# THÈSE

présentée à
l'École polytechnique

Pour obtenir le titre de Docteur en Sciences
Spécialités : Mathématiques et Informatique

Alin Bostan

## Algorithmique efficace pour des opérations de base en Calcul formel

Soutenue le 9 décembre 2003
devant le jury composé de :

| | | |
|---|---|---|
| Jean-Pierre | Ramis, | président, |
| Richard | Brent, | |
| Gilles | Villard, | rapporteurs, |
| Philippe | Flajolet, | |
| Marc | Giusti, | |
| François | Morain, | |
| Bruno | Salvy, | examinateurs. |

# Table des matières

# Première partie

# Introduction

# Chapitre 1

# Introduction

## Contents

## 1.1 Problématique et contributions : vue d'ensemble

Cette thèse concerne la conception et l'implantation d'algorithmes efficaces pour des opérations de base en calcul formel, ainsi que leurs applications aux domaines connexes, comme la cryptographie ou la théorie effective des nombres. Un objectif important est le défrichage systématique du champ algorithmique des polynômes, en vue d'une extension vers leurs analogues non-commutatifs – les opérateurs différentiels linéaires – *du point de vue de la complexité*. Ce mémoire traite d'abord des questions dans un contexte univarié, en y développant une boîte à outils qui sera ensuite importée dans le cadre plus compliqué des systèmes de polynômes à deux ou plusieurs variables, pour aboutir à des résultats concernant l'algorithmique des opérateurs différentiels.

**Complexité des algorithmes.** L'idéal de tout programmeur est de proposer le *meilleur* algorithme possible pour le problème à résoudre. Le mot *meilleur* a bien sûr, de nombreuses acceptions, mais ici, il s'agit de minimiser la durée des calculs et, lorsque cela est possible, l'encombrement en mémoire. Pour ce faire, une tâche capitale est de donner une estimation du temps et de l'espace mémoire que nécessitera l'exécution d'un algorithme avant même de le programmer. C'est la notion de *complexité d'un algorithme* qui formalise ce genre de mesures. Elle aide à comparer les performances de différents algorithmes entre eux, de la manière la plus intrinsèque possible, c'est-à-dire, idéalement, en dehors de toute dépendance vis-à-vis de l'ordinateur ou du langage de programmation utilisé.

**Complexité arithmétique et complexité binaire.** Sans donner une définition précise, par *complexité arithmétique* d'un algorithme manipulant des objets appartenant à une certaine structure algébrique $R$, on entendra le nombre d'opérations qu'il effectue dans $R$. En règle générale, le décompte des opérations arithmétiques n'est pas un indicateur fidèle du temps réel de calcul, car des phénomènes de croissance peuvent se produire à l'intérieur même de $R$ (des exemples typiques sont $R = \mathbb{Z}$ et $R = \mathbb{Z}[X]$) [1]. Par contre, si $R$ est un corps fini, il est légitime de considérer que les opérations arithmétiques ont un coût constant.

Dans cette thèse, sauf mention contraire, on n'estimera systématiquement que la complexité arithmétique des algorithmes présentés. Lorsqu'il s'agira de tester leur comportement pratique pour confirmer les résultats théoriques de complexité, on choisira systématiquement un corps fini $R$.

**Les objectifs.** De nombreuses formules mathématiques n'ont qu'une valeur théorique et ne débouchent pas sur des calculs pratiques. Un exemple célèbre est constitué par le calcul du déterminant d'une matrice ; dans ce cas, la définition mathématique fournit un algorithme de complexité exponentielle, qui devient vite impraticable et auquel on préférera sans aucun doute l'algorithme du pivot de Gauss, de complexité cubique.

Bien sûr, cet exemple porte en lui une violence pédagogique. Faire baisser la complexité d'un algorithme exponentiel dans la classe polynomiale relève d'une petite révolution scientifique

[1] Une mesure plus réaliste est la *complexité binaire*, qui représente le nombre d'opérations élémentaires (bit) que doit réaliser le processeur pour exécuter un algorithme.

et, historiquement, cela est un phénomène assez rare. Typiquement, la quasi-totalité des questions sur lesquelles je me pencherai dans cette thèse admettent des solutions de complexité polynomiale, c'est-à-dire en $O(n^\alpha)$, où $n$ est la taille de la sortie et $\alpha \geq 1$. L'enjeu est de trouver des solutions algorithmiques qui diminuent l'exposant $\alpha$ et, lorsque cela est faisable, les constantes cachées dans les $O(\ )$. En effet, il existe des algorithmes de bonne complexité asymptotique qui, à cause d'énormes constantes cachées derrière le $O(\ )$, ne deviennent efficaces qu'à partir d'un seuil inintéressant d'un point de vue pratique, pour la technologie actuelle.

Nous considérons aussi des problèmes pour lesquels on connaît déjà des algorithmes *quasi optimaux*, c'est-à-dire presque linéaires en la taille de la sortie, à des facteurs logarithmiques près. Dans de telles situations, on s'intéresse à gagner des facteurs logarithmiques et même constants. À première vue, cela peut paraître une préoccupation d'importance mineure, mais en pratique, pour certaines applications au domaine de la cryptographie, par exemple, les tailles des objets manipulés peuvent atteindre plusieurs dizaines de milliers et le gain d'un facteur constant s'avère un progrès d'importance capitale.

**Bornes inférieures de complexité.** Face à un problème donné, on peut se demander si l'algorithme qu'on utilise est le meilleur, ou bien si on peut espérer trouver mieux. Ceci amène à étudier la borne inférieure des complexités de tous les algorithmes susceptibles de résoudre le problème donné. Ce n'est pas une tâche facile, puisqu'il faut prendre en compte une infinité d'algorithmes. Les questions liées aux bornes inférieures font l'objet d'un domaine de recherche très actif – la théorie de la complexité algébrique. À très peu d'exceptions près, je ne vais pas aborder ce point dans la suite.

⋆ ⋆ ⋆ ⋆ ⋆ ⋆

### 1.1.1 Quelques problèmes illustratifs

Voici une brève liste de problèmes qui illustre les progrès algorithmiques réalisés dans cette thèse. Après avoir donné leur énoncés, j'indique la complexité de la meilleure solution précédemment connue, ainsi que la complexité de la solution proposée dans cette thèse.

1. Étant donné un polynôme $P$ de degré $d$ à coefficients dans un corps $k$, ainsi que des éléments $x_0, \ldots, x_d \in k$, déterminer le nombre d'opérations arithmétiques (additions, multiplications et inversions dans $k$) nécessaires pour calculer les valeurs $P(x_i)$, pour $i = 0, \ldots, d$. À exprimer en fonction du coût $\mathsf{M}(d)$ de la multiplication de deux polynômes de degré $d$.

**Avant :**

$$\frac{7}{2}\,\mathsf{M}(d)\log(d) + O(\mathsf{M}(d)).$$

**Cette thèse :**

$$\frac{3}{2}\,\mathsf{M}(d)\log(d) + O(\mathsf{M}(d)).$$

**Remarque.** Le gain est ici d'un facteur constant et se fait clairement sentir en pratique pour les tailles des polynômes manipulés en cryptographie ($d$ est de l'ordre de plusieurs centaines de milliers). Le même type de gain est obtenu pour le problème inverse – l'interpolation, ainsi que, de manière systématique, pour une classe très vaste de problèmes connexes, voir les Chapitres 4 et 5.

2. Étant donnés des éléments distincts $x_0, \ldots, x_d$ *en progression géométrique* dans un corps $k$, ainsi que des points quelconques $y_0, \ldots, y_d \in k$, quel est le nombre d'opérations arithmétiques nécessaires pour déterminer l'unique polynôme $P \in k[X]$ de degré au plus $d$, tel que $P(x_i) = y_i$, pour $i = 0, \ldots, d$.

**Avant :**

$$\frac{9}{2}\,\mathsf{M}(d)\log(d) + O(\mathsf{M}(d)).$$

**Cette thèse :**

$$2\,\mathsf{M}(d) + O(d).$$

**Remarque.** Ce résultat peut être vu comme une généralisation de la transformée de Fourier inverse. Le résultat analogue étendant la transformée de Fourier directe est connu depuis les années 1970 [198, 27]. Combinant ces deux résultats, on obtient des gains logarithmiques dans les approches évaluation-interpolation, lorsque le choix des points d'évaluation est libre. Un exemple d'application immédiate est *l'amélioration de la complexité du produit des matrices polynomiales*. Ces résultats sont détaillés dans le Chapitre 5.

3. Soit $k$ un corps de caractéristique zéro ou supérieure à $n$, où $n$ est une puissance de 2. Soit $g$ un polynôme de $k[x]$, de degré inférieur à $n$ et soit $m \geq 1$. Quel est le nombre d'opérations dans $k$ nécessaires pour déterminer les coefficients de la somme définie $\sum_{i=0}^{m} g(i)$, en tant que polynôme en $m$ ?

**Avant :**

$$O(n^2).$$

**Cette thèse :**

$$2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n)).$$

**Remarque.** Ce résultat est une conséquence d'algorithmes de conversion rapide entre la base monomiale et la base des factorielles descendantes. Ces algorithmes sont décrits dans le Chapitre 5.

4. Étant donnés deux polynômes $f$ et $g$ de degré $d$, à coefficients dans un corps $k$ de caractéristique nulle ou supérieure à $d^2$, quel est le nombre d'opérations dans $k$ nécessaires pour déterminer le polynôme de $k[X]$ de degré $d^2$, dont les racines sont les sommes $\alpha + \beta$ d'une racine $\alpha$ de $f$ et $\beta$ de $g$ ?

   **Avant :**
   $$O_{\log}\Big(d^2\,\mathsf{M}(d)\Big),$$
   où la notation $O_{\log}$ indique la présence de facteurs logarithmiques en $d$.

   **Cette thèse :**
   $$O\big(\mathsf{M}(d^2)\big).$$

   **Remarque.** La solution classique passe par un calcul de résultant bivarié. Le gain obtenu dans cette thèse est d'un facteur linéaire en $d$, si la multiplication rapide (par FFT) des polynômes est utilisée. L'algorithme correspondant est *quasi-optimal*, car presque linéaire en la taille de la sortie. Le même résultat est valable si au lieu des sommes $\alpha + \beta$, on recherche les produits $\alpha\beta$ ou les quotients $\alpha/\beta$. Ces résultats sont présentés au Chapitre 7.

5. Étant donnés deux polynômes $f$ et $g$ de degré $d$, à coefficients dans un corps $k$ de caractéristique nulle ou supérieure à $d^2$, ainsi qu'un polynôme $H(X, Y) \in k[X, Y]$, quel est le nombre d'opérations dans $k$ nécessaires pour déterminer le polynôme de $k[X]$ de degré $d^2$, dont les racines sont les fonctions $H(\alpha, \beta)$ d'une racine $\alpha$ de $f$ et $\beta$ de $g$ ?

   **Avant :**
   $$O_{\log}\big(d^4\,\mathsf{M}(d)\big).$$

   **Cette thèse :**
   $$O\Big(d\,\big(\mathsf{M}(d^2) + \mathsf{MM}(d)\big)\Big),$$
   où $\mathsf{MM}(d)$ désigne le nombre d'opérations requises par la multiplication de deux matrices $d \times d$.

   **Remarque.** La solution classique passe par un calcul de résultant trivarié. Si la multiplication rapide (par FFT) des polynômes est utilisée, notre alternative économise (au moins) un facteur linéaire en $d$. En utilisant l'algorithme sous-cubique de Strassen [234] pour la multiplication des matrices, notre algorithme devient sous-quadratique en la taille de la sortie. D'un point de vue théorique, en utilisant l'estimation plus fine $\mathsf{MM}(d) = O(d^{2.376})$ [69] notre solution fournit une borne supérieure de $O(d^{1.688})$, qui n'est pas loin de la borne inférieure $O(d^{1.5})$ suggérée par un résultat général de Paterson et Stockmeyer [190]. On renvoie au Chapitre 7 pour une présentation détaillée de ces résultats.

6. Étant donné un polynôme $f \in k[X]$ de degré $d$ et un entier $N \geq 0$, quel est le nombre d'opérations dans $k$ nécessaires pour déterminer le coefficient de $X^{dN/2}$ du polynôme $f^N$ ?

**Avant :**

$$O\big(\mathsf{M}(dN)\big).$$

**Cette thèse :**

$$O\Big(\mathsf{MM}(d)\sqrt{N} + d^2\ \mathsf{M}(\sqrt{N})\log(N)\Big).$$

**Remarque.** La solution classique consiste à calculer tous les coefficients de $f^{dN/2}$ modulo $X^{dN/2+1}$ par exponentiation dichotomique. Si le degré $d$ de $f$ est constant et seul $N$ varie, notre algorithme, présenté au Chapitre 9, gagne un facteur de l'ordre de $\sqrt{N}$. La solution se généralise au problème du calcul d'un terme d'une suite récurrente linéaire à coefficients polynomiaux.

7. Étant données les valeurs que prend un polynôme $P \in k[X]$ de degré $d$ sur la suite arithmétique $0, 1, \ldots, d$, et un élément $a \in k$, quel est le nombre d'opérations dans $k$ nécessaires pour déterminer les *valeurs translatées* $P(a), \ldots, P(a+d)$ ?

**Avant :**

$$O\big(\mathsf{M}(d)\log(d)\big).$$

**Cette thèse :**

$$O\big(\mathsf{M}(d)\big).$$

**Remarque.** La solution classique consiste à déterminer les coefficients de $P$ dans la base monomiale par interpolation aux points $0, 1, \ldots, d$, et à évaluer ensuite $P$ sur les points $a, \ldots, a+d$. Notre solution repose sur des manipulations de séries génératrices bien choisies. Le gain obtenu est d'un facteur logarithmique, voir le Chapitre 9 pour plus de détails.

8. Étant donné un opérateur différentiel linéaire $\mathcal{L}$ d'ordre $n$ à coefficients polynomiaux de degré $n$, quel est le nombre d'opérations nécessaires pour déterminer la relation de récurrence à coefficients polynomiaux satisfaite par une solution série formelle $S$ de l'équation différentielle $\mathcal{L}S = 0$ ?

**Avant :**

$$O\big(n^2\,\mathsf{M}(n)\big).$$

**Cette thèse :**

$$O\big(n\,\mathsf{M}(n)\log(n)\big).$$

**Remarque.** La solution classique est une adaptation de la méthode des coefficients indéterminés. Notre méthode, développée au Chapitre 5, exploite les calculs rapides avec les nombres de Stirling et gagne un facteur $n$.

### 1.1.2 Autres contributions

Voici une liste complémentaire des contributions qu'apporte cette thèse à l'algorithmique de base en calcul formel. Pour une présentation systématique et détaillée, on consultera la Section 1.3.

**Chapitre 4** La mise en pratique du *principe de transposition de Tellegen* et des versions transposées *explicites* des algorithmes de base pour les opérations linéaires sur les polynômes à une variable.

**Chapitre 5** De nouveaux algorithmes pour les *conversions rapides* entre diverses bases polynomiales (monomiale, de Newton, de Bernstein) et sur certains ensembles distingués de points (suites arithmétiques ou géométriques). Amélioration de la complexité du produit des matrices polynomiales. Calcul rapide des nombres de Stirling et application à la sommation symbolique des polynômes.

**Chapitre 6** Équivalence entre l'évaluation multipoint et l'interpolation dans la base monomiale.

**Chapitre 8** Calcul rapide de polynômes minimaux et caractéristiques dans des algèbres quotient de dimension zéro. Amélioration du calcul des paramétrisations rationnelles univariées des variétés de dimension zéro.

**Chapitre 9** Calcul rapide d'un terme de large indice d'une suite récurrente à coefficients polynomiaux.

**Chapitre 10** Une approche de type évaluation / interpolation pour le calcul des plus petit commun multiple et produit symétrique d'opérateurs différentiels linéaires.

⋆ ⋆ ⋆ ⋆ ⋆ ⋆

## 1.2 Paradigmes algorithmiques

Pour construire des algorithmes, il y a peu de recettes générales. Cependant, quelques principes méthodologiques sont à la base de la plupart des algorithmes fondamentaux les plus efficaces en calcul formel. Il s'agit des techniques de *diviser pour régner* et des *pas de bébés / pas de géants*, ainsi que des *conversions rapides* entre différentes structures de données utilisées. Passons brièvement en revue ces paradigmes algorithmiques, qui sont à la base des réponses qu'a pu apporter cette thèse aux questions présentées dans la section précédente.

### 1.2.1 Diviser pour régner

Une stratégie algorithmique bien connue est le *diviser pour régner*. Informellement, pour résoudre un problème de taille $n$, ceci consiste à :

1. diviser les données en $b \geq 1$ paquets de taille à peu près égale à $n/c$, où $c \geq 2$ ;
2. exécuter récursivement la manipulation désirée sur chaque paquet ;
3. combiner les solutions de tous les sous-problèmes pour former la solution du problème initial.

Cette démarche se prête bien à une écriture récursive : le diviser pour régner est répété sur les données morcelées jusqu'à n'avoir à traiter que des données élémentaires, sur lesquelles le travail est immédiat. La complexité $\mathsf{C}(n)$ de la procédure est gouvernée par la récurrence

$$\mathsf{C}(n) \leq b\,\mathsf{C}(n/c) + \mathsf{S}(n), \quad \text{pour tout } \; n \geq 1,$$

où $\mathsf{S}(n)$ désigne le nombre d'opérations nécessaires pour finir d'accomplir la tâche désirée sur toutes les données, lorsqu'elle a déjà été effectuée sur chacun des $b$ paquets. Si, de plus, la fonction de surcoût $\mathsf{S}$ est *super-linéaire*, c'est-à-dire, si elle vérifie les conditions

$$\mathsf{S}(m+n) \geq \mathsf{S}(m) + \mathsf{S}(n) \;\text{ et }\; \mathsf{S}(n) \geq n, \;\text{ pour tout }\; m, n \geq 1,$$

alors il est aisé de déduire l'expression

$$\mathsf{C}(n) \;=\; \begin{cases} O\left(\mathsf{S}(n)\right) & \text{si} \quad b < c, \\ O\left(\mathsf{S}(n)\log_c(n)\right) & \text{si} \quad b = c, \\ O\left(\mathsf{S}(n)\, n^{\log_c(b)-1}\right) & \text{si} \quad b > c. \end{cases}$$

Le cas $b = 1$ est parfois appelé *récursion extrapolative*, tandis que le cas $b = 2$ est appelé *diviser pour régner récursif.*

**Récursion extrapolative.** L'exemple probablement le plus simple de diviser pour régner est *l'exponentiation binaire (ou dichotomique)* : pour un élément $r$ d'un anneau $R$, et pour un entier $n \geq 2$, le calcul de $r^n$ par la formule

$$r^n \;=\; \begin{cases} \left(r^{\frac{n}{2}}\right)^2 & \text{si} \quad n \;\text{ est pair}, \\ r\left(r^{\frac{n-1}{2}}\right)^2 & \text{si} \quad n \;\text{ est impair}, \end{cases}$$

mène à un algorithme utilisant $O(\log(n))$ opérations dans $R$, au lieu de $O(n)$ opérations requises par l'approche naïve. Par exemple, $f^{1000000}$ coûte 23 multiplications dans $R$.

Notons aussi que la méthode d'itération de Newton, traitée au Chapitre 3 de cette thèse, peut être vue comme une version itérative de récursion extrapolative.

**Un exemple de diviser pour régner récursif.** Considérons le problème suivant : étant donné un polynôme $P$ de degré $d$ à coefficients dans un anneau $R$ et un élément $a \in R$, calculer les coefficients du polynôme translaté $Q(X) = P(X + a)$. De manière équivalente, il s'agit de calculer les valeurs des $d$ premières dérivées de $P$ au point $a$.

Notons pour commencer que la solution naïve requiert $O(d^2)$ opérations dans $R$. Une solution de meilleure complexité [253, 25] consiste à diviser $P = P_0 + X^{d/2}P_1$, avec $P_0$ et $P_1$ de degrés inférieurs à $d/2$. Ainsi, l'égalité $P(X + a) = P_0(X + a) + (X + a)^{d/2}P_1(X + a)$ permet de retrouver $P(X + a)$ par une approche diviser pour régner. Par exponentiation binaire, le polynôme $(X + a)^{d/2}$ se calcule en $O(\mathsf{M}(d))$ opérations dans $R$. Ainsi la complexité $\mathsf{C}(d)$ vérifie la récurrence $\mathsf{C}(d) = 2\mathsf{C}(d/2) + O(\mathsf{M}(d))$, d'où $\mathsf{C}(d) = O(\mathsf{M}(d)\log(d))$.

Notons enfin que sous l'hypothèse supplémentaire que les éléments $1, 2, \ldots, d$ sont inversibles dans $R$, il existe une solution de nature différente [5], qui économise le facteur logarithmique en $d$, voir aussi le Chapitre 5 de cette thèse.

### 1.2.2 Pas de bébés / pas de géants

Un deuxième principe algorithmique souvent utilisé tout au long de cette thèse est la technique dite des *pas de bébés / pas de géants.* Illustrons cette méthode sur les deux exemples suivants, de nature très similaire.

Prenons $k$ un corps, $N$ un entier positif et $A$ une $k$-algèbre de dimension $D$ (pas forcément commutative). Par exemple, $A$ peut être une algèbre de polynômes ou de séries formelles, une algèbre de type quotient, ou bien une algèbre de matrices.

- *Évaluation polynomiale dans $A$.* Étant donné un polynôme $P \in k[X]$ de degré $N-1$ et un élément $u \in A$, calculer $P(u)$.
- *Projection des puissances dans $A$.* Étant donné $u \in A$ et une application $k$-linéaire $\ell : A \to k$, calculer les éléments

$$\mathcal{L} \;=\; \big[\ell(1), \ell(u), \ell(u^2), \ldots, \ell(u^{N-1})\big].$$

Ces questions apparaîtront de manière récurrente dans cette thèse. Elles interviennent, par exemple, dans le calcul efficace de polynômes minimaux ou caractéristiques dans $A$.

Les deux problèmes admettent une solution naïve, utilisant $O(N)$ multiplications dans $A$ (des opérations *non-scalaires*) et $O(N)$ additions et multiplications par des éléments de $k$ (des opérations *scalaires*). Notons qu'une opération scalaire coûte $D$ opérations dans $k$. Par la suite, la notation $\mathcal{M}(A)$ désignera le nombre d'opérations dans $k$ nécessaires pour effectuer une multiplication dans $A$. Il est naturel de supposer que $\mathcal{M}(A) \geq D$.

Dans [190], Paterson et Stockmeyer ont proposé l'algorithme suivant pour l'évaluation polynomiale, qui utilise des *pas de bébés / pas de géants* et qui fait une économie de $O(\sqrt{N})$ multiplications dans $A$ par rapport à la méthode directe. Pour simplifier, supposons dans ce qui suit que $N$ est un carré parfait. L'idée est d'obtenir la valeur de $P(u)$ en écrivant l'égalité

$$P(u) = \sum_{0 \leq j < \sqrt{N}} P_j(u)\, u^{j\sqrt{N}}, \qquad \text{avec } \deg(P_j) < \sqrt{N}, \tag{1.1}$$

qui suggère la procédure suivante :

**Pas de bébés** Calculer $1, u, u^2, \ldots, u^{\sqrt{N}}$, en utilisant $\sqrt{N}\mathcal{M}(A)$ opérations dans $k$.

**Pas de géants** Calculer $u^{\sqrt{N}}, u^{2\sqrt{N}}, \ldots, u^{(\sqrt{N}-1)\sqrt{N}}$ et retrouver la valeur $P(u)$ à l'aide de l'équation (1.1). Cela nécessite $\sqrt{N}\mathcal{M}(A) + O(ND)$ opérations dans $k$, car le calcul de chacune des $\sqrt{N}$ valeurs $P_j(u)$ requiert $O(\sqrt{N})$ opérations scalaires.

La complexité de cet algorithme est de $O(\sqrt{N}\mathcal{M}(A) + ND)$ opérations dans $k$ [2].

Pour le problème de projection des puissances, Shoup [225, 227] a proposé une solution similaire, qui repose sur l'utilisation de la structure de $A$-module de l'espace dual $\widehat{A}$ des formes $k$-linéaires sur $A$ : pour $a \in A$ et $\ell \in \widehat{A}$, on définit le *produit transposé*

$$a \circ \ell \;: b \longmapsto \ell(ba).$$

[2] Qui plus est, il est montré dans [190] que cet algorithme est optimal par rapport aux opérations non-scalaires, car il existe des polynômes de $k[X]$ de degré $N$ qui ne peuvent pas s'évaluer en moins de $\sqrt{N}$ multiplications dans $A$.

L'idée est alors d'utiliser l'équation

$$\ell(u^{i\sqrt{N}+j}) = \big(u^{i\sqrt{N}} \circ \ell\big)(u^j), \qquad 0 \leq i, j \leq \sqrt{N} - 1. \tag{1.2}$$

pour en déduire l'algorithme suivant :

**Pas de bébés** Calculer les éléments $1, u, \ldots, u^{\sqrt{N}-1}$ et $v = u^{\sqrt{N}}$.

**Pas de géants** Calculer les formes linéaires $\ell, v \circ \ell, \ldots, v^{\sqrt{N}-1} \circ \ell$ et les évaluer sur les éléments $u^i$, $i < \sqrt{N}$, pour enfin retrouver les valeurs de $\mathcal{L}$ par l'équation (1.2).

