Fast polynomial computations with space constraints

Bruno Grenet

Abstract / Résumé

The works presented in this habilitation concern the algorithmics of polynomials. This is a central topic in computer algebra, with numerous applications both within and outside the field—cryptography, error-correcting codes, etc. For many problems, extremely efficient algorithms have been developed since the 1960s. Here, we are interested in how this efficiency is affected when space constraints are introduced.

The first part focuses on the time-space complexity of fundamental polynomial computations—multiplication, division, interpolation, … While *naive* algorithms typically have constant space complexity, fast algorithms generally require linear space. We develop algorithms that are both time- and space-efficient. This leads us to discuss and refine definitions of space complexity for function computation.

In the second part, the space constraints are put on the inputs and outputs. Algorithms for polynomials assume in general a *dense* representation for the polynomials, that is storing the full list of coefficients. In contrast, we work with *sparse* polynomials, in which most coefficients vanish. In particular, we describe the first quasi-linear algorithm for *sparse interpolation*, which plays a role analogous to the *Fast Fourier Transform* in the sparse settings. We also explore computationally hard problems concerning divisibility and factorization of sparse polynomials.

Les travaux présentés dans cette habilitation concernent l'algorithmique des polynômes. Il s'agit d'un sujet central en calcul formel, avec de nombreuses applications tant à l'intérieur qu'à l'extérieur du domaine — cryptographie, codes correcteurs, etc. Pour de nombreux problèmes, des algorithmes extrêmement efficaces ont été développés depuis les années 1960. Nous nous intéressons ici à la manière dont cette efficacité est influencée par l'introduction de contraintes d'espace.

La première partie se concentre sur la complexité temps–espace de calculs polynomiaux fondamentaux — multiplication, division, interpolation, … Alors que les algorithmes naïfs ont en général une complexité en espace constante, les algorithmes rapides nécessitent généralement un espace linéaire. Nous développons des algorithmes qui sont efficaces à la fois en temps et en espace. Cela nous amène à discuter et affiner les définitions de la complexité en espace pour le calcul de fonctions.

Dans la seconde partie, les contraintes de mémoire portent sur les données d'entrée et de sortie. Les algorithmes sur les polynômes supposent en général une représentation dense, c'est-à-dire le stockage de l'ensemble de leurs coefficients. À l'inverse, nous travaillons avec des polynômes creux, dont la plupart des coefficients sont nuls. En particulier, nous décrivons le premier algorithme quasi-linéaire pour l'interpolation creuse, qui joue un rôle analogue à celui de la transformée de Fourier rapide dans le contexte creux. Nous explorons également des problèmes algorithmiquement difficiles concernant la divisibilité et la factorisation des polynômes creux.

Acknowledgements

First of all, I would like to warmly thank all the members of my committee. I am truly honored to present my work in front of colleagues I deeply admire — not only for their scientific contributions, but also for their generosity and kindness toward others, myself included. I am especially grateful to Meena, Mark, and Emmanuel for the considerable effort they devoted to reviewing this manuscript.

Les travaux présentés dans ce document sont pour la plupart le fruit d'efforts collectifs. Merci à mes co-autrices et co-auteurs avec qui j'ai pris plaisir à travailler, et pour tout ce qu'ils et elles m'ont appris. Mon principal co-auteur, Pascal, a joué un rôle déterminant depuis mon recrutement comme maître de conférences. À mon arrivée à Montpellier, je me suis fortement investi dans l'enseignement. Redémarrer une activité de recherche, indépendante de mes encadrants de thèse et de post-docs, dans une thématique – le calcul formel – encore assez nouvelle pour moi à ce moment là, n'a pas été facile. Pascal m'a proposé des pistes à explorer ensemble, avec son enthousiasme habituel. Mon habilitation lui doit beaucoup.

Le calcul formel n'était pas mon domaine initial. Mes encadrants de post-doc, Reynald puis Grégoire et Joris, m'ont permis d'y faire mes premiers pas, merci à eux ! Merci également aux étudiantes et étudiants du cours de M2 *Calcul formel, codes et cryptographie* de l'Université de Montpellier : on dit la meilleure manière d'apprendre un sujet est de l'enseigner, et vous m'y avez largement aidé. Enfin, les échanges avec la communauté du calcul formel, notamment lors des Journées Nationales de Calcul Formel, ont enrichi ma culture du domaine. Je suis très heureux d'en faire partie !

J'ai eu la chance d'effectuer mes recherches dans des équipes dans lesquelles je me sentais à ma place. Je remercie vivement les membres de l'équipe ECO du LIRMM qui m'ont très bien accueilli à mon arrivée, et qui, je l'espère, ne m'en ont pas voulu lors de mon départ[1]. Travailler au sein d'un groupe d'amis a été un vrai plaisir. Je n'oublie pas les nombreux collègues du LIRMM et du département informatique de l'Université de Montpellier avec qui j'ai aimé collaborer, certains devenant des amis. Je veux aussi avoir une pensée particulière pour Nico.

J'ai retrouvé dans l'équipe CAS³C³ à laquelle j'appartiens aujourd'hui la même ambiance chaleureuse qui est très motivante au quotidien. La composition actuelle est assez différente de celle à mon arrivée, alors je remercie aussi bien les collègues qui sont présents depuis mon arrivée, que celles et ceux qui sont partis ou arrivés entre temps. À Grenoble non plus, mon environnement de travail ne se limite pas à mon équipe, et j'ai apprécié l'accueil que j'ai reçu au sein du LJK, de l'IM²AG et plus généralement à l'UGA.

En débordant un peu du seul cadre professionnel, merci à Agathe, Aurore et bien sûr Laurie pour tout ce qu'elles m'apportent !

[1] En tout cas vous ne me l'avez pas dit, ce que j'apprécie !

Contents

List of algorithms

Introduction

Computer algebra is the study of algorithms that manipulate exactly representable mathematical objects, mostly of an algebraic nature.[2] This includes computing with integers, rational numbers, and finite fields, as well as polynomials and matrices over these rings. It is part of the larger field of *computational mathematics*. It builds on *computer arithmetic* which is primarily concerned with computing with integers, and is a close cousin of *numerical computation* which focuses on computing with approximations of real or complex numbers. Computer algebra extends from (algebraic) complexity theory on the theoretical side to implementations on the practical side,[3] with strong foundations in various domains of mathematics such as algebra and number theory.

With a touch of bad faith, one could argue that computer algebra is actually the origin of the field of algorithmics. The word *algorithm* comes from the name of the Persian mathematician Muḥammad ibn Mūsā al-Khwārizmī (780–850), while the word *algebra* comes from the title of his book, *Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala*,[4] in which he introduces some of the earliest algorithms for solving quadratic equations. In fact, other algorithms describing algebraic computations were described earlier. One example is Euclid's algorithm, which Knuth refers to as the "granddaddy of all algorithms" [141].

Figure 0.1. Pages from al-Khwārizmī's book containing geometrical solutions to two quadratic equations (public domain, *via* Wikimedia commons[5]).

[2] The names *symbolic computation* or *algebraic computing* are almost synonymous.

[3] Widely used *computer algebra systems* are available, such as SageMath or commercial alternatives.

[4] *The Concise Book of Calculation by Restoration and Balancing*. The word *al-jabr*, meaning *balancing*, became the modern *algebra*.

[5] https://commons.wikimedia.org/wiki/File:Bodleian_MS._Huntington_214_roll332_frame36.jpg.

While computer algebra extends beyond polynomial computation, it undoubtedly represents one of its great successes. Very fast algorithms have been developed for basic polynomial operations such as multiplication, Euclidean division and GCD computation, as well as for more complex problems such as factorization and polynomial system solving. Implementations of these algorithms in computer algebra systems or more specialized libraries are used every day to solve practical problems. Polynomial computations lie at the heart of some related fields such as cryptography or error-correcting codes, but they are also essential in seemingly unrelated fields such as robotics.

Although the development of even-faster algorithms and implementations for polynomial computations is still an active area of research, we take a step to the side and introduce space constraints to these fast algorithms. These constraints fall into two categories. In Part I, we impose some restrictions on the memory usage of the algorithms. Indeed, faster algorithms were obtained at the cost of increased memory usage. We investigate the extent to which we can achieve both time- and space-efficiency with a single algorithm. In Part II, the restrictions concern the inputs and outputs of the algorithms. Our focus is on sparse polynomials, which have few terms compared to their degree. Traditional algorithms usually do not exploit the structure of these polynomials that benefit from a very compact representation. Consequently, fast algorithms appear slow with respect to the compact input size. Our goal is to develop new fast algorithms that take full account of the sparsity of the inputs and outputs.

The works presented in this document represent most of the research I conducted since my PhD thesis. The original publications contain more details, and are referred to throughout the text.[6] Two publications are not presented in the document: one about a new analysis of Euclid's algorithm [C4], and a very recent one in the field of cryptography [C13].

[6]The publications I (co-)authored have labels [C$\langle n \rangle$] (for conference publications), [J$\langle n \rangle$] (for journal publications), [M$\langle n \rangle$] (for unpublished manuscripts) or [S$\langle n \rangle$] (for software). Other references have digit-only labels.

1 Preliminaries

In this chapter, we introduce the necessary tools from classical computer algebra. Section 1.1 fixes our (sometimes unusual) notation. Subsequent sections introduce very classical results in computer algebra that can be found in standard textbooks [72, 76, 173]. They can be safely skipped by most readers.

1.1 Notations

1.1.1 Algebra

Let $\mathbb{Z}$ be the ring of integers, $\mathbb{Q}$ be the field of rational numbers, $\mathbb{R}$ the field of real numbers, $\mathbb{C}$ the field of complex numbers, and $\mathbb{F}_q$ be the finite field with q elements for a prime power q. We denote an abstract (commutative) ring (with identity) by the letter R. Its group of units is denoted $\mathsf{R}^\times$. When it makes sense, we use the notation $\mathsf{R}_{>0}$ (resp. $\mathsf{R}_{\geq 0}$) to denote the positive (resp. nonnegative) elements of R. We denote by $\mathsf{R}[x]$ the ring of (univariate) polynomials over R, by $\mathsf{R}[x]_{<n}$ the set of polynomials of degree less than n and by $\mathsf{R}[x_1, \ldots, x_n]$ the ring of n-variate polynomials over R. The ring of power series over R is $\mathsf{R}[[x]]$.

Vectors are written in bold font, *e.g.* $\boldsymbol{v} \in \mathsf{R}^n$. Matrices are written in capital letter, *e.g.* $M \in \mathsf{R}^{m \times n}$. A polynomial $f \in \mathsf{R}[x]_{<n}$ can be identified with its vector of coefficients, written $\boldsymbol{f}$. Due to this identification, we use the unusual convention that vectors and matrices are 0-indexed. If $\boldsymbol{v} \in \mathsf{R}^n$, its entries are written either $\boldsymbol{v}_{[0]}, \ldots, \boldsymbol{v}_{[n-1]}$ or $v_0, \ldots, v_{n-1}$. For $0 \leq i < j < n$, $\boldsymbol{v}_{[i,j[}$ is the vector $(v_i, \ldots, v_{j-1})$. The entries of $M \in \mathsf{R}^{m \times n}$ are denoted $M_{[i,j]}$ or $M_{i,j}$ for $0 \leq i < m$ and $0 \leq j < n$. For a vector $\boldsymbol{v} \in \mathsf{R}^n$, $\boldsymbol{v}^{\leftarrow}$ denotes the vector defined by $\boldsymbol{v}^{\leftarrow}_{[i]} = \boldsymbol{v}_{[n-1-i]}$ for $0 \leq i < n$. This notation is extended to polynomials and $f^{\leftarrow}$ denotes the polynomial $x^{\deg(f)} f(1/x)$ whose vector of coefficients is $\boldsymbol{f}^{\leftarrow}$. The *size* of a polynomial is the size of its vector of coefficients, that is $1 + \deg(f)$

1.1.2 Complexity analyses

We use the Landau notation for asymptotic complexity analyses. Given two nondecreasing functions $f, g : \mathbb{R} \to \mathbb{R}_{\geq 0}$, we write $f = O(g)$ if there exists x_0 and c such that $f(x) \leq c \cdot g(x)$ for all $x \geq x_0$. We write $f = o(g)$ if for all $c > 0$, there exists x_0 such that $f(x) < c \cdot g(x)$ for all $x \geq x_0$. We also use the notations $f = \Omega(g)$ if $g = O(f)$, $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$, and $f = \omega(g)$ if $g = o(f)$. We extend the standard notation $O(\cdot)$ and write $f = \widetilde{O}(g)$ if there exists $k \in \mathbb{Z}_{>0}$ such that $f = O(g(\log g)^k)$. A function is said *quasi-linear in* x if $f = \widetilde{O}(x)$.

There are two very natural time complexity analyses in computer algebra. The *algebraic complexity* of an algorithm over some ring R is the number of ring operations performed by the algorithm. (Over a field K, we consider field operations, including inversions and

divisions.) The *bit complexity* of an algorithm is the number of bit operations it performs. It takes into account the cost of each ring (or field) operation.

We denote by $\mathsf{Z}(n)$ the (bit) cost of multiplying two n-bit integers. A recent celebrated result provides the bound $\mathsf{Z}(n) = O(n \log n)$ [90]. Computing a Euclidean division of a $2n$-bit integer by an n-bit integer has cost $O(\mathsf{Z}(n))$ and computing the GCD of two n-bit integers has cost $O(\mathsf{Z}(n) \log n)$ [36]. This implies that the field operations in $\mathbb{Q}$ have cost $O(\mathsf{Z}(n) \log n)$ if the numerators and denominators have at most n bits. For a prime finite field $\mathbb{F}_p$, addition and subtraction have cost $O(\log p)$, multiplication has cost $O(\mathsf{Z}(\log p)) = O(\log p \log\log p)$ and inversion and divisions have cost $O(\mathsf{Z}(\log p) \log\log p) = O(\log p \log^2 \log p)$. In a nonprime finite field $\mathbb{F}_q$, elements are represented by polynomials, and the cost of basic polynomial operations is presented in the next sections.

A *randomized algorithm* is an algorithm that makes some random choices during its execution. It is called a *Las Vegas algorithm* if its correctness does not depend on these random choices but its complexity does. It is called a *Monte Carlo algorithm* if its complexity does not depend on the random choices but its correctness does. Finally it is called an *Atlantic City algorithm* if both the complexity and the correctness depend on the random choices. Note that is is always possible to turn a Las Vegas algorithm or an Atlantic City algorithm into a Monte Carlo algorithm.

1.2 Polynomial products

Given two polynomials $f \in \mathsf{R}[x]_{<m}$ and $g \in \mathsf{R}[x]_{<n}$, their product $h = f \times g \in \mathsf{R}[x]_{<m+n-1}$ is defined by $h_k = \sum_{i+j=k} f_i g_j$ for $0 \le k < m + n - 1$.

Definition 1.2.1. A function $\mathsf{M}_\mathsf{R} : \mathbb{R}_{\ge 0} \to \mathbb{R}_{\ge 0}$ is a *polynomial multiplication time* for R if

- two polynomials $f, g \in \mathsf{R}[x]_{<n}$ can be multiplied in $\mathsf{M}_\mathsf{R}(n)$ operations in R, and
- the function $n \mapsto \mathsf{M}_\mathsf{R}(n)/n$ is nondecreasing.

As long as the context is clear, we shall drop the subscript R and write $\mathsf{M}(n)$ for $\mathsf{M}_\mathsf{R}(n)$. Over any ring R, the classical polynomial multiplication algorithm requires $O(n^2)$ operations in R. We shall therefore always assume that $\mathsf{M}(n)/n = O(n)$. Better algorithms are also known, either ring-agnostic or specialized. Over any ring, Karatsuba's algorithm [135] computes a polynomial product in $O(n^{\log 3})$ operations in R,[1] and its generalizations known as Toom-Cook's algorithms [53, 206] compute it in $O(n^{\log_{r+1}(2r+1)})$ operations in R for every $r > 0$. If R contains a $2n$th principal root of unity in R,[2] FFT-based polynomial multiplication uses $O(n \log n)$ operations in R [54, 77, 171]. In the general case, one can *create* such a root of unity, to get an algorithm that performs $O(n \log n \log\log n)$ operations in R [43]. For the important case of a finite field $\mathbb{F}_q$, the number of operations in $\mathbb{F}_q$ may be $O(n \log n)$ if an appropriate root of unity exists, and this translates into $O(n \log n \cdot \log(q) \log\log(q))$

[1] In the whole document, $\log(\cdot)$ denotes the base-2 logarithm, and $\log_b(\cdot)$ denotes the base-b logarithm.

[2] A *principal root of unity* of order n is an element $\omega \in \mathsf{R}$ such that $\omega^n = 1$, and $\omega^i - 1 \in \mathsf{R}^\times$ for $0 < i < n$. If R is an integral domain, principal roots of unity coincide with primitive roots of unity.

bit operations by means of fast integer multiplication [91]. This bound is not attained for all finite fields, and is anyway not the best achievable. Unconditionally, two polynomials of degree $< n$ can be multiplied in $O(n \log q \log(n \log q) 4^{\log^*(n \log q)})$ bit operations [89], where $\log^*$ denotes the iterated logarithm.[3] Under some number-theoretic assumptions, the complexity bound becomes $O(n \log q \log(n \log q))$ [91].

Polynomial multiplication is the basis of most algorithms on polynomials. In many cases, the full result is not required but rather only part of it. The standard product of polynomials is called the *full product*. We also define partial products.

Definition 1.2.2. Let $f, g \in \mathsf{R}[x]$ of respective sizes m and n.

- Their *lower product* is the polynomial made of the lower m coefficients of their product, written $\mathsf{LowProd}(f, g) = (f \times g) \bmod x^m$.
- Their *upper product* is the polynomial made of the upper $n-1$ coefficients of their product, written $\mathsf{UppProd}(f, g) = (f \times g) \operatorname{quo} x^m$.
- If $m \geq n$, their *middle product* is the polynomial made of the central $m-n+1$ coefficients of their product, written $\mathsf{MidProd}(f, g) = \left((f \times g) \operatorname{quo} x^{n-1}\right) \bmod x^{m-n+1}$.

Remark. As defined, the lower and upper products are not commutative. They satisfy $f \times g = \mathsf{LowProd}(f, g) + x^m \mathsf{UppProd}(f, g) = \mathsf{LowProd}(g, f) + x^n \mathsf{UppProd}(g, f)$.

We extend the notation $\mathsf{M}(\cdot)$ and denote by $\mathsf{M}(m, n)$ the cost of an algorithm that multiplies two polynomials $f \in \mathsf{R}[x]_{<m}$ and $g \in \mathsf{R}[x]_{<n}$. Assuming without loss of generality that $m \geq n$, the product can be performed using $\lceil m/n \rceil$ products of size-n polynomials, whence $\mathsf{M}(m, n) \leq \lceil m/n \rceil \mathsf{M}(n)$. The case $m = n$ is called a *balanced* (full) product. We also define the balanced lower and upper products when $m = n$, and the balanced middle product when $m = 2n - 1$ and the result has size n.

Proposition 1.2.3 ([88]). *Let $f, g \in \mathsf{R}[x]$ of respective sizes m and n. Then $\mathsf{LowProd}(f, g)$ can be computed in $\mathsf{M}(m)$ operations, and $\mathsf{UppProd}(f, g)$ in $\mathsf{M}(n-1)$ operations. The middle product $\mathsf{MidProd}(f, g)$ can be computed in $\mathsf{M}(m, n)$ operations.*

1.3 Other polynomial and power series computations

Let $\phi \in \mathsf{R}[[x]]$ be some power series. In computations, it is represented by a *truncation at precision n*, that is its value modulo x^n for some n. If $f = \phi \bmod x^n$, f is said to be a *truncated power series at precision n*. A truncated power series at precision n is thus a size-n polynomial.

If $\phi, \psi \in \mathsf{R}[[x]]$ and $f = \phi \bmod x^n$, $g = \psi \bmod x^n$, the truncation at precision n of $\phi \times \psi$ is $\mathsf{LowProd}(f, g)$.

Proposition 1.3.1. *Given the truncations at precision n of two power series $\phi, \psi \in \mathsf{R}[[x]]$, the truncation at precision n of $\phi \times \psi$ can be computed in $\mathsf{M}(n)$ operations*

[3] It is defined by $\log^*(x) = 0$ if $x \leq 1$ and $\log^*(x) = 1 + \log^*(\log x)$ otherwise.

in R.

If $\phi(0)$ is a unit of R, the series ϕ is invertible, that is there exists $\phi^{-1} \in \mathsf{R}[[x]]$ such that $\phi \times \phi^{-1} = 1 \in \mathsf{R}[[x]]$. Given a truncation of ϕ at precision n, the truncation of ϕ^{-1} at precision n can be computed by means of Newton iteration.

Proposition 1.3.2 ([146, 202]). *Given the truncation at precision n of a power series $\phi \in \mathsf{R}[[x]]$ whose constant coefficient is a unit, the truncation at precision n of ϕ^{-1} can be computed in $O(\mathsf{M}(n))$ operations in* R.

From this, one can also compute a division of power series.

Corollary 1.3.3. *Given the truncations at precision n of two power series $\phi, \psi \in \mathsf{R}[[x]]$, where $\psi(0)$ is a unit, the truncation at precision n of ϕ/ψ can be computed in $O(\mathsf{M}(n))$ operations in* R.

Power series inversion and division are the basis for fast Euclidean division of polynomials. Indeed, the quotient $q = f \operatorname{quo} g$ can be computed as a *reversed power series division* (see Section 1.7 and Figure 3.4 in Chapter 3), and the remainder $r = f \bmod g$ is obtained as $f - gq$.

Proposition 1.3.4 ([203]). *Given two polynomials f and $g \in \mathsf{R}[x]$ of respective sizes $m+n-1$ and n such that the leading coefficient of g is a unit, the quotient $q = f \operatorname{quo} g$ and the remainder $r = f \bmod g$ such that $f = bq + r$ and $\deg(r) < \deg(g)$ can be computed in $O(\mathsf{M}(m) + \mathsf{M}(n))$ operations in* R.

Given a size-n polynomial $f \in \mathsf{R}[x]$ and n points $\alpha_0, \ldots, \alpha_{n-1} \in \mathsf{R}$, the problem of *multipoint evaluation* is to compute $f(\alpha_0), \ldots, f(\alpha_{n-1})$. Interpolation does the converse. Given $\alpha_0, \ldots, \alpha_{n-1} \in \mathsf{R}$ such that $\alpha_i - \alpha_j$ is a unit for all $i \neq j$, and $\beta_0, \ldots, \beta_{n-1}$, the goal is to compute the unique size-n polynomial $f \in \mathsf{R}[x]$ such that $f(\alpha_i) = \beta_i$ for $0 \leq i < n$.

Proposition 1.3.5 ([29]). *Given a size-n polynomial $f \in \mathsf{R}$ and a vector $\boldsymbol{\alpha} \in \mathsf{R}^n$, the vector $(f(\alpha_0), \ldots, f(\alpha_{n-1}))$ can be computed in $O(\mathsf{M}(n)\log n)$ operations in* R. *Given two vectors $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathsf{R}^n$ such that $\alpha_i - \alpha_j$ is a unit for $i \neq j$, the unique size-n polynomial f such that $f(\alpha_i) = \beta_i$ for $0 \leq i < n$ can be computed in $O(\mathsf{M}(n)\log n)$ operations in* R.

Note that it makes also sense to define multipoint evaluation for a size-m polynomial and n points when $m \neq n$. If $m < n$, one can perform $\lceil n/m \rceil$ multipoint evaluations in size m, in $O(\frac{n}{m}\mathsf{M}(m)\log m) = O(\mathsf{M}(n)\log m)$ operations in R. If $m < n$, f is first reduced modulo $\prod_i (x - \alpha_i)$ (which is to be computed anyway by the algorithm) and multipoint evaluation is applied to the size-n remainder, for a total of $O(\mathsf{M}(m-n) + \mathsf{M}(n)\log n)$ operations in R. When the vector $\boldsymbol{\alpha}$ has some structure, the computation can usually be sped up [33]. In particular if it is a geometric progression, the complexities drop to $O(\mathsf{M}(n))$ [28, 33].

Given two monic polynomials f and g, their greatest common divisor (GCD) can be computed with a fast variant of Euclid's algorithm. This also provides Bézout coefficients u and v such that $uf + vg = \gcd(f, g)$.

Proposition 1.3.6 ([35]). *Given two monic polynomials $f, g \in \mathsf{R}[x]$ of respective size m and n where $m \geq n$, their GCD and the corresponding Bézout coefficients can be computed in $O(\mathsf{M}(m-n) + \mathsf{M}(n)\log n)$ operations in* R.

Remark. In all the previous complexities, the logarithmic factors disappear in the terms $\mathsf{M}(n)\log(n)$ as soon as $\mathsf{M}(n) = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$. This is the case if the underlying multiplication algorithm is for instance the naive one ($\mathsf{M}(n) = O(n^2)$) or Karatsuba's ($O(n^{\log 3})$).

In practice, the constants of the leading terms in the complexities play a major role for the efficiency of the algorithms. For better constants than the original algorithms, we refer to more recent works [32, 33, 88, 97].

Equipped with these results, we obtain the bit costs of the field operations in a finite field $\mathbb{F}_q$ where $q = p^s$ for some prime number p. Since each element of $\mathbb{F}_q$ is represented by a polynomial in $\mathbb{F}_p[x]_{<s}$, addition and subtraction use s operations in $\mathbb{F}_p$, that is $O(s\mathsf{Z}(\log p))$ bit operations. Multiplication can be computed in $O(\mathsf{M}_{\mathbb{F}_p}(s))$ operations in $\mathbb{F}_p$, that is $\widetilde{O}(s\log p)$ bit operations. Under some number-theoretic assumptions, the complexity becomes $O(s\log p\log(s\log p))$ [91]. Inversion and division are computed in $O(\mathsf{M}_{\mathbb{F}_p}(s)\log s)$ operations in $\mathbb{F}_p$, that is $\widetilde{O}(s\log p)$ bit operations.

1.4 Linear recurrent sequences

Linear recurrent sequences, also known as *constant-recursive* or *C-finite sequences*, are a very useful tool in computer algebra. For a sequence $(u_n)_{n\geq 0} \in \mathsf{R}^{\mathbb{N}}$, its *generating series* is the power series $\phi = \sum_{n\geq 0} u_n x^n$.

Definition 1.4.1. A sequence $(u_n)_{n\geq 0} \in \mathsf{R}^{\mathbb{N}}$ is *linear recurrent* if there exist $a_0, \ldots, a_{k-1}$ such that for all $n \geq 0$,

$$u_{n+k} = \sum_{j=0}^{k-1} a_j \cdot u_{n+j}.$$

The polynomial $p = x^k - \sum_{j=0} a_j x^j$ is a *characteristic polynomial* of $(u_n)_n$. The *minimal polynomial* of $(u_n)_n$ is its least-degree characteristic polynomial. The *order* of $(u_n)_n$ is the degree of its minimal polynomial.

The following proposition states some fundamental equivalent characterizations of a linear recurrent sequence. We state it only in the simple situation where the minimal polynomial splits and is square-free[4] over R. We make the assumption in this document that the minimal polynomials of all the linear recurrent sequences split and are square-free in R. Recall that the polynomial $p^{\leftarrow}$ is defined as $x^{\deg(p)}p(1/x)$.

[4] A degree-d polynomial *splits* over R if it has d roots in R, counting with multiplicities. It is square-free if its roots are pairwise distinct.

Proposition 1.4.2. *Let $(u_n)_{n\geq 0} \in \mathsf{R}^{\mathbb{N}}$, and $\phi = \sum_{n\geq 0} u_n x^n$ its generating function. The following statements are equivalent:*

(i) $(u_n)_{n\geq 0}$ *is linear recurrent with minimal polynomial* $p = \prod_{i=0}^{k-1}(x - \rho_i)$ *with pairwise distinct roots* $\rho_0, \ldots, \rho_{k-1} \in \mathsf{R}$;

(ii) $\phi = q/p^{\leftarrow}$ *for some polynomial* $q \in \mathsf{R}[x]$ *of degree* $< k$;

(iii) $u_n = \sum_{i=0}^{k-1} \lambda_i \rho_i^n$ *for some* $\lambda_0, \ldots, \lambda_{k-1}$ *that only depend on the initial conditions* $u_0, \ldots, u_{k-1}$.

Similar and further equivalences can be given in more general settings [61]. An algorithmic view of this proposition gives different representations for a linear recurrent sequence.

Proposition 1.4.3. *A linear recurrence sequence $(u_n)_{n\geq 0}$ of order at most k is completely determined by any of the following data:*

(i) $2k$ *initial terms* $u_0, \ldots, u_{2k-1}$;

(ii) its minimal polynomial p, *together with the* k *initial terms* $u_0, \ldots, u_{k-1}$;

(iii) a rational representation $q/p^{\leftarrow}$ *of its generating series* ϕ;

(iv) the vectors $\boldsymbol{\lambda}$ *and* $\boldsymbol{\rho}$ *such that* $u_n = \sum_{i=0}^{k-1} \lambda_i \rho_i^n$ *for* $n \geq 0$.

Note that in all cases, the representation consists of $2k$ elements from R. Conversions between these different representations are known under various names in the literature, and associated to some classical algorithms. These are represented in Figure 1.1.

Both representations (ii) and (iii) contain the minimal polynomial p. The conversion between $2k$ initial terms of $(u_n)_n$ and each of these two representations give rise to two families of algorithms. In information theory, the conversion (i) $\to$ (ii) is known as *LFSR synthesis*.[5] It is computed using Berlekamp-Massey's algorithm [19, 21, 159] in quadratic time and serves as building block for decoding algorithms of some error-correcting codes related to BCH codes. A fast variant of this algorithm has complexity $O(\mathsf{M}(k)\log k)$. The other direction is known as *recurrence extension* and can be computed as a *transposed* Euclidean division [197].

Originating in complex analysis, a Padé approximant is a rational function approximation of a power series. The conversion (i) $\to$ (iii) is exactly a Padé approximant computation, viewing the $2k$ initial terms as the truncated power series $\phi \bmod x^{2k}$. The extended Euclidean algorithm can be used to compute it in quadratic time. Using fast Euclidean algorithm (Proposition 1.3.6), the complexity drops to $O(\mathsf{M}(k)\log k)$. The other direction is *Taylor expansion* and can be computed using Euclidean division.

It has been noticed that Berlekamp-Massey's algorithm and the Euclidean algorithm are in a sense dual of each other [58]. Using any of the two fast variants provides the following result.

[5] LFSR stands for Linear-Feedback Shift Register.

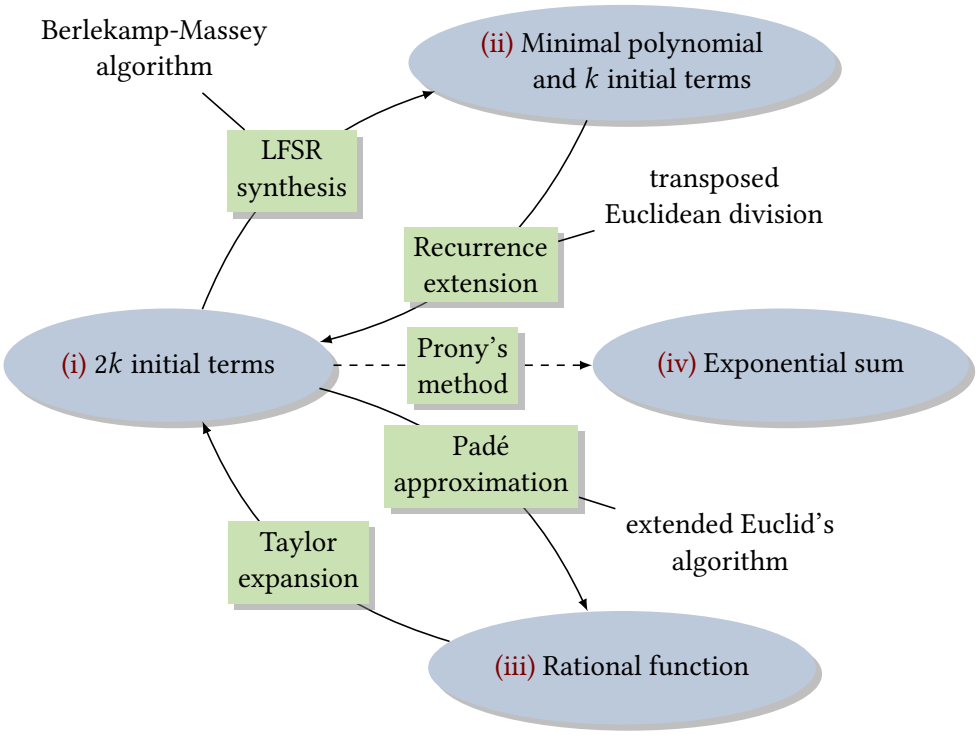


Figure 1.1. Conversion between representations of linear recurrent sequences

Proposition 1.4.4. *Given $u_0, \ldots, u_{2k-1} \in \mathsf{R}$, the minimal polynomial of $(u_i)_{0 \le i < 2k}$ can be computed with $O(\mathsf{M}(k) \log k)$ operations in* R.

Finally, the conversion (i) $\to$ (iv) is known as *Prony's method* in numerical analysis, and the conversion is made by first computing the minimal polynomial, that is using either the conversion (i) $\to$ (ii) or (i) $\to$ (iii) first. More details on this method are given in Section 7.1.

1.5 Polynomial operations as structured linear algebra

Most polynomial and power series computations presented so far are bilinear maps. Therefore, one can fix any of the inputs to get a linear map that has a matrix representation. The corresponding matrices have structures that we present now. Note that all these structures can be encompassed in the more general framework of *low displacement rank* [120]. Pan has written a thorough treatment of the links between polynomial computations and structured matrix computations [173].

Recall that a polynomial $f \in \mathsf{R}[x]_{<n}$ is identified with its vector of coefficients $\boldsymbol{f} \in \mathsf{R}^n$. Conversely, a vector $\boldsymbol{v} \in \mathsf{R}^n$ can be viewed as a polynomial $v \in \mathsf{R}[x]_{<n}$. The polynomial $f^{\leftarrow}$ is defined by $f^{\leftarrow}(x) = x^{\deg(f)} f(1/x)$ and its vector of coefficients $\boldsymbol{f}^{\leftarrow}$ is defined by $f_i^{\leftarrow} = f_{\deg(f)-i}$.

We first consider polynomial products. They correspond to Toeplitz matrix-vector products.

Definition 1.5.1. Let $\boldsymbol{\alpha} \in \mathsf{R}^{m+n-1}$. The *Toeplitz matrix* $T_{m,n}(\boldsymbol{\alpha})$ is the $m \times n$ matrix

$$T = \begin{pmatrix} \alpha_{m-1} & \alpha_m & \dots & \alpha_{m+n-2} \\ \alpha_{m-2} & \alpha_{m-1} & \dots & \alpha_{m+n-3} \\ \vdots & \vdots & & \vdots \\ \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \alpha_0 & \alpha_1 & \dots & \alpha_{n-1} \end{pmatrix}$$

defined by $T_{i,j} = \alpha_{m-1+j-i}$ for $0 \le i < m$ and $0 \le j < n$. If $m = n$, we denote the square matrix $T_{m,m}(\boldsymbol{\alpha})$ by $T_m(\boldsymbol{\alpha})$.
If $\boldsymbol{\alpha} \in \mathsf{R}^m$, the *lower* and *upper triangular Toeplitz matrices* are the $m \times m$ matrices $L_m(\boldsymbol{\alpha}) = T_m(\boldsymbol{\alpha} \| \mathbf{0}_{m-1})$ and $U_m(\boldsymbol{\alpha}) = T_m(\mathbf{0}_{m-1} \| \boldsymbol{\alpha})$ where $\mathbf{0}_{m-1} \in \mathsf{R}^{m-1}$ is the all-zero vector, and $\cdot \| \cdot$ denotes concatenation.

Proposition 1.5.2. *Let $f, g \in \mathsf{R}[x]$ of respective sizes m and n. Then*

- *$h = f \times g$ is equivalent to $\boldsymbol{h} = T_{m+n-2,n}(\mathbf{0}_{n-2} \| \boldsymbol{f}^{\leftarrow} \| \mathbf{0}_{n-1}) \cdot \boldsymbol{g}$;*
- *$h = \mathsf{MidProd}(f, g)$ is equivalent to $\boldsymbol{h} = T_{m,n}(\boldsymbol{f}^{\leftarrow}) \cdot \boldsymbol{g}$;*
- *$h = \mathsf{LowProd}(f, g)$ is equivalent to $\boldsymbol{h} = L_m(\boldsymbol{f}^{\leftarrow}) \cdot \boldsymbol{g}$;*
- *$h = \mathsf{UppProd}(f, g)$ is equivalent to $\boldsymbol{h} = U_{n-1}(\boldsymbol{f}^{\leftarrow}_{[m-n+1,m[}) \cdot \boldsymbol{g}$.*

As a result, Toeplitz matrix-vector products can be computed in $O(\mathsf{M}(m, n))$ operations in R. Lower triangular Toeplitz system solving corresponds to power series division and can be computed in $O(\mathsf{M}(n))$ operations in R.

A special case of Toeplitz matrix is the case of a circulant matrix.

Definition 1.5.3. Let $\boldsymbol{\alpha} \in \mathsf{R}^m$. The *circulant matrix* $C_m(\boldsymbol{\alpha})$ is the $m \times m$ matrix

$$C = \begin{pmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \dots & \alpha_{m-2} & \alpha_{m-1} \\ \alpha_{m-1} & \alpha_0 & \alpha_1 & \dots & \alpha_{m-3} & \alpha_{m-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_{m-1} & \alpha_0 \end{pmatrix}$$

defined by $C_{i,j} = \alpha_{(j-i) \bmod m}$ for $0 \le i, j < m$.
For $\lambda \in \mathsf{R}$, the *λ-circulant matrix* $C^{\lambda}_m(\boldsymbol{\alpha})$ is the $m \times m$ matrix

$$C^{\lambda} = \begin{pmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \dots & \alpha_{m-2} & \alpha_{m-1} \\ \lambda \cdot \alpha_{m-1} & \alpha_0 & \alpha_1 & \dots & \alpha_{m-3} & \alpha_{m-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ \lambda \cdot \alpha_1 & \lambda \cdot \alpha_2 & \lambda \cdot \alpha_3 & \dots & \lambda \cdot \alpha_{m-1} & \alpha_0 \end{pmatrix}$$

defined by $C^{\lambda}_{i,j} = \alpha_{j-i}$ for $0 \le i \le j < m$ and $C^{\lambda}_{i,j} = \lambda \cdot \alpha_{m-i+j}$ for $0 \le j < i < m$.

These matrices correspond to *polynomial convolutions*.

Proposition 1.5.4. *Let $f, g \in \mathsf{R}[x]_{<n}$ and $\lambda \in \mathsf{R}$. Then $h = f \times g \bmod x^n - \lambda$ is equivalent to $\boldsymbol{h} = C_n^\lambda(\boldsymbol{f})^\mathsf{T} \cdot \boldsymbol{g}$. Both problems can be computed in $\mathsf{M}(n)$ operations in R.*

We now turn to evaluation and interpolation, that correspond to Vandermonde matrices.

Definition 1.5.5. Let $\boldsymbol{\alpha} \in \mathsf{R}^m$. The *Vandermonde matrix* $V_{m,n}(\boldsymbol{\alpha})$ is the $m \times n$ matrix

$$V = \begin{pmatrix} 1 & \alpha_0 & \alpha_0^2 & \dots & \alpha_0^{n-1} \\ 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha_{m-2} & \alpha_{m-2}^2 & \dots & \alpha_{m-2}^{n-1} \\ 1 & \alpha_{m-1} & \alpha_{m-1}^2 & \dots & \alpha_{m-1}^{n-1} \end{pmatrix}$$

defined by $V_{i,j} = \alpha_i^j$ for $0 \le i < m$ and $0 \le j < n$. If $m = n$, we denote the square matrix $V_{m,m}(\boldsymbol{\alpha})$ by $V_m(\boldsymbol{\alpha})$.
If $\alpha \in R$, a special case of Vandermonde matrix is the *Fourier matrix*[6] $F_m(\alpha) = V_m(1, \alpha, \alpha^2, \dots, \alpha^{m-1})$.

Note that a Fourier matrix and its transpose have both the Vandermonde structure.

Proposition 1.5.6. *Let $f \in \mathsf{R}[x]$ of size n, $\boldsymbol{\alpha} \in \mathsf{R}^m$ and $\boldsymbol{e} = (f(\alpha_0), \dots, f(\alpha_{m-1})) \in \mathsf{R}^m$. Then $\boldsymbol{e} = V_{m,n}(\boldsymbol{\alpha}) \cdot \boldsymbol{f}$. Conversely, if $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathsf{R}^m$, the vector of coefficients of the unique size-m polynomial such that $f(\alpha_i) = \beta_i$, $0 \le i < m$, is $\boldsymbol{f} = V_m(\boldsymbol{\alpha})^{-1} \cdot \boldsymbol{\beta}$.*

A consequence of these equivalences is that (square) Vandermonde matrix-vector products and Vandermonde system solving can be computed in $O(\mathsf{M}(n) \log n)$ operations in R. In the case of a Fourier matrix, both can be computed with $O(\mathsf{M}(n))$ operations.

1.6 Straight-line programs and arithmetic circuits

Straight-line programs are a representation of polynomials by programs of evaluation that have neither loop nor test. They have been very successfully used in computer algebra for instance for polynomial factorization [123] or for polynomial system solving [83]. We shall need them as inputs of some sparse interpolation algorithm, *cf.* Chapter 7. And the transposition principle presented in Section 1.7 can be phrased with *linear* straight-line programs.

Definition 1.6.1. A *straight-line program* (SLP) $\mathcal{S}$ with n variables $x_1, \dots, x_n$ over some ring R is a list of ℓ instructions. The ith instruction, $0 \le i < \ell$, is of the form $r_i \leftarrow u \star v$ where r_i is a *register*, u (resp. v) is either a variable, a constant from R or a register r_j, $j < i$, and $\star \in \{+, -, \times\}$. Over a field K, an SLP *with divisions* allows divisions, that is $\star \in \{+, -, \times, /\}$.

[6] When α has order m, it is also known as the *DFT matrix*, for *discrete Fourier transform*.

The output of an SLP is a tuple $(r_{i_0}, \ldots, r_{i_m})$ of *output registers*. A single-output SLP has $r_{\ell-1}$ as unique output register.
The result of an SLP (resp. an SLP with divisions) on inputs $\alpha_1, \ldots, \alpha_n \in \mathsf{R}$ (resp. K) is the m values of its output registers when each variable x_j is replaced by the value α_j, and each instruction is executed in order by interpreting the operation $\star$ by the corresponding operation in R (resp. K).

A single-output straight-line program computes a polynomial function of its inputs (or a rational function if it allows divisions). It admits a graphical representation called an *arithmetic circuit*. It is formally a directed acyclic graph with one vertex per variable, one vertex per constant, and one vertex per instruction. The vertices associated to the variables and the constant have in-degree 0. The vertex associated to an instruction $r_i \leftarrow u \star v$ has two incoming arcs, one from the vertex associated to u and one from the vertex associated to v. An example is given in Figure 1.2.

Variable: x
Constant: 1

1 $r_0 \leftarrow x + 1$
2 $r_1 \leftarrow x - 1$
3 $r_2 \leftarrow r_0 \times r_1$

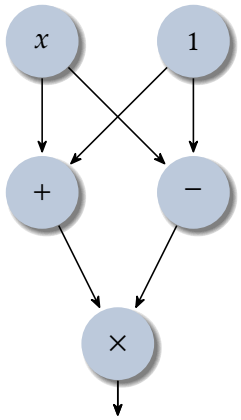


Figure 1.2. Straight-line program for the polynomial $x^2 - 1$ (left) and its arithmetic circuit representation (right).

1.7 Transposition principle and reversion

The *transposition principle* [62, 63, 134, 198] relates, for any matrix $M \in \mathsf{R}^{m \times n}$, the cost of computing $M \cdot v$ for $v \in \mathsf{R}^n$ and the cost of $M^\mathsf{T} \cdot w$ for $w \in \mathsf{R}^m$, or equivalently to compute $w \cdot M$. (Here and thereafter, we adopt the convention that a vector is understood as a column vector in a matrix-vector product while it is understood as a row vector in a vector-matrix product.) It can be formalized with the use of *directed acyclic graphs* or *linear programs* [32]. A linear program is a variant of a straight-line program, with only linear operations.

