k,e,e'} e \frac{\mathsf{W}_q(d)}{d} \\ &\leqslant \sum_{e \mid e' \mid d} p_k \nu_{k,e,e'} e\, \lambda(e') \frac{\mathsf{W}_q(d)}{d} \\ &= \sum_{e' \mid d} N_{k+1,e'} \lambda(e') \frac{\mathsf{W}_q(d)}{d}.\end{aligned}$$

This completes the induction and we conclude that the total conversion cost is bounded by

$$\bar{\mathsf{V}}_0 + \cdots + \bar{\mathsf{V}}_{\ell-1} \leqslant n \lambda(d) \frac{\mathsf{W}_q(d)}{d}.$$

This concludes the proof of the following result:

Theorem 4.4. *If $n = p_0 \cdots p_{\ell-1}$ where $p_0, \ldots, p_{\ell-1}$ are all prime, then*

$$\Psi_q(n, d) \leqslant \frac{n}{d} \big(\ell \mathsf{F}_{q^d}(\max p_i) + \ell \mathsf{M}_{q^d}(1) + \lambda(d) \mathsf{W}_q(d) \big),$$

Remark 4.6. If the p_i are very small, then we have shown in section 4.2.2 that (4.4) can be further sharpened into $\Psi_{q^e}(p_i, e'/e) \sim e/e'\, \mathsf{F}_{q^{e'}}(p_i)$, hence

$$\bar{\Psi}_k \lesssim \frac{n}{d} \left(\frac{\mathsf{F}_{q^d}(p_k)}{p_k} + \mathsf{M}_{q^d}(1) \right).$$

As a consequence, the bound for $\Psi_q(n, d)$ becomes

$$\Psi_q(n, d) \lesssim \frac{1}{d} \mathsf{F}_{q^d}(n) + \frac{n}{d} \lambda(d) \mathsf{W}_q(d).$$

Remark 4.7. It is also possible to treat separately the case when $e = e'$ in the bound for $\bar{\Psi}_k$, thereby avoiding the factor $p_k \geqslant e'/e$ whenever possible. A similar analysis as for the cost of the conversions then yields

$$\Psi_q(n,d) \leqslant \frac{n}{d}\left(\ell\frac{\mathsf{F}_{q^d}(\max p_i)}{\max p_i} + \ell\mathsf{M}_{q^d}(1) + \lambda(d)\mathsf{F}_{q^d}(\max p_i) + \lambda(d)\mathsf{W}_q(d)\right).$$

Whenever we can manage to keep $\max p_i$ and $\mathsf{W}_q(d)/d$ small with respect to ℓ, this means that we gain a factor d with respect to a full DFT.

4.4 Multiple Fourier transforms

In this section, we present an alternative approach that packs d Fourier transforms over $\mathbb{F}_q$ into one transform over $\mathbb{F}_{q^d}$. As an analogy with the complex number, assume $A, B \in \mathbb{R}[X]$; then the two DFTs of A, B can be computed from the single DFT of $P := A + \mathrm{i}B$.

4.4.1 Parallel lifting

Let $\mathbb{F}_{q^d} \cong \mathbb{F}_q[T]/\langle\mu(T)\rangle$ be an extension of $\mathbb{F}_q$. Consider the primitive element $\alpha := T \mod \mu \in \mathbb{F}_{q^d}$; that is $1, \alpha, \ldots, \alpha^{d-1}$ is a basis of $\mathbb{F}_{q^d}$. Given d polynomials $P^{(0)}, \ldots, P^{(d-1)} \in \mathbb{F}_q[X]^{<n}$, we may then form the polynomial

$$\tilde{P} := P^{(0)} + P^{(1)}\alpha + \cdots + P^{(d-1)}\alpha^{d-1} \in \mathbb{F}_{q^d}[X].$$

If $n \mid q-1$ and ω is a primitive n-th root of unity *in the base field* $\mathbb{F}_q$, then we notice that

$$\mathrm{DFT}_\omega(P) = \mathrm{DFT}_\omega(P^{(0)}) + \cdots + \mathrm{DFT}_\omega(P^{(d-1)})\alpha^{d-1}. \tag{4.5}$$

In other words, the discrete Fourier transform operates coefficientwise with respect to the basis $1, \alpha, \ldots, \alpha^{d-1}$ of $\mathbb{F}_{q^d}$.

The map $P^{(0)}, \ldots, P^{(d-1)} \mapsto P^{(0)} + P^{(1)}\alpha + \cdots + P^{(d-1)}\alpha^{d-1}$ and its inverse boil down to matrix transpositions in memory as follows:

$$\Big(\sum_{i<n} P_i^{(0)}X^i, \ldots, \sum_{i<n} P_i^{(d-1)}X^i\Big) \leftrightarrow \sum_{i<n}\Big(\sum_{j<d} P_i^{(j)}\alpha^j\Big)X^i$$

On a Turing machine, they can therefore be computed in time $O(nd\log d\log q)$. If $\tilde{\mathsf{F}}_q(n,t)$ stands for the cost of computing t Fourier transforms of length n over $\mathbb{F}_q$, then it follows from (4.5) that

$$\tilde{\mathsf{F}}_q(n,d) \leqslant \mathsf{F}_{q^d}(n) + O(nd\log d\log q). \tag{4.6}$$

4.4.2 Symmetric multiplexing

Let us next consider an s-th root of unity η and a primitive n-th root of unity ω *in the extension field* $\mathbb{F}_{q^d}$. Now reconstructing the DFTs of each $P^{(i)}$ from the DFT

of $\tilde{P}$ requires a bit more work than in the previous subsection. Given $u = \eta\omega^i$, we have for all k,

$$\phi_q^k(P(\phi_q^{-k}(u))) = P^{(0)}(u)\phi_q^k(1) + \cdots + P^{(d-1)}(u)\phi_q^k(\alpha^{d-1}).$$

Abbreviating $\Phi_{P,k}(u) := \phi_q^k(P(\phi_q^{-k}(u)))$ and setting

$$\mathbf{B} := \begin{pmatrix} 1 & \cdots & \alpha^{d-1} \\ \vdots & & \vdots \\ \phi_q^{d-1}(1) & \cdots & \phi_q^{d-1}(\alpha^{d-1}) \end{pmatrix}$$

$$\mathbf{\Phi}^{(P)}(u) := \begin{pmatrix} \Phi_{P,0}(u) \\ \vdots \\ \Phi_{P,d-1}(u) \end{pmatrix} \qquad \mathbf{V}^{(P)}(u) := \begin{pmatrix} P^{(0)}(u) \\ \vdots \\ P^{(d-1)}(u) \end{pmatrix}$$

it follows that

$$\mathbf{\Phi}^{(P)}(u) = \mathbf{B}\mathbf{V}^{(P)}(u)\,. \tag{4.7}$$

Now given the twiddled discrete Fourier transform $\mathrm{DFT}_{\eta,\omega}(P) := (P(\eta\omega^i))_{i<n}$ of P, we in particular know the values $P(u), \ldots, P(\phi_q^{d-1}(u))$. Letting the Frobenius automorphism ϕ_q act on these values, we obtain the vector $\mathbf{\Phi}^{(P)}(u)$. Using one matrix-vector product $\mathbf{V}^{(P)}(u) = \mathbf{B}^{-1}\mathbf{\Phi}^{(P)}(u)$, we may then retrieve the values of the individual twiddled transforms $\mathrm{DFT}_{\eta,\omega}(P_i)$ at u. Doing this for each $u \in \mathcal{S}$ in a cross section of $\eta\langle\omega\rangle$ under the action of ϕ_q, this yields a way to retrieve the twiddled DFFTs of $P_0, \ldots, P_{d-1}$ from the twiddled DFT of P.

Since $\mathbf{B}$ is a Vandermonde matrix, the matrix-vector product $\mathbf{B}^{-1}\mathbf{\Phi}^{(P)}(u)$ can be computed in time $O(\mathsf{M}_q(d)\log d)$ using polynomial interpolation [GG13, Chapter 10]. Given $\mathrm{DFT}_{\eta,\omega}(P)$, we may compute each individual vector $\Phi_P(u)$ in time $O(d\mathsf{M}_q(d)\log(dq))$ using the Frobenius automorphism ϕ_q. Since there are $|\mathcal{S}|$ orbits in $\langle\omega\rangle$ under the action of ϕ_q, we may thus retrieve the $\mathrm{DFFT}_{\eta,\omega}(P_i)$ from $\mathrm{DFT}_{\eta,\omega}(P)$ in time $O(|\mathcal{S}|\,d\,\mathsf{M}_q(d)\log q^d)$. In other words, if $\tilde{\Psi}_q(n,d,t)$ denotes the complexity of computing the DFFTs of t elements of $\mathbb{F}_q[X]^{<n}$ over $\mathbb{F}_{q^d}$, we have

$$\tilde{\Psi}_q(n,d,d) \leqslant \mathsf{F}_{q^d}(n) + O\big(|\mathcal{S}|\,d\,\mathsf{M}_q(d)\log q^d\big)\,. \tag{4.8}$$

4.4.3 Multiplexing over an intermediate field

In the extreme case when $\eta, \omega \in \mathbb{F}_q$, every $\eta\omega^i \in \eta\langle\omega\rangle$ has order one under the action of ϕ_q and $|\mathcal{S}| = n$. In that case, the bound (4.8) is not very sharp, but (4.6) already provides us with a good alternative bound for this special case. For $r \notin \{1, d\}$ with $r \mid d$, let us now consider the case when $u = \eta\omega^i$ has order at most r under the action of ϕ_q for all i, so that $u \in \mathbb{F}_{q^r}$. Let $\beta \in \mathbb{F}_{q^r}$ be a primitive element in $\mathbb{F}_{q^r}$ over $\mathbb{F}_q$, so that $\mathbb{F}_{q^r} = \mathbb{F}_q[\beta]$ and $\mathbb{F}_{q^d} = \mathbb{F}_{q^r}[\alpha]$. Given our d polynomials $P^{(0)}, \ldots, P^{(d-1)} \in \mathbb{F}_q[X]^{<n}$, we may form

$$\begin{aligned} \tilde{P}^{(i)} &= P^{(ir)} + \cdots + P^{(ir+r-1)}\beta^{r-1} \;\in\; \mathbb{F}_{q^r}[X]^{<n} \quad (i < d/r)\,, \\ \bar{P} &= \tilde{P}^{(0)} + \tilde{P}^{(1)}\alpha + \cdots + \tilde{P}^{(d/r-1)}\alpha^{d/r-1} \;\in\; \mathbb{F}_{q^d}[X]^{<n}\,. \end{aligned}$$

Then we have

$$\mathrm{DFT}_{\eta,\omega}(\bar{P}) = \mathrm{DFT}_{\eta,\omega}(\tilde{P}^{(0)}) + \cdots + \mathrm{DFT}_{\eta,\omega}(\tilde{P}^{(d/r-1)})\alpha^{d/r-1}\,.$$

Moreover, we may compute $(\mathrm{DFFT}_{\eta,\omega}(P^{(ir+j)}))_{j<r}$ from $\mathrm{DFT}_{\eta,\omega}(\tilde{P}^{(i)})$ for each $i < d/r$, using the algorithm from the previous subsection. Notice the additional property that at least one $u \in \eta\langle\omega\rangle$ has maximal order $r = [\mathbb{F}_{q^r} : \mathbb{F}_q]$ under the action of ϕ_q.

If n is prime and $\eta \in \langle\omega\rangle$, then the above discussion shows that, without loss of generality, we may assume that ω has order d under the action of ϕ_q. This means that ω^i has maximal order d for every $i \in \{1, \ldots, n-1\}$, whereas $\omega^0 = 1$ has order one. Hence, $|\mathcal{S}| = (n-1)/d+1$. Similarly, if n is prime and $\eta \notin \langle\omega\rangle$, then we obtain a reduction to the case when $\eta\omega^i$ has maximal order d for all $i \in \{0, \ldots, n-1\}$, whence $|\mathcal{S}| = n/d$. In both cases, we obtain the following:

Proposition 4.5. *If n is prime, then*

$$\tilde{\Psi}_q(n, d, d) = \mathsf{F}_{q^d}(n) + O\big(n\mathsf{M}_q(d)\log q^d\big).$$

Remark 4.8. We recall that $n \leqslant q^d - 1$, whence $\log n \leqslant \log q^d$. When using the best known bound $\mathsf{M}_q(n) = \Theta(n \log q \log(n \log q))$ and $\mathsf{F}_{q^d}(n) \asymp \mathsf{M}_{q^d}(n)$, it follows for some constant $C > 0$ that

$$\begin{aligned}\frac{\tilde{\Psi}_q(n, d, d)}{\mathsf{F}_{q^d}(n)} &\geqslant 1 + C\frac{d\log q\log(d\log q)}{\log(nd\log q)}\\ &\geqslant 1 + C\frac{\log(d\log q)}{2}\,.\end{aligned}$$

In other words, we cannot hope to gain more than an asymptotic factor of $d/\log d$ with respect to a full DFT.

If n is not necessarily prime, then the technique from this section can still be used for every individual u. However, the rewritings of elements in $\mathbb{F}_{q^d}$ as elements in $\mathbb{F}_{q^r}[\alpha]$ and $\mathbb{F}_q[\beta][\alpha]$ have to be done individually for each u using modular compositions. Denoting by r_i the order of $\eta\omega^i$ under ϕ_q, it follows that

$$\begin{aligned}\tilde{\Psi}_q(n, d, d) &= \mathsf{F}_{q^d}(n) + \sum_{i\in\mathcal{I}} \mathsf{V}_q(d, r_i) + \sum_{i\in\mathcal{I}} O\big(r_i\mathsf{M}_{q^{r_i}}(d/r_i)\log q^d\big)\\ &\leqslant \mathsf{F}_{q^d}(n) + n\mathsf{W}_q(d) + O\big(n\mathsf{M}_q(d)\log q^d\big).\end{aligned}$$

Notice that conversions of the same type correspond to modular compositions with the same modulus. If n is smooth, then it follows that we may use the algorithms from [HL18d] and keep the cost of the conversions quasi-linear (as a more practical alternative to Kedlaya-Umans, see Remark 4.2).

Chapter 5

Multiplication of polynomials over the binary field

Contents

Note. *The results from this chapter were published in* [HLL17]. *The implementation is part of the* JUSTINLINE *package of* MATHEMAGIX [HLM+02], *available at* http://www.mathemagix.org. *Compared to the paper* [HLL17], *section 5.3 includes an additional comparison with a software released in 2018; also section 5.4 adds a discussion about other special cases where the same technique would work.*

A situation where the Frobenius FFT would be especially useful is for the multiplication of polynomials over small finite fields. Indeed, the lack of evaluation points limits evaluation-interpolation strategies and forces to work in a certain extension, which implies an overhead. However, the theoretical speedup promised by the Frobenius FFT seems hard to achieve in practice: the algorithm from the previous chapter involves a lot of technicalities, in particular computations in all intermediate fields $\mathbb{F}_{q^e}$ between $\mathbb{F}_q$ and $\mathbb{F}_{q^d}$. For this reason, making the Frobenius FFT competitive with highly-optimized plain FFT implementations is a real challenge in general, but in the specific case of the extension $\mathbb{F}_{2^{60}}$ over $\mathbb{F}_2$, it turns out that the Frobenius FFT adapts particularly well.

As mentioned earlier, the field $\mathbb{F}_{2^{60}}$ has remarkable properties that can be used for an efficient multiplication in $\mathbb{F}_2[X]$, as shown in [HHL16a]. Naturally, one cannot afford the factor 60 overhead that comes from the naive embedding $\mathbb{F}_2[X] \hookrightarrow \mathbb{F}_{2^{60}}[X]$.

The traditional solution is to pack $30 = 60/2$ input coefficients into one element of the extension field using Kronecker substitution

$$\mathbb{F}_2[X] \hookrightarrow \mathbb{F}_2[Y]^{<30}[Z] \hookrightarrow \mathbb{F}_{2^{60}}[Z]\,.$$

With this method (already used in [HHL16a]), the overhead is reduced to a factor 2. Then, if we were able to eliminate the overhead completely using the Frobenius FFT, we would obtain an implementation that is twice as efficient as the previous one. This chapter shows that such a speedup is actually possible in practice, in the specific case of $\mathbb{F}_{2^{60}}$.

5.1 Fast reduction from $\mathbb{F}_2[X]$ to $\mathbb{F}_{2^{60}}[X]$

This section describes a variant of the Frobenius FFT for the special extension $\mathbb{F}_{2^{60}}$ over $\mathbb{F}_2$. Using a single rewriting step, this new algorithm reduces the computation of a Frobenius DFT to the computation of an ordinary DFT over $\mathbb{F}_{2^{60}}$, thereby avoiding computations in any intermediate fields $\mathbb{F}_{2^e}$ with $1 < e < 60$ and $e \mid 60$.

5.1.1 Variant of the Frobenius DFT

To efficiently reduce a multiplication in $\mathbb{F}_2[X]$ into DFTs over $\mathbb{F}_{2^{60}}$, we use an order n that divides $2^{60} - 1$ and such that $n = 61m$ for some integer m. We perform the Cooley-Tukey decomposition (2.3) with $n_1 = 61$ and $n_2 = m$. Let ω be a primitive n-th root of unity in $\mathbb{F}_{2^{60}}$. The discrete Fourier transform of $P \in \mathbb{F}_2[X]^{<n}$, given by $(P(1), P(\omega), P(\omega^2), \ldots, P(\omega^{n-1})) \in \mathbb{F}_{2^{60}}^n$, can be reorganized into 61 slices as follows

$$\mathrm{DFT}_\omega(P) = \left((P(\omega^{61i}))_{0\leqslant i<m}, (P(\omega^{61i+1}))_{0\leqslant i<m}, \ldots, (P(\omega^{61i+60}))_{0\leqslant i<m}\right).$$

The variant of the Frobenius DFT of P that we introduce in the present chapter corresponds to computing only the second slice:

$$\begin{aligned} E_\omega : \mathbb{F}_2[X]^{<60m} &\to \mathbb{F}_{2^{60}}^m \\ P &\mapsto \left(P(\omega^{61i+1})\right)_{0\leqslant i<m} \end{aligned}$$

Let us show that this transform is actually a bijection. The following lemma shows that the slices $(P(\omega^{61i+2}))_{0\leqslant i<m}, \ldots, (P(\omega^{61i+60}))_{0\leqslant i<m}$ can be deduced from the second slice $(P(\omega^{61i+1}))_{0\leqslant i<m}$ using the action of the Frobenius map ϕ_2.

Lemma 5.1. *Let $\Omega_i = \{\omega^{61j+i} : 0 \leqslant j < m\}$ for $1 \leqslant i < 61$. Then the action of $\langle\phi_2\rangle$ is transitive on the pairwise disjoint sets $\Omega_1, \ldots, \Omega_{60}$.*

Proof. Let $1 \leqslant i < 61$ and $0 \leqslant j < m$, we have $\phi_2(\omega^{61j+i}) = \omega^{61j'+(2i \bmod 61)}$ for some integer $0 \leqslant j' < m$, so the action of $\langle\phi_2\rangle$ onto $\Omega_1, \ldots, \Omega_{60}$ is well defined. Notice that 2 is primitive for the multiplicative group $\mathbb{F}_{61}^\times$. This implies that for any $1 \leqslant i < 61$ there exists k such that $2^k \equiv i \mod 61$. Consequently we have $\phi_2^{\circ k}(\omega^{61j+1}) = \omega^{61j'+i}$ for some $0 \leqslant j' < m$, whence $\phi_2^{\circ k}(\Omega_1) \subseteq \Omega_i$. Since ϕ_2 is injective, the latter inclusion is an equality. □

If we needed the complete $\mathrm{DFT}_\omega(P)$, then we would still have to compute the first slice $(P(\omega^{61i}))_{0\leqslant i<m}$. The second main new idea with respect to Chapter 4 is to discard this first slice, and to restrict ourselves to input polynomials A of degrees $< 60m$. In this way, E_ω can be inverted, as proved in the following proposition.

Proposition 5.2. *E_ω is bijective.*

Proof. The dimensions of the source and destination spaces of E_ω over $\mathbb{F}_2$ being the same, it suffices to prove that E_ω is injective. Let $P \in \mathbb{F}_2[X]^{<60m}$ be such that $E_\omega(P) = 0$. By construction, P vanishes at m distinct values, namely ω^{61i+1} for $0 \leqslant i < m$. Under the action of $\langle \phi_2 \rangle$ it also vanishes at $60(m-1)$ other values by Lemma 5.1, whence $P = 0$. □

Remark 5.1. The transformation E_ω being bijective is due to the fact that 2 is primitive in the multiplicative group $\mathbb{F}_{61}^\times$. Among the prime divisors of $2^{60} - 1$, the factors 3, 5, 11 and 13 also have this property, but taking $n_1 = 61$ allows us to divide the size of the evaluation-interpolation scheme by 60, which is optimal.

5.1.2 Frobenius encoding

We decompose the computation of E_ω into two routines. The first routine is written F_ω and called the *Frobenius encoding*:

$$\begin{aligned} F_\omega : \mathbb{F}_2[X]^{<60m} &\to \mathbb{F}_{2^{60}}[X]^{<m} \\ P = \sum_{0\leqslant k<60m} P_k X^k &\mapsto \sum_{0\leqslant k<m} \omega^k \left(\sum_{0\leqslant l<60} P_{k+ml}\theta^l \right) X^k, \text{ where } \theta = \omega^m \end{aligned} \tag{5.1}$$

Below, we will choose θ in such a way that F_ω is essentially a simple reorganization of the coefficients of P.

We observe that the coefficients of $F_\omega(P)$ are part of the values of the inner DFTs of P in the Cooley–Tukey formula (2.3), applied with $n_1 = 61$ and $n_2 = m$. The second task is the computation of the corresponding outer DFT of order m:

$$\begin{aligned} \mathrm{DFT}_{\tilde\omega} : \mathbb{F}_{2^{60}}[X]^{<m} &\to \mathbb{F}_{2^{60}}^m \\ \tilde P &\mapsto (\tilde P(\tilde\omega^i))_{0\leqslant i<m}, \text{ where } \tilde\omega = \omega^{61} \end{aligned}$$

Proposition 5.3. $E_\omega = \mathrm{DFT}_{\tilde\omega} \circ F_\omega$.

Proof. This formula follows from (2.3):

$$P(\omega^{61i+1}) = \sum_{0\leqslant k<m} \omega^k \left(\sum_{0\leqslant \ell<61} P_{k+m\ell}\theta^\ell \right) \tilde\omega^{ki} = F_\omega(P)(\tilde\omega^i)\,. \qquad \square$$

Summarizing, we have reduced the computation of a DFT of size $60n/61$ over $\mathbb{F}_2$ to a DFT of size $m = n/61$ over $\mathbb{F}_{2^{60}}$. Notice that this reduction preserves data size.

5.1.3 Direct transforms

The computation of F_ω involves the evaluation of m polynomials in $\mathbb{F}_2[X]^{<60}$ at $\theta = \omega^m \in \mathbb{F}_{2^{60}}$. In order to perform these evaluations fast, we fix the representation of $\mathbb{F}_{2^{60}} = \mathbb{F}_2[Z]/\langle \mu(Z) \rangle$ and the primitive root ζ of unity of maximal order $2^{60}-1$ to be given by

$$\mu(Z) := (Z^{61}-1)/(Z-1)\,, \tag{5.2}$$

$$\zeta := Z^{18} + Z^6 + 1 \mod \mu(Z)\,. \tag{5.3}$$

Setting $\omega = \zeta^{(2^{60}-1)/n}$ and $\theta = \zeta^{(2^{60}-1)/61}$, it can be checked that $\theta = Z \mod \mu(Z)$. Evaluation of a polynomial in $\mathbb{F}_2[X]^{<60}$ at θ can now be done efficiently.

Algorithm 5.1. Frobenius Encoding

Prototype: `encode`(n, P, ω)

Input: An integer m, a polynomial $P \in \mathbb{F}_2[X]^{<60m}$, and the n-root ω as above, with $n := 61m$ divides $2^{60}-1$.

Output: The polynomial $\tilde{P} := F_\omega(P) \in \mathbb{F}_{2^{60}}[X]^{<m}$.

1: Set $\tilde{P}_i := \sum_{0 \leqslant j < 60} P_{i+mj} Z^j \mod \mu(Z) \in \mathbb{F}_{2^{60}}$ for each $i \in \{0, \ldots, m-1\}$.
2: **return** $\tilde{P} := \tilde{P}_0 + \omega \tilde{P}_1 X + \omega^2 \tilde{P}_2 X^2 + \cdots + \omega^{m-1} \tilde{P}_{m-1} X^{m-1}$.

Proposition 5.4. *Algorithm 5.1 is correct.*

Proof. This follows immediately from the definition of F_ω in formula (5.1), using the fact that $\theta = Z \mod \mu(Z)$ in our representation. □

Notice that Algorithm 5.1 corresponds to a matrix transposition in memory, up to the alignment for elements of $\mathbb{F}_{2^{60}}$. Combining this with the FFT algorithm, we get a variant of the Frobenius FFT:

Algorithm 5.2. Frobenius FFT (variant for $\mathbb{F}_{2^{60}}$)

Prototype: `FFFT_2_60`$(\mathbf{v}, P)$

Input: A vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, and a polynomial $P \in \mathbb{F}_2[X]^{<60m}$, with $m := p_0 \cdots p_{d-1}$ and $n := 61m \mid 2^{60}-1$.

Output: The vector $\mathbf{E} := \left(P(\omega^{61[i]_\mathbf{v}+1}\right)_{i<m}$ (this is $E_\omega(P)$ in mirrored indexation), with ω as above.

1: Set ζ as in (5.3) and $n := 61 p_0 \cdots p_{d-1}$.
2: Set $\omega := \zeta^{(2^{60}-1)/n}$ and $\tilde{\omega} := \omega^{61}$.
3: Compute $\tilde{P} :=$ `encode`(n, P, ω). ▷ Algorithm 5.1
4: **return** `FFT`$(\mathbf{v}, \tilde{\mathbf{P}}, \tilde{\omega})$ with $\tilde{\mathbf{P}}$ the vector of coefficients of $\tilde{P}$. ▷ Algorithm 2.1

Proposition 5.5. *Algorithm 5.2 is correct.*

Proof. The correctness simply follows from Propositions 5.3 and 5.4. □

5.1.4 Inverse transforms

Inverting the above algorithms is straightforward

Algorithm 5.3. Frobenius decoding

Prototype: `decode`$(n, \tilde{P}, \omega)$
Input: An integer m, a polynomial $\tilde{P} \in \mathbb{F}_{2^{60}}[X]^{<m}$, and the n-root ω as above, with $n := 61m$ divides $2^{60} - 1$.
Output: The polynomial $P := F_\omega^{-1}(P) \in \mathbb{F}_2[X]^{<60m}$.

1: Let $P^{(i)}(Z) \in \mathbb{F}_2[Z]^{<60}$ be the preimage of $\omega^{-i}\tilde{P}_i \in \mathbb{F}_{2^{60}} \cong \mathbb{F}_2[Z]/\langle \mu(Z) \rangle$, for each $i < m$.
2: **return** $\sum_{0 \leqslant i < m} \sum_{0 \leqslant j < 60} P_j^{(i)} X^{i+mj}$.

Proposition 5.6. *Algorithm 5.3 is correct.*

Algorithm 5.4. Inverse Frobenius FFT (variant for $\mathbb{F}_{2^{60}}$)

Prototype: `IFFFT_2_60`$(\mathbf{v}, \mathbf{E})$
Input: A vector $\mathbf{v} = (p_0, \ldots, p_{d-1})$, and a vector $\mathbf{E} \in \mathbb{F}_{2^{60}}^m$, with $m := p_0 \cdots p_{d-1}$ and $n := 61m \mid 2^{60} - 1$.
Output: The polynomial P such that $\mathbf{E} =$ `FFFT_2_60`$(\mathbf{v}, P)$.

1: Set ζ as in (5.3) and $n := 61p_0 \cdots p_{d-1}$.
2: Set $\omega := \zeta^{(2^{60}-1)/n}$ and $\tilde{\omega} := \omega^{61}$.
3: Compute $\tilde{\mathbf{P}} :=$ `IFFT`$(\mathbf{v}, \mathbf{E}, \tilde{\omega})$. ▷ Algorithm 2.2
4: Let $\tilde{P}$ be the corresponding polynomial.
5: **return** `decode`$(n, \tilde{P}, \omega)$. ▷ Algorithm 5.3

Proposition 5.7. *Algorithm 5.4 is correct.*

5.1.5 Multiplication in $\mathbb{F}_2[X]$

Using the standard technique of multiplication by evaluation-interpolation, we may now compute products in $\mathbb{F}_2[X]$ as follows:

Algorithm 5.5. Multiplication in $\mathbb{F}_2[X]$

Prototype: `mul`(A, B)
Input: Polynomials $A, B \in \mathbb{F}_2[X]$ with $60(\deg A + \deg B)/61 < 2^{60} - 1$.
Output: The polynomial AB.

1: Choose $m \geqslant (\deg A + \deg B)/60$ such that $n = 61m$ divides $2^{60} - 1$.
2: Set $\mathbf{v} := (p_0, \ldots, p_{d-1})$ with $m = p_0 \cdots p_{d-1}$ a factorization of m.
3: Compute $\hat{\mathbf{A}} :=$ `FFFT_2_60`$(\mathbf{v}, A)$ and $\hat{\mathbf{B}} :=$ `FFFT_2_60`$(\mathbf{v}, B)$. ▷ Algorithm 5.2
4: Compute $\hat{\mathbf{C}} := (\hat{\mathbf{A}}_i \times \hat{\mathbf{B}}_i)_{i<m}$
5: **return** `IFFFT_2_60`$(\mathbf{v}, \hat{\mathbf{C}})$ ▷ Algorithm 5.4

Proposition 5.8. *Algorithm 5.5 is correct.*

Proof. The correctness simply follows from Propositions 5.5 and 5.7, and using the fact that $E_\omega(AB) = E_\omega(A)E_\omega(B)$, since $m \geqslant (\deg A + \deg B)/60$. □

For step 1, the actual determination of m has been discussed in [HHL16a, section 3]. In fact it is often better not to pick the smallest possible value for m but a slightly larger one that is also very smooth. Since $2^{60} - 1$ admits many small prime divisors, such smooth values of m usually indeed exist.

5.2 Implementation details

This section presents a practical implementation of the above algorithm, with a practical speedup close to two with respect to the previous work [HHL16a] (the timings are given in section 5.3). Notice that in both cases, DFTs over $\mathbb{F}_{2^{60}}$ represent the bulk of the computation, but the lengths of the DFTs are halved for the new algorithm: they have size $d/30$ with Kronecker substitution and $d/60$ with the Frobenius variant (where d is the degree of the output). Hence, the observed acceleration is due to our new algorithm and not the result of *ad hoc* code tuning or hardware specific optimizations.

We follow Intel's terminology and use the term *quad word* to denote a unit of 64 bits of data. The source code is presented using the C99 standard. In particular, a quad word representing an unsigned integer is considered of type `uint64_t`.

The new polynomial product is implemented in the justinline library of Mathemagix [HLM+02] (`http://www.mathemagix.org`). The source code is freely available from revision 10681 of the SVN server (`https://gforge.inria.fr/projects/mmx/`). The main source files are

- `justinline/src/frobenius_encode_f2_60.cpp` for the Frobenius encoding,
- `justinline/mmx/polynomial_f2_amd64_avx2_clmul.mmx` for the top level functions.

Related test and bench files are also available from dedicated directories of the justinline library. Recall that Mathemagix functions may be easily exported to C++ [HL13a, HL13b].

The implementation targets a modern processor supporting the AVX/AVX2 and CLMUL instruction sets, and an operating system compliant to System V Application Binary Interface. The C++ library numerix of Mathemagix defines wrappers for AVX types. In particular, `avx_uint64_t` represents an SIMD vector of 4 elements of type `uint64_t`. Recall that the platform disposes of 16 AVX registers which must be allocated accurately in order to minimize read and write accesses to the memory.

5.2.1 Packed representations

Polynomials over $\mathbb{F}_2$ are supposed to be given in *packed representation*, which means that coefficients are stored as a vector of contiguous bits in memory. For instance, a

polynomial of degree $\ell - 1$ is stored into $\lceil \ell/64 \rceil$ quad words, starting with the low-degree coefficients: the constant term is the least significant bit of the first word. The last word is suitably padded with zeros.

Reading or writing one coefficient or a range of coefficients of a polynomial in packed representation must be done carefully to avoid invalid memory access. Let P be such a polynomial of type `uint64_t*`. Reading the coefficient P_i of degree i in P is done as `(P[i >> 6] >> (i & 63)) & 1`. However, reading or writing a single coefficient should be avoided as much as possible for efficiency, so we prefer handling ranges of 256 bits. In the sequel the function of prototype

```
void load (avx_uint64_t& d, const uint64_t* P,
           const uint64_t& ℓ, const uint64_t& i,
           const uint64_t& e);
```

returns the $e \leqslant 256$ bits of P starting from i into d; bits beyond position ℓ are considered to be zero.

Implementation of arithmetic operations in $\mathbb{F}_{2^{60}}$ is presented in [HHL16a, section 3.1]. In the sequel we only make use of the function

```
uint64_t f2_60_mul (const uint64_t& a, const uint64_t& b);
```

that multiplies the two elements a and b of $\mathbb{F}_{2^{60}}$ in packed representation.

We also use a packed column-major representation for matrices over $\mathbb{F}_2$. For instance, an 8×8 bit matrix $(\mathbf{M}_{i,j})_{i<8,\,j<8}$ is encoded as a quad word whose $(8j+i)$-th bit is $\mathbf{M}_{i,j}$. Similarly, a $256 \times \ell$ matrix $(\mathbf{M}_{i,j})_{i<256,\,j<\ell}$ may be seen as a pointer `avx_uint64_t* v`, so $\mathbf{M}_{i,j}$ corresponds to the i-th bit of `v[j]`.

5.2.2 Matrix transposition

The Frobenius encoding essentially boils down to matrix transpositions. Our main building block is 256×64 bit matrix transposition. We decompose this transposition in a suitable way with regards to data locality, register allocation and vectorization.

For the computation of general transpositions, we repeatedly make use of the well-known divide and conquer strategy: to transpose an $n \times \ell$ matrix M, where n and ℓ are even, we decompose $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, where A, B, C, D are $n/2 \times \ell/2$ matrices; we swap the anti-diagonal blocks B and C and recursively transpose each block A, B, C, D.

Transposing packed 8×8 bit matrices

The basic task we begin with is the transposition of a packed 8×8 bit matrix. The solution used here is borrowed from [War12, Chapter 7, section 3]. In steps 8 and 9, the anti-diagonal 4×4 blocks are swapped. In steps 10 and 11, the matrix N is seen as four 4×4 matrices whose anti-diagonal 2×2 blocks are swapped. In steps 12 and 13, the matrix N is seen as sixteen 2×2 matrices whose anti-diagonal elements are swapped.

```
uint64_t
packed_matrix_bit_8x8_transpose (const uint64_t& M) {
  uint64_t N = M;
  static const uint64_t mask_4 = 0x00000000f0f0f0f0;
  static const uint64_t mask_2 = 0x0000cccc0000cccc;
  static const uint64_t mask_1 = 0x00aa00aa00aa00aa;
  uint64_t a;
  a = ((N >> 28) ^ N) & mask_4; N = N ^ a;
  a = a << 28; N = N ^ a;
  a = ((N >> 14) ^ N) & mask_2; N = N ^ a;
  a = a << 14; N = N ^ a;
  a = ((N >> 7) ^ N) & mask_1;  N = N ^ a;
  a = a << 7;  N = N ^ a;
  return N;
}
```

All in all, 18 instructions, 3 constants and one auxiliary variable are needed to transpose a packed 8×8 bit matrix in this way. One advantage of the above algorithm is that it admits a straightforward AVX vectorization that we will denote by

```
avx_uint64_t
avx_packed_matrix_bit_8x8_transpose (const avx_uint64_t& M);
```

This routine transposes four 8×8 bit matrices `M[0],M[1],M[2],M[3]` that are packed successively into an AVX register of type `avx_uint64_t`. Notice that this task is *not* the same as transposing a 32×8 or 8×32 bit matrices.

Remark 5.2. The BMI2 technology gives another method for transposing 8×8 bit matrices:

```
uint64_t mask = 0x0101010101010101;
uint64_t N = 0;
for (unsigned i = 0; i < 8; i++)
  N |= _pext_u64 (M, mask << i) << (8 * i);
```

The loop can be unrolled while precomputing the shift amounts and masks, which leads to a faster sequential implementation. Unfortunately this approach cannot be vectorized with the AVX2 technology. Other sequential solutions also exist, based on lookup tables or integer arithmetic, but their vectorization is again problematic. Practical efficiencies are reported in section 5.3.

Transposing four 8×8 byte matrices simultaneously

Our next task is to design a transposition algorithm of four packed 8×8 *byte* matrices simultaneously. More precisely, it performs the following operation on a

packed 32×8 byte matrix:

$$\begin{pmatrix} \mathbf{M}^{(0)} \\ \mathbf{M}^{(1)} \\ \mathbf{M}^{(2)} \\ \mathbf{M}^{(3)} \end{pmatrix} \longrightarrow \begin{pmatrix} (\mathbf{M}^{(0)})^\top \\ (\mathbf{M}^{(1)})^\top \\ (\mathbf{M}^{(2)})^\top \\ (\mathbf{M}^{(3)})^\top \end{pmatrix},$$

where the $\mathbf{M}^{(i)}$ are 8×8 blocks. This operation has the following prototype in the sequel:

```
void avx_packed_matrix_byte_8x8_transpose
  (avx_uint64_t* dest, const avx_uint64_t* src);
```

This function works as follows. First the input `src` is loaded into eight AVX registers `r_0`,...,`r_7`. Each `r_i` is seen as a vector of four `uint64_t`: for `j` $\in \{0, \ldots, 3\}$, `r_0[j]`,...,`r_7[j]` thus represent the 8×8 byte matrix $\mathbf{M}^{(j)}$. Then we transpose these four matrices simultaneously in-register by means of AVX shift and blend operations over 32, 16 and 8 bits entries in the spirit of the aforementioned divide and conquer strategy.

Transposing 256×64 bit matrices

Having the above subroutines at our disposal, we can now present the algorithm to transpose a packed 256×64 bit matrix. The input bit matrix of type `avx_int64_t*` is written $(\mathbf{M}_{i,j})_{i<256,\, j<64}$. The transposed output matrix is written $(\mathbf{N}_{i,j})_{i<256,\, j<64}$ and has type `avx_int64_t*`.