Comme le produit transposé $v\circ : \widehat{A} \to \widehat{A}$ par l'élément $v$ est la transposée de l'application $k$-linéaire $v\cdot : A \to A$, un théorème algorithmique – le principe de transposition de Tellegen – implique qu'une multiplication transposée coûte $\mathcal{M}(A)$ opérations dans $k$. Ceci montre que l'algorithme précédent est également de complexité $\sqrt{N}\,\mathcal{M}(A) + O(DN)$.

Notons que cette estimation est impliquée par un argument théorique (un théorème d'existence) mais ne fournit pas un algorithme tant que l'opération produit transposé n'est pas rendue effective. La vraie difficulté consiste donc à *exhiber* un algorithme qui calcule ce produit transposé. Un tel algorithme a été obtenu par Shoup [227] dans le cas $A = k[X]/(f)$. L'une des contributions de cette thèse est de montrer comment, à partir d'un algorithme pour la multiplication dans $A$, obtenir de manière *automatique* un algorithme de même complexité pour le produit transposé. Ceci est possible grâce aux outils de transposition effective rendus explicites dans un contexte univarié dans le Chapitre 4 et étendus à une situation multivariée aux Chapitres 7 et 8.

### 1.2.3 Conversions entre diverses représentations

En calcul formel les objets mathématiques n'existent que par leur *représentation finie* : les calculs sont toujours faits dans un espace vectoriel de dimension finie, les manipulations se faisant sur les coordonnées des éléments dans une *base fixée*.

Le choix de la base peut s'avérer d'une importance cruciale pour la simplification des calculs. Par exemple, les polynômes sont usuellement représentés par leurs coefficients dans la base monomiale, mais il existe d'autres représentations alternatives, par exemple, par les valeurs prises sur un ensemble fixé de points. Il est important de noter que certaines tâches algorithmiques, comme la multiplication, sont plus faciles à résoudre dans la seconde représentation, alors que la première est plus adaptée pour d'autres opérations, comme la division.

Ainsi, pour des besoins d'efficacité des calculs, il est crucial d'examiner dans quelle représentation un problème donné est plus facile à traiter, et aussi, de trouver des algorithmes rapides pour la conversion d'une représentation à l'autre.

Illustrons notre propos par quelques exemples de telles conversions pour les polynômes et séries à une variable, ainsi que pour les opérateurs différentiels à coefficients polynomiaux.

1. Une fraction rationnelle $p(X)/q(X)$ peut être représentée par son développement en série à l'ordre $\deg(p) + \deg(q)$. Un algorithme de conversion directe exploite l'inversion rapide des séries, basée sur l'itération de Newton, voir le Chapitre 3. La conversion

d'une série rationnelle en fraction rationnelle s'opère de manière efficace par un algorithme d'approximation de Padé rapide, qui revient essentiellement à un algorithme d'Euclide étendu, voir la Section 3.3.4. Dans le cas particulier très important où $p = q'$, on peut avantageusement utiliser l'exponentielle rapide d'une série comme alternative de meilleure complexité. Ces conversions sont utilisées à maintes reprises dans cette thèse, notamment aux Chapitres 6 et 7.

2. Un polynôme de degré $d$ peut être représenté par les valeurs qu'il prend sur un ensemble fixé de $d+1$ points $a_0, \ldots, a_d$, ou, plus généralement, par les restes de sa division par $d+1$ polynômes fixés. Algébriquement, les conversions d'une représentation à l'autre reviennent à une forme effective du théorème des restes chinois ; dans le cas des réductions par $X - a_i$, il s'agit de l'évaluation multipoint et de l'interpolation de Lagrange. Les Chapitres 4, 5 et 6 de cette thèse contiennent des contributions à l'algorithmique de l'évaluation et l'interpolation.
3. Un polynôme $P$ de degré $d$ peut être représenté par ses $d$ premières sommes de puissances (de Newton) $N_i(P) = \sum_{P(\alpha)=0} \alpha^i$. Algébriquement, les conversions d'une représentation à l'autre traduisent le passage de l'algèbre quotient $A = k[X]/(P)$ vers son dual $\widehat{A}$. En effet, les $N_i(P)$ représentent les coordonnées de la forme linéaire Trace $\in \widehat{A}$ sur la base duale de la base canonique $1, x, \ldots, x^{d-1}$. Algorithmiquement, ces conversions sont réalisées de manière efficace en exploitant le fait que la série génératrice $\sum_{i\geq 0} N_i(P) X^i$ est rationnelle, égale à réc$(P)'$/réc$(P)$, et en utilisant le point 1. Ce type de conversion est utilisé intensivement au Chapitre 7.
4. Un opérateur différentiel en $d/dx$ à coefficients dans $k(x)$ peut être représenté en tant qu'opérateur différentiel en la dérivation d'Euler $\delta = x\frac{d}{dx}$. Les conversions entre ces deux représentations s'avèrent équivalentes aux conversions d'un polynôme de $k[x]$ entre la base monomiale $1, x, x^2, \ldots$ et la base des factorielles descendantes $1, x, x(x-1), x(x-1)(x-2), \ldots$ Un algorithme rapide pour ces questions est donné au Chapitre 5 de cette thèse.
5. Un opérateur différentiel d'ordre $D$ en $d/dx$ à coefficients polynomiaux de degré au plus $N$ peut être représenté par (une base) des solutions séries tronquées à l'ordre $O(ND)$, au voisinage d'un point ordinaire. La conversion directe revient à la résolution d'une récurrence linéaire à coefficients polynomiaux. La conversion réciproque peut se faire par un calcul d'approximants de Padé-Hermite, ou bien par une approche liée aux Wronskiens. L'efficacité de ces conversions est analysée au Chapitre 10.

⋆ ⋆ ⋆ ⋆ ⋆ ⋆

Voici maintenant un résumé détaillé du contenu de ce mémoire. Les résultats de complexité ne sont pas toujours donnés dans leur forme la plus précise, qui nécessite davantage de notations ; on consultera notamment les introductions de chaque chapitre pour plus de détails.

## 1.3 Détail des contributions

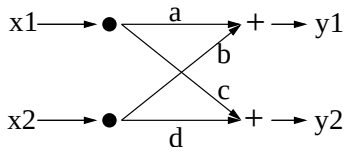
### 1.3.1 Principe de Tellegen

Le Chapitre 4 traite de la mise en pratique du *principe de transposition de Tellegen*, ensemble de règles de transformation pour les algorithmes linéaires. Y sont décrites des versions transposées explicites des algorithmes de base sur les polynômes, ainsi que de nouveaux algorithmes pour l'évaluation multipoint et l'interpolation, qui améliorent les complexités précédemment connues. Les résultats donnés dans cette partie seront systématiquement utilisés tout au long de ce mémoire. Ce chapitre fait l'objet de l'article [35] co-écrit avec G. Lecerf et É. Schost.

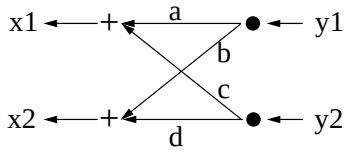
#### Vue d'ensemble

Le *principe de transposition de Tellegen* affirme que, étant donnée une matrice $\mathbf{M}$, l'existence d'un algorithme qui calcule le produit matrice-vecteur $\mathbf{Mv}$ entraîne l'existence d'un algorithme qui calcule le produit $\mathbf{M^t w}$ de la matrice transposée par un vecteur, en utilisant essentiellement le *même nombre d'opérations arithmétiques*.

Considérons l'exemple suivant, qui illustre ce principe en utilisant la représentation par graphes des programmes linéaires. L'algorithme direct prend les valeurs $x_1$ et $x_2$ en entrée et renvoie $y_1 = ax_1 + bx_2$ et $y_2 = cx_1 + dx_2$ en sortie. Les arêtes représentent des multiplications par les valeurs constantes $a, b, c, d$.




Transposer cet algorithme revient à inverser le flot du calcul, c'est-à-dire, inverser le sens des flèches, échanger les $+$ par les $\bullet$ et échanger les entrées et les sorties. L'algorithme ainsi obtenu prend $y_1, y_2$ en entrée et renvoie $x_1 = ay_1 + cy_2$ et $x_2 = by_1 + dy_2$. Il calcule donc bien l'application transposée de l'application initiale. De plus, le nombre d'opérations arithmétiques utilisées par les deux algorithmes est le même : 4 multiplications et 2 additions.




Historiquement, cette technique de transformation des programmes linéaires est issue du domaine des circuits électroniques [239, 29, 192, 8] et le principe même remonte aux années 50 ; on en trouve les traces dans un article de Tellegen [239] sur les réseaux électriques. Le théorème de Tellegen a été redémontré dans les années 80, par Kaminski, Kirkpatrick et Bshouty [128], dans le cadre de la théorie de la complexité algébrique. En calcul formel, il a

été introduit par Fiduccia, Hopcroft et Musinski [80, 81, 119] et popularisé dans les travaux de Ben-Or, Tiwari [19], Kaltofen, Canny, Lakshman [49, 124], Shoup [223, 225, 227], Lecerf, Schost [152], Hanrot, Quercia, Zimmermann [115], Zippel [267], ...

Canny et al. [49] remarquent que le principe de transposition est un cas particulier du *mode inverse* en différentiation automatique pour le calcul du gradient d'une fonction. En effet, les entrées de $\mathbf{M^t v}$ sont les dérivées partielles de $\mathbf{uM^t v}$ par rapport aux coordonnées de $\mathbf{u}$. Ce produit n'est autre que $\mathbf{vMu}$, et le théorème de Baur-Strassen [14] montre que l'on peut calculer ces dérivées partielles au prix d'un surcoût linéaire.

Dans la littérature du calcul formel, à peu d'exceptions près [115, 225], l'usage classique du principe de Tellegen se résume au schéma suivant : connaissant un algorithme pour une certaine opération, on en déduit l'existence d'un algorithme pour l'opération duale, de même complexité. Cependant, lorsqu'il s'agit d'exhiber un tel algorithme, le principe n'est guère utilisé ; à la place, on préfère développer des algorithmes spécifiques pour chaque problème.

Le but principal de ce chapitre, qui est, en même temps, l'un des fils conducteurs de toute cette thèse, est de démontrer que la mise en pratique du principe de Tellegen est bien possible et peut se faire de manière systématique et (quasi-)automatique.

Informellement parlant, on pourrait dire que les programmes sont directement transposés, d'une manière similaire à celle utilisée en différentiation automatique [96], mais en tirant profit des spécificités linéaires. Par ailleurs, lorsqu'un problème linéaire doit être résolu, la démarche adoptée consiste à comprendre son problème dual, pour lequel on peut espérer trouver un algorithme rapide. Si tel est le cas, en re-transposant ce dernier, une solution rapide pour le problème de départ est également obtenue.

Illustrons notre propos par un exemple. Considérons le problème de l'évaluation d'un polynôme $P$ de degré $n$ en une valeur $a$. Ceci est une opération linéaire sur les coefficients de $P$, de matrice $\mathbf{M} = [1, a, \ldots, a^n]$ dans les bases canoniques. En regardant la transposée de $\mathbf{M}$, on déduit aisément que le problème transposé est le suivant : pour une valeur donnée $x_0$, calculer les produits $a^i x_0$, pour $0 \leq i \leq n$. Pour ce problème, un algorithme naturel consiste à multiplier $x_0$ par $a$, ensuite multiplier le résultat par $a$, et ainsi de suite.

Comment obtenir le transposé de cet algorithme ? En simplifiant, la réponse est que l'on n'a qu'à parcourir l'algorithme direct en sens inverse, échanger les entrées et les sorties, puis remplacer chaque instruction par sa *transposée*, obtenue en appliquant un nombre très restreint de règles syntaxiques. Dans ce processus, les boucles `for` montantes deviennent des boucles `for` descendantes.

De cette manière, on obtient *automatiquement* un algorithme pour le problème de départ, à savoir, l'évaluation de $P$ sur $a$. Dans notre cas, il s'avère que l'algorithme transposé coïncide avec la fameuse *méthode de Horner*, voir la Figure 1.1.

Enfin, on peut se demander pourquoi l'algorithme transposé utilise $n$ opérations de plus que l'algorithme direct. Cette perte s'explique par le théorème de Tellegen : il s'agit tout simplement de la différence entre le nombre de colonnes et le nombre de lignes de la matrice $\mathbf{M}$.

$$\mathbf{M} = [1, a, \ldots, a^n]$$

$x_0 \mapsto [x_0, ax_0, \ldots, a^n x_0]$ $\qquad$ $[p_0, \ldots, p_n] \mapsto \sum_{i=0}^{n} p_i a^i$

| | |
|---|---|
| **Input** $x_0$. | **Input** $p = [p_0, \ldots p_n]$. |
| $p_0 \leftarrow x_0$; | **for** $j$ **from** $n$ **downto** 1 **do** |
| **for** $j$ **from** 1 **to** $n$ **do** | $p_j \leftarrow a p_j$; |
| $p_j \leftarrow p_{j-1}$; | $p_{j-1} \leftarrow p_j + p_{j-1}$; |
| $p_j \leftarrow a p_j$; | $x_0 \leftarrow p_0$; |
| **Output** $p = [p_0, \ldots, p_n]$. | **Output** $x_0$. |

FIG. 1.1 – Le schéma de Horner (à droite) obtenu en transposant l'algorithme qui résout le problème dual.

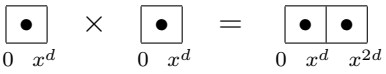
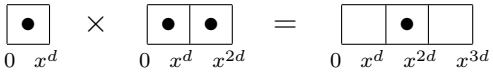
| problème direct | problème transposé |
|---|---|
| multiplication | produit médian |
|   |   |
| division euclidienne | extension de récurrences |
| évaluation multipoint | sommes de Newton pondérées |
| interpolation | décomposition en éléments simples |
| (systèmes de Vandermonde) | (systèmes de Vandermonde transposés) |
| composition modulaire | projection des puissances |
| ... | ... |

FIG. 1.2 – Dictionnaire de Tellegen pour les polynômes univariés.

Notre principale contribution est de démontrer l'utilité pratique du principe de Tellegen. Dans cette optique, nous avons utilisé les mêmes techniques de transformation de programmes linéaires que dans l'exemple précédent afin de transposer de manière systématique les algorithmes concernant les opérations linéaires de base sur les polynômes à une variable : multiplication, division euclidienne, évaluation et interpolation multipoint.

Par conséquent, et contrairement à un préjugé longuement entretenu, il en ressort qu'il est utile de transposer des algorithmes, cela permettant parfois de trouver des solutions algorithmiques de meilleure complexité ou de clarifier le statut de certains algorithmes existant dans la littérature, qui se trouvent être, en fait, les transposés d'algorithmes bien connus.

**Détail des contributions** Nous avons établi une liste d'opérations linéaires de base sur les polynômes à une variable (voir la Figure 1.2) et transposé les algorithmes connus pour les opérations directes.

### Produit médian

Nous avons commencé par transposer la multiplication des polynômes de degré au plus $n$ par un polynôme fixé de degré $m$. La transposée de cette opération consiste à extraire les coefficients de $x^m, x^{m+1}, \ldots, x^{m+n}$ du produit d'un polynôme de degré au plus $m+n$ par un polynôme fixé de degré $m$. Naïvement, le produit transposé peut être calculé en multipliant deux polynômes de degré $m$ et $m+n$, avant d'extraire les coefficients désirés. Le principe de Tellegen implique que pour tout algorithme de multiplication de polynômes de degrés $m$ et $n$, il existe un algorithme qui calcule le produit transposé en le même nombre d'opérations arithmétiques, à des facteurs linéaires en $m$ et $n$ près. Si $m = n$, un facteur 2 est ainsi gagné.

Dans le cas $m = n = d$, cette opération est appelée *produit médian* (voir la Figure 1.2) et a été indépendamment étudiée par Hanrot, Quercia et Zimmermann [115] dans le contexte de l'accélération des calculs de base sur les séries formelles (inversion, extraction de racine carrée, ...), à l'aide de l'opérateur de Newton.

Je signale que cette opération est centrale dans toute la suite de ma thèse. Elle constitue une brique de base, sur laquelle se fonde une bonne partie des algorithmes présentés. Calculer vite des produits transposés est donc un problème important à résoudre avant de passer à la transposition d'autres algorithmes plus compliqués.

Il y a plusieurs algorithmes pour multiplier des polynômes, à chacun correspond donc un algorithme pour calculer le produit transposé. Il s'agit d'expliciter chacun de ces algorithmes, ce que nous avons fait pour l'algorithme de multiplication classique, pour l'algorithme de Karatsuba et pour la transformée de Fourier rapide (FFT).

### Division avec reste et extension des récurrences à coefficients constants

Nous avons ensuite traité la division avec reste d'un polynôme par un polynôme fixé ; nous avons montré que son dual est l'extension des récurrences linéaires à coefficients constants : étant donnés les $n$ premiers termes d'une suite qui vérifie une récurrence linéaire à coefficients constants, il s'agit de calculer la tranche des $n$ termes suivants. Ceci permet de clarifier le statut de l'algorithme de Shoup [223, 227] pour l'extension de récurrences, originellement conçu pour éviter le principe de transposition : il est en fait la transposée de l'algorithme de Strassen [235] pour la division avec reste des polynômes.

### Évaluation multipoint et interpolation

En calcul formel, une opération importante est l'évaluation multipoint : étant donné un polynôme $P = \sum_{i=0}^{m} p_i T^i$ de degré $m$, il s'agit de calculer les valeurs que prend $P$ sur un ensemble de points $a_0, \ldots, a_m$. C'est un problème algorithmique intéressant en soi, mais qui trouve une application notoire, en conjonction avec l'interpolation, dans les approches modulaires. C'est aussi un outil de base pour un certain nombre de questions algorithmiques concernant la multiplication (dense ou creuse) des polynômes et des séries multivariés [49, 127, 152].

En termes matriciels, l'évaluation multipoint se traduit par un produit matrice-vecteur entre la matrice de Vandermonde associée aux points $a_i$ et le vecteur des coefficients du

polynôme $P$, voir la Figure 1.3. Ainsi, le problème transposé est la multiplication d'une matrice de Vandermonde transposée par un vecteur, c'est-à-dire, le calcul de sommes de Newton pondérées (on les appelle ainsi car si tous les $p_i$ valent 1, on retrouve les sommes des puissances des $a_i$).

$$\begin{bmatrix} 1 & a_0 & \cdots & a_0^m \\ 1 & a_1 & \cdots & a_1^m \\ \vdots & \vdots & & \vdots \\ 1 & a_m & \cdots & a_m^m \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} P(a_0) \\ P(a_1) \\ \vdots \\ P(a_m) \end{bmatrix} \text{ et } \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a_0 & a_1 & \cdots & a_m \\ \vdots & \vdots & & \vdots \\ a_0^m & a_1^m & \cdots & a_m^m \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} \sum p_i \\ \sum a_i p_i \\ \vdots \\ \sum a_i^m p_i \end{bmatrix}.$$

FIG. 1.3 – L'évaluation multipoint et sa transposée, les sommes de Newton pondérées.

Nous proposons un algorithme rapide pour le calcul de ces sommes de Newton. Par transposition, nous en déduisons un algorithme pour l'évaluation multipoint de même complexité,

$$\frac{3}{2}\mathsf{M}(m)\log(m) + O\big(\mathsf{M}(m)\big),$$

où $\mathsf{M}(m)$ représente le nombre d'opérations dans le corps de base nécessaires pour multiplier deux polynômes de degré au plus $m$.

Ceci améliore (d'un facteur constant) les algorithmes classiques d'évaluation, dus à Borodin et Moenck [30, 169] et revisités par Strassen [235] et Montgomery [171], voir aussi [255, Section 10.1]. Comme les algorithmes classiques, notre nouvel algorithme repose sur une stratégie diviser pour régner, mais remplace, à chaque niveau de récursion, les divisions avec reste par des produits médians, moins coûteux.

Comme conséquence, nous obtenons également des accélérations (par des facteurs constants) de l'interpolation (c'est-à-dire, la résolution des systèmes de Vandermonde) et de son problème dual (la résolution de systèmes de Vandermonde transposés). De plus, nos méthodes s'étendent au cas plus général du Théorème des restes chinois.

Enfin, nous avons implanté dans la librairie NTL [226] de Shoup tous les algorithmes traités précédemment. Ces implantations fournissent une validation pratique convaincante du principe de Tellegen. En effet, le rapport de 1 entre les temps d'exécution des algorithmes directs et leurs transposés, est bien respecté en pratique. La Figure 1.4 en donne une illustration, dans le cas de l'évaluation multipoint. L'axe des abscisses représente le degré des polynômes, l'axe des ordonnées donne le rapport de temps d'exécution entre l'algorithme direct et l'algorithme transposé. En bas, j'indique l'algorithme utilisé pour la multiplication des polynômes. Naturellement, les versions transposées des multiplications sont utilisées dans l'algorithme transposé de l'évaluation multipoint.

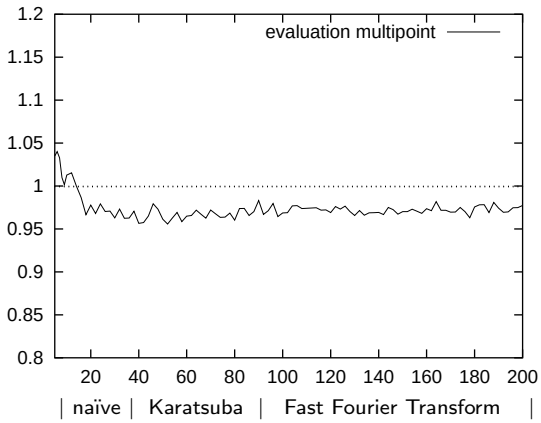



FIG. 1.4 – Rapports de temps de calcul – algorithme direct et transposé.

### 1.3.2 Sur les complexités de l'évaluation et de l'interpolation

Nous donnons des estimations de complexité pour les problèmes d'évaluation et interpolation multipoint dans diverses bases polynomiales. Nous étudions en particulier des cas où les points d'évaluation forment une progression arithmétique ou géométrique. Pour ces deux questions, nous donnons des algorithmes améliorant (d'un facteur constant ou logarithmique) les résultats précédemment connus. Comme applications, nous obtenons des algorithmes rapides pour des questions liées aux calculs avec les opérateurs différentiels, pour la sommation symbolique polynomiale, ainsi que pour la multiplication des matrices polynomiales. Dans un deuxième temps, nous montrons que, sur des corps de caractéristique nulle, l'évaluation et l'interpolation multipoint ont des complexités équivalentes, à un nombre constant de multiplications de polynômes près. Les résultats de ces chapitres sont contenus dans deux prépublications en collaboration avec É. Schost [38, 37].

Nous avons décrit dans le chapitre précédent des algorithmes rapides pour l'évaluation et l'interpolation dans la base monomiale, sur un ensemble arbitraire de points. Ces algorithmes ont une complexité de $O(\mathsf{M}(n)\log(n))$ opérations, où $\mathsf{M}(n)$ représente la complexité de l'algorithme utilisé pour la multiplication des polynômes de degré $n$. Si la FFT est employée, $\mathsf{M}(n)$ est de l'ordre de $O(n\log(n)\log(\log(n)))$, donc ces algorithmes sont presque optimaux, à des facteurs logarithmiques près.

Dans ce chapitre, nous nous penchons sur des questions connexes à l'évaluation et à l'interpolation. Plus précisément, d'une part nous étudions des algorithmes rapides d'évaluation / interpolation dans d'autres bases polynomiales, d'autre part nous cherchons de possibles améliorations, pour des ensembles distingués de points d'évaluation.

Un exemple de base alternative associée aux points $x_0, \ldots, x_{n-1}$ est la *base de Newton*, formée par les polynômes

$$1, (x-x_0), (x-x_0)(x-x_1), \ldots, (x-x_0)\cdots(x-x_{n-2}).$$

Un cas particulier important est constitué par la base des *factorielles descendantes*

$$1,\ x^{\underline{1}} = x,\ x^{\underline{2}} = x(x-1),\ x^{\underline{3}} = x(x-1)(x-2), \ldots,$$

utilisée en algorithmique des opérateurs différentiels et en sommation symbolique [3, 193, 191, 255, 92].

L'évaluation et l'interpolation de Newton sont alors définies comme les problèmes d'évaluation-interpolation dans la base de Newton :

– Étant donnés $x_0, \ldots, x_{n-1}$ et $v_0, \ldots, v_{n-1}$, *l'interpolation de Newton* consiste à déterminer les coefficients $f_0, \ldots, f_{n-1}$ tels que le polynôme

$$F = f_0 + f_1(x - x_0) + f_2(x - x_0)(x - x_1) + \cdots + f_{n-1}(x - x_0)\cdots(x - x_{n-2}) \qquad (1.3)$$

satisfasse $F(x_i) = v_i$, pour $i = 0, \ldots, n-1$.

– Étant donnés $x_0, \ldots, x_{n-1}$ et $f_0, \ldots, f_{n-1}$, *l'évaluation de Newton* consiste à calculer les valeurs $v_0 = F(x_0), \ldots, v_{n-1} = F(x_{n-1})$, où $F$ est le polynôme donné dans la formule (1.3).

**Problématique et détail des contributions**

Nous considérons les trois problèmes suivants :

1. les conversions entre les bases monomiale et celle de Newton,
2. l'évaluation et l'interpolation dans la base monomiale,
3. l'évaluation et l'interpolation de Newton.