Definition 1.7.1. A *linear program* with n inputs over R is a straight-line program where each instruction is of the form $r_i \leftarrow \pm u \pm v$ or $r_i \leftarrow \lambda \cdot u$ where u (resp. v) is either a variable, a constant or a register r_j, $j < i$, and $\lambda \in \mathsf{R}$.

A linear program computes a linear mapping. A linear program for a matrix M is a linear program that, on input $v \in \mathsf{R}^n$, computes $w = M \cdot v$. The transposition principle is

a program transformation technique to obtain a linear program for the transposed of a matrix.

Proposition 1.7.2. *Any linear program of size ℓ for $M \in \mathsf{R}^{m\times n}$ can be turned into a linear program of size $\ell + m - n$ for the transpose matrix M^{T}.*

Examples of transposition are very common in polynomial computation. For instance, the transposed version of (full) polynomial multiplication is the middle product. The lower and upper products are (almost[7]) transposed of each other. Some of the fastest algorithms have been found by means of the transposition principle, first designing a fast algorithm for the transposed problem [32, 33].

We shall use another program transformation technique, coined *reversion*. Given a linear program for a matrix $M \in \mathsf{R}^{m\times n}$, we can obtain a same size linear program for the matrix $M^{\leftarrow} \in \mathsf{R}^{m\times n}$ defined by $M^{\leftarrow}_{i,j} = M_{m-1-i,n-1-j}$. The computation $\boldsymbol{w} \leftarrow M^{\leftarrow} \cdot \boldsymbol{v}$ is equivalent to $\boldsymbol{w}^{\leftarrow} \leftarrow M \cdot \boldsymbol{v}^{\leftarrow}$. The transformation is extremely simple: reverse the order of the input registers (r_{-n+1} becomes r_0, r_{-n+2} becomes r_{-1}, etc.) and of the output registers (the output tuple becomes $(i_{m-1}, \ldots, i_0)$). With this transformation, we can for instance define the *reversed power series division* that takes as inputs $f = \phi \bmod x^n$ and $g = \psi \bmod x^n$ such that the leading coefficient of g is a unit, and returns h such that $h^{\leftarrow} = f^{\leftarrow}/g^{\leftarrow} \bmod x^n$. This is a basic operation in the fast Euclidean algorithm, *cf.* Figure 3.4 on page 35.

Proposition 1.7.3. *Any linear program of size ℓ for $M \in \mathsf{R}^{m\times n}$ can be turned into a linear program of the same size for the reversed matrix $M^{\leftarrow}$.*

Remark. Propositions 1.7.2 and 1.7.3 look superficially very similar. Yet the transposition principle requires a proof, even if not very involved, and some questions about it remain open [122]. The reversion on the other hand is the simple remark that one can reverse the order of the indices in a program that manipulates vectors or arrays.

[7] In our definition, the matrix of the lower product has a nonzero diagonal, contrary to the matrix of the upper product.

I Time- and space-efficient polynomial computations

Summary

As presented in Chapter 1, the complexity of many polynomial operations has been reduced from quadratic for the classical algorithms to subquadratic and even quasi-linear time. This is due to faster algorithms for polynomial multiplication on the one hand, and algorithmic reductions from many operations to multiplication with the smallest possible overhead on the other hand.

Nevertheless, the price to pay for these faster algorithms is an increase in the space complexity. The quadratic polynomial multiplication algorithm is easily seen to require no extra space. Other quadratic algorithms can be implemented without extra space as well [160]. But fast multiplication algorithms or other fast polynomial algorithms require at least a linear amount of space, and sometimes even more. As a simple example, consider Karatsuba's multiplication algorithm [135]. To compute $f \times g$, the two polynomials are written as $f_0 + x^m f_1$ and $g_0 + x^m g_1$, and their product is written $f_0 g_0 + x^m((f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1) + x^{2m} f_1 g_1$. Using three half-sized recursive calls decreases the complexity of $O(n^{\log 3})$. But in terms of space complexity, a linear amount of space is required to store the intermediate results $f_0 + f_1$, $g_0 + g_1$, $f_0 g_0$, $f_1 g_1$ and $(f_0 + f_1)(g_0 + g_1)$.

The goal of this part is to present some simultaneously fast and constant-space algorithms for many basic polynomial operations. In Chapter 2, we first discuss the models of computation. There are some subtleties to properly define time-space complexity classes for functions, and we argue that the standard complexity classes are not very well suited for this case. We also exhibit the links with traditional complexity theory, and in particular explain that our *constant space* more or less corresponds to the traditional *logarithmic space*. Chapters 3 and 4 present two series of incomparable results. The first one is in a more restricted model. The second one uses a more permissive model but provides stronger results, focusing on *cumulative* operations such as $c \mathrel{+}= a \times b$. Finally, Chapter 5 presents a partial automatization of the results of Chapter 4, that is algorithms to produce constant-space variants of standard algorithms with the same asymptotic running times. We also apply these techniques to linear-algebraic problems.

This is based on a series of works with Pascal Giorgi (U. Montpellier) and Daniel S. Roche (U.S. Naval Academy) [C3, C6] and with Jean-Guillaume Dumas (U. Grenoble Alpes) [C10, C11, M2].

Notations and conventions

In this part, we consider polynomials over an abstract ring R. We assume that it is an integral domain, although several results still hold in more general settings.

The coefficient of degree i of a polynomial f is denoted $f_{[i]}$. We keep non-bracketed subscripts such as f_0 or f_1 to denote parts of the polynomial f. The size-$(j-i)$ polynomial made of terms of degree i to $j-1$ is either denoted $f_{[i,j[}$ or $[f]_i^j$. Using the more mathematical

notation $f \bmod x^j$ and $f \operatorname{quo} x^i$, we have the equality

$$f_{[i,j[} = [f]_i^j = (f \bmod x^j) \operatorname{quo} x^i = (f \operatorname{quo} x^i) \bmod x^{j-i} = \sum_{d=0}^{j-i-1} f_{[d+i]} x^d.$$

Our algorithms make use of the two program transformations presented in Section 1.7, namely *transposition* and *reversion*. The *reverse* of a polynomial f is $f^{\leftarrow} = x^{\deg(f)} f(1/x)$. Combining both notations, $f^{\leftarrow}_{[i,j[}$ denotes the polynomial $\sum_{d=0}^{j-i-1} f_{[j-1-d]} x^d$.

As mentioned earlier, we describe cumulative algorithms. Therefore, our basic operations are not only assignments but also *compound assignments* or *fused operations*. We use the notation $x := v$ for the assignment of the value v to x, $x \mathrel{+}= v$ for $x := x + v$, $x \mathrel{-}= v$ for $x := x - v$, $x \mathrel{*}= v$ for $x := x \times v$ and $x \mathrel{/}= v$ for $x := x/v$. We also extend the latter to $x \mathrel{*}= v \bmod m$ for $x := (x \times v) \bmod m$ and $x \mathrel{/}= v \bmod m$ for $x := (x/v) \bmod m$.

Several algorithms that we describe have complexity $O(\mathsf{M}(n) \log n)$. Actually, the extra logarithmic factor occurs when $\mathsf{M}(n)$ is quasi-linear. Otherwise, the complexity is actually $O(\mathsf{M}(n))$. We introduce the notation $\mathsf{M}^*(n)$ for these complexities. Formally, $\mathsf{M}^*(n) = O(\mathsf{M}(n) \log(n))$, or $O(\mathsf{M}(n))$ if $\mathsf{M}(n) = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$.

Finally, the literature on constant-space algorithms, although small, uses a very diverse and inconsistent vocabulary to describe the properties of these algorithms.[8] Below are the terms used in this document:

- *constant-space algorithm*: an algorithm that uses $O(1)$ extra space, beyond its input(s) and output(s);
- *in-place algorithm*: an algorithm that replaces (part of) its inputs by the output;
- *cumulative algorithm*: an algorithm that adds its output to (part of) its inputs.

An *in-place* algorithm may use *constant space*, but this is not required. A *linear-space in-place algorithm* replaces its input by its output, using a work space of linear size. A *cumulative* algorithm is a special case of an in-place algorithm.

[8] I plead guilty.

2 The computational models – and why they do matter

To express our algorithms and analyze their time-space complexities, a proper model of computation must be defined. Straight-line programs suffer two limitations. They handle fixed-size inputs, and there is no meaningful notion of space complexity attached to them. In Section 2.1, we introduce our models of computation based on the *algebraic Random Access Machine (RAM)* [4, 182]. While standard computational complexity theory defines space complexity using machines with read-only inputs and write-only outputs [14], we argue in Section 2.2 that this model is not suitable for the algebraic computations we are interested in, and we define several models of space complexity. Section 2.3 discusses space complexity of recursive algorithms, in particular the role of the call stack. Finally, we relate our definitions to standard space complexity classes in Section 2.4.

The title of this chapter is borrowed from [14, Chapter 1]: *The computational model—and why it doesn't matter.*

2.1 Algebraic algorithms and their models of computation

We first define our objects of study, namely algebraic problems.

Definition 2.1.1. A *fixed-size algebraic problem* over some ring R is a mapping $\pi : \mathsf{R}^n \to \mathsf{R}^n$. An *algebraic problem* over R is a family $(\pi_n)_{n\geq 0}$ of fixed-size algebraic problem, where $\pi_n : \mathsf{R}^n \to \mathsf{R}^n$.

We view an algebraic problem as a rewriting process. Given $v \in \mathsf{R}^n$, the goal is to replace v by $\pi(v)$. In particular, an algorithm that computes an algebraic problem π is given as inputs the entries of v in some registers, and must replace the value $v_{[i]}$ in the ith register by $\pi(v)_{[i]}$.

In our definition of an algebraic problem, there is *a priori* no distinction between inputs and outputs. For the example of polynomial multiplication, we would like to say that f and g are the inputs and h the output. We define a notion of inputs and outputs that can be used informally.

Definition 2.1.2. Let $\pi : \mathsf{R}^n \to \mathsf{R}^n$ be a fixed-size algebraic problem.

- The *inputs* of π are the indices $i \in \{0, \dots, n-1\}$ such that $\pi(v)$ *depends on* $v_{[i]}$, that is such that there exists $v, v' \in \mathsf{R}^n$ such that $v_{[i]} \neq v'_{[i]}$ but $v_{[j]} = v'_{[j]}$ for $i \neq j$, and $\pi(v) \neq \pi(v')$.
- The *outputs* of π are the indices $i \in \{0, \dots, n-1\}$ such that there exists $v \in \mathsf{R}^n$ where $\pi(v)_{[i]} \neq v_{[i]}$.

In many cases, it is more natural that an algebraic problem operates on tuple of vectors. For instance, a polynomial multiplication $h := f \times g$ operates on the triple of coefficient vectors $(\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h})$. Viewing the triple as one long vector or three smaller size vectors is equivalent. The definition of an algebraic problem is general enough to encompass several situations. Classical functions from computer science have an input and a (separate) output. A computation such as $y := f(x)$ where the input is x and the output is y is represented by $(x, y) \mapsto (x, f(x))$ in the model. Clearly, x is the input, and $y = f(x)$ is the output. But the model also allows cumulative computations such as $y \mathrel{+}= f(x)$, represented by $(x, y) \mapsto (x, y + f(x))$. In such a case, x is still an input, and y is both an input and an output. Finally, one can also have in-place computations such as $x := f(x)$, modeled as $x \mapsto f(x)$ where x is both the input and output. Without further precision, an algebraic problem is thus an in-place problem. Note that cumulative problems are a special case of in-place problems.

An SLP computes a fixed-size algebraic problem. To be able to define space complexity, we refine Definition 1.6.1.

Definition 2.1.3. A *fixed-size algebraic program* over R has n input-output registers $r_0, \ldots, r_{n-1}$, s temporary registers $t_0, \ldots, t_{s-1}$ and ℓ instructions of the form $u := v \star w$ where $u, v, w \in \{r_0, \ldots, r_{n-1}, t_0, \ldots, t_{s-1}\} \cup \mathrm{R}$ ($u \notin \mathrm{R}$) and $\star \in \{+, -, \times\}$.
It computes a fixed-size algebraic problem π if, given the initialization of its input-output registers r_i to v_i, $0 \leq i < n$ and of its temporary registers to 0, the final value of the input-output registers is $\pi(\boldsymbol{v})_{[i]}$, $0 \leq i < n$.

From a fixed-size algebraic program, we can build an SLP of the same length by replacing the i-th instruction $u := v \star w$ by $r_i \leftarrow u \star w$ and the subsequent uses of u on the right-hand-side of an instruction by r_i. To handle inputs of any size, one can use families of fixed-size algebraic programs. This defines a *nonuniform* model of computation. Although it is very much adapted to algebraic complexity theory and in particular to proving lower bounds [41], it is much less so to design algorithms and prove upper bounds that reflect the practice of programming. Instead of adding an outside uniformity requirement, we prefer work directly with a uniform model of computation, namely the algebraic RAM.

An algebraic RAM is parameterized by some ring R. It is made of *algebraic registers*, each containing a element from R, and *pointer registers* that store integers. We provide one possible definition. Other definitions are possible, *cf.* for instance [44, 194] for recent formalizations.

Definition 2.1.4. An *algebraic RAM* over a ring R has algebraic registers $(r_i)_{i \geq 0}$ storing elements of R, and pointer registers $(p_i)_{i \geq 0}$ storing nonnegative integers. It is controlled by a list of numbered instructions of one of the following forms:

algebraic instructions $r_a := r_b \star r_c$, $r_a := r_b$ or $r_a := \lambda$ where a, b and c are either integer constants or pointer registers (*indirect addressing*), $\lambda \in \mathrm{R}$, and $\star \in \{+, -, \times\}$;

pointer instructions $p_a := p_b \star p_c$, $p_a := p_b$ or $p_a := m$ where a, b and c are either integer constants or pointer registers (*indirect addressing*), $m \in \mathbb{Z}_{\geq 0}$, and $\star \in \{+, -, \times, /, \%\}$;

branching IF $r_a = 0$ THEN GOTO ℓ, where a is either an integer constant or a pointer register and ℓ is an instruction number.

On input $v \in \mathsf{R}^n$, v_i is stored in r_i for $0 \leq i < n$, p_0 contains n, and the other registers are initialized to 0. The instructions are executed in order, but if a GOTO is encountered and the corresponding register contains 0. Algebraic instructions are interpreted in R and pointer instructions in $\mathbb{Z}_{\geq 0}$: $a - b$ is actually $\max(0, a - b)$, $a/b = a \operatorname{quo} b$ and $a \,\%\, b = a \bmod b$. The computation stops when the last instruction is executed (if no GOTO is applied). The result is the content of the registers $r_0, \ldots, r_{n-1}$.
An algebraic RAM is *honest* if for any input size n, the values of the pointer registers are $O(\log n)$ during the computation.

There is no limitation in our definition on the number of registers used, or the magnitudes of the integers within the pointer registers. An *honest* algebraic RAM prevents any *cheating* and corresponds to the *transdichotomous model* [66] where the word size of the pointers is large enough to write the size of the inputs and outputs in $O(1)$ pointers. In the rest of this document, all algebraic RAMs are honest. Actually, we do not write the algorithms with the formal syntax of algebraic RAMs, rather in a standard pseudocode using conditional statements, loops and function calls (including recursive calls). A translation from the former to the latter is a classical exercise.

2.2 Space complexity of algebraic algorithms

The standard model for space complexity in computational complexity theory assumes that the inputs are read-only and the outputs write-only [14]. The space complexity of an algorithm is the number of extra registers required by the algorithm, not counting the inputs and outputs. We depart from this model for several reasons. On the theoretical side, a time-space quadratic lower bound is known for polynomial multiplication in this model [1]. This means that fast multiplication algorithms, thereby fast algorithms on polynomials more generally, require a polynomial amount of extra space. On the practical side, a programmer allows some memory for the output. There is no good reason to forbid the use of this space as work space. Our definition of algebraic problems as a rewriting process makes the traditional space complexity model inoperative. *It's not a bug, it's a feature.*

Our goal is to analyze the time- and space-complexity of algorithms for algebraic problems. There are some subtleties in defining the space complexity of a function, especially for low-space algorithms. The common feature of all possible definitions is that only the extra space is counted, not the inputs nor the outputs. In our definition of an algebraic problem, this means that the *algebraic space* of an algorithm computing some problem $\pi : \mathsf{R}^n \longrightarrow \mathsf{R}^n$ is the number of algebraic registers used by the algorithm, *in addition to the n input/output registers*. We also define the *pointer space* of an algorithm as the number of pointers it uses.

Definition 2.2.1. The *space complexity* of a fixed-size algebraic program is the number of temporary registers it uses.

An algebraic RAM computing some algebraic problem $(\pi_n)_{n\geq 0}$ has *algebraic space complexity* $s_a(n)$ if, on any input of size n, the only algebraic registers modified by the machine are $r_0, \ldots, r_{n+s_a(n)-1}$. It has *pointer space complexity* $s_p(n)$ if the only modified pointer registers are $p_0, \ldots, p_{s_p(n)-1}$.

Remark. We shall always assume in the definition of an algebraic problem $\pi_n : \mathsf{R}^n \longrightarrow \mathsf{R}^n$ that each of the n entries is either an input, an output, or both in the sense of Definition 2.1.2. Indeed, an entry that is neither an input nor an output plays no role in the computation and is only a placeholder. Prohibiting these prevents cheating when defining space complexity.

Our definition of space complexity does not put any restriction on the use of the input and output registers of the machine. This allows to define richer notions of space complexity, based on read-write permissions for the registers. We define three *permission models* for algebraic RAMs. They are based on the notions of *input-only* registers, that are inputs but not outputs, and *output-only* registers.

Definition 2.2.2. An algebraic RAM has permissions

- `ro/wo` if it never writes in an input-only register, and never reads from an output-only register;
- `ro/rw` if it never writes in an input-only register;
- `rw/rw` if it has no read nor write restriction.

The standard model `ro/wo` has one theoretically attractive feature. Consider two functions $f : \mathsf{R}^\ell \longrightarrow \mathsf{R}^m$ and $g : \mathsf{R}^m \longrightarrow \mathsf{R}^n$ and their composition $h = g \circ f : \mathsf{R}^\ell \longrightarrow \mathsf{R}^n$. (In our model, this would be $\pi : (\boldsymbol{u}, \boldsymbol{v}) \mapsto (\boldsymbol{u}, f(\boldsymbol{u}))$, $\rho : (\boldsymbol{v}, \boldsymbol{w}) \mapsto (\boldsymbol{v}, g(\boldsymbol{v}))$ and $\rho \circ \pi : (\boldsymbol{u}, \boldsymbol{w}) \mapsto (\boldsymbol{u}, g \circ f(\boldsymbol{u}))$ where $\boldsymbol{u} \in \mathsf{R}^\ell$, $\boldsymbol{v} \in \mathsf{R}^m$ and $\boldsymbol{w} \in \mathsf{R}^n$.) If f and g can both be computed in space $s(n)$ for some s, then $g \circ f$ can also be computed in the same space. To avoid storing the intermediate result $\boldsymbol{v} = f(\boldsymbol{u})$, the technique is to recompute each entry of $\boldsymbol{v}$ when it is needed during the computation of g [14, Chapter 4]. Unfortunately, this *composition theorem* does not hold in the time-space settings. Since entries of $f(\boldsymbol{u})$ must be (in general) computed several times, the time complexity of the composition of the two algorithms is not the sum of their original time complexities. This makes this model much less attractive for time-space complexity considerations.

The model `ro/rw` corresponds fairly closely to practice. Once the output space has been allocated, the programmer can use it as she wants. On the other hand, it is quite common to declare inputs (that are *input-only*) as constant using keywords such as `const` in C or C++, making an input read-only. This is in particular useful for parallel programming if there are parallel accesses to the inputs.

The model `rw/rw` is also natural in practice, especially in the *rewriting* view of algebraic problems. For instance for matrix computations, it is possible to replace a matrix M by its LU decomposition, where L is stored in the lower triangular part of M and U in the upper triangular part. For such computations, the inputs must obviously not be declared constant since they are modified by definition. The drawback of this model is to make parallel accesses to the inputs more complex.

2.3 Call stack and tail recursion

The model of algebraic RAM is able to simulate recursive calls. This requires to use a call stack. In our model the call stack will be made of pointers only. In most cases, the call stack for an algorithm operating on a size-n vector will be of size $O(\log n)$. Therefore, several of our algorithms use $O(1)$ algebraic registers and $O(\log n)$ pointers for the call stack.

A special case of recursive algorithms are tail recursive algorithms where the only recursive call is the last instruction of the algorithm. In such a case, the call stack is not required. This means that a tail recursive algorithm using $O(1)$ pointers can be simulated by loops still using $O(1)$ pointers. The situation can be generalized using *tail recursion modulo cons* [68] and its generalizations such as *tail recursion modulo context* [150], or *continuation-passing style* [188]. In our case, we shall not need such involved programming techniques. We only need to generalize tail recursion to the following situation. Consider an algebraic problem $(v, w) \mapsto (v, f(v, w))$ and a recursive algebraic algorithm $\mathcal{A}$ that makes only one recursive call. We assume that after the recursive call, $\mathcal{A}$ only operates on v. Then, one can define another algorithm $\mathcal{A}'$, tail recursive, that simply ignores the post-treatment on v. Then $\mathcal{A}'$ computes some problem $(v, w) \mapsto (x, f(v, w))$. And the post-treatment on v is some algorithm $\mathcal{A}$" that computes $x \mapsto v$. Since $\mathcal{A}'$ is tail recursive, it can be simulated by an algebraic RAM without call stack, and the sequential application of $\mathcal{A}'$ and $\mathcal{A}$" simulates $\mathcal{A}$. Therefore, $\mathcal{A}$ can be simulated an algebraic RAM without call stack.

2.4 Comparisons with standard space complexity theory

The main difference in our model is the relaxation on the inputs and outputs that are not read-only and write-only respectively. The more traditional model suffers from quadratic lower bounds as explained, but is also completely irrelevant for cumulative or in-place computations. The second difference is that we focus on time-space complexity classes. Our ultimate goal is algorithms that have quasi-linear time complexity, ideally $O(\mathsf{M}(n))$, and use a constant number of algebraic and pointer registers. This could be phrased in the settings of *fine-grained* complexity theory [207], and more specifically in terms of fine-grained time-space complexity classes [156].

Let us focus on space complexity. To compare our results with standard complexity classes such as L, we need to get back to bit complexity. Since algebraic algorithms naturally manipulate two kinds of data (pointers and algebraic elements), they do not nicely fit within the model. Yet consider an algorithm that takes as inputs n elements from a finite field $\mathbb{F}_p$. The input bit size is $O(n \log p)$, and a logarithmic space means an extra space of size $O(\log(n \log p)) = O(\log n + \log\log p)$. Depending on the relative magnitudes of n and p, a same algorithm could be considered in L or not. We can distinguish two regimes:

- If $\log p = \omega(\log n)$, it is not even possible to store a single algebraic element beyond the inputs and outputs. On the other hand, it is possible to store up to $\log_n p$ pointers. In this regime, details on the authorized algebraic operations in the model impact the space complexity. A cumulative product of ring elements $c \mathrel{+}= a \times b$ can be performed in logarithmic space if some kind of *fused multiply-add* is in the set of operations of the machine. But if only additions and multiplications are allowed, the intermediate result $t = a \times b$ must be stored, and the algorithm does not run in logarithmic space.

- If $\log p = O(\log n)$, only a constant number of pointers can be stored, but it is possible to store a larger number of algebraic registers (at most $\log_p n$).

Therefore, with the caveat of very large fields, one can consider that an algorithm that uses a constant number of registers of both kinds has a logarithmic space complexity. Nevertheless, as the above discussion shows, classical complexity classes are not the best option to study algebraic algorithms.

Another comparison can be made with two space complexity models that have been recently highlighted by the spectacular result that $\mathsf{TIME}(t(n)) \subseteq \mathsf{SPACE}(\sqrt{t(n)\log n})$ [210]. Both models use the multi-tape Turing machine. The first model, coined *global storage model*, is due to Goldreich [84]. It is a Turing machine that has a *global tape* where both the input and the output are written (as well as oracle queries), and one or several *local tapes* that serve as work space. This model is very close to the model `rw/rw` we defined. The second model is known as *catalytic computation* [40]. In this model, a *catalytic tape* is given that initially contains some data. The Turing machine is allowed to write on this catalytic tape, but it must be ultimately restored in its initial state. In our model `rw/rw`, this corresponds to adding a *dummy* input $\boldsymbol{c}$ that is neither an input nor an input. A problem π becomes $\pi^* : (u, c) \mapsto (\pi(u), c)$.

Algorithms in the ro/rw model 3

The main line of work for space-efficient polynomial computations is the investigation of low-space polynomial multiplication algorithms in the ro/rw model. The exact space complexity of Karatsuba's algorithm is analyzed by Maeder when one preallocates all the necessary memory once [157]. Then, an unpublished note by Thomé shows how to implement the algorithm using exactly n extra algebraic registers [205]. Roche describes several low-space multiplication algorithms: a variant of Karatsuba's algorithm that uses only constant algebraic space, and a constant-space FFT-based algorithm for polynomials of power-of-two size [189, 190]. Together with Harvey, they extend this latest result to any size [92]. Low-space Toom-Cook algorithms have been investigated in the context of polynomials over $\mathbb{F}_2$ [204].

In our work with Pascal Giorgi and Daniel S. Roche, we first provide generic (*algorithm-agnostic*) reductions for polynomial multiplication, proving that any linear-space multiplication algorithm has a constant-space variant with the same asymptotic time complexity. We also investigate *subproducts* such as lower and upper products or middle product. In particular, we highlight the links between the different problems in a *fine-grained time-space* framework. In a second work, we extend the results to other classical computer algebra operations such as power series inversion, Euclidean division, multipoint evaluation and interpolation. We refer to the original publications [C3, C6] for detailed analyses of the implied *hidden* constants in the time complexities.

The general idea of our algorithms is to use the free space in the output space as work space. Since the size of this space decreases over time while new coefficients of the result are computed, we need to adapt the standard algorithms to take this decrease into account. We design (tail) recursive algorithms that compute fewer coefficients at each recursive call to keep the space complexity constant.

For recursive algorithms, it is customary to assume that the input size is a power of two, or at least even. This is classically ensured by *padding* the input with zeroes if necessary. In the context of constant-space algorithms, this is not possible. Nevertheless, we can always use *fake padding* on the inputs. While accessing a nonexisting index in an array usually results in an error, we only need a data structure implementation where the error is replaced by returning 0. (This is easily implementable by catching exceptions in most programming languages.) Note though that fake padding cannot be used on the output in our context, since we use the output space as work space.

3.1 Generic reductions for polynomial multiplication

In this section, we describe *reductions* from any linear-space multiplication algorithm to a constant-space variant with close time complexity. We consider the standard full product of two polynomials $f, g \in \mathsf{R}[x]$, as well as the middle, lower and upper products. We then unravel the links that exist between low-space algorithms for all these variants.

3.1.1 *Full product*

The reduction starts with a full product algorithm that, given two size-n polynomials f and g, computes their product $f \times g$ in $O(\mathsf{M}(n))$ operations using $\leq cn$ extra registers. Our goal is to obtain a constant-space algorithm that computes $h := f \times g$. Writing $f = f_b + x^k f_t$ and $g = g_b + x^k g_t$ where f_b and g_b have size k for some $k < n$,

$$h = f_b g + x^k f_t g_b + x^{2k} f_t g_t.$$

The strategy is to compute $f_b g$ and $f_t g_b$ using several calls to the linear-space multiplication algorithm and $f_t g_t$ by a (tail) recursive call. Yet $f_b g + x^k f_t g_b$ must be written in $h_{[0,n+k-1[}$ and $x^{2k} f_t g_t$ in $h_{[2k,2n-1[}$. Since they overlap, the recursive call is not possible. Therefore, we generalize the problem to a *semi-cumulative full product*, that is $h \mathrel{+}= f \times g$ where $h \operatorname{quo} x^{n-1} = 0$. The recursive call becomes $h_{[2k,2n-1[} \mathrel{+}= f_t \times g_t$, where f_t and g_t have size $n-k$, and $h \operatorname{quo} x^{n-k-1} = 0$, hence it is legitimate. It remains to set k so that the computations of $f_b g$ and $f_t g_b$ can be performed in constant space, using the free space in the output to store intermediate results. The algorithm is illustrated as a Toeplitz matrix-vector product in Figure 3.1.

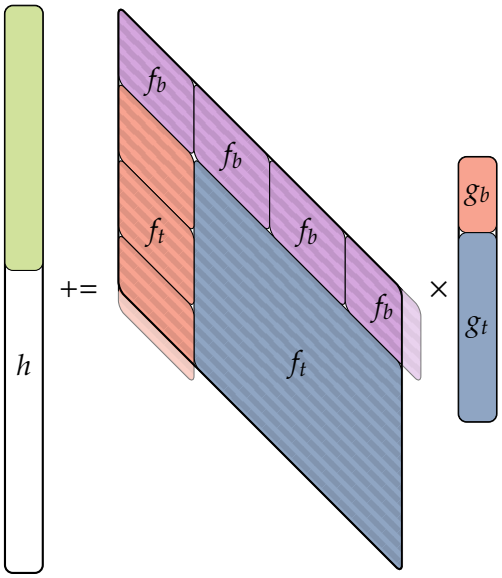


Figure 3.1. Algorithm 3.1.1 as a Toeplitz matrix-vector product. The first step is $f_0 g$ (purple), then $f_1 g_0$ (red) and finally a tail recursive call $f_1 g_1$ (blue). *Fake padding* is used to handle the shaded parts.

Since $f_b g$ can be computed using $\lceil n/k \rceil$ calls to the linear-space multiplication algorithm in size k, this computation requires $ck + 2k - 1$ extra space. The same holds for $f_t g_b$. And the free space in h is $h_{[n+k-1,2n-1[}$, of size $n-k$. Therefore, k must satisfy $(c+2)k - 1 \leq n - k$, that is $k \leq (n+1)/(c+3)$. The formal description is given as Algorithm 3.1.1.

Theorem 3.1.1 ([C3]). *Algorithm 3.1.1 (SemiCumulativeProduct) is correct, requires $O(1)$ extra space, and performs $O(\mathsf{M}(n))$ operations.*

Algorithm 3.1.1. SemiCumulativeProduct

Inputs: $f, g \in \mathsf{R}[x]$ of size n — *read-only*
$h \in \mathsf{R}[x]$ of size $2n-1$ such that $h \operatorname{quo} x^{n-1} = 0$ — *read-write*
Output: $h \mathrel{+}= f \times g$
Required: full product algorithm with space complexity $\leq cn$
Notations: $k = \lfloor \frac{n+1}{c+3} \rfloor$ and $\ell = \lceil n/k \rceil$
write $f = f_b + x^k f_t$ and $g = g_b + x^k g_t$

1 if $k = 0$: $h \mathrel{+}= fg$ — constant space
2 $h_{[0,n+k-1[} \mathrel{+}= f_b \times g$ — ℓ products, free: $h_{[n+k-1,2n[}$
3 $h_{[k,n+k-1[} \mathrel{+}= f_t \times g_b$ — $\ell - 1$ products, free: $h_{[n+k-1,2n[}$
4 $h_{[2k,2n[} \mathrel{+}= f_t \times g_t$ — tail recursive call

3.1.2 *Lower and upper products*

We are now given lower and upper product algorithms that take as inputs two size-n polynomials $f, g \in \mathsf{R}[x]$ and return $(f \times g) \bmod x^n$ and $(f \times g) \operatorname{quo} x^n$ respectively, both in $O(\mathsf{M}(n))$ operations and using $\leq cn$ extra space. We describe a constant-space algorithm to compute $h = (f \times g) \bmod x^n$. The reversed algorithm (Section 1.7) provides a constant-space upper algorithm.

First write $f = f_b + x^{n-k} f_t$, $g = g_b + x^{n-k} g_t$ and $h = h_b + x^{n-k} h_t$ for some k. Since $h_b = f_b \times g_b \bmod x^{n-k}$, it can be computed by a tail recursive call. We focus on computing h_t. An illustration as a triangular Toeplitz matrix-vector product is given in Figure 3.2.

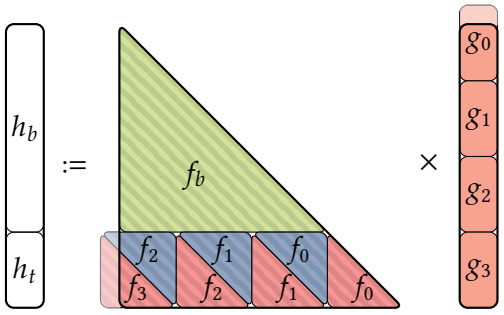


Figure 3.2. Algorithm 3.1.2 as a lower triangular Toeplitz matrix-vector product. The first steps correspond to the bottom strip covered by triangular Toeplitz matrices. The tail recursive call corresponds to the top triangular part.

Let $\ell = \lceil n/k \rceil$ and $r = k\ell - n$. Write now f and g as a sum of size-k polynomials, namely $f = \sum_{i=0}^{\ell-1} f_i x^{ki}$ and $g = g_0 + \sum_{j=1}^{\ell-1} g_j x^{kj-r}$ where $f_0, \ldots, f_{\ell-2}, g_1, \ldots g_{\ell-1}$ have size k, and $f_{\ell-1}$ and g_0 have size $k-r$. Their product can be expanded as

$$f \times g = \sum_{i=0}^{\ell-1} \sum_{j=0}^{\ell-1} f_i g_j x^{k(i+j)-r_{[j\neq 0]}}$$

where $r_{[j\neq 0]} = r$ if $j \neq 0$ and 0 otherwise. We want to compute $h_1 = [f \times g]_{n-k}^{n}$. The degrees of nonzero terms of $f_i g_j x^{k(i+j)-r_{[j\neq 0]}}$ are between $k(i+j) - r_{[j\neq 0]}$ and $k(i+j+2) - r_{[j\neq 0]} - 2$.

Therefore, if $k(i+j) - r_{[j\neq 0]} \geq n$ or $k(i+j+2) - r_{[j\neq 0]} - 2 < n-k$, that is if $i+j \geq \ell$ or $i+j < \ell - 2$, this summand plays no role in the computation of h_1. In other words,

$$h_1 = \left[x^{n-k} \cdot \sum_{i=0}^{\ell-1} f_i g_{\ell-1-i} + x^{n-2k} \cdot \sum_{i=0}^{\ell-2} f_i g_{\ell-2-i} \right]_{n-k}^{n}.$$

All nonzero terms in the first sum have degree $\geq n-k$, whence we need to compute $\sum_i f_i g_{\ell-1-i} \bmod x^k$ using ℓ lower products. The second sum has degree at most $n-2$. Therefore, we simply need to compute $\sum_{i=0}^{\ell-2} f_i g_{\ell-2-i} \operatorname{quo} x^k$ using $\ell - 1$ upper products.

Finally, to get the algorithm, we have to check that the linear-space lower and upper multiplication algorithms have enough free space in the output space. The results of these calls are written in h_1 of size k, therefore $n-k$ free registers are available in the output space. To compute the two sums, we need k registers to store intermediate results in addition to the ck registers required by the linear-space algorithms. Therefore, the algorithm works as long as $(c+1)k \leq n-k$, or $k \leq n/(c+2)$. The formal description is given as Algorithm 3.1.2.

Algorithm 3.1.2. LowerProduct

Inputs: $f, g \in \mathsf{R}[x]$ of size n — *read-only*
Output: $h := f \times g \bmod x^n$ — *read-write*
Required: lower and upper products algorithms with space complexity $\leq cn$
Notations: $k = \lfloor \frac{n}{c+3} \rfloor$, $\ell = \lceil n/k \rceil$ and $r = k\ell - n$
write $f = \sum_{i=0}^{\ell-1} f_i x^{ki}$ and $g = g_0 + \sum_{j=1}^{\ell-1} g_j x^{kj-r}$

1 if $n < c+2$: $h := f \times g \bmod x^n$ — constant space
2 for $i = 0$ to $\ell - 1$:
3 $\quad h_{[n-k,n[} \mathrel{+}= f_i \times g_{\ell-1-i} \bmod x^k$ — lower product, free: $h_{[0,n-k[}$
4 for $i = 0$ to $\ell - 2$:
5 $\quad h_{[n-k,n[} \mathrel{+}= f_i \times g_{\ell-2-i} \operatorname{quo} x^k$ — upper product, free: $h_{[0,n-k[}$
6 $h_{[0,n-k[} := f_{[0,n-k[} \times g_{[0,n-k[} \bmod x^{n-k}$ — tail recursive call

Theorem 3.1.2 ([C3]). *Algorithm 3.1.2 (LowerProduct) is correct, requires $O(1)$ extra space, and performs $O(\mathsf{M}(n))$ operations.*

As for Algorithm 3.1.1 (SemiCumulativeProduct), the polynomial h may contain some data initially. Assume for instance that the top $n-s$ coefficients are nonzero for some s. The first step is to compute $h_{[s,n[} \mathrel{+}= (f \times g)_{[s,n[}$. Similarly to the computation made in Lines 2 to 5, this can be computed using lower and upper products. Using the constant-space LowerProduct and its reversed algorithm UpperProduct, one can compute parts of the results (of size $n-s$) in the free space of h, and then add it to $h_{[s,n[}$. As long as $s \geq n/2$, this strategy works. Actually, the same strategy adapts when $s < n/2$ by computing $(fg)_{[s,n[}$ by chunks of size s. Then, the computation $h_{[0,s[} = fg \bmod x^s$ is another call to LowerProduct.

Corollary 3.1.3 (unpublished). *Given $f, g, h \in \mathsf{R}[x]$ of size n such that $h \bmod x^s = 0$ for some $s > 0$, one can compute $h \mathrel{+}= fg \bmod x^n$ with $O(1)$ extra space and $O(\mathsf{M}(n))$ operations if $s \geq n/2$, and $O((\frac{n}{s})^2\mathsf{M}(s))$ otherwise.*

Note that $(\frac{n}{s})^2\mathsf{M}(s) = O(\frac{n}{s}\mathsf{M}(n))$. If further $s \geq \alpha n$ for some constant α, $O((\frac{n}{s})^2\mathsf{M}(s)) = O(\frac{1}{\alpha}\mathsf{M}(n)) = O(\mathsf{M}(n))$. Therefore, if the amount of free space is a constant fraction of n, the algorithm has the same asymptotic complexity as Algorithm 3.1.2 (LowerProduct).

Later, we shall need a special case of this algorithm, when g has itself size s. In this variant, small values of s make the computation easier.

Corollary 3.1.4 (unpublished). *Given $f, g, h \in \mathsf{R}[x]$ of respective sizes n, s and n such that $s \leq n$ and $h \bmod x^s = 0$, one can compute $h \mathrel{+}= fg \bmod x^n$ with $O(1)$ extra space and $O(\mathsf{M}(n))$ operations.*

3.1.3 *Middle product*

As a starting point of the reduction, we use a *balanced* middle product algorithm that computes the middle product of $f \in \mathsf{R}[x]_{<2n-1}$ and $g \in \mathsf{R}[x]_{<n}$, in $O(\mathsf{M}(n))$ operations and using $\leq cn$ extra space. We use it to design a constant-space algorithm for the more general *unbalanced* middle product that takes as inputs $f \in \mathsf{R}[x]_{<m+n-1}$ and $g \in \mathsf{R}[x]_{<n}$ returns their middle product $[f \times g]_{n-1}^{m+n-1}$. The generalization is required to set up the recursion. An illustration as a triangular Toeplitz matrix-vector product is given in Figure 3.3.

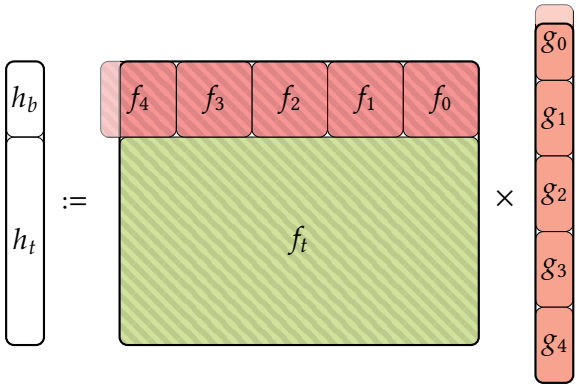


Figure 3.3. Algorithm 3.1.3 as a rectangular Toeplitz matrix-vector product. The first step corresponds to the top red strip and the tail recursive call to the bottom green strip.

Let $h = [f \times g]_{n-1}^{m+n-1}$ and write $h = h_b + x^k h_t$ for some $k < m$. Then $h_b = [f \times g]_{n-1}^{n-1+k} = [f_b \times g]_{n-1}^{n-1+k}$ where $f_b = f_{[0,n-1+k[}$ and $h_1 = [f \times g]_{n-1+k}^{m+n-1} = [f_t \times g]_{n-1}^{m-k+n-1}$ where $f_t = f_{[k,m+n-1[}$. These are two middle products, in size $(n+k-1, n)$ and $(m+n-k-1, n)$ respectively. The algorithm computes h_b using $\lceil n/k \rceil$ balanced middle products with extra space $(c+1)k$, and h_t with a tail recursive call. As long as $(c+1)k \leq m-k$, that is $k \leq m/(c+2)$, the algorithm is in constant space. Since the recursive call is in size $(m+n-k-1, n)$ and does not decrease the size of g, the complexity becomes $O(\mathsf{M}^*(n))$ (in the balanced case $m = n$). The formal description is given as Algorithm 3.1.3.

Algorithm 3.1.3. MiddleProduct

Inputs: $f, g \in \mathsf{R}[x]$ of respective sizes $m+n-1$ and n — *read-only*
Output: $h := [f \times g]_{n-1}^{m+n-1}$ — *read-write*
Required: *Balanced* middle product algorithm with space complexity $\leq cn$
Notation: $k = \lfloor \frac{m}{c+2} \rfloor$

1 if $m < c+2$: $h = [f \times g]_n^{m+n-1}$ — constant space
2 $h_{[0,k[} := [f_{[0,n-1+k[} \times g]_{n-1}^{n-1+k}$ — $\lceil \frac{n}{k} \rceil$ middle products, free: $h_{[k,m[}$
3 $h_{[k,m[} := [f_{[k,m+n-1[} \times g]_{n-1}^{m-k+n-1}$ — tail recursive call

Theorem 3.1.5 ([C3]). *Algorithm 3.1.3 (MiddleProduct) is correct, requires $O(1)$ extra space, and performs $O(\frac{\nu}{\mu}\mathsf{M}^*(\mu))$ operations, where $\mu = \min(m,n)$ and $\nu = \max(m,n)$.*

3.1.4 *Time-space preserving reductions between polynomial products*

We have proved that for the full, upper, and middle products, any linear-space algorithm can be turned into a constant-space variant, with a limited increase in time complexity. We now consider the relationships between these problems, in terms of time-space complexity.

Definition 3.1.6. A problem A is *time-space reducible* to a problem B, denoted $A \leq_{\mathsf{TISP}} B$, if there exists an algorithm for A that, given access to an algorithm for B that runs in time $t(n)$ and space $s(n)$, runs in time $O(t(n))$ and space $O(s(n))$. We write $A \equiv_{\mathsf{TISP}} B$ if $A \leq_{\mathsf{TISP}} B$ and $B \leq_{\mathsf{TISP}} A$.

To compare the problems, we define them formally.

Definition 3.1.7. Let f, g, h, ℓ and $u \in \mathsf{R}[x]$ where f, g and ℓ have size n, h has size $2n-1$ and u has size $n-1$. We define the following problems by the computation they perform:

- FullProd: $h := f \times g$;
- FullProd$^+$: $h \mathrel{+}= f \times g$, assuming that $h \operatorname{quo} x^n = 0$;
- LowProd: $\ell := f \times g \bmod x^n$;
- UppProd: $u := f \times g \operatorname{quo} x^n$;
- MidProd: $f := [h \times g]_{n-1}^{2n-1}$.

By reversal, lower and upper products are equivalent, that is LowProd $\equiv_{\mathsf{TISP}}$ UppProd:

$$(f \times g) \operatorname{quo} x^n = \left((f \operatorname{quo} x)^{\leftarrow} \times (g \operatorname{quo} x)^{\leftarrow} \bmod x^{n-1}\right)^{\leftarrow}.$$

Since $f \times g = (fg) \bmod x^n + x^n \left[(fg) \operatorname{quo} x^n\right]$, the previous equivalence implies FullProd $\leq_{\mathsf{TISP}}$ LowProd. For FullProd$^+$, one can first compute $(fg) \bmod x^n$ in the free

(upper) part of h, add it to the lower part, and finally compute (fg) quo x^n in the upper part. Therefore, FullProd$^+$ $\leq_{\mathsf{TISP}}$ LowProd.