We first compute the auxiliary byte matrix $\mathbf{T}$ as follows:

```
static avx_uint64_t T[64];
for (int i = 0; i < 8; i++) {
  avx_packed_matrix_byte_8x8_transpose (T+8*i, M+8*i);
  for (int k = 0; k < 8; k++)
    T[8*i+k] =
        avx_packed_matrix_bit_8x8_transpose(T[8*i+k]);
}
```

If we write $\mathbf{M}_{i,k:l}$ for the byte representing the packed bit vector $(\mathbf{M}_{i,k}, \ldots, \mathbf{M}_{i,l})$, and $\mathbf{T}^{(i,k:l)}$ for the 8×8 byte matrix given by

$$\mathbf{T}^{(i,k:l)} := \begin{pmatrix} \mathbf{M}_{i,k:l} & \cdots & \mathbf{M}_{i+56,k:l} \\ \vdots & & \vdots \\ \mathbf{M}_{i+7,k:l} & \cdots & \mathbf{M}_{i+63,k:l} \end{pmatrix},$$

then $\mathbf{T}$ contains the following 32×64 byte matrix:

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}^{(0,0:7)} & \mathbf{T}^{(0,8:15)} & \ldots & \mathbf{T}^{(0,56:63)} \\ \mathbf{T}^{(64,0:7)} & \mathbf{T}^{(64,8:15)} & \ldots & \mathbf{T}^{(64,56:63)} \\ \mathbf{T}^{(128,0:7)} & \mathbf{T}^{(128,8:15)} & \ldots & \mathbf{T}^{(128,56:63)} \\ \mathbf{T}^{(192,0:7)} & \mathbf{T}^{(192,8:15)} & \ldots & \mathbf{T}^{(192,56:63)} \end{pmatrix}.$$

Now, for all $0 \leqslant i \leqslant 7$, we load column $8i$ into the AVX register `r_`i. We interpret these registers as forming a 32×8 byte matrix that we transpose in-registers. This transposition is again performed in the spirit of the aforementioned divide and conquer strategy and makes use of various specific AVX2 instructions. We obtain the matrix

$$\left(\begin{array}{cccc|cccc|c} \mathbf{M}_{0,0:7} & \mathbf{M}_{1,0:7} & \cdots & \mathbf{M}_{7,0:7} & \mathbf{M}_{64,0:7} & \mathbf{M}_{65,0:7} & \cdots & \mathbf{M}_{71,0:7} & \cdots \\ \mathbf{M}_{0,8:15} & \mathbf{M}_{1,8:15} & \cdots & \mathbf{M}_{7,8:15} & \mathbf{M}_{64,8:15} & \mathbf{M}_{65,8:15} & \cdots & \mathbf{M}_{71,8:15} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \\ \mathbf{M}_{0,56:63} & \mathbf{M}_{1,56:63} & \cdots & \mathbf{M}_{7,56:63} & \mathbf{M}_{64,56:63} & \mathbf{M}_{65,56:63} & \cdots & \mathbf{M}_{71,56:63} & \cdots \end{array}\right),$$

where each `avx_uint64_t` register contains 4 consecutive columns. We save the registers `r_0`, ..., `r_7` at the addresses $\mathbf{N}$, $\mathbf{N}+4$, $\mathbf{N}+64$, $\mathbf{N}+68$, $\mathbf{N}+128$, $\mathbf{N}+132$, $\mathbf{N}+192$ and $\mathbf{N}+196$.

This operation is then repeated similarly: for each $k = 1, \ldots, 7$, we build a similar 32×8 byte matrix from the columns $k, 8+k, \ldots, 56+k$ of $\mathbf{T}$, and transpose this matrix using the same algorithm. This time the result is saved at the addresses $\mathbf{N}'$, $\mathbf{N}'+4$, $\mathbf{N}'+64$, $\mathbf{N}'+68$, $\mathbf{N}'+128$, $\mathbf{N}'+132$, $\mathbf{N}'+192$ and $\mathbf{N}'+196$, where $\mathbf{N}' := \mathbf{N} + 8k$. This yields an efficient routine for transposing $\mathbf{M}$ into $\mathbf{N}$, whose prototype is given by

```
void packed_matrix_bit_256x64_transpose}
  (uint64_t* N, (const avx_uint64_t*) M);
```

5.2.3 Frobenius encoding

If the input polynomial P has degree less than $\ell \leqslant 60m$ and is in packed representation, then it can also be seen as a $m \times 60$ matrix in packed representation (except a padding with zeros could be necessary to adjust the size).

In this setting, the elements $\tilde{P}_i \in \mathbb{F}_{2^{60}}$ of Algorithm 5.1 are simply read as the rows of the matrix. Therefore, to compute the Frobenius encoding $F_\omega(P)$, we only need to transpose this matrix, then add 4 rows of zeros for alignment (because we store one element of $\mathbb{F}_{2^{60}}$ per quad word), and finally multiply by twiddle factors. This leads to the following implementation:

```
 1 void encode (uint64_t* d, const uint64_t& m,
 2               const uint64_t* P, const uint64_t& ℓ) {
 3 uint64_t c = 1, i = 0, e = 0;
 4 avx_uint64_t v[64]; uint64_t w[256];
 5 while (i < m) {
 6   e = min (m - i, 256);
 7   for (int j = 0; j < 64; j++)
 8     load (v[j], P, ℓ, i + m * j, e);
 9   packed_matrix_bit_256x64_transpose (w, v);
10   for (int j = 0; j < e; j++) {
11     d[i + j] = f2_60_mul (w[j], c);
12     c = f2_60_mul (c, ω); }
13   i += e; }
```

Remark 5.3. To optimize read accesses, it is better to run loop 7 for `j` $< \lceil \ell/m \rceil$ and to initialize the remaining `v[j]` to zero. Indeed, for a product of degree d, we typically multiply two polynomials of degree $\simeq d/2$, which means $\ell < 30m$ when computing the direct transform.

The Frobenius decoding consists in inverting the encoding. The implementation issues being the same, the reader should refer to the source code for further details.

5.3 Timings

In this section, we compare performances between the implementation from the previous section and other reference libraries. As references, we consider in particular the GF2X library [BGTZ08], and the original $\mathbb{F}_{2^{60}}$-FFT from [HHL16a]. We also compare with the additive FFT implementations by Chen et al.: the initial version [CCK+17] using Kronecker substitution, and the improved version [LCK+18, CCK+18] based on Frobenius DFTs.

The platform considered in this work is equipped with an Intel(R) Core(TM) i7-6700 CPU at 3.40 GHz and 32 GB of 2133 MHz DDR4 memory. This CPU features AVX2, BMI2 and CLMUL technologies (family number 6 and model number 94). The platform runs the Stretch GNU Debian operating system with a 64 bit Linux kernel version 4.3. We compile with GCC [GCC87] version 5.4. The precise version of the software to be compared are the following:

- Mathemagix/justinline with svn revision 10681 (this work: Frobenius FFT over $\mathbb{F}_{2^{60}}$). `svn://scm.gforge.inria.fr/svn/mmx`
- gf2x version 1.2 (July 2017). `https://gforge.inria.fr/projects/gf2x/`
- Mathemagix/justinline with svn revision 10663 (previous version: FFT over $\mathbb{F}_{2^{60}}$ with Kronecker). `svn://scm.gforge.inria.fr/svn/mmx`
- bitpolymul with version from september 5, 2017 / commit cfb42ab (additive FFT with Kronecker). `https://github.com/fast-crypto-lab/bitpolymul`
- bitpolymul2 with version from july 2nd, 2019 / commit 09cc6df (Frobenius additive FFT)[1]. `https://github.com/fast-crypto-lab/bitpolymul2`

Frobenius encoding

Concerning the cost of the Frobenius encoding and decoding, the transposition of 8×8 bit matrices (function `packed_matrix_bit_8x8_transpose`) takes about 20 CPU cycles when compiled with the sole `-O3` option. With the additional options `-mtune=native -mavx2 -mbmi2`, the BMI2 version of Remark 5.2 takes about 16 CPU cycles. The vectorized version `avx_packed_matrix_bit_8x8_transpose`)

[1] Comparison with this work was not included in the paper [HLL17].

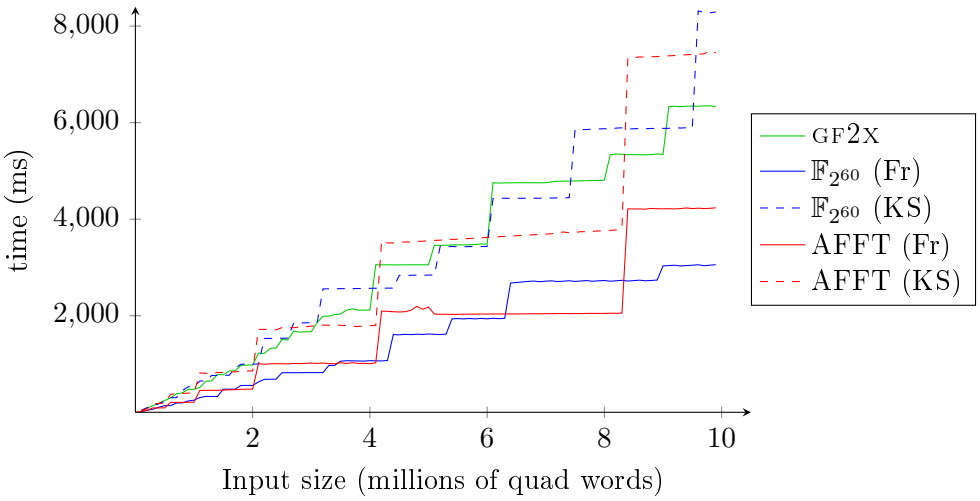


Figure 5.1: Products in $\mathbb{F}_2[X]^{<\ell}$, input size $\lceil \ell/64 \rceil$ quad words, timings in milliseconds.

transposes four packed 8×8 bit matrices simultaneously in about 20 cycles, which makes an average of 5 cycles per matrix.

It is interesting to examine the performance of the sole transpositions made during the Frobenius encoding and decoding (that is discarding products by twiddle factors in $\mathbb{F}_{2^{60}}$). From sizes of a few kilobytes this average cost per quad word is about 8 cycles with the AVX2 technology, and it is about 23 cycles without. Unfortunately the vectorization speed-up is not as close to 4 as we would have liked.

Since the encoding and decoding costs are linear, their relative contribution to the total computation time of polynomial products decreases for large sizes. For two input polynomials in $\mathbb{F}_2[X]$ of 2^{16} quad words, the contribution is about 15%; for 2^{22} quad words, it is about 10%.

Polynomial product

Figure 5.1 gives timings in milliseconds for multiplying two polynomials in $\mathbb{F}_2[X]^{<\ell}$, hence each of input size $\lceil \ell/64 \rceil$ quad words (indicated in abscissa). The results are obtained from `justinline/bench/polynomial_f2_bench.mmx`. The "GF2X" curve corresponds to the library with the same name. The "$\mathbb{F}_{2^{60}}$ (Fr)" curve corresponds to FFTs in $\mathbb{F}_{2^{60}}$ with Frobenius encoding (this work). The "$\mathbb{F}_{2^{60}}$ (KS)" curve corresponds to the previous version with Kronecker substitution [HHL16a]. The "AFFT (KS/Fr)" curves correspond to the additive FFT strategy, with Kronecker substitution [CCK+17] and Frobenius FFT [LCK+18, CCK+18] respectively.

The previous implementation $\mathbb{F}_{2^{60}}$ (KS) was faster than version 1.1 of GF2X, but is now of speed similar to version 1.2. The additive FFT strategy (Kronecker substitution) achieves a noticeable speed-up in favorable cases, but because of its staircase-effect its runtime is roughly similar to the one of GF2X in average.

The new implementation based on the Frobenius FFT is faster than the one

based on Kronecker substitution, with a speed-up that is not far from the factor of 2 predicted by the complexity analysis. Notice that the same is also true for the Additive FFT strategy: the Frobenius version is about twice as fast as the Kronecker substitution version. Also, the two Frobenius-based implementation ($\mathbb{F}_{2^{60}}$ (Fr) and AFFT (Fr)) achieve similar performances. Let us mention that the $\mathbb{F}_{2^{60}}$ (Fr) implementation becomes faster than GF2X for input sizes larger than 2048 quad words ($\lceil \ell/64 \rceil \geqslant 2048$).

Polynomial matrix product

As in [HHL16a], one major advantage of DFTs over the field $\mathbb{F}_{2^{60}}$ is the compactness of the evaluated FFT-representation of polynomials. This makes linear algebra over $\mathbb{F}_2[X]$ particularly efficient: instead of multiplying $r \times r$ matrices over $\mathbb{F}_2[X]^{<\ell}$ naively by means of r^3 polynomial products of degree $< \ell$, we use the standard evaluation-interpolation approach. In our context, this comes down to:

1. computing the $2r^2$ Frobenius encodings followed direct DFTs of all entries of the two matrices to be multiplied
2. performing the $\approx 2\ell/60$ products of $r \times r$ matrices over $\mathbb{F}_{2^{60}}$
3. computing the r^2 inverse DFTs and Frobenius decodings of the so-computed matrix products.

Timings for matrices over $\mathbb{F}_2[X]$ are are reported in Table 5.1; The related code is found in `justinline/bench/matrix_polynomial_f2_bench.mmx`. The row "Evaluation-interpolation" corresponds to the above fast approach (within our implementation). The row "naive" is obtained by doing r^3 polynomial multiplications using GF2X. This confirms the practical gain even for rather small matrices.

Notice that the evaluation-interpolation method can be used similarly with the additive FFT as implemented by BITPOLYMUL. Besides Schönhage's algorithm, GF2X also implements an additive FFT, which is used within the CADO-NFS software [CAD17] for polynomial matrix products. In principle, even Schönhage's algorithm can be combined with an evaluation-interpolation approach for matrix products [Hoe10, Section 2], but the implementation effort would be considerable.

r	1	2	4	8	16	32
Evaluation-interpolation	12	51	212	896	3969	18953
Naive	22	182	1457	11856	92858	745586

Table 5.1: Products of $r \times r$ matrices over $\mathbb{F}_2[X]$, for degree $64 \cdot 2^{16}$, in milliseconds.

5.4 Extension to other finite fields

As an attempt of generalization, let us briefly discuss in which cases the new techniques could apply. This can be useful if one wants to multiply polynomials over

another small finite field (for example in characteristic > 2). Moreover, processor architecture might have increased word size in the future, then there could be better choices of extension field than $\mathbb{F}_{2^{60}}$ to multiply polynomials in $\mathbb{F}_2[X]$.

The field $\mathbb{F}_{2^{60}}$ was initially chosen [HHL16a] because its order $2^{60}-1$ is smooth (a consequence of 60 being smooth), and because the polynomial $(Z^{61}-1)/(Z-1)$ is irreducible over $\mathbb{F}_2$ (which gives a privileged representation of $\mathbb{F}_{2^{60}}$ for efficient arithmetic). "Fortunately", it turned out that the Frobenius FFT also adapted especially well: there is a primitive 61-th root of unity θ, and $\langle\theta\rangle$ has two orbits for the action of the Frobenius map (the trivial orbit $\{1\}$, and the orbit $\{\theta,\dots,\theta^{60}\}$ of size $60 = [\mathbb{F}_{2^{60}} : \mathbb{F}_2]$). Notice that it corresponds to the "full Frobenius action" case from section 4.2.3.

In fact, the reason why $\mathbb{F}_{2^{60}}$ was chosen in the first place is precisely why the Frobenius FFT works so well. First, recall that if $d+1$ is prime and not a divisor of q, then $d+1$ divides q^d-1 by Fermat's theorem, so that there is a primitive $(d+1)$-th root of unity $\theta \in \mathbb{F}_{q^d}$. Also, the polynomial $(Z^{d+1}-1)/(Z-1)$ is irreducible over $\mathbb{F}_q$ if and only if its roots $\theta,\dots,\theta^d$ are conjugates for the action of ϕ_q; in other words if $\langle\theta\rangle$ has two orbits for the action of the Frobenius map (the trivial orbit $\{1\}$, and the orbit $\{\theta,\dots,\theta^d\}$ of size $d = [\mathbb{F}_{q^d} : \mathbb{F}_q]$). Notice that this is also equivalent to q being a generator of $(\mathbb{Z}/(d+1)\mathbb{Z})^\times$, which is easy to check.

Remark 5.4. It is always possible to choose $\theta = Z \bmod \mu(Z)$ where μ is the defining polynomial $\mu(Z) := (Z^{d+1}-1)/(Z-1)$. Indeed, let ξ be a primitive (q^d-1)-th root of unity in $\mathbb{F}_{q^d}$, and let $\alpha := \xi^{(q^d-1)/(d+1)}$. By definition, θ and α are primitive $(d+1)$-th roots, i.e. roots of μ. Since all roots of μ are conjugates under the action of ϕ_q, there is $k \in \mathbb{N}$ such that $\theta = \phi_q^k(\alpha)$. Now let $\zeta := \phi_q^k(\xi)$. By construction, ζ is a primitive (q^d-1)-th root of unity in $\mathbb{F}_{q^d}$ and we have

$$\zeta^{(q^d-1)/(d+1)} = \theta = Z \bmod \mu(Z)\,.$$

In fact, this is how the root $\zeta := Z^{18} + Z^6 + 1 \mod \mu(Z)$ as in (5.3) was found for $\mathbb{F}_{2^{60}}$; initially, the paper [HHL16a] used the root $\xi := Z^3 + Z + 1 \mod \mu(Z)$.

Summarizing, the variant of the Frobenius DFT described in section 5.1.1 will work similarly in the extension $[\mathbb{F}_{q^d} : \mathbb{F}_q]$ if $d+1$ is prime and does not divide q, and q generates $(\mathbb{Z}/(d+1)\mathbb{Z})^\times$. Also, d should be chosen smooth for the FFT over $\mathbb{F}_{q^d}$ to be efficient. For example, candidates for multiplication in $\mathbb{F}_2[X]$ with 128/256/512-bits architectures are:

128-bits $\mathbb{F}_{2^{100}}$

256-bits $\mathbb{F}_{2^{210}}$

512-bits $\mathbb{F}_{2^{460}}$, or $\mathbb{F}_{2^{490}}$ but the order is not as smooth.

Unfortunately, these do not fit as tightly as $\mathbb{F}_{2^{60}}$ in the corresponding machine word. Notice that $\mathbb{F}_{2^{106}}$, $\mathbb{F}_{2^{226}}$, $\mathbb{F}_{2^{466}}$ and $\mathbb{F}_{2^{508}}$ would also work but the order is not smooth (only 3 prime factors < 10000).

Part II

Reduction of polynomials in two variables

Chapter 6

Towards faster polynomial reduction

Contents

Note. *This chapter introduces the formalism common to Chapters 7 and 8; the corresponding results were published in* [HL18a] *and* [HL19a] *respectively.*

In this chapter and the next two, we consider an algebra $\mathbb{A} := \mathbb{K}[X_1, \ldots, X_r]/I$, where I is a finitely generated ideal. For actual computations in $\mathbb{A}$, we have three main tasks:

T1 define a non-ambiguous representation for elements in $\mathbb{A}$;

T2 design a multiplication algorithm for $\mathbb{A}$;

T3 show how to convert between different representations for elements in $\mathbb{A}$.

Fast polynomial arithmetic based on FFT-multiplication allows for a quasi-optimal solution in the univariate case, because the ring $\mathbb{K}[X]$ is principal so

$$I = \langle P^{(0)}, \ldots, P^{(\ell)} \rangle = \gcd(P^{(0)}, \ldots, P^{(\ell)}) \cdot \mathbb{K}[X] \,.$$

Therefore, computations in $\mathbb{A}$ as above reduce to GCD computations and Euclidean divisions, which are quasi-optimal. However, $\mathbb{K}[X_1, \ldots, X_r]$ is no longer principal if $r > 1$ so reduction modulo an ideal of multivariate polynomials is non-trivial.

The most common approach for computations modulo ideals of polynomials is based on Gröbner bases (a short presentation is given next in section 6.1). This immediately solves the first task, using the fact that any polynomial admits a unique normal form modulo a given Gröbner basis [Buc65]. The second task is solved by reducing the product of two polynomials modulo the Gröbner basis. Finally, given a Gröbner basis with respect to a first term ordering, one may use the FGLM algorithm [FGLM93] to compute a reduced Gröbner basis with respect to a second term ordering; algorithms for the corresponding conversions are obtained as a by-product.

Most (if not all) currently known fast algorithms for Gröbner basis computations rely on linear algebra. At this point, one may wonder whether there is an intrinsic reason for this fact, or whether quasi-optimal FFT-based arithmetic might be used to accelerate Gröbner basis computations (after all, this is a problem on polynomials!). Instead of directly addressing this difficult problem, one may investigate whether such accelerations are possible for simpler problems in this area. One good candidate for such a problem is the reduction of a polynomial P with respect to a fixed reduced Gröbner basis $G := (G^{(0)}, \ldots, G^{(n)})$. In that case, the algebra $\mathbb{A}$ is given once and for all, so it becomes a matter of precomputation to obtain G and any other data that could be useful for efficient reductions modulo G.

One step in this direction was made in [Hoe15]. Using relaxed multiplication [Hoe02], it was shown that the reduction of P with respect to G can be computed in quasi-linear time in terms of the size of the equation

$$P = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} + R\,.$$

However, even in the case of bivariate polynomials, this is not necessarily optimal. To see the reason for this fact, consider $\mathbb{A} := \mathbb{K}[X, Y]/I$, where I is an ideal generated by two generic polynomials of total degree δ. Then $\dim_{\mathbb{K}} \mathbb{A} = \delta^2$, but the Gröbner basis for I with respect to the usual total degree ordering contains $\delta + 1$ polynomials with $\Theta(\delta^2)$ coefficients. This means that we need $\Theta(\delta^3)$ space, merely to write down G. One crucial prerequisite for even faster algorithms is therefore to design a terser representation for Gröbner bases; section 6.2 sketches the main ideas of the proposed solution.

Structure of this chapter. First of all, section 6.1 gives a general presentation of Gröbner bases, to introduce the required definitions and provide some context for the contributions. Then, section 6.2 explains the techniques at the core of the new algorithms. Also, section 6.3 shows briefly how the results of Chapters 7 and 8 compare with each other. Finally, section 6.4 reviews the classical algorithmic tools that will be needed in the complexity analysis.

6.1 Introduction to Gröbner basis theory

This section aims to recall generalities about Gröbner bases and some useful notations. More details can be found in [GG13, Chapter 21] or [BW93, CLO92] and references therein.

6.1.1 Definition and basic properties

Let $\mathcal{M} := \{X_1^{i_1} \cdots X_r^{i_r} : i_1, \ldots, i_r \in \mathbb{N}\}$ denote the set of *monomials* in r variables. A *monomial ordering* $\prec$ is a total well-order on $\mathcal{M}$ that is compatible with multiplication; that is $M_1 \preccurlyeq M_1 M_2$ for any monomials $M_1, M_2 \in \mathcal{M}$.

Given a polynomial in r variables $P = \sum_{M \in \mathcal{M}} P_M M \in \mathbb{K}[X_1, \ldots, X_r]$, its *support* $\operatorname{supp} P$ is the (finite) set of monomials $M \in \mathcal{M}$ with $P_M \neq 0$. If $P \neq 0$, then $\operatorname{supp} P$ admits a maximal element for $\prec$ that is called its *leading monomial* denoted by $\operatorname{lm}(P)$. If $M \in \operatorname{supp} P$, then we say that $P_M M$ is a *term* in P; in particular the *leading term* is the term corresponding to the leading monomial ($\operatorname{lt}(P) := P_{\operatorname{lm}(P)} \operatorname{lm}(P)$).

Given a tuple $A := (A^{(0)}, \ldots, A^{(n)})$ of polynomials in $\mathbb{K}[X_1, \ldots, X_r]$, we say that P is *reduced* with respect to A if $\operatorname{supp} P$ contains no monomial that is a multiple of the leading monomial of one of the $A^{(i)}$. Otherwise, a family $(Q^{(0)}, \ldots, Q^{(n)}, R)$ such that

$$P = Q^{(0)} A^{(0)} + \cdots + Q^{(n)} A^{(n)} + R \text{ and } R \text{ is reduced w.r.t. } A \tag{6.1}$$

is called an *extended reduction* of P; the polynomials $Q^{(i)}$ are called the *quotients* and R is the *remainder*. In this case, we also say that P *reduces to* R. Such an extended reduction always exist, because of the following naive algorithm (notice that it is not unique because of the arbitrary choice in line 4) :

Algorithm 6.1. Naive reduction of a multivariate polynomial

Input: A polynomial P and a tuple of polynomials $A := (A^{(0)}, \ldots, A^{(n)})$.
Output: An extended reduction $(Q^{(0)}, \ldots, Q^{(n)}, R)$ as in (6.1).

1: Set $(Q^{(0)}, \ldots, Q^{(n)}, R) := (0, \ldots, 0)$ and $S := P$
2: **while** $S \neq 0$ **do** ▷ $\operatorname{lt}(S)$ decreases strictly at each step
3: **if** $\exists i$: $\operatorname{lm}(A^{(i)})$ divides $\operatorname{lm}(S)$ **then**
4: Choose one such i.
5: Set $T := \operatorname{lt}(S) / \operatorname{lt}(A^{(i)})$.
6: Update $Q^{(i)} \mathrel{+}= T$ and $S \mathrel{-}= T A^{(i)}$.
7: **else**
8: Update $R \mathrel{+}= \operatorname{lt}(S)$ and $S \mathrel{-}= \operatorname{lt}(S)$.
9: **end if**
10: **end while**
11: **return** $(Q^{(0)}, \ldots, Q^{(n)}, R)$

A Gröbner basis of an ideal I is a finite family $G := (G^{(0)}, \ldots, G^{(n)}) \subset I$ such that if P is reduced with respect to G and $P \in I$, then $P = 0$. In particular, it implies that the remainder is unique and we call it the *normal form* of P: if

$$P = Q^{(0)} G^{(0)} + \cdots + Q^{(n)} G^{(n)} + R = \tilde{Q}^{(0)} G^{(0)} + \cdots + \tilde{Q}^{(n)} G^{(n)} + \tilde{R}$$

are two extended reductions of P, then

$$R - \tilde{R} = (\tilde{Q}^{(0)} - Q^{(0)}) G^{(0)} + \cdots + (\tilde{Q}^{(n)} - Q^{(n)}) G^{(n)}$$

is an element of I that is reduced with respect to G, that is $R - \tilde{R} = 0$. Also, any polynomial $P \in I$ reduces to 0 with respect to G; this provides a test for ideal membership.

Remark 6.1. There is another definition of Gröbner bases: a Gröbner basis is a finite family $G := (G^{(0)}, \dots, G^{(n)})$ such that the ideals $\langle \mathrm{lt}(G^{(0)}), \dots, \mathrm{lt}(G^{(n)}) \rangle$ and $\langle (\mathrm{lt}(P))_{P \in I} \rangle$ coincide. For polynomials with coefficients in a field, the two definitions are equivalent. For polynomials with coefficients in a ring, they correspond respectively to the notions of *strong* and *weak* Gröbner bases [Möl88].

The similar concept of standard bases was introduced by Hironaka [Hir64] for multivariate power series; the definition of Gröbner bases as above is due to Buchberger [Buc65], he named them after his thesis advisor. In both cases, the definition was accompanied by a proof that such bases actually exist.

6.1.2 Classical algorithms

The first algorithm to compute Gröbner bases, due to Buchberger [Buc65], is as follows. First, for any $A, B \in \mathbb{K}[X_1, \dots, X_r]$, define their S-polynomial

$$S(A, B) := \frac{\mathrm{lcm}(\mathrm{lm}(A), \mathrm{lm}(B))}{\mathrm{lt}(A)} A - \frac{\mathrm{lcm}(\mathrm{lm}(A), \mathrm{lm}(B))}{\mathrm{lt}(B)} B \,.$$

Then, initialize a partial basis G with the generators of the ideal (given as input). Now, compute all S-polynomials for elements of G and reduce them with respect to G; the remainders that are not zero are added to G. Finally, repeat this last step until all S-polynomials reduce to 0. More formally (see [GG13, Theorem 21.34] for a correctness proof):

Algorithm 6.2. Buchberger's algorithm

Input: A family of polynomials $P^{(0)}, \dots, P^{(\ell)}$.
Output: A Gröbner basis of the ideal $I := \langle P^{(0)}, \dots, P^{(\ell)} \rangle$

1: Set $G := (P^{(0)}, \dots, P^{(\ell)})$.
2: **while** `true` **do**
3: Write $G = (G^{(0)}, \dots, G^{(k)})$.
4: **for** $0 \leqslant i < j \leqslant k$ **do**
5: Set $R := S(G^{(i)}, G^{(j)}) \operatorname{rem} G$.
6: **if** $R \neq 0$ **then** add R to G.
7: **end if**
8: **end for**
9: **if** $\#G = k$ **then return** G. ▷ No S-polynomial was added
10: **end if**
11: **end while**

Refinements of this algorithm reduce the number of critical pairs to be considered at each step, by eliminating the S-polynomials that are known *a priori* to reduce to 0 [Buc79]. Also, it was noticed that the runtime of the algorithm strongly depends on

the order in which the pairs are considered; there are different strategies to choose an appropriate order [GMN+91].

Faugère's F4 algorithm [Fau99] improves this again by expressing the reduction of S-polynomials as a linear algebra problem. This allows us to handle several S-polynomials at once, and to implement the algorithm more efficiently (linear algebra is easier to optimize than polynomial arithmetic, especially in the sparse case). In this algorithm, many S-polynomials are still reduced to 0, then the F5 algorithm [Fau02] aims to spare some of these computations with additional criteria to eliminate more critical pairs; a reference implementation is found in the FGb software [Fau10]. At the time of writing, many computer algebra systems, such as Macaulay2 [GS], Magma [BCP97] or Maple [MGH+05] implement at least one of F4 or F5 algorithms. Other systems like Singular [DGPS17] feature a highly optimized implementation of Buchberger's algorithm.

Remark 6.2. The monomial ordering $\prec$ plays also an important role in the complexity of Gröbner basis algorithms. In general, the degree reverse lexicographic order gives the fastest computation, while the lexicographic order is the hardest to solve [BS88]. On the other hand, the lexicographic basis often provides the most information about the actual solutions of the system. In practice, to compute the lexicographic basis, one rather starts with the degree reverse lexicographic one, then uses a Gröbner walk [CKM97] or FGLM algorithm [FGLM93] to change the monomial ordering.

Although the problem of computing a Gröbner basis requires exponential space in the worst case [May89], it is actually tractable for many practical instances, and the computer algebra systems mentioned previously generally give a result within acceptable time. In fact, Faugère's algorithms are very efficient if the system has sufficient regularity. In this case, as mentioned in section 1.2.2, we have bounds of

$$O\left(rD\binom{r+D}{D}^{\Omega}\right) \quad \text{and} \quad O\left(\frac{r(3\delta^3)^r}{\delta}\right) \tag{6.2}$$

field operations for F4 and F5 respectively [BFS14], where D is the Macaulay bound, Ω is the exponent of matrix multiplication, and δ is the degree of the input polynomials. Notice that this is polynomial in terms of the expected output size.

6.1.3 Case of bivariate systems

The zero set of a bivariate polynomial in $\mathbb{R}[X, Y]$ is a plane algebraic curve; then a bivariate system corresponds to the intersection of two curves. The solution of such systems has been extensively studied from a numerical point of view, see for example [BK12, GVK96, GVN02, DET09]. From a complexity point of view, a bound of $O(\delta^8 + \delta^7\tau)$ bit operations has been established in [ES12], where τ is the bit size of input coefficients). The bound was later improved to $\tilde{O}(\delta^6 + \delta^5\tau)$ deterministic, or $\tilde{O}(\delta^5 + \delta^4\tau)$ probabilistic Las Vegas [BLM+16]. For non-singular

solutions, a better bound of $\tilde{O}(\delta^{(\Omega+7)/2}+\delta^{(\Omega+5)/2}\tau)$ bit operations (deterministic), or $\tilde{O}(\delta^{4+\epsilon}+\delta^{3+\epsilon}\tau)$ (probabilistic) has been established in [LMS13]. Also, the algebraic complexity is $\tilde{O}(\delta^3)$ by this last paper.

It is also interesting to observe what the bounds (6.2) for Gröbner basis algorithms become in this case. Setting $r := 2$, we obtain $O(\delta^{2\Omega+1})$ for F4 and $O(\delta^5)$ for F5 (experimentally, the actual runtime is in fact slightly above cubic). Similarly, the bound for the Kronecker solver [GLS01] (geometric resolution) becomes $\tilde{O}(\delta^6)$, or $O(\delta^{3+\epsilon})$ for the variant from [HL18b]. Notice that the latter bound assumes fast modular composition, which impairs its practicality as mentioned earlier. Since elimination methods and geometric resolution are based on this tool, recall the bound $O(\delta^{(3-1/\Omega)(1+\epsilon)})$ for the generic bivariate resultant [Vil18]. An implementation of this last algorithm has been proposed in [HNS19].

6.2 Reduction of large polynomials in two variables

In this chapter and the next two, the polynomials defining I have a fixed number of variables and their degrees becomes large. (Another possibility would be to consider fixed degrees and an increasing number of variables, but polynomial arithmetic is less efficient in this setting.) More precisely, we will only consider the first non-trivial case of two variables X, Y. We then wish to compute in the ring $\mathbb{A} := \mathbb{K}[X, Y]/I$.

In the following, $G := (G^{(0)}, \ldots, G^{(n)})$ is a Gröbner basis of I with respect to a (possibly weighted) degree lexicographic order, and ordered such that the leading monomials have increasing degree in X (and decreasing degree in Y).

Recall that a major obstruction in designing quasi-optimal algorithms is the size of the Gröbner basis: if I is generated by two generic polynomials of total degree δ, then $\dim_{\mathbb{K}} \mathbb{A} = \delta^2$, but we need $\Theta(\delta^3)$ space merely to write down G. As shown in Chapters 7 and 8, it is possible to compress the relevant information within $\tilde{O}(\delta^2)$ space. For both chapters, the compression relies essentially on the same techniques. This section aims to give the intuition behind these new techniques.

6.2.1 The dichotomic selection strategy

When reducing a multivariate polynomial, its terms are reduced one after the other against some basis element. As mentioned in Algorithm 6.1, there may be several possibilities to choose this basis element. When this happens, the usual strategy is to select the first valid choice, however this strategy gives quotients all with roughly the same degree. It would be better if we could control the degrees of the quotients. Ideally, we want to ensure that most quotients are very small and only a few have potentially high degree (the exact bounds may be different depending on the situation).

To do so, we assume that the leading monomials of the Gröbner basis have a certain regularity, and we introduce the so-called *dichotomic selection strategy*. Recall that the 2-*adic valuation* of an integer i is the largest $\lambda \in \mathbb{N}$ such that 2^λ divides i; the usual notation is $\lambda = \mathrm{val}_2(i)$. The idea of the dichotomic selection

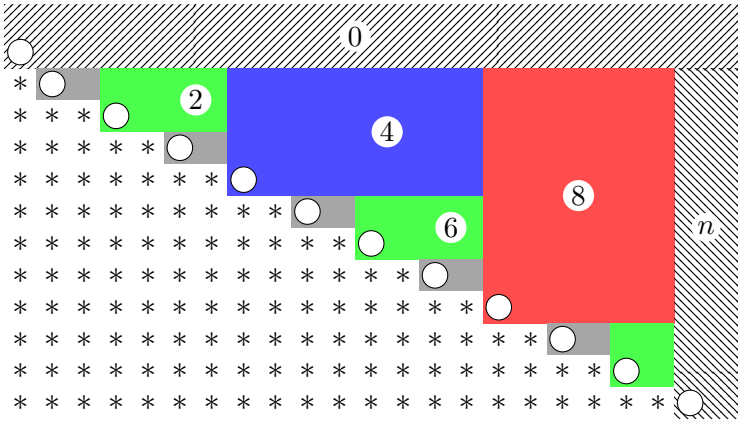


Figure 6.1: The dichotomic selection strategy.

strategy is to reduce each monomial preferably against one end of the Gröbner basis ($G^{(0)}$ or $G^{(n)}$), or the $G^{(i)}$ such that i has the highest 2-adic valuation. More precisely, the selection strategy is determined by the following function Φ_G on terms:

$$\Phi_G(\tau) := \begin{cases} (0, \tau/\operatorname{lt}(G^{(0)})) & \text{if } \operatorname{lt}(G^{(0)}) \text{ divides } \tau \\ (n, \tau/\operatorname{lt}(G^{(n)})) & \text{if } \operatorname{lt}(G^{(n)}) \text{ divides } \tau \text{ (and } \operatorname{lt}(G^{(0)}) \text{ does not)} \\ (i, \tau/\operatorname{lt}(G^{(i)})) & \text{if } \operatorname{lt}(G^{(i)}) \text{ divides } \tau \text{ with } \operatorname{val}_2(i) \text{ maximal} \\ (-1, \tau) & \text{if no } \operatorname{lt}(G^{(i)}) \text{ divides } \tau \end{cases}$$

Notice that this definition is non-ambiguous: there is only one index i with $\operatorname{val}_2(i)$ maximal such that $\operatorname{lt}(G^{(i)})$ divides τ. Indeed, if $i < j$ have the same valuation λ, then there is some k with $i < k < j$ and $\operatorname{val}_2(k) > \lambda$. Moreover, if $\operatorname{lt}(G^{(i)})$ and $\operatorname{lt}(G^{(j)})$ both divide τ, then so does $\operatorname{lt}(G^{(k)})$, because

$$\deg_X \tau \geqslant \deg_X \operatorname{lt}(G^{(j)}) \geqslant \deg_X \operatorname{lt}(G^{(k)}),$$
$$\deg_Y \tau \geqslant \deg_Y \operatorname{lt}(G^{(i)}) \geqslant \deg_Y \operatorname{lt}(G^{(k)}).$$

This dichotomic selection strategy is illustrated in Figure 6.1. White dots (◯) represent the leading term of each basis element. The asterisks (∗) below the stair denote monomials that are already in normal form with respect to G (i.e. the canonical basis of $\mathbb{A}$). Finally, the areas above the stair are the sets of terms that will be reduced against each given basis element: the area labelled with the number i contains the monomials that are reduced against $G^{(i)}$.

6.2.2 Truncated Gröbner basis elements

The second ingredient is to *truncate to an appropriate precision* each basis element, according to the degree bounds established before. The goal is to keep as little information as possible and still be able to perform the operation efficiently. For example over the integers, we may notice that the quotient

$$125\,231\,546\,432 \text{ quo } 12\,358\,748\,151 = 10$$

depends only on the three most significant digits of each operand.

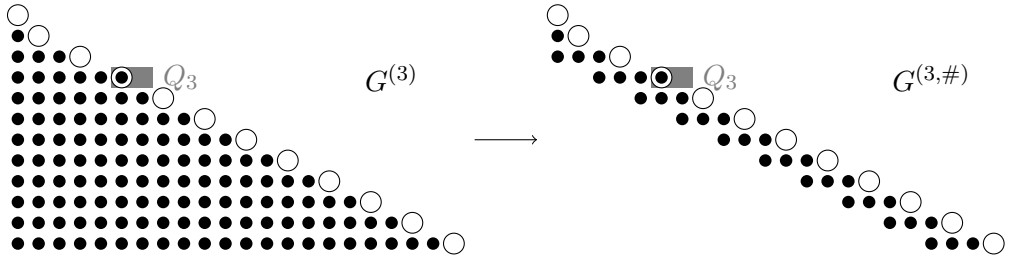


Figure 6.2: Truncated Gröbner basis elements.

Similarly, consider an extended reduction of multivariate polynomials as in (6.1). If we know that $\deg Q^{(i)} \leqslant \delta$, then $Q^{(i)}$ depends only on the terms of $G^{(i)}$ with degree at least $\deg G^{(i)} - \delta$ and the remaining terms may be discarded. This allows to store significantly fewer coefficients, as shown for example in Figure 6.2. Black dots (•) represent the terms in the basis element (here $G^{(3)}$) and its truncated analogue (denoted $G^{(3,\#)}$). Again, white dots represent the leading monomials of each $G^{(i)}$.

These truncated elements constitute the first part of the compressed representation of the Gröbner basis.

6.2.3 Rewriting the equation

The third idea is to *rewrite equation* (6.1). to ensure that the final result is correct despite the aforementioned truncations. This is done by keeping track of the relations that exist among the $G^{(i)}$: recall that G contains far more coefficients than the two generators of I, so there must be some redundancy.

For example, Figure 6.3 shows the substitutions that happen in the algorithm of Chapter 7. Initially, equation $P = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} + R$ involves every element in the basis. Using linear combinations among the basis elements, about half of the terms are replaced, to keep only those with even indices (plus $0, 1, n$ for technical reasons). The coefficients in the linear combinations are polynomials $C^{(1,i,j)} \in \mathbb{K}[X, Y]$ of small degree δ. Then, the number of terms is halved again and only indices that are multiples of 4 remain; now the coefficients $C^{(2,i,j)}$ are a bit larger, with degree 2δ. Repeating this $\lceil \log n \rceil$ times, we are left with the formula

$$P = S^{(0)}G^{(0)} + S^{(1)}G^{(1)} + S^{(n)}G^{(n)} + R\,,$$

where $S^{(0)}, S^{(1)}, S^{(n)}$ are linear combinations of $Q^{(0)}, \ldots, Q^{(n)}$. Now this equation has sufficiently few terms to be evaluated directly. The substitutions used in the algorithm of Chapter 8 have a different shape but they serve the same purpose.