Pour toutes ces questions, des algorithmes de complexité $O(\mathsf{M}(n)\log(n))$ existent déjà [25]. Un premier objectif est d'expliciter les constantes cachées derrière le $O(\ )$ dans ces estimations de complexité.

Notre but principal est alors d'obtenir des algorithmes de meilleure complexité pour ces trois questions dans des cas où la suite des points d'évaluation $(x_i)$ a des propriétés spéciales.

En effet, il est bien connu que les formules de *différences divisées* qui sont à la base des algorithmes quadratiques pour ces trois questions [133, 208] se simplifient considérablement lorsque les points $x_i$ forment une progression *arithmétique* ou *géométrique*. Nous montrons comment obtenir des algorithmes *rapides* exploitant de telles simplifications.

**Le cas d'une progression arithmétique.** Dans le cas arithmétique, Gerhard [92] a montré que l'évaluation et l'interpolation de Newton peuvent se réaliser en $\mathsf{M}(n) + O(n)$ opérations [3]. À partir de ce résultat, nous obtenons des améliorations (par des facteurs constants) des deux autres questions. Ainsi, utilisant la base de Newton pour les calculs intermédiaires, nous obtenons un nouvel algorithme pour l'interpolation monomiale sur une suite arithmétique. En faisant rentrer en jeu le principe de Tellegen, nous en déduisons également comme conséquence une amélioration de la complexité de l'évaluation monomiale.

[3] Nous montrons au passage que ces algorithmes sont les transposés des algorithmes de [5] pour le décalage rapide des polynômes.

Ceci permet d'obtenir des conversions rapides entre la base monomiale et celle de Newton, en utilisant à nouveau la base de Newton comme étape intermédiaire.

Notre interêt initial pour l'amélioration de l'évaluation et de l'interpolation sur une suite arithmétique a été motivé par l'étude des récurrences linéaires à coefficients polynomiaux présentée dans le Chapitre 9 de cette thèse. Le record cryptographique obtenu dans [34] demande de travailler avec des polynômes de degré de dizaines (voire centaines) de milliers, si bien que le gain d'un facteur constant devient intéressant dans ces ordres de grandeur.

D'autres applications sont également présentées ; elles concernent principalement les conversions entre la base monomiale et celle des factorielles descendantes et mènent à des améliorations des calculs des *nombres de Stirling*. Ceux-ci sont à la base des algorithmes rapides pour la sommation symbolique des polynômes, ou encore pour la traduction entre équations différentielles linéaires et récurrences linéaires. Cette traduction est une brique de base pour les algorithmes décrits dans le dernier chapitre de la thèse.

**Le cas d'une progression géométrique.** Dans le cas géométrique, nous obtenons également des algorithmes de complexité $\mathsf{M}(n)+O(n)$ pour l'évaluation et l'interpolation de Newton. Ils reposent sur la traduction judicieuse en termes d'égalités de séries génératrices des formules de $q$-différences divisées, similaires à celles utilisées dans le cas arithmétique dans [92].

En considérant les problèmes transposés, nous déduisons que les conversions entre la base monomiale et celle de Newton peuvent se faire en $\mathsf{M}(n)+O(n)$ opérations. Ces résultats ont des conséquences intéressantes pour l'évaluation et l'interpolation dans la base monomiale. Il est connu [198, 27, 5] que l'évaluation d'un polynôme de degré inférieur à $n$ sur $n$ points en progression géométrique coûte $2\mathsf{M}(n)+O(n)$ [4]. Le résultat analogue pour le problème inverse — l'interpolation dans la base monomiale sur des points en progression géométrique — était, à notre connaissance, ouvert. En utilisant la base de Newton pour les calculs intermédiaires, nous aboutissons à un algorithme de complexité $2\mathsf{M}(n)+O(n)$ pour cette question.

En résumé, ceci permet d'exhiber des ensembles distingués de points *dans l'anneau de base*, pour lesquels l'évaluation et l'interpolation sont moins chères que dans le cas général, à savoir d'un facteur logarithmique.

Ce résultat a des conséquences importantes, car bon nombre d'algorithmes fondés sur une approche modulaire peuvent bénéficier du gain du facteur logarithmique induit par notre méthode. Comme exemple, nous améliorons la complexité de la multiplication des matrices polynomiales. Cette dernière est en elle-même un problème algorithmique important, qui est à la base, par exemple, des algorithmes d'approximation de Padé-Hermite (LLL-polynomial) et des traitements rapides d'opérateurs différentiels et de récurrences à coefficients polynomiaux.

**Équivalence entre évaluation et interpolation** Il est bien connu que les complexités de l'évaluation et de l'interpolation sont intimement liées. Par exemple, les algorithmes clas-

[4] En réalité, l'algorithme de [5] a une complexité de $4\mathsf{M}(n)+O(n)$, mais l'emploi adéquat du produit médian décrit dans le chapitre précédent fournit la borne annoncée.

siques d'interpolation rapide [156, 80, 169, 30, 235, 35] reposent tous sur l'évaluation multipoint comme brique de base. Dans ce chapitre nous montrons que les questions d'évaluation et d'interpolation sont équivalentes. Au lieu de donner des algorithmes particuliers, nous comparons les complexités des deux questions, en étudiant des réductions d'une question à l'autre.

De telles réductions ont déjà été proposées dans la littérature [127, 49, 184, 83, 104, 105, 25, 189]. Cependant, aucun théorème d'équivalence n'a encore été établi pour ces questions. Plus précisément, tous les résultats connus font entrer en jeu l'opération additionnelle suivante : étant donnés $x_0, \ldots, x_n$, calculer les coefficients de $\prod_{i=0}^{n}(T - x_i)$, c'est-à-dire les fonctions symétriques élémentaires en les $x_0, \ldots, x_n$. Si on note $\mathsf{E}(n)$, $\mathsf{I}(n)$ et $\mathsf{S}(n)$ les complexités de l'évaluation multipoint, de l'interpolation et respectivement, du calcul des fonctions symétriques élémentaires, les références citées donnent :

$$\mathsf{I}(n) \in O(\mathsf{E}(n) + \mathsf{S}(n)) \quad \text{et} \quad \mathsf{E}(n) \in O(\mathsf{I}(n) + \mathsf{S}(n)). \tag{1.4}$$

Notons que les meilleurs résultats connus [255, Ch. 10] fournissent $\mathsf{S}(n) \in O(\mathsf{M}(n)\log(n))$, où $\mathsf{M}(n)$ est le coût de la multiplication des polynômes de degré $n$.

L'objectif de ce chapitre est de montrer qu'on peut remplacer les termes $\mathsf{S}(n)$ par $\mathsf{M}(n)$ dans les estimations (1.4). Techniquement, notre contribution est de montrer que le calcul des fonctions symétriques se réduit à celui d'une interpolation ou d'une évaluation, à un nombre constant de multiplications de polynômes près.

De fait, nous démontrons un résultat plus précis, qui prend en compte la possible spécificité des points d'évaluation, pour lesquels les problèmes d'évaluation / interpolation se simplifient (cf. Chapitre précédent). Plus exactement, nous montrons que :

– étant donné un algorithme qui exécute l'évaluation sur une famille distinguée de points, on peut en déduire un algorithme qui exécute l'interpolation sur la même famille de points avec la même complexité, à un nombre constant de multiplications polynomiales près ;

– étant donné un algorithme qui exécute l'interpolation sur une famille distinguée de points, on peut en déduire un algorithme qui exécute l'interpolation sur la même famille de points avec la même complexité, à un nombre constant de multiplications polynomiales près.

Nous renvoyons au Chapitre 6 pour une formulation précise de ces résultats. Comme corollaires, nous déduisons en première instance les estimations annoncées :

$$\mathsf{I}(n) \in O(\mathsf{E}(n) + \mathsf{M}(n)) \quad \text{et} \quad \mathsf{E}(n) \in O(\mathsf{I}(n) + \mathsf{M}(n)).$$

Une deuxième conséquence concerne des résultats de [5]. Dans cet article sont étudiées des familles de $n$ points de $\mathbb{C}$ sur lesquelles tout polynôme de degré $n$ s'évalue en $O(\mathsf{M}(n))$ opérations. Nos résultats montrent que ces familles sont précisément celles sur lesquelles tout polynôme de degré $n$ peut être interpolé en $O(\mathsf{M}(n))$ opérations. Par exemple, [5] montre que si $a, b, c, z \in \mathbb{C}^4$, tout polynôme de degré $n$ s'évalue sur les points de la suite $a + bz^i + cz^{2i}$ en temps $O(\mathsf{M}(n))$. On en déduit alors que si ces points sont tous distincts, l'interpolation d'un polynôme de degré $n$ sur ces points s'effectue aussi en temps $O(\mathsf{M}(n))$.

### 1.3.3 Algorithmes rapides pour deux nombres algébriques

Nous étudions certaines opérations élémentaires sur les nombres algébriques : les *produits et sommes composés* et leur généralisation, le *produit diamant de Brawley et Carlitz*. Pour ces questions, nous donnons des algorithmes améliorant sensiblement la quasi-totalité des résultats connus. Ils reposent sur l'utilisation judicieuse de séries génératrices et de la dualité algébrique, qui se traduit algorithmiquement par l'emploi des méthodes de type *pas de bébés / pas de géants*. Les résultats de cette partie sont contenus dans un article écrit en collaboration avec Ph. Flajolet, B. Salvy et É. Schost [33].

#### Vue d'ensemble

Considérons deux polynômes $f$ et $g$ en une variable à coefficients dans un corps $k$. On définit leur produit composé $f \otimes g$ comme le polynôme unitaire de degré $D = \deg(f)\deg(g)$ dont les racines sont les produits $\alpha\beta$, où $\alpha$ et $\beta$ parcourent l'ensemble des racines de $f$ et $g$, comptées avec leurs multiplicités. La définition de la somme composée est d'inspiration analogue. Plus généralement, étant donné un polynôme $H \in k[X, Y]$, on définit le *produit diamant* de $f$ et $g$ par la formule

$$f \diamond_H g \;=\; \prod_{\substack{f(\alpha)=0\\ g(\beta)=0}} \big(T - H(\alpha, \beta)\big).$$

L'opération $\diamond_H$ a été introduite par Brawley et Carlitz [39], qui ont indiqué son utilité pour la construction de polynômes irréductibles de grand degré sur un corps fini [39, 40, 222, 224]. Classiquement, le calcul des polynômes $f \oplus g$ et $f \otimes g$ se fait via les formules suivantes qui les expriment en termes de résultants de polynômes bivariés [159] :

$$(f \oplus g)(x) = \mathrm{Res}_y(f(x - y), g(y)),$$
$$(f \otimes g)(x) = \mathrm{Res}_y(y^m f(x/y), g(y)).$$

Une formule similaire exprime le produit diamant $f \diamond_H g$ comme un résultant trivarié :

$$(f \diamond_H g)(x) = \mathrm{Res}_y\Big(\mathrm{Res}_z\big(x - H(z, y), f(z)\big), g(y)\Big).$$

Si $f$ et $g$ ont des degrés de l'ordre de $\sqrt{D}$, l'exploitation directe de ces formules, par le biais des meilleures méthodes de calcul des résultants multivariés [218, 154, 199, 255, 155], mène à des algorithmes de complexité $O_{\log}\big(D^{3/2}\big)$ pour le produit et la somme composés et de complexité $O_{\log}\big(D^{5/2}\big)$ pour le produit diamant (la notation $O_{\log}$ indique la présence des termes logarithmiques en $D$ cachés).

#### Contributions

Nous proposons comme alternative à ces méthodes classiques des algorithmes fondés sur l'utilisation de la *représentation de Newton* des polynômes à une variable par les sommes

des puissances de leurs racines. Cette idée de changement de représentation a été suggérée par Dvornicich et Traverso dans [74].

Les clés de notre approche sont, d'une part, l'utilisation d'algorithmes rapides [212, 188] pour la conversion entre cette représentation alternative et la représentation monomiale et, d'autre part, la reformulation en termes de séries génératrices des formules dans [74] exprimant $f \otimes g$ et $f \oplus g$ dans leur représentation de Newton.

Ceci nous permet d'obtenir des algorithmes de complexité $O_{\log}(D)$, donc *quasi-optimaux* (c'est-à-dire, linéaires, à des facteurs logarithmiques près, en la taille de la sortie) pour le produit et la somme composés, dans le cas où la caractéristique du corps de base est nulle ou suffisamment grande.

Si l'algorithme pour le produit composé est facile à adapter au cas de la caractéristique quelconque, la situation est différente pour la somme composée. La difficulté provient du fait que la formule pour $f \oplus g$ fait intervenir la série exponentielle, qui n'est pas définie en petite caractéristique.

En introduisant une nouvelle idée de nature combinatoire, le calcul des sommes composées en petite caractéristique $p$ est ramené à une multiplication de deux séries multivariées tronquées en degré $p$ en chaque variable. L'algorithme qui en résulte est de complexité $O_{\log}(D^{1+1/\log(p)})$, donc meilleur que la méthode des résultants, au moins pour $p \geq 5$. Qui plus est, sa non-optimalité provient uniquement du fait qu'à l'heure actuelle on ne connaît pas d'algorithme optimal pour la multiplication de séries multivariées ; toute amélioration de ce dernier problème aurait un impact positif sur notre algorithme.

Nous proposons aussi un algorithme rapide pour le calcul des produits diamants. Le point clé de notre méthode consiste à relier la représentation de Newton de $f \diamond_H g$ aux traces de la multiplication par les puissances successives de $H$ dans l'algèbre quotient $k[X,Y]/(f(X), g(Y))$. Par ce biais, le calcul de $f \diamond_H g$ est ramené à la résolution d'une instance particulière du problème de *projection des puissances* dans l'algèbre quotient $Q = k[X,Y]/(f(X), g(Y))$.

Pour ce dernier, nous proposons un algorithme explicite, de complexité

$$O\left(\sqrt{D}\big(\mathsf{M}(D) + D^{\omega/2}\big)\right),$$

où la fonction $\mathsf{M}$ représente le nombre d'opérations dans $k$ requises pour multiplier deux polynômes de $k[T]$, tandis que $\omega$ est l'exposant de l'algèbre linéaire sur le corps $k$, c'est-à-dire l'infimum des réels $\Omega$ tels que deux matrices $n \times n$ sur $k$ peuvent être multipliées en $O(n^\Omega)$ opérations dans $k$.

Notons que le même résultat de complexité pour la projection des puissances a déjà été obtenu par Shoup [224], voir aussi [124]. Cependant, son résultat est un théorème d'existence (obtenu en appliquant le théorème de Tellegen) et aucun algorithme de cette complexité n'y est exhibé. Par contre, notre algorithme est complètement explicite, voir la Section 7.4 du Chapitre 7 pour plus de détails historiques à ce sujet.

Combinant cet algorithme avec les techniques de conversion rapide entre la représentation de Newton et la représentation monomiale, nous obtenons un algorithme pour le produit

diamant de complexité $O\left(\sqrt{D}\big(\mathsf{M}(D)+D^{\omega/2}\big)\right)$. En prenant la meilleure borne connue actuellement $\omega < 2.376$ [69], et en utilisant la FFT pour la multiplication des séries, pour laquelle $\mathsf{M}(D) = O_{\log}(D)$, la complexité théorique de notre algorithme est sous-quadratique, en $O(D^{1.688})$. Je précise que même en utilisant de l'algèbre linéaire naïve ($\omega = 3$), nos résultats améliorent la complexité de la méthode basée sur le calcul de résultants par un facteur de $\sqrt{D}$.

Les résultats de complexité principaux de ce chapitre sont résumés dans le théorème ci-dessous.

**Théorème 1** *Soit $k$ un corps de caractéristique $p$, soient $f$ et $g$ deux polynômes unitaires dans $k[T]$, de degrés $m$ et $n$ et soit $D = mn$.*

1. *Si $p = 0$ ou $p > D$, alors $f \otimes g$ et $f \oplus g$ peuvent être calculés en $O(\mathsf{M}(D))$ opérations dans $k$.*
2. *Si le produit composé $f \otimes g$ n'a aucune racine de multiplicité supérieure à $p$, alors il peut être calculé en $O\left(p\,\mathsf{M}\big(\frac{D}{p}\big)\log\big(\frac{D}{p}\big)+\mathsf{M}(D)\right)$ opérations dans $k$.*
3. *Si la somme composée $f \oplus g$ n'a aucune racine de multiplicité supérieure à $p$, alors elle peut être calculée en $O\left(p\,\mathsf{M}\big(\frac{D}{p}\big)\log\big(\frac{D}{p}\big)+\mathsf{M}\big(D^{1+\frac{1}{\log(p)}}\big)\right)$ opérations dans $k$.*

*Supposons que $H \in k[X,Y]$ est de degré au plus $m-1$ en $X$ et au plus $n-1$ en $Y$.*

4. *Si le produit diamant $f \diamond_H g$ n'a aucune racine de multiplicité supérieure à $p$, alors il peut être calculé en $O\left(\sqrt{D}\big(\mathsf{M}(D)+D^{\omega/2}\big)\right)$ opérations en $k$.*

### 1.3.4 Algorithmes rapides pour les systèmes polynomiaux

Nous généralisons les méthodes de type *pas de bébés / pas de géants* au cadre des systèmes de polynômes. Nous mettons au point un algorithme accéléré de calcul de *polynômes minimaux* dans des algèbres quotient et nous donnons de nouvelles formules de calcul d'une *paramétrisation rationnelle* fournissant une représentation commode des solutions d'un système polynomial de dimension zéro. Ces formules étendent des résultats de F. Rouillier [204]. Les algorithmes sous-jacents sont implantés en `Magma` [32]. Ce travail a été mené en collaboration avec B. Salvy et É. Schost [36].

**Cadre général.** Cette partie concerne la résolution des systèmes polynomiaux admettant un nombre fini de solutions. Pour fixer les notations, soit $k$ un corps et $\mathcal{I}$ un idéal de dimension zéro de $k[X_1,\ldots,X_n]$ engendré par des polynômes $F_1,\ldots,F_s$. Soit $\mathcal{V}(\mathcal{I})$ l'ensemble fini des solutions (dans $\overline{k}^{\,n}$) du système $F_1 = 0$, …, $F_s = 0$. La problématique de ce chapitre est *le calcul d'une description de la variété de dimension zéro $\mathcal{V}(\mathcal{I})$*, à partir de la donnée des polynômes $F_1,\ldots,F_s$.

Plus précisément, on cherche à déterminer une *représentation* des points de $\mathcal{V}(\mathcal{I})$ à l'aide de polynômes à une variable uniquement. Le calcul de ce type de représentation relève des

questions d'*élimination algébrique*, dont le pendant géométrique est l'idée de *projection* : on cherche à éliminer le plus possible de variables du système d'entrée, le but étant de se ramener à ne manipuler que des polynômes à une variable, pour lesquels il est plus aisé de compter, approcher ou isoler les solutions.

L'exemple suivant, emprunté à [98, 217], va guider notre intuition tout au long de cette section. Considérons les deux courbes planes définies par les polynômes $F_1 = X_1^2 - 1$ et $F_2 = X_1^2 + 2X_2^2 - 2$ de $\mathbb{Q}[X_1, X_2]$ ; leur intersection est la variété de dimension zéro

$$\mathcal{V}(\mathcal{I}) = \left\{ \Big( -1, -\frac{1}{\sqrt{2}} \Big), \Big( -1, \frac{1}{\sqrt{2}} \Big), \Big( 1, -\frac{1}{\sqrt{2}} \Big), \Big( 1, \frac{1}{\sqrt{2}} \Big) \right\}.$$

Regardons pour commencer les projections de $\mathcal{V}(\mathcal{I})$ sur les axes de coordonnées ; il s'agit respectivement des ensembles $\{-1, 1\}$ et $\{-1/\sqrt{2}, 1/\sqrt{2}\}$. Dans les deux cas, ces projections ne sont pas injectives ; on ne saura donc pas les inverser pour retrouver les points de la variété à partir des points projections, voir la Figure 1.5.

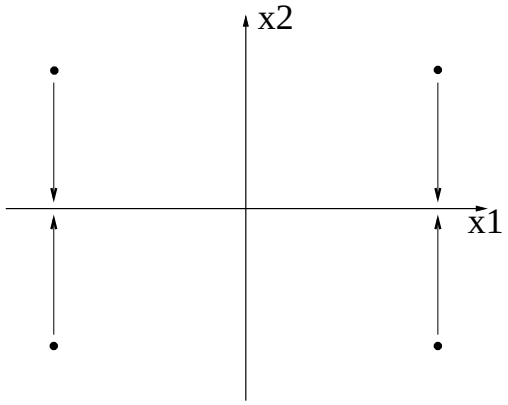



FIG. 1.5 – Projection de $\mathcal{V}(\mathcal{I})$ sur l'axe $X_1$.

La situation est différente si l'on considère la projection sur la droite oblique $(D)$ d'équation $X_1 - X_2 = 0$ (voir la Figure 1.6). Vu que la projection se fait parallèlement aux droites d'équations $X_1 + X_2 =$ constante, l'espace d'arrivée $(D)$ est muni de la coordonnée

$$U = \frac{1}{\sqrt{2}} (X_1 + X_2) \ ^{5}.$$

Les valeurs prises par la forme linéaire $u$ sur $\mathcal{V}(\mathcal{I})$ sont

$$-1 - \frac{1}{\sqrt{2}}, \quad -1 + \frac{1}{\sqrt{2}}, \quad 1 - \frac{1}{\sqrt{2}}, \quad 1 + \frac{1}{\sqrt{2}}$$

[5] C'est un fait de géométrie affine élémentaire : pour toute forme linéaire $u = \sum_i a_i x_i$ la distance du point $\alpha$ à l'hyperplan d'équation $u = 0$ est égale à $\frac{u(\alpha)}{\sqrt{\sum_i a_i^2}}$.

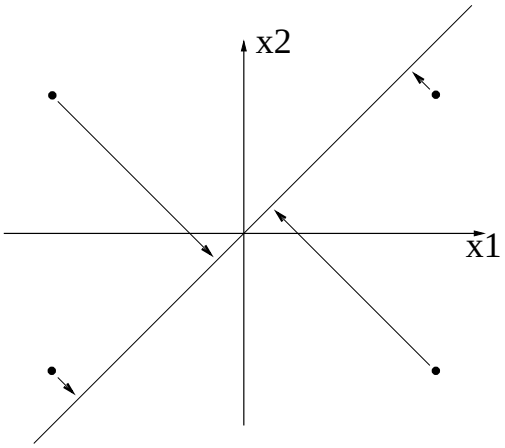



FIG. 1.6 – Projection de $\mathcal{V}(\mathcal{I})$ sur la droite $(D) : X_1 - X_2 = 0$.

et constituent les coordonnées des projections sur la droite $(D)$. Ces valeurs coïncident avec les racines du polynôme $m_u = U^4 - 3\,U^2 + \frac{1}{4}$, qui n'est autre que le polynôme minimal de $u$ dans l'algèbre quotient $A = \mathbb{Q}[X_1, X_2]/(F_1, F_2)$.

La variété projection étant gouvernée par le polynôme univarié $m_u$, le calcul de ce dernier est une *opération d'élimination*.

Il reste à élucider si (et comment) les points de $\mathcal{V}(\mathcal{I})$ peuvent être retrouvés à partir de leurs projections. Ce n'est rien d'autre qu'un problème d'*interpolation*. En effet, étant injective sur $\mathcal{V}(\mathcal{I})$, la fonction de projection est inversible au-dessus de son image.

Prenons par exemple la coordonnée $X_1$ : il existe un unique polynôme $G_1$ de degré 3, tel que

$$G_1\left(-1-\frac{1}{\sqrt{2}}\right) = -1, \quad G_1\left(-1+\frac{1}{\sqrt{2}}\right) = -1, \quad G_1\left(1-\frac{1}{\sqrt{2}}\right) = 1, \quad G_1\left(1+\frac{1}{\sqrt{2}}\right) = 1.$$

En résumé, on obtient donc l'existence de deux polynômes $G_1, G_2$ de degré 3 fournissant une description des points de $\mathcal{V}(\mathcal{I})$, de la forme :

$$\mathcal{V}(\mathcal{I}) \;=\; \left\{ \Big(G_1(a),\; G_2(a)\Big) \quad \Big| \quad a^4 - 3\,a^2 + \frac{1}{4} = 0 \right\}.$$

La situation décrite dans cet exemple se généralise facilement au cas des systèmes sans racines multiples. En effet, prenons $\mathcal{I}$ un idéal radical et $A$ l'algèbre quotient $k[X_1, \ldots, X_n]/\mathcal{I}$, de dimension $D$ sur $k$. Si $u \in A$ prend des valeurs distinctes sur les points de $\mathcal{V}(\mathcal{I})$ (on dira que $u$ est un *élément séparant*), alors la famille $1, \ldots, u^{D-1}$ est une base de $A$ comme $k$-espace vectoriel ; en particulier les coordonnées $x_i$ s'écrivent comme des polynômes en $u$.

Revenons maintenant au cas général. On supposera que l'on travaille sur un corps de caractéristique zéro (ou sur un corps fini, ou plus généralement sur un corps *parfait*[6]) et que

[6]Le corps $k$ est parfait si tout polynôme irréductible de $k[X]$ a toutes ses racines simples, dans une clôture algébrique $\overline{k}$ de $k$.