Conversely, given an algorithm for FullProd$^+$, write $f = f_0 + x^{\lfloor n/2 \rfloor} f_1$ and $g = g_0 + x^{\lfloor n/2 \rfloor} g_1$. Then $(fg) \bmod x^n = f_0 g_0 + x^{\lfloor n/2 \rfloor}((f_0 g_1 + f_1 g_0) \bmod x^{\lceil n/2 \rceil})$. This computation reduces to FullProd$^+$: First compute $h = f_0 g_1$ since h is free; Erase the upper coefficients of h and compute h += $f_1 g_0$; Finally move the lower coefficients to the upper part of h and compute h += $f_0 g_0$. This shows LowProd $\leq_{\mathsf{TISP}}$ FullProd$^+$, whence the equivalence.

Finally, writing $(fg) \bmod x^n = [(x^{n-1} \cdot f) \times g]_{n-1}^{2n-1}$ proves LowProd $\leq_{\mathsf{TISP}}$ MidProd using *fake padding*.

As a result, we obtain the following reductions and equivalences.

Theorem 3.1.8 ([C3]). *There exist time-space reductions between full, lower, upper and middle products as depicted below:*

$$\mathsf{FullProd} \leq_{TISP} \mathsf{FullProd}^+ \equiv_{TISP} \mathsf{LowProd} \equiv_{TISP} \mathsf{UppProd} \leq_{TISP} \mathsf{MidProd}.$$

3.2 Power series inversion and Euclidean division

The goal of this part is to prove that the fast Euclidean division algorithm can be made constant-space while preserving the same asymptotic time complexity $O(\mathsf{M}(n))$. To this end, we first investigate the case of power series inversion and division which serve as building blocks.

3.2.1 *Power series inversion and division*

Our goal is to adapt Newton iteration for power series inversion to make it work in constant space. Let $\phi \in \mathsf{R}[[x]]$ be an invertible power series and $\psi \in \mathsf{R}[[x]]$ be its inverse. We assume that $f = \phi \bmod x^n$ is its truncation at precision n, and we aim to compute $g = \psi \bmod x^n$. Assume that at some point, $g = \psi \bmod x^k$ for some $k \leq n/2$. Then one step of Newton iteration updates g as

$$g := g + (1 - gf) \cdot g \bmod x^{2k}$$

so that the new value of g satisfies $g = \psi \bmod x^{2k}$ [72]. Since $gf = 1 \bmod x^k$ by assumption, the lower part of $(1 - gf)$ is zero. For efficiency reasons, and in particular space efficiency, the update can be computed as

$$g := g - x^k \times \left(\left[(f^* \bmod x^{2k-1}) \times g \right]_{k-1}^{2k-1} \times g \right) \bmod x^k$$

where $f^* = f$ quo x. That is, g is updated using a middle product followed by a lower product. If ψ is to be computed at precision n, g is stored as a vector of size n. When k is small enough, we can use the free space $g_{[2k,n[}$ to compute the middle and lower products, without extra space. But when k approaches $n/2$, no free space is available anymore. The solution is to slow down the computation and compute fewer coefficients than what a standard step would do. This is summarized in the following lemma.

Lemma 3.2.1. *Let $\phi \in \mathsf{R}[[x]]$ invertible, $f = \phi \bmod x^{k+\ell}$ and g be the inverse of ϕ at precision k. Then*

$$g - x^k \times \left(\left[f^* \times g \right]_{k-1}^{k+\ell-1} \times g \right) \bmod x^\ell$$

is the inverse of ϕ at precision $k+\ell$, where $f^* = f \operatorname{quo} x$.

The algorithm adjusts the value of ℓ to keep enough free space. It requires ℓ registers for the result of the middle product, ℓ other registers for intermediate results since the middle product is unbalanced, and $c\ell$ registers as extra space for the middle product computation itself. The same holds for the lower product. Together, the condition on ℓ is $(c+2)\ell \leq n-k$. We remark that the formula of Lemma 3.2.1 computes ℓ new coefficients of the inverse by using $k+\ell-1$ coefficients of f. The number of coefficients keeps increasing during the algorithm, even if ℓ itself decreases. As for the constant-space middle product, the time complexity increases to $O(\mathsf{M}^*(n))$.

Algorithm 3.2.1. Inversion

Inputs: $f = \phi \bmod x^n$ such that ϕ_0 is a unit — *read-only*
Output: $g := \psi \bmod x^n$ such that $\phi \times \psi = 1$ in $\mathsf{R}[[x]]$ — *read-write*
Required: Middle and lower product algorithms with space complexity $\leq cn$

1 $g_{[0]} := f_{[0]}^{-1}$
2 let $k = 1$ and $\ell = 1$
3 while $\ell > 0$:
4 $\quad g_{[n-\ell,n[} := \left[f_{[1,k+\ell[} \times g_{[0,k[} \right]_{k-1}^{k+\ell-1}$ — middle product, free: $g_{[k,n-\ell[}$
5 $\quad g_{[k,k+\ell[} := -(g_{[0,\ell[} \times g_{[n-\ell,n[}) \bmod x^\ell$ — lower product, free: $g_{[k+\ell,n-\ell[}$
6 $\quad$ update $k = k+\ell$ and $\ell = \min(k, \lfloor \frac{n-k}{c+2} \rfloor)$
7 $g_{[k,n[} := -g_{[0,n-k[} \times \left[f_{[1,n[} \times g_{[0,k[} \right]_{k-1}^{n-1} \bmod x^{n-k}$ — constant space

Theorem 3.2.2 ([C3]). *Algorithm 3.2.1 (Inversion) is correct, requires $O(1)$ extra space, and performs $O(\mathsf{M}^*(n))$ operations.*

The division ϕ/ψ of two power series $\phi, \psi \in \mathsf{R}[[x]]$ can be computed as $\phi \times \psi^{-1}$. The drawback is that it requires to store the intermediate result ψ^{-1}. Karp and Markstein's trick incorporates the multiplication into the final step of Newton iteration when computing ψ^{-1} [136]. Since the last iteration is replaced in our constant-space variant by several iterations, we need to incorporate the multiplication into several iterations. The following lemma is a generalization of their method, that can also be seen as a generalization of the standard Newton iteration for ϕ/ψ.

Lemma 3.2.3. *Let $\phi, \psi \in \mathsf{R}[[x]]$, ψ invertible, $f = \phi \bmod x^{k+\ell}$, $g = \psi \bmod x^{k+\ell}$,*

$g^* = g \operatorname{quo} x$, and $h = \phi/\psi \bmod x^k$. Then

$$h^* = h + x^k \cdot \left((g^{-1} \bmod x^\ell) \times \left(f \operatorname{quo} x^k - [g^* \times h]_{k-1}^{k+\ell-1}\right)\right) \bmod x^\ell$$

satisfies $h^* = \phi/\psi \bmod x^{k+\ell}$.

The algorithm has the same structure as Algorithm 3.2.1 (Inversion), using the free output space as work space. Note that g^{-1} is needed during the whole algorithm, but with less and less precision since ℓ has to decrease. Therefore, we can compute it once at the beginning, and store it in the reversed order as the last coefficients of h and progressively overwrite the unneeded coefficients.

Algorithm 3.2.2. Division

Inputs: $f = \phi \bmod x^n$, $g = \psi \bmod x^n$ such that ψ_0 is a unit — *read-only*
Output: $h := \phi/\psi \bmod x^n$ — *read-write*
Required: middle and lower product, and power series inversion algorithms with space complexity $\le cn$

1 let $k = \lfloor \frac{n}{c+2} \rfloor$ and $\ell = \lfloor \frac{n-k}{c+3} \rfloor$
2 $h^{\leftarrow}_{[n-k,n[} := g^{-1}_{[0,k[} \bmod x^k$ — inversion, free: $h_{[0,n-k[}$
3 $h_{[0,k[} := (f_{[0,k[} \times h^{\leftarrow}_{[n-k,n[}) \bmod x^k$ — lower product, free: $h_{[k,n-k[}$
4 while $\ell > 0$:
5 $\quad h_{[n-2\ell,n-\ell[} := -[g_{[1,k+\ell[} \times h_{[0,k[}]_{k-1}^{k+\ell-1}$ — middle product, free: $h_{[k,n-2\ell[}$
6 $\quad h_{[n-2\ell,n-\ell[}$ += $f_{[k,k+\ell[}$
7 $\quad h_{[k,k+\ell[} := (h_{[n-2\ell,n-\ell[} \times h^{\leftarrow}_{[n-\ell,n[}) \bmod x^\ell$ — lower product, free: $h_{[k+\ell,n-2\ell[}$
8 $\quad$ update $k = k + \ell$ and $\ell = \lfloor \frac{n-k}{c+3} \rfloor$
9 $h_{[k,n[} := (f_{[k,n[} - [g_{[1,n[} \times h_{[0,k[}]_{k-1}^{n-1}) \times h^{\leftarrow}_{[k,n[} \bmod x^{n-k}$ — constant space

Theorem 3.2.4 ([C3]). *Algorithm 3.2.2 (Division) is correct, requires $O(1)$ extra space, and performs $O(\mathsf{M}^*(n))$ operations.*

Constant-space power series division is the main ingredient of a constant-space Euclidean division of polynomials. In this use, the dividend ϕ is actually an intermediate result. Therefore, it can be overwritten during the computation. Lemma 3.2.3 shows than once the first k coefficients of ϕ/ψ have been computed, the first k coefficients of ϕ are not needed anymore. The idea is then to actually replace $\phi \bmod x^n$ by $\phi/\psi \bmod x^n$, using an extra linear space. As a result, the complexity is back to $O(\mathsf{M}(n))$.

We describe the algorithm in full generality when the extra space has any size s. At each iteration, $\ell = \lfloor \frac{s}{c+3} \rfloor$ new coefficients are computed. The algorithm thus uses $O(n/s)$ iterations of cost $O(\frac{n}{s}\mathsf{M}(s))$.

Theorem 3.2.5 ([C3]). *Algorithm 3.2.3 (SmallSpaceInPlaceDivision) is correct, requires s extra space, and performs $O((\frac{n}{s})^2\mathsf{M}(s))$ operations in* R.

Algorithm 3.2.3. SmallSpaceInPlaceDivision

Inputs: $f = \phi \bmod x^n$ *read-write*
$g = \psi \bmod x^n$ such that ψ_0 is a unit *read-only*
Output: $f := \phi/\psi \bmod x^n$
Required: middle product, lower product and inversion algorithms with space complexity $\leq cn$, and an extra space t of size s

1 let $k = 0$ and $\ell = \lfloor \frac{s}{c+3} \rfloor$
2 $t_{[0,\ell[} := g^{-1}_{[0,\ell[} \bmod x^\ell$ — inversion, free: $t_{[\ell,s[}$
3 $t_{[\ell,2\ell[} := (f_{[0,\ell[} \times t_{[0,\ell[}) \bmod x^\ell$ — lower product, free: $t_{[2\ell,s[}$
4 $f_{[0,\ell[} := t_{[\ell,2\ell[}$
5 while $k < n$:
6 update $k = k + \ell$ and $\ell = \min(\lfloor \frac{s}{c+3} \rfloor, n - k)$
7 $t_{[\ell,2\ell[} := -[g_{[1,k+\ell[} \times f_{[0,k[}]_{k-1}^{k+\ell-1}$ — middle product, free: $t_{[2\ell,s[}$
8 $t_{[\ell,2\ell[}$ += $f_{[k,k+\ell[}$
9 $f_{[k,k+\ell[} := (t_{[\ell,2\ell[} \times t_{[0,\ell[}) \bmod x^\ell$ — lower product, free: $t_{[2\ell,s[}$

If $s = \alpha n$ for some constant α, the complexity becomes $O(\mathsf{M}(n))$. For $s = 1$ on the other hand, the complexity is $O(n^2)$ and the algorithm corresponds to the naive algorithm, which requires no extra space.

3.2.2 *Euclidean division*

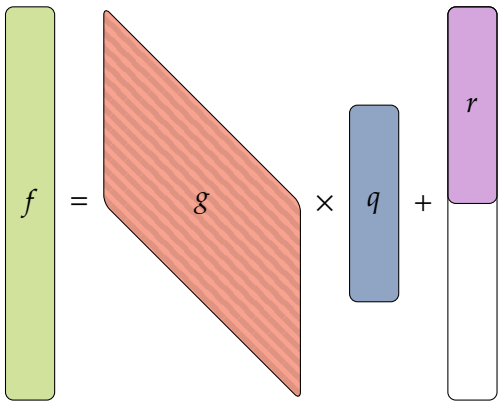
Given $f \in \mathsf{R}[x]_{<m+n-1}$ and $g \in \mathsf{R}[x]_{<n}$ whose leading coefficient is a unit, the goal is to compute a size-m quotient q and a size-$(n-1)$ remainder r such that $f = gq + r$. We also consider the computations of the sole quotient or the sole remainder. Fast algorithms for Euclidean division start by computing the quotient q. Since $f \operatorname{quo} x^{n-1} = (g \times q) \operatorname{quo} x^{n-1}$ is an upper product, q can be computed as the inverse of an upper product, namely a *reversed* power series division $q := \left(f^{\leftarrow}/g^{\leftarrow} \bmod x^m\right)^{\leftarrow}$. The remainder is computed as $r := f - gq$. Figure 3.4 is a linear-algebraic presentation of this idea for $m = n$.

As a first remark, computing the quotient only without any space for the remainder is equivalent to power series division.

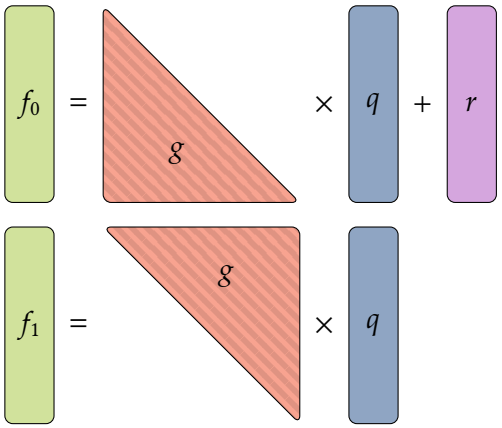
If $m < n-1$, the size-m quotient can be computed using the free space of the remainder as the work space, in time $O(\mathsf{M}(m))$. Then, r is computed as $r = (f - gq) \bmod x^{n-1}$ with a constant-space lower product $gq \bmod x^{n-1}$ in time $O(\mathsf{M}(n))$, for a total complexity $O(\mathsf{M}(m) + \mathsf{M}(n))$. If the same algorithm is used to compute the remainder only, the space for the quotient counts as extra space.

Let us focus on the case $m \geq n-1$ where the best complexity $O(\frac{m}{n}\mathsf{M}(n))$ relies on the long division algorithm that computes the quotient block by block. For the first blocks, there is enough free space in q to use non-constant-space algorithms. The difficulty is to compute the last block of the quotient. It actually boils down to the case $m = n-1$. In this situation, $f = f_0 + x^{n-1} f_1$ has size $2n-1$ and g has size n. Hence, we need to compute the n coefficients of q and $(n-1)$ coefficients of r. To compute q, we rely on Algorithm 3.2.3 (SmallSpaceInPlaceDivision). Since $q^{\leftarrow} = f_1^{\leftarrow}/g^{\leftarrow} \bmod x^n$, we set $q := f_1$, then $q^{\leftarrow} \mathrel{/}= g^{\leftarrow} \bmod x^{n-1}$ using the free space of r as extra space. Finally, the remainder is

(a) Start with the equation $f = gq + r$ that defines the Euclidean division.



(b) Split it into two equations $f_0 = (gq) \bmod x^n + r$ and $f_1 = (gq) \operatorname{quo} x^n$.



(c) Solve the two equations to get $r = f_0 - gq \bmod x^n$ and $q^{\leftarrow} = (g^{\leftarrow}/f_1^{\leftarrow}) \bmod x^n$.

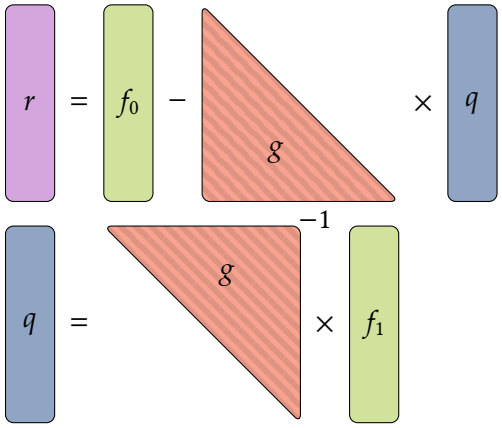


Figure 3.4. Derivation of the fast Euclidean division algorithm *via* linear algebra. In the last part, an upper product must be inverted. This corresponds to a reversed power series division. The remainder is computed with a lower product.

computed using a constant-space lower product.

In the presentation below, for simplicity, we use the same method to compute each block of the quotient even though only the last division needs to be compute in place. Replacing these in-place divisions by linear-space divisions would improve the overall complexity by a constant.

Algorithm 3.2.4. EuclideanDivision

Inputs: f of size $m+n-1$, g of size n, $m \geq n-1$ — *read-only*
Outputs: q of size m and r of size $n-1$ such that $f = gq + r$ — *read-write*
Required: constant-space lower product algorithm
in-place reversed division algorithm with space complexity $< n$
Notations: $k = \lfloor \frac{m}{n} \rfloor$, $\ell = m \bmod n$
write $q = \sum_{j=0}^{k} q_j x^{jn}$, $f = \sum_{j=0}^{k+1} f_j x^{jn}$
let $g_* = g_{[0,n-1[}$, $q_{j*} = q_{j[0,n-1[}$ and $q_j^* = q_{j[1,n[}$

1 $q_k := f_{k+1}$
2 $q_k^{\leftarrow} \mathrel{/}= g_{[n-\ell,n[}^{\leftarrow} \bmod x^{\ell}$ — reversed division, free: r, $q_{[0,kn[}$
3 $q_{k-1}^* := -(g_{[0,\ell[} \times q_k) \bmod x^{\ell}$ — lower product, free: r, $q_{[0,(k-1)n[}$
4 $q_{k-1} \mathrel{+}= f_k$
5 for $j = k-1$ to 1:
6 $\quad q_j^{\leftarrow} \mathrel{/}= g^{\leftarrow} \bmod x^{n}$ — reversed division, free: r, $q_{[0,jn[}$
7 $\quad q_{j-1}^* := -(g_* \times q_{j*}) \bmod x^{n-1}$ — lower product, free: r, $q_{[0,(j-1)n[}$
8 $\quad q_{j-1} \mathrel{+}= f_j$
9 $q_0^{\leftarrow} \mathrel{/}= g^{\leftarrow} \bmod x^{n}$ — reversed division, free: r
10 $r := -(g_* \times q_{0*}) \bmod x^{n-1}$ — lower product with no work space
11 $r \mathrel{+}= f_{[0,n-1[}$

Theorem 3.2.6 ([C3]). *Algorithm 3.2.4 (EuclideanDivision) is correct, requires $O(1)$ extra space, and performs $O(\frac{m}{n}\mathsf{M}(n))$ operations in* R.

If only the remainder is to be computed, one can actually forget about the coefficients of the quotient during the computation. This means that only one block of the quotient needs to fit in the extra space. Using the small-space in-place reversed division and the semi-cumulative lower product, any space $s < n$ is sufficient to compute the remainder. (In the presentation, we assume $s \leq n-1$ to be able to store blocks of size s within the remainder space.)

Theorem 3.2.7 (unpublished). *Algorithm 3.2.5 (SmallSpaceRemainder) is correct, requires an extra space of size s, and performs $O(\frac{m}{s}\mathsf{M}(n))$ operations.*

For $s = \Theta(1)$, the complexity bound becomes $O(m\mathsf{M}(n))$. Actually, a slightly finer analysis shows that the complexity is $O(mn)$. One can notice that this is the standard long division algorithm, that indeed requires no extra space. For $s = \Theta(n)$, the complexity is $O(\frac{m}{n}\mathsf{M}(n))$ as for the linear-space algorithm.

Algorithm 3.2.5. SmallSpaceRemainder

Inputs: f of size $m+n-1$, g of size n, $m \geq n-1$ — *read-only*
Outputs: r of size $n-1$ such that $r = f \bmod g$ — *read-write*
Required: constant-space semi-cumulative lower product algorithm
in-place reversed division algorithm with space complexity $\leq n-1$
extra space t of size $s \leq n-1$
Notations: $k = \lfloor \frac{m}{s} \rfloor$, $\ell = m \bmod s$
$f_j = f_{[n-1+(j-1)s, n-1+js[}$ for $0 \leq j \leq k$ and $f_{k+1} = f_{[m+n-1-\ell, m+n-1[}$
$r^* = r_{[n-1-s, n-1[}$, $g^* = g_{[n-s,n[}$ and $g_* = g_{[0,n-1[}$

1 $t_{[0,\ell[} := f_{k+1}$
2 $t^{\leftarrow}_{[0,\ell[} \mathrel{/}= g^{\leftarrow}_{[n-\ell,n[} \bmod x^{\ell}$ — reversed division, free: r
3 $r := (g_* \times t) \bmod x^{n-1}$ — lower product in constant space
4 $t := r^* + f_k$
5 for $j = k-1$ to 0:
6 $\quad r_{[s,n[} := r_{[0,n-1-s[}$ — right shift
7 $\quad t^{\leftarrow} \mathrel{/}= (g^*)^{\leftarrow} \bmod x^{s}$ — reversed division, free: $r_{[0,s[}$
8 $\quad r \mathrel{-}= (g_* \times t) \bmod x^{n-1}$ — lower product in constant space
9 $\quad t := r^* + f_j$
10 $r := t$
11 $r_{[0,n-1-s[} \mathrel{+}= f_{[0,n-1-s[}$

3.3 Multi-point evaluation and interpolation

The standard and fastest algorithms for multipoint evaluation and interpolation do not use only a linear amount of extra space, but rather an extra space of size $n \log n + O(n)$ to store the so-called subproduct tree. Von zur Gathen and Shoup noticed that the space complexity of multipoint evaluation can be reduced to linear with the same asymptotic time complexity [74, 75]. For instance consider the evaluation of a size-n polynomial f on n points. By grouping the points as $\log(n)$ groups of $n/\log(n)$ points, one can perform $\log(n)$ multipoint evaluations that each require $n + O(n/\log n)$ extra space. The complexity is $O(\mathsf{M}(\frac{n}{\log n}) \log(\frac{n}{\log n}) \log n) = O(\mathsf{M}(n) \log n)$. Once a linear-space algorithm is known for multipoint evaluation, obtaining a constant-space version uses the same ideas as before. First evaluate f on $k < n$ points using the free output space as work space, and recur. We obtain the following result.

Theorem 3.3.1 ([C3]). *Let $f \in \mathsf{R}[x]$ of size n and $a_0, \ldots, a_{n-1}$ in R. One can compute $f(a_0), \ldots, f(a_{n-1})$ in $O(\mathsf{M}(n)\log(n))$ operations using $O(1)$ extra registers.*

Our approach may appear a bit cumbersome. We fix a number k of points on which to perform a partial multipoint evaluation. Then, in order for this multipoint evaluation to be computable in linear space, this set of points itself is split into $\log(k)$ blocks of $k/\log(k)$ points. It is of course possible to directly choose a value of k so that there is enough free space to perform the standard multipoint evaluation of space complexity $k\log(k) + O(k)$. This probably improves slightly the complexity by a constant factor, but makes the analysis

painful.

The main difficulty lies in interpolation. Let $(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})$ be n pairs of evaluations with invertible differences $a_i - a_j$, $i \neq j$. The goal is to compute the unique size-n polynomial $f \in \mathsf{R}[x]$ such that $f(a_i) = b_i$ for $0 \leq i < n$, with a constant-space algorithm. The first ingredient is to provide a variant of polynomial interpolation that computes $f \bmod x^k$ using $O(k)$ extra space. For simplicity, let us assume that k divides n. For $1 \leq i \leq n/k$, let $m_i = \prod_{j=k(i-1)}^{ki-1}(x - a_j)$ and $s_i = m/m_i$ where $m = \prod_{i=1}^{n}(x - a_i)$. Note that $s_i = \prod_{j \neq i} m_j$. Lagrange interpolation formula can be written by block as

$$f(x) = \sum_{i=1}^{n/k} \sum_{j=k(i-1)}^{ki-1} b_j \cdot \frac{m_j(x)}{m_j(a_j)} = m(x) \sum_{i=1}^{n/k} \frac{n_i(x)}{m_i(x)} = \sum_{i=1}^{n/k} n_i(x) s_i(x)$$

for some size-k polynomials $n_1, \ldots, n_{n/k}$. Therefore, we need to compute

$$f \bmod x^k = \sum_{i=1}^{n/k} n_i \cdot (s_i \bmod x^k) \bmod x^k.$$

One can observe that for $k(i-1) \leq j < ki$, $f(a_j) = n_i(a_j)s_i(a_j)$. Since $m_i(a_j) = 0$, $s_i(a_j) = (s_i \bmod m_i)(a_j)$. Therefore, n_i is the unique size-k polynomial satisfying $n_i(a_j) = b_j/(s_i \bmod m_i)(a_j)$. It can be computed using evaluation and interpolation: First compute $s_i \bmod m_i$; Evaluate it at the a_j's; Perform k divisions to get each $n_i(a_j)$; Finally interpolate n_i.

The second ingredient is to generalize the previous approach when some initial coefficients of f are known. Writing $f = g + x^s h$ where g is known, we want to compute $h \bmod x^k$ from some evaluations of f. Since $f = g + x^s h$, we can write $h(a_j) = (f(a_j) - g(a_j))/a_j^s$. Therefore, the algorithm sketched for $f \bmod x^k$ can be generalized for $h \bmod x^k$, by computing the evaluations of h using multipoint evaluation and fast exponentiation. Algorithm 3.3.1 describes this approach.

Lemma 3.3.2. *Algorithm 3.3.1 (PartialInterpolation) is correct, requires $O(k)$ extra space, and has complexity $O(\frac{(n-s)^2}{k^2}\mathsf{M}(k)\log k)$.*

From this algorithm, a constant-space interpolation algorithm directly follows. We start with $h = 0$, and set k so that the extra space required by Algorithm 3.3.1 (PartialInterpolation) fits within the free output space of size $n - k$. We progressively compute new coefficients of the interpolant f. Altogether, we obtain the following result.

Theorem 3.3.3 ([C3]). *Given n pairs $(a_i, b_i)_{0 \leq i < n}$ where $a_i - a_j$ is invertible for $i \neq j$, the unique size-n interpolant f such that $f(a_i) = b_i$ for $0 \leq i < n$ can be computed in $O(\mathsf{M}(n)\log n)$ operations and $O(1)$ extra space.*

The constant in the time complexity of this algorithm is quite large. While the fastest quasi-linear-space algorithm runs in time $\frac{5}{2}\mathsf{M}(n)\log(n) + O(n)$ [32], our algorithm is estimated to run in time $105\mathsf{M}(n)\log(n) + O(n)$. As mentioned earlier, an approach to reduce the constant could be to bypass the linear-space multipoint evaluation and interpolation algorithms, and to directly work with the fastest known algorithm.

Algorithm 3.3.1. PartialInterpolation

Inputs: $g \in \mathsf{R}[x]$ of size s, $(a_i, b_i)_{0 \le i < n-s}$ with $a_i - a_j$ invertible for $i \neq j$, an integer $k \le n - s$ *read-only*

Output: $h \bmod x^k$ where $f = g + x^s \cdot h$ is the unique size-n polynomial such that $f(a_i) = b_i$, $0 \le i < n - s$ *read-write*

Required: linear-space MP-EVALUATION and INTERPOLATION
constant-space full product and Euclidean division

1 for $i = 1$ to $(n-s)/k$:
2 $\quad m_i := \prod_{j=k(i-1)}^{ki-1}(x - a_j)$
3 $\quad s_i^{(k)} := 1$; $s_i^{(m)} := 1$
4 $\quad$ for $j = 1$ to $(n-s)/k$, $j \neq i$:
5 $\quad\quad m_k := \prod_{\ell=k(j-1)}^{kj-1}(x - a_\ell)$
6 $\quad\quad s_i^{(k)} := s_i^{(k)} \times m_j \bmod x^k$
7 $\quad\quad s_i^{(m)} := s_i^{(m)} \times m_j \bmod m_i$
8 $\quad g^{(m)} \leftarrow g \bmod m_i$
9 $\quad (c_0, \ldots, c_{k-1}) :=$ MP-EVALUATION$(s_i^{(m)}, (a_{k(i-1)}, \ldots, a_{ki-1}))$
10 $\quad (d_0, \ldots, d_{k-1}) :=$ MP-EVALUATION$(s_i^{(m)}, (a_{k(i-1)}, \ldots, a_{ki-1}))$
11 $\quad$ for $j = 0$ to $k-1$: $c_j := (b_{j+k(i-1)} - d_j)/(a^s_{j+k(i-1)} c_j)$
12 $\quad n_i :=$ INTERPOLATION$((a_{j+k(i-1)}, c_j)_{0 \le j < k})$
13 $\quad h \mathrel{+}= n_i \times s_i^{(k)} \bmod x^k$

Open problem 3.1. What are the smallest values of c_e and c_i such that multipoint evaluation and interpolation can be computed in constant space and time complexity $c_e \mathsf{M}(n)\log(n) + O(n)$ and $c_i \mathsf{M}(n)\log(n) + O(n)$ respectively?

Algorithms in the rw/rw model 4

We have seen constant-space variants of several polynomial algorithms, including Euclidean division. But for the case of computing the remainder only, the best we could achieve is (small) linear space. In this chapter, we use the more permissive model rw/rw to investigate this problem. To achieve the result, we must revisit the whole chain of algorithms: full, lower, upper and middle products, power series inversion and division, and finally Euclidean division. In particular, we design algorithms for *cumulative* and *in-place* variants of these operations. As a result, we can also perform more general operations such as a cumulative modular product, that is $r \mathrel{+}= f \times g \bmod h$.

In contrast with the algorithms in the ro/rw model where the output was initially free, we start here with a full output. Therefore, the techniques must be completely different and cannot rely on the output space to serve as work space. One of our main techniques is to use pre- and post-additions on the outputs to *distribute* an intermediate result at different places. Assume that we need to compute $y \mathrel{+}= f(x)$ and $z \mathrel{+}= f(x)$ for some f. A standard algorithm would compute $f(x)$ once and store it into a temporary space, before adding it to y and z. In our case, assuming we know an algorithm to perform $y \mathrel{+}= f(x)$, we pre-subtract y from z ($z \mathrel{-}= y$), compute $y \mathrel{+}= f(x)$, and post-add y to z ($y \mathrel{+}= z$). More generally, our techniques can be viewed as transforming some algorithms to their *reversible* counterparts where some or all of the algorithm can be *undone*.

This kind of transformations can be applied automatically on any *bilinear* algorithm to derive a constant-space variant of it, at least in a nonuniform model. In many cases, the transformation can be applied uniformly to produce a constant-space uniform algorithm. Actually, the space is not fully constant. For recursive algorithms, there exists *a priori* a call stack of (usually) logarithmic size, that amounts to $O(\log n)$ pointers.

In this chapter, we describe algorithms that ultimately lead to a fast remainder algorithm and a fast modular multiplication algorithm with constant algebraic space. The series of reductions is summarized in Figure 4.3 on p. 50. Some algorithms for full products can be obtained automatically, and the ones that we present are optimized variants. The technique for such an automatization is postponed to Chapter 5, where some consequences in linear algebra are also presented.

4.1 Cumulative full products

As usual in computer algebra, the building block is polynomial multiplication. The goal is to perform $h \mathrel{+}= f \times g$ in constant space. Contrary to what we had in the ro/rw model, we do not provide a generic reduction from a standard multiplication algorithm, but rather revisit the main polynomial multiplication algorithms.

4.1.1 *Karatsuba's algorithm*

Let f and g of size n, and $h = f \times g$. Write $f = f_0 + x^m f_1$ and $g = g_0 + x^m g_1$. Karatsuba's algorithm expresses the product $h = f \times g$ as

$$h = f_0 g_0 + x^{2m} f_1 g_1 + x^m (f_0 g_0 + f_1 g_1 - (f_0 - f_1) \times (g_0 - g_1))$$

which leads to a recursive algorithm with three recursive calls, and a running time $O(n^{\log 3})$ [135].

To make it cumulative and constant-space, the first remark is that it is easy to compute $f_0 - f_1$ and $g_0 - g_1$ in the input space, since it can be easily undone. The main challenge is to compute only once $f_0 g_0$ and $f_1 g_1$ and *distribute* them at two distinct locations. Algorithm 4.1.1 uses pre- and post-additions to this end. Its data movements are represented in Figure 4.1.

Algorithm 4.1.1. CumulativeKaratsuba

Inputs: $f, g \in \mathsf{R}[x]$ of size n, $h \in \mathsf{R}[x]$ of size $2n-1$ *read-write*
Output: $h \mathrel{+}= f \times g$
Notations: $m = \lceil n/2 \rceil$ and write $f = f_0 + x^m f_1$, $g = g_0 + x^m g_1$

1 if $n < 2$: $h \mathrel{+}= f \times g$ constant space
2 $h_{[m,2m[} \mathrel{-}= h_{[0,m[}$; $h_{[2m,3m[} \mathrel{-}= h_{[m,2m[}$; $h_{[3m,2n-1[} \mathrel{-}= h_{[2m,2n-1-m[}$
3 $h_{[0,2m-1[} \mathrel{+}= f_0 \times g_0$ recursive call
4 $h_{[m,3m-1[} \mathrel{+}= f_1 \times g_1$ recursive call
5 $h_{[3m,2n-1[} \mathrel{+}= h_{[2m,2n-1-m[}$; $h_{[2m,3m[} \mathrel{+}= h_{[m,2m[}$; $h_{[m,2m[} \mathrel{+}= h_{[0,m[}$
6 $f_0 \mathrel{-}= f_1$; $g_0 \mathrel{-}= g_1$
7 $h_{[m,3m-1[} \mathrel{-}= f_0 \times g_0$ recursive call
8 $f_0 \mathrel{+}= f_1$; $g_0 \mathrel{+}= g_1$

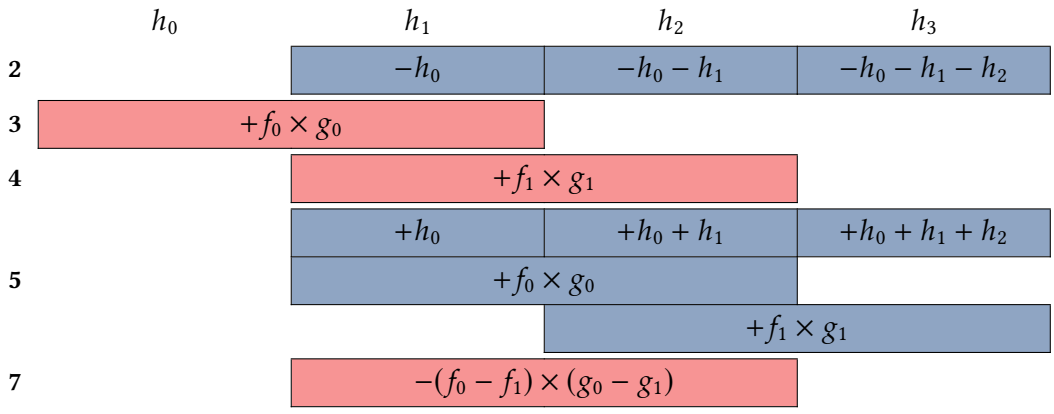


Figure 4.1. Data movements in Algorithm 4.1.1, where $h_0 = h_{[0,m[}$, $h_1 = h_{[m,2m[}$, $h_2 = h_{[2m,3m[}$ and $h_3 = h_{[3m,2n-1[}$. In blue, the pre- and post-additions; in red, the results of the recursive calls.

Theorem 4.1.1 ([C10, M2]). *Algorithm 4.1.1 (CumulativeKaratsuba) is correct, requires $O(n^{\log 3})$ operations, and only uses a call stack of $O(\log n)$ pointers as extra space.*

The same algorithm can be extended to the case where f and g do not have the same size.

Corollary 4.1.2. *Given three polynomials f, g and h of respective sizes m, n and $m+n-1$, $m \geq n$, one can compute $h \mathrel{+}= f \times g$ in $O(mn^{\log 3-1})$ operations, using a call stack of $O(\log n)$ pointers.*

4.1.2 FFT/TFT-based algorithm

The FFT-based algorithms are well-suited to perform a cumulative full product. Given f and g of size n and h of size $2n-1$, one can compute three size-$(2n-1)$ FFTs of f, g and h to get $\hat{f}$, $\hat{g}$ and $\hat{h}$ such that $\hat{f}_i = f(\omega^i)$ where ω is a $(2n-1)$st principal root of unity, and similarly for $\hat{g}$ and $\hat{h}$. Then, $\hat{h}_i \mathrel{+}= \hat{f}_i \hat{g}_i$ for $0 \leq i < 2n$ followed by an inverse FFT provides $h \mathrel{+}= fg$.

To make this algorithm constant-space, we can use the fact that the original FFT algorithm works *in place*, that is a size-n polynomial f is replaced by its size-n FFT, in time $O(n \log n)$ [54]. This *a priori* only covers cases where n is a power of two, but it has been shown that the truncated Fourier transform (TFT) [96] and its inverse can also be computed in place [7, 55, 92, 190]. Yet, one difficulty remains. A size-$(2n-1)$ TFT of f and g is needed while they only have size n. The solution is to compute these TFTs in several steps, restoring the initial f and g between two steps. Note that without permutation, a *decimation-in-frequency* FFT algorithm computes the *bit-reversed* DFT of the input polynomial, that is $(f(\omega^{[0]_k}), f(\omega^{[1]_k}), \ldots, f(\omega^{[2^k-1]_k}))$, where $[i]_k = \sum_{j=0}^{k-1} d_j 2^{k-j-1}$ is the length-k bit-reversal of $i = \sum_{j=0}^{k-1} d_j 2^j$, $d_j \in \{0,1\}$.

Algorithm 4.1.2. PartialFourierTransform

Inputs: f of size n, two integers k and ℓ such that $2^\ell \leq n$ *read-write*
a principal 2^pth root of unity ω, with $(k+1)2^\ell \leq 2^p$
Output: the first 2^ℓ coefficients of f replaced by $f(\omega^{[k\cdot 2^\ell+i]_p})$, $0 \leq i < 2^\ell$

1 for $i = 0$ to $n-1$: $f_{[i]} \mathrel{*}= \omega^{i[k\cdot 2^\ell]_p}$
2 for $i = 2^\ell$ to $n-1$: $f_{[i-2^\ell]} \mathrel{+}= f_i$
3 $f_{[0,2^\ell[} := \mathrm{FFT}(f_{[0,2^\ell[}, \omega^{2^{p-\ell}})$

This algorithm is easily inverted by undoing each operation. From this partial Fourier transform and its inverse denoted PartialFourierTransform^{-1}, we can build a cumulative FFT-based multiplication algorithm.

Theorem 4.1.3 ([C10, M2]). *Algorithm 4.1.3 is correct, uses $O(n \log n)$ operations, and uses no extra space.*

Algorithm 4.1.3. CumulativeFFTMultiplication

Inputs: f, g, h of size m, n and $m+n-1$ respectively, $m \le n$ *read-write*
a principal 2^pth root of unity ω, where $p = \lceil \log(m+n-1) \rceil$
Output: $h \mathrel{+}= f \times g$

1 $h := \mathrm{TFT}(h, \omega)$ h replaced by $(h(1), h(\omega), \ldots, h(\omega^{m+n-2}))$
  let $r = m+n-1$
2 while $r > 0$:
3 let $\ell = \lfloor \log \min(r, m) \rfloor$ and $t = \lfloor \log \min(r, n) \rfloor - \ell$
4 let $k = m+n-1-r$
5 $g := \textsc{PartialFourierTransform}_{k,\ell+t}(g, \omega)$
6 for $s = 0$ to $2^t - 1$:
7 $f := \textsc{PartialFourierTransform}_{s+k\cdot 2^t,\ell}(f, \omega)$
8 for $i = 0$ to $2^\ell - 1$: $h_{[i+(k\cdot 2^t+s)\cdot 2^\ell]} \mathrel{+}= a_{[i]} b_{[i+s\cdot 2^\ell]}$
9 $f := \textsc{PartialFourierTransform}^{-1}_{s+k\cdot 2^t,\ell}(f, \omega)$
10 $g := \textsc{PartialFourierTransform}^{-1}_{k,\ell+t}(g, \omega)$
11 let $r = r - 2^{\ell+t}$

4.2 Cumulative convolutions and short products

From a cumulative constant-space full product algorithm, we can actually derive cumulative convolutions $h \mathrel{+}= f \times g \bmod x^n - \lambda$, including the short product ($\lambda = 0$). We first assume that $\lambda \neq 0$. Assume for simplicity that n is even, and $\lambda = 1$. Writing $f = f_0 + x^m f_1$ and $g = g_0 + x^m g_1$ for $m = n/2$, $fg = f_0 g_0 + x^m(f_0 g_1 + f_1 g_0) + x^n f_1 g_1$. Computing modulo $x^n - 1$, $x^n f_1 g_1 \equiv f_1 g_1$. Similarly, $f_0 g_1 \equiv (f_0 g_1)_{[m,n[} + x^m (f_0 g_1)_{[0,m[}$ (and the same holds for $f_1 g_0$). Therefore, given a cumulative full product algorithm, we can easily compute $fg \bmod x^n - 1$ by writing these subproducts in the right registers.

Algorithm 4.2.1 generalizes the approach to any unit λ, and any n. It is illustrated in Figure 4.2. Note that a Karatsuba-like approach decreases the number of calls to the full product subroutine to three instead of four [M2].

The case $\lambda = 0$ is to be treated differently. The subproduct $f_1 \times g_1$ is unneeded. But the lower parts of the two cross products $f_0 g_1$ and $f_1 g_0$ must be computed. This could be done using recursive calls, but the algorithm would make two recursive calls. This would add an extra logarithmic factor to the complexity when $\mathsf{M}(n)$ is quasi-linear, and would require a call stack.

Write $f = f_0 + x^m f_1$ and $g = g_0 + x^m g_1$. Our goal is to compute $h \mathrel{+}= f \times g \bmod x^n$. We can expand $f \times g \bmod x^n = f_0 g_0 + x^m (f_0 g_1 + f_1 g_0 \bmod x^{n-m})$. The first product $f_0 g_0$ is a full product, but $f_0 g_1$ and $f_1 g_0$ must be recursive calls. If we rewrite the expansion in a slightly unnatural way as

$$f \times g \bmod x^n = (f_0(g_0 - g_1) + f_0 g_1) + x^m (f_0 g_1 + f_1 g_0 \bmod x^{n-m})$$

we can make only one recursive call to compute $f_1 g_0 \bmod x^{n-m}$, and compute the full product $f_0 g_1$ before distributing it.

This is Algorithm 4.2.2, illustrated in Figure 4.2. We can actually improve the leading constant in the complexity of this algorithm by using different formulas [M2].