The collection of coefficients $C^{(\ell,i,j)}$ as above allow to compute $S^{(0)}, S^{(1)}, S^{(n)}$ from $Q^{(0)}, \ldots, Q^{(n)}$. These coefficients constitute the second part of the compressed representation of the Gröbner basis.

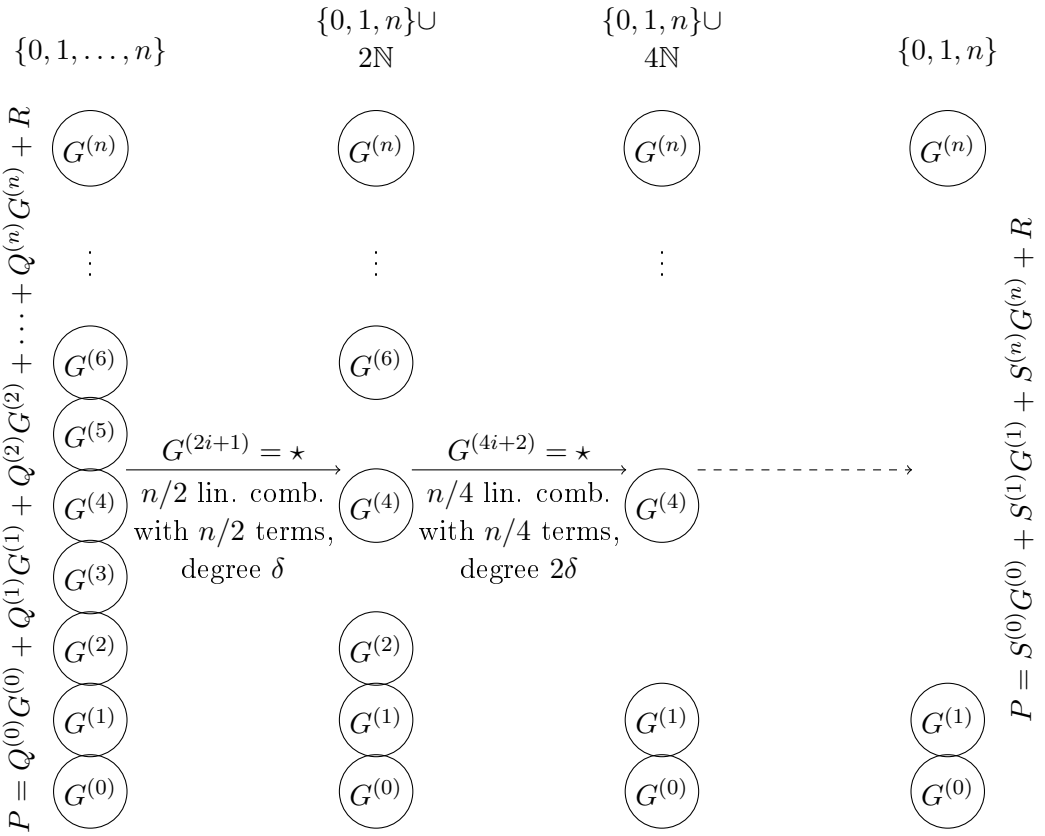


Figure 6.3: Rewriting the equation.

6.3 Summary of the results

The results from chapters 7 and 8 may seem similar at first: both give reduction algorithms with $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A} + |P|)$ complexity, and both use the techniques described in the previous section. However, there are certain fundamental differences that need to be outlined.

Chapter 7 defines a class of so-called *vanilla Gröbner bases*, that verify certain regularity assumptions for the reduction algorithm to work. Experimentally, it turns out that for many types of input polynomials and many monomial orderings, the resulting basis is vanilla. Such bases admit a *terse representation* that may be precomputed once, then it becomes faster to obtain the normal form of any given polynomial. Since this precomputation is rather expensive, one must perform many operations in $\mathbb{A}$ before the initial cost is amortized.

Chapter 8 considers only the degree-lexicographic order, and requires the support of the input polynomials to match this ordering. In this case, the corresponding basis is *not* vanilla and has no terse representation. Yet the structure of the system allows us to construct a *concise representation*, with the same advantages. The major difference is that obtaining the concise representation is possible with quasi-optimal complexity from the input polynomials.

Moreover, the structures of *terse* and *concise* representations are slightly different. The design of the reduction algorithm also varies between the two settings, in fact the algorithm from Chapter 8 is more complicated. These details will be treated more extensively in the corresponding chapters.

Summarizing, the results from chapters 7 and 8 are in fact complementary. On the one hand, the algorithm from Chapter 7 applies to a wider variety of settings, but it requires expensive precomputation. On the other hand, the result of Chapter 8 is considerably stronger because there is no precomputation this time, but its scope is more restrictive. Notice also that the hypotheses of the two chapters cannot be satisfied simultaneously.

6.4 Algorithmic prerequisites

In this section, we quickly review some fundamental operations on polynomials that will be useful for the reduction algorithms in the next chapters. Notice that complexity results presented in this section are not specific to the bivariate case.

6.4.1 Polynomial multiplication

Recall the notation $\mathsf{M}(d)$ for the cost of multiplying two dense univariate polynomials of degree d in $\mathbb{K}[X]$. As seen in Section 2.3.3, multiplication of "block" polynomials in several variables reduces to the univariate case through Kronecker substitution. However, we need to manipulate polynomials with more general supports, typically the truncated bivariate polynomials from section 6.2.2.

Such polynomials are in fact dense polynomials in $\mathbb{K}[X,Y]$ whose supports are contained in sets of the form $\mathcal{S}_{l,h} := \{M \in \mathcal{M} : l \leqslant \deg M \leqslant h\}$. Modulo the change of variables $X^aY^b \mapsto T^{h-a-b}U^b$, such a polynomial can be rewritten as $P(X,Y) = \tilde{P}(T,U)$, where $\tilde{P} \in \mathbb{K}[T,U]^{\deg_T \leqslant h-l, \deg_U \leqslant h}$ is a block polynomial. Notice that the block has size $(h-l+1)\times(h+1)$, which is essentially the size of $\mathcal{S}_{l,h}$. For a product of two truncated polynomials with a support of size d, this means that the product can again be computed in time $O(\mathsf{M}(d))$.

6.4.2 Relaxed multiplication

For the above polynomial multiplication algorithms, we assume that the input polynomials are entirely given from the outset. In specific settings, the input polynomials may be only partially known at some point, and it can be interesting to anticipate the computation of the partial output. This is particularly true when working with formal power series $f = f_0 + f_1 z + \cdots \in \mathbb{K}[[z]]$ instead of polynomials, where it is common that the coefficients are given as a stream.

In this so-called "relaxed" (or "online") computation model, the coefficient $(fg)_d$ of a product of two series $f, g \in \mathbb{K}[[z]]$ must be output as soon as $f_0, \ldots, f_d$ and $g_0, \ldots, g_d$ are known. This model has the advantage that subsequent coefficients

$f_{d+1}, f_{d+2}, \ldots$ and $g_{d+1}, g_{d+2}, \ldots$ are allowed to depend on the result $(fg)_d$. This often allows us to solve equations involving power series f by rewriting them into *recursive equations* of the form $f = \Psi(f)$, with the property that the coefficient $\Psi(f)_{d+1}$ only depends on earlier coefficients $f_0, \ldots, f_d$ for all d. For instance, in order to invert a power series of the form $1 + zg$ with $g \in \mathbb{K}[[z]]$, we may take $\Psi(f) = 1 - zfg$. Similarly, if $\mathbb{K}$ has characteristic zero, then the exponential of a power series $g \in \mathbb{K}[[z]]$ with $g_0 = 0$ can be computed by taking $\Psi(f) = 1 + \int fg'$.

From a complexity point of view, let $\mathsf{R}(d)$ denote the cost of the relaxed multiplication of two polynomials of degree $< d$. The relaxed model prevents us from directly using fast "zealous" multiplication algorithms from the previous section; those being typically based on FFT-multiplication. Fortunately, it was shown in [Hoe02, FS74] that

$$\mathsf{R}(d) = O\left(\mathsf{M}(d) \log d\right). \tag{6.3}$$

This relaxed multiplication algorithm admits the advantage that it may use any zealous multiplication as a black box. Through the direct use of FFT-based techniques, the following bound has also been established in [Hoe14]:

$$\mathsf{R}(d) = d \log d e^{O\left(\sqrt{\log \log d}\right)}.$$

We make the same usual assumptions ($\mathsf{R}(d)/d$ is increasing and $\mathsf{R}(2d) = O\left(\mathsf{R}(d)\right)$) as for classical multiplication. This implies in particular that $\mathsf{R}(d) + \mathsf{R}(e) \leqslant \mathsf{R}(d+e)$. It is also natural to assume that $\mathsf{M}(d) \leqslant \mathsf{R}(2d)$, since ordinary multiplications can be done in a relaxed manner.

6.4.3 Multivariate polynomial reduction

The computation of an extended reduction as in Algorithm 6.1 is a good example of a problem that can be solved efficiently using relaxed multiplication and recursive equations [Hoe15]. With an appropriate change of variable and Kronecker substitution, we may transform $P \in \mathbb{K}[X_1, \ldots, X_r]$ into $\tilde{P} \in \mathbb{K}[Z]$ where the coefficients of P appear in $\tilde{P}$ in decreasing order with respect to $\prec$. In other words, there is a decreasing function $\sigma_P : \mathcal{M} \to \mathbb{Z}$ such that $\sigma_P(\operatorname{lm} P) = 0$ and $P_M = \tilde{P}_{\sigma_P(M)}$. Now, the extended reduction as in equation (6.1) may be rewritten as a recursive equation on power series

$$(\tilde{Q}^{(0)}, \ldots, \tilde{Q}^{(n)}, \tilde{R}) = \Psi_{P,A}(\tilde{Q}^{(0)}, \ldots, \tilde{Q}^{(n)}, \tilde{R}).$$

For a multivariate polynomial T with dense support of any of the types discussed in section 6.4.1, let $|T|$ denote a bound for the size of its support. In this case, the extended reduction can be computed by [Hoe15] in time

$$\mathsf{R}(|Q^{(0)}A^{(0)}|) + \cdots + \mathsf{R}(|Q^{(n)}A^{(n)}|) + O(|R|). \tag{6.4}$$

This implies in particular that the extended reduction can be computed in quasi-linear time in the size of the equation $P = Q^{(0)}A^{(0)} + \cdots + Q^{(n)}A^{(n)} + R$. However, recall that this equation is in general much larger than the input polynomial P.

As mentioned in section 6.2.1, there are various *selection strategies* that may lead to different quotients, and in particular the size of their support depends on the chosen strategy. The initial formulation [Hoe15] assumed the simple strategy "select the first valid choice"; *mutatis mutandis*, the results remain correct with any selection strategy (in particular the dichotomic one from section 6.2.1).

Remark 6.3. The results from [Hoe15] were actually stated for more general types of supports, and not only those discussed in section 6.4.1. However, it relies on a multivariate generalization of the relaxed multiplication, so bound (6.4) is only valid with R as in (6.3).

On the other hand, with sufficiently dense supports (like the truncated bivariate polynomials from section 6.2.2), the equivalence with univariate arithmetic as in the beginning of this subsection is more straightforward, then any fast relaxed algorithm could be used.

For comparison, assume again the setting of an ideal generated by two bivariate polynomials of degree δ. As a naive algorithm, one may precompute the matrix of size $O(\delta^2) \times O(\delta^2)$ corresponding to the linear map of reduction to normal form (assuming input of degree at most 2δ for example). Then actually computing a normal form boils down to a matrix-vector product of cost $O(\delta^4)$. Since the product of two normal forms can be computed in quasi-linear time $\tilde{O}(\delta^2)$, it follows that multiplications in $\mathbb{A}$ take time $O(\delta^4)$. Similarly, changes of monomial orderings lead to $\delta^2 \times \delta^2$-matrices for representing the corresponding base changes. Naive conversions can then be performed in time $O(\delta^4)$.

Summarizing, the naive algorithm has complexity $O(\delta^4)$ for each of the three tasks given at the beginning of this chapter. The bound (6.4) reduces the complexity to $\tilde{O}(\delta^3)$. The next two chapters will give $\tilde{O}(\delta^2)$ algorithms (which is quasi-optimal), under certain regularity assumptions.

Chapter 7

Vanilla Gröbner bases in two variables

Contents

Note. *The results in this chapter were published in* [HL18a]. *A proof-of-concept implementation of the algorithms for* SageMath *is available at* https://hal.archives-ouvertes.fr/hal-01770408/file/implementation.zip.

This chapter presents a first situation where it is actually possible to perform polynomial reductions with quasi-optimal complexity. For simplicity, we will restrict our attention to bivariate polynomials and to ideals that satisfy suitable regularity conditions; the concept of a "vanilla Gröbner basis" captures the assumptions that are needed for the new algorithms.

For such bases, it is possible to precompute a more compact description called the *terse representation*, that holds all necessary information in $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$ space. Given this terse representation, a polynomial of degree d can be reduced in normal form in $\tilde{O}(d^2 + \dim_{\mathbb{K}} \mathbb{A})$ operations. In particular, multiplication in $\mathbb{A}$ can be done in time $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$, which is intrinsically quasi-optimal. Similarly, one can convert between normal forms with respect to vanilla Gröbner bases for different monomial orderings in time $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$: the idea is based on a Gröbner walk [CKM97] with a logarithmic number of intermediate monomial orderings.

7.1 Vanilla Gröbner bases

As mentioned earlier, various monomial orderings are suitable for Gröbner basis computations. It is convenient to restrict our attention to a specific type of bivariate monomial ordering; the following order will allow us to explicitly describe certain Gröbner stairs and to explicitly compute certain dimensions.

Definition 7.1. Let $k \in \mathbb{N} \setminus \{0\}$. We define the *k-degree* of a monomial $X^a Y^b$ with $a, b \in \mathbb{N}$ by

$$\deg_k(X^a Y^b) = a + kb\,.$$

Similarly, the *k-order* is the monomial order $\prec_k$ such that

$$X^a Y^b \prec_k X^u Y^v \Leftrightarrow \begin{cases} \text{either} & a + kb < u + kv\,, \\ \text{or} & a + kb = u + kv \text{ and } a < u\,. \end{cases}$$

The k-order $\prec_k$ is also known as the weighted degree lexicographic order for the weight vector $(1, k)$. Similarly, $\prec_1$ corresponds to the usual total degree order.

In the following, k is fixed and $G = (G^{(0)}, \ldots, G^{(n)})$ is the reduced Gröbner basis of some zero-dimensional ideal $I \subset \mathbb{K}[X, Y]$ with respect to the k-order $\prec_k$. Also, the *degree* of the ideal denotes the dimension of $\mathbb{A} := \mathbb{K}[X, Y]/I$ as a $\mathbb{K}$ vector space, for simplicity we define $D := \dim_{\mathbb{K}} \mathbb{A}$.

7.1.1 Vanilla Gröbner stairs

Let $\mathcal{N}_G$ be the set of monomials $X^a Y^b$ that are in normal form with respect to G. In other words, $\mathcal{N}_G$ corresponds to the set of D monomials "under the Gröbner stair". For a sufficiently generic ideal of degree D, we expect $\mathcal{N}_G$ to consist exactly of the smallest D elements of $\mathcal{M}$ with respect to $\prec_k$.

Definition 7.2. We say that the leading monomials of G form a *vanilla Gröbner stair* if $\mathcal{N}_G$ coincides with the set $\mathcal{M}_{k,D}$ of the D smallest elements of $\mathcal{M}$ for $\prec_k$.

Figure 7.1 shows an example of a Gröbner basis whose leading monomials form a vanilla Gröbner stair (white dots (○) represent the leading monomials of G, and asterisks ($*$) represent monomials in $\mathcal{N}_G$). We observe that the stair admits almost constant slope k. In fact, the set $\mathcal{M}_{k,D}$ can be described explicitly:

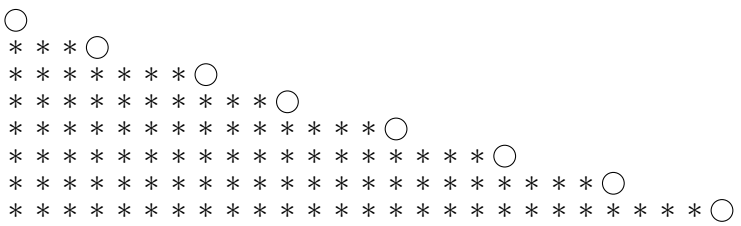


Figure 7.1: A vanilla Gröbner stair with respect to $\prec_4$.

Proposition 7.1. *Let I be an ideal of degree D with Gröbner basis G for $\prec_k$ with $k \geqslant 2$. Assume that the leading monomials of G form a vanilla Gröbner stair and define*[1]

$$\begin{aligned} n &:= \left\lceil \frac{\sqrt{8D/k+1}-1}{2} \right\rceil, \\ u &:= D - k\frac{n(n-1)}{2}, \\ q &:= u \operatorname{quo} n, \\ r &:= u \operatorname{rem} n. \end{aligned}$$

Then G has $n+1$ elements $G^{(0)}, \ldots, G^{(n)}$ and for $0 \leqslant i \leqslant n$, the leading monomial of $G^{(i)}$ (denoted by M_i) can be expressed in terms of n, k, q, r. Assuming the basis elements are ordered such that the M_i's have increasing degree in the variable X, we have:

- $M_0 = Y^n$.
- *For all* $i \in \{1, \ldots, r\}$, $M_i = X^{q+k(i-1)+1}Y^{n-i}$.
- *For all* $i \in \{r+1, \ldots, n\}$, $M_i = X^{q+k(i-1)}Y^{n-i}$.

Proof. With this expression of M_i, we first notice that this sequence $M_0, \ldots, M_n$ can indeed be the leading monomials for a reduced Gröbner basis, that is M_i does not divide M_j for any $i \neq j$. This is clear for $(i,j) \neq (1,0)$, so let us prove that M_1 does not divide M_0. We have $D = kn'(n'+1)/2$ with $n' := \left(\sqrt{8D/k+1}-1\right)/2$, so that

$$\frac{kn(n-1)}{2} < D \leqslant \frac{kn(n+1)}{2}.$$

In particular, this implies $u > 0$, whence $q > 0$ or $r > 0$.

It remains to prove that the sequence $M_0, \ldots, M_n$ form a vanilla Gröbner stair for degree D ideal as claimed. Notice that a monomial X^aY^b is under the stair $M_0, \ldots, M_n$ (i.e. in normal form w.r.t. G) if and only if $b < n$ and X^aY^b does not divide M_{n-b} (that is $a < q + k(n-b-1)+1$). Knowing this, it is not hard to check that there are D monomials under the stair, and that a monomial M is under the stair if and only if $M \prec_k M_{r+1}$. □

Corollary 7.2. *Let $G = (G^{(0)}, \ldots, G(n))$ be as above, and let M_i be the leading monomial of $G^{(i)}$ for $0 \leqslant i \leqslant n$. With q, r as in Proposition 7.1, the k-degree of M_i is given by*

$$\deg_k M_i = \begin{cases} kn & \text{if } i = 0, \\ k(n-1)+q+1 & \text{if } 0 < i \leqslant r, \\ k(n-1)+q & \text{if } r < i \leqslant n. \end{cases}$$

In particular, for all $i \in \{1, \ldots, n\}$, we have

$$\deg_k M_i \leqslant \deg_k M_{i-1}, \text{ and } \deg_k M_1 - 1 \leqslant \deg_k M_i \leqslant \deg_k M_1.$$

[1] In the example of Figure 7.1, the parameters are $D = 100$ and $k = 4$, hence $(n, q, r) = (7, 2, 2)$.

Remark 7.1. The results of Proposition 7.1 and Corollary 7.2 remain valid for $\prec_1$ with some precautions: if $r \geqslant 1$, one has to leave out $G^{(r)}$ since M_r is divisible by M_{r+1} with the given formulas. Then G consists of n elements

$$G^{(0)}, \dots, G^{(r-1)}, G^{(r+1)}, \dots, G^{(n)}.$$

Knowing the shape of the stair, it is now possible to bound the degrees of the quotients obtained with the dichotomic selection strategy from section 6.2.1:

Lemma 7.3. *Assume that the leading monomials of $(G^{(0)}, \dots, G^{(n)})$ form a vanilla Gröbner stair, and let $Q^{(0)}, \dots, Q^{(n)}$ be the quotients obtained with the dichotomic selection strategy from section 6.2.1. Then the bound*

$$\deg_k(Q^{(i)}) < 2k2^{\operatorname{val}_2 i}$$

holds for all $0 < i < n$.

Proof. Let $X^aY^b \in \operatorname{supp} Q^{(i)}$ with $0 < i < n$, so that $\Phi_G(M) = (i, X^aY^b)$ for $M := X^aY^b \operatorname{lm}(G^{(i)})$, and denote $\ell := 2^{\operatorname{val}_2 i}$. Then we observe that $b < \ell$: if not, then $\operatorname{lm}(G^{(i-\ell)})$ would divide M, whereas $\operatorname{val}_2(i-\ell) > \operatorname{val}_2 i$. A similar reasoning with $G^{(i+\ell)}$ (or $G^{(n)}$, whenever $i + \ell > n$) shows that $a < k\ell$. It follows that $\deg_k(X^aY^b) < 2k\ell$. □

7.1.2 Existence of relations

As mentioned in section 6.2.3, one ingredient of the fast reduction algorithm is to rewrite the equation $P = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} + R$ into another linear combination with much fewer terms. In particular, it should be possible to express each $G^{(i)}$ as a linear combination of elements in a suitable subset Σ of $\{G^{(0)}, \dots, G^{(n)}\}$ (this subset then generates the ideal I), with degrees that can be controlled.

It turns out that two elements are generally not enough, but that the subset $\Sigma := \{G^{(0)}, G^{(1)}, G^{(n)}\}$ generically works. In order to control the degrees in the linear combinations, we may also consider intermediate sets between $\{G^{(0)}, G^{(1)}, G^{(n)}\}$ and the full set $\{G^{(0)}, \dots, G^{(n)}\}$, such as

$$\Sigma_\ell := \{G^{(0)}, G^{(1)}, G^{(\ell)}, G^{(2\ell)}, \dots, G^{(\lfloor n/\ell \rfloor \ell)}, G^{(n)}\}$$

for various integer "step lengths" $\ell \geqslant 2$. This leads us to the following definition:

Definition 7.3. Let $\ell \geqslant 1$ be an integer and consider the set of indices

$$\mathcal{I}_\ell := \{0, 1, n\} \cup \{\ell, 2\ell, \dots, \lfloor n/\ell \rfloor \ell\}. \tag{7.1}$$

We say that a family of polynomials $P^{(0)}, \dots, P^{(n)} \in \mathbb{K}[X, Y]$ is *retractive* for step length ℓ and k-degree δ if for all $i \in \{0, \dots, n\}$ we can write

$$P^{(i)} = \sum_{j \in \mathcal{I}_\ell} A^{(i,j)} P^{(j)}$$

for some $(A^{(i,j)})_{j \in \mathcal{I}_\ell} \in \mathbb{K}[X, Y]^{|\mathcal{I}_\ell|}$ with $\deg_k A^{(i,j)} \leqslant \delta$.

Consider a Gröbner basis $G^{(0)}, \ldots, G^{(n)}$ as in Proposition 7.1 and a linear combination $C = \sum_{j \in \mathcal{I}_\ell} A^{(j)} G^{(j)}$ with $\deg_k A^{(j)} \leqslant \delta$ for all $j \in \mathcal{I}_\ell$. Making rough estimates, the number of monomials in $\mathcal{M}$ of k-degree $\leqslant d$ is $d^2/(2k)$, whence the number of monomials of k-degree between d and $d + \delta$ is bounded by $(d + \delta)\delta/k$. The set $\mathcal{N}_G = \mathcal{M}_{k,D}$ roughly corresponds to the set of monomials of k-degree at most nk, whence the support of C contains at most $(nk+\delta)\delta/k$ monomials that are *not* in $\mathcal{N}_G$. Notice that such a combination C is uniquely determined by its terms *not* in $\mathcal{N}_G$: if all the terms of $C - C' \in I$ are in $\mathcal{N}_G$, then $C - C' = 0$ by definition of a Gröbner basis.

On the other hand the polynomials $A^{(j)}$ with $j \in \mathcal{I}_\ell$ are determined by approximately $(n/\ell)\delta^2/(2k)$ coefficients. As soon as $\delta > 2k\ell$, it follows that

$$(n/\ell)\delta^2/(2k) > (nk + \delta)\delta/k\,,$$

and it becomes likely that non-trivial relations of the type $G^{(i)} = \sum_{j \in I_\ell} A^{(j)} G^{(j)}$ indeed exist. A refined analysis and practical experiments show that the precise threshold is located at

$$\delta \geqslant k(2\ell - 1) - 1\,,$$

although this empirical fact has not been formally proven.

7.1.3 Vanilla Gröbner bases

We are now in a position to describe the class of Gröbner bases with enough regularity for the fast reduction algorithm to work.

Definition 7.4. Let $G = (G^{(0)}, \ldots, G^{(n)})$ be the reduced Gröbner basis for an ideal $I \subset \mathbb{K}[X, Y]$ with respect to $\prec_k$. We say that G is a *vanilla Gröbner basis* if

1. the leading monomials of G form a vanilla Gröbner stair;
2. for each $\ell = 2, \ldots, n$, the family $G^{(0)}, \ldots, G^{(n)}$ is retractive for step length ℓ and k-degree $k(2\ell - 1) - 1$.

It seems that reduced Gröbner bases of sufficiently generic ideals are always of vanilla type, although this statement is only empirical so far. It is not even clear whether vanilla Gröbner bases exist for arbitrary fields $\mathbb{K}$ (with sufficiently many elements) and degrees D. Nevertheless, practical computer experiments suggest that sufficiently random ideals of degree D admit Gröbner bases of this kind. More precisely, this was checked for ideals that are generated as follows by two random polynomials:

- for $I = \langle A(X), Y - B(X) \rangle$, where A and B are random univariate polynomials of degrees D and $D - 1$, and for any ordering $\prec_k$;
- for $I = \langle A, B \rangle$, where A and B are random bivariate polynomials of total degree δ (in this case the degree of the ideal is $D = \delta^2$), and for any ordering $\prec_k$ with $k \geqslant 2$;

- for $I = (A, B)$, where A and B are random bivariate polynomials of degree δ in both variables (in this case the degree of the ideal is $D = 2\delta^2$), and for any ordering $\prec_k$ with $k \geqslant 2$.

The tests were made in $\mathbb{Z}_p$ (with p a 16-bit prime) and for a degree D in the range of a few hundreds, using the proof-of-concept implementation mentioned at the beginning of this chapter.

Remark 7.2. In each of these cases, the threshold $k(2\ell - 1) - 1$ seems to be sharp. Nevertheless, for our complexity bounds, a threshold of the type $Kk\ell$ would suffice, for any constant $K > 0$.

7.2 Terse representations of vanilla Gröbner bases

Notice that the definition of vanilla Gröbner basis was precisely made to match the ingredients from section 6.2. Then by definition, if $G := (G^{(0)}, \ldots, G^{(n)})$ is vanilla, it admits a *terse representation* (in $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$ space) as described in this section. Recall that $D := \dim_{\mathbb{K}} \mathbb{A} = \Theta(kn^2)$ by Proposition 7.1.

7.2.1 Retraction coefficients

For each $\ell \geqslant 1$, let $\mathcal{I}_\ell$ be as in (7.1). Also, for $\lambda \in \{0, \ldots, \lceil \log_2 n \rceil\}$, let $\mathcal{J}_\lambda$ be a shorthand for $\mathcal{I}_{2^\lambda}$. Since G is a vanilla Gröbner basis, Definition 7.4 ensures in particular the existence of coefficients $C^{(\lambda,i,j)} \in \mathbb{K}[X, Y]$ for $\lambda \in \{0, \ldots, \lceil \log_2 n \rceil - 1\}$ and $i \in \mathcal{J}_\lambda \setminus \mathcal{J}_{\lambda+1}$ and $j \in \mathcal{J}_{\lambda+1}$, such that

$$G^{(i)} = \sum_{j \in \mathcal{J}_{\lambda+1}} C^{(\lambda,i,j)} G^{(j)}\,, \tag{7.2}$$

$$\deg_k C^{(\lambda,i,j)} \leqslant k(2^{\lambda+2} - 1) - 1. \tag{7.3}$$

We call these $C^{(\lambda,i,j)}$ the *retraction coefficients* for G. For each given i, λ, the computation of the retraction coefficients $C^{(\lambda,i,j)}$ reduces to a linear system of size $u \times v$ with $u, v = O(kn2^\lambda)$ (for the image space, consider only the monomials that are *above* the Gröbner stair). This system is easily solved by Gaussian elimination. Notice that the space needed to write the retraction coefficients is much smaller than the Gröbner basis:

Lemma 7.4. *The family of all retraction coefficients for G takes space $O(kn^2 \log n)$.*

Proof. For every ℓ, there are $\lceil n/\ell \rceil + 1$ indices in $\mathcal{I}_\ell$, and we notice that $\mathcal{I}_{2\ell} \subseteq \mathcal{I}_\ell$. For any given λ, the retraction coefficients involve at most $n/2^{\lambda+1} + 1$ indices i and $n/2^{\lambda+1} + 2$ indices j, whence at most $n^2/4^{\lambda+1} + 3n/2^{\lambda+1} + 2$ pairs (i, j). Since the support of $C^{(\lambda,i,j)}$ has size $O(k4^\lambda)$ by (7.3), it follows that all retraction coefficient together require space $O(kn^2 \log n)$. □

We observe that the space needed to write all relations has the same order of magnitude as $\dim_{\mathbb{K}} \mathbb{A}$, up to a logarithmic factor.

7.2.2 Upper truncations

As per the second ingredient (section 6.2.2), we wish to keep only enough coefficients in $G^{(i)}$ to compute the associated quotient $Q^{(i)}$. This is done by "truncating" the basis element $G^{(i)}$ as follows:

Definition 7.5. Given a polynomial $P \in \mathbb{K}[X, Y]$, we define its *upper truncation with k-precision p* as the polynomial $P^{\#}$ such that

- all terms of $P^{\#}$ of k-degree less than $\deg_k P - p$ are zero;
- all terms of $P^{\#}$ of k-degree at least $\deg_k P - p$ are equal to the corresponding terms in P.

Notice that this upper truncation $P^{\#}$ has only $O((\deg_k P)p/k)$ coefficients. For the dichotomic selection strategy from section 6.2.1, we have for $0 < i < n$

$$\deg_k Q^{(i)} < 2k2^{\operatorname{val}_2 i}$$

by Lemma 7.3, so this is the precision required for $G^{(i)}$. Combining this with the retraction coefficients from the previous subsection, it becomes natural to define the terse representation as follows:

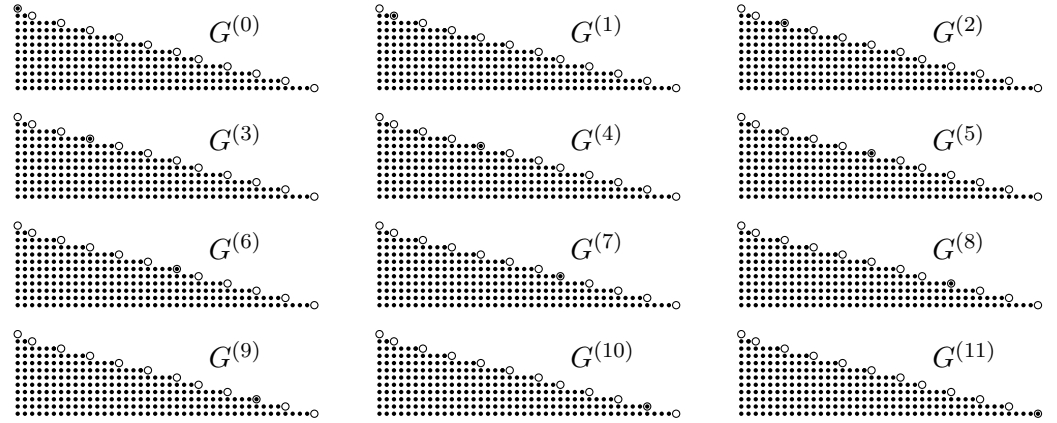
Definition 7.6. Let $G = (G^{(0)}, \ldots, G^{(n)})$ be a vanilla Gröbner basis for an ideal $I \subset \mathbb{K}[X, Y]$ with respect to $\prec_k$. The *terse representation* of G consists of the following data:

- the sequence of truncated elements $G^{(0,\#)}, \ldots, G^{(n,\#)}$, where
  - $G^{(i,\#)} := G^{(i)}$ for $i \in \{0, 1, n\}$;
  - $G^{(i,\#)}$ is the upper truncation of $G^{(i)}$ at precision $2k2^{\operatorname{val}_2 i}$ for all other i;
- the collection of all retraction coefficients $C^{(\lambda,i,j)}$ as in section 7.2.1.

Proposition 7.5. *The terse representation of G fits in space $O(kn^2 \log n)$.*

Proof. The upper truncation $G^{(i,\#)}$ requires space $O(kn2^{\operatorname{val}_2 i})$ for all $1 < i < n$. For each $\lambda < \log_2 n$, there are at most $n/2^\lambda$ indices i such that $\operatorname{val}_2 i = \lambda$; therefore, $G^{(2,\#)}, \ldots, G^{(n-1,\#)}$ take $O(kn^2 \log n)$ space. The elements $G^{(0,\#)}$, $G^{(1,\#)}$ and $G^{(n,\#)}$ require $O(kn^2)$ additional space, whereas the coefficients $C^{(\lambda,i,j)}$ account for $O(kn^2 \log n)$ more space by Lemma 7.4. □

Figure 7.2 shows an example of vanilla Gröbner basis with $D = 237$ and $k = 4$ (hence $n = 11$), together with its terse representation. As usual, the large white dots (○) represent the leading monomials of G, and black dots (•) represent monomials in the support of each polynomial. Notice that the terse representation has much fewer coefficients than G.



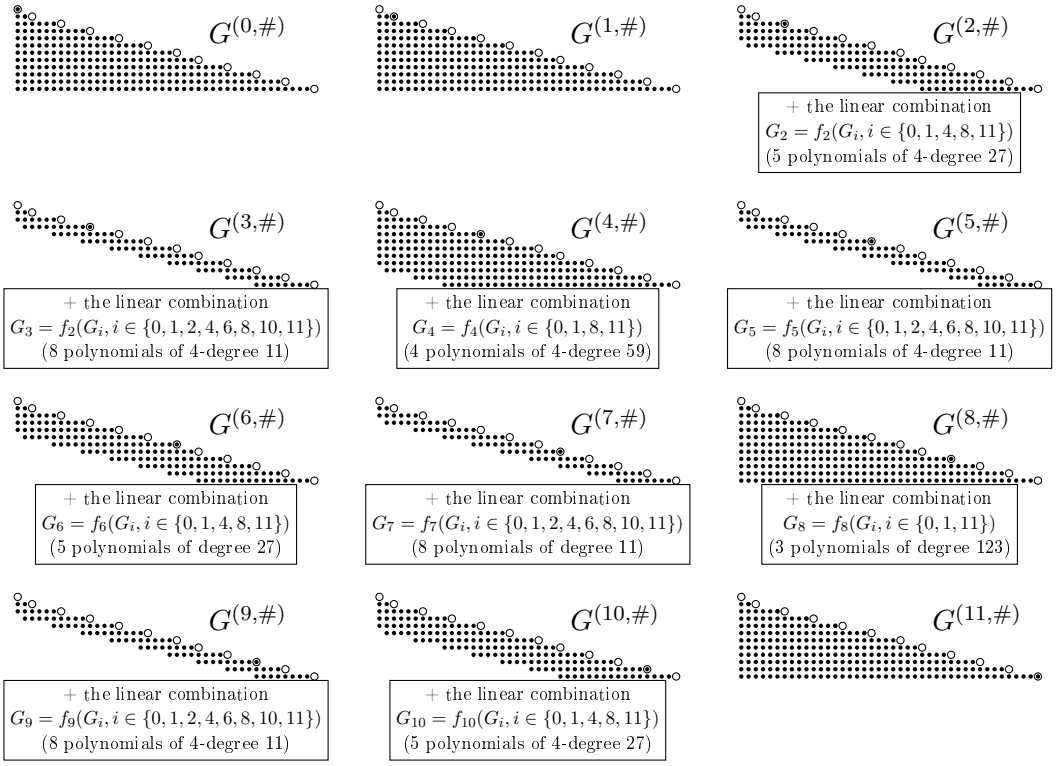


Figure 7.2: Example of vanilla Gröbner basis (above) and its terse representation (below). Parameters are $D = 237$ and $k = 4$, hence $(n, q, r) = (11, 2, 6)$.

7.3 Fast reduction

In this section, we assume that $G := (G^{(0)}, \dots, G^{(n)})$ is vanilla and that its terse representation has been precomputed. Also, let P be a polynomial of k-degree d, that is with $\Theta(d^2/k)$ coefficients. In this case, we can design an algorithm to compute an extended reduction

$$P = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} + R$$

in $\tilde{O}(kn^2 + d^2/k) = \tilde{O}(\dim_{\mathbb{K}} \mathbb{A} + |P|)$ operations. In particular, this gives the normal form of P w.r.t. G with quasi-optimal complexity.

The reduction algorithm proceeds in two steps: in a first stage, we compute the quotients $Q^{(0)}, \dots, Q^{(n)}$; we next evaluate the remainder $R := P - Q^{(0)}G^{(0)} - \cdots - Q^{(n)}G^{(n)}$ by rewriting the linear combination $Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)}$ using fewer and fewer terms, as in section 6.2.3.

7.3.1 Computing the quotients

To compute the quotients, we compute an extended reduction of P with respect to the truncated basis $G^{\#} := (G^{(0,\#)}, \dots, G^{(n,\#)})$, assuming the dichotomic selection strategy. This is done using the algorithm from [Hoe15], with a straightforward adaptation to use the dichotomic selection strategy instead of the naive one. The complexity of this step is given by the following lemma:

Lemma 7.6. *The extended reduction $P = Q^{(0)}G^{(0,\#)} + \cdots + Q^{(n)}G^{(n,\#)} + R^{\#}$ (with P of degree d) can be computed in time*

$$O\left(\mathsf{R}(kn^2)\log n + \mathsf{R}(d^2/k)\right)$$

using the algorithm from [Hoe15] and a dichotomic selection strategy. Moreover, the total size of the quotients is

$$|Q^{(0)}| + \cdots + |Q^{(n)}| = O(d^2/k + kn^2)$$

Proof. Recall that $deg_k Q^{(i)} < 2k2^{\mathrm{val}_2\, i}$ by Lemma 7.3. Denoting $\ell := 2^{\mathrm{val}_2\, i}$, we get $|Q^{(i)}| < 2\ell(2k\ell + 1) = O(k\ell^2)$ for any $0 < i < n$. Since the number of indices $0 < i < n$ with $\ell = 2^{\mathrm{val}_2\, i}$ is bounded by n/ℓ, we get

$$|Q^{(1)}| + \cdots + |Q^{(n-1)}| = O(2kn + 4kn + \cdots + 2^{\lfloor \log_2 n \rfloor} kn) = O(kn^2)\,.$$

Also, $|Q^{(i)}G^{(i,\#)}| = O(kn\ell)$ for any $1 < i < n$, and $|Q^{(1)}G^{(1,\#)}| = O(kn^2)$; therefore

$$\mathsf{R}(|Q^{(1)}G^{(1,\#)}|) + \cdots + \mathsf{R}(|Q^{(n-1)}G^{(n-1,\#)}|) = O\left(\mathsf{R}(kn^2)\log n\right)\,.$$

On the other hand, $\deg_k(Q^{(0)}G^{(0,\#)}) \leqslant \deg_k P$ and similarly for $Q^{(n)}G^{(n,\#)}$, whence $|Q^{(0)}| + |Q^{(n)}| = O(d^2/k)$ and

$$\mathsf{R}(|Q^{(0)}G^{(0,\#)}|) + \mathsf{R}(|Q^{(n)}G^{(n,\#)}|) = O\left(\mathsf{R}(d^2/k)\right)\,. \qquad \square$$

The next important observation is that the quotients $Q^{(0)}, \dots, Q^{(n)}$ with respect to $G^{\#}$ are also valid quotients with respect to G:

Proposition 7.7. *Let $Q^{(0)}, \dots, Q^{(n)}$ be as in Lemma 7.6 and consider*

$$R := P - Q^{(0)}G^{(0)} - \cdots - Q^{(n)}G^{(n)}.$$

Then R is reduced with respect to G.