$u \in A$ est séparant. Sous ces hypothèses on démontre que les coordonnées des points de $\mathcal{V}(\mathcal{I})$ peuvent encore être exprimées comme des *fractions rationnelles* en les racines de $m_u$.

Géométriquement, si $u$ est une forme linéaire en les $x_i$, son polynôme minimal $m_u$ détermine la projection de $\mathcal{V}(\mathcal{I})$ sur l'axe orthogonal à l'hyperplan $u = 0$. (Le pendant algébrique est le théorème de Stickelberger, qui affirme que les racines de $m_u$ sont exactement les valeurs que prend $u$ sur les points de $\mathcal{V}(\mathcal{I})$). L'injectivité de cette projection (équivalant algébriquement à la séparabilité de $u$) assure alors l'existence des formules d'interpolation rationnelle.

Notre effort se concentrera sur le calcul du *polynôme minimal* de $u$ et d'une *paramétrisation rationnelle* de $\mathcal{V}(\mathcal{I})$, c'est-à-dire une famille de polynômes à une variable $\left(\{G_i\}_{1\leq i\leq n}\ ,\ G\right)$ telle que

$$\mathcal{V}(\mathcal{I}) \ = \ \left\{ \ \left(\tfrac{G_1(a)}{G(a)}, \ldots, \tfrac{G_n(a)}{G(a)}\right) \ \middle| \ m_u(a) = 0 \ \right\}.$$

Dans le cas radical, ce problème est habituellement réduit à des manipulations d'algèbre linéaire dans $A$. Des méthodes plus efficaces traitant le cas général ont été proposées par Rouillier [203, 204]. Notre but est d'améliorer l'efficacité de ces méthodes.

Pour ce faire, nous proposons des solutions algorithmiques nouvelles, qui étendent notamment des idées introduites par Shoup [224, 227]. Dans la suite, on suppose connue la structure de $k$-espace vectoriel de $A$. De plus, nos algorithmes exigent le précalcul soit de matrices de multiplication dans $A$, soit de toute la table de multiplication de $A$. Ces objets peuvent être obtenus, par exemple, par le calcul d'une *base de Gröbner* de l'idéal $\mathcal{I}$ [45, 79, 77].

**Le calcul du polynôme minimal.** Soit $u$ dans $A$ et soit $m_u$ son *polynôme minimal*, à calculer. On suppose connue une borne $\delta$ sur le degré de $m_u$. Naturellement, le choix $\delta = D$ convient, mais il y a des situations où l'on connaît a priori une borne plus fine, voir [55, 86, 90, 216].

L'algorithme classique pour le calcul de $m_u$ consiste à exprimer les $\delta$ premières puissances de $u$ sur une base de $A$, puis à rechercher une dépendance linéaire entre elles. Notons que si l'entrée de cet algorithme est la matrice de multiplication de $u$, sa complexité est de $O(\delta D^2)$ pour le calcul des puissances de $u$ et de $O(D^\omega)$ pour la partie algèbre linéaire.

> Dans notre exemple, une base monomiale de $A = \mathbb{Q}[X_1, X_2]/(F_1, F_2)$ comme espace vectoriel est $1, x_2, x_1, x_1x_2$ et la matrice des $\delta = 4$ premières puissances de $u = x_1+x_2$ dans cette base est
>
> $$\begin{bmatrix} 1 & 0 & 3/2 & 0 & 17/4 \\ 0 & 1 & 0 & 7/2 & 0 \\ 0 & 1 & 0 & 5/2 & 0 \\ 0 & 0 & 2 & 0 & 6 \end{bmatrix}.$$
>
> Une base de son noyau est formée par le seul élément $[1 \quad 0 \quad -12 \quad 0 \quad 4]$. Ceci montre que le polynôme minimal de $u$ est $m_u = T^4 - 3T^2 + 1/4$.

Une première amélioration est de considérer les valeurs prises par une forme linéaire $\ell$ sur les puissances de $u$ : la suite $\left(\ell(u^i)\right)_{i\geq 0}$ admet une relation de récurrence minimale, qui coïncide génériquement avec $m_u$ et qui peut être calculée efficacement à l'aide de l'algorithme de

Berlekamp-Massey. Ceci suggère une première méthode : calculer les $2\delta$ premières puissances de $u$, y évaluer $\ell$, puis en déduire un candidat pour le polynôme minimal recherché. Ceci exige la possibilité de multiplier par $u$ ; c'est pourquoi, l'entrée de ce premier algorithme sera la matrice de multiplication par $u$ dans $A$.

> Choisissons la forme linéaire $\ell$ dont les coordonnées dans la base duale $\widehat{1}, \widehat{x_2}, \widehat{x_1}, \widehat{x_1x_2}$ sont $2, -1, -3, 1$. Alors les valeurs de $\ell$ sur $1, u, \ldots, u^7$ sont
>
> $$\left[\begin{array}{cccccccc} 2 & -4 & 5 & -11 & 29/2 & -32 & 169/4 & -373/4 \end{array}\right].$$
>
> La récurrence minimale satisfaite par cette suite peut être déterminée par l'algorithme de Berlekamp-Massey, ou bien en calculant un approximant de Padé de type $(3, 4)$ à l'ordre 8 pour sa série génératrice
>
> $$2 - 4T + 5T^2 - 11T^3 + 29/2T^4 - 32T^5 + 169/4T^6 - 373/4T^7 + O(T^8).$$
>
> Ainsi, on obtient $m_u = T^4 - 3T^2 + 1/4$.

Notons que le goulot d'étranglement de cet algorithme est le calcul de toutes les $\delta$ premières puissances de $u$. Ainsi, dans le cas *générique* $\delta = D$, sa complexité asymptotique n'est pas meilleure que celle de l'algorithme classique.

Dans le contexte de la factorisation des polynômes sur des corps finis, Shoup [224, 227] a montré comment accélérer ces calculs dans le cas des polynômes à une variable, quand $A = k[X]/(f)$. Son idée est d'adapter la méthode des *pas de bébés / pas de géants* pour l'évaluation rapide des polynômes, due à Paterson et Stockmeyer [190], en faisant appel à la structure de $A$-module du dual $\widehat{A}$ ; l'utilisation astucieuse de cette structure permet d'éviter le calcul de toutes les $2\delta$ puissances de l'élément $u$.

Nous démontrons que l'idée s'étend aux cas des polynômes à plusieurs variables. Ceci fournit une deuxième méthode pour le calcul du polynôme minimal. La difficulté principale de cette approche consiste à obtenir une implantation efficace des opérations dans $\widehat{A}$. Notre solution exige pour le moment une entrée plus forte que ci-dessus : la table de multiplication de $A$ ; cette entrée est également utilisée par exemple dans les algorithmes de [6, 204]. Nos résultats sont précisés dans le théorème suivant.

**Théorème 2** *Soit $D$ la dimension de $A$ comme $k$-espace vectoriel, soit $u$ dans $A$ et $m_u$ son polynôme minimal. On suppose que $\delta$ est une borne sur le degré de $m_u$.*

1. *Si la matrice de la multiplication par $u$ est connue, alors $m_u$ peut être calculé par un algorithme probabiliste en $O(\delta D^2)$ opérations dans $k$.*
2. *Si la table de multiplication de $A$ est connue, alors $m_u$ peut être calculé par un algorithme probabiliste en $O(2^n \delta^{1/2} D^2)$ opérations dans $k$.*

*Dans les deux cas, l'algorithme choisit $D$ valeurs dans $k$ ; si ces valeurs sont choisies dans un sous-ensemble fini $\Gamma$ de $k$, tous les choix, excepté au plus $\delta|\Gamma|^{D-1}$, assurent le succès.*

La complexité est $O(D^3)$ dans le premier cas et $O(2^n D^{5/2})$ dans le deuxième. Pour $n$ fixé, le gain est d'ordre $\sqrt{D}$, ce qui est typique pour les techniques du type *pas de bébé / pas de géant*, qui sont à la base de notre deuxième approche.

L'aspect probabiliste provient du choix d'une forme linéaire sur $A$. Pour les mauvais choix, la sortie de nos algorithmes est un diviseur strict du polynôme minimal recherché. Si le degré de la sortie coïncide avec la borne supérieure $\delta$, alors cette sortie est nécessairement correcte. Nous pouvons également estimer la probabilité d'un choix malheureux, ou évaluer le polynôme minimal candidat sur $u$.

**Le calcul d'une paramétrisation rationnelle.** Étant donné l'élément $u$ et une forme linéaire $\ell$ sur $A$, nous introduisons la série génératrice de $k[[U^{-1}]]$

$$R(u,\ell) := \sum_{i\geq 0} \frac{\ell(u^i)}{U^{i+1}}.$$

Un premier résultat concerne la rationalité de cette série. Nous montrons que $R(u,\ell)$ est de la forme

$$\frac{G_{u,\ell}}{m_u},$$

où $G_{u,\ell}$ est un polynôme de degré $< \deg(m_u)$. Qui plus est, cette fraction est irréductible pour $\ell$ *générique*. Supposons dans ce qui suit que tel est le cas.

Si de plus $u$ est séparant, nous montrons que de telles séries permettent de calculer des paramétrisations des points de $\mathcal{V}(\mathcal{I})$ ; ceci donne des formules qui étendent celles de Rouillier [204]. Nos formules sont valides sous certaines hypothèses de généricité, précisées dans le théorème ci-dessous. Notons que ces hypothèses sont automatiquement remplies si $\mathcal{I}$ est un idéal radical.

**Proposition 1** *Si*

$$R(u,\ell) = \frac{G_{u,\ell}}{m_u} \quad \textit{et} \quad R(u, x_i \circ \ell) = \frac{G_{u,x_i\circ\ell}}{m_u}$$

*alors la famille*

$$(\{G_{u,x_i\circ\ell}\}_{1\leq i\leq n}\ ,\ G_{u,\ell})$$

*est une paramétrisation rationnelle de* $\mathcal{V}(\mathcal{I})$.

Plus généralement, nous démontrons que, pour tout $v \in A$, l'égalité suivante a lieu :

$$\frac{G_{u,v\circ\ell}(u(\alpha))}{G_{u,\ell}(u(\alpha))} = v(\alpha). \tag{1.5}$$

C'est le résultat central de ce chapitre et sa démonstration dans le cas général est assez technique. Pour en donner l'idée, j'indique maintenant brièvement les ingrédients de sa preuve dans le cas simple où $\mathcal{I}$ est *radical* :

- $m_u = \prod_{\alpha\in\mathcal{V}(\mathcal{I})}(U - u(\alpha))$. C'est une conséquence du théorème de Stickelberger, dont l'interprétation géométrique a été donnée dans les sections précédentes.

- $\widehat{A}$ est engendré (comme $k$-espace vectoriel) par les évaluations $\{\mathrm{ev}_\alpha\}$. C'est une généralisation des formules d'interpolation de Lagrange : pour toute racine $\alpha$, il existe un élément $p_\alpha$ de $A$ qui prend la valeur 1 en $\alpha$ et qui s'annule sur les autres racines. Ceci implique l'indépendance linéaire des $\{\mathrm{ev}_\alpha\}$ : si la combinaison linéaire $\Lambda = \sum_\alpha c_\alpha \mathrm{ev}_\alpha$ est nulle, alors $\Lambda(p_\alpha) = c_\alpha$ sont tous nuls.

Supposons que $\ell = \sum_\alpha c_\alpha \mathrm{ev}_\alpha$. Alors, on a la suite d'égalités de séries génératrices

$$R(u, v\circ\ell) = \sum_{i\geq 0} \frac{\ell(vu^i)}{U^{i+1}} = \sum_{i\geq 0} \frac{\sum_\alpha c_\alpha v(\alpha)u(\alpha)^i}{U^{i+1}} = \sum_\alpha c_\alpha v(\alpha) \sum_{i\geq 0} \frac{u(\alpha)^i}{U^{i+1}} = \sum_\alpha \frac{c_\alpha v(\alpha)}{U-u(\alpha)},$$

qui fournit une description *explicite* des numérateurs $G_{u,v\circ\ell}$ comme polynômes interpolant

$$G_{u,v\circ\ell} = \sum_\alpha c_\alpha v(\alpha) \prod_{\beta\neq\alpha} \Big(U - u(\beta)\Big),$$

description de laquelle la formule (1.5) découle immédiatement.

Dans notre exemple, le calcul des 8 premiers termes de chacune des séries $R(u,\ell)$, $R(x_1\circ u,\ell)$ et $R(x_2\circ u,\ell)$ permet de retrouver leur forme rationnelle. Ainsi

$$R(u,\ell) = \frac{2U^3-4U^2-U+1}{U^4-3U^2+\frac{1}{4}}$$

$$R(x_1\circ u,\ell) = \frac{-3U^3+3U^2+\frac{5}{2}U-\frac{1}{2}}{U^4-3U^2+\frac{1}{4}} \quad \text{et} \quad R(x_2\circ u,\ell) = \frac{-U^3+2U^2-\frac{3}{2}U}{U^4-3U^2+\frac{1}{4}}.$$

Ceux-ci fournissent la paramétrisation voulue

$$\left(\frac{-3U^3+3U^2+\frac{5}{2}U-\frac{1}{2}}{2U^3-4U^2-U+1}, \frac{-U^3+2U^2-\frac{3}{2}U}{2U^3-4U^2-U+1}\right) = \left(-U^3+\frac{7}{2}U, U^3-\frac{5}{2}U\right).$$

Ce résultat est une généralisation des formules de Rouillier [203, 204] ; il ne requiert pas l'utilisation d'une forme linéaire spécifique. Dans [203, 204], cette forme spécifique, la *trace*, est calculée à partir de la table de multiplication de $A$. Les formules précédentes permettent d'éviter ce précalcul, car presque toute forme peut être utilisée. Une conséquence importante de ce fait est que les algorithmes correspondant à la première approche n'exigent en entrée que la matrice de multiplication de $u$.

Pour utiliser ces formules en pratique, la tâche calculatoire est tout à fait semblable à celle exigée pour calculer un polynôme minimal : évaluer des formes linéaires sur des puissances de $u$. Ainsi, comme précédemment, deux méthodes coexistent : l'approche directe, qui nécessite seulement des matrices de multiplication, et son amélioration basée sur l'idée de Shoup, qui exige toute la table de multiplication de $A$.

La première approche donne un algorithme dont la complexité est du même ordre que celle de l'algorithme de Rouillier [204], mais notre entrée est plus faible. La deuxième approche fournit un algorithme qui prend la même entrée que celui de [204] ; sa complexité est de l'ordre $O(n2^nD^{5/2})$. Ceci s'avère meilleur quand le nombre de variables est fixé, le gain étant, comme précédemment, de l'ordre de $\sqrt{D}$.

**Théorème 3** *Soient $k$ un corps parfait, $u$ un élément séparant dans $A$ et $m_u$ son polynôme minimal. Soit $D$ la dimension de $A$ comme $k$-espace vectoriel et soit $\delta$ une borne sur le degré de $m_u$. On suppose que :*
*– la caractéristique du corps $k$ est soit égale à zéro, soit au moins égale à $\min\{s,\ \sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}\}$*
*– le degré de $m_u$ est le degré du polynôme minimal d'un élément générique dans $A$.*

1. *Si les matrices de multiplication par $u$ et $x_1, \ldots, x_n$ sont connues, alors une paramétrisation de $\mathcal{V}(\mathcal{I})$ peut être calculée par un algorithme probabiliste en $O(\delta D^2 + nD^2)$ opérations dans $k$.*
2. *Si la table de multiplication de $A$ est connue, une telle paramétrisation peut être calculée par un algorithme probabiliste en $O(n2^n\delta^{1/2}D^2)$ opérations dans $k$.*

*Dans les deux cas, l'algorithme choisit $D$ valeurs dans $k$ ; si ces valeurs sont choisies dans un sous-ensemble fini $\Gamma$ de $k$, tous les choix, excepté au plus $\delta|\Gamma|^{D-1}$, assurent le succès.*

L'aspect probabiliste est de même nature que dans le Théorème 2, et provient du choix d'une forme linéaire sur $A$. Si $\mathcal{I}$ est un idéal radical, il est facile de vérifier l'exactitude de la sortie. La dernière affirmation du Théorème 3 permet d'estimer la probabilité de choisir une mauvaise forme linéaire.

Nous avons implanté les algorithmes mentionnés dans les Théorèmes 2 et 3 dans le système de calcul formel Magma [32]. Nos expériences ont montré leur bon comportement par rapport à une implantation *naïve* de l'algorithme de Rouillier [204].

### 1.3.5 Algorithmes rapides pour les récurrences linéaires à coefficients polynomiaux

Nous étudions la question du calcul rapide d'*un terme* d'une suite récurrente linéaire à coefficients polynomiaux. Nous améliorons un algorithme dû à Chudnovsky et Chudnovsky [59] qui permet de calculer un terme quelconque d'une telle suite en évitant le calcul de tous les termes intermédiaires. Cet algorithme repose sur une technique de type *pas de bébés / pas de géants* et requiert un nombre d'opérations essentiellement linéaire en $\sqrt{n}$ pour le calcul du $n$-ième terme. Comme application, nous obtenons des améliorations théoriques et pratiques de méthodes de comptage de points utilisées en cryptographie. Ce travail a été mené en collaboration avec P. Gaudry et É. Schost [34].

Dans ce chapitre, nous étudions des questions de complexité liées aux suites récurrentes linéaires à coefficients polynomiaux. Pour fixer les notations, soit $R$ un anneau commutatif unitaire et soit $A$ une matrice $m \times m$ dont les entrées sont des polynômes linéaires de $R[X]$. Supposons que $(U_i)_{i\geq 0}$ est une suite de vecteurs de $R^m$ définie par la récurrence linéaire

$$U_{i+1} = A(i+1)\,U_i, \quad \text{pour tout} \quad i \geq 0.$$

Notre objectif principal est de calculer rapidement un terme arbitraire $U_n$ d'une telle suite.

Un cas particulier bien connu est celui des récurrences à coefficients constants : le $n$-ième terme peut être calculé en complexité linéaire en $\log(n)$, par exponentiation binaire de la matrice constante $A$.

Dans le cas général, les choses se compliquent : à ce jour, on ne connaît pas d'algorithme polynomial en $\log(n)$. Nous illustrons les idées de cette section par l'exemple simple suivant. Prenons $a \in \mathbb{Q}$. On se propose de calculer le $n$-ième terme de la suite

$$u_i = (a+1)(a+2)\cdots(a+i).$$

Cette suite vérifie la récurrence à coefficients polynomiaux

$$u_{i+1} - (a+i+1)u_i = 0, \quad i \geq 0.$$

Notons pour commencer que l'approche directe exige $O(n)$ opérations dans $\mathbb{Q}$ pour le calcul de $u_n$. La meilleure solution précédemment connue a été proposée par Chudnovsky et Chudnovsky [59] ; il s'agit d'une généralisation d'un algorithme de Pollard [195] et Strassen [237] pour la factorisation déterministe des entiers. Cet algorithme utilise des *pas de bébés / pas de géants* et requiert un nombre d'opérations arithmétiques presque linéaire (à des facteurs logarithmiques près) en $\sqrt{n}$ pour le calcul du $n$-ième terme d'une suite.

J'explique maintenant l'algorithme de [195, 237, 59] sur ce même exemple et je présente ensuite nos améliorations. Pour simplifier, supposons que $n$ est un carré parfait. L'idée commune aux algorithmes [195, 237, 59] est de poser

$$C(X) = (X+a+1)(X+a+2)\cdots(X+a+\sqrt{n}),$$

afin d'obtenir la valeur de $u_n$ à l'aide de l'équation

$$u_n = \prod_{j=0}^{\sqrt{n}-1} C\big(j\sqrt{n}\big). \tag{1.6}$$

Cette égalité suggère la procédure suivante :

**Pas de bébés** Calculer les coefficients de $C$. Ceci peut être fait en $O(\mathsf{M}(\sqrt{n})\log(n))$ opérations dans $k$, de manière récursive (en construisant l'arbre binaire de feuilles $X+a+i$).

**Pas de géants** Évaluer $C$ sur les points $\big\{0, \sqrt{n}, 2\sqrt{n}, \ldots, (\sqrt{n}-1)\sqrt{n}\big\}$ et retrouver la valeur de $u_n$ à l'aide de l'équation (1.6). En utilisant un algorithme d'évaluation multipoint rapide, ceci peut se faire en $O(\mathsf{M}(\sqrt{n})\log(n))$ opérations dans $k$.

Le coût total de cet algorithme est de $O\big(\mathsf{M}\big(\sqrt{n}\big)\log(n)\big)$ opérations dans $k$ pour le calcul du terme $u_n$. Si la FFT est utilisée pour la multiplication des polynômes, le gain par rapport à la méthode directe est de l'ordre de $\sqrt{n}$, à des facteurs logarithmiques près, ce qui est typique des approches *pas de bébés / pas de géants*.

Lorsque $n$ est très grand (c'est le cas des algorithmes de factorisation d'entiers), supprimer des facteurs logarithmiques peut induire un gain sensible en pratique. Chudnovsky et Chudnovsky [59] indiquent comment diminuer la complexité des *pas de bébés* à $O\big(\mathsf{M}\big(\sqrt{n}\big)\big)$ en temps et linéaire en $\sqrt{n}$ en mémoire. L'idée est de diviser pour régner, en exploitant le fait que les points $1, 2, \ldots, n$ sont en progression arithmétique. Dans le cas matriciel, elle requiert des produits de matrices polynomiales $m \times m$ de degré $O(\sqrt{n})$, ce qui explique la complexité en $O(m^\omega\, \mathsf{M}(\sqrt{n}))$ obtenue dans [59] [7].

Par contre, pour les *pas de géants*, aucun algorithme d'évaluation sur une progression arithmétique économisant le facteur log n'est actuellement connu (la situation est différente pour une progression géométrique, voir le Chapitre 5 pour plus de détails sur ces questions). C'est pourquoi, dans [59] le facteur log n'est pas gagné sur l'ensemble de l'algorithme précédent.

Un premier objectif de ce chapitre est de faire disparaître la dépendence logarithmique en $n$ dans la complexité en espace mémoire. Une deuxième contribution concerne *le cas matriciel* : nous améliorons la complexité en temps des *pas de bébés* de [59], et arrivons à remplacer $O(m^\omega\, \mathsf{M}(\sqrt{n}))$ par $O(m^\omega\, \sqrt{n})$. En pratique, pour de grandes valeurs de $n$, cela se traduit par des gains importants ; dans notre application, $n$ est de l'ordre de $2^{32}$.

La clé de notre approche concerne un point d'algorithmique de base : étant données les valeurs prises par un polynôme $P(X)$ sur un ensemble de points, calculer rapidement les valeurs que prend le même polynôme $P$ sur une *translation* de l'ensemble de points. La meilleure solution précédemment connue consiste à utiliser les techniques d'évaluation et interpolation rapides. Nous proposons un algorithme de meilleure complexité (tant en temps qu'en espace mémoire) dans le cas particulier où la suite des points est en progression arithmétique.

**Proposition 2** *Soit $R$ un anneau commutatif unitaire et $d \in \mathbb{N}$ tel que $1, \ldots, d$ sont inversibles dans $R$. Soit $P$ dans $R[X]$ de degré $d$ et supposons connues les valeurs*

$$P(0), \ldots, P(d).$$

*Si $a \in R$, tel que $a - d, \ldots, a + d$ sont inversibles dans $R$, alors les valeurs translatées*

$$P(a), \ldots, P(a + d)$$

*peuvent être calculées en utilisant $O(\mathsf{M}(d))$ opérations dans $R$ et espace mémoire $O(d)$.*

Ce résultat nous permet d'utiliser une idée de changement de représentation dans l'algorithme précédent : au lieu de travailler avec les coefficients du polynôme $C$, on va plutôt manipuler ses *valeurs*. En effet, pour exploiter l'équation (1.6), il suffit de connaître les valeurs de $C$ sur les points $0, \sqrt{n}, \ldots, (\sqrt{n} - 1)\sqrt{n}$. Pour ce faire, on commence par montrer que les valeurs $C(0), C(1), \ldots, C(\sqrt{n})$ peuvent être calculées en complexité $O(\sqrt{n})$ en temps et en mémoire. L'idée est d'utiliser la récurrence

$$C(i + 1) = C(i) \cdot \frac{i + a + \sqrt{n} + 1}{i + a + 1},$$

[7] Notons que nos résultats du Chapitre 5 concernant la multiplication des matrices polynomiales améliorent déjà cette complexité à $O(m^\omega \sqrt{n} + m^2\, \mathsf{M}(\sqrt{n}))$.

dont le terme initial $C(0) = (a+1)\cdots(a+\sqrt{n})$ se calcule en $O(\sqrt{n})$ opérations.

Par la Proposition 2, les valeurs translatées $C(\sqrt{n}), C(\sqrt{n}+1), \ldots, C(2\sqrt{n})$ peuvent être alors obtenues en complexité $O(\mathsf{M}(\sqrt{n}))$ en temps et $O(\sqrt{n})$ en mémoire. À ce stade, on connaît les $\sqrt{n}$ valeurs de $C(0), C(2), \ldots, C(2\sqrt{n})$, donc une nouvelle application de la Proposition 2 permet d'obtenir les valeurs de $C(2\sqrt{n}), \ldots, C(4\sqrt{n})$ pour le même coût. En appliquant ce schéma $\log(n)$ fois, on arrive à récupérer les valeurs $C(0), C(\sqrt{n}), \ldots, C((\sqrt{n}-1)\sqrt{n})$, pour un coût global de $O(\mathsf{M}(\sqrt{n})\log(n))$ en temps et seulement $O(\sqrt{n})$ en mémoire [8].