Algorithm 4.2.1. CumulativeConvolution

Inputs: f, g, h of size n, $\lambda \in \mathrm{R}^\times$ — *read-write*
Output: $h \mathrel{+}= f \times g \bmod x^n - \lambda$
Required: a cumulative constant-space full product algorithm
Notations: $m = \lceil n/2 \rceil$ and $s = n \bmod 2$
write $f = f_0 + x^m f_1$ and $g = g_0 + x^m g_1$

1 $h_{[0,2m-1[} \mathrel{+}= f_0 \times g_0$
2 $h \mathrel{/}= \lambda$
3 $h_{[s,n-1[} \mathrel{+}= f_1 \times g_1$
4 $h_{[m,n[} \mathrel{*}= \lambda$
5 $h_{[m,n[} \| h_{[0,m[} \mathrel{+}= f_0 \times g_1$ — $\cdot\|\cdot$ denotes concatenation
6 $h_{[m,n[} \| h_{[0,m[} \mathrel{+}= f_1 \times g_0$
7 $h_{[0,m[} \mathrel{*}= \lambda$

Algorithm 4.2.2. CumulativeLowerProduct

Inputs: f, g, h of size n — *read-write*
Output: $h \mathrel{+}= f \times g \bmod x^n$
Required: a cumulative constant-space full product algorithm
Notations: $m = \lceil n/2 \rceil$
write $f = f_0 + x^m f_1$ and $g = g_0 + x^m g_1$ and $g_0^* = g_0 \bmod x^{n-m}$

1 if $n < 2$: $h \mathrel{+}= f \times g \bmod x^n$ — constant space
2 $g_0 \mathrel{-}= g_1$; $h_{[0,n[} \mathrel{+}= f_0 \times g_0$; $g_0 \mathrel{+}= g_1$ — $f_0(g_0 - g_1)$
3 $h_{[m,n[} \mathrel{-}= h_{[0,n-m[}$; $h_{[0,n[} \mathrel{+}= f_0 \times g_1$; $h_{[m,n[} \mathrel{+}= h_{[0,n-m[}$ — $f_0 g_1$
4 $h_{[m,n[} \mathrel{+}= f_1 \times g_0^* \bmod x^{n-m}$ — tail recursive call

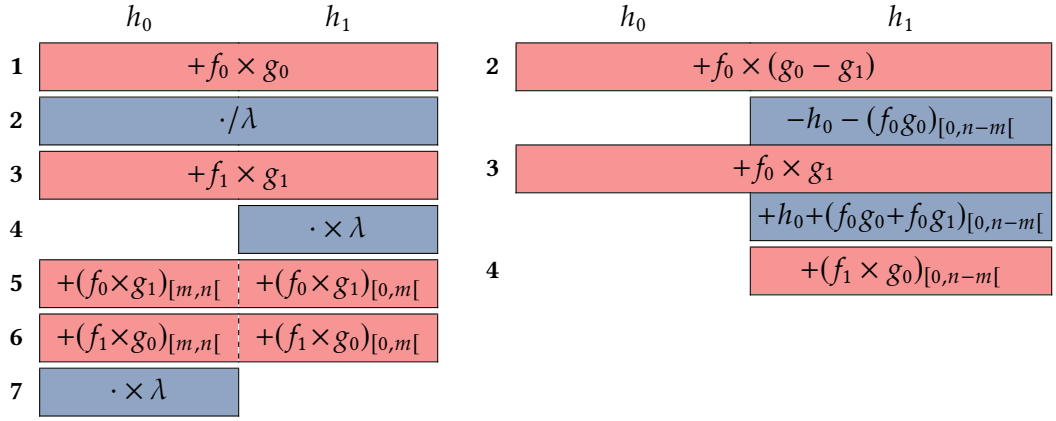


Figure 4.2. Illustration of Algorithms 4.2.1 (left) and 4.2.2 (right). Pre- and post-operations are in blue, and products in red.

Theorem 4.2.1 ([C11, M2]). *Algorithm 4.2.2 (CumulativeLowerProduct) is correct, requires no extra space, and runs in time $O(\mathsf{M}(n))$.*

As a consequence of this algorithm, we can actually compute any *slice* of a polynomial product, in constant space with accumulation. The first remark is that we can compute an upper product with the same algorithm, used in reversed mode. Then, for a size-m polynomial f and a size-n polynomial g, the *slice* $[f \cdot g]_s^t$ for $0 \le s < t \le m+n$ can be decomposed as a sum of lower and upper products. The nature of cumulative algorithms make their sequential composition straightforward.

Corollary 4.2.2. *Let f, g, h of respective sizes m, n, and r where $0 < r < m+n$, and s such that $0 \le s < m+n-r$. Then $h \mathrel{+}= [f \cdot g]_s^{r+s}$ can be computed with no extra space in time $O(\mathsf{M}(r))$.*

4.3 In-place power series computations

We consider two power series ϕ and ψ, and the operations of multiplication $\phi \times \psi$ and division ϕ/ψ. By *in-place* we mean that we aim to replace (the truncation of) ϕ by (the truncation of) $\phi \times \psi$ or ϕ/ψ. The first operation corresponds to a polynomial lower product.

Let $f = \phi \bmod x^n$, $g = \psi \bmod x^n$ and $h = \phi \times \psi \bmod x^n$. Write $f = f_0 + x^k f_1$, $g = g_0 + x^k g_1$ and $h = h_0 + x^k h_1$ where $k = \lceil n/2 \rceil$. Then $h_0 = f_0 \times g_0 \bmod x^k$, and $h_1 = [f_0 \times g]_k^n + (f_1 \times g_0 \bmod x^{n-k})$. This yields the following algorithm that replaces f by h, by first computing h_1. Due to the two recursive calls needed to compute h_0 and h_1, the complexity becomes $O(\mathsf{M}^*(n))$.

Algorithm 4.3.1. InPlaceLowerProduct

Inputs: $f = \phi \bmod x^n$ and $g = \psi \bmod x^n$ — *read-write*
Output: $f = \phi \times \psi \bmod x^n$
Notation: $k = \lceil n/2 \rceil$

1	if $n < 2$: $f \mathrel{*}= g \bmod x^n$	constant space
2	$f_{[k,n[} \mathrel{*}= g_{[0,n-k[} \bmod x^{n-k}$	recursive call
3	$f_{[k,n[} \mathrel{+}= [f_{[0,k[} \times g_{[1,n[}]_k^n$	Corollary 4.2.2
4	$f_{[0,k[} \mathrel{*}= g_{[0,k[} \bmod x^k$	recursive call

We now turn to power series division. A solution to compute φ/ψ in place could be to first invert ψ in place, and then to multiply φ by ψ^{-1} in place (replacing one or the other of the series). Unfortunately, it is not clear that in-place inversion is possible. Instead, the solution is to directly use Newton iteration with φ/ψ. With the same notations as before, the algorithm is: $h_0 := f_0/g_0 \bmod x^k$; $h_1 := (f_1 - [g \times h_0]_k^n)/g_0 \bmod x^{n-k}$. This is exactly the inverse algorithm of Algorithm 3.1.2 (LowerProduct).

Algorithm 4.3.2. InPlaceDivision

Inputs: $f = \phi \bmod x^n$ and $g = \psi \bmod x^n$ — *read-write*
Output: $f = \phi/\psi \bmod x^n$
Notation: $k = \lceil n/2 \rceil$

1 if $n < 2$: $f \mathrel{/}= g \bmod x^n$ — constant space
2 $f_{[0,k[} \mathrel{/}= g_{[0,k[} \bmod x^k$ — recursive call
3 $f_{[k,n[} \mathrel{-}= [f_{[0,k[} \times g_{[1,n[}]_k^n$ — Corollary 4.2.2
4 $f_{[k,n[} \mathrel{/}= g_{[0,n-k[} \bmod x^{n-k}$ — recursive call

Theorem 4.3.1 ([C11, M2]). *Algorithm 4.3.1 (InPlaceLowerProduct) and Algorithm 4.3.2 (InPlaceDivision) are correct, require $O(\mathsf{M}^*(n))$ operations, and only use a call stack of $O(\log n)$ pointers as extra space.*

The second algorithm will be used mainly in *reversed mode* in the sense of Proposition 1.7.3, that is to compute $f^{\leftarrow} \mathrel{/}= g^{\leftarrow} \bmod x^n$ where $f = \phi \bmod x^n$ and $g = \psi \bmod x^n$. This algorithm is called a *reversed power series division algorithm*, or in short *reversed division*.

As mentioned above, this approach does not compute the inversion in place. It is not even clear that there exists an algorithm, even a slow one, for this task.

Open problem 4.1. Given $f = \phi \bmod x^n$, invertible, is it possible to compute $f := \phi^{-1} \bmod x^n$ in place?

4.4 Remainder computation

To get a constant-space remainder algorithm, the idea is to start from Algorithm 3.2.5 (SmallSpaceRemainder) and replace the basic operations by their in-place variants. As a result, the algorithm has complexity $O(\frac{m}{n}\mathsf{M}^*(n))$ rather than $O(\frac{m}{n}\mathsf{M}(n))$ due to the use of in-place building blocks that have complexity $O(\mathsf{M}^*(n))$.

Theorem 4.4.1 ([C11, M2]). *Algorithm 4.4.1 (Remainder) is correct, requires $O(\frac{m}{n}\mathsf{M}^*(n))$ operations, and only uses a call stack of $O(\log n)$ pointers as extra space.*

As a variant of the preceding algorithm, it is actually possible to *replace* the input f by the quotient and the remainder, still using no extra space. An important remark is that the obtained algorithm is *reversible*, that is it can be inverted to restore f from the quotient and the remainder.

Theorem 4.4.2 ([C10, M2]). *Algorithm 4.4.2 (InPlaceEuclideanDivision) is correct, requires $O(\frac{m}{n}\mathsf{M}^*(n))$ operations, and uses only a call stack of $O(\log n)$ pointers as extra space. Further, it is reversible and the computation can be undone within the same time and space complexity bounds.*

Algorithm 4.4.1. Remainder

Inputs: f of size $m+n-1$, g of size n, $m \geq n-1$ *read-write*
Output: r of size $n-1$ such that $r = f \bmod g$
Required: In-place lower product and reversed division algorithms
Notations: $k = \lfloor m/n \rfloor$, $\ell = m \bmod n$, $g^* = g \operatorname{quo} x$ and $g_* = g \bmod x^{n-1}$

```
1 r := f_[m+n-1-ℓ,m+n-1[
2 for j = k − 1 to 1:
3     r^← /= (g^*)^← mod x^{n-1}        in-place reversed division
4     r *= −g_* mod x^{n-1}             in-place lower product
5     r += f_[jn-1,(j+1)n-1[
6 r^← /= (g^*)^← mod x^{n-1}            in-place reversed division
7 r *= −g_* mod x^{n-1}                 in-place lower product
8 r += f_[0,n-1[
```

Algorithm 4.4.2. InPlaceEuclideanDivision

Inputs: f of size $m+n-1$, g of size n *read-write*
Output: $f_{[0,n[} := r$ and $f_{[n,n+m-1[} := q$ s.t. $f = bq + r$, $\deg(q) < m$, $\deg(r) < n-1$
Required: Cumulative lower product and in-place reversed division algorithms
Notations: $k = \lfloor m/n \rfloor$, $\ell = m \bmod n$, $g^* = g \bmod x$ and $g_* = g \bmod x^{n-1}$
write $f = \sum_{i=0}^{k-1} f_i x^{ni}$, with $\deg(f_{k-1}) = \ell$

```
1 f_{k-1}^← /= g_[n-ℓ,n[^← mod x^ℓ      in-place reversed division
2 f_{k-2} −= g_* × f_{k-1} mod x^n      cumulative lower product
3 for i = k − 2 to 1:
4     f_i^← /= (g^*)^← mod x^n          in-place reversed division
5     f_{i-1} −= g_* × f_i mod x^n      cumulative lower product
```

A consequence of the reversibility is that one can use this algorithm to perform some cumulative remainder computation $r \mathrel{+}= f \bmod g$. First, replace f by the remainder and quotient $[f \bmod g, f \operatorname{quo} g]$; Then compute $r \mathrel{+}= f \bmod g$; Finally, restore f.

Corollary 4.4.3. *Given f of size $m+n-1$, g of size n and r of size $n-1$, one can compute $r \mathrel{+}= f \bmod g$ in $O(\frac{m}{n}\mathsf{M}^*(n))$ operations, with a call stack of $O(\log n)$ pointers.*

4.5 Modular product

Our last goal is to extend the previous results to the modular product. More precisely, given f, g, p and r, the goal is to compute $r \mathrel{+}= f \times g \bmod p$. One example of such a computation is the multiplication is finite field extensions, where p defines the extension. To stick with this example, let us assume that f, g and r have the same size n and p has

size $n+1$.

Let $h = f \times g$ of size $2n-1$, and write $h = h_0 + x^n h_1$ where h_0 has size n and h_1 has size $n-1$. Then $h \bmod p = h_0 + (x^n h_1 \bmod p)$. Adding h_0 to r is easy and only requires a constant-space cumulative lower product algorithm. To compute $x^n h_1 \bmod p$, we first compute the size-$(n-1)$ quotient as $q^{\leftarrow} = h_1^{\leftarrow}/(p^*)^{\leftarrow} \bmod x^{n-1}$ where $p^* = p \operatorname{quo} x^2$. Then we get $x^n h_1 \bmod p$ as $(x^n h_1 - q \times p) \bmod x^n = -(q \times p) \bmod x^n$. Since h is known only as $f \times g$, we first have to compute h_1 into the higher-degree $n-1$ coefficients f^* of f as $f^* \mathrel{*}= g^* \operatorname{quo} x^{n-1}$ using an in-place upper product. This computation can be undone as $(f^*)^{\leftarrow} \mathrel{/}= (g^*)^{\leftarrow} \bmod x^{n-1}$ using an in-place reversed division.

Algorithm 4.5.1. ModularMultiplication

Inputs: f, g, r of size n, p of size $n+1$ *read-write*
Output: $r \mathrel{+}= f \times g \bmod p$
Required: Cumulative lower product algorithm
In-place upper product and reversed division algorithms
Notations: $f^* = f_{[1,n[}$, $g^* = g_{[1,n[}$, $p^* = p_{[2,n+1[}$, $p_* = p_{[0,n[}$
$f_*^{\leftarrow} = f_{[1,n[}^{\leftarrow}$, $g_*^{\leftarrow} = g_{[1,n[}^{\leftarrow}$, $p_*^{\leftarrow} = p_{[2,n+1[}^{\leftarrow}$,

1. $r \mathrel{+}= f \times g \bmod x^n$ — $r \mathrel{+}= h_0$, cumulative lower product
2. $f^* \mathrel{*}= g^* \operatorname{quo} x^n$ — in-place upper product
3. $f_*^{\leftarrow} \mathrel{/}= p_*^{\leftarrow} \bmod x^{n-1}$ — in-place reversed division
4. $r \mathrel{-}= f^* \times p_* \bmod x^{n-1}$ — $r \mathrel{+}= x^n h_1 \bmod p$, cumulative lower product
5. $f^* \mathrel{*}= p^* \operatorname{quo} x^n$ — undo 3, in-place upper product
6. $f_*^{\leftarrow} \mathrel{/}= g_*^{\leftarrow} \bmod x^{n-1}$ — undo 2, in-place reversed division

Theorem 4.5.1 ([C11, M2]). *Algorithm 4.5.1 (ModularMultiplication) is correct, requires $O(\mathsf{M}^*(n))$ operations, and only uses a call stack of size $O(\log n)$ as extra space.*

Although the example of the multiplication in finite field extension assumes that f, g and r have size n and p has size $n+1$, the reversible in-place Euclidean division algorithm makes it possible to extend the result to any sizes.

Algorithm 4.5.2. ModularMultiplicationAllSizes

Inputs: f, g, r and p of respective sizes ℓ, m, n and $n+1$ *read-write*
Output: $r \mathrel{+}= f \times g \bmod p$
Required: In-place Euclidean division algorithm and its inverse
Cumulative modular multiplication algorithm
Notations: $q_f = f \operatorname{quo} p$, $r_f = f \bmod p$, $q_g = g \operatorname{quo} p$, $r_g = g \bmod p$

1. $f := [r_f, q_f]$ — only if $\ell > n$, in-place Euclidean division
2. $g := [r_g, q_g]$ — only if $m > n$, in-place Euclidean division
3. $r \mathrel{+}= f_{[\ell-n,\ell[} \times g_{[m-n,n[} \bmod p$ — cumulative modular multiplication
4. $g := q_g \times p + r_g$ — undo 2 if $m > n$, inverted Euclidean division
5. $f := q_f \times p + r_f$ — undo 1 if $\ell > n$, inverted Euclidean division

Theorem 4.5.2 ([C10, M2]). *Algorithm 4.5.2 (ModularMultiplicationAllSizes) is correct, requires $O(\frac{\ell+m}{n}\mathsf{M}^*(n))$ operations, and only uses a call stack of size $O(\log n)$ as extra space.*

For the case of multiplication in finite field extensions, the modulus is often chosen to be sparse. Our algorithms can be analyzed in this case and the complexities refined [M2].

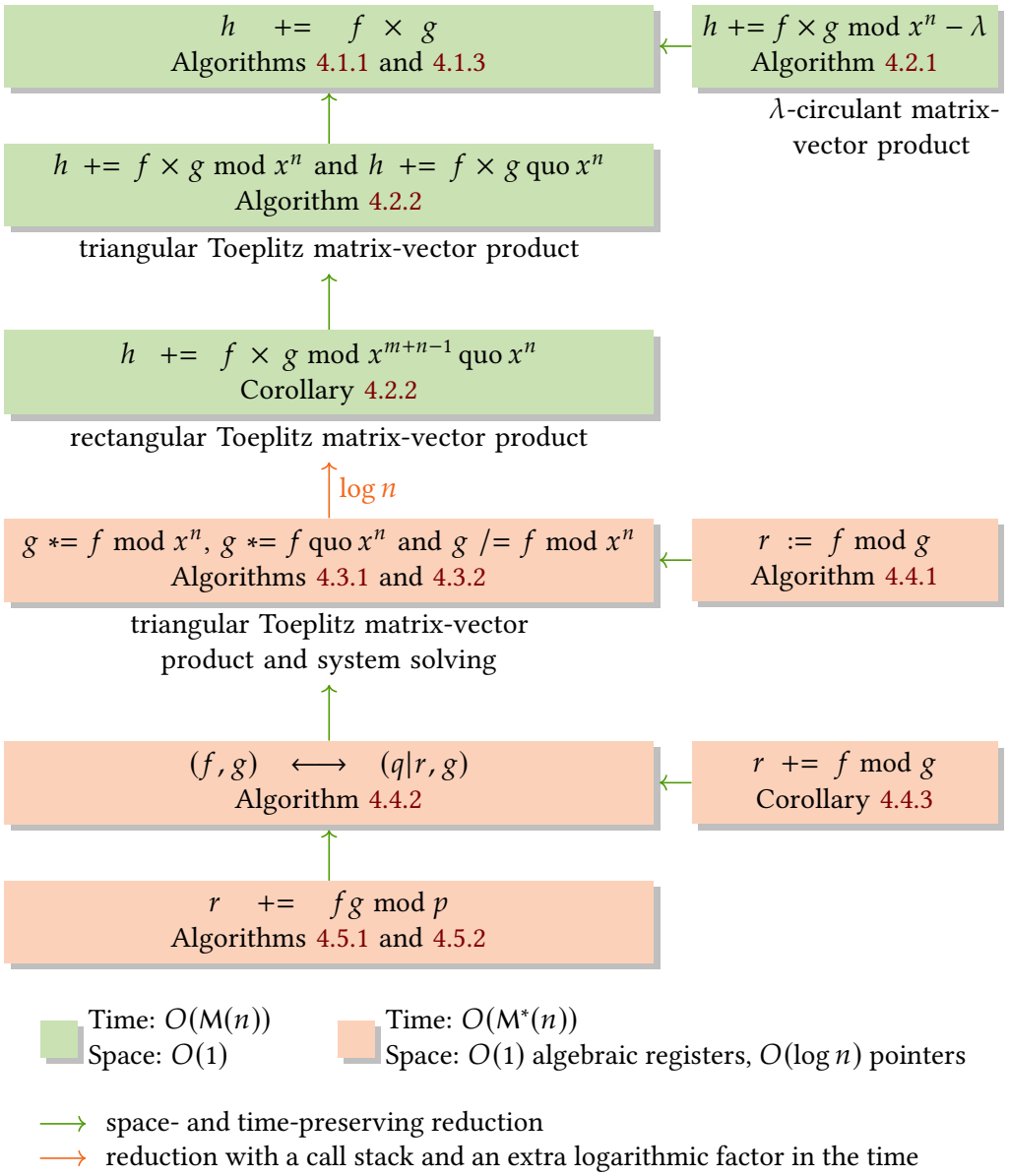


Figure 4.3. Summary of the reductions of the chapter.

5 The automatic approach for algorithms in the rw/rw model

The algorithms described in Section 4.1 have been obtained semi-automatically. More precisely, they are improved version of algorithms that were obtained automatically. This chapter is devoted to this automatization. We present algorithms that produce constant-space variants of standard algorithms with similar complexity. It uses the framework of bilinear algorithms, that can be formalized by a direct extension of linear programs (Definition 1.7.1) to *bilinear programs*. A C++ implementation of these algorithms and (heuristic) optimizations is available in the PLinOpt library [S1].

Section 5.2 applies this automatization to linear-algebraic problems.

5.1 The general framework

Definition 5.1.1. A *cumulative bilinear algorithm* is given by a triple (A, B, C) where $A \in \mathsf{R}^{t\times m}$, $B \in \mathsf{R}^{t\times n}$ and $C \in \mathsf{R}^{s\times t}$ such that neither A nor B nor C contains an all-zero row. Given $\boldsymbol{x} \in \mathsf{R}^m$, $\boldsymbol{y} \in \mathsf{R}^n$ and $\boldsymbol{z} \in \mathsf{R}^s$, it computes $\boldsymbol{z}$ += $C((A\boldsymbol{x}) \odot (B\boldsymbol{y}))$ where $\odot$ is the component-wise product, that is

$$z_k \mathrel{+}= \sum_{\ell=0}^{t-1} C_{k,\ell} \left[\left(\sum_{i=0}^{m-1} A_{\ell,i} x_i \right) \cdot \left(\sum_{j=0}^{n-1} B_{\ell,j} y_j \right) \right]$$

for $0 \le k < s$.

From the representation (A, B, C) of a cumulative bilinear algorithm, an actual sequence of operations to compute $\boldsymbol{z}$ += $C((A\boldsymbol{x}) \odot (B\boldsymbol{y}))$ is easily obtained. Yet, it requires some extra space to store $A\boldsymbol{x}$ and $B\boldsymbol{y}$, and then their component-wise product. Using techniques of pre- and post-additions as used in Chapter 4, we can obtain a constant space algorithm. To avoid any unneeded technicalities, we assume that the input matrices have entries in a field. Algorithm 5.1.1 is implemented as the tool `inplacer` of PLinOpt [S1].

A cumulative bilinear algorithm has three kinds of operations: additions, multiplications and *scalar* multiplications where one multiplicand is a constant from A, B or C. Let $\sigma(M)$ and $\tau(M)$ denote the number of nonzero entries and the number of entries $\notin \{0, 1, -1\}$ in M, respectively.

Theorem 5.1.2 ([C10, M2]). *Given a cumulative bilinear algorithm $(A, B, C) \in \mathsf{R}^{t\times m} \times \mathsf{R}^{t\times n} \times \mathsf{R}^{s\times t}$, Algorithm 5.1.1 (ConstantSpaceBilinear) computes $\boldsymbol{z}$ += $C((A\boldsymbol{x}) \odot (B\boldsymbol{y}))$ in constant space using t cumulative products z_k += $x_i y_j$, $2(\sigma(A) + \sigma(B) + \sigma(C)) - 5t$ additions, and $2(\tau(A) + \tau(B) + \tau(C))$ in-place scalar multiplications, without any further copy.*

Algorithm 5.1.1. ConstantSpaceBilinear

Inputs: $A \in \mathsf{K}^{t\times m}$, $B \in \mathsf{K}^{t\times n}$, $C \in \mathsf{K}^{s\times t}$ *read-only*
$\boldsymbol{x} \in \mathsf{K}^m$, $\boldsymbol{y} \in \mathsf{K}^n$, $\boldsymbol{z} \in \mathsf{K}^s$ *read-write*
Output: $\boldsymbol{z}$ += $C((A\boldsymbol{x}) \odot (B\boldsymbol{y}))$

1 for $u = 0$ to $t-1$:
2 $\quad x_i := \sum_{\ell=0}^{m} A_{u,\ell} x_\ell$ for some i s.t. $A_{u,i} \neq 0$
3 $\quad y_j := \sum_{\ell=0}^{n} B_{u,\ell} y_\ell$ for some j s.t. $B_{u,j} \neq 0$
4 $\quad z_k$ /= $C_{k,u}$ for some k s.t. $C_{k,u} \neq 0$
5 $\quad z_\ell$ −= $C_{\ell,u} z_k$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
6 $\quad z_k$ += $x_i \cdot y_j$
7 $\quad z_\ell$ += $C_{\ell,u} z_k$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
8 $\quad z_k$ *= $C_{k,u}$
9 $\quad y_j := (y_j - \sum_{\ell\neq j} B_{u,\ell} y_\ell)/B_{u,j}$
10 $\quad x_i := (x_i - \sum_{\ell\neq i} A_{u,\ell} x_\ell)/A_{u,i}$

Without space constraint, a cumulative bilinear algorithm performs $(\sigma(A) - m) + (\sigma(B) - n) + (\sigma(C) - t) + s$ additions, t multiplications and $(\tau(A) + \tau(B) + \tau(C))$ scalar multiplications. The number of operations is less than doubled by the constant-space variant. While the number of operations without space constraint is exact, the constant-space algorithm can be optimized by a clever scheduling of its operations.

Note that this result also applies to *recursive* algorithm, where the products z_k += $x_i y_j$ are replaced by recursive calls. This is in particular the case for matrix multiplication, as shown in Section 5.2. The same technique cannot be directly applied to (recursive) polynomial multiplication algorithms such as Karatsuba's algorithm. The problem is that the output size of a recursive call is not the same as the input size.

If we consider the product *à la* Karatsuba of two size-2 polynomials $f = f_0 + f_1 x$ and $g = g_0 + g_1 x$, the result $h = h_0 + h_1 x + h_2 x^2$ can be computed as $h_0 = f_0 g_0$, $h_1 = f_0 g_1 - f_1 g_0$ and $h_2 = f_0 g_0 + f_1 g_1 - (f_0 - f_1)(f_0 - g_1)$. Therefore, we obtain the three matrices

$$A = B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{pmatrix} \text{ and } C = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Algorithm 5.1.1 (ConstantSpaceBilinear) produces a Karatsuba-like constant-space algorithm with only three multiplications for the product of two size-2 polynomials. The problem is the extension to a recursive algorithm. Assume that f and g have size $n = 2k$. We need to replace the three products by recursive calls to size-k polynomial products. But the result of the recursive calls is a size-$(2k-1)$ polynomial, so that there is some overlap between for instance h_0 and h_1.

To formalize the extension, we consider a two-dimensional version of bilinear algorithms.

Definition 5.1.3. Let $\circ : \mathsf{R} \times \mathsf{R} \longrightarrow \mathsf{R}^2$ be an operator, and $\circledcirc : \mathsf{R}^t \times \mathsf{R}^t \longrightarrow \mathsf{R}^{t+1}$ be its component-wise extension defined by $\boldsymbol{u} \circledcirc \boldsymbol{v} = \boldsymbol{w}$ where $w_i = (u_{i-1} \circ v_{i-1})_{[1]} + (u_i \circ v_i)_{[0]}$ for $1 \leq i < t$, $w_0 = (u_0 \circ v_0)_{[0]}$ and $w_t = (u_{t-1} \circ v_{t-1})_{[1]}$. A *2D-cumulative*

bilinear algorithm is given by a (A, B, C) where $A \in \mathsf{R}^{t\times m}$, $B \in \mathsf{R}^{t\times n}$ and $C \in \mathsf{R}^{s\times(t+1)}$. Given $\boldsymbol{x} \in \mathsf{R}^m$, $\boldsymbol{y} \in \mathsf{R}^n$ and $\boldsymbol{z} \in \mathsf{R}^s$, it computes $\boldsymbol{z}$ += $C((A\boldsymbol{x}) \circledcirc (B\boldsymbol{y}))$.

Algorithm 5.1.1 (ConstantSpaceBilinear) can be extended to 2D-bilinear algorithms. The operations on the inputs remain the same. Assume that some result $\pi_u = (\sum_i A_{u,i}x_i) \circ \left(\sum_j B_{u,j}y_j\right)$ has to be distributed in two places: $\left[\begin{smallmatrix} z_k \\ z_{k+1} \end{smallmatrix}\right]$ += $C_{k,u}\pi_u$ and $\left[\begin{smallmatrix} z_\ell \\ z_{\ell+1} \end{smallmatrix}\right]$ += $C_{\ell,u}\pi_u$. If $\ell > k+1$, there is no overlap and we can proceed as in the standard bilinear case. We simply have to replace for instance z_k /= $C_{k,u}$ by $\left[\begin{smallmatrix} z_k \\ z_{k+1} \end{smallmatrix}\right]$ /= $C_{k,u}$, that is z_k /= $C_{k,u}$ and z_{k+1} /= $C_{k,u}$. Other operations on $\boldsymbol{z}$ can be dealt with similarly. But if $\ell = k+1$, there is an overlap. This forces to be careful in the order of the computations. Finally, we replace Lines 4 to 8 by the following ones.

4. z_k /= $C_{k,u}$ for some k s.t. $C_{k,u} \neq 0$
5. z_ℓ −= $C_{\ell,u}z_k$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
6. z_{k+1} /= $C_{k,u}$
7. $z_{\ell+1}$ −= $C_{\ell,u}z_{k+1}$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
8. $\left[\begin{smallmatrix} z_k \\ z_{k+1} \end{smallmatrix}\right]$ += $x_i \circ y_j$
9. $z_{\ell+1}$ += $C_{\ell,u}z_{k+1}$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
10. z_{k+1} *= $C_{k,u}$
11. z_ℓ += $C_{\ell,u}z_k$ for each $\ell \neq k$ s.t. $C_{\ell,u} \neq 0$
12. z_k *= $C_{k,u}$

This is implemented as the tool `trilplacer` of PLinOpt [S1].

This algorithm can be applied to any recursive polynomial multiplication algorithm. We note that to exactly fits within the framework, one has to define the operator $u_i \circ v_j = (0, u_i \cdot v_j)$, so that $\boldsymbol{u} \circledcirc \boldsymbol{v}$ corresponds to a polynomial product extended with a zero as leading coefficient since a polynomial product does not double the size of the input.

Algorithm 4.1.1 (CumulativeKaratsuba) can almost directly be obtained from the three matrices A, B and C using this algorithm. The actual algorithm removes some useless computations and in particular deals with the useless leading zeroes. The same technique can be used to get a constant-space variant of Toom-Cook algorithms.

5.2 Application to linear algebra

In this section, we investigate the implications of Theorem 5.1.2 for linear-algebraic computations. First, the theorem can be applied to matrix multiplication. Let us start with a 2×2 cumulative matrix multiplication Z += XY. Strassen-Winograd algorithm computes Z += XY using seven products rather than eight for a naive algorithm. The recursive extension of this algorithm to $n \times n$ matrices has time complexity $O(n^{\log 7})$ and requires $\Theta(n^2)$ temporary registers. The most space-efficient variants of this algorithm reduce the number of temporary registers to n^2 [114] or even $\frac{2}{3}n^2$ [34]. These results were in the `ro/rw` model. Using Theorem 5.1.2, we obtain a constant-space variant of this algorithm in the `rw/rw` model. To fit within the framework of Definition 5.1.1, a matrix $X = \left(\begin{smallmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{smallmatrix}\right)$ is viewed as the vector $\boldsymbol{x} = (x_{00}, x_{01}, x_{10}, x_{11})$. The resulting algorithm can be slightly optimized *by hand* to obtain an *optimal* variant.

Algorithm 5.2.1. ConstantSpaceStrassenWinograd

Inputs: $X, Y, Z \in \mathsf{K}^{n\times n}$ for some $n = 2^k$ *read-write*
Output: $Z \mathrel{+}= X \cdot Y$
Notations: $X = \left(\begin{smallmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{smallmatrix}\right), Y = \left(\begin{smallmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{smallmatrix}\right), Z = \left(\begin{smallmatrix} Z_{00} & Z_{01} \\ Z_{10} & Z_{11} \end{smallmatrix}\right)$ with blocks of size $\frac{n}{2}$

1 if $n = 1$: return $z_{00} \mathrel{+}= x_{00}y_{00}$
2 $X_{10} \mathrel{-}= X_{00}$; $Y_{01} \mathrel{-}= Y_{11}$; $Z_{10} \mathrel{-}= Z_{11}$
3 $Z_{11} \mathrel{+}= X_{10} \cdot Y_{01}$ recursive call
4 $X_{10} \mathrel{+}= X_{11}$; $Y_{01} \mathrel{-}= Y_{00}$; $Z_{01} \mathrel{-}= Z_{11}$
5 $Z_{11} \mathrel{-}= X_{10} \cdot Y_{01}$ recursive call
6 $Z_{00} \mathrel{-}= Z_{11}$
7 $Z_{11} \mathrel{+}= X_{00} \cdot Y_{00}$ recursive call
8 $Z_{00} \mathrel{+}= Z_{11}$; $Y_{01} \mathrel{+}= Y_{10}$; $Z_{10} \mathrel{+}= Z_{11}$
9 $Z_{10} \mathrel{+}= X_{11} \cdot Y_{01}$ recursive call
10 $Y_{01} \mathrel{+}= Y_{11}$; $Y_{01} \mathrel{-}= Y_{10}$; $X_{10} \mathrel{-}= X_{01}$
11 $Z_{01} \mathrel{-}= X_{10} \cdot Y_{11}$ recursive call
12 $X_{10} \mathrel{+}= X_{01}$; $X_{10} \mathrel{+}= X_{00}$
13 $Z_{11} \mathrel{+}= X_{10} \cdot Y_{01}$ recursive call
14 $Z_{01} \mathrel{+}= Z_{11}$; $Y_{01} \mathrel{+}= Y_{00}$; $X_{10} \mathrel{-}= X_{11}$
15 $Z_{00} \mathrel{+}= X_{01} \cdot Y_{10}$ recursive call

Theorem 5.2.1 ([C10, M2]). *Algorithm 5.2.1 (ConstantSpaceStrassenWinograd) is correct. If n is a power of two, it uses $8n^{\log 7} + O(n)$ operations and only uses a call stack of $O(\log n)$ pointers as extra space. Furthermore, the constant 8 is the best achievable for algorithms performing $O(n^{\log 7})$ operations with the same space complexity.*

It is well known that many linear-algebraic computations reduce to matrix multiplications. In particular, these reductions have been made space-efficient in [59, 60]. The only extra space required is due to the underlying matrix multiplications. Plugging our constant-space Strassen-Winograd algorithm provides fast constant-space algorithms for these tasks.

Corollary 5.2.2. *Let $M, T \in \mathsf{K}^{m\times m}$ and $A, B \in \mathsf{K}^{m\times n}$ where T is upper triangular, and let $\omega = \log 7$.*

i. The following task can be computed in time $O(m^{\omega-1}n)$ using a call stack of size $O(\log n)$ as only extra space:

- *$A := T \cdot A$ (TRMM);*
- *$A := T^{-1} \cdot A$ (TRSM);*
- *$M \mathrel{+}= \alpha A \cdot A^{\mathsf{T}}$ (SYRK);*
- *$\mathit{Low}(M) \mathrel{+}= \mathit{Low}(A \cdot B^{\mathsf{T}} + B \cdot A^{\mathsf{T}})$ (SYR2K), where $\mathit{Low}(M)$ denotes the lower triangular part of M.*

ii. The following tasks can be computed in time $O(m^{\omega})$ using a call stack of size $O(\log n)$ as only extra space:

- $T := T^{-1}$ *(INVT);*
- $N \mathrel{+}= M^2$ *(SQUARE).*

iii. The following tasks can be computed in time $O(m^\omega)$, using $O(n)$ pointers for storing permutations:[1]

- $A := \left[\begin{smallmatrix} L\backslash U & V \\ M & 0 \end{smallmatrix}\right]$ *(PLUQ), where $L\backslash U$ denotes the square matrix with lower part L and upper part U;*
- $M := M^{-1}$ *(INV).*

In these results, the asymptotic time complexity remains the same as for the original algorithms. In terms of space complexity, $O(\log n)$ pointers are needed in the call stack. The naive matrix multiplication algorithm only requires $O(1)$ pointers. For polynomial multiplication, FFT-based algorithms do without call stack.

Open problem 5.1. Given $X, Y, Z \in \mathsf{K}^{n\times n}$, is it possible to compute $Z \mathrel{+}= X \cdot Y$ in $O(n^{\log 7})$ operations, with no extra space?

[1] The PLUQ decomposition of A is $A = P\left[\begin{smallmatrix} L \\ M \end{smallmatrix}\right]\left[\begin{smallmatrix} U & V \end{smallmatrix}\right]Q$ where P and Q are $n \times n$ permutation matrices, and L (resp. U) is an $r \times r$ lower (resp. upper) triangular matrix. Due to the dimensions, the four matrices L, M, U and V can be stored in A as $\left[\begin{smallmatrix} L\backslash U & V \\ M & 0 \end{smallmatrix}\right]$. The permutation matrices P and Q may be stored in the bottom right square of dimensions $(n-r)\times(n-r)$, only if r is small enough. Otherwise, one needs $O(n)$ pointers to store these permutations. But if the base ring is large enough (more than mn elements), it is actually possible to store them inside A. A permutation is required only when a null pivot is encountered, and this zero can be replaced by some indices. The same applies for the inverse. More details in [M2, Remark 18].

Conclusions and perspectives 6

We have shown that many polynomial computations, hence many structured linear-algebraic computations, as well as many unstructured linear-algebraic computations, admit time- and space-efficient algorithms. (Our results on polynomials can be to some extent generalized to matrices with low displacement rank [M2].) These results hold in different models. As mentioned earlier, the traditional space complexity model (`ro/wo` in our language) is unsuitable to time-space complexity analysis. The arguably most natural relaxation of this model is `ro/rw` where the inputs remain read-only. (Even for cumulative operations such as $c \mathrel{+}= a \times b$, it is natural to ask for a and b to be read-only.) Nevertheless, some operations such as computing the remainder in a Euclidean division of polynomials have still no fast and constant-space algorithm in this model. We investigated an even more relaxed model, `rw/rw`, where we showed that any bilinear operations (and actually more general operations too) can be performed both fast and in constant *algebraic* space. This means that these algorithms may still require a call stack of logarithmic size to store some pointers. In this model, we also provide some *in-place* algorithms where the inputs are overwritten by the output.

6.1 Further constant-space algebraic algorithms

This work on time- and space-efficient algorithms in computer algebra is relatively new and many questions remain open. A first set of open questions is to improve the current algorithms: remove the need for a call stack when there is one, transfer some algorithms from the model `rw/rw` to the model `ro/rw`, reduce the hidden constants in the complexities, ... Theoretical studies on the relative power of these models would also be of great interest. For instance, quadratic lower bounds for polynomial multiplications in the model `ro/wo` [1] proved this model unsuitable for time-space complexity. Is it possible to prove some similar lower bounds for the model `ro/rw` which would show the necessity of the model `rw/rw`? In the model `rw/rw`, the special case of in-place computations of the form $x := f(x)$ where the input is replaced by the output even raises some computability questions. We have shown for instance that power series multiplication or division can be computed in place, one of the inputs being replaced by the output. For division, the dividend is replaced but it is not clear whether the same result holds with the divisor being replaced instead. It is also open how to perform power series inversion (where there is only one input) in place. Maybe the simplest question about in-place computations was raised by Roche [189], about polynomial multiplication. Note that the product of two monic polynomials of respective degrees m and n has degree $m + n$. Therefore, if one represents a monic polynomial by the vector of its non-leading coefficients, this problem is an algebraic problem $\pi : \mathrm{R}^{m+n} \to \mathrm{R}^{m+n}$.

Open problem 6.1. Is it possible to replace two monic polynomials f and g by their product, without any extra space? Or at least with a sublinear amount of space?

Note that this question may well receive different answers depending on the exact model of computation: single-tape or multi-tape Turing machine, algebraic RAM, … To phrase it in the traditional complexity-theoretic framework, one may restrict to $\mathsf{R} = \mathbb{F}_2$. This becomes a question about a particular function $p : \{0,1\}^* \to \{0,1\}^*$ which is length-preserving.

6.2 Space-preserving transposition principle

As mentioned in Section 1.7, the transposition principle relates the time complexities of a linear program and its transposed. Relating their time-space complexities is raised as an open problem by Kaltofen [122, Open Problem 6]. To have a meaningful notion of space complexity, this can be phrased in the model of fixed-size algebraic program (Definition 2.1.3), restricted to linear operations. A first answer to Kaltofen's question was given by Bostan, Lecerf and Schost [32]. In their model, they prove that the number of operations and the number of registers are exactly preserved by transposition. But their model not only gives read-write permissions on the inputs but also allows their destruction, that is their programs compute $(\boldsymbol{v}, \boldsymbol{w}) \mapsto (\boldsymbol{x}, M \cdot \boldsymbol{w})$ where $\boldsymbol{x}$ can be anything. By contrast, Theorem 5.1.2 shows that the cumulative problem $(\boldsymbol{v}, \boldsymbol{w}) \mapsto (\boldsymbol{v}, \boldsymbol{w} + M \cdot \boldsymbol{v})$ can be computed using no extra register while asymptotically preserving the number of operations, in the model `rw/rw`. By transposition, the same applies to the transposed problem. These different and incomparable results suggest an updated and generalized version of Kaltofen's question.

Open problem 6.2. Given a fixed-size linear algebraic program to compute $(\boldsymbol{v}, \boldsymbol{w}) \mapsto (\boldsymbol{v}, M \cdot \boldsymbol{v})$ with t instructions using s registers, what are the feasible pairs (S, T) such that there exists a fixed-size linear algebraic program for $(\boldsymbol{w}, \boldsymbol{v}) \mapsto (\boldsymbol{w}, M^{\mathsf{T}} \cdot \boldsymbol{v})$ with T instructions using S registers?

The problem is not completely defined. The answer may depend on the permission model. It can be asked in the two models `ro/rw` and `rw/rw`, and even in a mixed case where the source program has `ro/rw` permissions while the target program has `rw/rw` permissions.

6.3 Beyond computer algebra

The questions investigated in this part extend beyond computer algebra. More generally, the question is about the time-space complexity for some function $f : \Sigma^* \to \Sigma^*$ over some finite alphabet Σ. As we have seen for algebraic computations, the permission models play a role. And the relative powers of these models remain to be understood. A close recent line of inquiry is catalytic computation [40], as presented in Section 2.4. For instance, a recent result [52] showed that if one algorithm computes some function f in polynomial time and another one computes f in catalytic logarithmic space, then there exists a single algorithm to compute f in both polynomial time and catalytic logarithmic space. The questions we investigated in this part are in some sense *fine-grained* versions of this result (but in a different model). Relevant to computer algebra would be to extend our results to the more classical catalytic model to determine if one can replace the `rw` permissions on the input by some catalytic space. Relevant to complexity theory would be to define and study complexity classes corresponding to catalytic logarithmic space, but where the catalytic space is actually restricted to the input and/or output.

6.4 Practical aspects

Although the presentation of our results is mainly theoretical, the relevance to practical computations is a central question. It is yet too early to provide definite answers, but some preliminary experimental results show that at least some of our algorithms from [C10, C11, M2] are in practice at least as fast as standard algorithms that require some extra space. For instance, Clément Pernet made an implementation[1] of the constant-space Strassen-Winograd algorithm (Algorithm 5.2.1) in the FFLAS-FFPACK library [59, 87]. Figure 6.1 provides a comparison of this implementation with the reference implementation of Strassen-Winograd algorithm in FFLAS-FFPACK that already tries to minimize memory allocations. The two implementations have similar running times. The constant-space implementation seems to exhibit a more stable behavior although no explanation is currently known for this phenomenon.

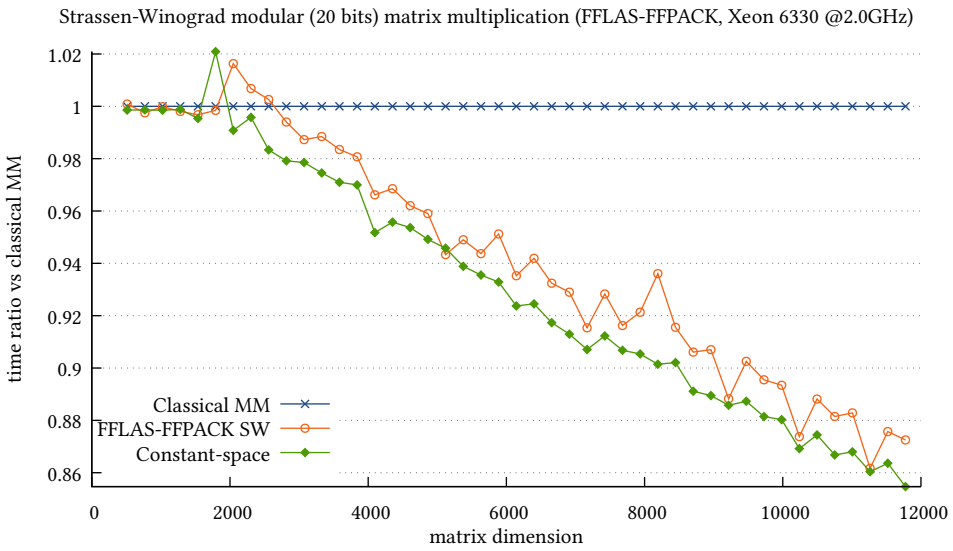


Figure 6.1. Timing comparisons between constant-space Strassen-Winograd algorithm and reference FFLAS-FFPACK implementation (with cubic implementation as reference).