Proof. Let $R^{\#} := P - Q^{(0)}G^{(0,\#)} - \cdots - Q^{(n)}G^{(n,\#)}$. By construction, $R^{\#}$ is reduced with respect to $G^{\#}$ and whence with respect to G, since $\operatorname{lm}(G^{(i)}) = \operatorname{lm}(G^{(i,\#)})$ for all i. For any $0 < i < n$, we also have $\deg_k(G^{(i)} - G^{(i,\#)}) < \deg_k G^{(i)} - 2k2^{\operatorname{val}_2 i}$, so that

$$\deg_k \left(Q^{(i)}G^{(i)} - Q^{(i)}G^{(i,\#)}\right) < \deg_k G^{(i)} - 1 \leqslant \min_{0 \leqslant j \leqslant n} \deg_k G^{(j)}$$

by Lemma 7.3 and Corollary 7.2. Since $G^{(0)} = G^{(0,\#)}$ and $G^{(n)} = G^{(n,\#)}$, this means that

$$\deg_k(R - R^{\#}) < \deg_k G^{(i)} \text{ for all } 0 \leqslant i \leqslant n.$$

In other words, the polynomials $R^{\#}$, $R - R^{\#}$, and therefore R are all reduced with respect to G. □

Remark 7.3. Here we have used the fact that all basis elements $G^{(i)}$ have roughly the same degree, so it is crucial that the monomials under the stair are the minimal elements with respect to $\prec_k$.

7.3.2 Computing the remainder

Once the quotients $Q^{(0)}, \dots, Q^{(n)}$ are known, we need to compute the remainder $R := P - Q^{(0)}G^{(0)} - \cdots - Q^{(n)}G^{(n)}$. We do this by rewriting (or *retracting*) the linear combination $Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)}$ into a linear combination

$$S^{(0)}G^{(0)} + S^{(1)}G^{(1)} + S^{(n)}G^{(n)}$$

as in section 6.2.3.

More precisely, we replace about half of the terms using the relations provided by $C^{(1,i,j)}$, to keep only the terms with indices in $\mathcal{J}_1$. Then, using the relations provided by $C^{(2,i,j)}$, the number of terms is halved again and only indices in $\mathcal{J}_2$ remain. Repeating this for every $\lambda < \log_2 n$, we are left with the expected linear combination. This leads to Algorithm 7.1 below.

Lemma 7.8. *Algorithm 7.1 is correct and runs in time $O\left(\mathsf{M}(kn^2)\log n\right)$.*

Algorithm 7.1. Retraction of the extended reduction for vanilla Gröbner bases

Prototype: `retrac`(Q, C)

Input: Quotients $Q = (Q^{(0)}, \dots, Q^{(n)})$ as in Lemma 7.6, and a family C of retraction coefficients as in section 7.2.1:

$$C := \left(C^{(\lambda,i,j)}\right)_{\lambda<\log_2 n, i\in\mathcal{J}_\lambda\setminus\mathcal{J}_{\lambda+1}, j\in\mathcal{J}_{\lambda+1}}$$

Output: $S^{(0)}, S^{(1)}, S^{(n)} \in \mathbb{K}[X,Y]$ such that

$$Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} = S^{(0)}G^{(0)} + S^{(1)}G^{(1)} + S^{(n)}G^{(n)}\,.$$

Set $Q^{(0,j)} := Q^{(j)}$ for $j = 0, \dots, n$; set $\ell := \lceil \log_2 n \rceil$.
for $\lambda = 1, \dots, \ell - 1$ **do**
  for $j = 0, \dots, n$ **do**
    if $1 < j < n$ and $\mathrm{val}_2\, j \leqslant \lambda$ **then**
      Set $Q^{(\lambda+1,j)} := 0$
    else
      Set $Q^{(\lambda+1,j)} := Q^{(\lambda,j)} + \sum_{i\in\mathcal{J}_\lambda\setminus\mathcal{J}_{\lambda+1}} Q^{(\lambda,i)}C^{(\lambda,i,j)}$.
    end if
  end for
end for
return $(S^{(0)}, S^{(1)}, S^{(n)}) := (Q^{(\ell,0)}, Q^{(\ell,1)}, Q^{(\ell,n)})$.

Proof. By construction, we notice that $Q^{(\lambda,j)} = 0$ if $1 < j < n$ and $\mathrm{val}_2\, j < \lambda$ (that is $j \notin \mathcal{J}_\lambda$). Let us now show by induction over λ that

$$Q^{(\lambda,0)}G^{(0)} + \cdots + Q^{(\lambda,n)}G^{(n)} = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)}\,.$$

This is clearly true for $\lambda = 0$, and we have

$$\begin{aligned}
\sum_{j\in\mathcal{J}_{\lambda+1}} Q^{(\lambda+1,j)}G^{(j)} &= \sum_{j\in\mathcal{J}_{\lambda+1}} \left(Q^{(\lambda,j)} + \sum_{i\in\mathcal{J}_\lambda\setminus\mathcal{J}_{\lambda+1}} Q^{(\lambda,i)}C^{(\lambda,i,j)}\right) G^{(j)} \\
&= \sum_{j\in\mathcal{J}_{\lambda+1}} Q^{(\lambda,j)}G^{(j)} + \sum_{i\in\mathcal{J}_\lambda\setminus\mathcal{J}_{\lambda+1}} Q^{(\lambda,i)} \sum_{j\in\mathcal{J}_{\lambda+1}} C^{(\lambda,i,j)}G^{(j)} \\
&= \sum_{j\in\mathcal{J}_{\lambda+1}} Q^{(\lambda,j)}G^{(j)} + \sum_{i\in\mathcal{J}_\lambda\setminus\mathcal{J}_{\lambda+1}} Q^{(\lambda,i)}G^{(i)} \\
&= \sum_{j\in J_\lambda} Q^{(\lambda,i)}G^{(i)}\,,
\end{aligned}$$

which proves the correctness of Algorithm 7.1.

Recall that by definition (7.3), $\deg_k C^{(\lambda,i,j)} < 4k2^\lambda$. Again by induction over λ, it is not hard to see that this bound implies

$$\deg_k(Q^{(\lambda,i)}) \leqslant \max(4k2^\lambda, 2k2^{\mathrm{val}_2\, i}) \text{ for } 1 < i < n. \tag{7.4}$$

Now, for $i \in \mathcal{J}_\lambda \setminus \mathcal{J}_{\lambda+1}$ and $j \in \mathcal{J}_\lambda$, the product $Q^{(\lambda,i)}C^{(\lambda,i,j)}$ is computed in time $O(k4^\lambda)$, and there are $O(n^2/4^\lambda)$ such products (see the proof of Lemma 7.4). Using the classical assumption that $\mathsf{M}(d)/d$ is non-decreasing, we conclude that each step can be computed in time $O(\mathsf{M}(kn^2))$. □

Combining our subalgorithms, we obtain the algorithm for extended reduction in quasi-linear time:

Algorithm 7.2. Extended reduction w.r.t. a vanilla Gröbner basis

Prototype: `reduce_vanilla`$(P, G^\#, C)$

Input: A polynomial $P \in \mathbb{K}[X,Y]$, and $G^\#, C$ the terse representation of a vanilla Gröbner basis G.

Output: An extended reduction $(Q^{(0)}, \dots, Q^{(n)}, R)$ of P modulo G.

1: Compute the extended reduction $(Q^{(0)}, \dots, Q^{(n)}, R^\#)$ with respect to $G^\#$, using the algorithm from [Hoe15] and the dichotomic selection strategy.
2: Compute $S^{(0)}, S^{(1)}, S^{(n)} :=$ `retrac`(Q, C).
3: Compute $R := P - S^{(0)}G^{(0,\#)} - S^{(1)}G^{(1,\#)} - S^{(n)}G^{(n,\#)}$
4: **return** $(Q^{(0)}, \dots, Q^{(n)}, R)$.

Theorem 7.9. *Algorithm 7.2 is correct and runs in time*

$$O\left(\mathsf{R}(kn^2)\log n + \mathsf{R}(d^2/k)\right).$$

Proof. Because of Lemma 7.6, the extended reduction with respect to $G^\#$ in step 1 is computed in time

$$O\left(\mathsf{R}(kn^2)\log n + \mathsf{R}(d^2/k)\right).$$

Proposition 7.7 ensures that the quotients are also valid with respect to G.

The next step is to evaluate the remainder $R := P - Q^{(0)}G^{(0)} - \cdots - Q^{(n)}G^{(n)}$. The $S^{(i)}$ are computed in time $O\left(\mathsf{M}(kn^2)\log n\right)$ using Lemma 7.8, and we have

$$\begin{aligned} S^{(0)}G^{(0,\#)} + S^{(1)}G^{(1,\#)} + S^{(n)}G^{(n,\#)} &= S^{(0)}G^{(0)} + S^{(1)}G^{(1)} + S^{(n)}G^{(n)} \\ &= Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)}. \end{aligned}$$

For $i \in \{0, 1, n\}$, it follows from (7.4) that $\deg_k(S^{(i)}G^{(i,\#)}) \leqslant \max(d, 5kn)$. Consequently, the evaluation of R takes time

$$O\left(\mathsf{M}(d^2/k) + \mathsf{M}(kn^2)\right).$$ □

7.4 Applications

This section presents two immediate consequences of Theorem 7.9: multiplication in $\mathbb{A}$ and conversion between different normal forms can be done in time $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$.

7.4.1 Multiplications in the quotient algebra

Let $G = (G^{(0)}, \dots, G^{(n)})$ be a vanilla Gröbner basis for an ideal $I \subset \mathbb{K}[X,Y]$ with respect to $\prec_k$, and assume that we have precomputed a terse representation for G. Elements in the quotient algebra $\mathbb{A} := \mathbb{K}[X,Y]/I$ can naturally be represented as polynomials in $\mathbb{K}[X,Y]$ that are reduced with respect to G. An immediate application of Theorem 7.9 is a multiplication algorithm for $\mathbb{A}$ that runs in quasi-linear time.

More precisely, with the notations from Proposition 7.1, given two polynomials $P, Q \in \mathbb{K}[X,Y]$ that are reduced with respect to G, we have $\deg_k P \leqslant kn$ and $\deg_k Q \leqslant kn$, whence $\deg_k PQ \leqslant 2kn$ and $|PQ| = O(kn^2) = O(D)$. It follows that PQ can be computed in time $O(\mathsf{M}(D))$, whereas the reduction of PQ with respect to G takes time $O(\mathsf{R}(D)\log D)$. This yields:

Theorem 7.10. *For I as above, multiplication in $\mathbb{A} := \mathbb{K}[X,Y]/I$ can be performed in time $O(\mathsf{R}(D)\log D)$, with $D := \dim_{\mathbb{K}} \mathbb{A}$.*

7.4.2 Changing the monomial ordering

Let us now assume that our ideal $I \subset \mathbb{K}[X,Y]$ admits a vanilla Gröbner basis $G^{[k]}$ with respect to the ordering $\prec_k$ for all k. The notation $\mathbb{A}^{[k]} = K[X,Y]/I$ denotes the quotient algebra when representing elements using normal forms with respect to $G^{[k]}$. If $k > D = \dim_{\mathbb{K}} \mathbb{A}$, then we notice that $G^{[k]}$ is also a Gröbner basis with respect to the lexicographical monomial ordering $\prec_\infty$. In order to efficiently convert between $\mathbb{A}^{[k]}$ and $\mathbb{A}^{[\ell]}$ with $k < \ell$, we first consider the case when $\ell \leqslant 2k$:

Lemma 7.11. *With the above notations and $k < \ell \leqslant 2k$, assume that we have precomputed terse representations for $G^{[k]}$ and $G^{[\ell]}$.*

Then back and forth conversions between $\mathbb{A}^{[k]}$ and $\mathbb{A}^{[\ell]}$ can be computed in time $O(\mathsf{R}(D)\log D)$.

Proof. Assume that $G^{[k]}$ has $n+1$ elements $G^{([k],0)}, \dots, G^{([k],n)}$ and $G^{[\ell]}$ has $m+1$ elements $G^{([\ell],0)}, \dots, G^{([\ell],m)}$. We know from Proposition 7.1 that

$$kn(n-1) < 2D \leqslant kn(n+1)$$

and similarly

$$\ell(m-1)m < 2D \leqslant \ell m(m+1)\,.$$

Now given $P \in \mathbb{K}[X,Y]$ that is reduced with respect to $G^{[k]}$, we have $\deg_k P \leqslant kn$, whence $\deg_\ell P \leqslant \ell n$ and $(\deg_\ell P)^2/\ell \leqslant \ell n^2 \leqslant 2kn^2 = O(D)$. Theorem 7.9 therefore implies that the normal form of P w.r.t. $G^{[\ell]}$ can be computed in time

$$O\left(\mathsf{R}(\ell m^2)\log m + \mathsf{R}(D)\right)$$

and we conclude using $\ell m^2 = O(D)$. The proof for the backward conversion is similar. □

For general $k < \ell$, let $a \leqslant b$ be such that $2^{a-1} < k \leqslant 2^a$ and $2^{b-1} < \ell \leqslant 2^b$. Then we may perform conversions between $\mathbb{A}^{[k]}$ and $\mathbb{A}^{[\ell]}$ using a Gröbner walk

$$\mathbb{A}^{[k]} \leftrightarrow \mathbb{A}^{[2^a]} \leftrightarrow \cdots \leftrightarrow \mathbb{A}^{[2^{b-1}]} \leftrightarrow \mathbb{A}^{[\ell]}.$$

All $G^{[k]}$ coincide for $k > D$, so we can assume that $1 \leqslant k < \ell \leqslant D+1$. Then there are at most $\log D$ conversions as above, so that:

Theorem 7.12. *With the above notations and $k < \ell \leqslant D+1$, assume that we have precomputed terse representations for the bases $G^{[k]}, G^{[2^a]}, \ldots, G^{[2^b]}, G^{[\ell]}$.*

Then back and forth conversions between $\mathbb{A}^{[k]}$ and $\mathbb{A}^{[\ell]}$ can be computed in time $O(\mathsf{R}(D)\log^2 D)$.

Perspectives

The setting of vanilla Gröbner bases is rather restrictive, so it is natural to ask whether some of the assumptions may be relaxed. An extension to more general term orders (starting with $\prec_k$ for $k \in \mathbb{Q}$) should be rather straightforward. Similarly, there is no apparent reason to think that vanilla Gröbner bases are limited to two variables (although they may be less frequent with $r > 2$ variables); however the implicit dependency in the number r of variables is likely to be $r!$. Indeed, the dichotomic selection strategy defines a parallelepiped based on the extremities of the Gröbner basis, while the monomials under the stair form a pyramid; and there is a ratio of $r!$ between the volumes of such shapes in r dimensions.

Another interesting question would be to prove that vanilla Gröbner bases are generic in the usual sense, that is for all inputs in an open subset of Zariski topology. In other words, we wish to find a nontrivial algebraic equation

$$E : (\bar{\mathbb{K}}[X,Y] \times \bar{\mathbb{K}}[X,Y]) \to \bar{\mathbb{K}}$$

($\bar{\mathbb{K}}$ being the algebraic closure of $\mathbb{K}$) such that the Gröbner basis of $I := \langle A, B\rangle$ is vanilla as long as $E(A,B) \neq 0$. This hypothesis is supported by the fact that random polynomials A, B give a vanilla Gröbner basis, but this does not constitute a proof.

Chapter 8

Case of the bivariate grevlex order

Contents

Note. *The results in this chapter were published in* [HL19a]. *In addition to this article, section 8.6 reports on an implementation available in the* LARRIX *package of* MATHEMAGIX `http://www.mathemagix.org`.

This chapter presents a second situation where quasi-optimal reduction is actually possible. The previous chapter made regularity assumptions on the Gröbner basis itself, but one may focus on the generating polynomials instead. If the ideal is defined by generic polynomials with dense support for the graded order, then the Gröbner basis presents a particular structure, as studied for example in [Gal74, FH94, Mor03]. This situation is often used as a benchmark for polynomial

Table 8.1: Asymptotic complexities for various problems on the ideal $\langle A, B\rangle$, assuming $n := \deg A \leqslant m := \deg B$ and A, B sufficiently generic.

Operation	This work	Previous best
Deglex Gröbner basis G	$O(\mathsf{R}(m^2) + \mathsf{R}(nm)n\log n)$ $= \tilde{O}(\lvert A, B, G\rvert)$	$O(n^2\mathsf{R}(nm))$
Structure of $\mathbb{A} := \mathbb{K}[X,Y]/\langle A, B\rangle$	$O(\mathsf{R}(m^2) + \mathsf{M}(nm)\log n)$ $= \tilde{O}(\lvert A, B\rvert)$	$O(n^2\mathsf{R}(nm))$
Normal form of P with $\deg P = d$	$O(\mathsf{R}(d^2) + \mathsf{R}(nm)\log n)$ $= \tilde{O}(\lvert P\rvert + \dim_{\mathbb{K}} \mathbb{A})$	$O(\mathsf{R}(d^2) + n\mathsf{R}(nm))$
Multiplication in $\mathbb{A}$	$O(\mathsf{R}(nm)\log n)$ $= \tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$	$O(n\mathsf{R}(nm))$

Notations: M and R represent cost functions for zealous and relaxed multiplication respectively (see section 6.4).
In the last two rows, for the computation of normal forms and multiplication in $\mathbb{A} := \mathbb{K}[X,Y]/\langle A, B\rangle$, we assume that the structure of $\mathbb{A}$ has been precomputed using the algorithm from the second row. Notice that the algorithm for the normal form (third line) also yields an ideal membership test for $\langle A, B\rangle$.

system solving: see the PoSSo problem [FGHR13]. As in the previous chapter, we restrict ourselves to the bivariate case, as studied for example in [LMS13].

In what follows, $A, B \in \mathbb{K}[X,Y]$ are generic polynomials of total degree n, m respectively. We consider the Gröbner basis of $\langle A, B\rangle$ with respect to the graded lexicographic order. Under these very particular assumptions, the Gröbner basis admits a *concise representation*; its structure is slightly different from the terse representation of vanilla bases (previous chapter), but it can similarly be used in fast reduction algorithms. In particular, this allows us to compute the normal form of any polynomial $P \in \mathbb{K}[X,Y]$ in $\tilde{O}(|P| + \dim_{\mathbb{K}} \mathbb{A})$ operations, and to perform multiplications in $\mathbb{A}$ in time $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$.

A major difference with the results from Chapter 7 is that it does not rely on expensive precomputations. Indeed, unlike terse representations, the concise representation can be computed directly from the generators in quasi-linear time $\tilde{O}(|A, B|)$. In fact, determining a Gröbner basis in concise representation essentially boils down to a univariate gcd computation. Summarizing, the results in this chapter require more restrictive assumptions than Chapter 7 (only the total degree order is considered, and the generators must be of a specific shape), but they are considerably stronger as they are valid *without precomputations*.

Combining these two algorithms, we obtain an ideal membership test (given P, A, B, decide whether $P \in^? \langle A, B\rangle$) in quasi-linear time $\tilde{O}(|P, A, B|)$. Also if needed[1], the reduced Gröbner basis in the classical sense can be computed in quasi-linear time with respect to the output size. These complexity results are summarized in Table 8.1, and compared with the complexity of classical algorithms from Chap-

[1] For most purposes, the concise representation is actually sufficient.

ter 6. For clarity, the algorithms are first presented in a simplified form, with slightly weaker complexity results; then the analysis is refined in section 8.5.

8.1 Presentation of the setting

This section aims to describe more formally the hypotheses made above, and shows how to construct a Gröbner basis with the expected properties. The monomial ordering used in this chapter is the usual degree lexicographic order with $X \prec Y$, that is

$$X^a Y^b \prec X^u Y^v \Leftrightarrow a+b < u+v \text{ or } (a+b = u+v \text{ and } b < v) .$$

8.1.1 Reduced Gröbner bases

We consider an ideal $I \subset \mathbb{K}[X,Y]$ generated by two generic polynomials A, B of total degree n, m and we assume $n \leqslant m$. Here the adjective "generic" should be understood as "no accidental cancellation occurs during the computation". This is typically the case if A, B are chosen at random: assuming that $\mathbb{K}$ has sufficiently many elements, the probability of an accidental cancellation is small. In this generic bivariate setting, a clever application of Buchberger's algorithm [Buc65] gives the reduced Gröbner basis $G^{\text{red}} = (G^{(\text{red},0)}, \ldots, G^{(\text{red},n)})$ of I with respect to $\prec$ as follows:

- Set $G^{(\text{red},1)} := B \operatorname{rem} A$ and $G^{(\text{red},0)} := A \operatorname{rem} G^{(\text{red},1)}$.
- $G^{(\text{red},i)} := S(G^{(\text{red},i-2)}, G^{(\text{red},i-1)}) \operatorname{rem} (G^{(\text{red},0)}, \ldots, G^{(\text{red},i-1)})$ for $i = 2, \ldots, n$.
- For all $i = 0, \ldots, n$, divide $G^{(\text{red},i)}$ by its leading coefficient to make it monic.

It is well known [Gal74] that the Gröbner stair has steps of height 1, so the algorithm stops when $i = n = \deg A$, or equivalently when the leading term of $G^{(\text{red},i)}$ is a power of X. It can also be checked that the width of each step is 2, except for the first one that has width $n - m + 1$. In other words, the leading monomials $\operatorname{lm}(G^{(\text{red},i)})$ of G^{red} are given by

$$\operatorname{lm}(G^{(\text{red},0)}) = Y^n \tag{8.1}$$

$$\operatorname{lm}(G^{(\text{red},i)}) = X^{m-n-1+2i} Y^{n-i}, \qquad i = 1, \ldots, n. \tag{8.2}$$

There are nm monomials under the stair; since the Bézout bound is reached for generic ideals, this ensures that G^{red} is indeed a reduced Gröbner basis.

8.1.2 From Euclidean division to Gröbner bases

The reduced Gröbner basis as above verifies relatively simple recurrence relations. In fact, it is possible to construct another (non-reduced) Gröbner basis

$$G = (G^{(0)}, \ldots, G^{(n)})$$

with even simpler recurrence relations. These recurrence relations will later be used to rewrite the equation as in section 6.2.3.

Definition 8.1. For a polynomial $P = \sum P_{i,j} X^i Y^j \in \mathbb{K}[X,Y]$ of total degree d, we define its dominant diagonal $\mathrm{Diag}(P) \in \mathbb{K}[Z]$ by $\mathrm{Diag}(P) = \sum_{j \leqslant d} P_{d-j,j} Z^j$.

We have the trivial properties that $\mathrm{Diag}(XP) = \mathrm{Diag}(P)$ and $\mathrm{Diag}(YP) = Z\,\mathrm{Diag}(P)$. For generic A and B, the diagonals $\mathrm{Diag}(A)$ and $\mathrm{Diag}(B)$ are also generic. Applying the Euclidean algorithm to these diagonals, it follows that the successive remainders follow a "normal sequence", i.e. their degrees decrease by exactly one at each step.

Now consider the sequence $G^{(0)}, \ldots, G^{(n)}$ with $n = \deg A$ defined by

$$G^{(0)} := A \tag{8.3}$$
$$G^{(1)} := B \operatorname{rem} A \tag{8.4}$$
$$G^{(i)} := X^{d_i} G^{(i-2)} - (u_i Y + v_i X) G^{(i-1)}, \qquad i = 2, \ldots, n, \tag{8.5}$$

where

$$u_i Z + v_i := \mathrm{Diag}(G^{(i-2)}) \operatorname{quo} \mathrm{Diag}(G^{(i-1)}), \qquad d_i := \begin{cases} m-n+1 & \text{if } i = 2\,, \\ 2 & \text{if } i > 2\,. \end{cases}.$$

Let us first notice that the term $X^{d_i} G^{(i-2)} - u_i Y G^{(i-1)}$ corresponds to the S-polynomial of $G^{(i-2)}$ and $G^{(i-1)}$, as in the classical Buchberger algorithm. Setting $D^{(i)} := \mathrm{Diag}(G^{(i)})$ for $i = 0, \ldots, n$, we next observe that

$$D^{(1)} = \mathrm{Diag}(B) \operatorname{rem} \mathrm{Diag}(A)$$
$$D^{(i)} = D^{(i-2)} \operatorname{rem} D^{(i-1)}, \qquad i = 2, \ldots, n,$$

so the diagonals are the successive remainders in the Euclidean algorithm and the corresponding quotients indeed all have degree 1. By induction on i, we deduce:

Lemma 8.1. *The $G^{(i)}$ as in (8.3–8.5) have the same leading monomials (8.1–8.2) as the $G^{(\mathrm{red},i)}$, so $G := (G^{(0)}, \ldots, G^{(n)})$ is a Gröbner basis of $\langle A, B \rangle$ with respect to $\prec$.*

Corollary 8.2. *Any $P \in \mathbb{K}[X,Y]$ has the same normal form with respect to G and G^{red}.*

Remark 8.1. The genericity assumptions on A and B can also be made more precise now: on the one hand, we need that $\deg D^{(i)} = n - i$ for $i = 0, \ldots, n$ and $\deg G^{(i)} = m + i - 1$ for $i = 1, \ldots, n$. On the other hand, we need $X^2 G^{(n-1)} - (u_n Y + v_n X) G^{(n)}$ to reduce to zero with respect to G, where $u_n Z + v_n := D^{(n-1)} \operatorname{quo} D^{(n)}$. This in particular provides us with an *a posteriori* sanity check for ensuring that the genericity assumptions are indeed satisfied.

8.1.3 Examples

Example 1. Consider $A := Y + aX + b$ and $B := Y^4 + \cdots$, then $G^{(0)} = Y + aX + b$ and $G^{(1)} = B(X, -aX - b) = cX^4 + \cdots$ for some c. The sequence stops here since $n := \deg A = 1$. Notice that $(G^{(0)}, G^{(1)}/c)$ is actually the reduced Gröbner basis in this case.

Example 2. Consider $A, B \in \mathbb{F}_{11}[X, Y]$ defined as

$$\begin{aligned} A &:= Y^4 - 3XY^3 + X^2Y^2 - 3X^3Y + 5X^4 + 3Y^3 + 2XY^2 - 3X^2Y - 4X^3\,, \\ B &:= Y^4 + 2XY^3 - 3X^2Y^2 + 4X^3Y - 3X^4 - 5Y^3 - 3XY^2 + 5X^2Y - 5X^3\,. \end{aligned}$$

The reader's favorite computer algebra system gives the following reduced Gröbner basis:

$$\begin{aligned} G^{(\mathrm{red},0)} =& Y^4 + 3X^2Y^2 - X^3Y - 2X^4 - 4Y^3 - XY^2 + 4X^2Y + 2X^3\,, \\ G^{(\mathrm{red},1)} =& XY^3 - 3X^2Y^2 - 3X^3Y + 5X^4 + 5Y^3 - XY^2 - 5X^2Y + 2X^3\,, \\ G^{(\mathrm{red},2)} =& X^3Y^2 - 2X^4Y + 4X^5 + 3X^2Y^2 - 5X^3Y - 3X^4 + Y^3 - 3XY^2 \\ & - X^2Y - 5X^3\,, \\ G^{(\mathrm{red},3)} =& X^5Y - X^6 - 4X^4Y - 4X^5 - 5X^2Y^2 - 3X^3Y - 4X^4 + Y^3 \\ & + 5XY^2 - X^2Y + X^3\,, \\ G^{(\mathrm{red},4)} =& X^7 - 5X^6 - 5X^4Y + 2X^5 + 5X^2Y^2 - X^3Y - X^4 - 3Y^3 - 4XY^2 \\ & + 3X^2Y - 3X^3\,. \end{aligned}$$

Computing the basis G as in (8.3–8.5), we obtain

$$\begin{aligned} G^{(0)} &= Y^4 - 3XY^3 + X^2Y^2 - 3X^3Y + 5X^4 + 3Y^3 + 2XY^2 - 3X^2Y - 4X^3\,, \\ G^{(1)} &= 5XY^3 - 4X^2Y^2 - 4X^3Y + 3X^4 + 3Y^3 - 5XY^2 - 3X^2Y - X^3\,, \\ G^{(2)} &= 4X^3Y^2 + 3X^4Y + 5X^5 - 5Y^4 + 4XY^3 - 4X^2Y^2 - 5X^3Y - 4X^4\,, \\ G^{(3)} &= -X^5Y + X^6 - 2Y^5 - 3XY^4 + 2X^2Y^3 - X^3Y^2 + 4X^4Y + 5X^5\,, \\ G^{(4)} &= X^7 + 3Y^6 - 4XY^5 + 4X^2Y^4 + 3X^3Y^3 + 5X^4Y^2 - X^5Y - 2X^6\,. \end{aligned}$$

Notice that $G^{(i)}$ and $G^{(\mathrm{red},i)}$ have the same leading monomial, so $(G^{(0)}, \ldots, G^{(4)})$ is a Gröbner basis as well. However, it is not reduced: $G^{(4)}$ contains the term $3Y^6$, which is divisible by the leading monomial of $G^{(0)} = Y^4 + \cdots$.

Example 3. Figure 8.1 shows a schematic representation of the behavior when A and B have degree 11; the same example will be used in the rest of the chapter. Recall that black dots (•) represent the supports of the polynomials, while larger white dots (○) give the shape of the Gröbner stair. Notice again that $G^{(i)}$ and $G^{(\mathrm{red},i)}$ have the same leading monomial.

Remark 8.2. The second example already involves a lot of coefficients, even for the modest degree 4. Such low degrees are not sufficient for the ingredients presented in section 6.2. This explains why examples are preferably given in a schematic form as in Figure 8.1.

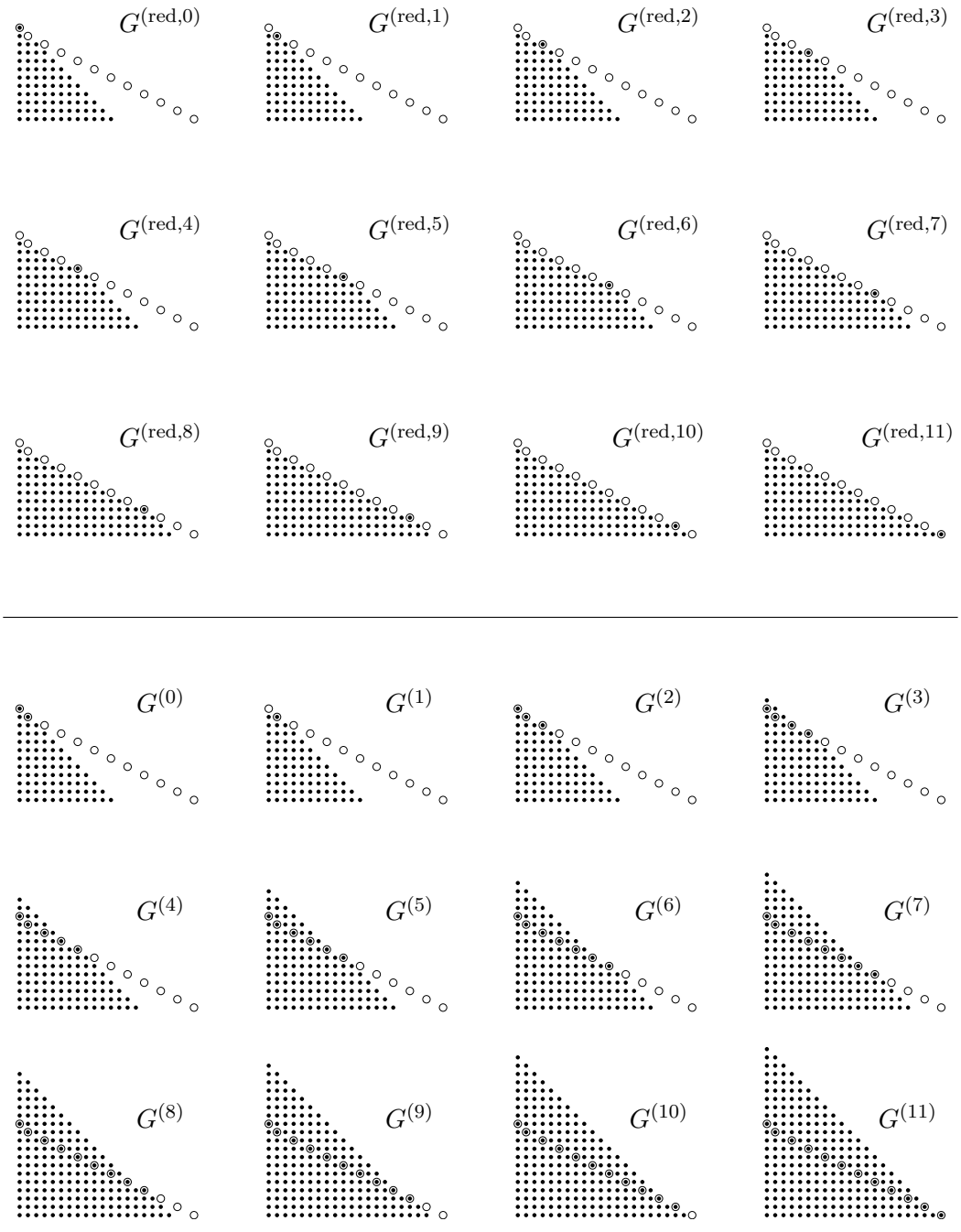


Figure 8.1: The difference between the reduced Gröbner basis (above) and the non-reduced one obtained as in (8.3–8.5) (below).

8.2 Concise representations of Gröbner bases

Unlike in the vanilla setting where suitable relations among elements of G were *assumed to exist*, for the grevlex basis considered here, we *explicitly know them* thank to the simple recurrence relations (8.3–8.5). This allows us to construct a *concise representation* in $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$ space on the same principle, but now we are even able to compute this representation efficiently.

8.2.1 Preparing the construction

The first task is to determine at which precision we can truncate the basis elements, depending on the degrees of the quotients. With the leading monomials as in (8.1–8.2), the dichotomic selection strategy from section 6.2.1 gives the following bound:

Lemma 8.3. *Let $(G^{(0)}, \ldots, G^{(n)})$ be a grevlex Gröbner basis as in (8.3–8.5), and let $Q^{(0)}, \ldots, Q^{(n)}$ be the quotients obtained with the dichotomic selection strategy from section 6.2.1. Then the bound*

$$\deg Q^{(i)} < 3 \times 2^{\mathrm{val}_2\, i}$$

holds for all $0 < i < n$.

Proof. Denote $\ell := 2^{\mathrm{val}_2\, i}$. With the same arguments as for Lemma 7.3, we have $\deg_X Q^{(i)} < 2\ell$ and $\deg_Y Q^{(i)} < \ell$. □

Naturally, the truncations are done according to the analogue of Definition 7.5 for the total degree order (i.e. $k = 1$ with the notations from Definition 7.5):

Definition 8.2. Given a polynomial $P \in \mathbb{K}[X, Y]$, we define its *upper truncation with precision p* as the polynomial $P^{\#}$ such that

- all terms of $P^{\#}$ of degree less than $\deg P - p$ are zero;
- all terms of $P^{\#}$ of degree at least $\deg P - p$ are equal to the corresponding terms in P.

The second task is to make the relations consistent with these degree bounds, so that each time a substitution happens, the precision actually increases. The formulas (8.3–8.5) are not satisfactory on this matter: for example $G^{(8)}$ would be expressed as a function of $G^{(7)}$, which is known with lower precision. However, from relations of the type

$$G^{(k+2)} = U^{(k)} G^{(k)} + V^{(k)} G^{(k+1)} \text{ for } k = 0, \ldots, n-2\,,$$

it is easy to deduce higher order recurrence relations

$$G^{(k+\ell)} = U^{(k,\ell)} G^{(k)} + V^{(k,\ell)} G^{(k+1)} \text{ for any } k \text{ and } \ell \text{ with } k + \ell \leqslant n.$$

With such relations, it is now possible to increase $\mathrm{val}_2\, i$ (hence the precision) at each substitution. It is convenient to write such recurrences in matrix form

$$\begin{pmatrix} G^{(k+\ell)} \\ G^{(k+\ell+1)} \end{pmatrix} = \mathbf{M}^{(k,\ell)} \begin{pmatrix} G^{(k)} \\ G^{(k+1)} \end{pmatrix}, \tag{8.6}$$

where, by convention, $G^{(n+1)} := 0$ to avoid case distinction. This presentation has the advantage that the $\mathbf{M}^{(k,\ell)}$ can be computed from one another using

$$\mathbf{M}^{(k,\ell+t)} = \mathbf{M}^{(k+\ell,t)}\mathbf{M}^{(k,\ell)}.$$

Moreover, the size of the coefficients of the $\mathbf{M}^{(k,\ell)}$ can be controlled as a function of k, ℓ. Notice that similar matrices appear in the half gcd algorithm [GG13, Chapter 11], which is the fastest known method for the computation of gcds.

Remark 8.3. To ensure that the precision increases after substitutions like (8.6), it is necessary that $G^{(k)}$ *and* $G^{(k+1)}$ have higher precision than $G^{(k+\ell)}, G^{(k+\ell+1)}$. Because of this, the precision of $G^{(i,\#)}$ will be $3 \times 2^{v(i)}$ with

$$v(i) := \max(\mathrm{val}_2\, i, \mathrm{val}_2(i-1)).$$

Notice that it changes the asymptotic number of required coefficients only by a constant factor.

More formally, the matrices $\mathbf{M}^{(k,\ell)}$ are defined as follows.

Definition 8.3. For $k = 2, \ldots, n$, let $u_k Z + v_k := D^{(k-2)} \operatorname{quo} D^{(k-1)}$ be the successive quotients in the Euclidean algorithm for the dominant diagonals

$$D^{(0)} := \mathrm{Diag}(G^{(0)}) \text{ and } D^{(1)} := \mathrm{Diag}(G^{(1)})$$

as in section 8.1. For each k, ℓ with $k + \ell \leqslant n$, define the matrix $\mathbf{M}^{(k,\ell)}$ by

$$\begin{aligned}
\mathbf{M}^{(0,1)} &:= \begin{pmatrix} 0 & 1 \\ X^{m-n+1} & -u_2 Y - v_2 X \end{pmatrix}, \\
\mathbf{M}^{(k,1)} &:= \begin{pmatrix} 0 & 1 \\ X^2 & -u_{k+2} Y - v_{k+2} X \end{pmatrix}, \text{ for } 0 < k < n-1, \\
\mathbf{M}^{(n-1,1)} &:= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \\
\mathbf{M}^{(k,\ell+1)} &:= \mathbf{M}^{(k+\ell,1)}\mathbf{M}^{(k,\ell)}.
\end{aligned}$$

As promised, they verify equation (8.6) and the degrees of the coefficients can be controlled:

Proposition 8.4. *Let the matrices $\mathbf{M}^{(k,\ell)}$ be as in Definition 8.3. For all k, ℓ with $k + \ell \leqslant n$, we then have*

$$\begin{pmatrix} G^{(k+\ell)} \\ G^{(k+\ell+1)} \end{pmatrix} = \mathbf{M}^{(k,\ell)} \begin{pmatrix} G^{(k)} \\ G^{(k+1)} \end{pmatrix}.$$

Also, $\mathbf{M}^{(k,\ell+t)} = \mathbf{M}^{(k+\ell,t)}\mathbf{M}^{(k,\ell)}$ for all k, ℓ, t with $k + \ell + t \leqslant n$.