En étendant ces idées au cas non-scalaire, nous obtenons le résultat principal de ce chapitre.

**Théorème 4** *Soit $A$ une matrice $m \times m$ dont les entrées sont des polynômes linéaires de $R[X]$ et soit $U_0$ un vecteur de $R^m$. Supposons que $(U_i)_{i\geq 0}$ est une suite d'éléments de $R^m$ définie par la récurrence linéaire*

$$U_{i+1} = A(i+1)\,U_i, \quad \textit{pour tout } \; i \geq 0.$$

*Soit $n > 0$ un entier tel que les éléments $1, \ldots, 2\lceil\sqrt{n}\rceil + 1$ soient inversibles dans $R$. Alors le vecteur $U_n$ peut être calculé en $O\big(m^\omega\sqrt{n} + m^2\,\mathsf{M}\big(\sqrt{n}\big)\log(n)\big)$ opérations dans $R$ et espace mémoire $O(m^2\sqrt{n})$.*

### Application au calcul rapide de l'opérateur de Cartier-Manin

Je conclus ce chapitre avec un exemple d'application de nos résultats sur les récurrences au domaine de la cryptographie. Plus exactement, on s'intéresse au comptage de points des Jacobiennes de courbes hyperelliptiques sur des corps finis. Ce type de calcul est une étape nécessaire pour mettre en œuvre des cryptosystèmes fondés sur le problème du logarithme discret. Notons que pour des raisons de sécurité cryptographique, le but est d'obtenir une courbe dont la Jacobienne soit un groupe de cardinal ayant un facteur entier de taille au moins $2^{130}$.

Soit donc $\mathcal{C}$ une courbe de genre $g$ sur un corps fini $K$ de caractéristique $p \geq 5$. On suppose que $\mathcal{C}$ est définie par l'équation $y^2 = f(x)$, où $f \in K[X]$ est un polynôme de degré $2g+1$. La *matrice de Hasse-Witt de $\mathcal{C}$* [116] est la matrice $g \times g$ à coefficients dans $K$, dont l'entrée $(i,j)$ est le coefficient de $X^{ip-j}$ dans $f^{(p-1)/2}$.

Par un résultat classique de Cartier [52] et Manin [163], la connaissance explicite de cette matrice fournit une méthode de calcul du cardinal de la Jacobienne de $\mathcal{C}$ modulo la caractéristique $p$ du corps de base. Si le genre $g$ de $\mathcal{C}$, ainsi que le degré $d$ de l'extension $\mathbb{F}_p \to K$ sont petits par rapport à la caractéristique $p$, le goulot d'étranglement de cette méthode est le calcul de la matrice de Hasse-Witt. Dans l'approche classique [88, 166], la construction de cette matrice est faite en élevant directement $f(X)$ à la puissance $(p-1)/2$, ce qui entraîne un coût essentiellement linéaire en $p$, à des facteurs logarithmiques près. En pratique, ceci impose dans [166] une barrière des calculs à $p$ de l'ordre $2^{23}$.

[8] En fait, cela est valable si $\sqrt{n}$ est une puissance de 2, mais le cas général se traite sans difficulté.

Notre accélération est basée sur la remarque clé suivante, empruntée à [85] : les entrées de la matrice de Hasse-Witt étant des coefficients d'une puissance du polynôme $f$, elles vérifient une récurrence d'ordre $2g+1$, à coefficients polynomiaux de degré 1. En utilisant notre résultat sur ce type de récurrence, nous en déduisons un algorithme de complexité linéaire en $\sqrt{p}$, à des facteurs logarithmiques près, pour le calcul de la matrice de Hasse-Witt. L'énoncé exact est donné dans le Théorème 5. Le gain est de l'ordre de $\sqrt{p}$ et se traduit en pratique par la possibilité de traiter un exemple où $p$ est de l'ordre $2^{32}$.

**Théorème 5** *Soit $p \geq 5$ un nombre premier, $d \geq 0$ et $\mathcal{C}$ une courbe hyperelliptique définie sur $\mathbb{F}_{p^d}$ par l'équation $y^2 = f(x)$, avec $f \in \mathbb{F}_{p^d}[X]$ unitaire, sans carrés et de degré $2g+1$. Alors, sous l'hypothèse que $g < p$, on peut calculer la matrice de Hasse-Witt de $\mathcal{C}$ en*

$$O\Big(\big(g^{\omega+1}\sqrt{p} + g^3\,\mathsf{M}(\sqrt{p})\log(p)\big)\,\mathsf{M}\big(dg\log(p)\big)\Big)$$

*opérations binaires et $O\left(dg^3\sqrt{p}\ \log(p)\right)$ espace mémoire.*

De manière pratique, nous démontrons l'intérêt de nos techniques par un exemple de comptage de points pour une courbe de genre 2 aléatoirement choisie sur $\mathbb{F}_{p^3}$, où $p$ est un nombre premier de taille 32 bits. Nous avons implanté notre algorithme dans la librairie NTL [226]. Le cardinal modulo $p$ de la Jacobienne de notre courbe est obtenu en moins de 4 heures, pour 1 Go de mémoire, sur un Athlon AMD, MP 2200+. Le cardinal de la Jacobienne est de l'ordre $2^{192}$ et a été obtenu dans un deuxième temps, en utilisant une variante de l'algorithme de Schoof [88, 90], ainsi que l'algorithme de [166]. Puisque ce cardinal a un facteur premier de taille de l'ordre $2^{158}$, la courbe est considérée *cryptographiquement sûre*. Le traitement d'un exemple de cette taille constitue un nouveau record sur un corps de caractéristique supérieure à 5.

### Améliorations récentes

Pendant la rédaction de cette introduction, nous avons trouvé une nouvelle idée algorithmique, qui permet de faire disparaître complètement la dépendance logarithmique en $n$ dans la complexité de notre algorithme.

Illustrons cette idée sur l'exemple de la suite $u_n$. Pour simplifier, on pose $N = \sqrt{n}$, qu'on suppose être une puissance de 2. Le problème du calcul des valeurs $C(j\sqrt{n})$ intervenant dans l'équation (1.6) se reformule ainsi : calculer les $N$ produits des éléments sur chaque ligne de la matrice

$$\left[\begin{array}{ccc|ccc}
a+1+0 & \cdots & a+\frac{N}{2}+0 & a+\frac{N}{2}+1+0 & \cdots & a+N+0 \\
\vdots & & \vdots & \vdots & & \vdots \\
a+1+\frac{N(N-1)}{2} & \cdots & a+\frac{N}{2}+\frac{N(N-1)}{2} & a+\frac{N}{2}+1+\frac{N(N-1)}{2} & \cdots & a+N+\frac{N(N-1)}{2} \\
\hline
a+1+\frac{N(N+1)}{2} & \cdots & a+\frac{N}{2}+\frac{N(N+1)}{2} & a+\frac{N}{2}+1+\frac{N(N+1)}{2} & \cdots & a+N+\frac{N(N+1)}{2} \\
\vdots & & \vdots & \vdots & & \vdots \\
a+1+N(N-1) & \cdots & a+\frac{N}{2}+N(N-1) & a+\frac{N}{2}+1+N(N-1) & \cdots & a+N+N(N-1)
\end{array}\right]$$

La structure spéciale des entrées de cette matrice permet d'utiliser une approche récursive. En effet, en divisant la matrice en quatre, notre problème définit un sous-problème (gauche, en haut) de taille la moitié. Supposons ce dernier résolu ; alors chacun des produits correspondant aux trois autres sous-matrices s'en déduit en $\mathsf{M}(N/2) + O(N)$, en appliquant notre Proposition 2 pour les valeurs translatées. Le coût $\mathsf{C}(N)$ de l'algorithme vérifie donc la récurrence $\mathsf{C}(N) = \mathsf{C}(N/2) + O(\mathsf{M}(N/2)) + O(N)$, ce qui implique $\mathsf{C}(N) = O(\mathsf{M}(N))$. Par conséquent, la valeur de $u_n$ est obtenue en temps $O(\mathsf{M}(\sqrt{n}))$ et espace mémoire $O(\sqrt{n})$.

La même technique s'étend au cas non-scalaire et résout le problème de réduire le coût du Théorème 4 à $O\big(m^\omega\sqrt{n} + m^2\,\mathsf{M}\big(\sqrt{n}\big)\big)$ opérations, en se débarrassant ainsi de tous les facteurs logarithmiques, tant en temps qu'en mémoire. La vérification en détail, ainsi que l'implantation du nouvel algorithme font l'objet d'un travail en cours. Je mentionne aussi que l'impact de cette nouvelle idée sur l'algorithme de factorisation de Pollard-Strassen fait également l'objet d'un travail en cours avec P. Gaudry et É. Schost.

### 1.3.6 Algorithmes rapides pour les opérateurs différentiels linéaires

> Le dernier chapitre de cette thèse est dédié au cadre différentiel. Nous y proposons une méthode de type évaluation / interpolation pour le calcul sur les opérateurs différentiels linéaires à coefficients polynomiaux, permettant d'aborder le ppcm en même temps que le produit tensoriel. Le rôle des *points* d'évaluation est joué par des séries formelles, sur lesquelles on *évalue* des opérateurs différentiels. L'interpolation est fournie par des *approximants de Padé-Hermite différentiels*. Pour chacun des problèmes, l'étude des singularités apparentes permet d'exhiber des bornes sur la taille des sorties.

Soient $L_1$ et $L_2$ deux opérateurs différentiels linéaires à coefficients polynomiaux. Supposons que l'on veut calculer l'opérateur $L = L_1 \star L_2$, où $\star$ est une *construction d'algèbre linéaire*, définie en termes d'espaces des solutions de $L_1$ et de $L_2$. Des exemples classiques de telles opérations sont fournies par le *plus petit commun multiple à gauche* de $L_1$ et de $L_2$ , ou encore, par leur *produit tensoriel* $L_1 \otimes L_2$.

Le but de ce chapitre est de proposer une stratégie commune de calcul de ce type d'opérations par une méthode *évaluation / interpolation*. Dans cette optique, le rôle des *points d'évaluation* est joué par des séries formelles. L'algorithme *générique* consiste alors en les étapes suivantes :

1. calculer des solutions séries tronquées $S_1$ et $S_2$ de chaque opérateur $L_1$ et $L_2$ ;
2. combiner $S_1$ et $S_2$ à l'aide de l'opération $\star$ pour en déduire une solution tronquée $S$ de l'opérateur $L$ ;
3. retrouver $L$ à partir de sa solution série $S$.

Les étapes (1) et (3) constituent des conversions entre deux types de représentations des opérateurs différentiels. En utilisant les résultats des Chapitres 4 et 5, le passage opérateur-solution admet une solution de complexité quasi-optimale. Par contre, l'étape de reconstruction de l'opérateur est plus délicate. Nous proposons et analysons deux méthodes pour cette

dernière : l'une utilise des approximants de Padé-Hermite, l'autre ramène les calculs à la résolution d'un système linéaire à coefficients fractions rationnelles.

Dans les deux cas, bien que bénéficiant de bornes fines sur les tailles des sorties, les algorithmes *génériques* utilisés n'ont pas un comportement optimal. Ainsi, les algorithmes rapides d'approximation de Padé-Hermite [18, 148, 97] calculent plus d'information que nécessaire, à savoir une *base d'approximants*, ce qui induit une perte d'efficacité dans notre situation structurée. De même, l'algorithme de Storjohann [233] que nous utilisons pour la résolution de systèmes n'arrive pas à tirer profit de la structure particulière des matrices polynomiales considérées.

Cependant, les algorithmes obtenus sont de complexité sous-quadratique en la taille de la sortie. Le théorème suivant résume les principaux résultats auxquels on aboutit.

**Théorème 6** *Soient $L_1$ et $L_2$ deux opérateurs d'ordre $n$, dont les coefficients sont polynomiaux de degré borné par $n$.*

1. *Le produit tensoriel $L = L_1 \otimes L_2$ est un opérateur d'ordre au plus $n^2$, ses coefficients ont des degrés bornés par $2\,n^4$. De plus, $L$ peut être calculé à partir de $L_1$ et de $L_2$ en $O_{\log}(n^{2\omega+3})$ opérations.*
2. *Le ppcm à gauche de $L_1$ et de $L_2$ est un opérateur d'ordre au plus $2n$, ses coefficients ont des degrés bornés par $3\,n^2$. De plus, il peut être calculé à partir de $L_1$ et de $L_2$ en $O_{\log}(n^{\omega+2})$ opérations.*

# Organisation du document

La suite de ce mémoire est organisée en trois parties. La Partie II traite des algorithmes fondamentaux du calcul formel. Elle contient une introduction aux algorithmes rapides utilisés dans la suite, ainsi que trois chapitres contenant des contributions à l'algorithmique de base des polynômes et des séries à une variable. La Partie III comporte deux chapitres traitant des algorithmes d'élimination commutative en deux et respectivement plusieurs variables. Enfin, la Partie IV contient des résultats concernant l'algorithmique des opérateurs différentiels et aux différences.

# Part II

# Fundamental algorithms

# Chapter 2

# Multiplication of polynomials, matrices and differential operators

This chapter gives a quick introductory overview of the basic algorithms that will be constantly used and referred to in the rest of this thesis. The adopted style is a narrative one; precise statements are sometimes replaced by intuitive arguments, illustrative examples and bibliographical references.

In Section 2.1, we briefly discuss various algorithms polynomial multiplication. Then, we focus in Section 2.2 on algorithms for matrix multiplication and on their impact in the solution of other problems related to linear algebra. We end this chapter by describing a recent algorithm for multiplying linear differential operators. Our contribution is to complete a result of [247], concerning the *computational equivalence* between the problem of operator multiplication and that of matrix multiplication.

## Contents

## 2.1 Multiplication of univariate polynomials and power series

### 2.1.1 Karatsuba's multiplication

Suppose that $P$ and $Q$ are two polynomials of degree $n$, to be multiplied. Karatsuba [129] proposed the following divide-and-conquer algorithm, computing the product $PQ$ in $O(n^{\log(3)})$ base ring operations. It consists in splitting each of the two polynomials into two polynomials of degree $k \approx n/2$:

$$P(T) = P_0(T) + T^k P_1(T), \quad Q(T) = Q_0(T) + T^k Q_1(T).$$

Then, the product $PQ$ is obtained using only 3 products of polynomials of degree $k$, due to the identities:

$$\begin{array}{rcl} P(T)Q(T) & = & P_0(T)Q_0(T) + T^k R(T) + T^{2k} P_1(T)Q_1(T) \\ R & = & (P_0 + P_1)(Q_0 + Q_1) - P_0Q_0 - P_1Q_1. \end{array}$$

Applying recursively this method, the running time $\mathsf{K}(n)$ satisfies the recurrence

$$\mathsf{K}(n) = 3\mathsf{K}(n/2) + O(n), \quad \mathsf{K}(1) = 0,$$

so that $\mathsf{K}(n) = O(n^{\log(3)})$.

#### Additional comments

We remark that this algorithm uses more additions than the classical one and this explains why, for small degrees, the latter is faster. Actually, the crossover between the two methods depends a lot on the implementation. Let us mention that Karatsuba's idea may be generalized, if one chooses to cut the input polynomials into $r$ parts of equal degree. In that case, the running time $T_r(n)$ is shown to verify the recurrence $T_r(rn) = (2r-1)T_r(n) + O(n)$, so that

$$T_r(n) = O\big(n^{\log_r(2r-1)}\big).$$

Thus, for increasing $r$, the resulting algorithm is asymptotically faster (with respect to $n$), but a serious problem lies in the growing of the constant hidden behind $O$, which depends on $r$ and cannot be uniformly bounded. From a theoretical point of view, this difficulty can be overcome by allowing $r$ to vary with $n$, meaning that in the cutting process one should do recursive calls to several Karatsuba-like algorithms (for various $r$). Still, for practical purposes, this idea seems quite unrealistic.

### 2.1.2 Fast Fourier Transform

A faster algorithm for polynomial multiplication is the Fast Fourier Transform, which is based on the important idea of change of representation: instead of the list of its coefficients, one may alternatively represent a polynomial of degree less than $n$ by its values on $n$ distinct points. We call the latter the *point-value representation*. The key feature is that in this representation, multiplication is linear in the size of the output. To put this idea at work, it suffices to exhibit a set of points with the property that the change of representation (monomial to point-value, this is *multipoint evaluation*, and its reciprocal, that is *interpolation*) is not too costly.
Suppose for a moment that the coefficient ring is $R = \mathbb{C}$ and that $n$ is a power of 2. We let the set of evaluation points to be the set $\Omega_n$ of the $n$th roots of unity; this choice is motivated by the following fact.

**Lemma 1** *If $n \geq 2$ is even, then the squares of the $n$ roots of unity in $\Omega_n$ are exactly the $\frac{n}{2}$ elements of $\Omega_{\frac{n}{2}}$.*

Based on this lemma, one can design a fast multipoint evaluation algorithm of a polynomial $A(x)$ of degree less than $n$ on the points of $\Omega_n$ using a divide-and-conquer strategy. Suppose for simplicity that $n$ is a power of 2.

We divide $A(x) = A_1(x^2) + xA_2(x^2)$, where $A_1$ and $A_2$ are polynomials of degree less than $\frac{n}{2}$, so that evaluating $A$ on the points in $\Omega_n$ amounts to solving two subproblems of the same form but of half size, plus some additional operations whose number is linear in $n$. If $\mathrm{DFT}(n)$ denotes the number of operations in $R$ to solve the problem, this yields the recurrence $\mathrm{DFT}(n) = 2\mathrm{DFT}(\frac{n}{2}) + \frac{3}{2}n$, so

$$\mathrm{DFT}(n) = \frac{3}{2} n \log(n).$$

The inverse problem, interpolating a polynomial at the points of $\Omega_n$ can be solved using the same number of operations in $R$, since it amounts to a multipoint evaluation. This can be seen using the following matrix interpretation. Writing $\omega$ for $e^{2\pi i/n}$, then $\Omega_n = \{1, \omega, \ldots, \omega^{n-1}\}$ and the values $A(\omega^i)$ are simply the entries of the matrix-vector product between the Vandermonde matrix

$$V_\omega = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}$$

and the column vector whose entries are the coefficients of $A$. Therefore, interpolation interprets as multiplying the inverse matrix $V_\omega^{-1}$ by a vector; moreover, it is easy to verify that $V_\omega^{-1}$ is the matrix $\frac{1}{n} V_{\omega^{-1}}$, so interpolation amounts indeed to a multipoint evaluation.
Thus, the total time to transform two polynomials in coefficient form into point-value form, multiply them in this representation, and finally interpolate the result back into coefficient form gives an algorithm for multiplying them in time $O(n \log(n))$, where $n$ is the degree of the product polynomial.

**Additional comments**

The Fast Fourier Transform has a long history and is one of the most celebrated algorithmic breakthroughs of the 20th century. J. Dongarra and F. Sullivan [72] placed it among the 10 algorithms with the greatest influence on the development and practice of software and engineering in the 20th century [1]. Its underlying idea goes back to Gauss (circa 1805), who needed fast ways to calculate orbits of asteroids. Since then, it was rediscovered several times, see [66, 67, 21] for more details. Schönhage & Strassen [214] showed that the restriction on the existence of roots of unity in the base ring $R$ can be dropped by introducing some *virtual* roots of unity, for an overhead time cost of $\log\log(n)$. Cantor and Kaltofen [50] generalized this result and derived an algorithm for multiplying degree $n$ polynomials with coefficients from an arbitrary (possibly non commutative) algebra using $O(n\log(n))$ algebra multiplications and $O(n\log(n)\log\log(n))$ algebra additions/subtractions. This method may be used for multiplying polynomial matrices.

### 2.1.3 Practicality issues of fast algorithms

Until recently, general-purpose computer algebra systems only implemented the classical method, and sometimes Karatsuba's. This is quite sufficient as long as one deals with polynomials of fairly small degree, say of order of 100, but for real-life purposes, fast algorithms become crucial.
Asymptotically fast algorithms are standard tools in many areas of computer science, for instance sorting algorithms like *quicksort* and *mergesort* are widely used in practice and offer satisfaction over classical naive algorithms for small sizes already. In contrast, fast algorithms for polynomial multiplication have received little attention in the computer algebra domain since their invention in the $70s$. The main reason is the unfortunate belief concerning the unrealistic crossover points starting from which these methods would become practically faster in practice. Nowadays, general purpose computer algebra systems like Magma [32] offer convincing implementations of these fast algorithms.

### 2.1.4 The function $\mathsf{M}$

Throughout this thesis, we present algorithms for various problems based on (fast) polynomial multiplication. In order to abstract from the underlying multiplication algorithm in our cost analyzes, we introduce the following classical notation, see for instance [255, Section 8.3].

**Definition 1** *Let $R$ be a ring. We call a function $\mathsf{M} : \mathbb{N} \to \mathbb{R}$ a* **multiplication time for** *$R[T]$ if polynomials in $R[T]$ of degree less than $n$ can be multiplied using at most $\mathsf{M}(n)$ operations in $R$. We assume that $\mathsf{M}(1) = 1$.*

The classical (naive) method gives $\mathsf{M}(n) = O(n^2)$, but methods which are faster for large $n$ are known. Asymptotically, the fastest method is that of Schönhage & Strassen [214], which

[1] The other nine are: the Metropolis Algorithm, the Simplex Method, the Krylov Subspace Iteration Methods, the Decompositional Approach to Matrix Computations, Fortran Optimizing Compiler, the QR Algorithm, the Quicksort Algorithm, the Integer Relation Detection Algorithm and the Fast Multipole Method.

gives

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

In order to ensure that our complexity results do not depend on the algorithm used for multiplication, we assume that the function $\mathsf{M}$ satisfies the following inequality:

$$\frac{\mathsf{M}(n)}{n} \geq \frac{\mathsf{M}(m)}{m}, \text{ for all } n \geq m \geq 1. \tag{2.1}$$

**Lemma 2** *The multiplication time function* $\mathsf{M}$ *enjoys the following properties:*

1. $\mathsf{M}$ *is super-linear, i.e.,*

$$\mathsf{M}(mn) \geq n\mathsf{M}(m),\ \mathsf{M}(m+n) \geq \mathsf{M}(m) + \mathsf{M}(n) \text{ and } \mathsf{M}(n) \geq n,$$

   *for all positive integers* $m$ *and* $n$*.*

2. *If* $n$ *is a power of* 2*, then*

$$\begin{aligned} \mathsf{M}(n/2) + \mathsf{M}(n/4) + \cdots + \mathsf{M}(1) &< \mathsf{M}(n), \text{ and} \\ 2\,\mathsf{M}(n/2) + 4\,\mathsf{M}(n/4) + \cdots + n\,\mathsf{M}(1) &< \mathsf{M}(n)\log(n). \end{aligned}$$

We summarize in the table below various polynomial multiplication algorithms and their running times

| Algorithm | $\mathsf{M}(n)$ |
|---|---|
| classical | $2n^2$ |
| Karatsuba | $O(n^{\log(3)}) \subset O(n^{1.585})$ |
| FFT | $O(n \log(n) \log\log(n))$ |

Table 2.1: Polynomial multiplication algorithms

## 2.2 Matrix multiplication

Consider the problem of multiplying two $n \times n$ matrices. The standard algorithm requires $n^3$ multiplications and $n^3 - n^2$ additions. At first, it may seem hopeless to attempt to reduce

dramatically the number of arithmetic operations involved. Strassen [234] was the first to infirm this belief; he observed that a pair of 2 by 2 matrices can be multiplied in 7 (rather than 8) multiplications by the following surprising identities (see [41] for a nice pictorial interpretation). If the matrices to be multiplied are

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad N = \begin{bmatrix} p & q \\ r & s \end{bmatrix}.$$

then computing

$$\begin{aligned} z_1 &= (a+d)(p+s) \\ z_2 &= (c+d)p \\ z_3 &= a(q-s) \\ z_4 &= d(-p+r) \\ z_5 &= (a+b)s \\ z_6 &= (-a+c)(p+q) \\ z_7 &= (b-d)(r+s) \end{aligned}$$

yields their product as

$$MN = \begin{bmatrix} z_1 + z_4 - z_5 + z_7 & z_3 + z_5 \\ z_2 + z_4 & z_1 - z_2 + z_3 + z_4 \end{bmatrix}.$$

The key point is that the algorithm does not make use of the commutativity of multiplication. This implies that the elements $a, b, c, d, p, q, r, s$ can themselves be matrices, and this yields a divide-and-conquer algorithm for matrix multiplication, which multiplies matrices of size $2^{k+1}$ in 7 multiplication of $2^k \times 2^k$ matrices and a constant number (18) of matrix additions. The number $\mathsf{MM}(n)$ of multiplications used to multiply two $n \times n$ matrices thus verifies the recursion

$$\mathsf{MM}(2^k) = 7\mathsf{MM}(2^{k-1}) + 18 \cdot 2^{2k-2}.$$

By padding with zeros (which at most doubles the sizes), this yields

$$\mathsf{MM}(n) = O(n^{\log(7)}) = O(n^{2.807}).$$

### Remarks

Prior to Strassen, Winograd [261] showed that for even $n$, any two $n \times n$ matrices can be multiplied using $(n-2)n^2/2$ multiplications; his method is based on the identity:

$$\sum_{j=1}^{\lfloor n/2 \rfloor} m_{ij} n_{jk} = \sum_{j=1}^{\lfloor n/2 \rfloor} (m_{i,2j-1} + n_{2j,k})(m_{i,2j} + n_{2j-1,k}) - \sum_{j=1}^{\lfloor n/2 \rfloor} m_{i,2j-1} m_{i,2j} - \sum_{j=1}^{\lfloor n/2 \rfloor} n_{2j-1,k} n_{2j,k}.$$

If this algorithm could be used recursively, then $\phi(n) = \log_n(n^3/2 + n^2 - n/2)$ would be such that $\mathsf{MM}(n) = O(\phi(n))$; the minimum of $\phi$ being reached for $n = 5$, this would lead to a $O(n^{2.76})$ matrix multiplication algorithm. However, Winograd's scheme requires the commutativity of the entries of the multiplicand matrices, so straightforward recursion is not possible. After Strassen's discovery, Winograd suggested the following improved algorithm, which also uses 7 non-commutative

multiplications, but only 15 $\pm$ operations. It gives the product $MN$ as

$$MN = \begin{bmatrix} ap + br & (w+v) + \Big(b - \big((c+d) - a\big)\Big)s \\ (w+u) - d\Big(\big(s - (q-p)\big) - r\Big) & (w+u) + v \end{bmatrix},$$

where

$$\begin{aligned} u &= (c-a)(q-s) \\ v &= (c+d)(q-p) \\ w &= \big((c+d)-a\big)\big(s-(q-p)\big) + ap. \end{aligned}$$

By judiciously performing padding with zeros, Strassen [234] showed that with his method $\mathsf{MM}(n) \leq 4.7n^{\log(7)}$. The same analysis yields $\mathsf{MM}(n) \leq 4.54n^{\log(7)}$ for Winograd's method [31]. Fischer [84] showed that a careful application of Winograd's form produces a $3.92\, n^{\log(7)}$ algorithm.