Similarly, Figure 6.2 provides a comparison between a recursive implementation that allocates some memory at each level of recursion, the implementation in NTL [200] that carefully allocates only once all the necessary memory, and the constant-space variant (Algorithm 4.1.1).[2] The constant-space variant is here the fastest. Nevertheless, these results are not fully reproducible. Using a different compiler (GCC instead of Clang) or a different processor produces different results. But on-the-fly allocations are always slower, and in all cases the constant-space implementation is at most 10% slower than the NTL implementation. Further investigation is needed to understand the dependency on the compiler and the processor architecture.

[1] Available in branch `strassen-inplace` of the git repository https://github.com/linbox-team/fflas-ffpack.

[2] These implementations are due to Jean-Guillaume Dumas.

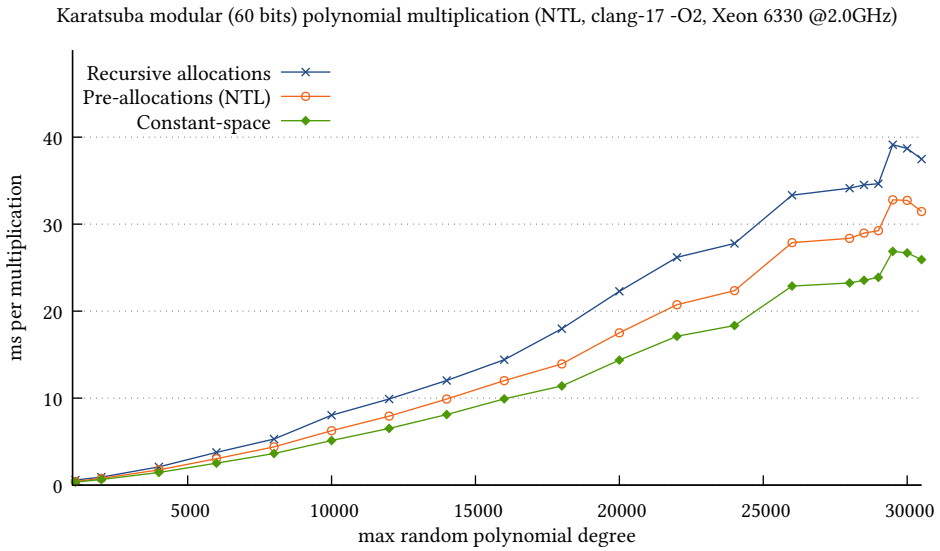


Figure 6.2. Timing comparisons between constant-space Karatsuba algorithm and the original algorithm, either with pre-allocated stack or with on-the-fly allocations

As a prudent conclusion, it seems that the constant-space variants at least partially offset the increase in the number of operations with fewer memory allocations. A more in-depth study is required to draw more definitive conclusions. Practical gains can probably be expected from algorithms in linear algebra, since memory allocation can play a significant role.

6.5 Constant-space quantum algorithms

On a more theoretical side, the algorithms we described in the model `rw/rw` are reversible [18, 149]. One application of reversible computation is quantum computing where, by definition, each computation must be reversible [170]. It can be shown that our reversible algorithms can indeed be phrased in the formalism of quantum computation.[3] In these settings, space complexity corresponds to auxiliary (or *ancilla*) qubits. Reducing the number of such additional qubits is of utmost importance due to the difficulty to build systems with many qubits. A quantum computing version of Algorithm 4.5.1 (ModularMultiplication) provides for instance a multiplication algorithm for finite field extensions that use no extra qubits while still maintaining fast asymptotics. (The algorithm itself can use either Karatsuba's algorithm or FFT-based multiplication as building block.) This improves some known results in the field of quantum computing [78, 104, 119, 138, 175]. Quite recently, some efforts have been made to reduce the space complexity of some important quantum algorithms. As an example, Regev has proposed a new quantum factoring algorithm [186] that has a lower (time) complexity than Shor's algorithm [196], and some subsequent works reduce the number of qubits required to implement it [46, 185]. It would be interesting to investigate to what extent our technique may improve the time-space complexity of this algorithm.

[3] This has been proved by Lucas Ottow for the case of Karatsuba's algorithm, during an internship co-supervised by Pascal Giorgi and myself. This result is unpublished.

II Sparse polynomial computations

Summary

Classical computer algebra algorithms for polynomial manipulations assume that the input in given in *dense representation*, that is by the vector of its coefficients. For a degree-d polynomial $f \in \mathsf{R}[x]$, the representation consists of $(d+1)$ elements from R. This representation is not adapted when most of the coefficients of f are zero, that is for *sparse polynomials*. Instead, one can use a *sparse representation* where only the nonzero terms are represented, by pairs (c, e) where c is the coefficient and e the exponent. If the polynomial has t nonzero terms and degree d, the representation requires t elements from R and $t \log d$ bits to represent the exponents. Compared to the dense representation, the sparse representation may be much more compact for polynomials with few nonzero terms.

Since the representation is more compact, applying standard algorithms to sparsely represented polynomials is inefficient. An algorithm with quasi-linear complexity in the dense representation may have exponential cost in the sparse representation. The point is that standard algorithms do not exploit the sparsity of their inputs. Some algorithms were developed early on for sparse polynomials [118, 213]. But early hardness results [177–179] have probably hindered the research on fast algorithms for sparse polynomials during several decades. More recently, sparse polynomial arithmetic has been the subject of renewed interest [13, 79, 98, 105, 112, 113, 124, 125, 161–163, 166–168], see also the very nice survey of Roche [191]. Despite these works and in contrast with dense polynomial arithmetic, no sparse polynomial multiplication algorithm with subquadratic complexity in the general case was described before 2021.

A central tool to develop algorithms for sparse polynomials is *sparse interpolation.* In very generic terms, the goal is to reconstruct a polynomial f in sparse (explicit) representation from some implicit representation. This is also known in algebraic complexity as *sparse polynomial reconstruction* or *sparse polynomial learning* [201]. Since the seminal work of Ben-Or and Tiwari [17], this problem has attracted considerable attention in computer algebra [5, 9–12, 27, 30, 49, 80, 81, 85, 86, 99–101, 106–111, 115, 116, 126–129, 132, 139, 147, 158, 165, 209]. But until recently, these attempts felt short of providing a quasi-linear time sparse interpolation algorithm.

In Chapter 7, we present the main approaches for sparse interpolation, as well as a new (dense) polynomial root-finding algorithm that is useful in this context. In Chapter 8, we present the first, and only to this date, quasi-linear algorithm for sparse interpolation. It works for polynomials with integer coefficients. Chapter 9 focuses on sparse polynomial arithmetic. We present fast algorithms for multiplication, exact division and divisibility testing. As a necessary tool, we develop fast algorithms for the verification of certain polynomial products. Finally, we also study the factorization of sparse polynomials.

This part is largely based on a series of works carried out as part of Armelle Perret du Cray's PhD thesis, co-supervised with Pascal Giorgi (U. Montpellier), some of which are also co-authored by Daniel S. Roche (U.S. Naval Academy) [C5, C7–C9, C12, J5, M1]. The root-finding algorithm is based on joint works with Grégoire Lecerf and Joris van der

Hoeven (CNRS, École polytechnique) [C2, J1]. Results about sparse factorization are based on single-authored works [C1, J2, J3].

Notations and conventions

This part deals with sparse polynomials of the form $f = \sum_{i=0}^{t-1} c_i x^{e_i}$. In such an expression, we shall always assume that $c_i \neq 0$ for all i, and $e_0 < \cdots < e_{t-1}$. The *support* of f is the set $\{e_i : 0 \leq i < n\}$ of its exponents. The degree of f is $\deg(f) = \max_i e_i = e_{t-1}$. Its *sparsity* $\#f$ is the number t of nonzero terms. The *height* of a polynomial $f \in \mathbb{Z}[x]$ is $\mathsf{h}(f) = \max_i |c_i|$ where $|\cdot|$ denote the absolute value. The bit size of the sparse representation of f is denoted by $\mathrm{bitsize}(f)$ and bounded by $O(\#f(\log \deg(f) + \log b))$, where $b = q$ if the base ring is $\mathbb{F}_q$, and $b = \mathsf{h}(f)$ if the base ring is $\mathbb{Z}$.

For an integer p (usually prime), $f^{[p]}$ denotes the polynomial $f \bmod x^p - 1$ and is called a *fold* of f. We say that a term cx^e belongs to some polynomial f, written $cx^e \in f$, if c is the coefficient of degree e of f.

Some algorithms are given *black box access* to a polynomial $f \in \mathsf{R}[x]$. This means that the algorithm is allowed to evaluate f on any point $\alpha \in \mathsf{R}$ at unit cost. The model of black boxes is further discussed in Section 7.3.

7 Main approaches for sparse interpolation

In this chapter, we review the two main lines of work for sparse interpolation: black box interpolation based on Prony's method (Section 7.1), and SLP interpolation based on reductions modulo $x^p - 1$ (Section 7.2). We then compare the two approaches, and the two models used as inputs in Section 7.3. While the focus of the first three sections is on univariate polynomial, Section 7.4 presents with the multivariate case. Finally, Section 7.5 is devoted to some special finite fields, which we will refer to as *FFT-friendly*, that allow for faster sparse interpolation algorithms. A thorough study of the case of polynomials with integer coefficients is deferred to the next chapter.

7.1 Black box sparse interpolation

The starting point of this approach is Prony's method [183]. Given $2t$ initial terms of a linear recurrent sequence $(u_i)_{i\geq 0}$, the goal is to express each term as an exponential sum $u_i = \sum_{j=0}^{t} \lambda_j \rho_j^i$. By Proposition 1.4.2, the ρ_j's are the roots of the minimal polynomial of $(u_i)_{i\geq 0}$. Recall that $V_t(\boldsymbol{\rho})$ denotes the Vandermonde matrix with second column $\boldsymbol{\rho} \in \mathsf{R}^t$.

Algorithm 7.1.1. Prony

Input: $2t$ terms $u_0, \ldots, u_{2t-1}$ such that $u_i = \sum_{j=0}^{t-1} \lambda_j \rho_j^i$ for $0 \leq i < 2t$, where $\boldsymbol{\lambda}, \boldsymbol{\rho} \in \mathsf{R}^t$

Output: The two vectors $\boldsymbol{\lambda}$ and $\boldsymbol{\rho}$

1 $p \leftarrow$ minimal polynomial of $\boldsymbol{u} = (u_0, \ldots, u_{2t-1})$ — Proposition 1.4.4
2 $(\rho_0, \ldots, \rho_{t-1}) \leftarrow$ roots of p
3 $(\lambda_0, \ldots, \lambda_{t-1}) \leftarrow V_t^{\mathsf{T}}(\boldsymbol{\rho})^{-1} \cdot \boldsymbol{u}_{[0,t[}$ — transposed Vandermonde syst.
4 Return $\boldsymbol{\lambda}, \boldsymbol{\rho}$

To fully specify the algorithm, an algorithm for Step 2 is required. Actually, there is no generic root-finding algorithm for an abstract ring R [69, 70]. We focus on the case where $\mathsf{R} = \mathbb{F}_q$ is a finite field. Let $p \in \mathbb{F}_q[x]$. We assume, as it is the case in Algorithm 7.1.1, that p is split and square-free in $\mathbb{F}_q$. In other words, the polynomial p has as many distinct roots in $\mathbb{F}_q$ as its degree. In this case, Berlekamp-Rabin algorithm [20, 184] can be used to compute its roots, as follows.

Since $\alpha^{q-1} = 1$ for any nonzero $\alpha \in \mathbb{F}_q$, the set of roots of the polynomial $x^{q-1} - 1$ is exactly $\mathbb{F}_q^\times$. Assume for simplicity that q is odd. Then $x^{q-1} - 1 = (x^{\frac{q-1}{2}} - 1) \cdot (x^{\frac{q-1}{2}} + 1)$. Therefore, the roots of $g = \gcd(p, x^{\frac{q-1}{2}} - 1)$ are exactly the roots α of p such that $\alpha^{\frac{q-1}{2}} = 1$, in other words the roots of order at most $(q-1)/2$. The roots of p/g are the roots of p of order $q - 1$. More generally, the polynomial $g_\sigma = \gcd(p, (x + \sigma)^{\frac{q-1}{2}}) = 1$ contains the

roots α of p such that $\alpha + \sigma$ has order at most $(q-1)/2$, and p/g_σ contains the rest of the roots. For any σ, the polynomial g_σ can be computed by fast exponentiation of $(x+\sigma)$ modulo p. Berlekamp-Rabin algorithm takes a random shift σ, computes g_σ and p/g_σ, and recursively computes their roots.

Theorem 7.1.1 ([20, 184]). *Let $p \in \mathbb{F}_q[x]$ of degree d with d distinct roots in $\mathbb{F}_q$. Its roots can be computed in $O(\mathsf{M}(d)\log(d)\log(q))$ operations in $\mathbb{F}_q$, or $\widetilde{O}(d\log^2 q)$ bit operations.*

Using this result, we can analyze Algorithm 7.1.1 (Prony) in the case of finite fields. Note that Step 3 can be solved in $\widetilde{O}(t)$ operations in $\mathbb{F}_q$ by transposition (Proposition 1.7.2) of dense interpolation (Proposition 1.3.5).

Theorem 7.1.2. *If $\boldsymbol{u}, \boldsymbol{\lambda}$ and $\boldsymbol{\rho}$ lie in the finite field $\mathbb{F}_q$, Algorithm 7.1.1 (Prony) requires $\widetilde{O}(t\log^2 q)$ bit operations.*

As a corollary of Proposition 1.4.2, we obtain the fundamental result that underlies black box interpolation. Remark that $f(\omega^j) = \sum_{i=0}^{t-1} c_i \omega^{j\cdot e_i} = \sum_{i=0}^{t-1} c_i (\omega^{e_i})^j$. The following result, sometimes known as Blahut's theorem, is the implication (i) $\Rightarrow$ (iii) of Proposition 1.4.2 where $u_j = f(\omega^j)$, $\rho_i = \omega^{e_i}$ and $\lambda_j = c_j$.

Corollary 7.1.3 ([25]). *Let $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathsf{R}[x]$ and $\omega \in \mathsf{R}$ of multiplicative order $\geq \deg(f)$. The sequence $(f(\omega^j))_{j\geq 0}$ is linear recurrent with minimal polynomial $p = \prod_{j=0}^{k-1}(x - \omega^{e_j})$.*

As a consequence, we obtain a black box sparse interpolation algorithm, whose principle is originally due to Ben-Or and Tiwari [17] in the context of multivariate sparse interpolation of polynomials with integer coefficients.

Algorithm 7.1.2. SparseInterpolationBB

Inputs: Black box access $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathsf{R}[x]$
Bound $\tau \geq \#f = t$
Element $\omega \in \mathsf{R}$ of multiplicative order $\geq \deg(f)$
Output: The sparse representation of f, viz the vectors $\boldsymbol{c}$ and $\boldsymbol{e}$

1 $\boldsymbol{\alpha} \leftarrow (f(\omega^0), \ldots, f(\omega^{2\tau-1}))$ — 2τ calls to the black box
2 $(\boldsymbol{c}, \boldsymbol{\rho}) \leftarrow \textsc{Prony}(\boldsymbol{\alpha})$ — $\alpha_i = \sum_j c_j \rho_j^i$
3 for $j = 0$ to $t-1$: $e_j \leftarrow \log_\omega(\rho_j)$ — $\rho_j = \omega^{e_j}$
4 return $\boldsymbol{c}, \boldsymbol{e}$

As for Prony's algorithm, one cannot provide a complete complexity analysis of Algorithm 7.1.2 (SparseInterpolationBB) over an abstract ring so focus on the case of finite fields. The most computationally expensive part is the discrete logarithm computations. With no assumption of $\mathbb{F}_q$, a discrete logarithm can be computed with $O(\sqrt{q})$ operations in $\mathbb{F}_q$ using the *baby steps / giant steps* algorithm [195]. The computation of t discrete

logarithms can be amortized. As long as $t \ll q$, they can be computed in $O(\sqrt{tq})$ bit operations [145]. This dominates the cost of the whole algorithm.

Theorem 7.1.4. *If $f \in \mathbb{F}_q[x]$, Algorithm 7.1.2 (SparseInterpolationBB) requires $\widetilde{O}(\sqrt{tq})$ bit operations.*

Remark. The bound given in the theorem can be improved by using better discrete logarithm algorithms. Over any finite field, there exist subexponential algorithms [2].[1] Also, if a bound $\delta \geq \deg(f)$ is given, the discrete logarithm computations can be sped up to be performed in time $\widetilde{O}(\sqrt{t\delta})$ instead of $\widetilde{O}(\sqrt{tq})$ [145, 181]. We do not discuss into more details these faster algorithms that remain too expensive in our case. In the following, we shall mainly explain how to circumvent discrete logarithm computations. An exception is for finite fields with multiplicative group of smooth cardinality, discussed in Section 7.5.

Algorithm 7.1.2 (SparseInterpolationBB) requires an upper bound τ on the sparsity t of the output polynomial. This is used for the computation of the minimal polynomial of the sequence $(f(\omega^j))_{j\geq 0}$, that is known to have degree at most τ. Hence, 2τ evaluations are sufficient to compute it. In order the remove this upper bound, one can use a randomized approach, coined *early termination* [128, 129]. Consider the sequence of polynomials $(f(x^j))_{j\geq 0}$. It is linear recurrent of order t, and has the same minimal polynomial as $(f(\omega^j))_{j\geq 0}$. Let $y_j = f(x^j)$ for all i. For any τ, define the Hankel matrix

$$H_\tau = \begin{bmatrix} y_{2\tau-1} & y_{2\tau-2} & \cdots & y_\tau \\ y_{2\tau-2} & y_{2\tau-3} & \cdots & y_{\tau-1} \\ \vdots & \vdots & \ddots & \vdots \\ y_\tau & y_{\tau-1} & \cdots & y_1 \end{bmatrix}.$$

For any $\tau > t$, H_τ is singular since the first column is, by definition of a linear recurrent sequence, a linear combination of the other ones. But for $\tau \leq t$, this matrix can be shown to be full rank. The sparsity of f is therefore the largest value τ for which H_τ is full rank, of equivalently for which $\det(H_\tau)$ does not vanish. Since $\det(H_\tau)$ is a polynomial, its vanishing can be tested by an evaluation on a random point $\alpha \in \mathbb{F}_q$. Algorithmically, this means that one computes the sequence $(f(\alpha^i))_{1\leq i\leq 2\tau-1}$ and tests whether the corresponding Hankel matrix is full rank. This can be done in a quasi-linear number of operations in $\mathbb{F}_q$. From this, it is clear that one can compute the sparsity with the same approach. Actually, this test can be made implicit within the computation of the minimal polynomial [128, 129].

Altogether, this provides a variant of Algorithm 7.1.2 (SparseInterpolationBB) that does not take as input a bound on the sparsity, if the base ring is large enough. Also, no element of large order needs to be provided since the evaluations are made on random elements of $\mathbb{F}_q$. Since the algorithm requires the field to be large enough, namely $q = \Omega(\deg(f)^4)$, random elements have large order with high probability.

[1] Many algorithms have been described for computing discrete logarithms problems. Some of them are proven while others are heuristic. Also, the best complexities depend on the relative size of the field compared to its characteristic. We refer to [71] for some more details.

7.2 SLP-based sparse interpolation

The second approach assumes a richer representation of the polynomial as input, namely a straight-line program. The starting point is the obvious inefficient algorithm. Given an SLP for $f = \sum_{j=0}^{t-1} c_j x^{e_j}$, one can explicitly compute the result of each instruction. This approach is inefficient since intermediate results may be dense polynomials much larger than the final result. The solution is then to compute a *fold* $f^{[p]} = f \bmod x^p - 1 = \sum_{j=0}^{t-1} c_j x^{e_j \bmod p}$. This is achieved by replacing each multiplication in the SLP by a multiplication modulo $x^p - 1$.

Definition 7.2.1. Let $f \in \mathsf{R}[x]$ and $p \in \mathbb{Z}_{>0}$. The *fold of f modulo p* is the polynomial $f^{[p]} = f \bmod x^p - 1$.

There are two difficulties to overcome:

(i) The exponents of f are reduced modulo p in $f^{[p]}$;

(ii) There can be *collisions*, that is two monomials whose degrees are congruent modulo p are mapped to a same monomial in $f^{[p]}$.

The first difficulty can be dealt with either using the Chinese Remainder Theorem (using several primes) with some technicalities, or using the derivative of f to *embed the exponents into the coefficients*. We only present the second technique which is more efficient, although less general since it requires the characteristic of R to be zero or large enough. The idea is to compute not only the fold of the polynomial f, but also the fold of xf' where f' is the derivative of f. A monomial cx^e of f becomes cex^e in xf', and they are mapped to $cx^{e \bmod p}$ and $cex^{e \bmod p}$ after reduction modulo $x^p - 1$, respectively. If there is no collision, from two monomials cx^d in $f^{[p]}$ and $c'x^d$ in $(xf')^{[p]}$, one can reconstruct the original monomial $cx^{c'/c}$ from f.

Algorithm 7.2.1. TentativeTerms

Inputs: $f^{[p]}$ and $(xf')^{[p]}$
Bound $\delta \geq \deg(f)$
Outputs: f_* that contains all terms of f that do not collide modulo p, and $g_* = xf_*'$

```
1 f_* ← 0; g_* ← 0
2 for each pair of terms cx^e ∈ f^[p] and c'x^e ∈ (xf')^[p]:
3     if c divides c' in ℤ and 0 ≤ c'/c ≤ δ:
4         f_* ← f_* + c · x^(c'/c)
5         g_* ← g_* + c' · x^(c'/c)
6 return f_* and g_*
```

Note that the previous algorithm correctly computes all the terms of f if no collision occurs. Otherwise, some terms can be missed and some spurious terms can be added to f_*. This is the second difficulty mentioned above. A first approach is to probabilistically avoid any collision, taking for p a random prime in a suitable interval. This requires to have

$p = O(t^2 \log d)$ where $t = \#f$ and $d = \deg(f)$ [13] and is too costly. A second approach is to allow *some* collisions [9]. A monomial cx^e of f is said *collision-free modulo* p if no other monomial of f has a degree congruent to e modulo p. Even in the presence of collisions, TentativeTerms returns some polynomial f_* that *approximates* f. It contains all the collision-free monomials of f, as well as some spurious monomials coming from collisions. For $p = O(t \log d)$, $\frac{2}{3}t$ monomials are probably collision-free [9]. If this is the case, at most $\frac{1}{3}t$ terms provide collisions. Since each collision involves at least two terms, the number of collisions is at most $\frac{1}{6}t$. If each collision creates a spurious term in f_*, then $f - f_*$ has at most $\frac{1}{3}t$ original terms from f and $\frac{1}{6}t$ spurious terms. Therefore, $f - f_*$ has at most $\frac{1}{2}t$ terms. This means that $\log t$ iterations are enough to fully reconstruct f.

The algorithm requires an SLP for xf' in addition to an SLP for f. It is easily computed in linear time using the standard rules for derivation: $(f_1 + f_2)' = f_1' + f_2'$ and $(f_1 \times f_2)' = f_1'f_2 + f_1f_2'$. The size of the new SLP is at most thrice the size of original one. (The generalization to multivariate SLPs is known as automatic differentiation and due to Baur and Strassen [15].[2]) This provides the following algorithm which is a slight modification of an algorithm of Huang [107].

Algorithm 7.2.2. SparseInterpolationSLP

Inputs: SLP $\mathcal{F}$ for $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathsf{R}[x]$
Bounds $\delta > \deg(f)$, $\tau > \#f$
Output: The sparse representation of f, with probability $\geq \frac{2}{3}$
Assume: $\mathsf{char}(\mathsf{R}) = 0$ or $\mathsf{char}(\mathsf{R}) > \deg(f)$
Constants: $k = \lceil \log \tau \rceil$; $\lambda = \lceil \frac{5}{3\varepsilon_k}(\tau - 1) \ln \delta \rceil$

1 $\mathcal{G} \leftarrow$ SLP for xf'
2 $f_* \leftarrow 0$; $g_* \leftarrow 0$
3 repeat k times:
4 $p \leftarrow$ random prime in $[\lambda, 2\lambda]$
5 $u \leftarrow f^{[p]} - f_*^{[p]}$ using $\mathcal{F}$ and f_*
6 $v \leftarrow (xf')^{[p]} - g_*^{[p]}$ using $\mathcal{G}$ and g_*
7 $(f_{**}, g_{**}) \leftarrow$ TentativeTerms(u, v, δ)
8 $f_* \leftarrow f_* + f_{**}$
9 $g_* \leftarrow g_* + g_{**}$
10 if $\#f_* > 2\tau$: return FAILURE
11 return f_*

Theorem 7.2.2. *Algorithm 7.2.2 (SparseInterpolationSLP) requires* $\tilde{O}(\ell\tau \log(\delta))$ *operations in* R *and* $O(\tau \log(\delta))$ *operations on integers of size* $O(\log(\tau \log(\delta)))$*, where* ℓ *is the size of* $\mathcal{F}$.

Remark. To improve the dependency on the failure probability, a classical solution is to repeat several times the algorithm and use a majority vote. To have a success probability

[2] This attribution makes sense in the computer algebra community. The same algorithm and its very close cousin named backpropagation are central to machine learning and have been independently described several times in that community [16].

$1-\varepsilon$, $O(\log(\frac{1}{\varepsilon}))$ repetitions are sufficient. In the context of this algorithm, a finer approach exists. Since any failure comes from too many collisions, one can also in each iteration sample a set of $O(\log(\frac{1}{\varepsilon}))$ random primes and keep the prime that maximizes the sparsity of $f^{[p]} - f_*^{[p]}$. This is the approach used in Huang's original algorithm [107].

Huang presented this algorithm in the context of a finite field $\mathbb{F}_q$ with large characteristic.

Theorem 7.2.3 ([107]). *Over a finite field $\mathbb{F}_q$ such that $\mathrm{char}(\mathbb{F}_q) \geq \deg(f)$, Algorithm 7.2.2 (SparseInterpolationSLP) returns the sparse representation of f with probability at least $1-\varepsilon$ in time $\widetilde{O}(\ell\tau \log\delta \log q \log\frac{1}{\varepsilon})$, where ℓ is the size of the SLP.*

We note that a slightly faster algorithm has been proposed by Huang [109], that uses a Prony-like approach combined with *exponent embedding*. This is presented in the next section.

7.3 Comparisons

In this section, we discuss alternative presentations of the classical algorithms described above, and links with other sparse interpolation algorithms.

First, note that Algorithm 7.1.2 (SparseInterpolationBB) actually evaluates the black box for f on powers of an element ω of order $n \geq \deg(f)$. Another presentation of this algorithm could consider that the input is the full vector $(f(\omega^i))_{0\leq i<n}$. With random access to this vector, it is possible to interpolate f within the complexity described earlier. In other words, this algorithm can be seen as a(n inverse) *sparse Fast Fourier Transform*. This name originates in the field of signal processing. There, the input is a vector $v \in \mathbb{C}^n$ whose discrete Fourier transform is known to be t-sparse. The goal is to compute it in time sublinear in n, closer to linear in t. Actually, the more general problem is to compute, given $v \in \mathbb{C}^n$, a t-sparse *approximation* of its DFT. The algorithms for sparse FFT can be traced back to a sparse polynomial interpolation algorithm of Mansour for polynomials over $\mathbb{Z}$, given a black box over $\mathbb{C}$ [5, 158]. The best sparse FFT algorithms have complexity $O(t\log n)$ for the exact case, and $O(t\log(n)\log(n/t))$ for the approximate case [93, 94]. Although these algorithms use techniques that are specific to the complex numbers, others are very close to the techniques present in sparse interpolation. For instance, they use the idea of *folding* the polynomials modulo $x^p - 1$ as in Algorithm 7.2.2 (SparseInterpolationSLP).

This idea of *folding* can also be used to provide an alternative view of Algorithm 7.2.2 (SparseInterpolationSLP). The SLP is indeed only used to compute $f^{[p]}$ and $(xf')^{[p]}$ for some primes p. This means that it can be described as taking as inputs *folding black boxes* for f and xf' which, on input p, return $f^{[p]}$ and $(xf')^{[p]}$ respectively. In other words, the algorithm does not make full use of its input since it does not exploit the *structure* of the SLP.[3] To the best of our knowledge, no sparse interpolation algorithm exploits this structure.

To summarize the previous discussion, let us define several models of black boxes for a polynomial.

[3] This structure is better viewed in the arithmetic circuit representation.

Definition 7.3.1. Let $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathsf{R}[x]$.

- A *regular black box* takes as input $r \in \mathsf{R}$, and returns $f(r)$.
- A *geometric black box*, parameterized by some $\omega \in \mathsf{R}$, takes as input i and returns $f(\omega^i)$.
- A *folding black box* takes as input $p \in \mathbb{Z}_{>0}$, and returns $f^{[p]}$, that is $f(\zeta) \in \mathsf{R}[\zeta]/\langle \zeta^p - 1 \rangle$.
- A *modular black box* takes as inputs $\mathcal{I}$ and $m \in \mathsf{R}/\mathcal{I}$ where $\mathcal{I}$ is an ideal of R, and returns $f(m) \in \mathsf{R}/\mathcal{I}$.
- An *extended black box* takes as inputs S and $s \in \mathsf{S}$ where S is an extension ring of R, and returns $f(s) \in \mathsf{S}$.

A black box *with derivative* not only returns the evaluation of f, but also the evaluation of its derivative f' (or equivalently of xf').

As discussed earlier, Algorithm 7.1.2 (SparseInterpolationBB) only requires a geometric black box rather than a regular black box and Algorithm 7.2.2 (SparseInterpolationSLP) only requires access to a folding black box with derivative instead of an SLP. A natural question is to understand which black box is needed is which context.

First, we can relate the power of some of these black boxes and of SLPs. Figure 7.1 summarizes this paragraph. First, an SLP for f is enough to implement any type of black box, with or without derivative. (For the geometric black box, this assumes that ω is given.) In this sense, the SLP model is the strongest one. In terms of complexity, the simulation of each call to a black box requires ℓ operations in the ring where the black box operates, where ℓ is the size of the SLP. For instance, a call to a folding black box requires $O(\ell \mathsf{M}(p))$ operations in R. The presentation in terms of black boxes requires some care to avoid any *cheating*. For instance, f can be read directly from one evaluation of a folding black box, if $p > \deg(f)$. Associating a cost to each call to the black box prevents this problem.

A regular black box can implement a geometric black box, given access to ω. A folding black box is a special case of an extended black box. An extended black box can implement a regular black box with derivative. Indeed, consider $\mathsf{R}_\varepsilon = \mathsf{R}[\varepsilon]/\langle \varepsilon^2 \rangle$. Then for any $r \in \mathsf{R}$, $f(r + \varepsilon) = f(r) + \varepsilon f'(r)$ and a call to the black box with inputs r and $r + \varepsilon$ provides both $f(r)$ and $f'(r)$. Note that if $\mathsf{R} = \mathbb{F}_p[\zeta]/\langle \psi \rangle$ for some prime p and $\psi \in \mathbb{F}_p[\zeta]$, the black box with derivative can also be implemented using $\mathsf{R}_2 = (\mathbb{Z}/p^2\mathbb{Z})[\zeta]/\langle \psi \rangle$ since $f(r \cdot (1 + p)) = f(r) + p \cdot (xf')(r) \in \mathsf{R}_2$. The same idea proves that a modular black box can always evaluate f and its derivative.

We are also interested in the limits of some of the models. First, for $\mathsf{R} = \mathbb{Z}$, it is quite clear that from the viewpoint of bit complexity, the regular black box is not satisfactory. The output $f(r)$ has exponential size in the size of the sparse representation of f in the worst case. Therefore, it is natural to use a modular black box and to reconstruct f from evaluations $f(\alpha) \bmod m$ for some m and $\alpha \in \mathbb{Z}/m\mathbb{Z}$. Note that a modular black box for f also provides a modular black box with derivative using the same technique as before to evaluate both f and xf' over some ring $\mathbb{Z}/m\mathbb{Z}$.

This difficulty disappears when $\mathsf{R} = \mathbb{F}_q$ since every element has the same size. But the

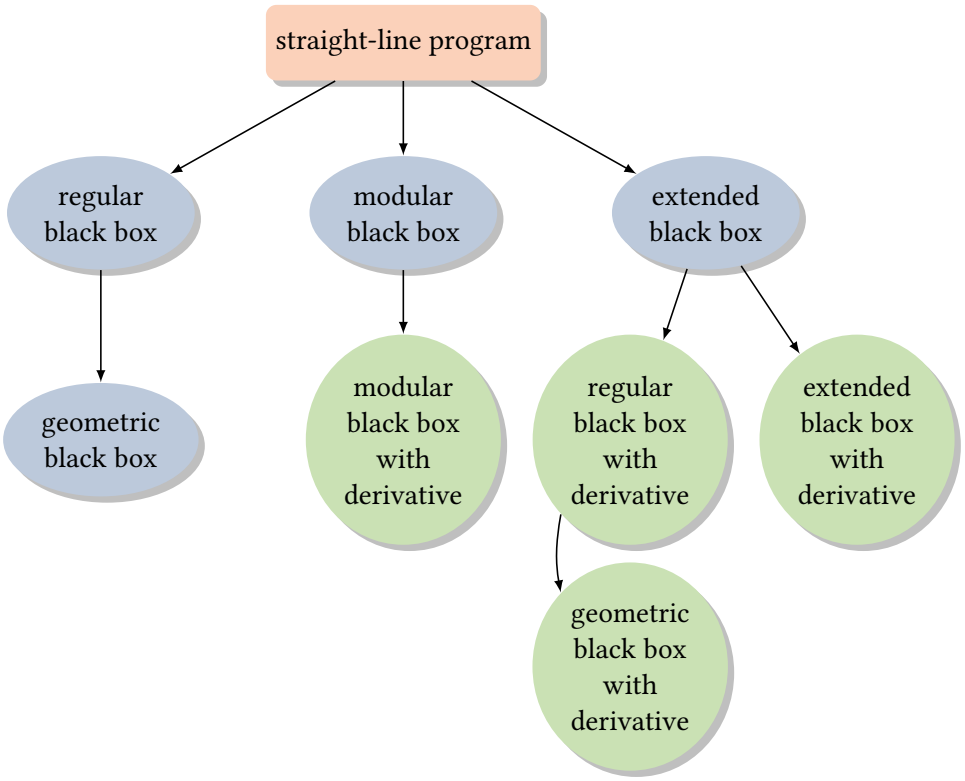


Figure 7.1. Relation between the main black box models and SLPs. An arrow indicates that the source is able to simulate the target.

complexity of Algorithm 7.1.2 (SparseInterpolationBB) is exponential because it requires some discrete logarithm computations. It is a natural question to ask whether they can be avoided. The answer is negative if the input is a geometric black box.

Proposition 7.3.2. *Consider an algorithm that, given access to a geometric black box for $f \in \mathbb{F}_q[x]$, computes the sparse representation of the f. This algorithm can be used to compute, given ω and $\alpha \in \mathbb{F}_q$, the discrete logarithm e of α that satisfies $\alpha^e = \omega$.*

Proof. Assume we are given an algorithm $\mathcal{A}$ that makes queries to a geometric black box for f and outputs the sparse representation of f. Given α, we want to compute e such that $\alpha = \omega^e$, using $\mathcal{A}$. To this end, define $f(x) = x^e$. We can apply $\mathcal{A}$ on f, simulating each query to the geometric black box. Indeed, $f(\omega^i) = \omega^{e \cdot i} = \alpha^i$ can be computed from α and i without knowing e. Therefore, $\mathcal{A}$ returns the sparse representation of f, that is the value e. □

One could argue that we apply the algorithm in some very specific case where f has sparsity one. There might exist an algorithm that does not compute discrete logarithms but only works for polynomials of sparsity at least 2, or at least some other constant. Actually, the reduction still works in these more general settings. It is enough to add to f any known sparse polynomial g, and even multiply x^e by some known coefficient. The queries to the geometric black box can still be simulated and the result holds.

The lower bound does not hold for a geometric black box with derivative, since there is no *a priori* way to simulate the black box calls for f'. Actually, a fast algorithm exists when the input is a geometric black box with derivative. The idea is to evaluate both f and xf' on powers of ω and to use Algorithm 7.1.1 (Prony) on both sequences. This provides in particular the coefficients of f and of xf', from which the exponents of f can be computed. This algorithm has been described in the case of finite fields by Huang [109], using an SLP as input. We present his result in more restricted settings.

Theorem 7.3.3 ([109]). *Given a geometric black box with derivative for $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathbb{F}_q[x]$ where $\mathrm{char}(\mathbb{F}_q) \geq \deg(f)$, and bounds $\tau \geq t$ and $\delta > \deg(f)$, one can compute the sparse representation of f using $\widetilde{O}(\tau)$ calls to the black box and $\widetilde{O}(t \log^2 q)$ bit operations.*

The most expensive part of this algorithm is actually the root finding part of Algorithm 7.1.1 (Prony). This is an intriguing open question whether this root finding step can be bypassed to get a better complexity.

7.4 Multivariate sparse interpolation

We have presented univariate interpolation algorithms. Actually, these algorithms were usually first presented in the multivariate settings. We now present the two main generic solutions to turn a univariate interpolation algorithm into a multivariate one.

The classical reduction from the multivariate case to the univariate case in Kronecker substitution [144]. Let $f \in \mathrm{R}[x_0, \ldots, x_{n-1}]$ where the individual degree of f in each variable is $< d$. We define $f_u = f(x, x^d, x^{d^2}, \ldots, x^{d^{n-1}})$. The map $f \mapsto f_u$ is easily inverted by writing the exponents of f_u in base d. If $\deg_{x_i}(f) < d_i$ for $0 \leq i < n$, this can be improved by setting $f_u = f(x, x^{d_0}, x^{d_0 d_1}, \ldots, x^{d_0 \cdots d_{n-2}})$. The resulting polynomial has the same number of terms as f, and degree $< d_0 \cdots d_{n-1} < d^n$ if $d_i < d$ for all i. Therefore, univariate algorithms can be used for the interpolation of multivariate polynomials. In the complexity analyses, a term $\log(d)$ is replaced by $\log(d^n) = n \log(d)$. This has two main drawbacks. First, a univariate algorithm that is not quasi-linear in $\log(d)$ will result in a multivariate algorithm that is not quasi-linear in the number n of variables. Second, if a univariate algorithm only works for rings of characteristic larger than the degree, the condition becomes that the characteristic be larger than d^n in the multivariate case.

A way to reduce the degree of the resulting univariate polynomial is to use a *randomized* Kronecker substitution [12]. The idea is to define $f_{\boldsymbol{a}} = f(x^{a_0}, \ldots, x^{a_{n-1}})$ for some randomly chosen vector $\boldsymbol{a}$ where $a_i = \widetilde{O}(nt \log d)$ for each i. The resulting degree is much smaller. But the difficulty is that the map $f \mapsto f_{\boldsymbol{a}}$ is not invertible anymore, since there may be some *collisions* between monomials. This can be dealt with by using several vectors $\boldsymbol{a}$. Many approaches, with sometimes incomparable complexities, have been proposed. One can use a set of random vector and rely on linear algebra [11, 12], or the columns of a random Hankel matrix and rely on structured linear algebra (attributed to Pernet [8]), or a set made of $\boldsymbol{a}$ and perturbations of it [8, 107, 110].

A slight issue with (randomized) Kronecker is that it requires to evaluate the polynomial to interpolate (be it given as a black box or an SLP) on several sets of inputs. The complexities therefore all contain a hidden n^2 factor [176]. Another approach using partial derivatives can remove this hidden factor [108].

7.5 FFT-friendly finite fields

The complexity of sparse interpolation over finite fields does not make any assumption on the field. In the case of black-box interpolation, the bottleneck is discrete logarithm computations which is not even polynomial in the output size. For specific finite fields where the cardinality of the multiplicative group is smooth, one can use Pohlig-Hellman algorithm [180] to fasten this computation. This approach works nicely when one can choose the finite field over which to work, for instance in a modular approach for sparse interpolation over the integers or the rational numbers. This has been proposed and analyzed by Kaltofen [121].

For such an approach, as well as the fastest SLP interpolation algorithm [107, 108], polynomial root finding is a costly step. The roots of a degree-d polynomial over $\mathbb{F}_q$ can be computed in $O(\mathsf{M}(d)\log(d)\log(q))$ operations in $\mathbb{F}_q$ using Berlekamp-Rabin algorithm [20, 184]. The bit complexity is not quasi-linear in the output size due to the factor $\log^2 q$ that appears. In practice, the root-finding step appears to be the bottleneck too [99].

In this section, we investigate practical improvements when the finite field has some special structure. We consider finite fields $\mathbb{F}_q$ such that $q-1$ is a smooth integer. In particular, we can focus on so-called *FFT-friendly* finite fields, where $q-1 = m\cdot 2^e$ for some $m = O(\log q)$. Example of primes q that satisfy the condition are $3\cdot 2^{12}+1$, $7\cdot 2^{26}+1$ or $5\cdot 2^{55}+1$. While it is not known how to generate prime numbers of this form within a given interval in general, examples are quite abundant and used in practice. The typical situation is some computation with integer polynomials with not-too-large coefficients, that is performed modulo an FFT-friendly prime. Their name come from the fact that $\mathbb{F}_q$ has a primitive root of unity of order e, making FFT-based algorithms very efficient.

For FFT-friendly finite fields, and more generally finite fields for which $q-1$ is smooth, most classical root-finding algorithms can be fastened [C2]. In the same article, we also introduce a new algorithm based on the *Graeffe transform*, a tool originating in numerical analysis.

Definition 7.5.1. Let $f = \prod_{i=0}^{d-1}(x-\alpha_i) \in \mathbb{F}_q[x]$. Its *Graeffe transform* Gf is the degree-d polynomial defined by either of two ways:

- $Gf(x^2) = (-1)^d f(x)f(-x)$, or
- $Gf(x) = \prod_{i=0}^{d-1}(x-\alpha_i^2)$.

The first definition provides a way to compute the Graeffe transform from the dense representation of f. The second definition shows that the effect of the Graeffe transform is to map the roots of f to their squares. If $q = m\cdot 2^e+1$, the eth iterate of the Graeffe transform $G^e f$ has roots α_i^e, $1\le i\le e$. They have multiplicative order at most m. If ω is a generator of $\mathbb{F}_q^\times$, the roots of $G^e f$ belong to the set $\{\omega^{s\cdot 2^e} : 0\le s<m\}$ and can be computed by multipoint evaluation. A root $\alpha = \omega^{s\cdot 2^e}$ of $G^e f$ corresponds to one or two roots of $G^{e-1}f$. The two possibilities are $\omega^{s\cdot 2^{e-1}}$ and $\omega^{(s+m)\cdot 2^{e-1}}$. Given the roots of $G^e f$, this provides a superset of the roots of $G^{e-1}f$. Using multipoint evaluation again, we can filter out elements of the superset that are not actual roots of $G^{e-1}f$. An algorithm follows: Compute the eth Graeffe transform of f; Compute the roots of $G^e f$ by exhaustive search using multipoint evaluation; Iteratively obtain the roots of each $G^i f$ until $i=0$.

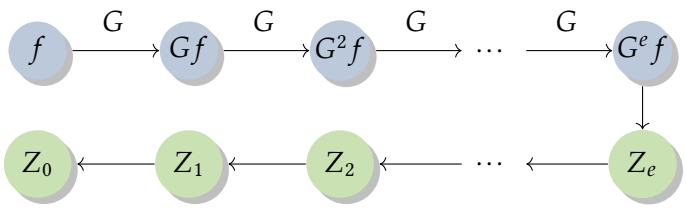


Figure 7.2. Illustration of root-finding using Graeffe transforms. In the first stage, the polynomial $G^e f$ is computed and its roots Z_e are obtained by multipoint evaluation. From Z_e, the sets $Z_{e-1}, \ldots, Z_0$ are computed by multipoint evaluation too.

A generalization of this algorithm, together with some optimization, provides the following deterministic algorithm.

Theorem 7.5.2 ([J1]). *Let $f \in \mathbb{F}_q[x]$ of degree d with d distinct roots in $\mathbb{F}_q$. Given the dense representation of f, the irreducible factorization of $q-1$ and a generator of $\mathbb{F}_q^\times$, the roots of f can be computed in $\widetilde{O}(\sqrt{P_1(q-1)}d \log^2 q) + (d \log^2 q)^{1+o(1)}$ bit operations, where $P_1(q-1)$ is the largest prime factor of $q-1$.*

In particular, if $q = m \cdot 2^e + 1$ with $m = O(\log q)$, the algorithm runs in the same asymptotic complexity $O(\mathsf{M}(d) \log d \log q)$ operations in $\mathbb{F}_q$ as the randomized algorithm of Berlekamp and Rabin. In general, its complexity slightly refines Shoup's complexity bounds [197]. An adaptation of this method provides better randomized algorithms. To this end, we introduce a generalization of the Graeffe transform.