Now consider the polynomials $U^{(k,\ell)}, V^{(k,\ell)}, \tilde{U}^{(k,\ell)}, \tilde{V}^{(k,\ell)}$ such that

$$\mathbf{M}^{(k,\ell)} = \begin{pmatrix} U^{(k,\ell)} & V^{(k,\ell)} \\ \tilde{U}^{(k,\ell)} & \tilde{V}^{(k,\ell)} \end{pmatrix}.$$

With the convention that the zero polynomial is homogeneous of any degree, we have

- *For all k, $V^{(k,\ell)}$ is homogeneous of degree $\ell - 1$ and $\tilde{V}^{(k,\ell)}$ is homogeneous of degree ℓ.*
- *$U^{(0,\ell)}$ is homogeneous of degree $m - n - 1 + \ell$ and $\tilde{U}^{(0,\ell)}$ is homogeneous of degree $m - n + \ell$.*
- *For all $k \geqslant 1$, $U^{(k,\ell)}$ is homogeneous of degree ℓ and $\tilde{U}^{(k,\ell)}$ is homogeneous of degree $\ell + 1$.*

Proof. This is immediate, by induction on ℓ, while using the formula

$$\mathbf{M}^{(k,\ell+1)} = \mathbf{M}^{(k+\ell,1)}\mathbf{M}^{(k,\ell)}. \qquad \square$$

8.2.2 Definition of the concise representation

Following remark 8.3 on the precision of the truncations, we can now define the concise representation.

Definition 8.4. The *concise representation* of $G := (G^{(0)}, \ldots, G^{(n)})$ consists of the following data:

- The sequence of truncated elements $G^{(0,\#)}, \ldots, G^{(n,\#)}$, where
  - $G^{(i,\#)} := G^{(i)}$ for $i \in \{0, 1, n\}$;
  - $G^{(i,\#)}$ is the upper truncation of $G^{(i)}$ at precision $3 \times 2^{\max(\mathrm{val}_2\, i, \mathrm{val}_2(i-1))}$ for all other i;
- For each $\lambda = 0, \ldots, \lceil \log_2 n \rceil$, the collection of rewriting matrices

$$\bar{\mathbf{M}}^{(\lambda)} := (\mathbf{M}^{(0,2^\lambda)}, \mathbf{M}^{(2^\lambda,2^\lambda)}, \ldots, \mathbf{M}^{(2^\lambda q, r)}),$$

  with $q := (n-1) \operatorname{quo} 2^\lambda$, $r := (n-1) \operatorname{rem} 2^\lambda + 1$, and the matrices $\mathbf{M}^{(k,\ell)}$ as in Definition 8.3.

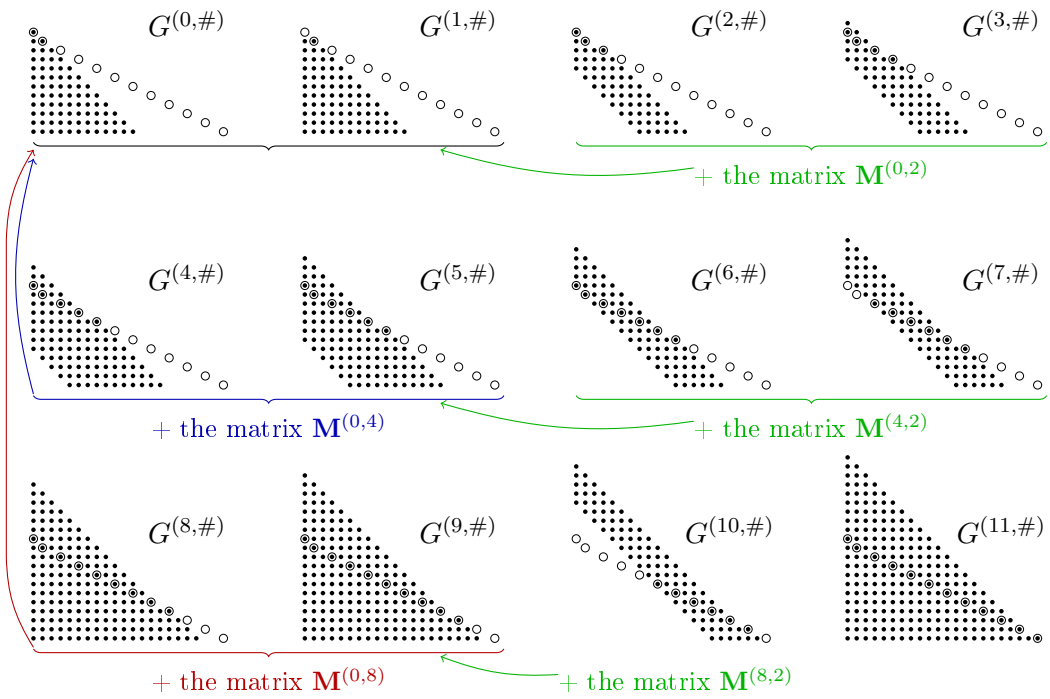


Figure 8.2: The concise representation of the Gröbner basis when $\deg A = \deg B = 11$.

An example of a concise representation is given in Figure 8.2. Notice that the truncated polynomial $G^{(2,\#)}$ contains much fewer terms than the corresponding $G^{(2)}$ (as seen in Figure 8.1). Notice also that the rewriting matrices allow us to express some elements in terms of others known with higher precision: for example $G^{(6)}$ and $G^{(7)}$ are expressed in terms of $G^{(4)}$ and $G^{(5)}$, themselves written as a function of $G^{(0)}$ and $G^{(1)}$.

The concise representation requires quasi-linear space with respect to the degree of the ideal:

Proposition 8.5. *The concise representation requires $O(nm \log n)$ space.*

Proof. It is easy to see that $G^{(i)}$ has degree at most $n + i \leqslant 2n$ in the variable Y, and at most $m + i \leqslant 2m$ in the variable X and in total degree. Then for $i = 0, 1, n$, each non-truncated element $G^{(i,\#)} := G^{(i)}$ takes $O(nm)$ space. Similarly, for each $i = 2, \ldots, n-1$, each truncated element $G^{(i,\#)}$ requires $O(m\,p(i))$ space, with

$$p(i) := 3 \times 2^{\max(\mathrm{val}_2\, i, \mathrm{val}_2(i-1))}\,.$$

For $\lambda = 1, \ldots, \lceil \log_2 n \rceil$, there are roughly $2n/2^\lambda$ indices i such that

$$\max(\mathrm{val}_2\, i, \mathrm{val}_2(i-1)) = \lambda\,,$$

so all elements together require $O(mn \log n)$ space.

There are $\lceil n/2^\lambda \rceil$ elements in $\bar{\mathbf{M}}^{(\lambda)}$, each consisting of four homogeneous polynomials of degree roughly 2^λ (by Proposition 8.4), except for $\mathbf{M}^{(0,2^\lambda)}$ that has two

polynomial entries of degree roughly 2^λ and two entries of degree roughly $m-n+2^\lambda$. Hence $\bar{\mathbf{M}}^{(\lambda)}$ requires $O(m)$ space and the collection of all $\bar{\mathbf{M}}^{(\lambda)}$ takes $O(m \log n)$ space. □

8.2.3 Computing concise Gröbner bases

The definition of the concise representation as above is constructive, but the order of the computation must be carefully chosen to achieve the desired quasi-linear complexity. First, by exploiting the recurrence relations $\mathbf{M}^{(k,\ell+t)} = \mathbf{M}^{(k+\ell,t)}\mathbf{M}^{(k,\ell)}$, it is easy to compute $\bar{\mathbf{M}}^{(\lambda+1)}$ from $\bar{\mathbf{M}}^{(\lambda)}$ using the following auxiliary function:

Algorithm 8.1. Computation of rewriting matrices

Prototype: `rewriting_2l`$(\bar{\mathbf{M}}^{(\lambda)})$
Input: The vector of matrices $\bar{\mathbf{M}}^{(\lambda)}$ as in Definition 8.4.
Output: The vector of matrices $\bar{\mathbf{M}}^{(\lambda+1)}$ as in Definition 8.4.

1: Set $L := \#\bar{\mathbf{M}}^{(\lambda)}$.
2: **for all** $i < L \operatorname{quo} 2$ **do**
3: Set $\bar{\mathbf{M}}_i^{(\lambda+1)} := \bar{\mathbf{M}}_{2i+1}^{(\lambda)}\bar{\mathbf{M}}_{2i}^{(\lambda)}$.
4: **end for**
5: **if** $L \operatorname{rem} 2 = 1$ **then**
6: Set $\bar{\mathbf{M}}_{L \operatorname{quo} 2}^{(\lambda+1)} := \bar{\mathbf{M}}_{L-1}^{(\lambda)}$.
7: **end if**
8: **return** $\bar{\mathbf{M}}^{(\lambda+1)}$.

Lemma 8.6. *Algorithm 8.1 is correct and takes time* $O(\mathsf{M}(m))$.

Proof. Recall that $\mathbf{M}^{(k,\ell+t)} = \mathbf{M}^{(k+\ell,t)}\mathbf{M}^{(k,\ell)}$ for all k, ℓ, t with $k + \ell + t \leqslant n$. In particular, taking $\ell = t = 2^\lambda$ shows that

$$\bar{\mathbf{M}}_i^{(\lambda+1)} = \bar{\mathbf{M}}_{2i+1}^{(\lambda)}\bar{\mathbf{M}}_{2i}^{(\lambda)}$$

if $2i + 1 < L - 1$. Concerning the last element of $\bar{\mathbf{M}}^{(\lambda+1)}$ (that is, $j = L \operatorname{quo} 2 - 1$ if L is even, $j = L \operatorname{quo} 2$ otherwise), define

$$\begin{aligned} q &:= (n-1) \operatorname{quo} 2^\lambda, \\ r &:= (n-1) \operatorname{rem} 2^\lambda + 1, \\ q' &:= (n-1) \operatorname{quo} 2^{\lambda+1}, \\ r' &:= (n-1) \operatorname{rem} 2^{\lambda+1} + 1. \end{aligned}$$

If $L \operatorname{rem} 2 = 0$, then q is odd, that is $q' = (q-1)/2$ and $r' = r + 2^\lambda$, so that $\mathbf{M}^{(2^{\lambda+1}q',r')} = \mathbf{M}^{(2^\lambda q,r)}\mathbf{M}^{(2^\lambda(q-1),2^\lambda)}$ is indeed the product of the last two elements of $\bar{\mathbf{M}}^{(\lambda)}$. Conversely if $L \operatorname{rem} 2 = 1$, then $q' = q/2$ and $r' = r$ so that $\bar{\mathbf{M}}^{(\lambda)}$ and $\bar{\mathbf{M}}^{(\lambda+1)}$ have the same last element.

The complexity bound is obtained with the same argument as for Proposition 8.5. □

The algorithm to compute the concise representation can be decomposed into three steps. First a Euclidean algorithm gives recurrence relations of order 1. Then we deduce higher order relations using the above algorithm. Finally, one has to compute the truncated basis elements $G^{(i,\#)}$. Starting with those of highest precision avoids computing unnecessary terms, so that quasi-linear complexity can be achieved.

Algorithm 8.2. Computation of a concise Gröbner basis

Prototype: `concise_repr`(A, B)

Input: Two generic polynomials $A, B \in \mathbb{K}[X, Y]$ of total degrees n and m with $n \leqslant m$.

Output: $(G^{\#}, \bar{\mathbf{M}})$, the concise representation of a Gröbner basis of $I := \langle A, B\rangle$ with respect to $\prec$.

1: Set $G^{(0,\#)} := A$ and $G^{(1,\#)} := B \operatorname{rem} A$.
2: Set $D^{(0)} := \operatorname{Diag}(G^{(0,\#)})$ and $D^{(1)} := \operatorname{Diag}(G^{(1,\#)})$.
3: **for** $i = 2, \ldots, n$ **do** ▷ Fail if the quotient has degree > 1
4: Set $D^{(i)} := D^{(i-2)} \operatorname{rem} D^{(i-1)}$ and $u_i Z + v_i := D^{(i-2)} \operatorname{quo} D^{(i-1)}$.
5: **if** $i = 2$ **then** set $d_i := m - n + 1$ **else** set $d_i := 2$. **end if**
6: Set $\mathbf{M}^{(i-2,1)} := \begin{pmatrix} 0 & 1 \\ X^{d_i} & -u_i Y - v_i X \end{pmatrix}$.
7: **end for**
8: Set $\mathbf{M}^{(n-1,1)} := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $\bar{\mathbf{M}}^{(0)} := (\mathbf{M}^{(0,1)}, \ldots, \mathbf{M}^{(n-1,1)})$.
9: **for** $\lambda = 0, \ldots, \lceil \log_2 n \rceil - 1$ **do**
10: Compute $\bar{\mathbf{M}}^{(\lambda+1)} := \texttt{rewriting\_2l}(\bar{\mathbf{M}}^{(\lambda)})$.
11: **end for**
12: Compute $\begin{pmatrix} G^{(n,\#)} \\ 0 \end{pmatrix} := M_{0,n} \begin{pmatrix} G^{(0,\#)} \\ G^{(1,\#)} \end{pmatrix}$. ▷ use $\bar{\mathbf{M}}^{(\lceil \log_2 n \rceil)} = (M_{0,n})$
13: **for** $\lambda = \lceil \log_2 n \rceil - 1, \ldots, 1$ **do**
14: **for** $j = 2, \ldots, n-1$ with $\operatorname{val}_2 j = \lambda$ **do**
15: Set $k := j - 2^\lambda$.
16: Set $\tilde{G}^{(k,\#)}, \tilde{G}^{(k+1,\#)} := G^{(k,\#)}, G^{(k+1,\#)}$, truncated at precision $3 \times 2^\lambda$.
17: Compute $G^{(j,\#)}, G^{(j+1,\#)}$ by ▷ use $\bar{\mathbf{M}}^{(\lambda)} = (\mathbf{M}^{(0,2^\lambda)}, \mathbf{M}^{(2^\lambda,2^\lambda)}, \ldots)$

$$\begin{pmatrix} G^{(j,\#)} \\ G^{(j+1,\#)} \end{pmatrix} := \mathbf{M}^{(k,2^\lambda)} \begin{pmatrix} \tilde{G}^{(k,\#)} \\ \tilde{G}^{(k+1,\#)} \end{pmatrix}.$$

18: Truncate $G^{(j,\#)}, G^{(j+1,\#)}$ at precision $3 \times 2^\lambda$.
19: **end for**
20: **end for**
21: **return** $G^{\#} := (G^{(0,\#)}, \ldots, G^{(n,\#)})$ and $\bar{\mathbf{M}} := (\bar{\mathbf{M}}^{(0)}, \ldots, \bar{\mathbf{M}}^{(\lceil \log_2 n \rceil)})$.

Theorem 8.7. *Algorithm 8.2 is correct and takes time* $O\big(\mathsf{R}(m^2) + \mathsf{M}(nm)\log(n)\big)$.

Proof. The reduction $G^{(1,\#)} := B \operatorname{rem} A$ can be done in a relaxed way in time $O\big(\mathsf{R}(m^2)\big)$. The first loop (lines 3-7) is clearly correct and each step requires

$O(\mathsf{M}(n))$ operations. Alternatively, the successive quotients can all be computed with the fast "half gcd" algorithm (see for example [GG13, Chapter 11]) using $O(\mathsf{M}(n)\log n)$ operations.

For a polynomial P, denote $\pi_p^{\#}(P)$ for the upper truncation of P at precision p. In the last loop on λ (lines 13-20), since the precision decreases at each step and since there is no accidental cancellation, the invariant

$$\begin{pmatrix} G^{(j,\#)} \\ G^{(j+1,\#)} \end{pmatrix} = \pi^{\#}_{3\times 2^\lambda} \begin{pmatrix} G^{(j)} \\ G^{(j+1)} \end{pmatrix} \text{ for all } j \text{ with } \mathrm{val}_2\, j = \lambda$$

holds. Indeed, regarding upper truncations, it is clear that $\pi_u^{\#}(P) = \pi_u^{\#}(\pi_v^{\#}(P))$ as soon as $u \leqslant v$, and also that $\pi_u^{\#}(PQ) = \pi_u^{\#}(\pi_u^{\#}(P)Q)$. Then we have

$$(\tilde{G}^{(k,\#)}, \tilde{G}^{(k+1,\#)}) = \pi^{\#}_{3\times 2^\lambda}(G^{(k)}, G^{(k+1)})\,,$$

hence

$$\pi^{\#}_{3\times 2^\lambda}\left(\mathbf{M}^{(k,2^\lambda)} \begin{pmatrix} \tilde{G}^{(k,\#)} \\ \tilde{G}^{(k+1,\#)} \end{pmatrix}\right) = \pi^{\#}_{3\times 2^\lambda}\left(\mathbf{M}^{(j-2^\lambda,2^\lambda)} \begin{pmatrix} G^{(j-2^\lambda)} \\ G^{(j-2^\lambda+1)} \end{pmatrix}\right),$$

which proves the correctness. Let us now evaluate the complexity of this loop. For each index j such that $\mathrm{val}_2\, j = \lambda$, the support of $\tilde{G}^{(k,\#)}, \tilde{G}^{(k+1,\#)}$ has size $O(m2^\lambda)$. Consequently, each iteration of the loop requires $O(\mathsf{M}(nm))$ operations. □

8.3 Fast reduction with respect to concise Gröbner bases

In this section, we design an algorithm to compute an extended reduction with quasi-optimal complexity, using the concise representation. Recall that the major obstruction for such a result was that the equation

$$P = Q^{(0)}G^{(0)} + \cdots + Q(n)G^{(n)} + R\,. \tag{8.7}$$

is much larger than the intrinsic complexity of the problem (i.e. the size of A, B, P). On the one hand, the concise representation solves this issue by providing essential information about $G = (G^{(0)}, \ldots, G^{(n)})$ using much less space. On the other hand, recall that for vanilla Gröbner bases as in Chapter 7, all $G^{(i)}$ had roughly the same degree, and notice that this is not the case here. Therefore, quotients with respect to $G^{\#}$ are not valid with respect to G and the algorithm from [Hoe15] can no longer be used in a blackbox manner.

8.3.1 Revisiting the relaxed reduction algorithm

The reduction algorithm for concise Gröbner bases is based on a modification of the algorithm from [Hoe15]. The present subsection gives more details about [Hoe15] to help understanding the modifications required later.

As mentioned in section 6.4.3, the idea is to rewrite equation (8.7) in terms of power series arithmetic. As an introductory example, let us perform a Euclidean

division in one variable using this method. So, let $f(Z), g(Z)$ be univariate polynomials with $\deg f > \deg g$, and assume to simplify that g is monic. We wish to find $q(Z), r(Z)$ such that $f = gq + r$ and $\deg g > \deg r$. This can also we written as

$$Z^{\deg g} q + r = f - (g - Z^{\deg g})q\,,$$

and we notice that the term of degree k in q is the term of degree $k + \deg g$ in $f - (g - Z^{\deg g})q$, that depends only on $q_{k+1}, q_{k+2}, \ldots, q_{\deg q}$. Seeing q as a stream of coefficients starting with the terms of highest degree, we get an equation on power series that is indeed recursive.

The algorithm from [Hoe15] is the multivariate generalization of this idea: for an extended reduction as in (8.7), we have

$$\begin{aligned} Q^{(0)} \operatorname{lt}(G^{(0)}) + \cdots + Q^{(n)} \operatorname{lt}(G^{(n)}) + R = & P - Q^{(0)}T^{(0)} - \cdots - Q^{(n)}T^{(n)} \\ & \text{with } T^{(i)} := G^{(i)} - \operatorname{lt} G^{(i)}. \end{aligned}$$

Assuming the streams of coefficients of each $Q^{(i)}$ are given in decreasing order with respect to $\prec$, the equation above is recursive and can be evaluated with series arithmetic (more precisely, by computing each product $Q^{(i)}T^{(i)}$ using relaxed multiplications).

8.3.2 Exploiting the concise representation

The concise representation contains only truncated variants $G^{(i,\#)}$ of the $G^{(i)}$. We can set similarly $T^{(i,\#)} := G^{(i,\#)} - \operatorname{lt}(G^{(i,\#)})$, then we observe that the formula

$$P - Q^{(0)}T^{(0,\#)} - \cdots - Q^{(n)}T^{(n,\#)}$$

is much smaller than the formula

$$P - Q^{(0)}T^{(0)} - \cdots - Q^{(n)}T^{(n)}$$

so that relaxed multiplications will compute the former polynomial much faster. However, since the terms of lower degree are dropped, the two streams of coefficients will diverge at some point. To avoid this, we will progressively rewrite equation (8.7) before this happens. More precisely, as soon as the quotient $Q^{(j)}$ is known, the product $Q^{(j)}G^{(j)}$ is replaced by some $S^{(k)}G^{(k)} + S^{(k+1)}G^{(k+1)}$, where $G^{(k)}, G^{(k+1)}$ are known with precision larger than $G^{(j)}$.

Remark 8.4. As in Chapter 7, we use truncated elements to speed-up the computation, followed by substitutions to maintain the correctness of the result. However, the substitutions are now done *on-the-fly* during the reduction algorithm, because it is not possible to wait until all quotients are known.

The concise representation contains the necessary information to perform these substitutions in the form of recurrence relations among the $G^{(k)}$: for any indices k, ℓ with $k + \ell \leqslant n$, we have

$$\begin{pmatrix} G^{(k+\ell)} \\ G^{(k+\ell+1)} \end{pmatrix} = \mathbf{M}^{(k,\ell)} \begin{pmatrix} G^{(k)} \\ G^{(k+1)} \end{pmatrix}. \tag{8.8}$$

For the correctness of the algorithm, we will need the following result:

Lemma 8.8. *Let $G^{(0)}, \dots, G^{(n)}$ be a Gröbner basis and let the matrices $\mathbf{M}^{(k,\ell)}$ be as in Definition 8.3 (for all indices k, ℓ such that $k + \ell \leqslant n$). Given quotients $Q^{(j)}, Q^{(j+1)}$ with $j := k + \ell$, define*

$$(S^{(k)}, S^{(k+1)}) := (Q^{(j)}, Q^{(j+1)})\mathbf{M}^{(k,\ell)}.$$

Then we have

$$S^{(k)}G^{(k)} + S^{(k+1)}G^{(k+1)} = Q^{(j)}G^{(j)} + Q^{(j+1)}G^{(j+1)}. \tag{8.9}$$

Assume now that λ is such that $Q^{(j)}, Q^{(j+1)}$ have degree less than $3 \times 2^{\lambda-1} - 1$ and $\ell < 2^\lambda$. Then:

- *If $k > 0$, then $S^{(k)}, S^{(k+1)}$ have degree less than $3\times 2^\lambda - 1$, and can be computed in time $O\left(\mathsf{M}(4^\lambda)\right)$.*
- *If $k = 0$, then $\deg S^{(k)} < m - n + 3 \times 2^\lambda$, $\deg S^{(k+1)} < 3 \times 2^\lambda - 1$ and they can be computed in time $O\left(\mathsf{M}((m-n)2^\lambda + 4^\lambda)\right)$.*

Proof. Equation (8.9) is an immediate consequence of the recurrence relations (8.8) among the $G^{(i)}$. Similarly, the bounds on $\deg S^{(k)}$ are consequences of the degree bounds from Proposition 8.4. □

8.3.3 Reduction algorithm

We can now adapt the extended reduction algorithm from [Hoe15] to perform these replacements during the computation. This leads to Algorithm 8.3 below and summarized in Figure 8.3. The reduction algorithm consists of two main tasks.

First one has to reduce terms one after the other and update the quotients accordingly (loop in lines 7-12). For example in Figure 8.3.a), the current term to be reduced is $t := \alpha X^{14}Y^6$, marked by a square; it is reduced against $G^{(6)}$ (with $\operatorname{lt} G^{(6)} = \beta X^{11}Y^5$) because of the dichotomic selection strategy, then we update the quotient $Q^{(6)} \mathrel{+}= \alpha/\beta X^3 Y$ to cancel this term. During the next step of the loop, the term to be reduced is $\gamma X^{15}Y^5$ marked by a diamond; this time it is reduced against $G^{(8)}$ which implies an update of $Q^{(8)}$, and so on.

For the second task, one has to rewrite the equation to maintain sufficient precision (loop in lines 13-25); this is possible because a new quotient is now entirely known. In our example on Figure 8.3.b), we just completed the computation of $Q^{(10)}$, so we find $S^{(8)}, S^{(9)}$ such that

$$Q^{(10)}G^{(10)} = S^{(8)}G^{(8)} + S^{(9)}G^{(9)}$$

and we perform the replacement in equation (8.7); this is represented by the green arrows in the picture.

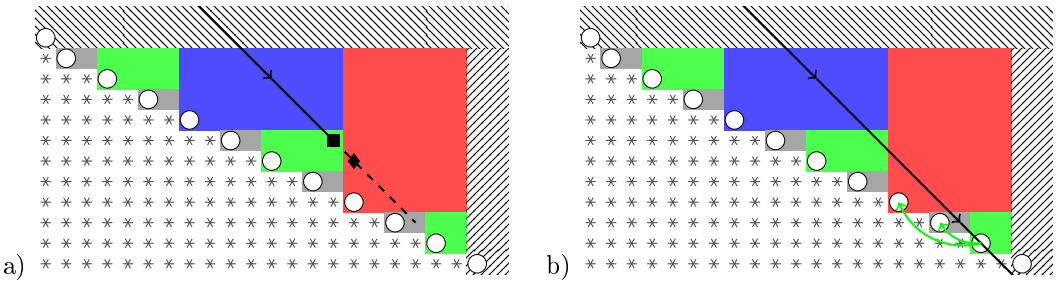


Figure 8.3: Summary of the reduction algorithm.

Algorithm 8.3. Extended reduction w.r.t. a concise Gröbner basis

Prototype: `reduce_concise`$(P, G^{\#}, \bar{\mathbf{M}})$

Input: A bivariate polynomial P of degree d, and a concise representation $(G^{\#}, \bar{\mathbf{M}})$ as in Definition 8.4.

Output: $(Q^{(0)}, \ldots, Q^{(n)}, R)$, the extended reduction of P with respect to G.

Set $(Q^{(0)}, \ldots, Q^{(n)}, R) := (0, \ldots, 0)$ and $P^{\text{subs}} := P$.
Set $(S^{(0)}, \ldots, S^{(n)}) := (0, \ldots, 0)$ ▷ substitutions as in Lemma 8.8
Set $\mathcal{I} := \{i \leqslant n : \deg G^{(i)} \leqslant d\} \cup \{n\}$. ▷ active indices for relaxed mult.
Set $T^{(i,\#)} := G^{(i,\#)} - \operatorname{lt} G^{(i,\#)}$ for each $i = 0, \ldots, n$.
for $d' = d, \ldots, 0$ **do** ▷ $\mathcal{J}$: active indices with $Q^{(i)} \neq 0$
    Set $\mathcal{J} := \{i \in \mathcal{I} : (i = 0) \vee (i = n) \vee (d' < \deg(G^{(i)}) + 3 \times 2^{\operatorname{val}_2(i)})\}$.
    for $a = 0, \ldots, d'$ **do**
        Set $c := P^{\text{subs}}_{a,d'-a}$.
        For each $i \in \mathcal{J}$, update $c \mathrel{-}= (Q^{(i)} T^{(i,\#)})_{a,d'-a}$ ▷ relaxed mult.
        Let $(i, \tau) := \Phi_G(cX^a Y^{d'-a})$. ▷ dichotomic selection (section 6.2.1)
        if $i < 0$ **then** update $R \mathrel{+}= \tau$ **else** update $Q^{(i)} := Q^{(i)} + \tau$. **end if**
    end for
    for all j (in decreasing order) such that $d' = \deg G^{(j)}$ **do** ▷ cf. Remark 8.5
        if $j < n$ **then**
            Update $S^{(j)} \mathrel{+}= Q^{(j)}$ and $\mathcal{I} := \mathcal{I} \setminus \{j\}$ and $P^{\text{subs}} \mathrel{-}= Q^{(j)} G^{(j,\#)}$
        end if
        if $1 < j < n$ and $\lambda := \operatorname{val}_2(j) > 0$ **then**
            Set $k := j - 2^{\lambda}$.
            Set $(\Delta^{(k)}, \Delta^{(k+1)}) := (S^{(j)}, S^{(j+1)})\mathbf{M}^{(k,2^{\lambda})}$.
            Update $P^{\text{subs}} \mathrel{-}= \Delta^{(k)} G^{(k,\#)} + \Delta^{(k+1)} G^{(k+1,\#)}$.
            Update $P^{\text{subs}} \mathrel{+}= S^{(j)} G^{(j,\#)} + S^{(j+1)} G^{(j+1,\#)}$.
            Update $(S^{(k)}, S^{(k+1)}) \mathrel{+}= (\Delta^{(k)}, \Delta^{(k+1)})$.
            Set $(S^{(j)}, S^{(j+1)}) := (0, 0)$.
        end if
    end for
end for
return $(Q^{(0)}, \ldots, Q^{(n)}, R)$.

Remark 8.5. Recall that $G^{(0)}$ has degree n and $G^{(i)}$ has degree $m+i-1$ for $i \geqslant 1$. Therefore, the loop on j such that $d' = \deg G^{(j)}$ in lines (13-25) is trivial: the set of such j contains at most 1 element, except when $d' = n = m$ in which case there are 2 elements (0 and 1).

Theorem 8.9. *Algorithm 8.3 is correct and runs in time*

$$O\left(\mathsf{R}(d^2) + \mathsf{R}(nm)\log n + \mathsf{M}(nm)\log^2 n\right) .$$

Proof. Let us first notice that the relaxed strategy can indeed be used. The quotients $Q^{(i)}$ are regarded as streams of coefficients. These coefficients are produced by the updates $Q^{(i)} \mathrel{+}= \tau$ and consumed in the relaxed evaluation of the products $Q^{(i)}T^{(i,\#)}$. We see that the coefficients are consumed in decreasing order w.r.t. $\prec$, and in this case the equation

$$Q^{(0)}\operatorname{lt}(G^{(0)}) + \cdots + Q^{(n)}\operatorname{lt}(G^{(n)}) + R = P - Q^{(0)}T^{(0)} - \cdots - Q^{(n)}T^{(n)} \tag{8.10}$$

is recursive as seen in 8.3.1. This ensures that the production of coefficients always occurs before their consumption.

For the correctness, we must check that Algorithm 8.3 actually evaluates equation (8.10), despite the truncations and substitutions. More precisely, we must show that for each d', at the start of each iteration of the loop on a (line 7), we have

$$\left(P^{\text{subs}} - \sum_{i\in\mathcal{J}} Q^{(i)}T^{(i,\#)}\right)_{a,d'-a} = \left(P - \sum_{i\leqslant n} Q^{(i)}T^{(i)}\right)_{a,d'-a} \tag{8.11}$$

Informally, equation (8.11) means that Algorithm 8.3 (left-hand side) and the algorithm from [Hoe15] (right-hand side) produce the same streams of coefficients; hence the correctness of [Hoe15] implies the correctness of Algorithm 8.3.

Let us start with the description of a few invariants at the start of the main loop on d' (line 6):

I1 $P^{\text{subs}} = P - \sum_{i\leqslant n} S^{(i)}G^{(i,\#)}$.

I2 $\sum_{i\leqslant n} S^{(i)}G^{(i)} = \sum_{i\notin\mathcal{I}} Q^{(i)}G^{(i)}$.

I3 $\deg S^{(i)}$ and $\deg S^{(i+1)}$ are at most $3\times 2^{\operatorname{val}_2(i)} - 1$ for all even $i \in \{1,\ldots,n-1\}$.

I4 For some $i_0 \leqslant n+1$, we have $\mathcal{I} = \{0,\ldots,i_0-1\} \cup \{n\}$.

I5 With i_0 as above, if $i = n$ or $i > i_0$ or ($i \geqslant i_0$ with i_0 even), then $S^{(i)} = 0$.

Invariant I1 is immediate. Invariants I2 and I3 follow from Lemma 8.8, using $\deg(Q^{(i)}) < 3\times 2^{\operatorname{val}_2(i)}-1$ by Lemma 8.3. For invariant I4, we recall that $\deg G^{(i')} \leqslant \deg G^{(i'+1)}$ for all $i' < n$. Finally, invariant I5 is preserved by the loop on lines 13-25: whenever j is removed from $\mathcal{I}$, $S^{(j)}$ and $S^{(j+1)}$ are set to 0 if j is even.

Let us now prove the main claim (8.11). Notice first that if $0 < i < n$ and $i \in \mathcal{I}$, then $\deg G^{(i)} \leqslant d'$. Recall also that $\deg Q^{(i)} < 3 \times 2^{\mathrm{val}_2(i)} - 1$ for $0 < i < n$. This means $\deg(Q^{(i)}G^{(i)}) < d'$ for $i \in \mathcal{I} \setminus \mathcal{J}$. Since by definition

$$G^{(0,\#)} = G^{(0)}, G^{(n,\#)} = G^{(n)}, \text{ and } G^{(i,\#)} = \pi^{\#}_{3\times 2^{v(i)}}(G^{(i)}) \text{ for } 0 < i < n\,,$$

again where $\pi^{\#}_p(P)$ denotes the upper truncation of P at precision p, and $v(i)$ defined as $\max\big(\mathrm{val}_2(i), \mathrm{val}_2(i-1)\big)$, we deduce that

$$\left(\sum_{i\in\mathcal{J}} Q^{(i)}T^{(i,\#)}\right)_{a,d'-a} = \left(\sum_{i\in\mathcal{I}} Q^{(i)}T^{(i,\#)}\right)_{a,d'-a} = \left(\sum_{i\in\mathcal{I}} Q^{(i)}T^{(i)}\right)_{a,d'-a}.$$

To complete the proof (by invariant I1), we show that

$$\begin{aligned}\left(\sum_{i\leqslant n} S^{(i)}G^{(i,\#)}\right)_{a,d'-a} &= \left(\sum_{i\leqslant n} S^{(i)}G^{(i)}\right)_{a,d'-a}\\ &= \left(\sum_{i\notin\mathcal{I}} Q^{(i)}G^{(i)}\right)_{a,d'-a}\\ &= \left(\sum_{i\notin\mathcal{I}} Q^{(i)}T^{(i)}\right)_{a,d'-a}.\end{aligned}$$

For the first identity, we contend that $S^{(i)}G^{(i)}$ and $S^{(i)}G^{(i,\#)}$ have the same terms of degree d' because of invariants I3 and I5. This is clear if $S^{(i)} = 0$, or if $i \leqslant 1$ since $G^{(0)} = G^{(0,\#)}$ and $G^{(1)} = G^{(1,\#)}$. Assume therefore that $i > 1$ and $S^{(i)} \neq 0$. By invariant I5, the index $i_0 := \min\{i \in \mathbb{N}, i \notin \mathcal{I}\}$ verifies $i \leqslant i_0 < n$, hence i_0 was removed from $\mathcal{I}$ during a previous iteration of the loop on d'. Since $\deg G^{(i'+1)} = \deg G^{(i')} + 1$ for all $0 < i' < n$, this actually happened during the previous iteration (with $d'+1$ instead of d'). It follows that $\deg G^{(i)} \leqslant d'+1 = \deg G^{(i_0)}$. By definition of $G^{(i,\#)}$ and invariant I3, the polynomials $S^{(i)}G^{(i)}$ and $S^{(i)}G^{(i,\#)}$ have the same terms of degree at least $\deg G^{(i)} - 3\times 2^{\mathrm{val}_2 i} + \deg S^{(i)}$ if i is even, or at least $\deg G^{(i)} - 3\times 2^{\mathrm{val}_2(i-1)} + \deg S^{(i)}$ if i is odd. In both cases, this degree bound is $\leqslant d'$.

The second identity follows immediately from invariant I2. The last identity follows from the implication $i \notin \mathcal{I} \Rightarrow d' < \deg G^{(i)}$, so that $Q^{(i)}G^{(i)}$ and $Q^{(i)}T^{(i)}$ have the same terms of degree d'.

For the complexity, relaxed multiplications are used to compute the coefficients of the $Q^{(i)}T^{(i,\#)}$, whose support is a subset of the support of $Q^{(i)}G^{(i,\#)}$. Then the relaxed multiplications take time

$$\mathsf{R}(|Q^{(0)}G^{(0,\#)}|) + \cdots + \mathsf{R}(|Q^{(n)}G^{(n,\#)}|) = O\left(\mathsf{R}(d^2) + \mathsf{R}(nm)\log n\right).$$

It remains to evaluate the cost of the zealous multiplications during the rewriting steps. To avoid case distinctions, given an even index $k \in \{0,\dots,n-1\}$, let $\overline{\mathrm{val}_2}(k)$ be defined as

$$\overline{\mathrm{val}_2}k := \begin{cases} \lceil \log_2 n \rceil & \text{if } k = 0,\\ \mathrm{val}_2\, k & \text{otherwise.}\end{cases}$$

Then for every such index k and for every $\lambda < \overline{\mathrm{val}_2} k$, there is an update of $S^{(k)}, S^{(k+1)}$, of cost $O\left(\mathsf{M}(4^\lambda)\right)$, followed by the evaluation of the products $S^{(k)}G^{(k,\#)}$ and $S^{(k+1)}G^{(k+1,\#)}$, which takes

$$O\left(\mathsf{M}\left(m2^{\overline{\mathrm{val}_2} k}\right)\right)$$

operations. This leads to a total of $O\left(\mathsf{M}\left(m2^{\overline{\mathrm{val}_2} k}\right)\log_2 n\right)$ operations. Summing over all k, we get a total cost of $O\left(\mathsf{M}(nm)\log^2 n\right)$ for all rewriting steps. □

8.4 Applications

Under some regularity assumptions, we designed a quasi-linear algorithm for polynomial reduction, but unlike Chapter 7, it does not rely on expensive precomputations. This leads to significant improvements in the asymptotic complexity for various problems. To illustrate the gain, let us assume to simplify that $n = m$, and neglect logarithmic factors. Then, ideal membership test and modular multiplication are essentially quadratic in n. Also, computing the reduced Gröbner basis has cubic complexity. In all these examples, the bound is intrinsically optimal, and corresponds to a speed-up by a factor n compared to the best previously known algorithms.

8.4.1 Ideal membership

From any fast algorithms for Gröbner basis computation and (multivariate) polynomial reduction, it is immediate to construct an ideal membership test:

Algorithm 8.4. Ideal membership in two variables

Input: (A, B, P), bivariate polynomials of degrees n, m, and d with $n \leqslant m$ and A, B generic.

Output: **true** if $P \in \langle A, B\rangle$, **false** otherwise.

1: Compute $(G^\#, \bar{\mathbf{M}}) := \texttt{concise\_repr}(A, B)$. ▷ Algorithm 8.2
2: Compute $(Q^{(0)}, \ldots, Q^{(n)}, R) := \texttt{reduce\_concise}(P, G^\#, \bar{\mathbf{M}})$. ▷ Algorithm 8.3
3: **return true** if $R = 0$, **false** otherwise.

Theorem 8.10. *Algorithm 8.4 is correct and takes time*

$$O\left(\mathsf{R}(m^2 + d^2) + \mathsf{R}(nm)\log n + \mathsf{M}(nm)\log^2 n\right).$$

8.4.2 Multiplication in the quotient algebra

The concise Gröbner basis provides a practical representation of the quotient algebra $\mathbb{A} := \mathbb{K}[X, Y]/\langle A, B\rangle$, that does not need more space (up to logarithmic factors) than the algebra itself, while still allowing for efficient computation. But unlike the terse representation from Chapter 7, it is easy to compute, so that multiplication in $\mathbb{A}$ can be done in quasi-linear time, including the cost for the precomputation:

Algorithm 8.5.