Hopcroft and Kerr [120] showed that 7 multiplications are necessary to multiply $2 \times 2$ matrices over a non-commutative ring and Winograd [262] showed that this number is minimal even when the commutative law is used. Probert [197] has shown that 15 additions are also necessary for any 7 multiplications algorithm (so the result of Winograd is optimal) and deduced that faster matrix multiplication must come from studying larger systems.

Indeed, from the analysis of Strassen's algorithm it is clear that given an algorithm that multiplies $k \times k$ matrices over a non-commutative ring using $m$ multiplications, then we have a $O(n^{\log_k(m)})$ algorithm for multiplying $n \times n$ matrices. For instance, Pan [182] showed that $68 \times 68$ matrices can be multiplied using 132464 multiplications, instead of $68^3 = 314432$, whence an $O(n^{\log_{68}(132464)}) = O(n^{2.795})$ algorithm for matrix multiplication.

**Matrix multiplication exponent.** For a field $k$, a real number $\Omega$ is called a *feasible matrix multiplication exponent* if any two matrices over $k$ can be multiplied within $O(n^\Omega)$ operations in $k$. The infimum of all these $\omega$ (for fixed $k$) is called the *matrix multiplication exponent* over $k$. Obviously, $2 \leq \omega \leq 3$. Strassen's algorithm shows that $\omega < \log(7)$. This result motivated the development of the theory of *bilinear complexity*, a subfield of algebraic complexity theory and produced an avalanche of results concerning upper estimates for $\omega$ by Pan, Bini, Capovani, Lotti, Romani, Schönhage, Coppersmith, Winograd,..., see [47, Chapter 15] for a comprehensive treatment and detailed historical notes.

On the theoretical side, Strassen [211] showed that $\omega$ is invariant under field extensions, thus the same for all fields of a fixed characteristic. From a practical point of view, all the exponents discovered so far work for all fields. The *world record* is $\omega < 2.376$ and it comes from Coppersmith and Winograd [69]. For the time being, their result is only of theoretical interest. It was conjectured that $\omega = 2$, but this is a difficult problem. A first step in this direction was done by Coppersmith [68], who proved some meta-theorems showing (roughly) that if one has an algorithm that runs in $O(n^\alpha)$, then one can find an algorithm that runs in $O(n^{\alpha-\epsilon})$. Note that here $\epsilon$ is a function, not a constant, so it does not yield the conjecture that an $O(n^2)$ algorithm exists for matrix multiplication.

## 2.3 Problems related to matrix multiplication

Further computational problems in linear algebra include matrix inversion [234, 209, 46], determinants [234, 46], minimal and characteristic polynomials [132], $LUP$-decompositions [46], transformation to echelon form [132], solving linear systems [234], reductions to various canonical forms [95, 232], ... It turns out that most of these problems can be solved using a constant number of matrix multiplications. Moreover, some of them are known to be computationally equivalent, see [47, Chapter 16]. An important open question is to know whether linear solving is as hard as matrix multiplication.

In what follows, we briefly discuss some of these problems.

### 2.3.1 Matrix inversion

Strassen [234] suggested a way to reduce inversion to (fast) matrix multiplication. Bunch and Hopcroft [46] described the method as a block factorization. Indeed, the 2 by 2 matrix $A = (a_{ij})$ factorizes

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{21}a_{11}^{-1} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & 0 \\ 0 & a_{22} - a_{21}a_{11}^{-1}a_{12} \end{bmatrix} \begin{bmatrix} 1 & a_{11}^{-1}a_{12} \\ 0 & 1 \end{bmatrix},$$

so its inverse is given by

$$A^{-1} = \begin{bmatrix} 1 & -a_{11}^{-1}a_{12} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11}^{-1} & 0 \\ 0 & \left(a_{22} - a_{21}a_{11}^{-1}a_{12}\right)^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -a_{21}a_{11}^{-1} & 1 \end{bmatrix}.$$

This method uses two inversions, six multiplications and two additions. Since it does not use the commutativity of multiplication, it can be applied recursively. Assuming that only invertible submatrices $a_{11}$ and $a_{22} - a_{21}a_{11}^{-1}a_{12}$ are encountered, its running time $\mathsf{I}(n)$ satisfies

$$\mathsf{I}(n) = 2\,\mathsf{I}(n/2) + 6\,\mathsf{MM}(n/2) + O(n^2).$$

Supposing $\mathsf{MM}(n) = O(n^\omega)$ (for an $\omega > 2$), this yields $\mathsf{I}(n) = O(\mathsf{MM}(n))$.

#### Remarks

Bunch and Hopcroft [46] showed how to treat the case where some intermediate submatrices are singular, by performing an appropriate permutation on the input matrix. Munro [177] remarked that matrix multiplication can be reduced to matrix inversion, using

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}.$$

This shows that the both problems are computationally equivalent.

### 2.3.2 Determinants

Based on the identity $\det(A) = \det(a_{11})\det(a_{22} - a_{21}a_{11}^{-1}a_{12})$, the determinant computation also has running time proportional to that of matrix multiplication.

### 2.3.3 Characteristic polynomials

The previous method has no straightforward fast application for computing the characteristic polynomial $\det(xI - A)$ of $A$. A possible solution comes from another direction: there is a class of matrices for which computing characteristic polynomials is immediate: given $a_0, a_1, \ldots, a_{n-1} \in k$, the matrix

$$\begin{bmatrix} 0 & & & -a_0 \\ 1 & \ddots & & \vdots \\ & \ddots & 0 & -a_{n-2} \\ & & 1 & -a_{n-1} \end{bmatrix}$$

has characteristic polynomial

$$P(x) \;=\; x^n + \sum_{j=0}^{n-1} a_j x^j.$$

Such a matrix is called of *Frobenius* or *companion* type.

Keller-Gehrig's algorithm [132] combines this with the following simple observation. If we find an invertible matrix $T$ such that $F = T^{-1}AT$ has Frobenius form, then the characteristic polynomial of $A$ and of $F$ coincide and can be computed for one inversion and two multiplications of $n \times n$ matrices. The same argument can be adapted if $F$ is only Frobenius by diagonal blocks.

Assume for simplicity that $n$ is a power of 2 and that the matrix $A$ is *generic enough* (the general case can also be handled by essentially the same method). In that case, the key point is that for any non-zero column vector $v \in k^n$ the matrix

$$T \;=\; \left[\; v \mid Av \mid \ldots \mid A^{n-1}v \;\right]$$

is invertible. This way, efficient computation of characteristic polynomials is reduced to the problem of computing $T$. The obvious solution requires $n$ matrix-vectors products, thus has complexity $O(n^3)$. The following divide-and-conquer solution has been designed in [132]: we first compute $A^2, A^4, \cdots, A^{n/2}$ for $\log(n)$ multiplications of matrices, then perform the following $\log(n)$ matrix products

$$\begin{array}{lcll} \big[Av\big] & = & & \big[Av\big] \\ \big[A^3v \mid A^2v\big] & = & A^2 & \big[Av \mid v\big] \\ & \vdots & & \\ \big[A^{n-1}\, v \mid \cdots \mid A^{\frac{n}{2}}v\big] & = & A^{\frac{n}{2}} & \big[A^{\frac{n}{2}-1}v \mid \cdots \mid v\big]. \end{array}$$

The total number of operations used by this algorithm is thus $O\big(\mathsf{MM}(n)\log(n)\big)$. We refer to Subsection 3.4.2 for the description of a different algorithm of similar complexity.

### 2.3.4 Powers of matrices

Powers of matrices are important, for instance, in computing with linear (systems of) recurrent sequences with constant coefficients, see Section 3.3.3 of this thesis. Computing the $N$th power of a $n \times n$ matrix $A$ can be done by binary powering in $O(\log(N)\ \mathsf{MM}(n))$ operations. For large $N$ a further improvement is possible [95]. The idea is that if $A$ writes as $T^{-1}FT$, then $A^N = T^{-1}F^NT$. Computing as above such a decomposition with $F$ of Frobenius type reduces the problem to fast powering of Frobenius matrices.

**Powers of Frobenius matrices.** Since the $k$-linear map $k[x]/(P) \xrightarrow{\cdot x} k[x]/(P)$ has matrix $F$ in the canonical basis $\{1, x, \ldots, x^{n-1}\}$ of $k[x]/(P)$, we have that for all column vector $v = [v_0, \cdots, v_{n-1}]$, the coordinates of the vector $F^N \cdot v$ are the coefficients of the polynomial $F^N \cdot (v_0 + \cdots + v_{n-1}x^{n-1}) \mod P$. As a consequence

$$F^N = \Big[x^N \mod P \ \Big|\ x^{N+1} \mod P \ \Big|\ \cdots \ \Big|\ x^{N+n-1} \mod P\Big]$$

can be computed for $\log(N) + n$ multiplications of polynomials modulo $P$, thus using

$$O\Big(\big(n + \log(N)\big)\mathsf{M}(n)\Big)$$

operations.

### 2.3.5 Evaluation of polynomials on matrices

A more general operation is the evaluation of a (monic) polynomial of degree $N$ on a $n \times n$ matrix $A$. The obvious Horner-like method requires $N$ multiplications of matrices, thus has complexity $O(N\ \mathsf{MM}(n))$.

The *baby step / giant step* algorithm of Paterson and Stockmeyer [190] solves the same problem with complexity $O(\sqrt{N}\ \mathsf{MM}(n))$, see the Introduction of this thesis.

If $N$ is large compared to $n$, this method can be improved by first computing the characteristic polynomial $\chi_A$ of $A$ as shown before, then to perform an Euclidean division of $P$ by $\chi_A$ using $O(N\,\mathsf{M}(n)/n)$ operations. Since, by Hamilton-Cayley's theorem $\chi_A(A) = 0$, this reduces the problem to the evaluation of a polynomial of degree at most $n$ at $A$, which can be solved using $O(\sqrt{n}\,\mathsf{MM}(n))$ by Paterson and Stockmeyer's algorithm. The total cost is thus

$$O\Big(\sqrt{n}\ \mathsf{MM}(n) + \frac{N}{n}\mathsf{M}(n)\Big).$$

#### Remark

Giesbrecht [95] also showed that evaluating a non-linear polynomial on a matrix is actually as hard as matrix multiplication.

## 2.4 Multiplication of linear differential operators

Consider the problem of computing the product of two linear differential operators of order $n$ in $\delta = x\frac{d}{dx}$ with polynomial coefficients in $x$ of degree $n$.

The naïve algorithm has time complexity $O(n^2\,\mathsf{M}(n))$. In [248], van der Hoeven proposed a faster algorithm; more exactly, he showed that this problem can be reduced to the problem of multiplying a fixed number of $n \times n$ matrices.

In this paragraph, we show that the two problems are actually equivalent. This was already asserted in the final remarks of van der Hoeven's article, but not proved. We begin by briefly recalling the algorithm in [248]. Its basic idea is to adapt the polynomial FFT's evaluation-interpolation strategy to the non-commutative case.

More precisely, let $\mathcal{P} = \sum_{i=0}^{n} p_i(x)\delta^i$ be a linear differential operator of order $n$ in $\delta$, with polynomial coefficients $p_i(x) = \sum_{j=0}^{n} p_{ij}x^j$. As in the polynomial case, there is an alternative way to represent the operator $\mathcal{P}$: instead of giving its list of coefficients $p_i(x)$, one may give the list of the evaluations of $\mathcal{P}$ at the polynomials $x^j$, for $j = 0, \ldots, n$. Let us first see that conversions from one representation to another can be done fast.

For any integer $s \geq 0$, we have the formula:

$$\mathcal{P} \cdot x^s = \big(p_{00} + p_{10}s + \cdots + p_{n0}s^n\big)x^s + \cdots + \big(p_{0n} + p_{1n}s + \cdots + p_{nn}s^n\big)x^{n+s}. \tag{2.2}$$

Thus computing the polynomial $\mathcal{P} \cdot x^s$ amounts to evaluating the polynomials

$$\widetilde{p_j}(x) = \sum_{i=0}^{n} p_{ij}x^i$$

at the points $s = 0, 1, \ldots, n$. Using fast evaluation algorithms, this can be done using $O\big(\mathsf{M}(n)\log(n)\big)$ base field operations for each polynomial, see Section 1.3.1.

Formula (2.2) also shows that given $n+1$ polynomials $q_i \in K[x]$ of degree $n$, one can compute within the same running time bound the unique linear differential operator $\mathcal{P}$ of order $n$ in $\delta$, with polynomial coefficients of degree at most $n$, such that

$$\frac{\mathcal{P} \cdot x^i}{x^i} = q_i(x), \quad \text{for all } \; 0 \leq i \leq n.$$

This suggests the following algorithm for multiplying two operators $\mathcal{P}$ and $\mathcal{Q}$ in $K[x][\delta]$: evaluate $\mathcal{Q}$ on the powers $x^j$, for $0 \leq j \leq 2n$, then evaluate $\mathcal{P}$ on the polynomials $A_j := \mathcal{Q} \cdot x^j$ and finally interpolate the product operator $\mathcal{P} \circ \mathcal{Q}$.

By the discussion above, the first and the third steps have cost $O(n\mathsf{M}(n)\log(n))$, thus nearly optimal in the size $n^2$ of the output. We now look at the second step.

If $A(x) = \sum_{k=0}^{m} a_k x^k$ is a polynomial in $K[x]$, then one can check that the coefficients of the polynomial $\mathcal{P} \cdot A$ are the entries of the following matrix-vector product

$$\begin{bmatrix} \widetilde{p_0}(0) & 0 & \dots & 0 & \dots & 0 \\ \widetilde{p_1}(0) & \widetilde{p_0}(1) & & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ \widetilde{p_n}(0) & \dots & \dots & \widetilde{p_0}(n) & & \vdots \\ 0 & \widetilde{p_n}(1) & \dots & \widetilde{p_1}(n) & \ddots & 0 \\ \vdots & 0 & \ddots & & \ddots & \widetilde{p_0}(m) \\ \vdots & \vdots & \ddots & \widetilde{p_n}(n) & \dots & \widetilde{p_1}(m) \\ \vdots & \vdots & & 0 & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \widetilde{p_n}(m) \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ \vdots \\ \vdots \\ a_m \end{bmatrix}. \tag{2.3}$$

This observation generalizes the well-known fact in the polynomial case that multiplication by a polynomial translates into a Toeplitz matrix-vector product. For convenience, we will denote the $(m+n+1) \times (m+1)$ left-hand matrix in Equation (2.3) by $\mathcal{M}_{\mathcal{P},m}$.

The key remark in [248] is that one can incorporate the evaluation of $\mathcal{P}$ on several polynomials $A_j$ into a single matrix multiplication between $\mathcal{M}_{\mathcal{P}}$ and the matrix whose $j$th column contains the coefficients of $A_j$. Indeed, the equality $\mathcal{M}_{\mathcal{P}\circ\mathcal{Q}} = \mathcal{M}_{\mathcal{P}}\mathcal{M}_{\mathcal{Q}}$ is easy to infer. This way, the second step of the previous algorithm uses $O(\mathsf{MM}(n))$ operations and dominates its whole cost.

Thus, we have shown that multiplying two linear differential operators of degree $n$ in $\delta$ with polynomial coefficients of degree $n$ can be done using a finite number of products of $n \times n$ matrices with entries in $K$. We now show that the converse is also true, that is that the two problems are computationally equivalent.

Suppose that we have to multiply two matrices $A$ and $B$. Write $B = U + V + D$, where $U$ is upper triangular, $V$ is lower triangular and $D$ is diagonal. Then computing the product $AB$ can be reduced to the products $AU, AV$ and $AD$. Since $AD$ is done for $O(n^2)$ base field operations, it is enough to prove that computing products of type $AU$ and $AV$ amounts to a (fixed) finite number of products of linear differential operators of order $n$ and degree $n$. Since $(AV)^t = V^t A^t$ and $V^t$ is upper triangular, it is sufficient to check the following assertion: if $A$ and $U$ are $[n/2] \times [n/2]$ matrices such that $U$ is upper triangular, than the computation of their products $AU$ and $UA$ amounts to the product of two differential operators of order and degree at most $n$.

We can compute the (unique) differential operator $\mathcal{P}$ such that its associated $(2n+1)\times(n+1)$ matrix $\mathcal{M}_{\mathcal{P}}$ has the form

| | |
|---|---|
| 0 | 0 |
| $A$ | 0 |
| 0 | 0 |
| 0 | 0 |

using $O(n\,\mathsf{M}(n)\log(n))$ base field operations.

Similarly, the (unique) differential operator $\mathcal{Q}$ whose associated $(n+1) \times \lfloor n/2 \rfloor$ matrix is

$$\mathcal{M}_{\mathcal{Q}} = \begin{array}{|c|}\hline U \\ \hline 0 \\ \hline \end{array}$$

can be computed for $O(n\,\mathsf{M}(n)\log(n))$ base field operations. Then, the matrix $\mathcal{M}_{\mathcal{P}\circ\mathcal{Q}}$ associated to the product $\mathcal{P} \circ \mathcal{Q}$ can be also computed within $O(n\,\mathsf{M}(n)\log(n))$ base field operations. By Equation (2.3), it equals

$$\mathcal{M}_{\mathcal{P}\circ\mathcal{Q}} = \begin{array}{|c|}\hline 0 \\ \hline AU \\ \hline 0 \\ \hline 0 \\ \hline \end{array}$$

This proves our claim concerning the product $AU$. The proof for $UA$ is entirely similar.

For completeness, we summarize the result in the following theorem

**Theorem 1** *Let $k$ be a field and let $n$ be a positive integer. Then:*

1. *the multiplication of two $n \times n$ matrices with entries in $k$ reduces to a fixed finite number of products of linear differential operators of order $n$ in $\delta$ and polynomial coefficients in $k[x]$ of degree at most $n$.*

2. *the product of two linear differential operators of order $n$ in $\delta$ and polynomial coefficients in $k[x]$ of degree at most $n$ reduces to a fixed finite number of multiplications of $n \times n$ matrices with entries in $k$.*

# Chapter 3

# Newton iteration

In numerical analysis, Newton's tangent method is used to approximate roots of differentiable functions. Adapted to an exact framework, it provides a major tool of computer algebra. In this chapter, we outline the method and exemplify its use in the derivation of fast algorithms for univariate power series and rational fractions. These basic algorithms are used all along this thesis. We conclude the chapter by describing Storjohann's application of Newton's iteration to rational system solving, which is used in the last part of the thesis. The algorithms discussed in this chapter are not new, our contribution is to give a unified presentation.

## Contents

## 3.1 Newton's algebraic iteration: generic algorithm

Given a power series at precision $n$, the computation of the first $n$ coefficients of its inverse, logarithm, exponential and powers, and more generally, the power series solutions of large families of differential equations, can be computed for $O(\mathsf{M}(n))$ ring operations. (Recall that $\mathsf{M}(n)$ denotes the computing cost in terms of coefficient arithmetic operations for multiplying two polynomials of degree $n$).

The efficiency of these computations is based on a symbolic variant of Newton's tangent method [179]. Let $F$ be a power series in $x$, whose coefficients are power series in $t$ over a field $k$. To compute a power series solution $x = x(t)$ of an equation $F(t, x) = 0$, Newton's iteration writes

$$x_{k+1}(t) = x_k(t) - \frac{F(t, x_k(t))}{\frac{\partial F}{\partial x}(t, x_k(t))} \mod t^{2^{k+1}}.$$

Here the notation mod $t^N$ means that terms of order higher than $N$ are not computed.

The iteration starts with $x_0 = \alpha$, where $\alpha$ is required to be a simple root of $F(0, x)$ in the base field $k$. The number of correct coefficients doubles with each successive iteration, see [47, Th. 2.31] for a proof. One says that Newton's iteration method has a quadratic rate of convergence. From the efficiency point of view, an important feature is that the arithmetic cost of Newton's method is usually dominated by that of the last iteration. For instance, if the iterations involve multiplications of polynomials or power series, this is explained by the equality

$$\mathsf{M}(n/2) + \mathsf{M}(n/4) + \cdots = O(\mathsf{M}(n)).$$

## 3.2 Application to operations on power series

### 3.2.1 Inversion and division

The naïve method for inverting or dividing power series at precision $n$ is the undetermined coefficients method, which has cost $O(n^2)$. We show that using Newtons's iteration, this cost drops to $O(\mathsf{M}(n))$. This result was obtained by Brent in [42].

Indeed, let $A(t)$ be a power series with non-zero constant coefficient. To compute the first $n = 2^r$ coefficients of its inverse, we apply Newton's iteration to the function $F(t, x) = A(t) - 1/x$. This yields the following recurrence:

$$x_{k+1} = x_k - \frac{A - 1/x_k}{1/x_k^2} = x_k + x_k(1 - Ax_k) \mod t^{2^{k+1}},$$

with initial condition $x_0 = 1/A(0)$.

The cost of the $k$th iteration is $\mathsf{M}(2^k) + \mathsf{M}(2^{k+1}) + 2^k$. Summing up, this entails a total number of operations of at most

$$\sum_{k=0}^{r-1} \left(\frac{3}{2}\mathsf{M}(2^{k+1}) + 2^k\right) \leq 3\,\mathsf{M}(2^r) + 2^r = 3\,\mathsf{M}(n) + n.$$

In $n$ is not a power of two, one can apply the previous algorithm to the first power of 2 larger than $n$; the complexity is still in $O(\mathsf{M}(n))$. However, in unbalanced cases, too many useless coefficients may be computed by this method; using a slight modification of Newton's iteration, a better algorithm can be written down [255] and has the same complexity

$$\mathsf{I}(n) \ \leq \ 3\,\mathsf{M}(n) + O(n).$$

### Further remarks

Recently, Hanrot, Quercia and Zimmermann [115] have been able to reduce the cost of power series inversion by Newton's iteration. Their improvement is based on the remark that while performing the iteration $x_{k+1} = x_k + x_k(1 - Ax_k)$, only the *middle part* of the product between $A$ and $x_k$ is needed, since the lower part of this product is known in advance to reduce to 1. This is the prototypical example of the *middle product* operation, which consists in extracting the coefficients of degree $n$ up to $2n$ of a product of polynomials of degree $n$ and $2n$. The cost of this operation is shown to be $\mathsf{M}(n) + O(n)$ instead of 2 $\mathsf{M}(n)$ and in the case of power series inversion, this reduces the upper bound to $\mathsf{I}(n) \leq 2\,\mathsf{M}(n) + O(n)$. In [35], we showed that the middle product is the *transposed operation* of the usual polynomial product; this fact, in conjunction with *Tellegen's transposition principle* explains the result in [115] and places it in a more general context, see Chapter 4 of this thesis. Note finally that depending on the multiplication model, the constant factor of 2 can be further improved: in Karatsuba's model [115], one inversion has roughly the same cost as one multiplication, while in the FFT case, Bernstein [22] does an inversion for the price of 3/2 multiplications.

Since division of power series can be reduced to one inversion and one multiplication, it has cost $O(\mathsf{M}(n))$ too, but it is legitimate to think that in terms of constant factors, one can do it better. A natural idea, due to Karp and Markstein [130] is to incorporate the dividend into the last Newton iteration; using this idea, division has complexity

$$\mathsf{D}(n) \ \leq \ \frac{5}{2}\,\mathsf{M}(n) + O(n),$$

see [115]. The constant is reached in the FFT model, while for the Karatsuba multiplication model, it drops to 4/3. Using a different approach, Bernstein showed that in the FFT model, the constant can even be dropped down to 13/6, see [22].