Definition 7.5.3. Let $f \in \mathbb{F}_q[x]$ and consider the extension ring $\mathbb{F}_q[\varepsilon]/\langle \varepsilon^2 \rangle$. The *tangent Graeffe transform* of f is the Graeffe transform of $f_\varepsilon = f(x+\varepsilon) \in \mathbb{F}_q[\varepsilon]/\langle \varepsilon^2 \rangle$.

Note that $f_\varepsilon = f(x) + \varepsilon f'(x)$ and $Gf_\varepsilon = Gf + \varepsilon g$ for some polynomial g. The roots of f_ε are of the form $\alpha - \varepsilon$ where α is a root of f, therefore they are mapped to $(\alpha-\varepsilon)^2 = \alpha^2 - 2\alpha\varepsilon$ in Gf_ε. Since $Gf_\varepsilon(\alpha^2 - 2\alpha\varepsilon) = Gf(\alpha^2) - 2\alpha\varepsilon(Gf)'(\alpha^2) + \varepsilon g(\alpha^2)$ and $Gf(\alpha^2) = 0$, α^2 is a root of Gf' if and only if $g(\alpha^2) = 0$. This provides a way to detect simple roots α^2 of Gf, and for each of them to compute the corresponding root α of f. Indeed, α^2 is a simple root if $g(\alpha^2) \neq 0$ and $\alpha = g(\alpha^2)/2(Gf)'(\alpha^2)$ in that case. The same argument actually works more generally for the ℓth iterate $G^\ell f$. This translates into a variant of the previous algorithm. Once the roots of some $G^i f$ are known, discard the multiple roots and compute directly the corresponding roots of f. (In Figure 7.2, this this would be a direct arc from Z_i to Z_0.) Randomization, working with $f(x+\tau)$ instead of f, ensures that for $i \simeq q/d^2$ all the roots of $G^i f$ are simple with good probability. With fast computation of the tangent Graeffe transform, we obtain the following complexity.

Theorem 7.5.4 ([C2]). *Let $f \in \mathbb{F}_q[x]$ of degree d with d distinct roots in $\mathbb{F}_q$, for $q = m \cdot 2^e + 1$ with $m = O(\log q)$. Given the dense representation of f and a generator of $\mathbb{F}_q^\times$, the roots of f can be computed in expected $O(\mathsf{M}(d) \log d \log q)$ operations in $\mathbb{F}_q$.*

The asymptotic complexity is the same as Berlekamp-Rabin algorithm. Nevertheless, it is in practice faster due to the use of multipoint evaluations instead of GCD computations. To make the algorithm still faster, the goal would be to compute the roots of f from $G^i f$ for a larger value of i. The difficulty is that we are not (yet) able to prove that randomization is sufficient. Anyway, we can make the following heuristic.

(H) For any subset $\{\alpha_0, \ldots, \alpha_{d-1}\} \subseteq \mathbb{F}_q$, $i = O(p/d)$,

$$\Pr_{\tau \in \mathbb{F}_q} \left[\#\left\{ (\alpha_j - \tau)^i : 0 \le j < d \right\} \le 2d/3 \right] \le \frac{1}{2}.$$

The heuristic states that at least $\frac{1}{3}$ of the roots of $f(x+\tau)$ remain simple in the ith iterate of the Graeffe transform with probability at least $\frac{1}{2}$. It is supported by a very similar result when instead of taking τ at random, f itself is chosen uniformly at random.

Using the heuristic, we obtain the following improved complexity.

Theorem 7.5.5 ([C2]). *Let $f \in \mathbb{F}_q[x]$ of degree d with d distinct roots in $\mathbb{F}_q$, for $q = m \cdot 2^e + 1$ with $m = O(\log q)$. Given the dense representation of f and a generator ω of $\mathbb{F}_q^\times$, the roots of f can be computed in expected $O(\mathsf{M}(d)(\log d + \log q))$ operations in $\mathbb{F}_q$, assuming heuristic **(H)**.*

The complexity is this time lower than Berlekamp-Rabin's algorithm. Further complexity analyzes and optimized implementations have demonstrated the relevance of our algorithm in practice [102, 103]. We also note that this root-finding method became popular in cryptographic applications where the field can once again be chosen [193, 212].

As a corollary, we obtain that for finite fields as in Theorem 7.5.5 and under the same heuristic, it is possible to reconstruct a t-sparse degree-d polynomial from a geometric black box with derivative in $\widetilde{O}(\mathsf{M}(t)(\log t + \log q))$ operations in $\mathbb{F}_q$ (*cf.* Theorem 7.3.3).

The situation with polynomial root finding is still unsatisfactory. A natural open question is to remove the heuristic assumption from Theorem 7.5.5. But more generally, the complexity of root finding is $\widetilde{O}(d \log q)$ operations in $\mathbb{F}_q$, that is $\widetilde{O}(d \log^2 q)$. This complexity is not quasi-linear.

Open problem 7.1. Given a degree-d polynomial with coefficients in $\mathbb{F}_q$ with d distinct roots, is it possible to compute its roots in $\widetilde{O}(d)$ operations in $\mathbb{F}_q$, or in $\widetilde{O}(d \log q)$ bit operations?

Quasi-linear sparse interpolation over the integers 8

We describe our quasi-linear time algorithms for sparse interpolation over the integers. The idea is to combine the techniques described so far, taking advantage of the possibility to embed the ring of integers into many modular rings. The algorithms are described in the model of *modular black box* described in Definition 7.3.1: Given m and $\alpha \in \mathbb{Z}/m\mathbb{Z}$, the black box returns $f(\alpha) \in \mathbb{Z}/m\mathbb{Z}$. This model encompasses the model of SLP since given a size-ℓ SLP for $f \in \mathbb{Z}[x]$ and $\alpha \in \mathbb{Z}/m\mathbb{Z}$, one can compute $f(\alpha)$ in $O(\ell \mathsf{Z}(\log m))$ bit operations. Note that the model of (standard) black box is not adapted to sparse integer polynomials since $f(\alpha)$ for $\alpha \in \mathbb{Z} \setminus \{-1, 0, 1\}$ is an integer of bit size $O(\deg(f) \log \mathsf{h}(f))$, exponential in the size of the sparse representation of f.

Our approach is to follow Huang's algorithm and compute f from several folds $f^{[p]}$ and $(xf')^{[p]}$. But while Huang computes these folds directly from an SLP for f, we use the techniques of Algorithm 7.1.2 (SparseInterpolationBB) to compute them.

We distinguish two cases. The first and simpler one is the case of a *balanced* polynomial where the coefficients of the polynomial f are assumed to be of similar bit sizes. The second one is the case of an *unbalanced* polynomial where the coefficients bit sizes may vary a lot.

8.1 The balanced case

As mentioned above, the goal is to reconstruct f from folds $f^{[p]}$ and $(xf')^{[p]}$. The only difference with Huang's algorithm is that $f^{[p]}$ and $(xf')^{[p]}$ are not directly computed from an SLP since that would be too costly. Instead, they are computed from a modular black box using the techniques of black box interpolation.

The first remark is that Algorithm 7.1.2 (SparseInterpolationBB) evaluates f on powers of an element ω of order $> \deg(f)$ in order to reconstruct f. If we apply the algorithm with an element ω of order p, then the output polynomial is actually $f^{[p]}$ since $f(\omega^i) = f^{[p]}(\omega^i)$ in this case. If we can find an element $\omega \in \mathbb{Z}/m\mathbb{Z}$ of order p where $m > 2\mathsf{h}(f)$, we can use this algorithm in $\mathbb{Z}/m\mathbb{Z}$ with ω to compute $f^{[p]} \bmod m$, and deduce $f^{[p]}$.

The second aspect is to be able to compute $(xf')^{[p]}$. To apply the same technique, one needs to be able to evaluate (xf') on some ω. If the input is actually an SLP, we can as in Huang's algorithm compute an SLP for xf'. As explained before, we can evaluate f on $(1+m)\omega$ in $\mathbb{Z}/m^2\mathbb{Z}$ to obtain $f(\omega) + m(xf')(\omega)$. Actually, ω as an element of $\mathbb{Z}/m^2\mathbb{Z}$ may have a larger order than $\omega \in \mathbb{Z}/m\mathbb{Z}$. Hence, we need to use $(1+m)\omega'$ where $\omega' \in \mathbb{Z}/m^2\mathbb{Z}$ has order p and $\omega' \bmod m = \omega$.

The remaining question is the choice of m and the computation of ω and ω'. First, we need $m > 2\mathsf{h}(f)$. The complexity of Algorithm 7.1.2 (SparseInterpolationBB) in $\mathbb{Z}/m\mathbb{Z}$ is then much too costly due to the discrete logarithm computations. The solution is to remark that SparseInterpolationBB can be (reorganized and) split into two parts. The first part computes the support of f while the second one uses the support of f to compute

its coefficients. And only the first part is very costly. Since the support of $f^{[p]}$ contains the support of $(xf')^{[p]}$, one can first compute the support of $f^{[p]}$ and then the coefficients of both $f^{[p]}$ and $(xf')^{[p]}$. For the computation of the support, the only condition on m is that with high probability, no coefficient of $f^{[p]}$ vanishes. We can take for m a small prime q. In a second stage, the coefficients of $f^{[p]}$ and $(xf')^{[p]}$ must be computed exactly. This requires a larger m, that we choose to be a power of q.

For this approach to work, we need a random prime p and prime q together with an element $\omega \in \mathbb{F}_q$ of order p (for the computations modulo q), and an element $\omega_k \in \mathbb{Z}/q^k\mathbb{Z}$ of the same order for the computations modulo q^k. The strategy is to first sample p. Then using effective versions of Dirichlet's theorem on arithmetic progressions, we can sample a prime number q of the form $q = ap + 1$. Then for $\zeta \in \mathbb{F}_q$, the order of ζ^a divides $(q-1)/a = p$. This means that an element ω of order p can be found by sampling $\zeta \in \mathbb{F}_q$ until $\zeta^a \neq 1$. Finally, we can *lift* $\omega \in \mathbb{F}_q$ to obtain an element $\omega_k \in \mathbb{Z}/q^k\mathbb{Z}$ of the same order by means of Hensel lifting [95]. Indeed, if ω_{2^i} has order p in $\mathbb{Z}/q^{2^i}\mathbb{Z}$, we can find $\omega^{2^{i+1}}$ of order p in $\mathbb{Z}/q^{2^{i+1}}\mathbb{Z}$ as $\omega_{2^{i+1}} = \omega_{2^i} + \sigma q^{2^i}$ for some σ. Solving the equation $(\omega_{2^i} + \sigma q^{2^i})^p \bmod q^{2^{i+1}} = 1$ provides a formula for σ.

Algorithm 8.1.1. PrincipalRootsOfUnity

Inputs: Prime number p
Bound γ
Outputs: Prime number $q \leq p^6$ such that p divides $q-1$
Integer k such that $q^k \geq \gamma$
Principal pth root of unity $\omega \in \mathbb{F}_q$
Principal pth root of unity $\omega_k \in \mathbb{Z}/q^k\mathbb{Z}$ such that $q^k \geq \gamma$

1 Sample odd integers $q = a \cdot p + 1$ for $t \in [p^5]$ until q is prime
2 Sample $\omega = \zeta^t$ for $\zeta \in \mathbb{F}_q^\times$ until $\omega \neq 1$
3 $k \leftarrow \lceil \log_q \gamma \rceil$; $\omega_k \leftarrow \omega$
4 for $i = 0$ to $\lfloor \log k \rfloor$:
5 $\quad \sigma \leftarrow (1 - \omega_k^p) \bmod q^{2^{i+1}} / q^{2^i}$
6 $\quad \sigma \leftarrow \sigma \cdot \omega_k \cdot p^{-1} \bmod q^{2^i}$
7 $\quad \omega_k \leftarrow \omega_k + \sigma \cdot q^{2^i}$
8 return $(q, k, \omega, \omega_k \bmod q^k)$

Lemma 8.1.1. *Algorithm 8.1.1 (PrincipalRootsOfUnity) is correct with high probability and uses $\widetilde{O}(\log \gamma \log p) + \log(p)^{O(1)}$ bit operations.*

The exact complexity of the term $\log(p)^{O(1)}$ and the exact failure probability depend on the algorithm used to test the primality of q. The algorithm is either Las Vegas using a (slower) deterministic test [3] or Monte Carlo using a (faster) randomized test [184].

Altogether, we can describe an algorithm which computes both $f^{[p]}$ and $(xf')^{[p]}$ for some prime p.
This algorithm can replace to computation of $f^{[p]}$ and $(xf')^{[p]}$ in Algorithm 7.2.2 (SparseInterpolationSLP) (Lines 5 and 6). As a result, we obtain a quasi-linear time

Algorithm 8.1.2. SparseFoldedInterpolation

Inputs: Black box for $f \in \mathbb{Z}[x]$
Bounds $\delta > \deg(f)$, $\tau \geq \#f$ and $\gamma \geq \mathsf{h}(f)$
Prime number p
Outputs: The sparse representations of $f^{[p]}$ and $(x \cdot f)^{[p]}$

Set up the rings — PrincipalRootsOfUnity$(p, 2\gamma)$
1 $(\omega, \omega_{2k}) \leftarrow p$th PRUs in $\mathbb{F}_q$ and $\mathbb{Z}/q^{2k}\mathbb{Z}$ resp., where $q^k \geq 2\gamma$

Compute the support of $f^{[p]}$ modulo q
2 $\boldsymbol{u} \leftarrow (f(\omega^i))_{0 \leq i < 2\tau}$ — 2τ calls to the black box
3 $(\boldsymbol{\nu}, \boldsymbol{\sigma}) \leftarrow$ Prony$(\boldsymbol{u})$ — $u_j = \sum_i \nu_i \sigma_i^j$
4 for $i = 0$ to $t-1$: $a_i \leftarrow \log_\omega \sigma_j$ — support of $f^{[p]}$

Compute the coefficients of $f^{[p]}$ and $(xf')^{[p]}$ modulo q^k
5 $\boldsymbol{v} \leftarrow (f(\omega_{2k}^i))_{0 \leq i < t}$ — t calls to the black box
6 $\boldsymbol{w} \leftarrow (f((1+q^k)\omega_{2k}^i))_{0 \leq i < t}$ — t calls to the black box
7 $(\rho_0, \ldots, \rho_{t-1}) \leftarrow (\omega_{2k}^{a_0}, \ldots, \omega_{2k}^{a_{t-1}})$
8 $(\lambda_0, \ldots, \lambda_{t-1}) \leftarrow V_t^{\mathsf{T}}(\boldsymbol{\rho})^{-1} \cdot \boldsymbol{v}$ — transposed Vandermonde syst. solv.
9 $(\mu_0, \ldots, \mu_{t-1}) \leftarrow V_t^{\mathsf{T}}(\boldsymbol{\rho})^{-1} \cdot [(\boldsymbol{w} - \boldsymbol{v})/q^k]$
10 Return $f^{[p]} = \sum_{i=0}^{t-1} \lambda_i x^{a_i}$ and $(x \cdot f')^{[p]} = \sum_{i=0}^{t-1} \mu_i x^{a_i}$

sparse interpolation algorithm over $\mathbb{Z}$.

Theorem 8.1.2 ([C9]). *There is an algorithm SparseInterpolationOver$\mathbb{Z}$ that, given a modular black box for a polynomial $f \in \mathbb{Z}[x]$, bounds $\delta \geq \deg(f)$, $\tau \geq \#f$ and $\gamma \geq \mathsf{h}(f)$, and $0 < \varepsilon < 1$, returns the sparse representation of f with probability at least $1-\varepsilon$ using $O(\tau)$ calls to the black box and $\widetilde{O}(\tau(\log\delta + \log\gamma)\log\frac{1}{\varepsilon})$ bit operations. If the input is an SLP of size ℓ, the calls to the black box have total bit cost $\widetilde{O}(\ell\tau(\log\delta + \log\gamma))$.*

This algorithm is of Monte Carlo type. To get a Las Vegas algorithm, one would need either a deterministic or a Las Vegas algorithm to verify if a modular black box (resp. an SLP) represents a given sparse polynomial. Such a deterministic algorithm is known, albeit not of quasi-linear complexity [26]. The existence of a quasi-linear Las Vegas algorithm is an open question.

As mentioned in Section 7.4, it is possible to turn a univariate algorithm into a multivariate one through Kronecker substitution. For n-variate polynomials, the resulting complexity is $O(\tau(n\log\delta + \log\gamma)\log\frac{1}{\varepsilon})$.

Another question is to get rid of the bound τ on the sparsity of f, by means of early termination as presented in Section 7.1. The difficulty for polynomials over $\mathbb{Z}$ is that this requires to work modulo some large prime number p of bit size $\Theta(\log\delta + \log\log\gamma)$. Producing a b-bit prime number has cost $\Omega(b^3)$ which is far from quasi-linear in b [199]. Therefore, we developed a technique of using *random primes without primality testing* [C8]. The idea is to replace a random prime p with a random integer m of slightly larger size. With

good probability, m has a prime factor $\geq \sqrt{m}$. We show that given an algorithm designed to work modulo a randomly chosen prime number, we can make it work modulo m instead. The idea is to precede any inversion modulo p or zero-test by a GCD computation with the current modulus m, and update if a common factor is found. The subtlety lies in the capacity for this transformation to preserve the success probability when the original algorithm is able to sample random elements modulo p. Applying our program transformation technique to early termination in sparse interpolation algorithms provide the following result.

Theorem 8.1.3 ([C8]). *There is an algorithm SparsityOver$\mathbb{Z}$ that, given a modular black box for a polynomial $f \in \mathbb{Z}[x]$, bounds $\delta \geq \deg(f)$ and $\gamma \geq \mathsf{h}(f)$, and $0 < \varepsilon < 1$, returns an integer $\tau \leq \#f$ that equals $\#f$ with probability at least $1 - \varepsilon$, using $O(\tau \log \frac{1}{\varepsilon})$ calls to the black box and $\widetilde{O}(\tau(\log \delta + \log\log \gamma) \log \frac{1}{\varepsilon})$ bit operations.*

We remark that this sparsity estimation algorithm is one-sided error in the sense that the approximation it returns is always at most $\#f$.

8.2 The unbalanced case

The complexity of Algorithm SparseInterpolationOver$\mathbb{Z}$ given in Theorem 8.1.2 is quasi-linear in the standard parameters $\tau \geq \#f$, $\delta \geq \deg(f)$ and $\gamma \geq \mathsf{h}(f)$. But if $s = \mathrm{bitsize}(f)$ denotes the bit size of f, the bound $s \leq \tau(\log \delta + \log \gamma)$ may be loose. If there is for instance one very large coefficient with $\Theta(\log \gamma)$ bits and $O(\tau)$ coefficients of $O(1)$ bits, $s = O(\tau \log \delta + \log \gamma)$. The term $\tau \log \gamma$ in the complexity of SparseInterpolationOver$\mathbb{Z}$ can therefore be quadratic in the bit size s. The same remark applies on the size of the exponents. There may be one term of degree δ and all the other exponents of degree $O(\tau)$. In this case, the representation of the support of f has size $O(\log \delta + \tau \log \tau)$. Combining both remarks, the size of the representation can be as small as $O(\tau \log \tau + \log \delta + \log \gamma)$.

In this part, we investigate the first situation, that is when the coefficient sizes are very unbalanced. Another work has investigated the second situation when the exponent sizes are very unbalanced [101]. It remains an open problem to combine both techniques to handle unbalancedness in both the coefficients and the exponents.

To manage unbalanced coefficients, there are two natural approaches: either a *bottom-up* approach that recovers first the many terms with small coefficients and then the few remaining ones that have large coefficients; or a *top-down* approach that first recovers the few terms with large coefficients and then the many terms with small coefficients.

8.2.1 Bottom-up approach

The first approach starts by interpolating $f_* = f \bmod m$ for some small m. The many coefficients $< m$ in f are preserved in f_*, whence the sparsity of $f - f_*$ is much less than the sparsity of f. The process can be iterated to interpolate $f - f_*$ modulo a larger integer m'. Once $m > 2\mathsf{h}(f)$, $f = f \bmod m$.

The heart of the algorithm is interpolating $f - f_*$ modulo some integer m where f is given by a modular black box, and f_* as a sparse polynomial. The idea is to reuse the same strategy as in the balanced case. The difference is in the computation $f^{[p]}$ and $(xf')^{[p]}$ using Algorithm 8.1.2 (SparseFoldedInterpolation) since we only need these polynomials modulo m. We can simply give the bound $\gamma = m$ as input to the algorithm. It will return $f^{[p]} \bmod q^k$ and $(xf')^{[p]} \bmod q^k$ for some $q^k \geq 2\gamma$ even if $\gamma < \mathsf{h}(f)$.

The problem of this approach is its complexity. From the beginning, the sparsity of f_* is of the same order as the sparsity of f. During the final step, the modulus must satisfy $m = \Theta(\mathsf{h}(f))$ to recover f exactly. This means that f_* must be evaluated on a large input. The cost is $\Omega(t \log d \log \mathsf{h}(f))$ bit operations where $d = \deg(f_*)$. The term $t \log \mathsf{h}(f)$ is exactly the one we try to avoid. This is why we turn to the top-down approach. Still, this method is adapted to unbalanced exponents [101]. The difficulty to combine both kinds of unbalancedness precisely lies in the fact that the unbalancedness of exponents and coefficients seem to need two opposite approaches.

8.2.2 *Top-down approach*

We turn to the second approach. The difficulty here is to only interpolate the *large* terms, that is with large coefficients. The solution is to treat the other terms as *noise*. Let us fix some bound β and consider the polynomial $f_\star$ made of the terms of f whose coefficients are larger than β (in absolute value). Then the fold $f_\star^{[p]}$ is close to $f^{[p]}$ in the sense that $f_\star^{[p]} - f^{[p]}$ has only small coefficients. If a large term cx^e does not collide modulo p, it can be reconstructed from the two corresponding terms $cx^{e \bmod p}$ and $cex^{e \bmod p}$ in $f^{[p]}$ and $(xf')^{[p]}$ as in Algorithm 7.2.1 (TentativeTerms). If it collides with small terms, the corresponding terms in $f^{[p]}$ and $(xf')^{[p]}$ are $\tilde{c}x^{e \bmod p}$ and $\tilde{c}'x^{e \bmod p}$ where $\tilde{c} \approx c$ and $\tilde{c}' \approx ce$. As long as the noise is small enough, the exponent e can still be recovered as the closest integer $\lfloor \tilde{c}'/\tilde{c} \rceil$ to $\tilde{c}'/\tilde{c}$. This way, we reconstruct an approximation $\tilde{c}x^e$ of cx^e. If all large terms can be approximated this way, we get an approximation f_* of $f_\star$, such that $f_* - f_\star$ has only small coefficients. As a result, since $f - f_* = (f - f_\star) - (f_* - f_\star)$ is a difference of two polynomials with small coefficients, it has itself only small coefficients. Iterating this process with $f - f_*$ provides better and better approximations f_* of f.

An iteration as presented above should compute approximations of all and only the largest terms. Since subsequent iterations only compute terms with smaller coefficients, a missing or surplus term with large coefficient cannot be corrected. There are two pitfalls. The first one is missing a large term. This happens if a large term collides with many small terms in $f^{[p]}$ and if the resulting coefficient is too small. To avoid this, we define a *buffer zone* of *medium* coefficients such that a sum of a large coefficient and many small ones remains larger than a sum of small coefficients. Ideally, we would choose a large p to avoid any collisions between medium or large terms, so that large terms only collide with small terms. But for efficiency reasons, the size of p we can afford only ensures that a constant fraction of the large terms only collide with small terms. Using several values for p provides a tentative list of large terms, that contains all actual large terms with high probability. The second pitfall is that this list may contain spurious terms. Instead of avoiding this situation, we only use this list to compute a superset of the exponents that serves as a filter afterwards. We redo the same process of computing large terms from several folds $f^{[p]}$. This time, we only reconstruct large terms of f that did not collide with any other exponent in the superset modulo p. Since there can still be collisions with medium or small terms, a large term may become too small because of collisions. We add another *buffer zone* inside the large terms. We distinguish between the mere *large* terms and the *huge* terms. In one iteration, only the huge ones are computed.

Definition 8.2.1. Let $f \in \mathbb{Z}[x]$ and $\gamma \geq \mathsf{h}(f)$. A term cx^e of f is said

- *small* if $|c| < \gamma^{\frac{1}{6}}$,
- *medium* if $|c| \geq \gamma^{\frac{1}{6}}$,
- *large* if $|c| \geq \frac{1}{2}\gamma^{\frac{13}{30}}$, and
- *huge* if $|c| \geq \gamma^{\frac{1}{2}}$.

Note that every *huge* term is also considered *large*, and every *huge* or *large* term is also considered *medium*. The algorithm can be summarized as follows. It makes use of Algorithm 8.1.2 (SparseFoldedInterpolation).

Algorithm 8.2.1. LargeSupportSuperset

Inputs: Modular black box for f
Bounds $s \geq \text{bitsize}(f)$, $\delta \geq \deg(f)$, $\gamma \geq \mathsf{h}(f)$
Output: Set $\mathcal{T} \subset \mathbb{Z}_{\geq 0}$ that contains the exponents of all large terms of f
Constants: $k = \lceil 2\log s \rceil + 3$; $N = \frac{60}{13} s \log_\gamma(4s)$; $\lambda = \max(21, 18s\log_\gamma \delta)$; $m = 4\gamma^{\frac{7}{6}}$

$\mathcal{T} \leftarrow \emptyset$
repeat k times:
  $p \leftarrow$ random prime in $[\lambda, 2\lambda]$ — SparseFoldedInterpolation with $\tau = p+1$ and $\gamma = m$
  $(g, h) \leftarrow (f^{[p]}, (xf')^{[p]})$
  for each pair (ax^e, bx^e) of terms of g and h:
    if ax^e is *large* and $0 \leq \lfloor b/a \rceil \leq \delta$: add $\lfloor b/a \rceil$ to $\mathcal{T}$ — $|a| \geq \frac{1}{2}\gamma^{\frac{13}{30}}$
    if $\#\mathcal{T} > N$: return FAILURE
return $\mathcal{T}$

Given $\mathcal{T}$, we can now compute all the huge coefficients of f. We use $f^{[p]}$ for $p = O(\#\mathcal{T}\log\delta)$ to mostly avoid collisions between elements of $\mathcal{T}$. Since they still occur, $\mathcal{T}$ is used to detect them and discard the spurious terms they create. With $\Theta(\log s)$ repetitions, each exponent of a large term is non-colliding modulo p for at least one p with high probability. If a huge term only collides with non-large terms, it remains large enough (that is $\geq \frac{1}{2}\gamma^{\frac{1}{2}}$) not to be missed.
After a call to HugeCoefficients, we get an approximation f_* of f such that $\mathsf{h}(f - f*) \leq \sqrt{\gamma}$. We can update γ to its square root, until we reach a small enough value that guarantees that the remaining polynomial $f - f_*$ is actually balanced, and finish the computation with SparseInterpolationOver$\mathbb{Z}$ (Theorem 8.1.2). Since the algorithm may fail to return a correct answer, and since we do not have a good verification algorithm in the unbalanced case, we repeat several times the algorithm and use a majority vote. Finally, a bound on the total bit size of f is enough to derive bounds on its sparsity and height.

Theorem 8.2.2 ([C12]). *There is an algorithm* UnbalancedInterpolation *that, given a modular black box for $f \in \mathbb{Z}[x]$ and bounds $s \geq \text{bitsize}(f)$ and $\delta \geq \deg(f)$, returns the sparse representation of f with probability at least $1 - 1/s$, using $\tilde{O}(s)$ calls to the*

Algorithm 8.2.2. HugeCoefficients

Inputs: Modular black box $\mathcal{B}$ for f
Bounds $s \geq \text{bitsize}(f)$, $\delta \geq \deg(f)$, $\gamma \geq \mathsf{h}(f)$
Output: $f_* \in \mathbb{Z}[x]$ such that $\mathsf{h}(f - f_*) \leq \sqrt{\gamma}$
Constants: $k = \lceil 2 \log s \rceil + 3$; $\lambda = \max(21, 3\#\mathcal{T} \log \delta)$; $m = 2\gamma$

$\mathcal{T} \leftarrow$ LargeSupportSuperset$(\mathcal{B}, s, \delta, \gamma)$
$f_* \leftarrow 0$
repeat k times:
  $p \leftarrow$ random prime in $[\lambda, 2\lambda]$
  $g \leftarrow f^{[p]} - f_*^{[p]}$ SparseInterpolationFolded with $\tau = p$ and $\gamma = m$
  $\mathcal{E} \leftarrow$ array of p sets such that $\mathcal{E}_{[i]} = \{e \in \mathcal{T} : e \bmod p = i\}$
  for each huge term cx^i of g: $|c| \geq \frac{1}{2}\gamma^{\frac{1}{2}}$
    if $\mathcal{E}_{[i]} = \{e\}$ and f_* has no term of degree e: e does not collide mod p
      $f_* \leftarrow f_* + cx^e$
      if $\text{bitsize}(f_*) > s$: return FAILURE
return f_*

modular black box and $\widetilde{O}(s \log \delta)$ bit operations.

When implemented by an SLP, the calls to the modular black box have total cost $\widetilde{O}(\ell s \log \delta)$. As mentioned earlier, the complexity of UnbalancedInterpolation is not quasi-linear in the output size due to the extra factor $\log \delta$. Removing it from the complexity is a challenging open problem.

Open problem 8.1. Given a modular black box for $f \in \mathbb{Z}[x]$ and a bound $s \geq \text{bitsize}(f)$, is it possible to compute the sparse representation of f with $O(s)$ calls to the black box and $\widetilde{O}(s)$ bit operations?

Sparse polynomial arithmetic 9

In this chapter, we present algorithms for sparse polynomials. The case of addition and subtraction is rather simple and the naive algorithm is already optimal. The situation becomes different for multiplication and division. In both cases, one specific difficulty of sparse polynomial arithmetic is the output sensitivity. For instance, the product of two polynomials of sparsity at most t may have between 2 and t^2 nonzero terms.

The strategy to compute a sparse polynomial product or quotient while the size of the output is initially unknown is to *guess* an output size, perform the computation assuming the correctness of the size, and *a posteriori* check whether the result is correct. For the strategy to be efficient, the verification must be extremely fast. The problem at hand is, given three polynomials f, g and h, to determine whether $h = f \times g$.

We present in Section 9.1 our verification algorithms. Section 9.2 is about multiplication and Section 9.3 about division and divisibility testing. Finally, Section 9.4 is independent from the preceding ones and focuses on computing low-degree factors of sparse polynomials.

9.1 Polynomial product verification

The classical strategy to verify a polynomial product relies on evaluation. Given f, g and $h \in \mathsf{R}[x]$, the equality $h = f \times g$ is tested by evaluation $h(\alpha) = f(\alpha) \times g(\alpha)$ where α is chosen at random in R. Beyond other issues such as the size of R, this strategy does not work for high-degree sparse polynomials since the evaluation itself is too costly. Since the evaluation of a polynomial on α corresponds to the computation of the remainder of this polynomial modulo $x - \alpha$, the equality $h(\alpha) = f(\alpha) \times g(\alpha)$ corresponds to $h \bmod x - \alpha = (f \times g) \bmod x - \alpha$. A generalization, introduced by Kaminski [133], is to test $h \bmod \ell = f \times g \bmod \ell$ for some polynomial ℓ. Kaminski uses polynomials ℓ of the form $x^i - 1$. The reduction $h \bmod \ell$ is computed in linear time. Then, the modular equality is verified by computing the product $(f \bmod \ell) \times (g \bmod \ell)$ and then its reduction. Using a small enough value for i, this product can be computed in sublinear time (in the input sizes).

To check a sparse polynomial product, we combine both methods. First we use Kaminski's framework but with prime exponents, that is we test whether $h \bmod x^p - 1 = f \times g \bmod x^p - 1$ for some prime p. This allows us to consider a much smaller value p than with Kaminski's approach. Furthermore, this equality is not verified by performing a polynomial multiplication, but instead by relying on evaluation. That is, we verify whether $(h \bmod x^p - 1)(\alpha) = (f \times g \bmod x^p - 1)(\alpha)$. This requires to be able to evaluate the right-hand side without computing the modular product.

This approach, that we developed for sparse polynomial product verification, is actually more general. It allows us to check modular equalities such as $h = f \times g \bmod \ell$ for a polynomial ℓ, both when the inputs are in dense and in sparse representation. We first

present our method for the evaluation of a modular product, and then its application to modular product verification, and finally to (standard) product verification.

9.1.1 Modular product evaluation

We first consider the evaluation of the (implicitly known) polynomial $fg \bmod x^p - 1$ on $\alpha \in \mathsf{R}$. Let us assume that f and g have been reduced modulo $x^p - 1$, hence have size p. The idea here is to rely on the linear-algebraic interpretation of polynomial arithmetic. The modular product $fg \bmod x^p - 1$ corresponds to the matrix-vector product $C_f \cdot \boldsymbol{g}$ where

$$C_f = \begin{bmatrix} f_0 & f_{p-1} & \cdots & f_1 \\ f_1 & f_0 & \cdots & f_2 \\ \vdots & \vdots & & \vdots \\ f_{p-1} & f_{p-2} & \cdots & f_0 \end{bmatrix}$$

is a circulant matrix. The evaluation of a size-p polynomial on α is given by the inner product with $\boldsymbol{\alpha}_n = (1, \alpha, \ldots, \alpha^{n-1})$. Therefore, the evaluation of $fg \bmod x^p - 1$ on α is given by $\boldsymbol{\alpha}_n \cdot \left(C_f \cdot \boldsymbol{g}\right)$. To perform the evaluation without computing the polynomial $fg \bmod x^p - 1$, the solution is to parenthesize the previous equation on the left: $\left(\boldsymbol{\alpha}_n \cdot C_f\right) \cdot \boldsymbol{g}$.

By taking into account the circulant structure of C_f and the fact that $\boldsymbol{\alpha}_n$ is a geometric progression, $\boldsymbol{\alpha}_n \cdot C_f$ can be computed in linear time, as was already noticed by Giorgi [82]. This approach can be extended to the case of sparse polynomial by leveraging the double structure, circulant and sparse.

Lemma 9.1.1. *Let $f, g \in \mathsf{R}[x]$ of size p and $\alpha \in \mathsf{R}$. The polynomial $f \times g \bmod x^p - 1$ can be evaluated on α in*

- *$O(p)$ operations in R, or*
- *$O((\#f + \#g) \log p)$ operations in R and $O(\#f + \#g)$ operations on the exponents.*

Remark. In the second complexity bound, the term $O((\#f + \#g) \log p)$ can actually be improved to $\log p + O((\#f + \#g) \log p / \log\log p)$, using Yao's simultaneous exponentiation algorithm [211]. The same remark holds for all the complexity bounds given in this part.

These results can be extended to any sparse monic modulus ℓ. The idea is still to express $(fg \bmod \ell)(\alpha)$ as $(\boldsymbol{\alpha}_n \cdot M_{f \bmod \ell}) \cdot \boldsymbol{g}$ where $M_{f \bmod \ell}$ is the matrix of the multiplication by f modulo ℓ. This matrix is not circulant anymore but keeps some structure which allows for a fast vector-matrix product.

If f and g are given in sparse representation, the complexity of the evaluation of $fg \bmod \ell$ depends on the ratio between the degree and the *second degree* of the monic polynomial ℓ, defined as $\deg_2(\ell) = \deg\left(\ell - x^d\right)$ where $d = \deg(\ell)$.

Lemma 9.1.2. *Let $f, g, \ell \in \mathsf{R}[x]$, ℓ monic, and $\alpha \in \mathsf{R}$. Let $d = \deg(\ell)$, $d_2 = \deg_2(\ell)$ and $\gamma = \lceil d_2/(d - d_2) \rceil$. Assume that $\deg\left(f\right), \deg\left(g\right) < d$, and $\#f \leq \#g$. Then $f \times g \bmod \ell$ can be evaluated on α in*

- *$O(\mathsf{M}(d))$ operations in R, or*
- *$O(d \cdot \#\ell)$ operations in R, or*

- $O((\#f\#\ell^\gamma + \#g)\log d)$ *operations in* R.

Remark. The complexity $O(\mathsf{M}(d))$ comes from computing $fg \bmod \ell$ before the evaluation. The complexity $O(d\#\ell)$ becomes smaller as soon as $\#\ell < \mathsf{M}(d)/d$. One can think of the second complexity as adapted for a modulus ℓ with a constant number of terms and dense polynomials f and g. The third complexity is the case of sparse polynomials f and g. Note that for $\deg_2(\ell) \le d/2$, $\gamma \le 1$ and the complexity simplifies to $O((\#f\#\ell + \#g)\log d)$.

Remark. The evaluation of a polynomial on a random point is the basis for polynomial product verification. If the base ring is too small, it is customary to select a random point in a suitable extension of the base ring. One can refine our complexity estimates to split between operations in the base ring and operations in the extension [J5].

9.1.2 *Modular product verification*

The algorithm for modular product verification is directly based on modular product evaluation. The idea is simply to choose a random point α in a large enough set, and perform the modular product evaluation. We obtain the following result.

Theorem 9.1.3 ([J5]). *Let $f, g, h, \ell \in \mathsf{R}[x]$, ℓ monic of degree d, f, g, h of degrees $< d$, and $0 < \varepsilon < 1$. Assume that $\#\mathsf{R} > \frac{1}{\varepsilon}(d-1)$. One can test whether $h = f \times g \bmod \ell$ in*

- $O(\mathsf{M}(d))$ *operations in* R, *or*
- $O(d \cdot \#\ell)$ *operations in* R, *or*
- $O((\#f\#\ell^\gamma + \#g + \#h)\log d)$ *operations in* R,

where $\gamma = \lceil \deg_2(\ell)/(d - \deg_2(\ell)) \rceil$, with a failure probability at most ε when $h \neq fg \bmod \ell$.

This general result can be specialized to some rings of interests, such as the integers or finite fields. For polynomials over the integers, the idea is to perform the computation modulo some prime number, to avoid coefficient swell.

Corollary 9.1.4. *Let f, g, h, ℓ be as in Theorem 9.1.3, with $\mathsf{R} = \mathbb{Z}$. Let $H = \max(\mathsf{h}(f), \mathsf{h}(g), \mathsf{h}(h), \mathsf{h}(\ell))$, $t = \max(\#f, \#g, \#h)$ and $s = \log(\frac{1}{\varepsilon} d \log H)$. The algorithm of Theorem 9.1.3 uses $\tilde{O}(\log^3 s)$ operations on s-bit integers to get a prime number and*

- $O(d(\#\ell + \log_s H))$ *operations on s-bit integers, or*
- $O(\#f\#\ell^\gamma \log d + (t + \#\ell)\log_s H)$ *operations on s-bit integers.*

For polynomials over finite fields, the problem is to have enough points in the finite field to ensure a good success probability. In the general case, the idea is to go to a field extension. Note that in the next statement, the value δ is actually < 1 for large fields. This is the case where no field extension is required.

Corollary 9.1.5. *Let f, g, h, ℓ be as in Theorem 9.1.3, with $\mathsf{R} = \mathbb{F}_q$. Let $t = \max(\#f, \#g, \#h)$ and $\delta = \log_q(\frac{1}{\varepsilon}d)$. The algorithm of Theorem 9.1.3 uses $\widetilde{O}(\delta^3 \log q)$ operations in $\mathbb{F}_q$ to get a degree-δ irreducible polynomial, and*

- *$O(d(\#\ell + \mathsf{M}(\delta)))$ operations in $\mathbb{F}_q$, or*
- *$O(\#f\#\ell^\gamma \mathsf{M}(\delta) \log d)$ operations in $\mathbb{F}_q$.*

To improve the complexity for small finite fields in the dense case, one can actually do without field extension. We reuse the idea of replacing an evaluation on α by a reduction modulo a random polynomial. The goal becomes to test if a random polynomial r divides $h - fg \bmod \ell$. We can once again rely on evaluation, but now the polynomials must be evaluated on a matrix. Indeed, a polynomial r is always minimal polynomial of its companion matrix C_r. This means that for any polynomial b, r divides b if and only if $b(C_r) = 0$. To get a fast algorithm, one cannot directly evaluate h and $fg \bmod \ell$ on C_r since this requires to perform some matrix multiplications in dimensions k where k is the size of r. Yet $h(C_r)$ and $(fg \bmod \ell)(C_r)$ are two (implicit) matrices whose equality must be checked. The standard solution is Freivalds' algorithm [67]: Take a random vector $\boldsymbol{v} \in \{0,1\}^k$ and test whether $\boldsymbol{v} \cdot h(C_r) = \boldsymbol{v} \cdot (fg \bmod \ell)(C_r)$. Using the same algorithm as before to evaluate $fg \bmod \ell$ on C_r, the final algorithm checks the equality

$$\boldsymbol{v} \cdot \boldsymbol{C}_r \cdot \boldsymbol{h} = \boldsymbol{v} \cdot \boldsymbol{C}_r \cdot M_{f \bmod \ell} \cdot \boldsymbol{g}.$$

For dense polynomials f and g and small fields, this technique improves the complexity by replacing the term $\mathsf{M}(\delta)$ by δ by using a random irreducible polynomial r. The same technique can also be used for another goal. When using field extensions, the modular product is verified using an algorithm that itself performs some modular products (for the computations in the finite field extension). Although this is not a problem in theory, it may be seen as unsatisfactory. The above algorithm does not rely on finite fields extensions but still performs some polynomial arithmetic to produce an irreducible polynomial r. To get rid of this dependency, we take for r any random polynomial without any irreducibility test. The same algorithm works as well albeit with worse success probability, and must therefore be repeated. The evaluation on a random companion matrix provides the following results.

Theorem 9.1.6 ([J5]). *Let $f, g, h, \ell \in \mathbb{F}_q[x]$, $\deg(f), \deg(g), \deg(h) \leq \deg(\ell) = d$, $0 < \varepsilon < 1$ and $\delta = \log_q(\frac{1}{\varepsilon}d)$. We can check whether $h = fg \bmod \ell$ in*

- *$O(d(\#\ell + \delta))$ operations in $\mathbb{F}_q$, or*
- *$O((\#\ell + \delta^2)d)$ operations in $\mathbb{F}_q$ without any polynomial multiplication, or*
- *$O(\#f\#\ell^\gamma \delta^3 \log(d))$ operations in $\mathbb{F}_q$ without any polynomial multiplication,*

with a probability of error $\leq \varepsilon$ if $h \neq fg \bmod \ell$.

9.1.3 Dense and sparse polynomial product verification

Let us come back to standard polynomial product verification. As mentioned earlier, the standard technique to test whether $h = f \times g$ is to test $h(\alpha) = f(\alpha)g(\alpha)$ for a random

point α. In the dense case, this is done in an optimal linear number of operations in the base ring as long as it is large enough. For small finite fields for instance, the complexity is not optimal anymore since it requires an extension field. For polynomial over $\mathbb{Z}$, the difficulty is the possible growth of the intermediate values, that requires to work modulo some prime number. And finally, this approach is not suitable at all for sparse polynomials where the evaluation is too costly.

To overcome these difficulties, the idea is to replace evaluation by modular reduction. Test if $h \bmod \ell = f \times g \bmod \ell$ for some random polynomial ℓ. The idea of Kaminski is to use for ℓ a polynomial $x^i - 1$ for some random i between d^{1-e} and $2d^{1-e}$ for some $0 < e < \frac{1}{2}$. Then, f, g and h are reduced modulo $x^i - 1$, and the product $f \times g \bmod x^i - 1$ is computed explicitly.

Theorem 9.1.7 ([133]). *Let $f, g, h \in \mathsf{R}[x]$ of degree $< n$ and $0 < e < \frac{1}{2}$. One can test whether $h = f \times g$ using $O(n)$ operations in R, with failure probability $O(\log\log n / n^{1-2e})$.*

Kaminski's algorithm has one main drawback. It relies directly on a polynomial multiplication algorithm to compute $f \times g \bmod x^i - 1$. And since $e < \frac{1}{2}$ the multiplication algorithm must have subquadratic complexity. Also, analyzed in the bit complexity model for $\mathsf{R} = \mathbb{Z}$ or $\mathsf{R} = \mathbb{F}_q$, the resulting algorithm is not optimal anymore. Using modular product verification instead, we can on the one hand get rid of any multiplication algorithm and on the other hand obtain a linear bit complexity. This fails to be true if the coefficients are extremely large compared to the degree of the polynomial. But for this case, we can rely on the integer product verification algorithm of Kaminski. Altogether we obtain the following optimal results.