Input: (A, B, P, Q), bivariate polynomials, with A, B generic of degrees $n \leqslant m$ and $P, Q \in \mathbb{A}$ in normal form.
Output: $PQ \in \mathbb{A}$ in normal form.

1: Compute $(G^{\#}, \bar{\mathbf{M}}) := \texttt{concise\_repr}(A, B)$. ▷ Algorithm 8.2
2: Compute PQ using any (zealous) multiplication algorithm.
3: Compute $(Q^{(0)}, \ldots, Q^{(n)}, R) := \texttt{reduce\_concise}(P, G^{\#}, \bar{\mathbf{M}})$. ▷ Algorithm 8.3
4: **return** R.

Theorem 8.11. *Algorithm 8.5 is correct and takes time*

$$O\left(\mathsf{R}(m^2) + \mathsf{R}(mn)\log n + \mathsf{M}(nm)\log^2 n\right) .$$

8.4.3 Reduced Gröbner basis

Since we can reduce polynomials in quasi-linear time, we deduce a new method to compute the reduced Gröbner basis: first compute the non-reduced basis, together with additional information to allow the efficient reduction (which can be done fast); then reduce each element with respect to the others.

Algorithm 8.6. Reduced Gröbner basis in two variables

Input: (A, B), generic bivariate polynomials of total degrees n and m with $n \leqslant m$.
Output: $G^{\mathrm{red}} := (G^{(\mathrm{red},0)}, \ldots, G^{(\mathrm{red},n)})$ the reduced Gröbner basis of $\langle A, B\rangle$ with respect to $\prec$.

1: Compute $(G^{\#}, \bar{\mathbf{M}}) := \texttt{concise\_repr}(A, B)$.
2: **for** $i = 0, \ldots, n$ **do**
3: Set $t_0 := Y^n = \operatorname{lm} G^{(0)}$ or $t_i := X^{m-n-1+2i}Y^{n-i} = \operatorname{lm} G^{(i)}$ if $i > 0$.
4: Compute $(Q^{(0,i)}, \ldots, Q^{(n,i)}, R^{(i)}) := \texttt{reduce\_concise}(t_i, G^{\#}, \bar{\mathbf{M}})$.
5: Set $G^{(\mathrm{red},i)} := t_i - R^{(i)}$.
6: **end for**
7: **return** $(G^{(\mathrm{red},0)}, \ldots, G^{(\mathrm{red},n)})$.

Theorem 8.12. *Algorithm 8.6 is correct and takes time*

$$O\left(\mathsf{R}(m^2)n\log n + n\mathsf{M}(nm)\log^2 n\right) .$$

Proof. Clearly $G^{(\mathrm{red},i)}$ is in the ideal and has the same leading monomial as $G^{(i)}$. Moreover, $G^{(\mathrm{red},i)}$ is monic and none of its terms is divisible by the leading term of any $G^{(j)}$, $j \neq i$. This proves G^{red} is indeed the reduced Gröbner basis of $\langle A, B\rangle$ with respect to $\prec$. □

8.5 Refined complexity analysis

The algorithms from sections 8.3 and 8.4 were purposely simplified to separate the main ideas of more technical details. Although the complexity bounds are

satisfactory, there are specific cases in which they are not optimal, and a more subtle analysis is required to improve them. This section presents the missing ingredients and proves the complexity bounds announced in Table 8.1.

8.5.1 Optimized algorithm using lazier substitutions

The bound given in Theorem 8.9 contains an unwanted term $O\left(\mathsf{M}(nm)\log^2(n)\right)$ that corresponds to the rewriting steps. This contribution is absorbed by the term $O\left(\mathsf{R}(nm)\log(n)\right)$ when using traditional relaxed multiplication [FS74, Hoe02] with $\mathsf{R}(d) \asymp \mathsf{M}(d)\log d$, but this is no longer true for faster relaxed algorithms, such as the one from [Hoe14]. With some optimizations, it is possible to decrease by a logarithmic factor the cost of the rewriting steps. Then this contribution can be absorbed into the term $O\left(\mathsf{R}(nm)\log(n)\right)$, independently of the relaxed multiplication algorithm being used.

For the general idea, notice that each time a new quotient $Q^{(j)}$ is known, we perform a substitution $(S^{(k)}, S^{(k+1)}) \mathrel{+}= (S^{(j)}, S^{(j+1)})\mathbf{M}^{(k,2^\lambda)}$ $(j = k + 2^\lambda)$, followed by a few products of the form $S^{(k)}G^{(k,\#)}$. This means that there are a logarithmic number of products $S^{(k)}G^{(k,\#)}$ for each k. Now up to a few adaptations, it is possible to reduce this to a constant number. Summing over all k, the total cost of rewriting the equation then drops from $O\left(\mathsf{M}(nm)\log^2 n\right)$ down to $O\left(\mathsf{M}(nm)\log n\right)$.

The modification is essentially as follows:

1. instead of rewriting as soon as $Q^{(j)}$ is known, we delay the substitution until $Q^{(k)}$ is known;
2. we then perform all substitutions $(S^{(k)}, S^{(k+1)}) \mathrel{+}= (S^{(j)}, S^{(j+1)})\mathbf{M}^{(k,2^\lambda)}$ for each $\lambda, j = k + 2^\lambda$;
3. we only perform the multiplications by $G^{(k,\#)}, G^{(k+1,\#)}$ after this loop.

Since the substitution is delayed, it is necessary to increase the precision of $G^{(j,\#)}$, to ensure a correct result up to the degree of $G^{(k)}$. If $k > 0$, then we have $\deg G^{(j)} = \deg G^{(k)} + 2^\lambda$, so increasing the precision by 2^λ is sufficient. However, if $k = 0$, then $\deg G^{(j)} = \deg G^{(k)} + 2^\lambda + m - n$, and $m - n$ may be large; a naive increase of the precision would thus cause an undesirable overhead. A possible workaround is to add a "virtual" basis element $G^{(1/2)} := X^{m-n}G^{(0)}$, that is used only for the substitutions (and we have $\deg G^{(j)} = \deg G^{(1/2)} + 2^\lambda - 1$ so now the increased precision remains reasonable).

Definition 8.5. The *augmented concise representation* has the same content as the concise representation, up to the following:

- The basis elements $G^{(i,\#)}$ are truncated at precision $4 \times 2^{\max(\mathrm{val}_2\, i, \mathrm{val}_2(i-1))}$ (instead of $3 \times 2^{\cdots}$);
- An additional element $G^{(1/2,\#)} := X^{m-n}G^{(0)}$;

- Additional matrices $\mathbf{M}^{(1/2,2^\lambda)} := \mathbf{M}^{(0,2^\lambda)} \begin{pmatrix} 1/X^{m-n} \\ 1 \end{pmatrix}$ for each $\lambda \leqslant \lfloor \log_2 n \rfloor$.

The following theorem is a straightforward adaptation of Proposition 8.5 and Theorem 8.7.

Theorem 8.13. *The augmented concise representation requires $O(mn \log n)$ space and can be computed in time $O\left(\mathsf{R}(m^2) + \mathsf{M}(nm)\log(n)\right)$ using a suitable adaptation of Algorithm 8.2.*

Assuming from now on the augmented concise representation, the loop in lines 13-25 of Algorithm 8.3 can be modified as follows:

Algorithm 8.7. (Replaces the loop in lines 13-25 of Algorithm 8.3)

13: **for** k (in decreasing order) such that $d' = \deg G^{(k)}$ **do** ▷ See Remark 8.5
14: **if** $k < n$ **then** set $S^{(k)} := Q^{(k)}$ **end if**
15: **if** $k < n$ and $\lfloor k \rfloor$ is even **then**
16: **if** $k + 1 < n$ **then**
17: Update $\mathcal{I} := \mathcal{I} \setminus \{k, \lfloor k+1 \rfloor\}$.
18: **else**
19: Update $\mathcal{I} := \mathcal{I} \setminus \{k\}$.
20: **end if**
21: **end if**
22: **if** $k = 0$ **then** update $P^{\text{subs}} -= S^{(0)}G^{(0,\#)}$ **end if**
23: **if** $(1 < k < n$ and $\Lambda := \text{val}_2(k) > 0)$ or $k = 1/2$ **then**
24: **if** $k = 1/2$ **then** set $\Lambda := \lceil \log_2 n \rceil$. **end if**
25: **for** $\lambda \in \{1, 2, \ldots, \Lambda - 1\}$ such that $j := \lfloor k \rfloor + 2^\lambda < n$ **do**
26: Update $(S^{(k)}, S^{(\lfloor k+1 \rfloor)}) += (S^{(j)}, S^{(j+1)})\mathbf{M}^{(k,2^\lambda)}$.
27: Update $P^{\text{subs}} += S^{(j)}G^{(j,\#)} + S^{(j+1)}G^{(j+1,\#)}$.
28: Set $(S^{(j)}, S^{(j+1)}) := (0, 0)$.
29: **end for**
30: Update $P^{\text{subs}} -= S^{(k)}G^{(k,\#)} + S^{(\lfloor k+1 \rfloor)}G^{(\lfloor k+1 \rfloor,\#)}$.
31: **end if**
32: **end for**

Theorem 8.14. *The above modification of Algorithm 8.3 is correct and runs in time*

$$O\left(\mathsf{R}(d^2) + \mathsf{R}(nm)\log n\right).$$

Proof. The correctness proof is essentially the same as for Theorem 8.9, although a bit more technical. Invariant I5 must be modified as follows:

I5* With i_0 as in Invariant I4, if i even with $i - 2^{\text{val}_2\, i} \geqslant i_0$ or $i = n$, then $S^{(i)} = 0$ and $S^{(i+1)} = 0$.

The rest of the proof is as before except for some degree bounds: if i is even, then the polynomials $S^{(i)}G^{(i)}$ and $S^{(i)}G^{(i,\#)}$ have the same terms of degree at least

$\deg G^{(i)} - 4 \times 2^{\mathrm{val}_2\, i} + \deg S^{(i)}$ (notice the term $4 \times 2^{\mathrm{val}_2\, i}$ instead of $3 \times 2^{\mathrm{val}_2\, i}$ because the precision in the concise representation was increased). Assuming $S^{(i)} \neq 0$, that is $i - 2^{\mathrm{val}_2\, i} < i_0$, we have $\deg G^{(i)} - 4 \times 2^{\mathrm{val}_2\, i} + \deg S^{(i)} \leqslant d'$. Similarly if i is odd, then the polynomials $S^{(i)}G^{(i)}$ and $S^{(i)}G^{(i,\#)}$ have the same terms of degree at least $\deg G^{(i)} - 4 \times 2^{\mathrm{val}_2(i-1)} + \deg S^{(i)}$, which is again $\leqslant d'$.

As for the complexity, it is clear that for each k, there are at most 2 products $S^{(k)}G^{(k,\#)}$: one during the substitution at step k (line 30 in Algorithm 8.7) and possibly one at step $k - 2^{\mathrm{val}_2\, k}$ (line 27). The complexity of the rewriting steps therefore drops to $O\left(\mathsf{M}(nm)\log n\right)$, as announced. □

8.5.2 Improved complexity analysis using refined support bounds

The bound given in Theorem 8.14 assumes that the input polynomial P has a "triangular" support of degree d; typically the bound is tight if $\mathrm{lm}\, P = Y^d$. However, it may happen that the degree in the variable Y is much smaller than the total degree: this is in particular the case in sections 8.4.2 and 8.4.3.

Let us first notice that the degree in the variable Y during the execution of Algorithm 8.3 can be controlled using the following elementary properties:

Lemma 8.15. *We have that for all i, $\deg_Y G^{(i)} \leqslant n + i$.*

Corollary 8.16. *If the input polynomial P (in Algorithm 8.3) verifies $\deg_Y P \leqslant Kn$ for some $K \geqslant 3$, then for each i we have $\deg_Y(Q^{(i)}G^{(i)}) \leqslant Kn$.*

Proof. The dichotomic selection strategy imposes that $\deg_Y Q^{(i)} \leqslant n$ for $i > 0$, hence $\deg_Y(Q^{(i)}G^{(i)}) \leqslant 3n$. The result for $i = 0$ is obtained from

$$Q^{(0)}G^{(0)} = P - \sum_{i>0} Q^{(i)}G^{(i)} - R\,.$$ □

We next extend the discussion from section 6.4.1 to a new type of supports. Initially, we restricted ourselves to supports of the form

$$\mathcal{S}_{l,h} := \{M \in \mathcal{M} \;:\; l \leqslant \deg M \leqslant h\}\,,$$

for which $|\mathcal{S}_{l,h}| = \Theta(h(h-l))$. We now need to bound the degree in the variable Y independently of the total degree, so we consider supports of the form

$$\mathcal{S}_{l,h,s} := \{M \in \mathcal{M} \;:\; l \leqslant \deg M \leqslant h \text{ and } \deg_Y M \leqslant s\}\,.$$

Using the same change of variables as before with $X^aY^b \mapsto T^{h-a-b}U^b$, it is not hard to check that multiplication can still be done in time $O\left(\mathsf{M}(|\mathcal{S}_{l,h,s}|)\right)$ for such more general supports, and similarly for relaxed multiplication. This allows us to prove the following result:

Proposition 8.17. *If $K \geqslant 3$ is such that $\deg_Y(P) \leqslant Kn$, then the improved variant of Algorithm 8.3 (from Theorem 8.14) runs in time*

$$O\left(\mathsf{R}(Knd) + \mathsf{R}(nm)\log n\right)\,.$$

Proof. By Corollary 8.16, the supports appearing during Algorithm 8.3 are all of the form $\mathcal{S}_{l,h,s}$ with $s \leqslant Kn$, and we have $|\mathcal{S}_{l,h,s}| = \Theta(s(h-l))$, so that the cost of the relaxed multiplications is

$$\mathsf{R}(|Q^{(0)}G^{(0,\#)}|) + \cdots + \mathsf{R}(|Q^{(n)}G^{(n,\#)}|) = O\left(\mathsf{R}(Knd) + \mathsf{R}(nm)\log n\right) ,$$

and the cost of the rewriting steps is

$$O\left(\mathsf{M}(nm)\log n\right) = O\left(\mathsf{R}(nm)\log n\right)$$

as in Theorem 8.14. □

8.5.3 Consequences of the refined complexity bounds

We are now in a position to improve the results from section 8.4. Given a fixed ideal $\langle A, B\rangle$, we may precompute an augmented concise Gröbner basis $(G^{\#}, \bar{\mathbf{M}})$ once and for all; by Theorem 8.13, this precomputation takes time $O\left(\mathsf{R}(m^2) + \mathsf{R}(mn)\log n\right)$. Theorem 8.14 then leads to the following improved complexity bound for the ideal membership test from section 8.4.1.

Theorem 8.18. *Given $P \in \mathbb{K}[X,Y]$ with $\deg P \leqslant d$, we may test whether $P \in \langle A, B\rangle$ in time*

$$O\left(\mathsf{R}(d^2) + \mathsf{R}(mn)\log n\right) ,$$

using Algorithm 8.4, when regarding step 1 as a precomputation.

If $P, Q \in \mathbb{K}[X,Y]$ are in normal form with respect to G, then $\deg(PQ) \leqslant 2(n+m)$ and $\deg_Y(PQ) \leqslant 2n$. Proposition 8.17 then implies the following quasi-optimal complexity bound for multiplication in the quotient algebra $\mathbb{K}[X,Y]/\langle A, B\rangle$.

Theorem 8.19. *Using Algorithm 8.5, one multiplication in $\mathbb{K}[X,Y]/\langle A, B\rangle$ can be performed in time*

$$O\left(\mathsf{R}(mn)\log n\right) ,$$

when regarding step 1 as a precomputation.

Remark 8.6. The new bound is quasi-linear in the dimension mn of the quotient algebra. If $n \ll m$, then the new bound improves upon the one from Theorem 8.11 due to the term $\mathsf{R}(m^2)$. Notice that this term occurred for two reasons. First of all, the computation of the concise Gröbner basis in particular involves the computation of $B \operatorname{rem} A$; we now regard this as a precomputation. The reduction of PQ, which has degree $d = 2(m+n)$, also lead to a cost $O\left(\mathsf{R}(d^2)\right)$ in Theorem 8.9. Exploiting the reduced degrees in Y of P and Q, the cost of this reduction is reduced to $O\left(\mathsf{R}(mn)\log n\right)$ by Proposition 8.17.

Similarly, applying the improved bound from Proposition 8.17 to Algorithm 8.6, the reduction of $t_0 := Y^n = \operatorname{lm} G^{(0)}$ or $t_i := X^{m-n-1+2i}Y^{n-i} = \operatorname{lt} G^{(i)}$ with $i > 0$ takes time $O\left(\mathsf{R}(nm)\log n\right)$. This yields:

Theorem 8.20. *Using Algorithm 8.6, the reduced deglex Gröbner basis of* $\langle A, B\rangle$ *can be computed in time*

$$O\left(\mathsf{R}(m^2) + \mathsf{R}(nm)n\log n\right) .$$

Remark 8.7. This bound is quasi-linear in m^2+n^2m, which is indeed quasi-optimal if we take into account both the input and output sizes.

8.6 Experimental results

The above algorithms were implemented in the MATHEMAGIX software [HLM+02], whose source code can be downloaded from `svn://scm.gforge.inria.fr/svnroot/mmx/`. The implementation of the bivariate Gröbner basis and normal form algorithms is gathered in the package LARRIX. The main algorithms are found in the file `include/larrix/ggg.hpp` (the abbreviation ggg stands for "generic Gröbner grevlex"). We run the following experiment:

- pick random bivariate polynomials A, B of degree n with coefficients in the prime field $\mathbb{Z}/65521\mathbb{Z}$. This field is sufficiently large so that random elements satisfy the genericity assumptions with very high probability.
- compute a Gröbner basis G in concise representation with our algorithm and measure the time needed for this task. If the genericity hypothesis is not satisfied, this can be detected at this point and the computation fails (but it does not happen in practice).
- pick a random polynomial P of degree $2n$, compute its normal form with respect to G and measure the time needed for this task.

First, let us check that the implementation achieves the announced complexity from Table 8.1 (notice that $n = m$ in this experiment). To do so, we compare the time needed for the concise Gröbner basis with the theoretical bound $\mathsf{M}(n^2)\log n$, where $\mathsf{M}(n^2)$ is estimated simply by multiplying polynomials of the appropriate degree with the builtin MATHEMAGIX functions. Similarly, the cost of the normal form is compared with $\mathsf{R}(n^2)\log n$ (with R corresponding to the builtin series arithmetic). The results are given in Figure 8.4; we see that the implementation has the expected complexity, with a big-Oh constant around 4.

Now it is interesting to compare the absolute timings for this implementation and other reference software. So let us run the above experiment again, and compare with the equivalent functionalities in FGB [Fau10] (Gröbner basis) and SAGEMATH [Sag17] (Gröbner basis and normal form) in the same situation. Let us mention that SAGEMATH relies on SINGULAR [DGPS17] as a backend for multivariate polynomials. The files corresponding to this experiment are `ggg_bench.cpp` for MATHEMAGIX, `ggg_FGb_bench.cpp` for FGB, and `ggg_sage_bench.sage` for SAGE); all in the `bench/` directory.

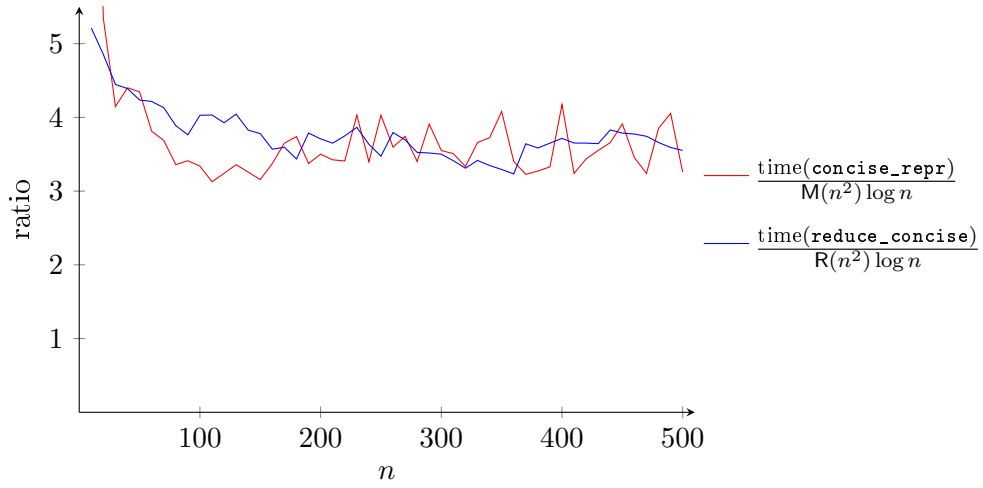


Figure 8.4: Measured execution time compared with the expected complexity bound.

All timings were measured on a platform equipped with an Intel(R) Core(TM) i7-6700 CPU at 3.40 GHz and 32 GB of 2133 MHz DDR4 memory. The benchmarks are done using the svn revision 10718 of Mathemagix, FGb/modp version 14538 and SageMath version 8.0 (which includes the version 4-1-0 of Singular); in each case the program uses a single thread.

The results are given in Figure 8.5 (a solid line represents the Gröbner basis computation, and a dashed line represents the reduction in normal form). We observe that the Mathemagix implementation becomes faster than the others for degrees as low as 20. For larger degrees, the speedup becomes really significant: for $n = 200$, the concise Gröbner basis is obtained in 188ms and the normal form in 1.4s with the new algorithms, while FGb and Sage need around 30s for each task. Notice however that FGb and Sage are designed to solve more general problems (including non generic systems in > 2 variables).

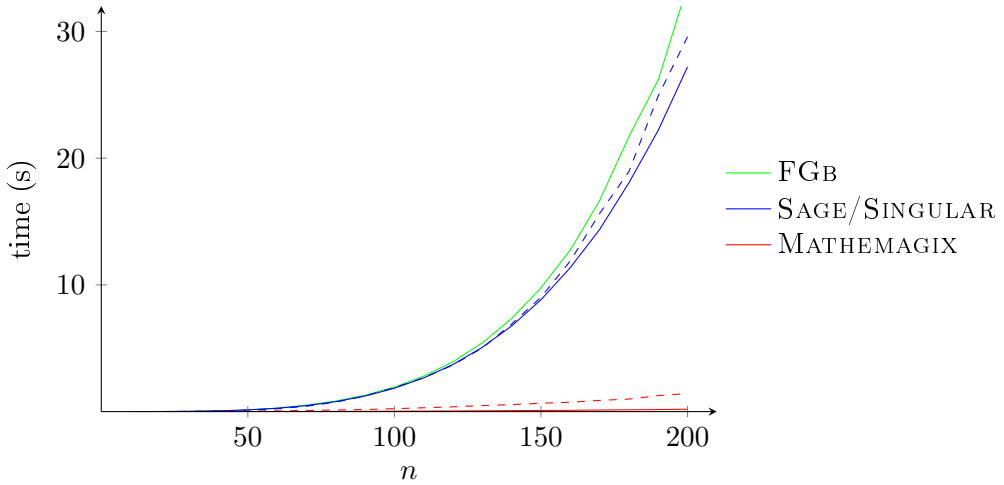


Figure 8.5: Comparison of the Mathemagix implementation with other software

Finally, to find out the most computationally expensive subtasks in each algorithm, we run the experiment for n in the order of a few thousands, and we measure the time needed for each part. (Let us mention that for $n = 1000$, FGb ran for 28 hours before running out of memory.) For the concise representation, we see that computing the truncated basis (matrix-vector products of bivariate polynomials) represent more than 99% of the time. This means that the univariate gcd and the computation of the matrices $\mathbf{M}^{(k,\ell)}$ require less than 1% of the time. For the normal form, evaluating the products $Q^{(i)}G^{(i)}$ using series arithmetic takes about 80-85% of the time, and the substitutions represents 14-18%.

degree n	1000	2000	4000
Truncated basis (s)	10.2	55.6	310
Total concise repr. (s)	10.3	56.0	312
Relaxed products (s)	74.4	447	2603
Substitutions (s)	16.3	83	422
Total normal form (s)	91.9	535	3046

Perspectives

As in the previous chapter, it would be desirable to relax some of the genericity assumptions.

One possibility would be to look at the univariate Euclidean algorithm while computing the concise Gröbner basis. Recall that the algorithm fails if one of the quotients is larger than 1. Generically this should not happen, but the probability of an accidental cancellation increases when working over a small finite field. It might be possible to adapt the algorithm if the number of accidental cancellations is small (say $O(\log n)$ in total), but this would be much more technical. Notice however that the algorithm cannot work if $\gcd(\operatorname{Diag} A, \operatorname{Diag} B) \neq 1$ or when $S(G^{(n-1)}, G^{(n)})$ does not reduce to zero, because the recurrence relations between the $G^{(i)}$ are lost after that. Also, it is not clear how to handle non-zero-dimensional ideals.

As another possibility, an intuitive result would be that the algorithms work modulo a generic linear change of variables, as long as the Bézout bound of n^2 solutions is reached. For example, the algorithm fails with $A := X^n$ and $B := Y^n$, but it works with $\tilde{A} := (aX + bY)^n$ and $\tilde{B} := (cX + dY)^n$. However, this conjecture seems out of reach for now.

Regarding an extension to more than 2 variables, this seems also out of reach at the moment. Some of the ideas still apply; for example the dichotomic selection strategy and the associated truncation extend quite naturally (for this reason, vanilla Gröbner bases as in Chapter 7 are likely to exist with $r > 2$ variables). Similarly, it is possible to construct a non-reduced Gröbner basis by regarding only the terms of highest degree, as a generalization of the dominant diagonal. However, the recurrence relations that appear are not small like in the bivariate case: ideally we would like $r \times r$ matrices, but already in 3 variables, recurrence relations with $\Theta(n)$ terms appear. To maintain small relations would require entirely new ideas.

About the applications of concise Gröbner bases, it is worth mentioning that van der Hoeven and Lecerf used them to build a quasi-quadratic algorithm for the bivariate resultant over finite fields [HL19c]. Recall that classical algorithms for this task are cubic, and the first known sub-cubic algorithm is due to Villard [Vil18] (with complexity $O(n^{(3-1/\Omega)(1+\epsilon)})$ for Ω the exponent of matrix multiplication). Roughly speaking, concise Gröbner bases provide a representation of $\mathbb{K}[X,Y]/\langle A,B\rangle$ in $\tilde{O}(n^2)$ operations. Then under additional assumptions, the resultant of A,B (with respect to Y) is the minimal polynomial of X in $\mathbb{K}[X,Y]/\langle A,B\rangle$, which can be found using fast modular composition. Notice that this algorithm is not practical, because it relies on Kedlaya-Umans modular composition [KU11] (hence Remark 4.2 applies again). Nevertheless, it could be interesting from a theoretical point of view to study the consequences in other areas, typically with geometric resolution and the Kronecker solver [GLS01], which heavily rely on the bivariate resultant.

Appendices

Appendix A

Computing finite field embeddings

Contents

As shown in Chapter 4 for the Frobenius FFT, there are situations where it is necessary to work simultaneously with several finite fields, or more precisely several extensions of the same finite field. This question also appears naturally when constructing an effective representation of $\overline{\mathbb{F}}_p$, the algebraic closure of $\mathbb{F}_p$. Indeed, recall that $\mathbb{F}_{p^d}$ contains the roots of irreducible polynomials of degree d. This means that for any element $x \in \overline{\mathbb{F}}_p$, there is an extension degree $d \in \mathbb{N}$ such that $x \in \mathbb{F}_{p^d}$; however it is not possible to fix a sufficiently large d because $\overline{\mathbb{F}}_p$ is infinite.

For both the Frobenius FFT or computation in $\overline{\mathbb{F}}_p$, we need at some point to perform operations with elements of different fields (for example, given $\alpha \in \mathbb{F}_{p^a}$ and $\beta \in \mathbb{F}_{p^b}$, compute $\alpha\beta$). This leads to the *finite field embedding problem*: given representations of $\mathbb{F}_{p^e}$ and $\mathbb{F}_{p^d}$ with $e \,|\, d$, find a function $\rho^{e\to d} : \mathbb{F}_{p^e} \to \mathbb{F}_{p^d}$ that represents algorithmically the injection $\mathbb{F}_{p^e} \hookrightarrow \mathbb{F}_{p^d}$.

Notice that the solution $\rho^{e\to d}$ is not unique: if $\mathbb{F}_{p^e}$ is represented by the quotient $\mathbb{F}_p[X]/\langle \mu^{(e)}(X)\rangle$ and $\zeta_1, \ldots, \zeta_e$ are the roots of $\mu^{(e)}$ in $\mathbb{F}_{p^d}$, then the map $X \mapsto \zeta_i$ for any i is a solution. To ensure the consistency of the system, it is necessary to add the constraint of *compatibility* [BCS97]:

$$\text{if } e \,|\, e' \,|\, d, \text{ then } \rho^{e\to d} = \rho^{e'\to d} \circ \rho^{e\to e'} . \tag{A.1}$$

This appendix presents a different solution to this problem, together with an experimental implementation.

A.1 The finite field embedding problem

The first classical algorithms to change representations of finite fields are due to Lenstra [Len91] and Allombert [All02]. These algorithms rather focus on the *finite*

$$
\begin{array}{ccccccc}
\vdots & & & & & & \\
\mathbb{F}_{p^9} & \subset & \mathbb{F}_{p^{18}} & \subset & \mathbb{F}_{p^{36}} & & \\
\cup & & \cup & & \cup & & \\
\mathbb{F}_{p^3} & \subset & \mathbb{F}_{p^6} & \subset & \mathbb{F}_{p^{12}} & & \\
\cup & & \cup & & \cup & & \\
\mathbb{F}_p & \subset & \mathbb{F}_{p^2} & \subset & \mathbb{F}_{p^4} & \cdots &
\end{array}
$$

Figure A.1: A lattice of finite field embeddings

field isomorphism problem, that is to convert between two representations of the same field $\mathbb{F}_{p^d}$. Such algorithms can be extended to compute embeddings, but they do not ensure the compatibility. For a more complete survey of solutions to the isomorphism problem, see [BDD+19, BDD+17] and references therein.

A.1.1 Manually enforcing compatibility

In the Bosma-Cannon-Steel framework [BCS97], the different fields are placed on a lattice as in Figure A.1, which is constructed iteratively when new extensions are added. To add a new field to the lattice, they start by computing one embedding (to or from one already present field), using typically Lenstra's or Allombert's algorithm. Then they compute all other relevant embeddings in a way that enforces the compatibility, using linear algebra techniques. Let us mention that this framework is used for the finite fields construction in Magma [BCP97] and, since recently, in Nemo [FHHJ17]. For example, if $\mathbb{F}_p$, $\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^3}$ are already present in the lattice, then adding $\mathbb{F}_{p^6}$ could be done as follows:

- compute $\rho^{2\to 6}$ using a classical algorithm,
- deduce $\rho^{1\to 6} := \rho^{2\to 6} \circ \rho^{1\to 2}$,
- compute $\rho^{3\to 6}$ such that $\rho^{1\to 6} := \rho^{3\to 6} \circ \rho^{1\to 3}$ using linear algebra.

With this method, constructing the representation of each field is efficient because $\mathbb{F}_{p^d}$ can be defined by any degree d irreducible polynomial. Also, evaluating an embedding (i.e. given $x \in \mathbb{F}_{p^e}$, compute $\rho^{e\to d}(x) \in \mathbb{F}_{p^d}$) boils down to a matrix-vector product which is very fast. However, adding new fields to the lattice becomes more and more expensive when the lattice grows, because there are more embeddings to compute each time; there is also a significant memory requirement because a quadratic number of embeddings needs to be stored (and each embedding $\rho^{e\to d}$ is represented by a $d \times e$ matrix).

A.1.2 Standard lattices

To reduce the memory requirement for storing the lattice of field embeddings, it is possible to represent the finite fields using a family of "standard" polynomials with special properties. With a well-chosen family of polynomials, there is a standard way to define embeddings which also enforces the compatibility. A classical example is the family of Conway polynomials [Par90, Sch92] $C^{(p,d)}$ such that

- the polynomial $C^{(p,d)} \in \mathbb{F}_p[X]$ is irreducible of degree d and monic;
- the roots of $C^{(p,d)}$ are primitive in $\mathbb{F}_{p^d}$ (each root α generates $\mathbb{F}_{p^d}^\times$ as a multiplicative group);
- for any e dividing d, $C^{(p,e)}$ is *norm-compatible* with $C^{(p,d)}$ in the following sense: if α is a root of $C^{(p,d)}$ and $r := (p^d-1)/(p^e-1)$, then α^r is a root of $C^{(p,e)}$;
- among all polynomials with the above properties, $C^{(p,d)}$ is lexicographically minimal (convention to ensure non-ambiguous definition).

The condition of *norm compatibility* echoes the fact that if α is primitive in $\mathbb{F}_{p^d}$, then α^r is primitive in $\mathbb{F}_{p^e}$ (again with $r := (p^d-1)/(p^e-1)$). With this definition, the canonical embedding for $\mathbb{F}_{p^e} \subset \mathbb{F}_{p^d}$ is given by

$$\begin{aligned} \rho^{e\to d}: \quad \mathbb{F}_p[X]/\langle C^{(p,e)}(X)\rangle &\to \mathbb{F}_p[Y]/\langle C^{(p,d)}(Y)\rangle \\ X \bmod C^{(p,e)}(X) &\mapsto Y^r \bmod C^{(p,d)}(Y) \end{aligned}$$

and it is clearly compatible in the sense of equation (A.1).

With such standard polynomials, the required storage is linear in the size of the lattice. To evaluate the embedding $\rho^{e\to d}$, it suffices to first compute (by modular exponentiation) $\rho^{e\to d}(X) := Y^r \operatorname{rem} C^{(p,d)}(Y)$, then $\rho^{e\to d}(P(X)) = P(\rho^{e\to d}(X))$ is obtained by modular composition. However, the computation of Conway polynomials is very expensive: the algorithm from [HL04] is only slightly better than exhaustive search. To make this method practical up to a certain point, some computer algebra systems embark pre-computed tables of Conway polynomials.

A variant proposed by De Feo, Randriam and Rousseau [DRR19] defines another family of standard polynomials. Their construction combines the Lenstra-Allombert algorithm [Len91, All02] with Conway polynomials to get extensions of higher degrees. More precisely, if the Conway polynomials are known up to degree n, then they derive another family of polynomials for all degrees dividing $p^n - 1$. However, they point out that getting all standard polynomials up to degree N with their construction is asymptotically equivalent to the computation of the first N Conway polynomials.

A.1.3 Combination of ℓ-adic towers

On the lattice of field extensions from Figure A.1, we observe two main sequences of embeddings: $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^4} \subset \cdots$ horizontally, and $\mathbb{F}_p \subset \mathbb{F}_{p^3} \subset \mathbb{F}_{p^9} \subset \cdots$

vertically. These sequences are then combined to form the other extensions in the quarter-plane. More generally, the lattice of all field extensions that constitute $\overline{\mathbb{F}}_p$ can be seen as a combination of the *ℓ-adic towers*

$$\mathbb{F}_p \subset \mathbb{F}_{p^\ell} \subset \mathbb{F}_{p^{\ell^2}} \subset \mathbb{F}_{p^{\ell^3}} \subset \cdots$$

for each prime ℓ. Such towers have the advantage that constructing compatible embeddings is easier: in most cases, subquadratic or even quasi-linear algorithms exist. The case $\ell = p$ (Artin-Schreier towers) has been studied in [Can89, Cou00, DS12]; the case $\ell = 2$ is treated in [DS15]; and solutions for the other cases are found in [DDS13]. To construct arbitrary extensions, it remains to combine the different towers.

The first solution is to notice that if Q is irreducible of degree n over $\mathbb{F}_p$ and if m, n are coprime, then Q is also irreducible over $\mathbb{F}_{p^m}$. If $d = \prod_{\ell \text{ prime}} \ell^{e(\ell)}$, then it is easy to construct successively the extensions

$$\mathbb{F}_p \subset \mathbb{F}_{p^{2^{e(2)}}} \subset \mathbb{F}_{p^{2^{e(2)}3^{e(3)}}} \subset \mathbb{F}_{p^{2^{e(2)}3^{e(3)}5^{e(5)}}} \subset \cdots \subset \mathbb{F}_{p^d}$$

using a different variable each time. For example, let $Q^{(i)} \in \mathbb{F}_p[X]$ be irreducible polynomials of degree i for $i \in \{2, 3, 5\}$. Then the extension $\mathbb{F}_{p^{30}}$ can be constructed using the isomorphisms

$$\begin{aligned}\mathbb{F}_{p^{30}} &\cong \mathbb{F}_{p^{15}}[X_2]/\langle Q^{(2)}(X_2)\rangle \\ &\cong \mathbb{F}_{p^5}[X_2, X_3]/\langle Q^{(2)}(X_2), Q^{(3)}(X_3)\rangle \\ &\cong \mathbb{F}_p[X_2, X_3, X_5]/\langle Q^{(2)}(X_2), Q^{(3)}(X_3), Q^{(5)}(X_5)\rangle\,.\end{aligned}$$

Similarly, if $Q^{(4)}$ is irreducible of degree 4, then the extension $\mathbb{F}_{p^{60}}$ can be represented as

$$\mathbb{F}_{p^{60}} \cong \mathbb{F}_p[X_2, X_3, X_5]/\langle Q^{(4)}(X_2), Q^{(3)}(X_3), Q^{(5)}(X_5)\rangle\,.$$

Assume that we know an embedding of $\mathbb{F}_{p^2} \cong \mathbb{F}_p[X]/\langle Q^{(2)}(X)\rangle$ into the extension $\mathbb{F}_{p^4} \cong \mathbb{F}_p[X]/\langle Q^{(4)}(X)\rangle$. Then this embedding can be extended into an embedding of $\mathbb{F}_{p^{30}}$ into $\mathbb{F}_{p^{60}}$, simply by applying the same transformation to the variable X_2 in the above multivariate representations. It is also clear that this method preserves the compatibility of combined embeddings (as long as the embeddings inside each ℓ-adic tower were compatible).

The second solution is to combine the towers using the composed product (if A has roots α_i and B has roots β_j, then the composed product of A, B is the polynomial with roots $\alpha_i\beta_j$). For example, with $Q^{(2)}, Q^{(3)}, Q^{(5)}$ as above, we let $Q^{(30)}$ be their composed product, and we use the representation $\mathbb{F}_p[X]/\langle Q^{(30)}(X)\rangle$ for $\mathbb{F}_{p^{30}}$. De Feo, Doliskani and Schost [DDS14] gave efficient algorithms to translate between multivariate representations like

$$\mathbb{F}_p[X_2, X_3, X_5]/\langle Q^{(2)}(X_2), Q^{(3)}(X_3), Q^{(5)}(X_5)\rangle$$

and univariate representations like

$$\mathbb{F}_p[X]/\langle Q^{(30)}(X)\rangle\,.$$

The idea of this construction is to use a univariate representation for all extensions; but to compute an embedding, the representation is first decomposed to a multivariate representation, then the embedding is performed as above and the result is finally recomposed in univariate representation. The overhead of back-and-forth translations is compensated by the improved arithmetic when using a univariate representation of $\mathbb{F}_{p^d}$ (multiplication of dense polynomials with many variables quickly becomes impractical). A combination of towers using this technique is referred to as a *compositum*.

A.2 Proposed solution

The construction proposed in this appendix is based on a combination of ℓ-adic towers as in section A.1.3. To combine the towers, we will use a multivariate representation, but we merge some variables using the composed product. Roughly speaking, this is a middlepoint between the two approaches of section A.1.3. It turns out that a compromise between these ideas leads to practical improvements.