### Historical remarks

The first speed-up of the division problem was due to Cook [65, pages 81–86], who applied Newton's method to compute arbitrarily high precision reciprocal for integers, which in turn serves to reduce the time for performing integer division to a constant number of integer multiplications. Fast algorithms for polynomial and power series division came some six years later. In their pioneering paper on multipoint evaluation, Borodin and Moenck [169] proposed a divide-and-conquer algorithm for polynomial division with complexity $O(\mathsf{M}(n)\log(n))$. Independently, Sieveking [228] devised a $O(\mathsf{M}(n))$ algorithm for power series inversion. Strassen [235, Section 3.5] pointed out that polynomial division reduces to power series inversion and produced an improved $O(\mathsf{M}(n))$ polynomial division algorithm, which was rapidly integrated in the second version [30] of Borodin

and Moenck's paper. An important conceptual simplification was made by Kung [138], who showed that Sieveking's algorithm for inverting power series is a particular instance of Newton's method applied over a power series domain. Moenck and Allen [168] gave a unified framework for division over abstract Euclidean domains. Lipson [157] emphasizes the large applicability of this powerful method to exact computations with power series: "*In a symbolic mathematics system setting, we feel that Newton's method offers a unified and easily implemented algorithm for solving a wide variety of power series manipulation problems that can be cast into a root-finding mold. Our experience attests to the practicality of Newton's method. We feel that it is a great algebraic algorithm.*"

### 3.2.2 Division of polynomials

Strassen [235] showed that division of polynomials can be reduced to the division of two formal power series.

Let $a$ and $b$ two polynomials and let $a = bq + r$ be the division with remainder of $a$ by $b$. The idea is to view this equality not at $x = 0$, but at infinity, that is, make the change of variable $x \mapsto x^{-1}$. Suppose that $\deg(a) = 2n$ and $\deg(b) = n$, so that $\deg(q) = n$ and $\deg(r) < n$. Denoting by $\mathrm{rev}(f, d)$ the reverse polynomial $f(1/x)x^d$ one gets the new identity

$$\mathrm{rev}(a, 2n) = \mathrm{rev}(b, n)\mathrm{rev}(q, n) + x^{n+1}\mathrm{rev}(r, n-1)$$

which implies that $\mathrm{rev}(q, n)$ can be obtained by computing the quotient of power series $\mathrm{rev}(a, 2n)/\mathrm{rev}(b, n)$ at precision $n + 1$. This can be done in $O(\mathsf{M}(n))$, so the quotient $q$ and the remainder $r = a - bq$ are also recovered within this complexity.

### 3.2.3 Logarithm and exponential

In [42, Section 13], Brent gave an algorithm for computing logarithms and exponentials of power series using Newton's method. We now recall these algorithms, which will be used in Chapters 6 and 7.

Suppose that $f$ is a power series with constant coefficient 1, given at precision $n$, over a ring of characteristic zero. Its *logarithm* is defined as the power series

$$\log(f) = -\sum_{i \geq 1} \frac{(1-f)^i}{i}.$$

Brent's solution for computing the logarithm of $f$ is based on the fact that the formal derivative of $\log(f)$ equals $f'/f$; consequently, the problem reduces to the term-wise integration of the quotient of two power series. Since integration has cost linear in $n$ and in view of the results of the preceding section, this can be done using at most

$$\mathsf{L}(n) \leq \frac{5}{2}\mathsf{M}(n) + O(n)$$

operations, which is in $O(\mathsf{M}(n))$.

Let now $f$ be a power series with zero constant coefficient over a field of characteristic zero. The exponential of $f$ is defined as the power series

$$\exp(f) = \sum_{i \geq 0} \frac{f^i}{i!}.$$

One can easily check that $\log(\exp(f)) = f$. This is the basis of Brent's algorithm, which applies Newton's method to the function $F(t, x) = \log(x) - t$, leading to the iteration $x_0 = 1$ and

$$x_{k+1} \;=\; x_k + x_k\big(f - \log(x_k)\big) \mod x^{2^{k+1}}.$$

Let us study the cost of the $k$th iteration; it requires a log computation at precision $2^{k+1}$ and a multiplication of series at precision $2^k$ (since the first terms of $f$ and $\log(x_k)$ coincide up to $2^k$), thus using at most $\mathsf{M}(2^k) + \frac{5}{2}\,\mathsf{M}(2^{k+1}) \leq 3\,\mathsf{M}(2^{k+1})$ operations. Summing up for $0 \leq k \leq \lfloor \log(n) \rfloor - 1$, the total cost is

$$\mathsf{E}(n) \;\leq\; 6\,\mathsf{M}(n) + O(n).$$

**Remark** In specific multiplication models this constant may be further improved. For instance, Bernstein [22] shows that in the FFT model, one can compute the exponential for $17/6$ multiplications of power series at precision $n$.

**Powers of power series** As a nice application of fast logarithms and exponentials, we mention the problem of computing the $m$th power of a power series at precision $n$. By repeated squaring (binary powering), this can be done in $O\big(\mathsf{M}(n)\log(m)\big)$ operations. Quite surprisingly, this upper bound can be improved to $O\big(\mathsf{M}(n) + \log(m)\big)$, see [42], by exploiting the fact that if $f(0) = 1$, then

$$f^m \;=\; \exp\big(m \log(f)\big).$$

**Power series solutions of linear differential equations** Another interesting application [43] concerns linear differential equations. Consider the equation

$$y' - A(x)y = B(x),$$

where $A$ and $B$ are power series. Its solutions are obtained by the method of *variation of parameters* [122, Section 2.13] and are given symbolically by

$$y = e^{\int A} \cdot \int B\, e^{-\int A}.$$

As a consequence, computing a power series solution up to order $n$ requires primitives, of complexity linear in $n$ and exponentials, of complexity $O(\mathsf{M}(n))$. This method extends to linear differential equation of arbitrary degree, leading to a complexity (with respect to $n$) of $O(\mathsf{M}(n))$ operations, see [43].

**Further remark** An improvement in efficiency is obtained if we restrict our attention to the special class of linear differential equations with *polynomial coefficients*. In this case, the coefficients of a power series solution satisfy a recurrence relation with *polynomial coefficients* (in $n$) and can be computed by an on-line algorithm using a number of arithmetic operations which is linear in $n$. We point out that the so called *holonomic functions*, which verify such differential equations with polynomial coefficients are ubiquitous, including large spectra of *special functions* [4, 7, 167]. Thus, for all these functions, the Taylor expansion around ordinary points and more generally, around regular singularities [57, 58, 59, 60, 246, 167] can be computed fast, see also Chapter 9 of this thesis, for the case when one is interested in computing only *one coefficient of large index*.

### 3.2.4 Other operations

In the remaining of this section, we briefly mention a few other operations on power series, that will be not used in the rest of this thesis.

**Composition of power series** Suppose that $A = \sum_{i\geq 0} a_i x^i$ and $B = \sum_{i\geq 0} b_i x^i$ are two power series such that $b_0 = 0$. Their *composition* $C = A \circ B$ is defined as the power series

$$\sum_{i\geq 0} c_i x^i = \sum_{j\geq 0} a_j \left( \sum_{i\geq 1} b_i x^i \right)^j .$$

We have $c_0 = a_0$ and for $n \geq 1$, $c_n$ is clearly a polynomial expression in $a_1, \ldots, a_n, b_1, \ldots, b_n$. The *composition problem* is to compute the first $n$ terms of the series $A \circ B$ from the first $n$ terms of the series $A$ and $B$. As special cases, we have already studied the cases $\exp(f) = \mathrm{E} \circ f$ and $\log(f) = \mathrm{Log} \circ f$, where

$$\mathrm{E} = \sum_{i\geq 0} \frac{x^i}{i!} \quad \text{and} \quad \mathrm{Log} = -\sum_{i\geq 1} \frac{(1-x)^i}{i}.$$

The straightforward approach solves the composition problem using $O\big(n\mathsf{M}(n)\big)$ operations. The technique of *baby steps / giant steps* due to Paterson & Stockmeyer [190] reduces this cost to $O\big(\sqrt{n}\,\mathsf{M}(n) + n^2\big)$.
The fast composition scheme of Paterson and Stockmeyer [190] also works if we replace the algebra of power series truncated at $x^n$ by any algebra of dimension $n$. For instance, it applies to quotient algebras $k[x]/(f(x))$, thus solving the problem of *modular compositions of polynomials* $g(h) \mod f$, where $\deg(g), \deg(h) < \deg(f) = n$, within $O\big(\sqrt{n}\,\mathsf{M}(n) + n^2\big)$ operations [256, 224].

Brent and Kung [43] pointed out that the exponent 2 can be dropped to $\frac{\omega+1}{2}$, where $\omega$ is the exponent of matrix multiplication. They also proposed an asymptotically faster method, using time

$$\mathsf{C}(n) \;=\; O\Big(\mathsf{M}(n)\sqrt{n\log(n)}\Big).$$

We mention that this method does not extend directly to the modular composition of polynomials, so that the question of speeding-up the generic solution of Paterson and Stockmeyer in this case is an open problem.

### Further remark

An interesting improvement was obtained by Bernstein [23], who proposed a faster composition algorithm in positive characteristic $p > 0$, of total complexity $O(p\ \mathsf{M}(n)\log(n))$, which is nearly optimal for fixed $p$.

### Reversion of power series

Given a power series $A(x) = \sum_{i\geq 1} a_i x^i$, one can show that there exists a unique power series $B(x) = \sum_{i\geq 1} b_i x^i$, such that

$$A \circ B \ = \ B \circ A \ = \ x.$$

The series $B$ is called the *reverse* or the *functional inverse* of $A$. Given $a_1, \ldots, a_n$, the *reversion problem* is to compute the coefficients $b_1, \ldots, b_n$ of its reverse. It is easy to show that they depend polynomially on $a_1, \ldots, a_n$.

In [43], Newton's iteration is applied to $F(t, x) = A(x) - t$ and helps reducing the reversion problem to the composition problem. The iteration is $x_0 = 0$ and

$$x_{k+1} = x_k - \frac{A(x_k) - t}{A'(x_k)} \mod t^{2^{k+1}}.$$

Using the fact that $A' \circ x_k \ = \ (A \circ x_k)'/x_k'$, the cost $\mathsf{R}$ of reversion satisfies the relation $\mathsf{R}(2^{k+1}) \leq \mathsf{R}(2^k) + \mathsf{C}(2^{k+1}) + O(\mathsf{M}(2^{k+1}))$. Under some natural regularity assumptions on the functions $\mathsf{C}$ and $\mathsf{R}$, this implies that $\mathsf{R}(n) \ = \ O(\mathsf{C}(n))$. Moreover, Brent and Kung [43] proved that composition and reversion are in fact computationally equivalent, that is

$$\mathsf{R}(n) \ = \ O(\mathsf{C}(n)) \ \text{ and } \ \mathsf{C}(n) \ = \ O(\mathsf{R}(n)).$$

## 3.3 Rational fractions and linear recurrence sequences with constant coefficients

Linearly recurrent sequences with constant coefficients and rational functions are incarnations of identical mathematical objects. Their main common specificity is that they can be specified by a finite amount of information: recurrence and initial conditions, respectively numerator and denominator. Algorithmically, this distiguishes rational power series among general power series.

We refer to the two recent surveys [54] and [249] on arithmetic and analytic aspects of recurrent sequences with constant coefficients and to [112, 51, 71, 113, 194, 221, 238, 260, 82, 111, 188] for algorithmic issues related to the computation with such sequences.

The aim of this section is to show that, due to their compact representation, the class of generating power series of linear recurrent sequences is better suited to fast manipulation than general power series.

### 3.3.1 Linear recurrent sequences with constant coefficients

Let $k$ be a field and let $p_0, \dots, p_{d-1}$ be elements of $k$. The sequence $(a_n)_{n\geq 0}$ is called *linearly recurrent* if it satisfies a recurrence relation with constant coefficients of the type

$$a_{n+d} = p_{d-1}a_{n+d-1} + \cdots + p_0 a_n, \quad \text{for all} \quad n \geq 0.$$

The polynomial $P = x^d - p_{d-1}x^{d-1} - \cdots - p_1 x - p_0$ is then called a *characteristic polynomial* of the sequence $(a_n)$. An important property of linearly recurrent sequences is that their generating power series are rational. Indeed, one has the formulas

$$\frac{N_0}{\mathrm{rev}(P)} = \sum_{n\geq 0} a_n x^n \quad \text{and} \quad \frac{N_\infty}{P} = \sum_{n\geq 0} \frac{a_n}{x^{n+1}}.$$

The numerators $N_0$ and $N_\infty$ have degree less than $d$ and encode the initial conditions of the sequence. A simple, but crucial remark is that the sequence $(a_n)_{n\geq D}$ verifies the same recurrence as $(a_n)_{n\geq 0}$. Thus the generating series $\sum_{n\geq D} a_n x^{n-D}$ is also rational, has the same denominator $\mathrm{rev}(P)$ and its numerator $N_D$ encodes the new initial values $a_D, \dots, a_{D+d-1}$.

**Example** The archetypal example of a linearly recurrent sequence with constant coefficients is the celebrated Fibonacci sequence $(F_n)_{n\geq 0}$ defined by

$$F_{n+2} = F_{n+1} + F_n, \quad \text{for all} \quad n \geq 0, \quad \text{with} \quad F_0 = F_1 = 1.$$

Its generating power series at zero and at infinity are

$$\frac{1}{1 - x - x^2} = \sum_{n\geq 0} F_n x^n \quad \text{and} \quad \frac{x}{x^2 - x - 1} = \sum_{n\geq 0} \frac{F_n}{x^{n+1}}.$$

### 3.3.2 Taylor expansion of rational fractions

Consider the following question: given two polynomials $b$ and $A$ of degree $d$, compute the Taylor expansion at order $N \geq 0$ of the rational fraction $b/A$.

The fast solution suggested by the preceding paragraphs consists in applying Newton iteration. If $N \geq d$, the complexity of this method is $O\big(\mathsf{M}(N)\big)$ operations.

Surprisingly, one can do better, by iterating a procedure which allows to pass from a slice of $d$ coefficients to the next one in only $O\big(\mathsf{M}(d)\big)$ operations. The number of needed iterations is $N/d$, so the total cost of this method is

$$O\left(N\,\frac{\mathsf{M}(d)}{d}\right).$$

If fast multiplication is used, this is nearly optimal, since the dependence is linear in the number of computed terms and (poly-)logarithmic in the order of the recurrence. This should be compared with the naïve method, which has complexity $O(dD)$.

We describe this algorithm in more detail. In the next sections, we show that it extends to the case where $A$ is a polynomial matrix and $b$ is a polynomial vector. This extension is the basis of a recent algorithm of Storjohann [233] for solving linear systems with rational function coefficients, which will be described in Section 3.4.2.

Let $P = x^d$ and

$$\frac{b}{A} = c_0 + c_1 P + \dots$$

the Taylor expansion we want to compute. Here $c_i$ are polynomials of degree less than $d$.

Let $C_0$ denote the first $d$ coefficients of the inverse of $A$; they can be computed using Newton's iteration in $O(\mathsf{M}(d))$ operations. Then, it is enough to show that the following formula holds:

$$c_m = -\left\lceil C_0 \ \left\lfloor A\, c_{m-1} \right\rfloor_d \right\rceil^d. \tag{3.1}$$

Here, the notation $\lceil . \rceil^h$, $\lfloor . \rfloor_l$ and $[.]_l^h$ represent truncation operations defined on a polynomial $P = \sum_i p_i T^i$ as follows: $\lceil P \rceil^h = \sum_{i=0}^{h-1} p_i T^i$, $\lfloor P \rfloor_l = \sum p_{i+l} T^i$ and $[P]_l^h = \sum_{i=0}^{h-l-1} p_{i+l} T^i$. In what follows, this notation will also be used for polynomial matrices.

To justify Equation (3.1), we use the equalities

$$\frac{b}{A} = c_0 + \dots + c_{m-1}P^{m-1} + \big(c_m + c_{m+1}P + \dots\big)P^m = c_0 + \dots + c_{m-1}P^{m-1} + \frac{r_m}{A}P^m$$

which translate the fact that the sequences $(c_i)_{i\geq 0}$ and $(c_i)_{i\geq m}$ verify the same recurrence, but have different initial conditions, encoded by $b = r_0$ and $r_m$. Multiplying by $A$ and comparing the coefficients of $P^m$, this equality gives $r_m = -\lfloor A c_{m-1} \rfloor_d$ and $c_m = \lceil r_m/A \rceil^d = \lceil C_0 r_m \rceil^d$, which together imply Equation (3.1).

### 3.3.3 Computing a selected term of a linearly recurrent sequence

We now consider the following problem: given a recurrence of order $d$ and the first $d$ terms of one solution, how fast can we compute one selected term of that solution? In terms of rational fractions, how fast can we determine the $D$th term in the Taylor expansion of a rational function of numerator and denominator of degree $d$?

Our interest in studying this question is motivated by the fact that its extension to the non-scalar case is one of the main ideas of Storjohann's algorithm presented in Section 3.4.2. A fast algorithm for the case of linear recurrences with *non-constant coefficients* is presented in Chapter 9.4.

Notice that for a quotient of general power series, this is as hard as computing all the first $D$ terms. Again, for quotients of polynomials we can solve this problem nearly optimally, that is with complexity $O(\mathsf{M}(d)\log(D))$. The solution relies on the equality

$$\begin{bmatrix} a_D \\ a_{D+1} \\ \vdots \\ a_{D+d-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ p_0 & \cdots & p_{d-2} & p_{d-1} \end{bmatrix}^D \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix}. \tag{3.2}$$

A direct application of Equation (3.2) yields an algorithm of complexity $O(\log(D)d^\omega)$. This is almost optimal in $D$, but not in $d$. A faster solution translates Equation (3.2) into a polynomial identity: the polynomial $a_D + a_{D+1}x + \cdots + a_{D+d-1}$ is the middle part of the product

$$\Big(x^D \bmod P\Big)\Big(a_0 + a_1 x + \cdots + a_{2d-2}x^{2d-2}\Big).$$

Since computations modulo $P$ can be done within $O(\mathsf{M}(d))$ the complexity of the resulting algorithm is $O(\log(D)\,\mathsf{M}(d))$, thus nearly optimal in both $d$ and $D$. A similar complexity result is obtained in [82].

### 3.3.4 Computing the minimal polynomial of a recurrent sequence.

In the preceding sections we considered fast algorithms for converting rational fractions to power series. We now treat the inverse problem: given the first $2N$ terms of a linearly recurrent sequence $(b_k)_{k\geq 0}$, recover the coefficients $u_k$ of the recurrence of minimal order, supposed upper bounded by $N$, of the type

$$\sum_{k=0}^{d} b_{k+i}u_k = 0 \qquad \text{for all } \ i \geq 0. \tag{3.3}$$

This problem is equivalent to the resolution of the linear (Hankel) system defined by the first $N$ equations given by Equation (3.3). In terms of generating series, it can be viewed as a special case of *Padé approximation*. Indeed, Equation (3.3) can be restated as

$$\Big(\sum_{k\geq 0} b_k X^k\Big)\cdot\Big(\sum_{i=0}^{d} u_i X^{d-i}\Big) \ \text{ is a polynomial of degree less than } \ d.$$

In other words, solving the recurrence (3.3) amounts to finding two polynomials $u, v \in K[X]$, such that $\deg(u) \leq N$, $\deg(u) < N$ which satisfy the equality

$$B(X)u(X) - v(X) = O(X^{2N}).$$

Such a pair $(u, v)$ will be called a *$N$-approximant for $B$*.

We now present two variants of Berlekamp-Massey's algorithm [20, 165, 73]. The first one contains the basic idea, but has a quadratic behaviour, while the second one has an almost linear complexity.

**A Quadratic Algorithm** A natural idea is to construct approximants incrementally, that is, to lift a $(k-1)$-approximant into a $k$-approximant, for increasing $k$. To do this, the key idea is to maintain, at step $t$, two couples of candidates $u_1^{(t)},\ u_2^{(t)}$ for $u$ and $v_1^{(t)},\ v_2^{(t)}$ for $v$ and to introduce error series $e_1^{(t)},\ e_2^{(t)}$ which measure how far we are from the desired precision $t = 2N$. A well-chosen linear combination of these error series will produce a new one with

valuation increased by 1; we will then define a candidate at step $t+1$ as the same linear combination of the two candidates at step $t$. The second candidate at step $t+1$ will be just a shift by $X$ of one of the candidates at step $t$. Formalizing this procedure in matrix notation, at step $t$, we have the equation:

$$\begin{bmatrix} B & -1 \end{bmatrix} \cdot \begin{bmatrix} u_1^{(t)} & u_2^{(t)} \\ v_1^{(t)} & v_2^{(t)} \end{bmatrix} = X^t \cdot \begin{bmatrix} e_1^{(t)} & e_2^{(t)} \end{bmatrix}. \tag{3.4}$$

As indicated above, passing from stage $t$ to stage $t+1$ is done by setting:

$$\begin{bmatrix} u_1^{(t+1)} & u_2^{(t+1)} \\ v_1^{(t+1)} & v_2^{(t+1)} \end{bmatrix} := \begin{bmatrix} u_1^{(t)} & u_2^{(t)} \\ v_1^{(t)} & v_2^{(t)} \end{bmatrix} \cdot P^{(t)},$$

where $P^{(t)}$ is one of the following unimodular matrices, of degree at most 1:

$$\begin{bmatrix} e_2^{(t)}(0) & 0 \\ -e_1^{(t)}(0) & X \end{bmatrix} \big(\text{if } e_2^{(t)}(0) \neq 0\big), \text{ or } \begin{bmatrix} X & -e_2^{(t)}(0) \\ 0 & e_1^{(t)}(0) \end{bmatrix} \big(\text{if } e_1^{(t)}(0) \neq 0\big).$$

(if $e_1^{(t)}(0) = e_2^{(t)}(0) = 0$, one simply sets $P^{(t)} = \mathrm{Id}_2$.)

With this choice, equation (3.4) is obviously satisfied at step $t+1$; moreover, at each iteration, the degree of the matrix containing candidates for $(u, v)$ increases by $1/2$ on average, so that after $2N$ iterations, a $N$-approximant will be given by one of its columns. Since the matrices $P^{(t)}$ are unimodular, this approximant will not be the trivial one.

To estimate the cost, we notice that at each iteration, only the constant terms of the $e_i$'s are necessary; their computation requires $O(t)$ operations. The computation of the new candidates by multiplication by $P^{(t)}$ has the same cost. Thus, the complexity of the entire algorithm is $O\Big(\sum_{t=1}^{N} t\Big) = O\big(N^2\big)$.

**An Almost Linear Algorithm** In the previous algorithm, the quadratic cost came from the computation of the $t$th coefficient of the product $Bf$, and also from the multiplication of $f$ by a degree 1 matrix, where $f$ is a polynomial of degree $t/2$.

We will now describe a divide-and-conquer approach that replaces these numerous very unbalanced multiplications by a few big polynomial multiplications.

In order to achieve this, we begin by remarking that in the previous algorithm, the matrix $P^{(t)}$ is simply a 1-approximant matrix for the error series $[e_1^{(t)}, e_2^{(t)}]$, that is

$$\begin{bmatrix} e_1^{(t)} & e_2^{(t)} \end{bmatrix} \cdot P^{(t)} = O\big(X\big) \tag{3.5}$$

This suggests the following divide-and-conquer strategy for computing an $(m+n)$-approximant matrix for $B$: first compute a $m$-approximant matrix $R$ for $B$, then compute an $n$-approximant matrix $Q$ for the error series; then the product of these two matrices represents an $(m+n)$-approximate matrix for $B$. Indeed, if

$$\begin{bmatrix} B & -1 \end{bmatrix} \cdot R = X^m \cdot \begin{bmatrix} e_1 & e_2 \end{bmatrix} \text{ and } \begin{bmatrix} e_1 & e_2 \end{bmatrix} \cdot Q = O\big(X^n\big),$$

then $P = RQ$ has degree $\deg(R) + \deg(Q)$ and satisfies

$$\left[\begin{array}{cc} B & -1 \end{array}\right] \cdot P = O\big(X^{m+n}\big).$$

Let us estimate the cost of this recursive algorithm: if $\mathsf{C}(n)$ denotes the number of operations needed to compute a $n$-approximant for $B$, then

$$\mathsf{C}(n) = 2\mathsf{C}(n/2) + O\big(\mathsf{M}(n)\big).$$

Indeed, the cost $O\big(\mathsf{M}(n)\big)$ comes from the computation of the error series and that of multiplication of the matrices $R$ and $Q$. We conclude that $\mathsf{C}(n) = O\big(\mathsf{M}(n)\log(n)\big)$.

**Remark** The previous algorithm can be extended so as to treat the problem of approximating *several* power series. This is called in the literature the *Padé-Hermite approximation* problem, see the references [17, 18, 148, 97] and the last Chapter of this thesis, where we use Padé-Hermite approximants to compute with differential operators.