Theorem 9.1.8 ([J5]). *Let $f, g, h \in \mathsf{R}[x]$ of degree $< n$, where R is either $\mathbb{Z}$ or $\mathbb{F}_q$. For $\mathsf{R} = \mathbb{Z}$, let $q = \max(\mathsf{h}(f), \mathsf{h}(g), \mathsf{h}(h))$. One can check if $h = f \times g$, with failure probability at most $\varepsilon = 1/n^{O(1)}$ if $h \neq f \times g$, in $O(n \log q)$ bit operations and without performing any polynomial multiplication.*

For sparse polynomials, we also rely on testing if $h \bmod \ell = f \times g \bmod \ell$ but this time $\ell = x^p - 1$ where p is a prime number. In this case, we are not able to get optimal results, only quasi-linear complexities.

Theorem 9.1.9 ([J5]). *Let $f, g, h \in \mathsf{R}[x]$ of degree $< n$ and sparsity at most t, where R is either $\mathbb{Z}$ or $\mathbb{F}_q$. For $\mathsf{R} = \mathbb{Z}$, let $q = \max(\mathsf{h}(f), \mathsf{h}(g), \mathsf{h}(h))$. Let $s = t(\log n + \log q)$ be the input size.*
One can check if $h = f \times g$, with a failure probability at most ε if $h \neq f \times g$, in

- *$O(s \log s \log\log s)$ bit operations if $\mathsf{R} = \mathbb{Z}$, or $\mathsf{R} = \mathbb{F}_q$ with $q < \frac{c}{\varepsilon}(\#f\#g + \#h)\log n)$, and*
- *$O(s \log^2 s)$ bit operations if $\mathsf{R} = \mathbb{F}_q$ and $q \geq \frac{c}{\varepsilon}(\#f\#g + \#h)\log n)$,*

where c is some constant. In the first case, the complexity becomes $O(s \log\log s)$ as soon as $t = O(\log^k n)$ for some k.

Remark. The input size $s = t(\log n + \log q)$ may be a pessimistic upper bound for polynomials over $\mathbb{Z}$, when the coefficients are very unbalanced. If s is replaced by the actual bit size of the sparse representations of f, g and h, the algorithm has complexity $O(s \log s \log\log s + s \log n \log\log n)$ [C12].

9.2 Sparse polynomial multiplication

In this section, we investigate the question of sparse polynomial multiplication. Given two sparse polynomials f and $g \in \mathsf{R}[x]$, how fast can we compute their product $h = f \times g$? The naive algorithm requires $O(\#f \cdot \#g)$ products between the coefficients of f and g. The exact complexity depends on the data structures used by the algorithm to represent the polynomials, and some work has been done to minimize it [118, 162]. Our goal is to provide an asymptotic improvement.

The difficulty for such an improvement is that the algorithm has to be *output-sensitive*. The reason is that the size of the output is not determined by the sole sizes of the inputs, contrary to, say, dense polynomial multiplication. Even the support is not enough to determine the size of the output, and one must also take into account the coefficients.

Example. Let $f = \sum_{i=0}^{t-1} x^i$, $g_1 = \sum_{i=0}^{t-1}(x^{ti+1} - x^{ti})$ and $g_2 = \sum_{i=0}^{t-1}(x^{ti+1} + x^{ti})$, in $\mathbb{Z}[x]$. While g_1 and g_2 have the same support, $f \times g_1 = x^{t^2-1} - 1$ has only two monomials while $f \times g_2 = x^{t^2} + \sum_{i=1}^{t^2-1} 2x^i + 1$ has sparsity t^2.

With this example in mind, one cannot improve on the upper bound $O(\#f \cdot \#g)$ for the worst-case complexity. The goal is to provide an output-sensitive bound that depends on $t = \max(\#f, \#g, \#h)$ where $h = f \times g$.

The idea to perform fast sparse multiplication is to rely on sparse interpolation. As noticed by several authors, one difficulty is due to the cancellations between coefficients. These cancellations are the reason why the support of the output cannot be computed from the sole support of the inputs. This motivated Arnold and Roche to introduce the *structural support* of a sparse polynomial product, defined as the sumset of the supports of the inputs. Formally, if S_f and S_g denote the supports of f and g, the structural support of $f \times g$ is the set $S_f + S_g = \{a + b : a \in S_f, b \in S_g\}$. They prove the following result.

Theorem 9.2.1 ([13]). *Let $f, g \in \mathbb{Z}[x]$ of size n and height $\leq h$. Let $t = \max(\#f, \#g, \#(fg))$ and s be the size of the structural support of $f \times g$. The polynomial $f \times g$ can be computed by a randomized algorithm in $\widetilde{O}(t \log h + s \log n)$ bit operations.*

The previous example shows the shortcoming of this result. The structural support may have size $O(t^2)$ while the actual output size is constant. Therefore, this algorithm remains quadratic in t in the worst case. We can remark that the structural support coincides with the support when the input polynomials only have nonnegative coefficients. In the discrete algorithms community, the problem is studied under the name *sparse convolution* and many results focus on the case of nonnegative coefficients [37, 50]. In particular, some efforts are made to make the algorithms deterministic [6, 38]. The most recent results [117] provide very precise complexity bounds in the word RAM model with $O(\log t + \log d)$-bit words where d is the input degree. Roughly speaking, a bit-complexity bound can be obtained by multiplying the result by $\widetilde{O}(\log t + \log d)$.

Theorem 9.2.2 ([117]). *Let $f, g \in \mathbb{Z}_{\geq 0}[x]$ of size n and height $\leq h$. The polynomial $f \times g$ can be computed by a Las Vegas randomized algorithm that uses $O(t \log t)$ word operations (with high probability), where $t = \#(f\,g)$.*

Finally, these results have also been investigated in the real RAM. This model allows for exact computations on real numbers, but is restricted to purely algebraic computations. This forces to replace some crucial steps of a Prony-like interpolation. Nevertheless, the obtained complexity still remains similar as in the previous works.

Theorem 9.2.3 ([64]). *Let $f, g \in \mathbb{R}_{\geq 0}[x]$ of degree d. The polynomial $f \times g$ can be computed by a Las Vegas algorithm in $\tilde{O}(t)$ operations on a real RAM (with high probability), where $t = \#(f\,g)$.*

Our goal is to get rid of either the structural sparsity in the complexity, or nonnegativity assumption. Since it is difficult to predict the size of the output, our approach is to guess it and check afterwards. More precisely, we assume an upper bound on the output sparsity, compute a tentative result using this upper bound, and check the result using our sparse polynomial product verification algorithm. The initial upper bound is set to equal the size of the inputs, and is doubled as long as the correct result is not found. Note that since the verification algorithm is randomized with some false-positive probability, this requires some careful probability analysis of the overall algorithm. In particular, a failure probability of ε in the verification algorithm does not imply the same failure probability for the overall algorithm.

Theorem 9.2.4 ([C5]). *Let $f, g \in \mathbb{Z}[x]$ of degree $\leq d$ and height $\leq h$. The polynomial $f \times g$ can be computed by an Atlantic City randomized algorithm that uses $\tilde{O}(t(\log d + \log h))$ bit operations where $t = \#(f\,g)$ (with high probability).*

As mentioned earlier, the upper bound $t(\log d + \log h)$ may be a pessimistic upper bound. Using our unbalanced interpolation algorithm, we can refine the bound.

Theorem 9.2.5 ([C12]). *Let $f, g \in \mathbb{Z}[x]$ of degree $\leq d$. The polynomial $f \times g$ can be computed by an Atlantic City randomized algorithm that uses $\tilde{O}(\ell \log d)$ bit operations where $\ell = \max(\mathit{bitsize}(f), \mathit{bitsize}(g), \mathit{bitsize}(f \times g))$ (with high probability).*

Due to the extra factor $\log(d)$, the algorithm is not quasi-linear for sparse polynomials. Indeed, the bit size may already contain a $\log(d)$ factor. But for moderate degrees, in particular for dense polynomials, the algorithm has quasi-linear complexity in the input and output sizes. This provides the first multiplication algorithm for (dense) polynomials over $\mathbb{Z}$ with quasi-linear complexity in the actual size of the input. All the previous approaches, which do not take the unbalancedness into account, have a worst-case complexity $\Omega(\ell^2)$.

To obtain a fully quasi-linear time algorithm, we would need to also take into account the unbalancedness of the exponents, rather than using a bound d on each of them. Some recent work investigates this problem [101], but using the pessimistic bound on the coefficients. Combining both results is a challenging open problem.

Open problem 9.1. Given $f, g \in \mathbb{Z}[x]$, can we compute their product $h = f \times g$ in quasi-linear time in the bit size of the sparse representations of f, g and h?

Another question concerns sparse polynomials over finite fields. Currently, the best known sparse interpolation algorithms over finite fields do not have quasi-linear complexity. Therefore, using them for sparse multiplication does not provide a quasi-linear algorithm. Some partial results are known, for instance for not-too-sparse polynomials [117]. Yet the general case remains open.

Open problem 9.2. Given $f, g \in \mathbb{F}_q[x]$, can we compute their product $h = f \times g$ in quasi-linear time in the bit size of the sparse representations of f, g and h?

A promising approach is to use some techniques developed for the real RAM [64]. In particular, due to some limitations of the real RAM, Fischer bypasses the root finding part of Prony's algorithm, which is the only non-quasi-linear part of Huang's fastest sparse interpolation algorithm over finite fields (Theorem 7.3.3).

9.3 Sparse polynomial division and divisibility

Sparse polynomial division suffers from the same difficulty of output-sensitivity as multiplication, but to a much larger scale. Indeed, the output size of a division may vary from constant to exponential in the size of the inputs. This is the case for the sparsity and the height.

Example. Let $f_1 = x^{2d+1} - x^{2d} + x^d$, $f_2 = x^{2d+1} + x^{2d} + x^d$, $f_3 = x^{2d+1} - x^{2d}$, and $g = x^{d+1} - x^d + 1$ in $\mathbb{Z}[x]$. Then

- g divides f_1 and the quotient is $q_1 = x^d$;
- g does not divide f_2 and the quotient is $x^d + \sum_{i=0}^{d-1} 2x^i$ and the remainder is $2x^d - \sum_{i=0}^{d-1} 2x^i$;
- g does not divide f_3 and the quotient is $x^d + \sum_{i=0}^{d-1} 2^{d-i-1}x^i$ and the remainder is $(2^d - 3)x^d - \sum_{i=0}^{d-1} 2^{d-i-1}x^i$.

In part due to the extreme variability of the output size, a fast output-sensitive algorithm for Euclidean division of sparse polynomials remains elusive. As a first step, we focus on the simpler problem of exact division. There are two distinct problems. The first one is to compute the quotient f/g when g is known to divide f. The second one is to decide g divides f.

The first problem can be tackled with similar techniques as multiplication. The idea is thus to interpolate $h = f/g$ while the division is exact. Nevertheless, two additional difficulties appear. The first one is that the interpolation algorithms need to evaluate q. The evaluation points must not be roots of g. Since the evaluations are performed on pth roots of unity in $\mathbb{F}_q$ or $\mathbb{Z}/q^{2k}\mathbb{Z}$ for some primes p and q and integer k, it is sufficient for g to be coprime with $\phi_p = \sum_{i=0}^{p-1} x^i$ in $\mathbb{F}_q[x]$. A careful choice of p and q makes this event most likely. The second difficulty concerns the size growth of the quotient. For multiplication algorithms, the sparsity can increase quadratically. In the case of exact

division, the sparsity can increase exponentially. Also, for polynomials over $\mathbb{Z}$ the size of the coefficients can increase a lot. Although recent results have reduced the bounds from exponential to polynomial when the division is exact [166, 167], quasi-linear time algorithms require two *a priori* bounds, one on the sparsity and one on the height. If the result appears to be incorrect, one of the two bounds has to be increased. The difficulty is to identify which bound is too small. It happens that detecting if the bound on the height is too small can actually be performed inside the interpolation algorithm. As a result, there are two distinct tests at two distinct steps to identify which bound(s) must be increased. For polynomials over finite fields, no coefficient growth occurs. But no quasi-linear sparse interpolation algorithm is known.

Theorem 9.3.1 ([C7, C9]). *Let $f, g \in \mathsf{R}[x]$ such that g divides f, $d = \deg(f)$ and $t = \max(\#f, \#g, \#(f/g))$. The polynomial f/g can be computed by an Atlantic City randomized algorithm that uses*

- $\tilde{O}(t(\log d + \log h))$ *bit operations if* $\mathsf{R} = \mathbb{Z}$ *and* $h = \max(\mathsf{h}(f), \mathsf{h}(g), \mathsf{h}(f/g))$;
- $\tilde{O}(t \log d \log q)$ *bit operations if* $\mathsf{R} = \mathbb{F}_q$ *has characteristic* $> d - \deg(g)$;
- $\tilde{O}(t \log^2(d)(\log d + \log q))$ *bit operations if* $\mathsf{R} = \mathbb{F}_q$ *has characteristic* $\leq d - \deg(g)$.

This theorem provides partial results. Beyond improving the complexities in the case of finite field, it is not yet known how to adapt the algorithm for polynomials over $\mathbb{Z}$ with unbalanced coefficients, as in the case of polynomial multiplication. A much more ambitious goal is to extend the result to the general euclidean division.

Open problem 9.3. Given $f, g \in \mathsf{R}[x]$, can we compute the quotient q and remainder r such that $f = gq + r$ in time quasi-linear in the sparse representations of f, g, q and r?

The second problem of testing whether g divides f remains open in full generality. Some special cases are easy though. When $\deg(g)$ or $\deg(f) - \deg(g)$ are small, the long division algorithm has polynomial running time, since in the first case the remainder cannot be large, and in the second case the quotient cannot be large. We extend these results to more general cases. In particular, we prove sparsity bounds on the quotient based on the structure of the divisor g. If $g = 1 + x^k g_1$ for some sufficiently large k, the quotient is sparse. Using the fact that g divides f if and only if $g^{\leftarrow} = x^{\deg(g)} g(1/x)$ divides $f^{\leftarrow} = x^{\deg(f)} f(1/x)$, the same holds if $g = x^d + g_1$ for a low-degree g_1. These bounds directly translate into polynomial-time algorithms. The bounds fail for more general divisors $g = g_0 + x^k g_1$. Nevertheless, using a factorization $g = g_0 \cdot (1 + x^k g_1/g_0)$, we prove that testing divisibility by such divisors is still polynomial. The recursive use of this idea gives our most general result. One can test divisibility by a divisor g if there exists in g a low-degree polynomial g_1 surrounded by large *gaps*.

Theorem 9.3.2 ([C7]). *Let f and $g \in \mathsf{K}[x]$ be two polynomials of respective degrees $m+n$ and m, and sparsity at most t. One can check whether g divides f in deterministic polynomial time if g can be written as $g_0 + x^k g_1 + x^\ell g_2$ where $g_0, g_1, g_2 \in \mathsf{K}[x]$ satisfy $\deg(g_1) = \mathrm{poly}(t \log(m+n))$, $k = \deg(g_0) + \Omega(n)$ and $\ell = \deg(x^k g_1) + \Omega(n)$.*

A general result remains out of reach.

Open problem 9.4. Given two sparse polynomials $f, g \in \mathsf{K}[x]$, what is the complexity of testing whether g divides f? Is it possible to perform the computation in polynomial time in the sparse representation of f and g? Is the problem NP-hard?

9.4 Factorization

In this section, we focus on a computationally more difficult problem, namely the factorization of sparse polynomials. Many open questions exist in this domain. In particular, it is not yet known how sparse or dense are the irreducible factors of sparse polynomials. More generally, understanding how the structure of a polynomial reflects in the structure of its irreducible factors is a challenging question. A very recent striking result provides some answers to this question in terms of arithmetic circuit depth [22].

Over finite fields, it is NP-hard (under randomized reductions) to decide whether a given sparse polynomial has roots, that is degree-1 factors, in the base field [23]. In characteristic zero, more computational results are known. A first algorithm was developed to compute the roots of sparse polynomials over $\mathbb{Z}$ [56]. This result was extended to $\mathbb{Q}$ and number fields, and to low-degree factors [154]. By generalizing the latter technique, it was shown possible to extend these results to multivariate polynomials as well [124, 125].

We propose another approach for multivariate polynomials. It is combinatorial in nature and applies to any field of characteristic zero. We prove that computing low-degree factors of multivariate polynomials *reduces* to the univariate case and to low-degree factorization. We distinguish two families of factors. A polynomial $f \in \mathsf{K}[x_1, \ldots, x_n]$ is *unidimensional* if it can be written $\boldsymbol{x}^{\boldsymbol{\mu}} f_u(\boldsymbol{x}^{\boldsymbol{\nu}})$ for some univariate polynomial f_u and multivariate monomials $\boldsymbol{x}^{\boldsymbol{\mu}}$ and $\boldsymbol{x}^{\boldsymbol{\nu}}$, and *multidimensional* otherwise.

The case of unidimensional factors is reduced to the univariate case. The reduction is straightforward. Unidimensional factors of a polynomial are factors of its *unidimensional components*, that can be computed easily from the support of the polynomial. Now a unidimensional factorization problem is equivalent to its univariate counterpart. The required algorithms for sparse univariate factorization are only known for $\mathbb{Z}$, $\mathbb{Q}$ and more generally number fields.

The case of multidimensional factors is also a reduction, this time to low-degree factorization. Algorithms for this problem are known for a large variety of base rings: $\mathbb{Z}$ or $\mathbb{Q}$ [153], number fields [148, 151, 152], real and complex numbers [130, 164, 174], the p-adic numbers [42] or the algebraic closure of $\mathbb{Q}$ (absolute factorization) [31, 47].

The approach for multidimensional factors can be seen as a generalization of the first root-finding algorithm for sparse polynomials [56], based on a so-called *Gap Theorem*. They consider a sparse univariate polynomial $f \in \mathbb{Z}[x]$. Assuming that $f = f_1 + x^k f_2$ where $k \gg \deg(f_1)$, they prove that any root $\alpha \in \mathbb{Z} \setminus \{-1, 0, 1\}$ of f must be a root of both f_1 and f_2. The argument is that whenever α is not a root of f_2, the integer $\alpha^k f_2(\alpha)$ is a much

larger integer than $f_1(\alpha)$ in absolute value, whence their sum cannot vanish. A recursive application of this result reduces the computation of the roots of f to the computation of the roots of its low-degree parts.

To mimic this technique to compute low-degree factors of multivariate polynomials, we need to consider roots of such factors. Consider first a bivariate polynomial $f \in \mathsf{K}[x,y]$ over some field K of characteristic 0, and some irreducible factor g of f. Viewing g as a polynomial in $\mathsf{K}[x][y]$, it has a root ψ in the field of Puiseux series $\overline{\mathsf{K}}\langle\langle x\rangle\rangle$. This root satisfy $g(x,\psi(x)) = 0$, therefore $f(x,\psi(x)) = 0$ as well. The notion of size used in the integer-case argument is played here by the valuation of the Puiseux series, that is the largest power of x that divides the series. We have the following result.

Theorem 9.4.1 ([C1, J3]). *Let $f = \sum_{j=1}^{\ell} c_j x^{\alpha_j} y^{\beta_j} \in \mathsf{K}[x,y]$ and $g \in \mathsf{K}[x,y]$ of individual degree d that does not divide f. There exists an explicit bound $\gamma_v(d,\ell) = O(d^2\ell^2)$ such that for any root $\psi \in \overline{\mathsf{K}}\langle\langle x\rangle\rangle$ of g, the valuation of $f(x,\psi(x))$ is at most $\min_{1\le j\le \ell}(\alpha_j + v\beta_j) + \gamma_v(d,\ell)$.*

To use it for multivariate polynomial, we single out two variables from n and work with the polynomial ring $\mathsf{K}(\boldsymbol{x} \setminus \{x_i,x_j\})[x_i,x_j]$ where $\mathsf{K}(\boldsymbol{x} \setminus \{x_i,x_j\})$ denotes the field of rational functions in $n-2$ variables. To handle all multidimensional factors, that may depend on only two variables each, we actually consider all the $\binom{n}{2}$ possible pairs of variables. The theorem is only proved in the case of a field of characteristic zero. For positive characteristic, one needs to consider roots as more general power series, namely Hahn series [137]. It is plausible that a generalization of our techniques can prove the result for some finite fields, at least when the characteristic is larger than the degree of f. This would provide a generalization of our previous results that allow to compute multilinear factors of sparse polynomial over finite field of large characteristic [J4].

Combining both the unidimensional case and the multidimensional case, we obtain the following algorithm.

Theorem 9.4.2 ([C1, J3]). *Let $f \in \mathsf{K}[x_1,\ldots,x_n]$ be a degree-D sparsity-t polynomial, where K is a field of characteristic zero.*

i. *Computing its irreducible degree-$\le d$* unidimensional *factors can be reduced to computing the degree-$\le d$ irreducible factors of* $\mathrm{poly}(t,n,\log D,d)$ *univariate polynomials of degree $\le D$ and sparsity $\le t$, plus* $\mathrm{poly}(t,n,\log D,d)$ *bit operations.*

ii. *Computing its irreducible degree-$\le d$* multidimensional *factors can be reduced to the irreducible factorization of at most t polynomials of degree* $\mathrm{poly}(t,d)$*, plus* $\mathrm{poly}(t,n,\log D,d)$ *bit operations.*

As a consequence, if K is some number field, one can compute all the irreducible factors of f of degree $\le d$ in time polynomial in the input size.

The natural generalization of the previous result is to put a bound on the sparsity of the factors, rather than on their degree.

Open problem 9.5. Given a polynomial $f \in \mathsf{K}[x_1, \ldots, x_n]$ of degree D and sparsity t, and a bound τ, can we compute all the irreducible factors of f of sparsity at most τ in time polynomial in the input and output size?

Conclusions and perspectives 10

We have shown that one can sometimes provide fast algorithms for computing with sparse polynomials. For basic arithmetic, one tool of choice is sparse interpolation which plays to some extent the same role for sparse polynomials as the FFT for dense polynomials. The comparison ends quickly. On the one hand, quasi-linear sparse interpolation is known only for polynomials over the integers, but not over finite fields for instance. On the other hand, many computational problems about dense polynomials reduce to polynomial multiplication, itself based on FFT in the asymptotic regime. For sparse arithmetic, only multiplication and exact division have been reduced to sparse interpolation for now.

10.1 Sparse interpolation

The main challenge in sparse interpolation is to have an algorithm that has quasi-linear complexity for polynomials over finite fields.

Open problem 10.1. Is there a quasi-linear time algorithm for sparse interpolation over (some, or all) finite fields? In which model(s)?

A priori, the algorithm is given access to a black box with derivative to avoid the exponential lower bound of Proposition 7.3.2. Yet this lower bound does not apply directly to regular black box interpolation. Indeed, it relies on the fact that the polynomial f is evaluated on some values ω^i where i is known. This includes Algorithm 7.1.2 (SparseInterpolationBB) but also includes any algorithm that would for instance evaluate f on random values. Indeed, sampling uniformly $r \in \mathsf{R}$ can be done by first sampling an integer i and setting $r = \omega^i$. But it does not rule out an algorithm that would for instance evaluate f on an arithmetic progression. No such algorithm is known yet, and it is not clear that few evaluations are sufficient.

Open problem 10.2. Let $f = \sum_{i=0}^{t-1} c_i x^{e_i} \in \mathsf{R}[x]$ and let $(r_i)_i$ be a sequence of elements of R in arithmetic progression. What is the smallest possible s such that f can be reconstructed uniquely from $(f(r_i))_{0 \le i < s}$?

This open question may receive different answers depending on the ring R. It can also be generalized and phrased as a standard interpolation problem.

Open problem 10.3. Given s evaluations of a t-sparse degree-d polynomial $f \in \mathsf{R}[x]$, on what condition(s) on s and the evaluation points can f be reconstructed?

Geometric black box interpolation shows that it is possible if $s \ge 2t$ and the evaluation points are in geometric progression. Lagrange interpolation theorem states that $s > d$ is

sufficient, whatever the points are. Other cases are treated by Borodin and Tiwari [30] although they do not solve the problem in full generality. One intriguing special case arises from the study of sparse polynomial interpolation codes [131]. If $\mathsf{R} = \mathbb{R}$, Descartes' rule of signs [57] implies that f has at most $t-1$ positive real roots and at most $t-1$ negative real roots. This implies for instance that the value of f on $2t$ positive real numbers uniquely defines it. (If two t-sparse polynomials f_1 and f_2 are consistent with the evaluations, their difference $f_1 - f_2$ has $2t$ positive real roots and at most $2t$ monomials, hence vanishes.) But it is not clear how to compute f from its evaluations. To stick with exactly representable polynomials, we phrase the question for polynomials with rational coefficients. In this case, an exhaustive search is possible. Since $\mathbb{Q}[x]$ is countable, one can enumerate all t-sparse polynomials in a certain order and recompute the evaluations to compare. This algorithm is extremely inefficient.

Open problem 10.4. What is the complexity of computing the sparse representation of a t-sparse polynomial $f \in \mathbb{Q}[x]$, given as inputs $2t$ evaluations of f on positive rational numbers?

Similar bounds depending on the sparsity of the polynomials have been proved when R is a number field or a p-adic field [155], and the same question of sparse interpolation can be asked. On the other hand, no such bound is known, nor even possible, in the case of finite fields [23, 45].

10.2 Sparse polynomial arithmetic

Computing with sparse polynomials remains a challenge. We can identify at least two reasons. First the complexity of many basic operations is still unclear. For instance, due to a lack of quasi-linear sparse interpolation algorithm over finite fields, it is currently not known if a quasi-linear multiplication algorithm exists for sparse polynomials over finite fields. Second the structures of the problems are not well understood. As mentioned earlier, one difficulty in sparse arithmetic is the output sensitivity. Some bounds are known for the output size for some problems (multiplication, exact division, ...) but their tightness is not always known. For instance, some non-trivial bounds exist on the bit size of the exact quotient of two sparse polynomials. But the case of Euclidean division in much less understood and no reduction of Euclidean division to sparse interpolation is currently known. As for exact division, the goal would be to have a fast output-sensitive algorithm. A related decision problem is wide open too: Given two sparse polynomials f and g, does g divide f? We have given some partial results but the general case still seems out of reach.

Structural questions also arise for polynomial factorization, or more simply root finding. Giving tight upper bounds on the number of roots of sparse polynomials is very challenging. The theory of fewnomials attempts to provide such results for systems of polynomials [140]. But even very basic questions remain open. For instance, consider polynomials over the real numbers. As a consequence of Descartes' rule of signs [57], a polynomial f with t nonzero terms has at most $2t-2$ nonzero roots, irrespective of its degree. The product fg of two such polynomials has at most t^2 nonzero terms, so Descartes' rule implies that it has $O(t^2)$ nonzero roots. But of course, the roots of fg are roots of f or g, therefore a tighter (and attained) bound is $4t-4$. But the argument fails for the very close polynomial $fg+1$ for instance.

Open problem 10.5. Let $f, g \in \mathbb{R}[x]$ with t nonzero terms each. What is the maximal number of roots of the polynomial $fg + 1$? In particular, is it always linear? Or are there examples where the number of roots is super-linear, even quadratic?

This and similar questions actually have implications in algebraic complexity theory [142]. The same holds for a variant that asks for the number of edges in the Newton polygon of a bivariate polynomial. For this variant, an upper bound $O(t^{\frac{4}{3}})$ can be proven instead of the trivial bound $O(t^2)$ [143]. Results are also known for random polynomials [39]. But the original question remains completely open.

For polynomial factorization, it is a notoriously hard problem to bound the sparsity of factors of sparse polynomials [73]. From the perspective of algorithms, no algorithm is known to output the sparse factors of a sparse polynomial. Very recent results on the factorization of low-degree polynomials represented by circuits [22] suggest a natural approach. Given a sparse polynomial, are there some low-depth (but high-degree!) arithmetic circuits for its irreducible factors? And can we compute them?

10.3 Sparse interpolation, error-correcting codes, and cryptography

Finally, outside polynomial arithmetic, sparse interpolation has strong links with the field of error-correcting codes [51] and applications in cryptography [65]. Sparse interpolation may be seen as an instance of the *Syndrome Decoding Problem* (see for instance [208]) underlying Niederreiter cryptosystem [169].

Consider a linear code C. A message $\boldsymbol{m}$ is encoded as a codeword $\boldsymbol{c}$. The code has a parity-check matrix H such that $H \cdot \boldsymbol{v} = \mathbf{0}$ if and only if $\boldsymbol{v}$ is a codeword. The codeword is sent through some channel and the received word is $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e}$ where $\boldsymbol{e}$ is an error vector. Given $\boldsymbol{r}$ and H, one can compute the *syndrome* $\boldsymbol{s} = H \cdot \boldsymbol{r} = H \cdot \boldsymbol{e}$ using a matrix-vector product. The Syndrome Decoding Problem (SDP) asks to reconstruct the sparse vector $\boldsymbol{e}$ from the syndrome $\boldsymbol{s}$, knowing the parity-check matrix. Niederreiter cryptosystem uses this problem with two parity-check matrices. The first one is the parity-check matrix of code for which SDP has an efficient algorithm. It serves as the secret key. The second parity-check matrix some randomly scrambled version of the first one that (hopefully) has lost all its structure due to randomization. It serves as the public key. To encrypt some data, it is first mapped to an error vector $\boldsymbol{e}$, that is a sparse vector. No codeword is used, or rather the only codeword is the all-zero vector. The encryption is the computation of the syndrome, as a matrix-vector product using the random-looking parity-check matrix. The decryption reduces to solving SDP with the structured parity-check matrix. The efficiency of decryption relies on a fast algorithm for SDP with this matrix, while the security relies on the hardness of SDP for the random-looking parity check matrix.

Sparse interpolation, and more precisely black box sparse interpolation, can be seen as solving SDP where the parity check matrix is

$$H = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega & \dots & \omega^{n-1} \\ 1 & \omega^2 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{2t-1} & \dots & \omega^{(2t-1)(n-1)} \end{pmatrix}.$$

Indeed, viewing $\boldsymbol{e}$ as a sparse polynomial $f = \sum_{j=0}^{t-1} e_{i_j} x^{i_j}$, the syndrome $\boldsymbol{s} = H \cdot \boldsymbol{e}$ corresponds to the vector of evaluations $(f(1), f(\omega), \ldots, f(\omega^{2t-1}))$. Therefore, computing sparse vector $\boldsymbol{e}$ from the syndrome $\boldsymbol{s}$, or computing the sparse polynomial f from its evaluations on $1, \omega, \ldots, \omega^{2t-1}$, are exactly equivalent. Note that the parity-check matrix H corresponds to a *normalized Reed-Solomon code* [192].

Beyond this formal equivalence, sparse interpolation can be used in cryptography for *oblivious ciphertext compression* and *decompression* [24, 65]. Ciphertext compression takes as input a sparse vector $\boldsymbol{v}$, encrypted under a linearly homomorphic encryption scheme (LHE), and outputs a compressed (still encrypted) version of $\boldsymbol{v}$. Upon reception, the owner of the private key can decrypt and decompress to reconstruct $\boldsymbol{v}$. We interpret $\boldsymbol{v}$ as an error vector, and compress it as the syndrome. This requires a product between a clear matrix and an encrypted vector, which is possible using an LHE. Once the syndrome is decrypted, the reconstruction of the vector $\boldsymbol{v}$ is an instance of SDP.

A ciphertext compression protocol can be used in a Private Information Retrieval protocol (PIR) [48]. In such a protocol, a server stores a database and a client makes queries to the database to retrieve some entries. But the server must not learn which entries were queried. One inefficient solution would be for the server to send the whole database. The objective of a PIR protocol is to minimize the amount of the data that the server must send to enable the client retrieve its entries. Without going into more details, some PIR protocols have a first step where the server gets the queries as an encrypted sparse vector, containing a 1 exactly for those entries that have been queried [172]. Then, the server computes a pointwise (homomorphic) product between the database and the query vector. This results in an encrypted sparse database containing only the relevant entries. Ciphertext compression allows then the server to send this sparse database with a low communication cost. Similar techniques can also be used for searchable encryption [187] where the database is now encrypted.

The interpretation of sparse interpolation as a special case of the Syndrome Decoding Problem forms the basis of ongoing work with Pascal Giorgi (U. Montpellier) and Mark Simkin (Aarhus U.) to improve the current state-of-the-art in ciphertext compression and decompression, with applications to PIR protocol and searchable encryption.

Publication list

Journal publications

[J1] Bruno Grenet, Joris van der Hoeven, and Grégoire Lecerf. "Deterministic Root Finding over Finite Fields Using Graeffe Transforms". *Applicable Algebra in Engineering, Communication and Computing* 27.3 (2016), pp. 237–257. DOI: 10.1007/s00200-015-0280-5 (pp. 64, 75).

[J2] Bruno Grenet. "Lacunaryx: Computing Bounded-degree Factors of Lacunary Polynomials". *ACM Communications in Computer Algebra* 49.4 (2016), pp. 121–124. DOI: 10.1145/2893803.2893807 (p. 64).

[J3] Bruno Grenet. "Bounded-Degree Factors of Lacunary Multivariate Polynomials". *Journal of Symbolic Computation* 75 (2016), pp. 171–192. DOI: 10.1016/j.jsc.2015.11.013 (pp. 64, 95).

[J4] Arkadev Chattopadhyay, Bruno Grenet, Pascal Koiran, Natacha Portier, and Yann Strozecki. "Computing the Multilinear Factors of Lacunary Polynomials without Heights". *Journal of Symbolic Computation* 104 (2021), pp. 183–206. DOI: 10.1016/j.jsc.2020.04.013 (p. 95).

[J5] Pascal Giorgi, Bruno Grenet, and Armelle Perret du Cray. "Polynomial Modular Product Verification and Its Implications". *Journal of Symbolic Computation* 116 (2023), pp. 98–129. DOI: 10.1016/j.jsc.2022.08.011 (pp. 63, 87–89).

Conference publications

[C1] Bruno Grenet. "Computing Low-Degree Factors of Lacunary Polynomials: A Newton-Puiseux Approach". *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. 2014, pp. 224–231. DOI: 10.1145/2608628.2608649 (pp. 64, 95).

[C2] Bruno Grenet, Joris van der Hoeven, and Grégoire Lecerf. "Randomized Root Finding over Finite FFT-fields Using Tangent Graeffe Transforms". *Proceedings of the 2015 ACM International Symposium on Symbolic and Algebraic Computation*. 2015, pp. 197–204. DOI: 10.1145/2755996.2756647 (pp. 64, 74–76).

[C3] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. "Generic Reductions for In-place Polynomial Multiplication". *Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation*. 2019, pp. 187–194. DOI: 10.1145/3326229.3326249 (pp. 17, 25, 26, 28, 30–33, 36–38).

[C4] Bruno Grenet and Ilya Volkovich. "One (More) Line on the Most Ancient Algorithm in History". *Proceedings of the 2020 Symposium on Simplicity in Algorithms*. 2020, pp. 15–17. DOI: 10.1137/1.9781611976014.3 (p. 2).

[C5] Pascal Giorgi, Bruno Grenet, and Armelle Perret du Cray. "Essentially Optimal Sparse Polynomial Multiplication". *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*. 2020, pp. 202–209. DOI: 10.1145/3373207.3404026 (pp. 63, 91).

[C6] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. "Fast In-Place Algorithms for Polynomial Operations: Division, Evaluation, Interpolation". *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*. 2020, pp. 210–217. DOI: 10.1145/3373207.3404061 (pp. 17, 25).

[C7] Pascal Giorgi, Bruno Grenet, and Armelle Perret du Cray. "On Exact Division and Divisibility Testing for Sparse Polynomials". *Proceedings of the 2021 International Symposium on Symbolic and Algebraic Computation*. 2021, pp. 163–170. DOI: 10.1145/3452143.3465539 (pp. 63, 93, 94).

[C8] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. "Random Primes without Primality Testing". *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation*. 2022, pp. 207–215. DOI: 10.1145/3476446.3536191 (pp. 63, 79, 80).

[C9] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. "Sparse Polynomial Interpolation and Division in Soft-Linear Time". *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation*. 2022, pp. 459–468. DOI: 10.1145/3476446.3536173 (pp. 63, 79, 93).

[C10] Jean-Guillaume Dumas and Bruno Grenet. "In-Place Accumulation of Fast Multiplication Formulae". *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. 2024, pp. 16–25. DOI: 10.1145/3666000.3669671 (pp. 17, 43, 47, 50, 51, 54, 59).

[C11] Jean-Guillaume Dumas and Bruno Grenet. "In-Place Fast Polynomial Modular Remainder". *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. 2024, pp. 26–35. DOI: 10.1145/3666000.3669672 (pp. 17, 46, 47, 49, 59).

[C12] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. "Fast Interpolation and Multiplication of Unbalanced Polynomials". *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. 2024, pp. 437–446. DOI: 10.1145/3666000.3669717 (pp. 63, 82, 90, 91).

[C13] Jean-Guillaume Dumas, Alexis Galan, Bruno Grenet, Aude Maignan, and Daniel S. Roche. "Optimal Communication Unbalanced Private Set Union". *Applied Cryptography and Network Security*. 2025, pp. 107–135. DOI: 10.1007/978-3-031-95764-2_5 (p. 2).

Unpublished manuscripts

[M1] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. *Random Primes in Arithmetic Progressions*. 2022. arXiv: 2202.05955 (p. 63).

[M2] Jean-Guillaume Dumas and Bruno Grenet. *Fast In-Place Accumulation*. 2025. HAL: hal-05000159 (pp. 17, 43, 44, 46, 47, 49–51, 54, 55, 57, 59).

Software

[S1] Jean-Guillaume Dumas, Bruno Grenet, Clément Pernet, and Alexandre Sedoglavic. *PLinOpt, a Collection of C++ Routines Handling Linear & Bilinear Programs*. Version v3.1. 2024. URL: https://github.com/jgdumas/plinopt (pp. 51, 53).

References

[1] Karl Abrahamson. "Time-Space Tradeoffs for Branching Programs Contrasted with Those for Straight-Line Programs". *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)*. 1986, pp. 402–409. DOI: 10.1109/SFCS.1986.58 (pp. 21, 57).

[2] Leonard M. Adleman and Jonathan DeMarrais. "A Subexponential Algorithm for Discrete Logarithms over All Finite Fields". *Mathematics of Computation* 61.203 (1993), pp. 1–15. DOI: 10.1090/S0025-5718-1993-1225541-3 (p. 67).

[3] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES Is in P". *Annals of Mathematics* 160.2 (2004), pp. 781–793. DOI: 10.4007/annals.2004.160.781 (p. 78).

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Pub. Co, 1974. 470 pp. ISBN: 978-0-201-00029-0 (p. 19).

[5] Noga Alon and Yishay Mansour. "ε-Discrepancy Sets and Their Application for Interpolation of Sparse Polynomials". *Information Processing Letters* 54.6 (1995), pp. 337–342. DOI: 10.1016/0020-0190(95)00032-8 (pp. 63, 70).

[6] Amihood Amir, Oren Kapah, and Ely Porat. "Deterministic Length Reduction: Fast Convolution in Sparse Data and Applications". *Combinatorial Pattern Matching*. 2007, pp. 183–194. DOI: 10.1007/978-3-540-73437-6_20 (p. 90).

[7] Andrew Arnold. "A New Truncated Fourier Transform Algorithm". *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*. 2013, pp. 15–22. DOI: 10.1145/2465506.2465957 (p. 43).

[8] Andrew Arnold. "Sparse Polynomial Interpolation and Testing". University of Waterloo, 2016. URL: https://uwspace.uwaterloo.ca/bitstream/handle/10012/10307/Arnold_Andrew.pdf (p. 73).

[9] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. "Faster Sparse Interpolation of Straight-Line Programs". *Computer Algebra in Scientific Computing*. 2013, pp. 61–74. DOI: 10.1007/978-3-319-02297-0_5 (pp. 63, 69).

[10] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. "Sparse Interpolation over Finite Fields via Low-order Roots of Unity". *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. 2014, pp. 27–34. DOI: 10.1145/2608628.2608671 (p. 63).

[11] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. "Faster Sparse Multivariate Polynomial Interpolation of Straight-Line Programs". *Journal of Symbolic Computation* 75 (2016), pp. 4–24. DOI: 10.1016/j.jsc.2015.11.005 (pp. 63, 73).

[12] Andrew Arnold and Daniel S. Roche. "Multivariate Sparse Interpolation Using Randomized Kronecker Substitutions". *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation - ISSAC '14*. 2014, pp. 35–42. DOI: 10.1145/2608628.2608674 (pp. 63, 73).

[13] Andrew Arnold and Daniel S. Roche. "Output-Sensitive Algorithms for Sumset and Sparse Polynomial Multiplication". *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*. 2015, pp. 29–36. DOI: 10.1145/2755996.2756653 (pp. 63, 69, 90).

[14] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st ed. Cambridge University Press, 2009. ISBN: 978-0-521-42426-4 (pp. 19, 21, 22).

[15] Walter Baur and Volker Strassen. "The Complexity of Partial Derivatives". *Theoretical Computer Science* 22.3 (1983), pp. 317–330. DOI: 10.1016/0304-3975(83)90110-X (p. 69).

[16] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic Differentiation in Machine Learning: A Survey". *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. URL: http://www.jmlr.org/papers/v18/17-468.html (p. 69).

[17] Michael Ben-Or and Prasoon Tiwari. "A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation". *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. 1988, pp. 301–309. DOI: 10.1145/62212.62241 (pp. 63, 66).

[18] Charles H. Bennett. "Logical Reversibility of Computation". *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. URL: https://ieeexplore.ieee.org/abstract/document/5391327/ (p. 60).

[19] Elwyn R. Berlekamp. "Nonbinary BCH Decoding". *IEEE Transactions on Information Theory* 14.2 (1968), pp. 242–242. DOI: 10.1109/TIT.1968.1054109 (p. 8).

[20] Elwyn R. Berlekamp. "Factoring Polynomials over Large Finite Fields". *Mathematics of computation* 24.111 (1970), pp. 713–735. DOI: 10.1090/S0025-5718-1970-0276200-X (pp. 65, 66, 74).

[21] Elwyn R. Berlekamp. *Algebraic Coding Theory*. Rev. ed. World scientific, 2015. ISBN: 978-981-4635-89-9 (p. 8).

[22] Somnath Bhattacharjee, Mrinal Kumar, Shanthanu Rai, Varun Ramanathan, Ramprasad Saptharishi, and Shubhangi Saraf. *Closure under Factorization from a Result of Furstenberg*. 2025. Electronic Colloquium on Computational Complexity: TR25-084 (pp. 94, 99).

[23] Jingguo Bi, Qi Cheng, and J. Maurice Rojas. "Sublinear Root Detection and New Hardness Results for Sparse Polynomials over Finite Fields". *SIAM Journal on Computing* 45.4 (2016), pp. 1433–1447. DOI: 10.1137/140990401 (pp. 94, 98).

[24] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. "Batch PIR and Labeled PSI with Oblivious Ciphertext Compression". *33rd USENIX Security Symposium*. 2024, pp. 5949–5966. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bienstock (p. 100).

[25] Richard E. Blahut. "Algebraic Codes in the Frequency Domain". *Algebraic Coding Theory and Applications*. 1979, pp. 447–494. DOI: 10.1007/978-3-662-39641-4_12 (p. 66).

[26] Markus Bläser, Moritz Hardt, Richard J. Lipton, and Nisheeth K. Vishnoi. "Deterministically Testing Sparse Polynomial Identities of Unbounded Degree". *Information Processing Letters* 109.3 (2009), pp. 187–192. DOI: 10.1016/j.ipl.2008.09.029 (p. 79).

[27] Markus Bläser and Gorav Jindal. "A New Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation". *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. 2014, pp. 51–58. DOI: 10.1145/2608628.2608648 (p. 63).

[28] Leo Bluestein. “A Linear Filtering Approach to the Computation of Discrete Fourier Transform”. *IEEE Transactions on Audio and Electroacoustics* 18.4 (1970), pp. 451–455. DOI: 10.1109/TAU.1970.1162132 (p. 6).

[29] Allan Borodin and Robert Moenck. “Fast Modular Transforms”. *Journal of Computer and System Sciences* 8.3 (1974), pp. 366–386. DOI: 10.1016/S0022-0000(74)80029-2 (p. 6).

[30] Allan Borodin and Prasoon Tiwari. “On the Decidability of Sparse Univariate Polynomial Interpolation”. *Computational Complexity* 1.1 (1991), pp. 67–90. DOI: 10.1007/BF01200058 (pp. 63, 98).