A.2.1 General framework

The idea of merging some (but not all) variables into one is inspired by [HL19b]. In their paper, van der Hoeven and Lecerf consider successive ring extensions $\mathbb{A}_0 \subset \mathbb{A}_1 \subset \mathbb{A}_2 \subset \cdots$ where $\mathbb{A}_{i+1} := \mathbb{A}_i[X_i]/\langle P^{(i)}(X_i)\rangle$ with $P^{(i)} \in \mathbb{A}_i[X_i]$. In other words, $\mathbb{A}_{i+1}$ is represented by polynomials in $X_0, \ldots, X_i$ modulo a triangular system. This representation gives trivial embeddings $\mathbb{A}_i \hookrightarrow \mathbb{A}_j$ for $i \leqslant j$, but on the other hand the arithmetic of $\mathbb{A}_i$ has an overhead that is exponential in i: with the current best algorithm [Leb15], multiplication in $\mathbb{A}_i$ has complexity $O(3^i\,\mathsf{M}([\mathbb{A}_i : \mathbb{A}_0]))$. To reduce the overhead while keeping efficient embeddings, van der Hoeven and Lecerf construct an *accelerated tower*

$$\tilde{\mathbb{A}}_0 \subset \tilde{\mathbb{A}}_{i_1} \subset \tilde{\mathbb{A}}_{i_2} \subset \cdots \quad \text{with} \quad \tilde{\mathbb{A}}_{i_k} \cong \mathbb{A}_{i_k} \quad \text{and} \quad \tilde{\mathbb{A}}_{i_{k+1}} = \tilde{\mathbb{A}}_{i_k}[X_k]/\langle \tilde{P}^{(k)}(X_k)\rangle$$

by collapsing certain levels of the original towers. This allows us to reduce the number of variables and it remains only to compute the embedding $\mathbb{A}_j \hookrightarrow \tilde{\mathbb{A}}_{i_{k+1}}$ whenever $i_k < j < i_{k+1}$, which is not too expensive as long as $[\mathbb{A}_{i_{k+1}} : \mathbb{A}_{i_k}]$ is not too large. In practice, they choose a parameter δ and they set

$$i_{k+1} := \min(j \text{ s.t. } j > i_k \text{ and } [\mathbb{A}_j : \mathbb{A}_{i_k}] \geqslant \delta)\,.$$

For computations in $\overline{\mathbb{F}}_p$, the setting is a bit different because the extensions form a lattice instead of a tower. In particular, there is no fixed sequence of intermediate fields between $\mathbb{F}_p$ and $\mathbb{F}_{p^d}$: $\mathbb{F}_{p^{12}}$ could be represented using any of the following

$$\begin{gathered}
\mathbb{F}_p \hookrightarrow \mathbb{F}_{p^4} \hookrightarrow \mathbb{F}_{p^{12}} \\
\mathbb{F}_p \hookrightarrow \mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^6} \hookrightarrow \mathbb{F}_{p^{12}} \\
\mathbb{F}_p \hookrightarrow \mathbb{F}_{p^3} \hookrightarrow \mathbb{F}_{p^{12}} \\
\vdots
\end{gathered}$$

and we wish to keep a uniform representation. For this reason, we prefer to start from the representation where we assign the variable X_ℓ to the ℓ-adic tower for each prime ℓ; then we may choose to combine two or more variables into one using the composed product (as in [DDS14]), and keep the other separated. For example, if we combine the 2-adic and 3-adic towers but not the 5-adic tower, $\mathbb{F}_{p^{30}}$ would be represented as

$$\mathbb{F}_p[X_{2,3}, X_5]/\langle Q^{(6)}(X_{2,3}), Q^{(5)}(X_5)\rangle$$

(where $Q^{(6)}$ is the composed product of $Q^{(2)}$ and $Q^{(3)}$), instead of

$$\mathbb{F}_p[X_2, X_3, X_5]/\langle Q^{(2)}(X_2), Q^{(3)}(X_3), Q^{(5)}(X_5)\rangle \quad \text{or} \quad \mathbb{F}_p[X]/\langle Q^{(30)}(X)\rangle$$

in the classical approaches. Of course, the towers to be combined do not necessarily correspond to consecutive primes. For example, if we consider the ℓ-adic towers for $\ell \in \{2, 3, 5, 7\}$, it could be appropriate to split this into $\{2, 7\}$ and $\{3, 5\}$. Moreover, the construction is incremental: when new towers are added, they can freely be assigned to an already existing compositum, or kept separated by creating a new variable. For example, if we add the 11-adic tower, we can insert it to the $\{2, 7\}$ compositum (the variable $X_{2,7}$ would be renamed $X_{2,7,11}$), or to the $\{3, 5\}$ compositum (thus renaming $X_{3,5}$ into $X_{3,5,11}$), or keep it separated (with a new variable X_{11}). Except for the renamed variable in the first two cases (which is computationally free), none of these choices change the representation of already constructed fields. However, moving towers from one compositum to another would change the representation so we will avoid this.

Representations like $\mathbb{F}_p[X_{2,3}, X_5]/\langle Q^{(6)}(X_{2,3}), Q^{(5)}(X_5)\rangle$ can still be seen as polynomials modulo a triangular system, but notice that each polynomial in the triangular system has only one variable. Because of this, the bound for modular multiplication can be refined. Let us say that the prime factors of d are split into k composita $\mathcal{C}_1, \ldots, \mathcal{C}_k$; so that $\mathbb{F}_{p^d}$ is represented by polynomials in k variables:

$$\mathbb{F}_{p^d} \cong \mathbb{F}_p[X_{\mathcal{C}_1}, \ldots, X_{\mathcal{C}_k}]/\langle Q^{(\mathcal{C}_1)}(X_{\mathcal{C}_1}), \ldots, Q^{(\mathcal{C}_k)}(X_{\mathcal{C}_k})\rangle\,.$$

Lebreton's algorithm [Leb15] would then give a bound of $O(3^k\mathsf{M}(d))$ operations for multiplication in $\mathbb{F}_{p^d}$. Recall that this algorithm was an improvement of [LMS09], that has $O(4^k\mathsf{M}(d))$ complexity. Roughly speaking, these algorithms consist of three steps:

- a recursive call to reduce with respect to $Q^{(\mathcal{C}_1)}, \ldots, Q^{(\mathcal{C}_{k-1})}$,
- a reduction with respect to $Q^{(\mathcal{C}_k)}$,
- another recursive call with respect to $Q^{(\mathcal{C}_1)}, \ldots, Q^{(\mathcal{C}_{k-1})}$, because the degree in $X_{\mathcal{C}_1}, \ldots, X_{\mathcal{C}_{k-1}}$ increased during the second step.

However, since $Q^{(\mathcal{C}_k)}$ is in fact univariate in $X_{(\mathcal{C}_k)}$, the degrees in the other variables do not increase during the second step, so the third step is not required and there is only one recursive call. Then it is easy to check that the complexity of the algorithms from [Leb15, LMS09] is in fact $O(2^k\mathsf{M}(d))$ in this case.

Remark A.1. In [LMS09], Li, Moreno Maza and Schost also give a variant for this case with a system of univariate polynomials. They show that multiplication takes at most $K_\epsilon d^\epsilon \mathsf{M}(d)$ operations for any $\epsilon > 0$ (for some constant K_ϵ). In principle, multiplication in $\mathbb{F}_{p^d}$ could be made essentially quasi-linear. The idea is to handle differently moduli with degree $\leqslant \delta_\epsilon$ and moduli with degree $> \delta_\epsilon$, for some bound δ_ϵ. They present an implementation where all moduli have degree 2, and the exponential dependency in the number of variables is indeed eliminated. However, the algorithm seems impractical for our problem, because moduli may have large degrees.

A.2.2 Experimental results

An experimental C++ implementation of this framework (in collaboration with Luca De Feo) shows that the construction is practical. The source code is available at `https://gitlab.inria.fr/rlarrieu/tower`. For now, ℓ-adic towers are only implemented for the Kummer case (ℓ divides $p-1$). For this reason, we run experiments with $p := 2311 = 2 \times 3 \times 5 \times 7 \times 11 + 1$ so we can construct ℓ-adic towers for $\ell \in \{2, 3, 5, 7, 11\}$. Polynomial arithmetic over $\mathbb{F}_p$ uses the NTL [Sho01] library. The main algorithms are in the `include/` directory:

- `irred_tower.h` and `compositum.h` define the representation of ℓ-adic towers and composita using univariate arithmetic.
- `multivariate.h` and `triangular*.h` define general purpose arithmetic for multivariate polynomials modulo triangular sets.
- `tensor_product.h` defines the tensor product (in multivariate representation) of several composita.

Additional test and benchmark programs are found in the `test` and `bench` directories.

Timings are measured on a platform equipped with an Intel(R) Core(TM) i7-6700 CPU at 3.40 GHz and 32 GB of 2133 MHz DDR4 memory. The platform runs the Stretch GNU Debian operating system with a 64 bit Linux kernel version 4.3. We compile with GCC [GCC87] version 6.3. We use NTL version 11.3.2 with the default compilation options, and the thresholds were tuned (by the configure script) during the installation. The execution times are given for the version from August 23, 2019 (commit f0d3c213). Unless specified otherwise, the programs use a single thread.

Let us first compare the construction from section A.2.1 with the standard lattices from [DRR19]. We measure the time needed to construct $\mathbb{F}_{p^d}$ and to compute the embedding $\mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^d}$. For the multivariate representation, we try different partitions of the towers among the variables (file `bench/ffembed.cpp`):

- 1 variable – compositum $\{2, 3, 5, 7, 11\}$,
- 3 variables – composita $\{2, 5\}$, $\{3, 7\}$ and $\{11\}$,
- 5 variables – each tower in a separate compositum.

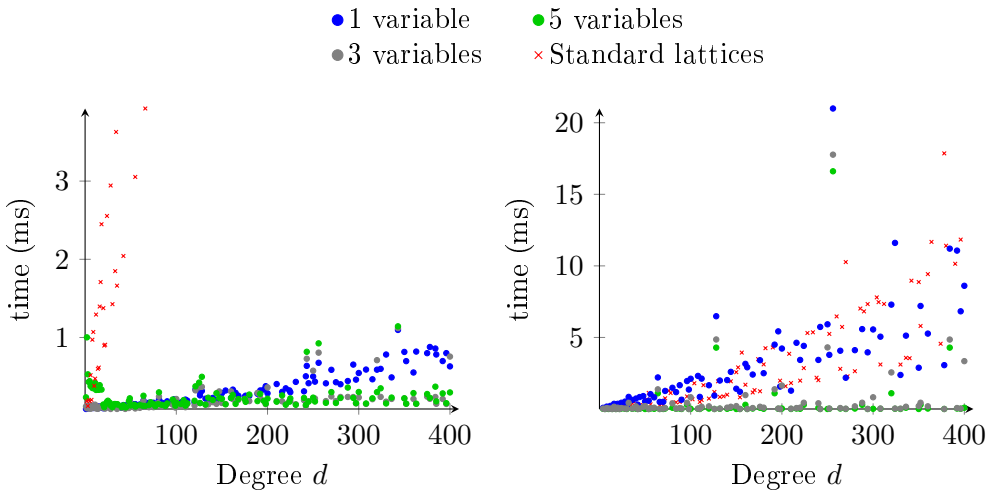


Figure A.2: Construction of $\mathbb{F}_{p^d}$ (left) and embedding $\mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^d}$ (right).

For the standard lattices (file `bench/lattice_GF_H90.jl`), we use the implementation from https://github.com/erou/LatticeGFH90.jl (version from May 21, 2019 – commit d52a2fb), using Julia [BEKS17] version 1.1.1 with Nemo [FHHJ17] version 0.14.1. The results are presented in Figure A.2.

The first observation is that the construction from section A.2.1 is much faster than standard lattices to obtain the representation of $\mathbb{F}_{p^d}$. Also, constructions with 3 and 5 variables are faster than the construction with 1 variable because the required composita are smaller. Notice that in this example d remains relatively small so the primes 2, 3, 5, 7, 11 do not appear simultaneously, which explains why the construction with 3 and 5 variables have similar performances for most values of d. For larger d, the construction with 3 variables would tend to become slower more often because larger composita will appear.

As a second observation, for the computation of embeddings, it is roughly equivalent to use standard lattices or to combine all towers in a univariate representation with composita. Using 3 or 5 variables, most embeddings come essentially for free because injections like

$$\mathbb{F}_p[X_2]/\langle Q^{(2)}(X_2)\rangle \hookrightarrow \mathbb{F}_p[X_2, X_3]/\langle Q^{(2)}(X_2), Q^{(3)}(X_3)\rangle$$

are trivial. In 5 variables, only the 2-adic valuation of d plays a role in the execution time. In 3 variables, the towers for $\ell \in \{2, 5\}$ are combined so that the 5-adic valuation of d is also important (but the ℓ-adic valuation for $\ell \in \{3, 7, 11\}$ is irrelevant). This explains in particular why the cost of $\mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^{128}}$ is much higher than the others. Notice that the choice of $\mathbb{F}_{p^2}$ as input field is especially favorable; with $\mathbb{F}_{p^6}$ as input field, the cost would also depend on the 3-adic valuation (and the 7-adic in the 3 variable construction), so more embeddings would be costly.

Remark A.2. In the implementation of [DRR19], embeddings are first computed as Julia functions, then they are evaluated by applying said function. The second

step is actually faster than the first one. If many elements are to be embedded in $\mathbb{F}_{p^e} \hookrightarrow \mathbb{F}_{p^d}$ for the same e, d, then the embedding function can be computed just once and evaluated many times which leads to a significant speedup.

Remark A.3. The comparison with standard lattices is only about general behavior, because they cannot construct the same fields. Indeed, with standard lattices, d must divide p^n-1 for relatively small n (because not many Conway polynomials have been computed over $\mathbb{F}_{2311}$). For example, $\mathbb{F}_{p^{64}}$ and $\mathbb{F}_{p^{128}}$ are not supported by the implementation of [DRR19]. We mention however that the embedding $\mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^{32}}$ is a bit faster with standard lattices.

Now we run a second experiment to compare the cost of the embedding and the cost of polynomial arithmetic (file `bench/ffembed-mult.cpp`). Assume that we wish to multiply $\alpha \in \mathbb{F}_{p^a}$ and $\beta \in \mathbb{F}_{p^b}$ (the result is in $\mathbb{F}_{p^d}$ with $d := \mathrm{lcm}(a, b)$). To do so, we need to

- compute the representations of $\mathbb{F}_{p^a}, \mathbb{F}_{p^b}, \mathbb{F}_{p^d}$,
- compute the embeddings $(\mathbb{F}_{p^a} \hookrightarrow \mathbb{F}_{p^d})$ for α and $(\mathbb{F}_{p^b} \hookrightarrow \mathbb{F}_{p^d})$ for β,
- compute a multiplication in $\mathbb{F}_{p^d}$.

So let us measure the time needed for each task; again we try different partitions of the towers among the variables. Table A.1 gives the results for $a := 2^2 \times 5^2 \times 11$ and $b := 3^2 \times 7^2$. Table A.1 gives the results for $a := 2^2 \times 3 \times 5^2 \times 7 \times 11$ and $b := 2 \times 3^2 \times 5 \times 7^2 \times 11$. In both cases, we get $d = 2^2 \times 3^2 \times 5^2 \times 7^2 \times 11$.

In the first situation, we see that the partition with 5 variables and the partition $\{2, 5\}$, $\{3, 7\}$, $\{11\}$ lead to almost instantaneous embeddings, for the reasons already mentioned in the first experiment. Surprisingly, the multiplication in this case is even faster than in the univariate case. To see why, notice that in the 5 variable case, $\alpha \in \mathbb{F}_p[X_2, X_5, X_{11}]$ while $\beta \in \mathbb{F}_p[X_3, X_7]$, then the degree of $\alpha\beta$ in the variable X_ℓ remains less than the degree of $Q^{(\ell)}$, so that the reduction modulo the triangular set may be skipped (the reason is similar for the partition $\{2, 5\}$, $\{3, 7\}$, $\{11\}$). For the partition $\{2, 3, 5\}$, $\{7\}$, $\{11\}$, there is some work to do to compute embeddings, but this remains considerably faster than with the univariate representation. Also the time for the multiplication is essentially the same; a possible explanation is that the reduction boils down to several univariate remainder operations of smaller degree instead of a single larger one, which may compensate the overhead of Kronecker substitution.

This experiment suggests that it is better to keep the towers separated, but the fact that a, b have different prime factors is very particular. If a, b have the same prime factors with different powers, then it forces nontrivial embeddings in all towers, and larger degrees in each variable for the product (which implies a reduction modulo the triangular system). In this situation, we expect to observe a compromise between efficient embeddings and fast multivariate arithmetic.

This is the situation that we consider in the benchmark of Table A.2. Here we see clearly the influence of the number of variables on the efficiency of modular

Table A.1: Multiplication of $\alpha \in \mathbb{F}_{p^a}$ and $\beta \in \mathbb{F}_{p^b}$, for $a := 2^2 \times 5^2 \times 11$ and $b := 3^2 \times 7^2$. All timings are in milliseconds.

	5 variables	3 variables		1 variable
Partition	$\{2\},\{3\},\{5\},$ $\{7\},\{11\}$	$\{2,3,5\},$ $\{7\},\{11\}$	$\{2,5\},$ $\{3,7\},$ $\{11\}$	$\{2,3,5,7,11\}$
Construction $\mathbb{F}_{p^a}$	0.38	0.37	0.35	2.22
Construction $\mathbb{F}_{p^b}$	0.19	0.19	0.84	0.78
Construction $\mathbb{F}_{p^d}$	116	112	111	906
$\mathbb{F}_{p^a} \hookrightarrow \mathbb{F}_{p^d}$	1.81	27.1	1.27	6888
$\mathbb{F}_{p^b} \hookrightarrow \mathbb{F}_{p^d}$	1.01	151	1.13	5088
Total embedding	2.82	178	2.40	11977
Multiplication	85.7	108	41.3	107

Table A.2: Multiplication of $\alpha \in \mathbb{F}_{p^a}$ and $\beta \in \mathbb{F}_{p^b}$, for $a := 2^2 \times 3 \times 5^2 \times 7 \times 11$ and $b := 2 \times 3^2 \times 5 \times 7^2 \times 11$. All timings are in milliseconds.

	5 variables	3 variables		1 variable
Partition	$\{2\},\{3\},\{5\},$ $\{7\},\{11\}$	$\{2,3,5\},$ $\{7\},\{11\}$	$\{2,5\},$ $\{3,7\},$ $\{11\}$	$\{2,3,5,7,11\}$
Construction $\mathbb{F}_{p^a}$	4.95	5.36	4.96	41.7
Construction $\mathbb{F}_{p^b}$	10.1	10.3	10.8	108
Construction $\mathbb{F}_{p^d}$	108	105	104	891
$\mathbb{F}_{p^a} \hookrightarrow \mathbb{F}_{p^d}$	4231	4211	4895	6983
$\mathbb{F}_{p^b} \hookrightarrow \mathbb{F}_{p^d}$	2445	1730	2612	3964
Total embedding	6678	5941	7507	10948
Multiplication	2303	587	592	110

multiplication: it is about 4 times more efficient with 3 variables than with 5, and about 5 times more efficient with 1 variable than with 3. This corresponds more or less to the theoretical overhead by a factor 2^k mentioned earlier, where k is the number of variables (the experiments in `bench/triangular_univariate.cpp` tend to confirm this). On the contrary, the cost of embeddings decreases as the number of variables increases: embeddings are about 15% faster for the partition $\{2\},\{3\},\{5\},\{7\},\{11\}$ than for the partition $\{2,5\}$, $\{3,7\}$, $\{11\}$, and 40% faster than in the univariate case. The partition $\{2,3,5\}$, $\{7\}$, $\{11\}$ gives even faster embeddings (by about 30% for $\mathbb{F}_{p^b} \hookrightarrow \mathbb{F}_{p^d}$), which is rather surprising and hard to explain. With a more detailed profiling, we see that with 5 variables, the embedding $\mathbb{F}_{p^b} \hookrightarrow \mathbb{F}_{p^d}$ consist of around 20 000 small conversions $\mathbb{F}_{p^2} \hookrightarrow \mathbb{F}_{p^4}$ and about as many conversions $\mathbb{F}_{p^5} \hookrightarrow \mathbb{F}_{p^{25}}$. The computation required for each task is small but the complete operation may be slowed down by the frequent memory access (in 3 variables, there are 539 conversions $\mathbb{F}_{p^{90}} \hookrightarrow \mathbb{F}_{p^{900}}$ instead, so this phenomenon may be less present). Finally, notice that in 1 and 3 variables, the cost of the modular multiplication is negligible compared to the cost of the embeddings; in 5 variables, the costs have the same order of magnitude (multiplication represents about 25% of the overall operation). In this experiment, it is then optimal to combine the towers using a mix of compositum techniques and multivariate arithmetic, and to distribute the 5 towers among 3 variables.

Remark A.4. Another advantage of the multivariate representation is that some operations naturally decompose into several independent smaller subtasks; then they can be parallelized. For example, using 8 threads, the operation of reduction modulo the triangular system becomes two to three times faster, which leads to an improvement of up to 40% for the multiplication in $\mathbb{F}_{p^d}$. For now, embeddings are not parallelized because one of the subroutines is currently not thread-safe, but in principle this could be similarly parallelized and we could expect a speed-up by a factor 2 or 3 as well.

Perspectives

These first experimental results are encouraging because the partial implementation is already competitive. The main future implementation challenge is to handle all cases of ℓ-adic towers (recall that only the case where ℓ divides $p-1$ is supported by the current implementation). Also, the code could probably be further optimized (there may remain some unnecessary copies and other similar problems), and the computation of embeddings could be parallelized as stated previously.

We observed that a mix of multivariate arithmetic and composita techniques can lead to a practical improvement; the next step is to analyze this behavior more precisely from a theoretical point of view to refine the thresholds. For the ultimate objective of computing in the algebraic closure $\overline{\mathbb{F}}_p$, it remains also to define a strategy to partition automatically the towers between the variables. For now, this last point is an open question but a theoretical analysis of the thresholds as mentioned might give some leads.

Annexe B

Résumé des travaux

Contents

L'objectif de cette thèse est d'étudier des algorithmes efficaces pour réaliser certaines opérations mathématiques ; plus précisément, on s'intéresse à l'arithmétique des polynômes à coefficients dans un corps fini, notamment la multiplication. Les anneaux de polynômes sur des corps finis sont fréquemment utilisés pour construire des systèmes de chiffrement ou des codes correcteurs d'erreurs. Pour qu'un système de chiffrement ou un code correcteur soit utilisable en pratique, il est crucial que les opérations sous-jacentes puissent être réalisées rapidement, y compris pour des entrées de grande taille. Ceci motive la recherche de nouveaux algorithmes (ou l'amélioration d'algorithmes existants) pour effectuer les opérations sur les polynômes plus efficacement. À noter que de nombreuses opérations de plus haut niveau peuvent se ramener à une multiplication de polynômes, d'où l'intérêt particulier pour ce problème.

B.1 Présentation générale

La multiplication de polynômes ressemble beaucoup au problème plus familier sur la multiplication des nombres entiers. Ces deux types ont en effet de nombreuses propriétés communes, car un nombre entier (écrit par exemple en base 10) peut être vu comme un polynôme sur $\mathbb{Z}$ évalué en 10 : $123 = 1 \times 10^2 + 2 \times 10 + 3$. Il n'est donc pas surprenant que les résultats de complexité successifs soient similaires pour les deux problèmes. Historiquement, un algorithme pour les entiers a souvent été suivi quelques années plus tard par une version équivalente pour les polynômes. Le décalage s'explique par quelques différences subtiles entre les deux contextes. D'un côté, les entiers posent une difficulté supplémentaire liée à la retenue ; de l'autre, les opérations élémentaires sur les coefficients peuvent être plus complexes dans le cas des polynômes (par exemple, la multiplication peut ne pas être commutative).

L'algorithme naïf, enseigné à l'école primaire, permet de multiplier des entiers de n chiffres en $O(n^2)$ opérations. Il est clair que cet algorithme est encore valable pour multiplier des polynômes à coefficients dans n'importe quel anneau. Jusqu'au milieu du XX$^{\text{e}}$ siècle, on a cru qu'il était impossible de faire mieux, puis Karatsuba a trouvé une méthode plus rapide [KO63], basée sur l'identité suivante :

$$(ax+b)(cx+d) = ux^2 + (w-u-v)x + v \quad \text{avec} \quad \begin{cases} u := a \times c \\ v := b \times d \\ w := (a+b) \times (c+d) \end{cases}.$$

Ici, il n'y a que 3 produits au lieu de 4 habituellement, donc doubler la taille de l'entrée multiplie le nombre d'opérations requises par 3 ; il faut donc $O(n^{\log_2 3})$ opérations pour multiplier deux entiers de n chiffres. Là encore, cet algorithme s'applique aussi aux polynômes pour des raisons évidentes.

Grâce à la découverte de la *transformée de Fourier rapide* (ou FFT pour *Fast Fourier Transform*) [CT65], les méthodes d'évaluation-interpolation permettent de multiplier en temps quasi-linéaire. Il est notamment possible de multiplier des polynômes de degré n à coefficients complexes en $O(n \log n)$ opérations sur $\mathbb{C}$. Le fait de compter les opérations sur $\mathbb{C}$ cache cependant que la précision requise au cours du calcul augmente avec n ; ce modèle n'est donc pas adapté pour évaluer la complexité de la multiplication entière. De plus, l'algorithme utilise l'existence de racines n-ièmes de l'unité ($\mathrm{e}^{2i\pi/n}$) pour tout n ; cette borne n'est donc valide que dans certains anneaux spécifiques. Plus tard, Schönhage et Strassen ont adapté cette méthode et donné un algorithme légèrement plus cher en $O(n \log n \log\log n)$ pour la multiplication d'entiers [SS71]. Cet algorithme est également valide pour les polynômes sur un corps commutatif (sauf en caractéristique 2, mais une variante spécifique [Sch77] résout le problème). Cantor et Kaltofen ont ensuite généralisé ce résultat pour des anneaux quelconques [CK91], y compris dans le cas non commutatif et/ou non associatif.

Cette borne est la meilleure connue à ce jour pour les polynômes dans le cas général. Cependant, des algorithmes encore plus rapides ont été trouvés pour les entiers, à commencer par l'algorithme de Fürer [Für09] avec une complexité de $O(n \log n\, K^{\log^* n})$, où K est une constante non spécifiée et $\log^*$ désigne le logarithme itéré (le nombre k minimal tel que si log est composé k fois, on a $\log\log\cdots\log n < 1$). Les techniques utilisées ici ne s'appliquent plus pour les polynômes, mais il est possible de les adapter dans le cas de coefficients dans un corps fini. Harvey, van der Hoeven et Lecerf ont ainsi prouvé une borne du même type dans ce cas ; en fait ils ont établi la borne $O(n \log n\, 8^{\log^* n})$ pour les entiers et les polynômes sur un corps fini [HHL16b, HHL17], explicitant ainsi la constante K dans la borne de Fürer.

C'est ce résultat qui a servi de point de départ pour la thèse, l'objectif initial étant d'en étudier les conséquences théoriques et pratiques. En particulier, dans un autre article [HHL16a], Harvey, van der Hoeven et Lecerf adaptent les nouvelles idées pour multiplier plus efficacement les polynômes sur $\mathbb{F}_2$, en utilisant les propriétés de l'extension $\mathbb{F}_{2^{60}}$. Cette application pratique a été une source d'inspiration à plusieurs reprises, notamment dans la première partie de la thèse.

Pour finir, on peut mentionner les résultats de ces dernières années sur le sujet. Plusieurs auteurs [HHL16b, CT19, HH19a] ont amélioré la complexité de la multiplication entière à $O(n \log n\, 4^{\log^* n})$, en supposant diverses conjectures de théorie des nombres. Peu après, une autre preuve sans ces conjectures a été apportée [HH19b]. Dans le même temps [HH19c], la borne $O(n \log n\, 4^{\log^* n})$ a été établie également pour les polynômes sur les corps finis. Très récemment, Harvey et van der Hoeven ont prouvé que la multiplication avait une complexité de $O(n \log n)$, aussi bien pour les entiers [HH19d] que pour les polynômes à coefficients dans un corps fini [HH19e]. Dans leur article de 1971 [SS71], près de 50 ans plus tôt, Schönhage et Strassen avaient conjecturé qu'il n'est pas possible de descendre en dessous de $n \log n$. Cette borne inférieure n'a pas encore été prouvée, mais elle est largement admise. Rappelons toutefois que l'algorithme naïf (quadratique) était initialement considéré optimal, à tort ; une preuve serait donc nécessaire pour apporter une réponse définitive. Mais en admettant la conjecture de Schönhage et Strassen, cela voudrait dire que l'algorithme de Harvey et van der Hoeven est optimal.

La Transformée de Fourier Rapide. Comme on l'a dit, les algorithmes rapides comme celui de Schönhage-Strassen [SS71] et les suivants reposent sur la transformée de Fourier Rapide ou FFT, et plus généralement sur le principe d'évaluation-interpolation.

Pour multiplier des polynômes par évaluation-interpolation, on va simplement évaluer les deux polynômes d'entrée en suffisamment de points, multiplier ces évaluations terme-à-terme, et interpoler le résultat. Ce principe s'étend à la multiplication entière en voyant les grands entiers comme des polynômes à petits coefficients entiers (par exemple $123 = P(10)$ avec $P := X^2 + 2X + 3$). Étant donnée une racine n-ième d'unité $\omega \in \mathbb{K}$ (c.a.d. avec $\omega^n = 1$), la Transformée de Fourier Discrète (DFT) de P est le vecteur $\big(P(1), P(\omega), \ldots, P(\omega^{n-1})\big)$. L'expression *Transformée de Fourier Rapide* désigne tout algorithme permettant de calculer une DFT efficacement. En d'autres termes, la Transformée de Fourier Rapide donne un schéma d'évaluation-interpolation efficace.

Les premières contributions de la thèse sont donc des variantes améliorées de la FFT, qui donneraient des algorithmes de multiplication plus rapides. Si ces variantes restent correctes dans un cadre plus général, elles étaient initialement motivées par l'exemple de $\mathbb{F}_{2^{60}}$ présenté dans [HHL16a].

Systèmes polynomiaux. La résolution de systèmes d'équations algébriques est une extension naturelle lorsqu'on parle de polynômes, et la thèse s'intéresse également à cette question. D'un point de vue calcul formel, ce problème est très vaste et il existe plusieurs manières d'y répondre.

Dans le langage courant, « résoudre » signifie « trouver toutes les solutions », c'est à dire donner une valeur approchée (à une précision arbitraire) de chaque solution, ou décrire l'hypersurface formée par l'ensemble des solutions. Des réponses plus partielles sont également acceptables, par exemple on ne s'intéresse parfois qu'au

nombre de solutions si celui-ci est fini. Éventuellement, on peut vouloir déterminer si un polynôme donné s'annule sur l'ensemble des racines communes du système. Une approche indirecte consiste à donner des algorithmes pour calculer modulo l'idéal engendré par le système ; répondre à cette question permet de déduire le nombre de solutions du système (égal à la dimension de l'algèbre quotient), ainsi que de détecter si un polynôme s'annule sur l'ensemble des racines (si et seulement si une de ses puissances appartient à l'idéal).

Dans une deuxième série de contributions, la thèse aborde la question de la résolution de système polynomiaux du point de vue arithmétique. Plus spécifiquement, on s'intéresse au sous-problème du calcul modulo l'idéal engendré par le système. Pour les raisons mentionnées précédemment, cela revient à résoudre le système d'équations en un certain sens, ou au moins peut constituer une étape vers une résolution complète.

B.2 Contributions

Les résultats de cette thèse se concentrent donc principalement autour de deux thèmes, à savoir la multiplication polynomiale et la résolution de systèmes d'équations algébriques. Pour des rappels historiques et un état de l'art sur ces deux sujets, on pourra se référer à l'introduction en anglais de ce manuscrit (section 1.2).

La première partie traite de la multiplication de polynômes à une variable d'un point de vue pratique. Il ne s'agit pas ici d'améliorer la complexité asymptotique, mais plutôt de réduire la constante cachée dans la notation $O(\cdot)$, dans l'espoir d'obtenir des implémentations plus rapides.

La deuxième partie s'intéresse à l'arithmétique des polynômes à plusieurs variables modulo un idéal. Ce problème étant en général très complexe (en espace exponentiel dans le pire des cas [May89]), on se concentre sur une situation simplifiée avec seulement deux variables et certaines hypothèses de régularité. Dans ce cas précis, on peut donner des algorithmes quasi-optimaux, c'est à dire linéaires en la taille de l'entrée/sortie à des facteurs logarithmiques près. Dans son champ d'application restreint, ce résultat théorique est alors bien meilleur que les algorithmes classiques plus généralistes.

B.2.1 Multiplication de polynômes à une variable

Comme on l'a vu un peu plus tôt, les algorithmes de multiplication rapide reposent sur le principe d'évaluation-interpolation et la transformée de Fourier rapide. C'est pourquoi l'amélioration des algorithmes de multiplication passent en fait par la définition de variantes de FFT, permettant de limiter certains calculs superflus. Avant de donner les résultats de cette partie, il est important de présenter le fonctionnement de l'évaluation-interpolation et de la FFT. Ces rappels classiques font l'objet du Chapitre 2.

La transformée de Fourier tronquée. La multiplication basée sur la FFT est connue pour son comportement « en escalier » : on observe des sauts importants lorsque la taille est une puissance de 2, et peu d'augmentation entre les sauts. L'objectif de la transformée de Fourier tronquée (ou TFT) est de limiter cet effet de saut pour avoir une augmentation plus progressive. Si une FFT de taille n peut être calculée en $\mathsf{F}(n)$ opérations, alors la TFT permet de réaliser une évaluation ou une interpolation pour $\ell < n$ points en temps

$$\frac{\ell}{n}\mathsf{F}(n) + O(n)\,.$$

Ce résultat était déjà connu dans le cas où n est une puissance d'un nombre premier. La première contribution de cette thèse est une généralisation de l'algorithme pour une taille n quelconque, et fait l'objet du Chapitre 3. Ce résultat a été publié dans [Lar17] et présenté à la conférence internationale ISSAC en 2017.

Si le cas le plus classique $n = 2^k$ est bien compris et implémenté dans divers logiciels (tels que FLINT [Har10], Mathemagix [HLM+02] ou NTL [Sho01]), le cas général est beaucoup plus difficile à traiter de manière compétitive par rapport à une FFT très optimisée. Les tailles de FFT avec une décomposition mixte $n = p_1^{e_1} \cdots p_k^{e_k}$ sont pourtant fréquentes dans les corps finis et il serait intéressant d'avoir une TFT applicable dans ce contexte. Sans viser une implémentation généraliste, il était initialement prévu de donner une version de la TFT pour $\mathbb{F}_{2^{60}}$ (motivée par l'article [HHL16a] déjà mentionné), mais le résultat suivant s'est avéré plus prometteur.

L'algorithme « Frobenius FFT ». Une deuxième contribution de la thèse est une variante de la FFT spécifique pour les corps finis, et plus précisément lorsqu'on calcule une transformée de Fourier dans une extension du corps des coefficients du polynôme d'entrée. Cela arrive notamment pour multiplier des polynômes sur un petit corps fini, car il n'y a alors pas assez de points pour la technique d'évaluation-interpolation ; il est alors nécessaire de travailler dans une extension plus grande, ce qui entraîne un surcoût. L'idée est d'utiliser des symétries fournies par le morphisme de Frobenius (d'où le nom de l'algorithme). La technique est en fait similaire à l'utilisation de la conjugaison complexe pour accélérer une « FFT réelle ». Concrètement, si P est un polynôme à coefficients dans $\mathbb{F}_q$ et qu'on souhaite calculer sa FFT dans $\mathbb{F}_{q^d}$, alors il est possible de le faire environ d fois plus vite que si P avait ses coefficients dans $\mathbb{F}_{q^d}$ (voir Chapitre 4). Ce résultat a été publié dans [HL17] et présenté à la conférence internationale ISSAC en 2017.

Ici encore, la technicité de l'algorithme rend difficile de réaliser une implémentation compétitive dans le cas général. Cependant, il fonctionne particulièrement bien pour multiplier des polynômes sur $\mathbb{F}_2$ en utilisant l'extension $\mathbb{F}_{2^{60}}$. L'implémentation dans ce cas précis fournit donc une application intéressante à l'algorithme de « Frobenius FFT ».

Multiplication dans $\mathbb{F}_2[X]$. Comme on l'a dit, les propriétés de l'extension $\mathbb{F}_{2^{60}}$ permettent de multiplier efficacement les polynômes sur $\mathbb{F}_2$. Pour faire la conversion de $\mathbb{F}_2[X]$ vers $\mathbb{F}_{2^{60}}[X]$, on peut simplement voir les coefficients du polynôme comme des éléments de $\mathbb{F}_{2^{60}}$, mais cela causerait un surcoût d'un facteur équivalent au degré de l'extension (60 dans ce cas), ce qui n'est évidemment pas acceptable. Par la technique classique de substitution de Kronecker [Kro82], il est possible de gagner simplement un facteur correspondant à la moitié du degré de l'extension ; le surcoût n'est donc plus que d'un facteur 2 (c'est cette méthode qui est utilisée dans le papier [HHL16a]). D'un autre côté, la technique de Frobenius FFT précédente permet en principe d'éliminer totalement le surcoût, donc on peut espérer une implémentation 2 fois plus rapide que la précédente. Grâce à une modification mineure de l'algorithme, cette accélération peut effectivement être observée en pratique. Les détails sont donnés dans le Chapitre 5.

Le programme est distribué dans le paquet JUSTINLINE du logiciel MATHEMAGIX [HLM+02], sous licence GPL. Cette implémentation, environ 2 fois plus rapide que les logiciels précédents, constitue la troisième contribution de la thèse. Les résultats ont été publiés dans [HLL17] et présentés à la conférence internationale MACIS en 2017.

B.2.2 Réduction des polynômes bivariés

Pour la deuxième partie, on considère le problème du calcul modulo un idéal de polynômes à plusieurs variables. Ce problème intervient notamment pour la résolution de systèmes polynomiaux, ou pour construire des anneaux avec certaines propriétés algébriques. La difficulté principale est le calcul de formes normales, c.a.d. le représentant canonique d'une classe d'équivalence.

En une variable (et pour des coefficients dans un corps), la solution est très simple : tout idéal est principal, engendré par le PGCD des polynômes définissant l'idéal, et une fois ce PGCD connu, le calcul de formes normales est simplement une division euclidienne. À noter que dans ce cas, tous les algorithmes sont linéaires à des facteurs logarithmiques près, donc essentiellement optimaux. On rappelle la notation $\tilde{O}(\cdot)$ dite « soft-Oh » [GG13, section 25.7] pour cacher ces facteurs logarithmiques. Pour les polynômes à plusieurs variables, les idéaux ne sont plus principaux, mais ils possèdent une base standard, ou *base de Gröbner* qui assure l'existence et l'unicité de la forme normale. Dans ce cas, les algorithmes ne sont en revanche pas optimaux ; l'objectif de cette partie est donc de proposer des algorithmes optimaux sous certaines hypothèses. Pour simplifier, on considère seulement le cas des polynômes à deux variables, et on suppose que l'idéal présente une certaine régularité. Le Chapitre 6 rappelle les généralités sur les bases de Gröbner puis présente les idées principales pour calculer des formes normales plus rapidement. Les Chapitres 7 et 8 détaillent ensuite la solution pour deux types d'hypothèses de régularité.

Techniques de réduction. Soient $A, B \in \mathbb{K}[X, Y]$ deux polynômes bivariés de degré n. Dans une situation générique, ils ont n^2 racines communes, et l'algèbre quo-

tient $\mathbb{A} := \mathbb{K}[X,Y]/\langle A, B\rangle$ est de dimension n^2. On pourrait donc espérer des algorithmes essentiellement quadratiques pour le calcul dans $\mathbb{A}$, pourtant les algorithmes classiques sont au moins cubiques. La raison principale est que la base de Gröbner de l'idéal $\langle A, B\rangle$ est de taille $\Theta(n^3)$, car elle contient $\Theta(n)$ éléments ayant $\Theta(n^2)$ coefficients chacun. Pour calculer plus rapidement des formes normales, nous devons donc compresser l'information contenue dans la base de Gröbner, idéalement en espace quadratique (à des facteurs logarithmiques près).

Soit $G := G^{(0)}, \ldots, G^{(n)}$ une base de Gröbner. Le but d'un algorithme de réduction est de déterminer des polynômes $Q^{(0)}, \ldots, Q^{(n)}$ tels que $P = Q^{(0)}G^{(0)} + \cdots + Q^{(n)}G^{(n)} + R$ où R est la forme normale (au sens où aucun de ses termes n'est divisible par le terme de tête d'un des $G^{(i)}$). Par analogie avec la division euclidienne, on appelle $Q^{(0)}, \ldots, Q^{(n)}$ les *quotients*, et R le *reste*. Au cours de l'algorithme, chaque terme de P doit être réduit par rapport à un certain élément de la base de Gröbner, mais il peut y avoir plusieurs candidats. Les algorithmes doivent donc faire un choix à chaque étape pour sélectionner un candidat ; si le reste ne dépend pas des choix effectués, les quotients eux en dépendent. C'est cette liberté qui autorise des techniques de réduction plus efficaces.

La première idée est d'utiliser une *stratégie de sélection dichotomique* pour contrôler le degré des quotients. De cette façon, on s'assure que la majorité des quotients ont un très petit degré, certains sont un peu plus gros, quelques uns encore un peu plus, et ainsi de suite. Grâce aux bornes sur le degré, on peut ensuite *tronquer les éléments de la base* : si on sait que le quotient $Q^{(i)}$ associé à $G^{(i)}$ est de degré δ, alors il suffit pour les calculs de connaître $G^{(i)}$ avec la précision correspondante, on ne va donc garder dans $G^{(i)}$ que les termes de degré au moins $\deg G^{(i)} - \delta$. Ceci permet de réduire la taille de la base à $\tilde{O}(n^2)$ coefficients utiles, mais le résultat ne serait plus correct. La troisième idée est donc de *réécrire l'équation* de réduction au cours du calcul, pour augmenter la précision. Pour ce faire, on stocke certaines relations bien choisies entre les $G^{(i)}$, ce qui permet de mener le calcul à bien tout en ne demandant qu'un espace raisonnable. Cette méthode, présentée dans le Chapitre 6 constitue une autre contribution de la thèse. Ces techniques se déclinent avec de légères différences dans les deux situations suivantes.

Bases de Gröbner « vanilla ». Le Chapitre 7 définit une classe de bases de Gröbner spécialement conçue pour que les techniques ci-dessus fonctionnent. Par définition, ces *bases de Gröbner « vanilla »* possèdent une représentation dite succincte (ou *terse representation* en anglais), laquelle demande un espace de stockage essentiellement quadratique comme espéré. Une fois cette représentation précalculée, il est possible de calculer une forme normale en temps quasi-linéaire par rapport à la taille du polynôme à réduire et à la dimension de l'algèbre $\mathbb{A}$ (en tant que $\mathbb{K}$-espace vectoriel). En particulier, la multiplication dans $\mathbb{A}$ peut alors se faire en $\tilde{O}(\dim_{\mathbb{K}} \mathbb{A})$ opérations.

Il peut sembler déraisonnable de choisir comme hypothèse de régularité « l'existence de la structure dont on a besoin pour faire fonctionner l'algorithme », mais expérimentalement cette situation semble être générique : un idéal engendré par des

polynômes aléatoires admet une base vanilla avec grande probabilité. Ce comportement est observé pour plusieurs ordres monomiaux, et plusieurs formes de support pour les générateurs de l'idéal. À l'avenir, il serait intéressant d'avoir une preuve de généricité, plus convaincante que cet argument empirique ; ce résultat (le premier algorithme quasi-optimal pour calculer une forme normale, bien que dans une situation assez spécifique) ouvre cependant des perspectives théoriques intéressantes. L'algorithme a été publié dans [HL18a] et présenté à la conférence internationale ISSAC en 2018. Une implémentation « preuve-de-concept » pour le système de calcul formel SageMath [Sag17] est disponible (sous licence GPL) à cette adresse :

`https://hal.archives-ouvertes.fr/hal-01770408/file/implementation.zip`

Cas de l'ordre Grevlex. Dans le Chapitre 8, on considère des polynômes générateurs donnés par rapport au degré total ($A = a_{0,n}Y^n + a_{1,n-1}XY^{n-1} + \cdots + a_{0,0}$, et de même pour B), et on calcule la base de Gröbner par rapport à l'ordre Grevlex connu pour être le plus efficace. Cette situation pourtant classique est une des exceptions au contexte des bases vanilla, mais on observe malgré tout un autre type de structure. En supposant qu'il n'y a pas d'annulation accidentelle (ce qui est une hypothèse de généricité raisonnable), on peut prouver qu'il existe une *représentation concise* pour la base de Gröbner, avec des propriétés similaires à la représentation succincte précédente. De plus, la base de Gröbner concise peut être calculée efficacement, en temps $\tilde{O}(n^2)$, à partir des générateurs A, B (alors que la représentation succincte était un précalcul possiblement onéreux). Ces résultats ont été présentés à la conférence internationale ACA en 2018, et publiés dans le journal « Applicable Algebra in Engineering, Communication and Computing » [HL19a].

L'implémentation « preuve-de-concept » mentionnée précédemment traite aussi cette situation. Par ailleurs, une implémentation plus efficace a été intégrée au logiciel Mathemagix [HLM+02], dans le paquet larrix (distribué sous licence GPL). Cette implémentation pour le cas spécifique considéré est nettement plus rapide que les logiciels généralistes de référence : pour $n = 200$, l'accélération est d'un facteur environ 150 (et l'écart est de plus en plus important quand n augmente, du fait de la meilleure complexité asymptotique). Ce paquet a fait l'objet d'une démonstration logicielle à ISSAC en 2019, et le rapport sera publié dans les "ACM SIGSAM Communications in Computer Algebra" [Lar19].

Comme application de ce résultat, van der Hoeven et Lecerf ont donné un algorithme de complexité $n^{2+o(1)}$ pour le résultant bivarié générique sur les corps finis [HL19c], sous l'hypothèse de composition modulaire rapide.

B.3 Liste des publications

Cette section liste les publications produites pendant la thèse. Tous les articles ont été publiés (après comité de lecture) dans des actes de conférences internationales ou des journaux scientifiques. Les logiciels sont librement distribués sous les termes de la GNU General Public License (GPL, version 2 ou suivantes).

Articles de recherche

[Lar17] Robin Larrieu. The Truncated Fourier Transform for mixed radices. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 261–268, New York, NY, USA, 2017. ACM

[HL17] Joris van der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 437–444, New York, NY, USA, 2017. ACM

[HLL17] Joris van der Hoeven, Robin Larrieu, and Grégoire Lecerf. Implementing fast carryless multiplication. In J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos, editors, *Proceedings of Mathematical Aspects of Computer and Information Sciences*, pages 121–136, Cham, 2017. Springer

[HL18a] Joris van der Hoeven and Robin Larrieu. Fast reduction of bivariate polynomials with respect to sufficiently regular Gröbner bases. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 199–206, New York, NY, USA, 2018. ACM

[HL19a] Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, June 2019

[Lar19] Robin Larrieu. Computing generic bivariate Gröbner bases with Mathemagix, 2019. ISSAC software demonstration, to appear in ACM SIGSAM Communications in Computer Algebra

Logiciels

- Routines pour la multiplication de polynômes dans $\mathbb{F}_2[X]$ (voir [HLL17]), en collaboration avec Grégoire Lecerf et Joris van der Hoeven. Fait partie du paquet JUSTINLINE du logiciel MATHEMAGIX [HLM+02].
  `http://www.mathemagix.org`

- Une implémentation « preuve-de-concept » pour SAGEMATH [Sag17] des algorithmes pour les bases de Gröbner bivariées génériques [HL18a, HL19a].
  `https://hal.archives-ouvertes.fr/hal-01770408/file/implementation.zip`

- Une bibliothèque efficace pour le calcul des bases de Gröbner bivariées génériques dans le contexte de [HL19a]. Intégré dans MATHEMAGIX [HLM+02] comme le paquet LARRIX. `http://www.mathemagix.org`

- Une implémentation expérimentale pour le calcul avec des extensions de corps finis, en collaboration avec Luca De Feo. Disponible à l'adresse suivante : `https://gitlab.inria.fr/rlarrieu/tower`
  La méthode et des résultats provisoires sont donnés en Annexe A.

Bibliography

[Abe12] Niels Henrik Abel. Mémoire sur les équations algébriques, où l'on démontre l'impossibilité de la résolution de l'équation générales du cinquième degré (1824). In L. Sylow and S. Lie, editors, *Oeuvres Complètes de Niels Henrik Abel: Nouvelle Édition*, volume 1. Cambridge University Press, 2012.

[AG90] Eugene L. Allgower and Kurt Georg. *Numerical Continuation Methods: An Introduction*. Springer-Verlag, Berlin, Heidelberg, 1990.

[AJJ96] Louis Auslander, Jeremy R. Johnson, and Robert W. Johnson. An equivariant Fast Fourier Transform algorithm. Drexel University Technical Report DU-MCS-96-02, 1996.

[All02] Bill Allombert. Explicit computation of isomorphisms between finite fields. *Finite Fields and Their Applications*, 8(3):332 – 342, 2002.

[ALM99] Philippe Aubry, Daniel Lazard, and Marc Moreno Maza. On the theories of triangular sets. *Journal of Symbolic Computation*, 28(1):105–124, July 1999.

[AM99] Philippe Aubry and Marc Moreno Maza. Triangular sets for solving polynomial systems: a comparative implementation of four methods. *Journal of Symbolic Computation*, 28(1):125 – 154, 1999.

[AS88] Winfried Auzinger and Hans J. Stetter. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In Ravi P. Agarwal, Y. M. Chow, and S. J. Wilson, editors, *Proceedings of the International Conference on Numerical Mathematics*, pages 11–30, Basel, 1988. Birkhäuser Basel.

[Bai89] David H. Bailey. FFTs in external or hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989.

[BCG+17] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. *Algorithmes Efficaces en Calcul Formel*. Frédéric Chyzak (auto-édit.), August 2017. 686 pages. Édition 1.0, https://hal.archives-ouvertes.fr/AECF/.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[BCS97] Wieb Bosma, John Cannon, and Allan Steel. Lattices of compatibly embedded finite fields. *Journal of Symbolic Computation*, 24(3):351 – 369, 1997.

[BDD+17] Ludovic Brieulle, Luca De Feo, Javad Doliskani, Jean-Pierre Flori, and Éric Schost. Computing isomorphisms and embeddings of finite fields. Extended version, http://arxiv.org/abs/1705.01221, 2017.

[BDD+19] Ludovic Brieulle, Luca De Feo, Javad Doliskani, Jean-Pierre Flori, and Éric Schost. Computing isomorphisms and embeddings of finite fields. *Mathematics of Computation*, 88(317):1391–1426, 2019.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[Ber68] Glenn D. Bergland. Numerical Analysis: A fast Fourier transform algorithm for real-valued series. *Communications of the ACM*, 11(10):703–710, 1968.

[Ber04] Daniel J. Bernstein. Scaled remainder trees, 2004. http://cr.yp.to/arith/scaledmod-20040820.pdf.

[Ber13] Ronny Bergmann. The fast Fourier transform and fast wavelet transform for patterns on the torus. *Applied and Computational Harmonic Analysis*, 35(1):39–51, 2013.

[BFS14] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of the F5 Gröbner basis algorithm. *Journal of Symbolic Computation*, pages 1–24, September 2014.

[BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in GF(2)[x]. In A. van der Poorten and A. Stein, editors, *Algorithmic Number Theory*, volume 5011 of *Lect. Notes Comput. Sci.*, pages 153–166. Springer, 2008.

[BHSW06] Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler. Bertini: Software for numerical algebraic geometry, 2006. https://bertini.nd.edu/.

[BK78] Richard P. Brent and Hsiang T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM (JACM)*, 25(4):581–595, October 1978.

[BK12] Egbert Brieskorn and Horst Knörrer. *Plane Algebraic Curves*. Springer Science & Business Media, 2012. Translated by John Stillwell.

[Ble07] Dieter Blessenohl. On the normal basis theorem. *Note di Matematica*, 27(1):5–10, 2007.

[BLM+16] Yacine Bouzidi, Sylvain Lazard, Guillaume Moroz, Marc Pouget, Fabrice Rouillier, and Michael Sagraloff. Solving bivariate systems using rational univariate representations. *Journal of Complexity*, 37:34 – 75, 2016.

[BLS03] Alin Bostan, Grégoire Lecerf, and Éric Schost. Tellegen's principle into practice. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, ISSAC '03, pages 37–44, New York, NY, USA, 2003. ACM.

[Blu70] Leo Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, December 1970.

[Bor57] Jan L. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research, Section A*, 6(1):1–74, January 1957.

[BS88] David Bayer and Michael Stillman. On the complexity of computing syzygies. *Journal of Symbolic Computation*, 6(2-3):135–147, December 1988.

[BS04] Alin Bostan and Éric Schost. On the complexities of multipoint evaluation and interpolation. *Theoretical Computer Science*, 329(1):223 – 235, 2004.

[BS05] Alin Bostan and Éric Schost. Polynomial evaluation and interpolation on special sets of points. *Journal of Complexity*, 21(4):420 – 446, 2005.

[BSS03] Alin Bostan, Bruno Salvy, and Éric Schost. Fast algorithms for zero-dimensional polynomial systems using duality. *Applicable Algebra in Engineering, Communication and Computing*, 14(4):239–272, November 2003.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal.* PhD thesis, Universitat Innsbruck, Austria, 1965.

[Buc79] Bruno Buchberger. A criterion for detecting unnecessary reductions in the construction of Gröbner-bases. In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pages 3–21, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner bases: a computational approach to commutative algebra*, volume 141 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1993.

[CA69] Stephen A Cook and Stål O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969.

[CAD17] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm, 2017. Release 2.3.0.

[Can89] David G. Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285 – 300, 1989.

[CCK+17] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. https://arxiv.org/abs/1708.09746, 2017.

[CCK+18] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean polynomials with Frobenius partitions in additive fast Fourier transform. *CoRR*, abs/1803.11301, 2018.

[CF94] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of computation*, 62(205):305–324, January 1994.

[CK91] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[CKM97] Stéphane Collart, Michael Kalkbrener, and Daniel Mall. Converting bases with the Gröbner walk. *Journal of Symbolic Computation*, 24(3):465 – 469, 1997.

[CLO92] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms: An introduction to computational algebraic geometry and commutative algebra*. Undergraduate Texts in Mathematics. Springer-Verlag, 1992.

[Cou00] Jean-Marc Couveignes. Isomorphisms between Artin-Schreier towers. *Mathematics of Computation*, 69(232):1625–1631, 2000.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, April 1965.

[CT19] Svyatoslav Covanov and Emmanuel Thomé. Fast integer multiplication using generalized Fermat primes. *Mathematics of Computation*, 88(317):1449–1477, 2019.

[DDS13] Luca De Feo, Javad Doliskani, and Éric Schost. Fast algorithms for ℓ-adic towers over finite fields. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, pages 165–172, New York, NY, USA, 2013. ACM.

[DDS14] Luca De Feo, Javad Doliskani, and Éric Schost. Fast arithmetic for the algebraic closure of finite fields. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 122–129, New York, NY, USA, 2014. ACM.

[DET09] Dimitrios I. Diochnos, Ioannis Z. Emiris, and Elias P. Tsigaridas. On the asymptotic and practical complexity of solving bivariate systems over the reals. *Journal of Symbolic Computation*, 44(7):818 – 835, 2009. International Symposium on Symbolic and Algebraic Computation.

[DGPS17] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-1-0 — A computer algebra system for polynomial computations, 2017. http://www.singular.uni-kl.de.

[Dre77] Franz-Josef Drexler. Eine Methode zur Berechnung sämtlicher Lösungen von Polynomgleichungssystemen. *Numerische Mathematik*, 29(1):45–58, March 1977.

[DRR19] Luca De Feo, Hugues Randriam, and Édouard Rousseau. Standard lattices of compatibly embedded finite fields. In *Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation*, ISSAC '19, pages 122–130, New York, NY, USA, 2019. ACM.

[DS12] Luca De Feo and Éric Schost. Fast arithmetics in Artin-Schreier towers over finite fields. *Journal of Symbolic Computation*, 47(7):771–792, July 2012.

[DS15] Javad Doliskani and Éric Schost. Computing in degree 2^k-extensions of finite fields of odd characteristic. *Designs, Codes and Cryptography*, 74(3):559–569, March 2015.

[ES12] Pavel Emeliyanenko and Michael Sagraloff. On the complexity of solving a bivariate polynomial system. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, pages 154–161, New York, NY, USA, 2012. ACM.

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61 – 88, 1999.

[Fau02] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, pages 75–83, New York, NY, USA, 2002. ACM.

[Fau10] Jean-Charles Faugère. FGb: a library for computing Gröbner bases. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 84–87, Berlin, Heidelberg, September 2010. Springer Berlin / Heidelberg.

[FGHR13] Jean-Charles Faugère, Pierrick Gaudry, Louise Huot, and Guénaël Renault. Polynomial systems solving by fast linear algebra. *arXiv preprint arXiv:1304.6039*, 2013.

[FGHR14] Jean-Charles Faugère, Pierrick Gaudry, Louise Huot, and Guénaël Renault. Sub-cubic change of ordering for Gröbner basis: A probabilistic approach. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 170–177, New York, NY, USA, 2014. ACM.

[FGLM93] Jean-Charles Faugere, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4):329–344, 1993.

[FH94] Ralf Fröberg and Joachim Hollman. Hilbert series for ideals generated by generic forms. *Journal of Symbolic Computation*, 17(2):149 – 157, 1994.

[FHHJ17] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/Hecke: Computer algebra and number theory packages for the Julia programming language. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 157–164, New York, NY, USA, 2017. ACM.

[FHK+17] Jean-Charles Faugere, Kelsey Horan, Delaram Kahrobaei, Marc Kaplan, Elham Kashefi, and Ludovic Perret. Fast quantum algorithm for solving multivariate quadratic equations. *arXiv preprint arXiv:1712.07211*, 2017.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[Fid72] Charles M. Fiduccia. On obtaining upper bounds on the complexity of matrix multiplication. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations*, pages 31–40. Springer US, Boston, MA, 1972.

[FJ03] Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of Hidden Field Equation (HFE) cryptosystems using Gröbner bases. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FS74] Michael J. Fischer and Larry J. Stockmeyer. Fast on-line integer multiplication. *Journal of Computer and System Sciences*, 9(3):317–331, 1974.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[Gal74] André Galligo. A propos du théoreme de préparation de Weierstrass. In *Fonctions de plusieurs variables complexes*, pages 543–579. Springer, 1974.

[Gau99] Carl Friedrich Gauss. *Demonstratio nova theorematis omnem functionem algebraicam rationalem integram unius variabilis in factores reales primi vel secundi gradus resolvi posse*. Apud CG Fleckeisen, 1799.

[Gau66] Carl Friedrich Gauss. Nachlass: Theoria interpolationis methodo nova tractata. In *Werke*, volume 3, pages 265–330. Königliche Gesellschaft der Wissenschaften, Göttingem, 1866.

[GB98] Vladimir P. Gerdt and Yuri A. Blinkov. Involutive bases of polynomial ideals. *Mathematics and Computers in Simulation*, 45(5):519 – 541, 1998.

[GCC87] GCC, the GNU Compiler Collection, from 1987. Software available at http://gcc.gnu.org.

[GG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, 2013. 3rd edition.

[GH91] Marc Giusti and Joos Heintz. Algorithmes – disons rapides – pour la décomposition d'une variété algébrique en composantes irréductibles et équidimensionnelles. In T. Mora and C. Traverso, editors, *Effective Methods in Algebraic Geometry*, pages 169–194. Birkhäuser Boston, Boston, MA, 1991.

[GHM+98] Marc Giusti, Joos Heintz, Jose Enrique Morais, Jacques Morgenstem, and Luis Miguel Pardo. Straight-line programs in geometric elimination theory. *Journal of Pure and Applied Algebra*, 124(1):101 – 146, 1998.

[GLS01] Marc Giusti, Grégoire Lecerf, and Bruno Salvy. A Gröbner free alternative for polynomial system solving. *Journal of Complexity*, 17(1):154 – 211, 2001.

[GM86] Abderrezak Guessoum and Russel M. Mersereau. Fast algorithms for the multidimensional discrete Fourier transform. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(4):937–943, August 1986.

[GM91a] Giovanni Gallo and Bhubaneswar Mishra. Efficient algorithms and bounds for Wu-Ritt characteristic sets. In T. Mora and C. Traverso, editors, *Effective Methods in Algebraic Geometry*, pages 119–142. Birkhäuser Boston, Boston, MA, 1991.

[GM91b] Giovanni Gallo and Bud Mishra. Wu-Ritt characteristic sets and their complexity. *Discrete and Computational Geometry: Papers from the DIMACS Special Year*, 6:111–136, 1991.

[GM10] Shuhong Gao and Todd Mateer. Additive fast Fourier transforms over finite fields. *IEEE Trans. Inform. Theory*, 56(12):6265–6272, 2010.

[GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "One sugar cube, please"; or selection strategies in the Buchberger algorithm. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, ISSAC '91, pages 49–54, New York, NY, USA, 1991. ACM.

[Goo58] Irving John Good. The interaction algorithm and practical Fourier analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958.

[Gra91] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, from 1991. http://gmplib.org/.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/.

[GVK96] Laureano González-Vega and M'hammed El Kahoui. An improved upper complexity bound for the topology computation of a real algebraic plane curve. *Journal of Complexity*, 12(4):527 – 544, 1996.

[GVN02] Laureano Gonzalez-Vega and Ioana Necula. Efficient topology determination of implicitly defined algebraic plane curves. *Computer Aided Geometric Design*, 19(9):719 – 743, 2002.

[Har09] David Harvey. A cache-friendly truncated FFT. *Theoretical Computer Science*, 410(27):2649 – 2658, 2009.

[Har10] William B. Hart. Fast Library for Number Theory: An Introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. http://flintlib.org.

[Har11] David Harvey. Faster algorithms for the square root and reciprocal of power series. *Mathematics of Computation*, 80(273):387–394, 2011.

[HH19a] David Harvey and Joris van der Hoeven. Faster integer multiplication using plain vanilla FFT primes. *Mathematics of Computation*, 88(315):501–514, 2019.

[HH19b] David Harvey and Joris van der Hoeven. Faster integer multiplication using short lattice vectors. In R. Scheidler and J. Sorenson, editors, *Proc. of the 13-th Algorithmic Number Theory Symposium*, Open Book Series 2, pages 293–310. Mathematical Sciences Publishes, Berkeley, 2019.

[HH19c] David Harvey and Joris van der Hoeven. Faster polynomial multiplication over finite fields using cyclotomic coefficient rings. *Journal of Complexity*, 2019.

[HH19d] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. Technical report, HAL, March 2019. https://hal.archives-ouvertes.fr/hal-02070778.

[HH19e] David Harvey and Joris van der Hoeven. Polynomial multiplication over finite fields in time $O(n \log n)$. Technical report, HAL, March 2019. https://hal.archives-ouvertes.fr/hal-02070816.

[HHL16a] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Fast polynomial multiplication over $\mathbb{F}_{2^{60}}$. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 255–262, New York, NY, USA, 2016. ACM.

[HHL16b] Davis Harvey, Joris van der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1 – 30, 2016.

[HHL17] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Faster polynomial multiplication over finite fields. *Journal of the ACM (JACM)*, 63(6):52:1–52:23, January 2017.

[Hil90] David Hilbert. Ueber die Theorie der algebraischen Formen. *Mathematische Annalen*, 36(4):473–534, December 1890.

[Hir64] Heisuke Hironaka. Resolution of singularities of an algebraic variety over a field of characteristic zero: I. *Annals of Mathematics*, 79(1):109–203, 1964.

[HL04] Lenwood S. Heath and Nicholas A. Loehr. New algorithms for generating Conway polynomials over finite fields. *Journal of Symbolic Computation*, 38(2):1003 – 1024, 2004.

[HL13a] Joris van der Hoeven and Grégoire Lecerf. Interfacing Mathemagix with C++. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, pages 363–370, New York, NY, USA, 2013. ACM.

[HL13b] Joris van der Hoeven and Grégroire Lecerf. Mathemagix User Guide. https://hal.archives-ouvertes.fr/hal-00785549, 2013.

[HL17] Joris van der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 437–444, New York, NY, USA, 2017. ACM.

[HL18a] Joris van der Hoeven and Robin Larrieu. Fast reduction of bivariate polynomials with respect to sufficiently regular Gröbner bases. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 199–206, New York, NY, USA, 2018. ACM.

[HL18b] Joris van der Hoeven and Grégoire Lecerf. On the complexity exponent of polynomial system solving. https://hal.archives-ouvertes.fr/hal-01848572, July 2018.

[HL18c] Joris van der Hoeven and Grégoire Lecerf. Directed evaluation. Technical report, HAL, 2018. http://hal.archives-ouvertes.fr/hal-01966428.

[HL18d] Joris van der Hoeven and Grégoire Lecerf. Modular composition via factorization. *Journal of Complexity*, 48:36 – 68, 2018.

[HL19a] Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, June 2019.

[HL19b] Joris van der Hoeven and Grégoire Lecerf. Accelerated tower arithmetic. *Journal of Complexity*, page 101402, 2019.

[HL19c] Joris van der Hoeven and Grégoire Lecerf. Fast computation of generic bivariate resultants. Technical report, HAL, 2019. http://hal.archives-ouvertes.fr/hal-02080426.

[HLL17] Joris van der Hoeven, Robin Larrieu, and Grégoire Lecerf. Implementing fast carryless multiplication. In J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos, editors, *Proceedings of Mathematical Aspects of Computer and Information Sciences*, pages 121–136, Cham, 2017. Springer.

[HLM+02] Joris van der Hoeven, Grégoire Lecerf, Bernard Mourrain, et al. Mathemagix, from 2002. http://www.mathemagix.org.

[HLS13] Joris van der Hoeven, Romain Lebreton, and Éric Schost. Structured FFT and TFT: Symmetric and lattice polynomials. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, pages 355–362, New York, NY, USA, 2013. ACM.

[HNS19] Seung Gyu Hyun, Vincent Neiger, and Éric Schost. Implementations of efficient univariate polynomial matrix algorithms and application to bivariate resultants. In *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation*, ISSAC '19, pages 235–242, New York, NY, USA, 2019. ACM.

[Hoe77] John Hoe. *Les Systèmes d'équations polynômes dans le Siyuan Yujian: 1303*, volume 6. Collège de France, Institut des hautes études chinoises, 1977.

[Hoe02] Joris van der Hoeven. Relax, but don't be too lazy. *Journal of Symbolic Computation*, 34(6):479 – 542, 2002.

[Hoe04] Joris van der Hoeven. The Truncated Fourier Transform and applications. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 290–296, New York, NY, USA, 2004. ACM.

[Hoe10] Joris van der Hoeven. Newton's method and FFT trading. *Journal of Symbolic Computation*, 45(8):857–878, 2010.

[Hoe14] Joris van der Hoeven. Faster relaxed multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 405–412, New York, NY, USA, 2014. ACM.

[Hoe15] Joris van der Hoeven. On the complexity of polynomial reduction. In I. Kotsireas and E. Martínez-Moro, editors, *Proceedings of Applications of Computer Algebra 2015*, volume 198 of *Springer Proceedings in Mathematics and Statistics*, pages 447–458, Cham, 2015. Springer.

[Hoe16] Joris van der Hoeven. Faster Chinese remaindering. Technical report, HAL, 2016. http://hal.archives-ouvertes.fr/hal-01403810.

[HR10] David Harvey and Daniel S. Roche. An in-place Truncated Fourier Transform and applications to polynomial multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 325–329, New York, NY, USA, 2010. ACM.

[HS13] Joris van der Hoeven and Éric Schost. Multi-point evaluation in higher dimensions. *Applicable Algebra in Engineering, Communication and Computing*, 24(1):37–52, 2013.

[Jan20] Maurice Janet. Sur les systèmes d'équations aux dérivées partielles. *Journal de Mathématiques Pures et Appliquées*, 170:65–152, 1920.

[JX07] Jeremy R. Johnson and Xu Xu. Generating symmetric DFTs and equivariant FFT algorithms. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, pages 195–202, New York, NY, USA, 2007. ACM.

[Kal91] Michael Kalkbrener. *Three contributions to elimination theory.* PhD thesis, Johannes Kepler University, Linz, 1991.

[KO63] Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[Kro82] Leopold Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882.

[KRO07] Andrzej Kudlicki, Maga Rowicka, and Zbyszek Otwinowski. The crystallographic fast Fourier transform. Recursive symmetry reduction. *Acta Crystallographica Section A*, 63(6):465–480, November 2007.

[KU11] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011.

[Kun74] Hsiang T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22(5):341–348, October 1974.

[Lar17] Robin Larrieu. The Truncated Fourier Transform for mixed radices. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 261–268, New York, NY, USA, 2017. ACM.

[Lar19] Robin Larrieu. Computing generic bivariate Gröbner bases with Mathemagix, 2019. ISSAC software demonstration, to appear in ACM SIGSAM Communications in Computer Algebra.

[Laz91] Daniel Lazard. A new method for solving algebraic systems of positive dimension. *Discrete Applied Mathematics*, 33(1):147 – 160, 1991.

[Laz92] Daniel Lazard. Solving zero-dimensional algebraic systems. *Journal of Symbolic Computation*, 13(2):117 – 131, 1992.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast Fourier transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 263–270, New York, NY, USA, 2018. ACM.

[Leb15] Romain Lebreton. Relaxed Hensel lifting of triangular sets. *Journal of Symbolic Computation*, 68(P2):230–258, May 2015.

[Len91] Hendrik W. Lenstra. Finding isomorphisms between finite fields. *Mathematics of Computation*, 56(193):329–347, 1991.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM.

[LMS09] Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: From theory to practice. *Journal of Symbolic Computation*, 44(7):891 – 907, 2009. International Symposium on Symbolic and Algebraic Computation.

[LMS13] Romain Lebreton, Esmaeil Mehrabi, and Eric Schost. On the complexity of solving bivariate systems: The case of non-singular solutions. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, pages 251–258, New York, NY, USA, 2013. ACM.

[Mac02] Francis S. Macaulay. Some formulæ in elimination. *Proceedings of the London Mathematical Society*, s1-35(1):3–27, 1902.

[Mar71] John D. Markel. FFT pruning. *IEEE Transactions on Audio and Electroacoustics*, 19(4):305–311, December 1971.

[Mat08] Todd Mateer. *Fast Fourier Transform Algorithms with Applications*. PhD thesis, Clemson University, Clemson, SC, USA, 2008. AAI3316358.

[May89] Ernst Mayr. Membership in polynomial ideals over Q is exponential space complete. *STACS 89*, pages 400–406, 1989.

[Mer74] Russell M. Mersereau. An algorithm for performing an inverse chirp z-transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(5):387–388, October 1974.

[MGH+05] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.

[Möl88] H. Michael Möller. On the construction of Gröbner bases using syzygies. *Journal of Symbolic Computation*, 6(2):345 – 359, 1988.

[Moo66] Ramon E. Moore. *Interval analysis*, volume 4. Prentice-Hall, Englewood Cliffs, NJ, USA, 1966.

[Mor03] Guillermo Moreno-Socías. Degrevlex Gröbner bases of generic complete intersections. *Journal of Pure and Applied Algebra*, 180(3):263 – 283, 2003.

[Mor09] Alexander Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*, volume 57. Siam, 2009.

[NR97] Jean-Louis Nicolas and Guy Robin. Highly composite numbers by Srinivasa Ramanujan. *The Ramanujan Journal*, 1(2):119–153, 1997.

[Pan94] Victor Y. Pan. Simple multivariate polynomial multiplication. *Journal of Symbolic Computation*, 18(3):183 – 186, 1994.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Par90] Richard Parker. Finite fields and Conway polynomials, 1990. Talk given at IBM Scientific Center Heidelberg.

[PH91] Jean-François Pommaret and Akli Haddak. Effective methods for systems of algebraic partial differential equations. In T. Mora and C. Traverso, editors, *Effective Methods in Algebraic Geometry*, pages 411–426. Birkhäuser Boston, Boston, MA, 1991.

[Rad68] Charles M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, June 1968.

[Riq10] Charles Riquier. *Les systèmes d'équations aux dérivées partielles.* Gauthier-Villars, Paris, 1910.

[Rit32] Joseph F. Ritt. *Differential equations from the algebraic standpoint*, volume 14. American Mathematical Society, 1932.

[Rit50] Joseph F. Ritt. *Differential algebra*, volume 33. American Mathematical Society, 1950.

[Ros59] Azriel Rosenfeld. Specializations in differential algebra. *Transactions of the American Mathematical Society*, 90(3):394–407, 1959.

[Rou99] Fabrice Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Applicable Algebra in Engineering, Communication and Computing*, 9(5):433–461, May 1999.

[RSR69] Lawrence R. Rabiner, Ronald W. Schafer, and Charles M. Rader. The chirp z-transform algorithm. *IEEE Transactions on Audio and Electroacoustics*, 17(2):86–92, June 1969.

[Sag17] The Sage Developers. SageMath, the Sage Mathematics Software System (version 8.0), 2017. https://www.sagemath.org.

[Sch77] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Infor.*, 7:395–398, 1977.

[Sch92] Alfred Scheerhorn. Trace- and norm-compatible extensions of finite fields. *Applicable Algebra in Engineering, Communication and Computing*, 3(3):199–209, September 1992.

[Sch03] Éric Schost. Complexity results for triangular sets. *Journal of Symbolic Computation*, 36(3):555 – 594, 2003. ISSAC 2002.

[Sho01] Victor Shoup. NTL: A library for doing number theory, 2001. http://www.shoup.net/ntl/.

[Sie72] Malte Sieveking. An algorithm for division of powerseries. *Computing*, 10(1):153–156, March 1972.

[SJHB87] Henrik V. Sorensen, Douglas L. Jones, Michael T. Heideman, and C. Sydney Burrus. Real-valued fast Fourier transform algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(6):849–863, 1987.

[Sma81] Steve Smale. The fundamental theorem of algebra and complexity theory. *Bulletin (New Series) of the American Mathematical Society*, 4(1):1–36, 01 1981.

[Sma86] Steve Smale. Algorithms for solving equations. In *Proceedings of the International Congress of Mathematicians*, pages 172 – 195, 1986.

[SS71] Arnold Schönhage and Volker Strassen. Fast multiplication of large numbers. *Computing*, 7(3):281–292, 1971.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[Str73] Volker Strassen. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numerische Mathematik*, 20(3):238–251, June 1973.

[Ten73] Lynn F. Ten Eyck. Crystallographic Fast Fourier Transform. *Acta Crystallographica Section A*, A29:183–191, 1973.

[Tho37] Joseph Miller Thomas. *Differential systems*, volume 21. American Mathematical Society, 1937.

[Tho63] Llewellyn H. Thomas. Using a computer to solve problems in physics. *Applications of digital computers*, pages 44–45, 1963.

[Ver99] Jan Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Transactions on Mathematical Softwware*, 25(2):251–276, June 1999.

[Vil18] Gilles Villard. On computing the resultant of generic bivariate polynomials. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 391–398, New York, NY, USA, 2018. ACM.

[VZ07] Andrew Vince and Xiqiang Zheng. Computing the Discrete Fourier Transform on a hexagonal lattice. *Journal of Mathematical Imaging and Vision*, 28(2):125–133, June 2007.

[Wan93] Dongming Wang. An elimination method for polynomial systems. *Journal of Symbolic Computation*, 16(2):83 – 114, 1993.

[War12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2nd edition, 2012.

[Wu78] Wen-Tsün Wu. On the decision problem and the mechanization of theorem-proving in elementary geometry. *Scientia Sinica*, 21(2):159–172, 1978.

[Wu87] Wen-Tsün Wu. A zero structure theorem for polynomial equations-solving. *Mathematics and Systems Science*, 1:2–12, 1987.

[WZ88] Yao Wang and Xuelong Zhu. A fast algorithm for the Fourier transform over finite fields and its VLSI implementation. *IEEE Journal on Selected Areas in Communications*, 6(3):572–577, 1988.

[ZB96] Alexey Yu. Zharkov and Yuri A. Blinkov. Involution approach to investigating polynomial systems. *Math. Comput. Simul.*, 42(4-6):323–332, January 1996.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Arithmétique rapide pour des corps finis

Mots clés : Corps finis, Algorithme, Arithmétique polynomiale, FFT, Bases de Gröbner

Résumé : La multiplication de polynômes est une opération fondamentale en théorie de la complexité. En effet, pour de nombreux problèmes d'arithmétique, la complexité des algorithmes s'exprime habituellement en fonction de la complexité de la multiplication. Un meilleur algorithme de multiplication permet ainsi d'effectuer les opérations concernées plus rapidement. Un résultat de 2016 a établi une meilleure complexité asymptotique pour la multiplication de polynômes dans des corps finis. Cet article constitue le point de départ de la thèse ; l'objectif est d'étudier les conséquences à la fois théoriques et pratiques de la nouvelle borne de complexité.
La première partie s'intéresse à la multiplication de polynômes à une variable. Cette partie présente deux nouveaux algorithmes censés accélérer le calcul en pratique (plutôt que d'un point de vue asymptotique). S'il est difficile dans le cas général d'observer l'amélioration prévue, certains cas précis sont particulièrement favorables. En l'occurrence, le second algorithme proposé, spécifique aux corps finis, conduit à une meilleure implémentation de la multiplication dans $\mathbb{F}_2[X]$, environ deux fois plus rapide que les logiciels précédents.
La deuxième partie traite l'arithmétique des polynômes à plusieurs variables modulo un idéal, telle qu'utilisée par exemple pour la résolution de systèmes polynomiaux. Ce travail suppose une situation simplifiée, avec seulement deux variables et sous certaines hypothèses de régularité. Dans ce cas particulier, la deuxième partie de la thèse donne des algorithmes de complexité asymptotiquement optimale (à des facteurs logarithmiques près), par rapport à la taille des entrées/sorties. L'implémentation pour ce cas spécifique est alors nettement plus rapide que les logiciels généralistes, le gain étant de plus en plus marqué lorsque la taille de l'entrée augmente.

Title : Fast finite fields arithmetic

Keywords : Finite fields, Algorithm, Polynomial arithmetic, FFT, Gröbner bases

Abstract : The multiplication of polynomials is a fundamental operation in complexity theory. Indeed, for many arithmetic problems, the complexity of algorithms is expressed in terms of the complexity of polynomial multiplication. For example, the complexity of Euclidean division or of multi-point evaluation/interpolation (and others) is often expressed in terms of the complexity of polynomial multiplication. This shows that a better multiplication algorithm allows to perform the corresponding operations faster. A 2016 result gave an improved asymptotic complexity for the multiplication of polynomials over finite fields. This article is the starting point of the thesis; the present work aims to study the implications of the new complexity bound, from a theoretical and practical point of view.
The first part focuses on the multiplication of univariate polynomials. This part presents two new algorithms that should make the computation faster in practice (rather than asymptotically speaking). While it is difficult in general to observe the predicted speed-up, some specific cases are particularly favorable. Actually, the second proposed algorithm, which is specific to finite fields, leads to a better implementation for the multiplication in $\mathbb{F}_2[X]$, about twice as fast as state-of-the-art software.
The second part deals with the arithmetic of multivariate polynomials modulo an ideal, as considered typically for polynomial system solving. This work assumes a simplified situation, with only two variables and under certain regularity assumptions. In this particular case, there are algorithms whose complexity is asymptotically optimal (up to logarithmic factors), with respect to input/output size. The implementation for this specific case is then significantly faster than general-purpose software, the speed-up becoming larger and larger when the input size increases.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France