[31] Alin Bostan, Grégoire Lecerf, Bruno Salvy, Éric Schost, and B. Wiebelt. “Complexity Issues in Bivariate Polynomial Factorization”. *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation - ISSAC ’04*. 2004, pp. 42–49. DOI: 10.1145/1005285.1005294 (p. 94).

[32] Alin Bostan, Grégoire Lecerf, and Éric Schost. “Tellegen’s Principle into Practice”. *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*. 2003, pp. 37–44. DOI: 10.1145/860854.860870 (pp. 7, 12, 13, 38, 58).

[33] Alin Bostan and Éric Schost. “Polynomial Evaluation and Interpolation on Special Sets of Points”. *Journal of Complexity* 21.4 (2005), pp. 420–446. DOI: 10.1016/j.jco.2004.09.009 (pp. 6, 7, 13).

[34] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. “Memory Efficient Scheduling of Strassen-Winograd’s Matrix Multiplication Algorithm”. *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*. 2009, pp. 55–62. DOI: 10.1145/1576702.1576713 (p. 53).

[35] Richard P. Brent, Fred G. Gustavson, and David YY Yun. “Fast Solution of Toeplitz Systems of Equations and Computation of Padé Approximants”. *Journal of Algorithms* 1.3 (1980), pp. 259–295. DOI: 10.1016/0196-6774(80)90013-9 (p. 7).

[36] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010. ISBN: 0-521-19469-5 (p. 4).

[37] Karl Bringmann, Nick Fischer, and Vasileios Nakos. “Sparse Nonnegative Convolution Is Equivalent to Dense Nonnegative Convolution”. *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 2021, pp. 1711–1724. DOI: 10.1145/3406325.3451090 (p. 90).

[38] Karl Bringmann, Nick Fischer, and Vasileios Nakos. “Deterministic and Las Vegas Algorithms for Sparse Nonnegative Convolution”. *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2022, pp. 3069–3090. DOI: 10.1137/1.9781611977073.119 (p. 90).

[39] Irénée Briquel and Peter Bürgisser. “The Real Tau-Conjecture Is True on Average”. *Random Structures & Algorithms* 57.2 (2020), pp. 279–303. DOI: 10.1002/rsa.20926 (p. 99).

[40] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. “Computing with a Full Memory: Catalytic Space”. *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*. 2014, pp. 857–866. DOI: 10.1145/2591796.2591874 (pp. 24, 58).

[41] Peter Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*. Springer, 2000. ISBN: 978-3-540-66752-0 (p. 20).

[42] David G. Cantor and Daniel M. Gordon. "Factoring Polynomials over p-Adic Fields". *Algorithmic Number Theory*. 2000, pp. 185–208. DOI: 10.1007/10722028_10 (p. 94).

[43] David G. Cantor and Erich L. Kaltofen. "On Fast Multiplication of Polynomials over Arbitrary Algebras". *Acta Informatica* 28.7 (1991), pp. 693–701. DOI: 10.1007/BF01178683 (p. 4).

[44] Baptiste Chanus, Damiano Mazza, and Morgan Rogers. "Unifying Boolean and Algebraic Descriptive Complexity". *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*. Vol. 337. 2025, 13:1–13:22. DOI: 10.4230/LIPIcs.FSCD.2025.13 (p. 20).

[45] Qi Cheng, Shuhong Gao, J. Maurice Rojas, and Daqing Wan. "Sparse Univariate Polynomials with Many Roots over Finite Fields". *Finite Fields and Their Applications* 46 (2017), pp. 235–246. DOI: 10.1016/j.ffa.2017.03.006 (p. 98).

[46] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher. *Reducing the Number of Qubits in Quantum Factoring*. 2024. IACR: 2024/222 (p. 60).

[47] Guillaume Chèze and André Galligo. "Four Lectures on Polynomial Absolute Factorization". *Solving Polynomial Equations: Foundations, Algorithms, and Applications*. 2005, pp. 339–392. DOI: 10.1007/3-540-27357-3_9 (p. 94).

[48] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. "Private Information Retrieval". *Journal of the ACM* 45.6 (1998), pp. 965–981. DOI: 10.1145/293347.293350 (p. 100).

[49] Michael Clausen, Andreas Dress, Johannes Grabmeier, and Marek Karpinski. "On Zero-Testing and Interpolation of k-Sparse Multivariate Polynomials over Finite Fields". *Theoretical Computer Science* 84.2 (1991), pp. 151–164. DOI: 10.1016/0304-3975(91)90157-W (p. 63).

[50] Richard Cole and Ramesh Hariharan. "Verifying Candidate Matches in Sparse and Wildcard Matching". *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*. 2002, pp. 592–601. DOI: 10.1145/509907.509992 (p. 90).

[51] Matthew T. Comer, Erich L. Kaltofen, and Clément Pernet. "Sparse Polynomial Interpolation and Berlekamp/Massey Algorithms That Correct Outlier Errors in Input Values". *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*. 2012, pp. 138–145. DOI: 10.1145/2442829.2442852 (p. 99).

[52] James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. "The Structure of Catalytic Space: Capturing Randomness and Time via Compression". *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*. 2025, pp. 554–564. DOI: 10.1145/3717823.3718112 (p. 58).

[53] Stephen A. Cook. "On the Minimum Computation Time of Functions". PhD thesis. Harvard, 1966 (p. 4).

[54] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". *Mathematics of computation* 19.90 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1 (pp. 4, 43).

[55] Nicholas Coxon. "An In-Place Truncated Fourier Transform". *Journal of Symbolic Computation* 110 (2022), pp. 66–80. DOI: 10.1016/j.jsc.2021.10.002 (p. 43).

[56] Felipe Cucker, Pascal Koiran, and Steve Smale. "A Polynomial Time Algorithm for Diophantine Equations in One Variable". *Journal of Symbolic Computation* 27.1 (1999), pp. 21–29. DOI: 10.1006/jsco.1998.0242 (p. 94).

[57] René Descartes. *La Géométrie*. Jan Maire, 1637. URL: https://gallica.bnf.fr/ark:/12148/btv1b86069594/f379 (p. 98).

[58] Jean-Louis Dornstetter. "On the Equivalence between Berlekamp's and Euclid's Algorithms (Corresp.)" *IEEE Transactions on Information Theory* 33.3 (1987), pp. 428–431. DOI: 10.1109/TIT.1987.1057299 (p. 8).

[59] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. "Dense Linear Algebra over Word-Size Prime Fields: The FFLAS and FFPACK Packages". *ACM Transactions on Mathematical Software* 35.3 (2008), 19:1–19:42. DOI: 10.1145/1391989.1391992 (pp. 54, 59).

[60] Jean-Guillaume Dumas, Clément Pernet, and Ziad Sultan. "Simultaneous Computation of the Row and Column Rank Profiles". *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*. 2013, pp. 181–188. DOI: 10.1145/2465506.2465517 (p. 54).

[61] Graham Everest, Alf van der Poorten, Igor Shparlinski, and Thomas Ward. *Recurrence Sequences*. v. 104. American Mathematical Society, 2003. 318 pp. ISBN: 978-0-8218-3387-2 (p. 8).

[62] Charles M. Fiduccia. "On Obtaining Upper Bounds on the Complexity of Matrix Multiplication". *Complexity of Computer Computations*. 1972, pp. 31–40. DOI: 10.1007/978-1-4684-2001-2_4 (p. 12).

[63] Charles M. Fiduccia. "On the Algebraic Complexity of Matrix Multiplication". PhD thesis. Brown University, 1973 (p. 12).

[64] Nick Fischer. "Sumsets, 3SUM, Subset Sum: Now for Real!" *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2025, pp. 4520–4546. DOI: 10.1137/1.9781611978322.155 (pp. 91, 92).

[65] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. "How to Compress Encrypted Data". *Advances in Cryptology – EUROCRYPT 2023*. Vol. 14004. 2023, pp. 551–577. DOI: 10.1007/978-3-031-30545-0_19 (pp. 99, 100).

[66] Michael L. Fredman and Dan E. Willard. "Surpassing the Information Theoretic Bound with Fusion Trees". *Journal of Computer and System Sciences* 47.3 (1993), pp. 424–436. DOI: 10.1016/0022-0000(93)90040-4 (p. 21).

[67] Rūsiņš Freivalds. "Fast Probabilistic Algorithms". *Mathematical Foundations of Computer Science 1979*. 1979, pp. 57–69. DOI: 10.1007/3-540-09526-8_5 (p. 88).

[68] Daniel P. Friedman and David S. Wise. *Unwinding Structured Recursions into Iterations*. TR19. Indiania University, 1975. URL: https://legacy.cs.indiana.edu/ftp/techreports/TR19.pdf (p. 23).

[69] Albrecht Fröhlich and John Cedric Shepherdson. "On the Factorisation of Polynomials in a Finite Number of Steps". *Mathematische Zeitschrift* 62.1 (1955), pp. 331–334. DOI: 10.1007/BF01180640 (p. 65).

[70] Albrecht Fröhlich and John Cedric Shepherdson. "Effective Procedures in Field Theory". *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 248.950 (1956), pp. 407–432. DOI: 10.1098/rsta.1956.0003 (p. 65).

[71] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012. 615 pp. ISBN: 978-1-107-01392-6 (p. 67).

[72] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. 3rd. Cambridge University Press, 2013. ISBN: 978-1-107-03903-2 (pp. 3, 31).

[73] Joachim von zur Gathen and Erich L. Kaltofen. "Factoring Sparse Multivariate Polynomials". *Journal of Computer and System Sciences* 31.2 (1985), pp. 265–287. DOI: 10.1016/0022-0000(85)90044-3 (p. 99).

[74] Joachim von zur Gathen and Victor Shoup. "Computing Frobenius Maps and Factoring Polynomials". *Computational Complexity* 2.3 (1992), pp. 187–224. DOI: 10.1007/BF01272074 (p. 37).

[75] Joachim von zur Gathen and Victor Shoup. "Computing Frobenius Maps and Factoring Polynomials". *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. 1992, pp. 97–105. DOI: 10.1145/129712.129722 (p. 37).

[76] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic, 1992. 585 pp. ISBN: 978-0-7923-9259-0 (p. 3).

[77] W. Morven Gentleman and G. Sande. "Fast Fourier Transforms: For Fun and Profit". *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*. 1966, pp. 563–578. DOI: 10.1145/1464291.1464352 (p. 4).

[78] Craig Gidney. *Asymptotically Efficient Quantum Karatsuba Multiplication*. 2019. arXiv: 1904.07356 (p. 60).

[79] Mark Giesbrecht, Qiao-Long Huang, and Éric Schost. "Sparse Multiplication for Skew Polynomials". *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*. 2020, pp. 194–201. DOI: 10.1145/3373207.3404023 (p. 63).

[80] Mark Giesbrecht, George Labahn, and Wen-shin Lee. "Symbolic-Numeric Sparse Interpolation of Multivariate Polynomials". *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*. 2006, pp. 116–123. DOI: 10.1145/1145768.1145792 (p. 63).

[81] Mark Giesbrecht, George Labahn, and Wen-shin Lee. "Symbolic–Numeric Sparse Interpolation of Multivariate Polynomials". *Journal of Symbolic Computation* 44.8 (2009), pp. 943–959. DOI: 10.1016/j.jsc.2008.11.003 (p. 63).

[82] Pascal Giorgi. "A Probabilistic Algorithm for Verifying Polynomial Middle Product in Linear Time". *Information Processing Letters* 139 (2018), pp. 30–34. DOI: 10.1016/j.ipl.2018.06.014 (p. 86).

[83] Marc Giusti, Joos Heintz, Jose Enrique Morais, Jacques Morgenstern, and Luis Miguel Pardo. "Straight-Line Programs in Geometric Elimination Theory". *Journal of Pure and Applied Algebra* 124.1 (1998), pp. 101–146. DOI: 10.1016/S0022-4049(96)00099-0 (p. 11).

[84] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. ISBN: 978-0-521-88473-0. DOI: 10.1017/CBO9780511804106 (p. 24).

[85] Dima Grigoriev and Marek Karpinski. "A Zero-Test and an Interpolation Algorithm for the Shifted Sparse Polynomials". *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. 1993, pp. 162–169. DOI: 10.1007/3-540-56686-4_41 (p. 63).

[86] Dima Yu. Grigoriev, Marek Karpinski, and Michael F. Singer. "Fast Parallel Algorithms for Sparse Multivariate Polynomial Interpolation over Finite Fields". *SIAM Journal on Computing* 19.6 (1990), pp. 1059–1063. DOI: 10.1137/0219073 (p. 63).

[87] The Fflas-Ffpack Group. *FFLAS-FFPACK*. Version 2.4.2. https://github.com/linbox-team/fflas-ffpack, 2019. URL: https://hal.archives-ouvertes.fr/hal-02128878 (p. 59).

[88] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. "The Middle Product Algorithm I". *Applicable Algebra in Engineering, Communication and Computing* 14.6 (2004), pp. 415–438. DOI: 10.1007/s00200-003-0144-2 (pp. 5, 7).

[89] David Harvey and Joris van der Hoeven. "Faster Polynomial Multiplication over Finite Fields Using Cyclotomic Coefficient Rings". *Journal of Complexity* 54 (2019), p. 101404. DOI: 10.1016/j.jco.2019.03.004 (p. 5).

[90] David Harvey and Joris van der Hoeven. "Integer Multiplication in Time $O(n \log n)$". *Annals of Mathematics* 193.2 (2021). DOI: 10.4007/annals.2021.193.2.4 (p. 4).

[91] David Harvey and Joris van der Hoeven. "Polynomial Multiplication over Finite Fields in Time $O(n \log n)$". *Journal of the ACM* 69.2 (2022), 12:1–12:40. DOI: 10.1145/3505584 (pp. 5, 7).

[92] David Harvey and Daniel S. Roche. "An In-Place Truncated Fourier Transform and Applications to Polynomial Multiplication". *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. 2010, pp. 325–329. DOI: 10.1145/1837934.1837996 (pp. 25, 43).

[93] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. "Simple and Practical Algorithm for Sparse Fourier Transform". *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2012, pp. 1183–1194. DOI: 10.1137/1.9781611973099.93 (p. 70).

[94] Haitham Al Hassanieh. "The Sparse Fourier Transform: Theory & Practice". PhD thesis. Massachusetts Institute of Technology, 2016. HDL: 1721.1/103715 (p. 70).

[95] Kurt Hensel. "Eine neue Theorie der algebraischen Zahlen". *Mathematische Zeitschrift* 2.3 (1918), pp. 433–452. DOI: 10.1007/BF01199422 (p. 78).

[96] Joris van der Hoeven. "The Truncated Fourier Transform and Applications". *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*. 2004, pp. 290–296. DOI: 10.1145/1005285.1005327 (p. 43).

[97] Joris van der Hoeven. "Optimizing the Half-Gcd Algorithm". *Applicable Algebra in Engineering, Communication and Computing* (2025). DOI: 10.1007/s00200-025-00690-w (p. 7).

[98] Joris van der Hoeven and Grégoire Lecerf. "On the Bit-complexity of Sparse Polynomial and Series Multiplication". *Journal of Symbolic Computation* 50 (2013), pp. 227–254. DOI: 10.1016/j.jsc.2012.06.004 (p. 63).

[99] Joris van der Hoeven and Grégoire Lecerf. "Sparse Polynomial Interpolation in Practice". *ACM Communications in Computer Algebra* 48.3/4 (2014), pp. 187–191. DOI: 10.1145/2733693.2733721 (pp. 63, 74).

[100] Joris van der Hoeven and Grégoire Lecerf. "Sparse Polynomial Interpolation: Faster Strategies over Finite Fields". *Applicable Algebra in Engineering, Communication and Computing* (2024). DOI: 10.1007/s00200-024-00655-5 (p. 63).

[101] Joris van der Hoeven and Grégoire Lecerf. "Fast Interpolation of Multivariate Polynomials with Sparse Exponents". *Journal of Complexity* 87 (2025), p. 101922. DOI: 10.1016/j.jco.2024.101922 (pp. 63, 80, 81, 91).

[102] Joris van der Hoeven and Michael Monagan. "Computing One Billion Roots Using the Tangent Graeffe Method". *ACM Communications in Computer Algebra* 54.3 (2020), pp. 65–85. DOI: 10.1145/3457341.3457342 (p. 76).

[103] Joris van der Hoeven and Michael Monagan. "Implementing the Tangent Graeffe Root Finding Method". *Mathematical Software – ICMS 2020*. 2020, pp. 482–492. DOI: 10.1007/978-3-030-52200-1_48 (p. 76).

[104] Iggy van Hoof. "Space-Efficient Quantum Multiplication Polynomials for Binary Finite Fields with Sub-Quadratoc Toffoli Gate Count". *Quantum Information and Computation* 20 (9&10 2020), pp. 721–735. DOI: 10.26421/QIC20.9-10-1 (p. 60).

[105] Jiaxiong Hu and Michael Monagan. "A Fast Parallel Sparse Polynomial GCD Algorithm". *Journal of Symbolic Computation* 105 (2021), pp. 28–63. DOI: 10.1016/j.jsc.2020.06.001 (p. 63).

[106] Ming-Deh A Huang and Ashwin J Rao. "Interpolation of Sparse Multivariate Polynomials over Large Finite Fields with Applications". *Journal of Algorithms* 33.2 (1999), pp. 204–228. DOI: 10.1006/jagm.1999.1045 (p. 63).

[107] Qiao-Long Huang. "Sparse Polynomial Interpolation over Fields with Large or Zero Characteristic". *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation - ISSAC '19*. 2019, pp. 219–226. DOI: 10.1145/3326229.3326250 (pp. 63, 69, 70, 73, 74).

[108] Qiao-Long Huang. "Sparse Polynomial Interpolation Based on Diversification". *Science China Mathematics* (2021). DOI: 10.1007/s11425-020-1791-5 (pp. 63, 73, 74).

[109] Qiao-Long Huang. "Sparse Polynomial Interpolation Based on Derivatives". *Journal of Symbolic Computation* 114 (2023), pp. 359–375. DOI: 10.1016/j.jsc.2022.06.002 (pp. 63, 70, 73).

[110] Qiao-Long Huang and Xiao-Shan Gao. "Revisit Sparse Polynomial Interpolation Based on Randomized Kronecker Substitution". *International Workshop on Computer Algebra in Scientific Computing*. 2019, pp. 215–235. DOI: 10.1007/978-3-030-26831-2_15 (pp. 63, 73).

[111] Qiao-Long Huang and Xiao-Shan Gao. "Faster Interpolation Algorithms for Sparse Multivariate Polynomials given by Straight-Line Programs". *Journal of Symbolic Computation* 101 (2020), pp. 367–386. DOI: 10.1016/j.jsc.2019.10.005 (p. 63).

[112] Qiao-Long Huang and Xiao-Shan Gao. "New Sparse Multivariate Polynomial Factorization Algorithms over Integers". *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation*. 2023, pp. 315–324. DOI: 10.1145/3597066.3597087 (p. 63).

[113] Qiao-Long Huang and Xiao-Shan Gao. "Bit Complexity of Polynomial GCD on Sparse Representation". *Mathematics of Computation* (2025). DOI: 10.1090/mcom/4054 (p. 63).

[114] Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. "Implementation of Strassen's Algorithm for Matrix Multiplication". *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 1996, 32–es. DOI: 10.1145/369028.369096 (p. 53).

[115] Erdal Imamoglu, Erich L. Kaltofen, and Zhengfeng Yang. "Sparse Polynomial Interpolation with Arbitrary Orthogonal Polynomial Bases". *Proc. ISSAC*. Vol. 18. 2018. DOI: 10.1145/3208976.3208999 (p. 63).

[116] Seyed Mohammad Mahdi Javadi and Michael Monagan. "Parallel Sparse Polynomial Interpolation over Finite Fields". *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. 2010, pp. 160–168. DOI: 10.1145/1837210.1837233 (p. 63).

[117] Ce Jin and Yinzhan Xu. "Shaving Logs via Large Sieve Inequality: Faster Algorithms for Sparse Convolution and More". *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 2024, pp. 1573–1584. DOI: 10.1145/3618260.3649605 (pp. 90–92).

[118] Stephen C. Johnson. "Sparse Polynomial Arithmetic". *ACM SIGSAM Bulletin* 8.3 (1974), pp. 63–71. DOI: 10.1145/1086837.1086847 (pp. 63, 90).

[119] Gregory D. Kahanamoku-Meyer and Norman Y. Yao. *Fast Quantum Integer Multiplication with Zero Ancillas*. 2024. arXiv: 2403.18006 (p. 60).

[120] Thomas Kailath, Sun-Yuan Kung, and Martin Morf. "Displacement Ranks of Matrices and Linear Equations". *Journal of Mathematical Analysis and Applications* 68.2 (1979), pp. 395–407. DOI: 10.1016/0022-247X(79)90124-0 (p. 9).

[121] Erich L. Kaltofen. "Greatest Common Divisors of Polynomials given by Straight-Line Programs". *Journal of the ACM* 35.1 (1988), pp. 231–264. DOI: 10.1145/42267.45069 (p. 74).

[122] Erich L. Kaltofen. "Challenges of Symbolic Computation: My Favorite Open Problems". *Journal of Symbolic Computation* 29.6 (2000), pp. 891–919. DOI: 10.1006/jsco.2000.0370 (pp. 13, 58).

[123] Erich L. Kaltofen. "Polynomial Factorization: A Success Story". *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*. 2003, pp. 3–4. DOI: 10.1145/860854.860857 (p. 11).

[124] Erich L. Kaltofen and Pascal Koiran. "On the Complexity of Factoring Bivariate Supersparse (Lacunary) Polynomials". *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*. 2005, pp. 208–215. DOI: 10.1145/1073884.1073914 (pp. 63, 94).

[125] Erich L. Kaltofen and Pascal Koiran. "Finding Small Degree Factors of Multivariate Supersparse (Lacunary) Polynomials over Algebraic Number Fields". *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*. 2006, pp. 162–168. DOI: 10.1145/1145768.1145798 (pp. 63, 94).

[126] Erich L. Kaltofen and Yagati N. Lakshman. "Improved Sparse Multivariate Polynomial Interpolation Algorithms". *International Symposium on Symbolic and Algebraic Computation*. 1988, pp. 467–474. DOI: 10.1007/3-540-51084-2_44 (p. 63).

[127] Erich L. Kaltofen, Yagati N. Lakshman, and John-Michael Wiley. "Modular Rational Sparse Multivariate Polynomial Interpolation". *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. 1990, pp. 135–139. DOI: 10.1145/96877.96912 (p. 63).

[128] Erich L. Kaltofen and Wen-shin Lee. "Early Termination in Sparse Interpolation Algorithms". *Journal of Symbolic Computation* 36.3 (2003), pp. 365–400. DOI: 10.1016/S0747-7171(03)00088-9 (pp. 63, 67).

[129] Erich L. Kaltofen, Wen-shin Lee, and Austin A. Lobo. "Early Termination in Ben-Or/Tiwari Sparse Interpolation and a Hybrid of Zippel's Algorithm". *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation*. 2000, pp. 192–201. DOI: 10.1145/345542.345629 (pp. 63, 67).

[130] Erich L. Kaltofen, John P. May, Zhengfeng Yang, and Lihong Zhi. "Approximate Factorization of Multivariate Polynomials Using Singular Value Decomposition". *Journal of Symbolic Computation* 43.5 (2008), pp. 359–376. DOI: 10.1016/j.jsc.2007.11.005 (p. 94).

[131] Erich L. Kaltofen and Clément Pernet. "Sparse Polynomial Interpolation Codes and Their Decoding Beyond Half the Minimum Distance". *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. 2014, pp. 272–279. DOI: 10.1145/2608628.2608660 (p. 98).

[132] Erich L. Kaltofen and Zhi-Hong Yang. "Sparse Interpolation With Errors in Chebyshev Basis Beyond Redundant-Block Decoding". *IEEE Transactions on Information Theory* 67.1 (2021), pp. 232–243. DOI: 10.1109/TIT.2020.3027036 (p. 63).

[133] Michael Kaminski. "A Note on Probabilistically Verifying Integer and Polynomial Products". *Journal of the ACM* 36.1 (1989), pp. 142–149. DOI: 10.1145/58562.214082 (pp. 85, 89).

[134] Michael Kaminski, David G. Kirkpatrick, and Nader H. Bshouty. "Addition Requirements for Matrix and Transposed Matrix Products". *Journal of Algorithms* 9.3 (1988), pp. 354–364. DOI: 10.1016/0196-6774(88)90026-0 (p. 12).

[135] Anatolii A. Karatsuba and Yu P. Ofman. "Multiplication of Multidigit Numbers on Automata". *Doklady physics* 7.7 (1963), pp. 595–596 (pp. 4, 17, 42).

[136] Alan H. Karp and Peter Markstein. "High-Precision Division and Square Root". *ACM Transactions on Mathematical Software* 23.4 (1997), pp. 561–589. DOI: 10.1145/279232.279237 (p. 32).

[137] Kiran S. Kedlaya. "The Algebraic Closure of the Power Series Field in Positive Characteristic". *Proceedings of the American Mathematical Society* 129.12 (2001), pp. 3461–3470. DOI: 10.1090/S0002-9939-01-06001-4 (p. 95).

[138] Shane Kepley and Rainer Steinwandt. "Quantum Circuits for $\mathbb{F}_{2^n}$-Multiplication with Subquadratic Gate Count". *Quantum Information Processing* 14.7 (2015), pp. 2373–2386. DOI: 10.1007/s11128-015-0993-1 (p. 60).

[139] Mohamed Khochtali, Daniel S. Roche, and Xisen Tian. "Parallel Sparse Interpolation Using Small Primes". *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*. 2015, pp. 70–77. DOI: 10.1145/2790282.2790290 (p. 63).

[140] Askold G. Khovanskii. *Fewnomials*. v. 88. American Mathematical Society, 1991. 139 pp. ISBN: 978-0-8218-4547-9 (p. 98).

[141] Donald Ervin Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. 3rd. Addison-Wesley Professional, 1997. ISBN: 978-0-201-89684-8 (p. 1).

[142] Pascal Koiran. "Shallow Circuits with High-Powered Inputs". *Innovations in Computer Science - ICS 2011, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*. 2011, pp. 309–320. URL: http://conference.iiis.tsinghua.edu.cn/ICS2011/content/papers/5.html (p. 99).

[143] Pascal Koiran, Natacha Portier, Sébastien Tavenas, and Stéphan Thomassé. "A τ-Conjecture for Newton Polygons". *Foundations of Computational Mathematics* 15.1 (2015), pp. 185–197. DOI: 10.1007/s10208-014-9216-x (p. 99).

[144] Leopold Kronecker. "Grundzüge einer arithmetischen Theorie der algebraischen Grössen. (Abdruck einer Festschrift zu Herrn E. E. Kummers Doctor-Jubiläum, 10. September 1881.)." *Journal für die reine und angewandte Mathematik* 92 (1882), pp. 1–122. URL: https://eudml.org/doc/148487 (p. 73).

[145] Fabian Kuhn and René Struik. "Random Walks Revisited: Extensions of Pollard's Rho Algorithm for Computing Multiple Discrete Logarithms". *Selected Areas in Cryptography*. 2001, pp. 212–229. DOI: 10.1007/3-540-45537-X_17 (p. 67).

[146] Hsiang-Tsung Kung. "On Computing Reciprocals of Power Series". *Numerische Mathematik* 22.5 (1974), pp. 341–348. DOI: 10.1007/BF01436917 (p. 6).

[147] Yagati N. Lakshman and B. David Saunders. "Sparse Polynomial Interpolation in Nonstandard Bases". *SIAM Journal on Computing* 24.2 (1995), pp. 387–397. DOI: 10.1137/S0097539792237784 (p. 63).

[148] Susan Landau. "Factoring Polynomials over Algebraic Number Fields". *SIAM Journal on Computing* 14.1 (1985), pp. 184–195. DOI: 10.1137/0214015 (p. 94).

[149] Yves Lecerf. "Machines de Turing Réversibles". *Comptes rendus hebdomadaires des séances de l'académie des sciences* 257.2 (1963), pp. 2597–2600. URL: https://gallica.bnf.fr/ark:/12148/bpt6k40088/f965 (p. 60).

[150] Daan Leijen and Anton Lorenzen. "Tail Recursion Modulo Context: An Equational Approach". *Proceedings of the ACM on Programming Languages* 7 (POPL 2023), 40:1152–40:1181. DOI: 10.1145/3571233 (p. 23).

[151] Arjen K. Lenstra. "Factoring Polynomials over Algebraic Number Fields". *Computer Algebra*. 1983, pp. 245–254. DOI: 10.1007/3-540-12868-9_108 (p. 94).

[152] Arjen K. Lenstra. "Factoring Multivariate Polynomials over Algebraic Number Fields". *SIAM Journal on Computing* 16.3 (1987). DOI: 10.1137/0216040 (p. 94).

[153] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. "Factoring Polynomials with Rational Coefficients". *Mathematische Annalen* 261.4 (1982), pp. 515–534. DOI: 10.1007/BF01457454 (p. 94).

[154] Hendrik W. Lenstra. "Finding Small Degree Factors of Lacunary Polynomials". *Number Theory in Progress*. 1999. DOI: 10.1515/9783110285581.267 (p. 94).

[155] Hendrik W. Lenstra. "On the Factorization of Lacunary Polynomials". *Number Theory in Progress*. 1999. DOI: 10.1515/9783110285581.277 (p. 98).

[156] Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. "Deterministic Time-Space Trade-Offs for k-SUM". *43rd International Colloquium on Automata, Languages, and Programming*. Vol. 55. 2016, 58:1–58:14. DOI: 10.4230/LIPIcs.ICALP.2016.58 (p. 23).

[157] Roman E. Maeder. "Storage Allocation for the Karatsuba Integer Multiplication Algorithm". *Design and Implementation of Symbolic Computation Systems*. Vol. 722. 1993, pp. 59–65. DOI: 10.1007/BFb0013168 (p. 25).

[158] Yishay Mansour. "Randomized Interpolation and Approximation of Sparse Polynomials". *SIAM Journal on Computing* 24.2 (1995), pp. 357–368. DOI: 10.1137/S0097539792239291 (pp. 63, 70).

[159] James L. Massey. "Shift-Register Synthesis and BCH Decoding". *IEEE Transactions on Information Theory* 15.1 (1969), pp. 122–127. DOI: 10.1109/TIT.1969.1054260 (p. 8).

[160] Michael Monagan. "In-Place Arithmetic for Polynomials over Zn". *Design and Implementation of Symbolic Computation Systems*. Vol. 721. 1993, pp. 22–34. DOI: 10.1007/3-540-57272-4_21 (p. 17).

[161] Michael Monagan and Qiao-Long Huang. "A New Sparse Polynomial GCD by Separating Terms". *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. 2024, pp. 134–142. DOI: 10.1145/3666000.3669684 (p. 63).

[162] Michael Monagan and Roman Pearce. "Parallel Sparse Polynomial Multiplication Using Heaps". *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation - ISSAC '09*. 2009, pp. 263–270. DOI: 10.1145/1576702.1576739 (pp. 63, 90).

[163] Michael Monagan and Roman Pearce. "Sparse Polynomial Division Using a Heap". *Journal of Symbolic Computation* 46.7 (2011), pp. 807–822. DOI: 10.1016/j.jsc.2010.08.014 (p. 63).

[164] Guillaume Moroz. "New Data Structure for Univariate Polynomial Approximation and Applications to Root Isolation, Numerical Multipoint Evaluation, and Other Problems". 2022, pp. 1090–1099. DOI: 10.1109/FOCS52979.2021.00108 (p. 94).

[165] Hirokazu Murao and Tetsuro Fujise. "Modular Algorithm for Sparse Multivariate Polynomial Interpolation and Its Parallel Implementation". *Journal of Symbolic Computation* 21.4 (1996), pp. 377–396. DOI: 10.1006/jsco.1996.0020 (p. 63).

[166] Ido Nahshon and Amir Shpilka. "New Bounds on Quotient Polynomials with Applications to Exact Division and Divisibility Testing of Sparse Polynomials". *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. 2024, pp. 91–99. DOI: 10.1145/3666000.3669679 (pp. 63, 93).

[167] Ido Nahshon and Amir Shpilka. *A New Bound on Cofactors of Sparse Polynomials*. Version 4. 2025. arXiv: 2308.03885 (pp. 63, 93).

[168] Vasileios Nakos. "Nearly Optimal Sparse Polynomial Multiplication". *IEEE Transactions on Information Theory* 66.11 (2020), pp. 7231–7236. DOI: 10.1109/TIT.2020.2989385 (p. 63).

[169] Harald Niederreiter. "Knapsack-Type Cryptosystems and Algebraic Coding Theory". *Problems of Control and Information Theory* 15.2 (1986), pp. 157–166. URL: https://real-j.mtak.hu/7997/ (p. 99).

[170] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th anniversary ed. Cambridge University Press, 2010. 676 pp. ISBN: 978-1-107-00217-3 (p. 60).

[171] Henri J. Nussbaumer. "Fast Polynomial Transform Algorithms for Digital Convolution". *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28.2 (1980), pp. 205–215. DOI: 10.1109/TASSP.1980.1163372 (p. 4).

[172] Rafail Ostrovsky and William E. Skeith. "A Survey of Single-Database Private Information Retrieval: Techniques and Applications". *Public Key Cryptography – PKC 2007*. 2007, pp. 393–411. DOI: 10.1007/978-3-540-71677-8_26 (p. 100).

[173] Victor Y. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Birkhäuser/Springer, 2001. 278 pp. ISBN: 978-0-8176-4240-2 (pp. 3, 9).

[174] Victor Y. Pan. "Univariate Polynomials: Nearly Optimal Algorithms for Numerical Factorization and Root-finding". *Journal of Symbolic Computation* 33.5 (2002), pp. 701–733. DOI: 10.1006/jsco.2002.0531 (p. 94).

[175] Alex Parent, Martin Roetteler, and Michele Mosca. "Improved Reversible and Quantum Circuits for Karatsuba-based Integer Multiplication". *12th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2017)*. 2018 (p. 60).

[176] Armelle Perret du Cray. “Algorithmes pour les polynômes creux : interpolation, arithmétique, test d’identité”. PhD thesis. Université de Montpellier, 2023. 171 pp. (p. 73).

[177] David A. Plaisted. “Sparse Complex Polynomials and Polynomial Reducibility”. *Journal of Computer and System Sciences* 14.2 (1977), pp. 210–221. DOI: 10.1016/S0022-0000(77)80013-5 (p. 63).

[178] David A. Plaisted. “Some Polynomial and Integer Divisibility Problems Are NP-Hard”. *SIAM Journal on Computing* 7.4 (1978), pp. 458–464. DOI: 10.1137/0207036 (p. 63).

[179] David A. Plaisted. “New NP-hard and NP-complete Polynomial and Integer Divisibility Problems”. *Theoretical Computer Science* 31.1 (1984), pp. 125–138. DOI: 10.1016/0304-3975(84)90130-0 (p. 63).

[180] Stephen C. Pohlig and Martin E. Hellman. “An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance”. *IEEE Transactions on Information Theory* 24.1 (1978), pp. 106–110. DOI: 10.1109/TIT.1978.1055817 (p. 74).

[181] John M. Pollard. “Monte Carlo Methods for Index Computation ($mod\ p$)”. *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: 10.1090/S0025-5718-1978-0491431-9 (p. 67).

[182] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1990. ISBN: 3-540-96131-3 (p. 19).

[183] Gaspard Riche de Prony. “Essai expérimental et analytique : Sur les lois de la Dilatabilité de fluides élastique et sur celles de la Force expansive de la vapeur de l’eau et de la vapeur de l’alkool, à différentes températures”. *Journal de l’École Polytechnique* 1 (1795 (Floréal et Prairial III)), pp. 24–76. URL: https://gallica.bnf.fr/ark:/12148/bpt6k433661n/f32.item (p. 65).

[184] Michael O. Rabin. “Probabilistic Algorithms in Finite Fields”. *SIAM Journal on Computing* 9.2 (1980), pp. 273–280. DOI: 10.1137/0209024 (pp. 65, 66, 74, 78).

[185] Seyoon Ragavan and Vinod Vaikuntanathan. “Space-Efficient and Noise-Robust Quantum Factoring”. *Advances in Cryptology – CRYPTO 2024*. 2024, pp. 107–140. DOI: 10.1007/978-3-031-68391-6_4 (p. 60).

[186] Oded Regev. “An Efficient Quantum Factoring Algorithm”. *Journal of the ACM* 72.1 (2025), 10:1–10:13. DOI: 10.1145/3708471 (p. 60).

[187] Kui Ren and Cong Wang. *Searchable Encryption: From Concepts to Systems*. Springer International Publishing, 2023. ISBN: 978-3-031-21377-9. DOI: 10.1007/978-3-031-21377-9 (p. 100).

[188] John C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages”. *Proceedings of the ACM Annual Conference - Volume 2*. 1972, pp. 717–740. DOI: 10.1145/800194.805852 (p. 23).

[189] Daniel S. Roche. “Space-and Time-Efficient Polynomial Multiplication”. *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*. 2009, pp. 295–302. DOI: 10.1145/1576702.1576743 (pp. 25, 57).

[190] Daniel S. Roche. “Efficient Computation with Sparse and Dense Polynomials”. PhD thesis. 2011. URL: https://uwspace.uwaterloo.ca/handle/10012/5869 (pp. 25, 43).

[191] Daniel S. Roche. “What Can (and Can’t) We Do with Sparse Polynomials?” *Proc. ISSAC’18*. 2018, pp. 25–30. DOI: 10.1145/3208976.3209027 (p. 63).

[192] Ron M. Roth. *Introduction to Coding Theory*. Cambridge University Press, 2006. 566 pp. ISBN: 978-0-521-84504-5 (p. 100).

[193] Antonio Sanso and Giuseppe Vitto. *Attacking Poseidon via Graeffe-Based Root-Finding over NTT-Friendly Fields*. 2025. IACR: 2025/937 (p. 76).

[194] Thomas Seiller. "Mathematical Informatics". Habilitation thesis. Université Sorbonne Paris Nord, 2024. URL: https://theses.hal.science/tel-04616661 (p. 20).

[195] Daniel Shanks. "Class Number, a Theory of Factorization, and Genera". *Proc. Sympos. Pure Math.* Vol. 20. 1971, pp. 415–440 (p. 66).

[196] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172 (p. 60).

[197] Victor Shoup. "A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic". *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*. 1991, pp. 14–21. DOI: 10.1145/120694.120697 (pp. 8, 75).

[198] Victor Shoup. "Fast Construction of Irreducible Polynomials over Finite Fields". *Journal of Symbolic Computation* 17.5 (1994), pp. 371–391. DOI: 10.1006/jsco.1994.1025 (p. 12).

[199] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. ISBN: 978-0-521-51644-0 (p. 79).

[200] Victor Shoup. *NTL: A Library for Doing Number Theory*. Version 11.5.1. 2021. URL: https://libntl.org (p. 59).

[201] A. Shpilka and A. Yehudayoff. *Arithmetic Circuits: A Survey of Recent Results and Open Questions*. Now, 2010. ISBN: 978-1-60198-400-5 (p. 63).

[202] Malte Sieveking. "An Algorithm for Division of Powerseries". *Computing* 10.1–2 (1972), pp. 153–156. DOI: 10.1007/BF02242389 (p. 6).

[203] Volker Strassen. "Die Berechnungskomplexität von Elementarsymmetrischen Funktionen Und von Interpolationskoeffizienten". *Numerische Mathematik* 20 (1973), pp. 238–251. DOI: 10.1007/BF01436566 (p. 6).

[204] Chen Su and Haining Fan. "Impact of Intel's New Instruction Sets on Software Implementation of GF(2)[x] Multiplication". *Information Processing Letters* 112.12 (2012), pp. 497–502. DOI: 10.1016/j.ipl.2012.03.012 (p. 25).

[205] Emmanuel Thomé. "Karatsuba Multiplication with Temporary Space of Size ≤ n". 2002. URL: https://members.loria.fr/EThome/files/kara.pdf (p. 25).

[206] Andrei L. Toom. "The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers". *Soviet Mathematics Doklady* 3 (1963), pp. 714–716 (p. 4).

[207] Virginia Vassilevska Williams. "On Some Fine-Grained Questions in Algorithms and Complexity". *Proceedings of the International Congress of Mathematicians*. 2019, pp. 3447–3487. DOI: 10.1142/9789813272880_0188 (p. 23).

[208] Violetta Weger, Niklas Gassner, and Joachim Rosenthal. *A Survey on Code-Based Cryptography*. 2024. arXiv: 2201.07119 (p. 99).

[209] Kai Werther. "The Complexity of Sparse Polynomial Interpolation over Finite Fields". *Applicable Algebra in Engineering, Communication and Computing* 5.2 (1994), pp. 91–103. DOI: 10.1007/BF01438278 (p. 63).

[210] R. Ryan Williams. “Simulating Time with Square-Root Space”. *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*. 2025, pp. 13–23. DOI: 10.1145/3717823.3718225 (p. 24).

[211] Andrew Chi-Chih Yao. “On the Evaluation of Powers”. *SIAM Journal on Computing* 5.1 (1976), pp. 100–103. DOI: 10.1137/0205008 (p. 86).

[212] Ziyu Zhao and Jintai Ding. *Breaking Poseidon Challenges with Graeffe Transforms and Complexity Analysis by FFT Lower Bounds*. 2025. IACR: 2025/950 (p. 76).

[213] Richard Zippel. “Probabilistic Algorithms for Sparse Polynomials”. *Symbolic and Algebraic Computation*. Vol. 72. 1979, pp. 216–226. DOI: 10.1007/3-540-09519-5_73 (p. 63).

Index

Index of notations

$f_{[i]}$	coefficient of degree i of f • 18
$f_{[i,j[}$ or $[f]_i^j$	polynomial $\sum_{d=i}^{j-1} f_{[d]} x^{d-i}$ • 18
$f \leftarrow_{[i,j[}$	polynomial $\sum_{d=i}^{j-1} f_{[d]} x^{j-d-1}$ • 18
$\mathsf{LowProd}(f, g)$	lower product of f by g • 5
$\mathsf{MidProd}(f, g)$	middle product of f by g • 5
$\mathsf{UppProd}(f, g)$	upper product of f by g • 5

Sparse polynomials

$cx^e \in f$	the coefficient of degree e of f is c • 64
$f^{[p]}$	fold f mod $x^p - 1$ of a sparse polynomial f • 64
$\deg(f)$, $\#f$, $\mathsf{h}(f)$	degree, sparsity and height of f • 64
$\deg_2(f)$	second degree $\deg\left(f - x^{\deg(f)}\right)$ of a monic polynomial • 86
$\mathsf{bitsize}(f)$	bit size of the sparse representation of f • 64

Vectors

$\boldsymbol{u} \| \boldsymbol{v}$	concatenation of two vectors • 10
$\boldsymbol{v}_{[i]}$ or v_i	ith entry of $\boldsymbol{v}$, $0 \le i < n$ • 3
$\boldsymbol{v}_{[i,j[}$	vector $(v_i, \ldots, v_{j-1})$ • 3
$\boldsymbol{v}^{\leftarrow}$	reversed vector of $\boldsymbol{v}$, defined by $\boldsymbol{v}^{\leftarrow}_{[i]} = \boldsymbol{v}_{[n-1-i]}$, $0 \le i < n$ • 3
$\mathbf{0}_m$	all-zero vector of length m • 10

Matrices

$M_{[i,j]}$ or $M_{i,j}$	entry (i, j) of M, $0 \le i < m$, $0 \le j < n$ • 3
$M^{\leftarrow}$	reversed of the matrix M • 13
M^{T}	transposed of the matrix M • 13
$C_m(\boldsymbol{a})$, $C_m^{\lambda}(\boldsymbol{a})$	circulant and λ-circulant $m \times m$ matrices with first row $\boldsymbol{a}$ • 10
$T_{m,n}(\boldsymbol{a})$, $T_m(\boldsymbol{a})$	Toeplitz $m \times n$ and $m \times m$ matrices built on $\boldsymbol{a}$ • 10
$L_m(\boldsymbol{a})$, $U_m(\boldsymbol{a})$	lower and upper triangular Toeplitz matrices built on $\boldsymbol{a}$ • 10
$V_{m,n}(\boldsymbol{a})$, $V_m(\boldsymbol{a})$	Vandermonde $m \times n$ and $m \times m$ matrices with second column $\boldsymbol{a}$ • 11
$F_m(\alpha)$	DFT matrix $V_m(1, \alpha, \ldots, \alpha^{m-1})$ • 11