## 3.4 Newton iteration for polynomial matrices

Let $k$ be a field and let $A \in \mathcal{M}_{n\times n}\big(k[x]\big)$ be a $n \times n$ invertible polynomial matrix, whose entries are polynomials of degree less than $d$. We say that $A$ has degree less than $d$. We denote by $\mathsf{MM}(n, d)$ the number of arithmetic operations required to multiply two $n \times n$ polynomial matrices of degree less than $d$. Viewing $A$ as a polynomial with matrix coefficients, one can always choose $\mathsf{MM}(n, d) = O(\mathsf{M}(d)n^\omega) = O_{\log}(dn^\omega)$, using the general result of Cantor and Kaltofen [50]. A better bound of $\mathsf{MM}(n, d) = O\big(n^\omega d + n^2\mathsf{M}(d)\big)$ can be obtained using our results on the fast chirp transforms described in Chapter 5.

Let $P(x) \in k[x]$ be a polynomial relatively prime with the determinant $\det(A)$ of $A$. In what follows, we will assume for simplicity that $A(0)$ is invertible, so we can choose $P$ a power of $x$, but the general case can be handled similarly. Under our assumption, the matrix $A^{-1}$ admits a $P$-adic expansion

$$A^{-1} = C_0 + C_1P + \cdots + C_bP^b + \ldots,$$

where $C_i$ are polynomial matrices of degree less than $\deg(P)$.

For instance, let us consider $P = x$. Then the $x$-adic expansion of $A^{-1}$ can be computed up to precision $N \geq 1$ by Newton's iteration

$$X_0 = A(0)^{-1}, \quad X_{i+1} = X_i + X_i\big(I - AX_i\big) \mod x^{2^{i+1}}$$

using

$$O\left(\sum_{i\geq 1} \mathsf{MM}(n, N/2^i)\right) = O\Big(\mathsf{MM}(n, N)\Big)$$

operations in $k$.

Sometimes, as in the scalar case, one only needs a single coefficient or a slice of coefficients in this expansion. This raises a natural question: can one compute faster, say, the $N$th coefficient $X_N$? Recall from the preceding sections that in the scalar case, the $N$th coefficient of the inverse of a polynomial of degree $d$ can be computed using $O\big(\log(N)\ \mathsf{M}(d)\big)$ operations. We will show now a similar result for polynomial matrices, due to Storjohann [233]. To this end, we begin by giving a slight generalization of Newton's iteration.

For any integers $a, b \geq 0$, we have that

$$C_{a+1}P^{a+1}+\cdots+C_{a+b+1}P^{a+b+1} = \big(C_0+\cdots+C_bP^b\big)\Big(1-A\cdot\big(C_0+\cdots+C_aP^a\big)\Big) \mod P^{a+b+2}.$$

**Proof** A quick proof is to multiply term by term the equalities

$$\begin{cases} 1 - A\cdot\big(C_0+\cdots+C_aP^a\big) &= A\cdot\big(C_{a+1}P^{a+1}+\ldots\big) \quad \text{and} \\ \\ C_0+\cdots+C_bP^b &= 1-A\cdot(C_{b+1}P^{b+1}+\ldots) \end{cases}$$

and to use the fact that the right-hand product

$$\big(1-A\cdot(C_{b+1}P^{b+1}+\ldots)\big)\cdot\big(C_{a+1}P^{a+1}+\ldots\big)$$

is equal to $C_{a+1}P^{a+1}+\cdots+C_{a+b+1}P^{a+b+1}$ modulo $P^{a+b+2}$. □

In particular, by comparing the coefficients of $P^{a+b+1}$, we deduce that $C_{a+b+1}$ can be computed from $C_a$, $C_{b-1}$ and $C_b$ in time $O\big(\mathsf{M}(n,d)\big)$. More exactly, the following equality holds

$$C_{a+b+1} = -\left[\big(C_{b-1}+C_bx^d\big)\cdot\lfloor AC_a\rfloor_d\right]_d^{2d}.$$

Choosing now $a = b = 2^{i-1}-1$ and for $a = 2^{i-1}-2$, $b = 2^{i-1}-1$, we deduce that the *high-order components* $C_{2^i-2}$ and $C_{2^i-1}$ $(1 \leq i \leq \ell)$ of the inverse of $A$ can be computed using $O\big(\ell\ \mathsf{M}(n,d)\big)$ operations in $k$.

**Example** Let $F_0 = 1$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$ Fibonacci's sequence. Then, setting $P = x^2$, $C_a = F_{2a} + F_{2a+1}x$ and $A = 1 - x - x^2$, we get

$$F_{2a+2b+2} + F_{2a+2b+3}x = -\quad F_{2b-2} + F_{2b-1}x + F_{2b}x^2 + F_{2b+1}x^3\ \cdot\ (1-x-x^2)(F_{2a}+F_{2a+1}x)\ {}_2\,{}^4_2.$$

This yields the equations

$$\begin{array}{rcl} F_{2a+2b+2} &=& F_{2a+1}F_{2b-1} + (F_{2a}+F_{2a+1})F_{2b}, \\ F_{2a+2b+3} &=& F_{2a+1}F_{2b} + (F_{2a}+F_{2a+1})F_{2b+1}. \end{array} \tag{3.6}$$

Using equation (3.6), we obtain the following doubling step iteration for computing high Fibonacci numbers

$$\Big(F_{2^i-4}, F_{2^i-3}, F_{2^i-2}, F_{2^i-1}\Big) \longmapsto \Big(F_{2^{i+1}-4}, F_{2^{i+1}-3}, F_{2^{i+1}-2}, F_{2^{i+1}-1}\Big).$$

This coincides with the classical method [221], which proceeds as follows:

$$\begin{array}{rcl} F_{2m} &=& F_m^2 + F_{m-1}^2 \\ F_{2m+1} &=& F_m^2 + 2F_{m-1}F_m. \end{array}$$

### 3.4.1 Application to rational system solving

We consider the fundamental problem of solving a linear system

$$Av = b,$$

where $A \in \mathcal{M}_{n\times n}\big(k[x]\big), b \in \mathcal{M}_{n\times 1}\big(k[x]\big)$ are polynomial matrices of degree less than $d$.

By Cramer's rule, the entries of the vector $v = A^{-1}b$ are rational fractions in $x$, whose numerators and denominators have degrees upper bounded by $nd$. Based on this remark, Carter and Moenck [170] proposed a method to compute $v$ by *adic expansion*: first expand $A^{-1}b$ at order $2nd$, then recover its (rational) entries by Padé approximation.

Since the cost of the reconstruction step is $O\big(n\,\mathsf{M}(nd)\log(nd)\big)$, thus nearly linear (up to logarithmic factors) in the size of the output $v$, we concentrate on the first step – computing the expansion of $v = A^{-1}b$ at precision $2nd$.

The naïve approach consists in computing the first $2nd$ terms in the $x$-adic expansion of the inverse $A^{-1}$, then multiplying by $b$. Using Newton's iteration described above, this has complexity $O(\mathsf{MM}(n, 2nd) + n^3\,\mathsf{M}(d)) = O_{\log}(n^{\omega+1}d)$.

Carter and Moenck [170] devised a faster method, which exploits the rationality of the entries of $v$. It is the matrix equivalent of the Taylor expansion of a rational fraction by slices, as explained in Section 3.3.2. Indeed, that method can be literally imported, in the setting of polynomial matrices, provided that one replace the capital letter polynomials by square matrices of polynomials and the small letter polynomials by column vectors of polynomials.

To compute the first $N$ terms in the $x$-adic expansion of $v = A^{-1}b$, this method requires the inversion of $A$ modulo $x^d$ and $2\frac{N}{d}$ matrix-vector multiplications of polynomial matrices of degre $d$, in other words

$$O\left(\mathsf{MM}(n,d) + \frac{Nn^2}{d}\mathsf{M}(d)\right)$$

operations in $k$. In our case, $N$ is bounded by $2nd$, so the complexity is in the class

$$O\big(n^3\,\mathsf{M}(d)\big) = O_{\log}\big(n^3\,d\big).$$

### 3.4.2 Storjohann's algorithm

We describe now a recent method due to Storjohann [233], which further decreases the complexity down to $O_{\log}(n^\omega d)$. This is nearly linear in the size $n^2d$ of the output, under the hypothesis that $\omega = 2$ and neglecting log factors.

Recall the notation

$$A^{-1} = C_0 + C_1P + \cdots + C_bP^b + \ldots$$
$$A^{-1}b = c_0 + c_1P + \cdots + c_{a-1}P^{a-1} + A^{-1}r_aP^a$$

and the fact that the family of *high order components* $\{C_{2^i-2} + C_{2^i-1}\,P\}_{0\le 2^i\le 2n}$, of total size $O\big(\log(n)\cdot n^2d\big)$, can be computed in $O\big(\log(n)\cdot\mathsf{M}(n,d)\big)$.

We have seen in the preceding section that computing $A^{-1}b$ amounts to computing the *residues* $r_a$ fast, for $0 \leq a \leq n$. In the linear $P$-adic lifting method of Carter and Moenck, the $r_a$ are computed one by one, each passage from $r_a$ to $r_{a+1}$ being achieved using a constant number of polynomial matrix-vector products, whence a complexity in $O\big(n^3\,\mathsf{M}(d)\big)$. To reduce the complexity of computing the $r_a$ (and, *a posteriori*, that of rational system solving), Storjohann's improvement is to replace the matrix-vector products by polynomial matrix-matrix products.

Informally, the idea is that passing from $r_a$ to $r_{a+b}$ can be done using a single matrix-vector product involving a particular high-order component of the inverse of $A$ which **only depends** on $b$. Thus, if several $r_a$ are already available, all their translations by $b$ can be computed using a single matrix-matrix product instead of several matrix-vector products.

More precisely, denoting $E_b = C_{b-2} + C_{b-1}P$, one has the formulas

$$r_{a+b} = -\left\lfloor A\,[E_b r_a]_d^{2d}\right\rfloor_d \quad \text{for all} \quad b \geq 2 \quad \text{and} \quad r_{a+1} = \left\lceil A\,[E_2 r_a]_d^{2d}\right\rceil^d.$$

**Proof** The relation $b = A\big(c_0 + c_1 P + \cdots + c_{a+b-1}P^{a+b-1}\big) + r_{a+b}P^{a+b}$ shows that

$$r_{a+b} = -\,\lfloor A c_{a+b-1}\rfloor_d\,. \tag{3.7}$$

On the other hand, $A^{-1}r_a$ rewrites as:

$$c_a + \cdots + c_{a+b-1}P^{b-1} + \ldots \;=\; \big(C_0 + \cdots + C_{b-2}P^{b-2} + C_{b-1}P^{b-1} + \ldots\big)r_a.$$

Comparing the coefficients of $P^{b-1}$ in these expressions, we also obtain

$$c_{a+b-1} = [E_b r_a]_d^{2d}\,. \tag{3.8}$$

Equations (3.7) and (3.8) prove the first equality. The second one is similar. □

This suggests the following divide-and-conquer strategy: from $b = r_0$, compute $r_n$, then translate by $n/2$ and recover $r_{n/2}$ and $r_{3n/2}$, then translate by $n/4$ and recover $r_{n/4}$, $r_{3n/4}$, $r_{5n/4}$ and $r_{7n/4}$, and so on. After $O(\log(n))$ steps, all the residues $r_0, \ldots, r_{2n-1}$ are computed. Step $i$ consists in (a constant number of) matrix multiplications between a $n \times n$ polynomial matrix of degree $d$ (a high-order lifting component) and a $n \times 2^i$ matrix of degree $d$ (built upon the previously constructed $r_i$'s).

Summing up, the total cost of this procedure and of the whole rational system solving algorithm based upon it is

$$O\Big(\mathsf{MM}(n,d)\log(n)\Big).$$

**Application: characteristic polynomials**

The results of the previous section yield a probabilistic algorithm for the computation of characteristic polynomials of *scalar matrices*. The idea comes from [183], but our complexity result is better.

To determine $\chi_M(x) = \det(M - xI)$, we choose a vector $b \in k^n$ and solve the linear system $(M - xI)v = b$ using Storjohann's algorithm. By Cramer's rule, the denominators of the entries of the solution $v$ generically equal $\chi_M$ (in degenerate cases, only a factor of $\chi_M$ is found). The cost of the algorithm is $O(\mathsf{MM}(n)\log(n))$, which is nearly the same as that of Keller-Gehrig's algorithm [132] explained in Subsection 2.3.3.

# Chapter 4

# Tellegen's Principle Into Practice

The transposition principle, also called Tellegen's principle, is a set of transformation rules for linear programs. Yet, though well known, it is not used systematically, and few practical implementations rely on it. In this chapter, we propose explicit transposed versions of polynomial multiplication and division but also new faster algorithms for multipoint evaluation, interpolation and their transposes. We report on their implementation in Shoup's NTL C++ library.

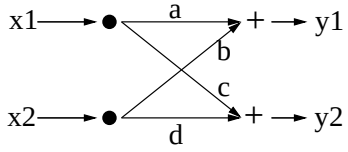
This chapter is joint work with G. Lecerf and É. Schost [35].

## Contents

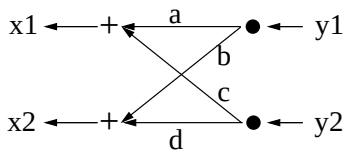
## 4.1 Introduction

The *transposition principle*, sometimes referred to as *Tellegen's principle*, asserts that a linear algorithm that performs a matrix-vector product can be *transposed*, producing an algorithm that computes the transposed matrix-vector product. Further, the transposed algorithm has almost the same complexity as the original one (see Section 4.3 for precise statements).

The following example illustrates this principle, using the computation graph representation. Taking $x_1, x_2$ as input, it computes $y_1 = ax_1 + bx_2, y_2 = cx_1 + dx_2$; edges perform multiplications by the constant values $a, b, c, d$.




Reversing all arrows and exchanging vertices + and • yield the following graph:




Taking $y_1, y_2$ as input, it computes the transposed map $x_1 = ay_1 + cy_2, x_2 = by_1 + dy_2$ (see [128] for details).

Such transformation techniques originate from linear circuit design and analysis [8, 29, 192, 239] and were introduced in computer algebra in [80, 81, 119, 128]. Since then, there has been a recurrent need for transposed algorithms [19, 49, 115, 124, 223, 225, 227, 267]. Yet, the transposition principle in itself is seldom applied, and specific algorithms were often developed to circumvent its use, with the notable exceptions of [115, 225].

**Contributions** In this chapter, we detail several linear algorithms for univariate polynomials and their transposes: multiplication, quotient, remainder, evaluation, interpolation and exemplify a systematic use of Tellegen's principle.

Our first contribution concerns univariate polynomial remaindering: we show that this problem is dual to extending linear recurrence sequences with constant coefficients. This clarifies the status of algorithms by Shoup [223, 227], which were designed as alternatives to the transposition principle: they are actually the transposes of well-known algorithms.

Our second contribution is an improvement (by a constant factor) of the complexities of multipoint evaluation and interpolation. This is done by designing a fast algorithm for transposed evaluation, and transposing it backwards. We also improve the complexity of performing several multipoint evaluations at the same set of points: discarding the costs of the precomputations, multipoint evaluation and interpolation now have very similar complexities.

Finally, we demonstrate that the transposition principle is quite practical. We propose (still experimental) NTL [226] implementations of all algorithms mentioned here and their

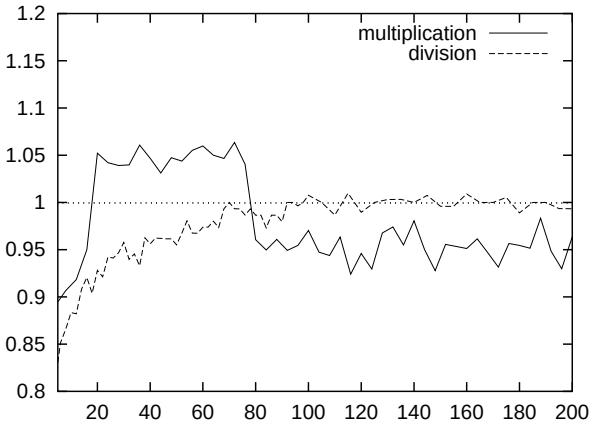



Figure 4.1: Transposed/direct ratios

transposes. They show that the expected time ratio of 1 between an algorithm and its transpose is well respected in practice. The source code can be downloaded from `http://www.math.uvsq.fr/~lecerf`.

**Contents** Section 4.2 introduces the notation used in the rest of this chapter. In Section 4.3, we state the transposition principle in two distinct computation models; we discuss the memory requirement question that was raised in Kaltofen's Open Problem 6 in [124].

In Section 4.4, we transpose polynomial multiplication. This was already done in [115], where transposed versions of Karatsuba's and Fast Fourier Transform (FFT) multiplications were given. We transpose NTL Karatsuba's multiplication, which was optimized to treat unbalanced situations. We also transpose plain, as well as FFT multiplication.
In Section 4.5, we transpose the operations of polynomial quotient and remainder and show the duality between polynomial remaindering and extending linear recurrence sequences.
In Section 4.6, we finally consider polynomial multipoint evaluation and interpolation. We give algorithms for these operations and their transposes that improve previously known algorithms by constant factors.

**Implementation** Our algorithms are implemented in the C++ library NTL [226]. Figures 4.1 and 4.2 describe the behavior of our implementation; the computations were done over $\mathbb{Z}/p\mathbb{Z}$, with $p$ a prime of 64 bit length, and the times were measured on a 32 bit architecture.

Figure 4.1 shows the time ratios between direct and transposed algorithms for polynomial multiplication and polynomial remainder, for degrees up to 200.

- For multiplication, the horizontal axis gives the degree $m$ of the input polynomials. Multiplication in NTL uses three different algorithms: plain multiplication for $m \leq 19$, Karatsuba's multiplication for $20 \leq m < 79$ and FFT for larger $m$. The same thresholds are used for the transposed versions. Note that Karatsuba's multiplication is slightly slower in its transposed version.

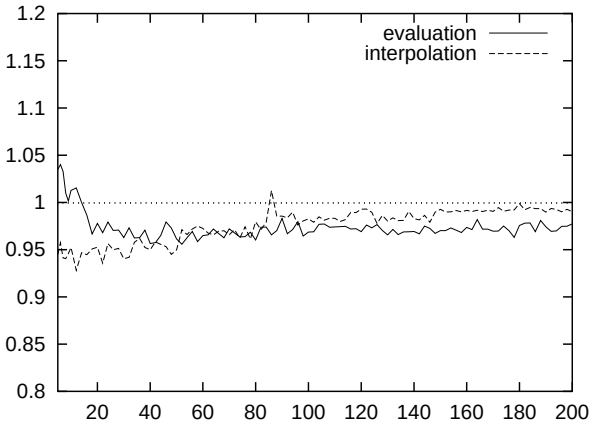



Figure 4.2: Transposed/direct ratios

- The division deals with polynomials of degrees $2m$ and $m$. NTL provides the plain remainder algorithm and a faster one based on Sieveking-Kung's algorithm, with further optimizations. The threshold is 90, we used the same value for the transposed algorithms.

Figure 4.2 presents the ratios between the direct and transposed versions for our new fast multipoint evaluation and interpolation algorithms, with degree on the horizontal axis (Section 4.6 gives their complexity estimates).

**Remark** Transposed algorithms were used for modular multiplication and power series inversion in NTL as soon as [225], yet only for FFT multiplication. More generally the transposed product is used in [115] to speed up some algorithms for power series.

## 4.2 Definitions and Notation

Let $R$ be a commutative ring with unity and $n \geq 0$. By $R[x]_n$ we denote the free $R$-module of polynomials of degree at most $n$. We use the monomial basis $1, x, \ldots, x^n$ on $R[x]_n$ to represent polynomials by vectors and linear maps by matrices. It will also be convenient to consider elements of $R[x]$ as infinite sequences with finite support: we will write a polynomial $a$ as $\sum_{i \geq 0} a_i x^i$, where almost all $a_i$ vanish. The degree of $a$ is denoted by $\deg(a)$, with $\deg(0) = -\infty$.

For $l \geq 0$ and $h \geq 0$, we introduce the following maps $\lceil . \rceil^h$, $\lfloor . \rfloor_l$, $[.]_l^h$ on $R[x]$ and the power series ring $R[[x]]$:

$$\lceil a \rceil^h = \sum_{i=0}^{h-1} a_i x^i, \ \lfloor a \rfloor_l = \sum_{i \geq 0} a_{i+l} x^i, \ [a]_l^h = \sum_{i=0}^{h-l-1} a_{i+l} x^i.$$

Observe that these maps satisfy the following relations:

$$a = \lceil a \rceil^h + x^h \lfloor a \rfloor_h , \quad [a]_l^h = \lceil \lfloor a \rfloor_l \rceil^{h-l} = \left\lfloor \lceil a \rceil^h \right\rfloor_l .$$

We define the *reversal* endomorphism $\mathrm{rev}(n, .)$ of $R[x]_n$ by $\mathrm{rev}(n, a) = \sum_{k=0}^{n} a_{n-k} x^k$, for all $a \in R[x]_n$.

By $\lfloor q \rfloor$ we denote the integer part of a rational $q$. Finally, we use a block matrix notation: $0_{h,l}$ denotes a $h \times l$ block filled with zeros and $1_h$ the $h \times h$ identity matrix.

## 4.3 Tellegen's principle

Tellegen's principle is usually stated using linear computation graphs [128, 239]. In this section, we first briefly recall it in terms of *linear straight-line programs*. We then define another computational model, and prove Tellegen's principle in that setting. The first model can be thought as taking only time complexity into account, while the second one considers both time and space.

**Linear Straight-Line Programs** Linear straight-line programs can be thought as "ordinary" straight-line programs, but with only linear operations, see [47, Chapter 13] for precise definitions. Their complexity is measured by their number of operations, the *size*, which reflects a *time* complexity. In this model, Tellegen's principle can be formulated as:

**Proposition 1** [47, Th. 13.20] *Let $\phi : R^n \to R^m$ be a linear map that can be computed by a linear straight-line program of size $L$ and whose matrix in the canonical bases has no zero rows or columns. Then the transposed map $\phi^t$ can be computed by a linear straight-line program of size $L - n + m$.*

We will use this proposition in Section 4.6, as the complexity estimates used in that section are given in the linear straight-line program model.

**Another Computational Model** We now introduce a measure of space complexity. Informally speaking, our model derives from the *random access memory* model by restricting to linear operations. In this context, a program $\mathcal{P}$ is given by:

- A finite set $\mathcal{D}$ of *registers*, with two distinguished subsets (non necessarily disjoint), $\mathcal{I}$ for the *input* and $\mathcal{O}$ for the *output*: before execution $\mathcal{I}$ contains the input values, at the end $\mathcal{O}$ contains the output.
- A finite sequence of instructions $(\phi_i)_{i \in \{1,\dots,L\}}$ of length $L$. Each instruction is a linear endomorphism of $R^{\mathcal{D}}$ of the following type, with $p, q$ in $\mathcal{D}$, $a$ in $R$ and $f$ in $R^{\mathcal{D}}$:
  - $\mathbf{p} \mathrel{+}= \mathbf{q}$ denotes the map that sends $f$ to the function $g$ defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = f(p) + f(q)$.
  - $\mathbf{p} \mathrel{*}= \mathbf{a}$ denotes the map that sends $f$ to the function $g$ defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = af(p)$.

– $\mathbf{p} = \mathbf{0}$ denotes the map that sends $f$ to the function $g$ defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = 0$.

Let $\mathsf{I}$ (resp. $\mathsf{P}$) be the injection $R^{\mathcal{I}} \to R^{\mathcal{D}}$ (resp. the projection $R^{\mathcal{D}} \to R^{\mathcal{O}}$). Then we say that the program $\mathcal{P}$ *computes* the linear map $\Phi : R^{\mathcal{I}} \to R^{\mathcal{O}}$ defined by $\mathsf{P} \circ \phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1 \circ \mathsf{I}$.

We can now define the *transpose* $\mathcal{P}^t(\mathcal{D}^t, \mathcal{I}^t, \mathcal{O}^t)$ of $\mathcal{P}$:

- The set of registers of $\mathcal{P}^t$ is still $\mathcal{D}^t = \mathcal{D}$ but we let $\mathcal{I}^t = \mathcal{O}$ and $\mathcal{O}^t = \mathcal{I}$: input and output are swapped.
- The instructions of $\mathcal{P}^t$ are $\phi_L^t, \phi_{L-1}^t, \ldots, \phi_1^t$.

Let us verify that $\mathcal{P}^t$ is well-defined. We examine the transpose of each type of instructions:

- The transpose $\phi^t$ of an instruction $\phi$ of type $p \mathrel{+}= q$ is the instruction $q \mathrel{+}= p$.
- The last two instructions are symmetric maps.

The equality $(\phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1)^t = \phi_1^t \circ \phi_2^t \circ \cdots \circ \phi_L^t$ then shows that $\mathcal{P}^t$ computes the linear map $\Phi^t$. This yields the following form of Tellegen's principle:

**Proposition 2** *According to the above notation, let $\phi : R^n \to R^m$ be a linear map that can be computed by a program with $D$ registers and $L$ instructions. Then $\phi^t$ can be computed by a program with $D$ registers and $L$ instructions.*

As an example, consider 5 registers, $x_1, x_2$ for input, $y_1, y_2$ for output and $r$ for temporaries. Given $a, b, c, d$ in $R$, consider the instructions: $y_1 \mathrel{+}= x_1$, $y_1 \mathrel{*}= a$, $r \mathrel{+}= x_2$, $r \mathrel{*}= b$, $y_1 \mathrel{+}= r$, $r = 0$, $y_2 \mathrel{+}= x_1$, $y_2 \mathrel{*}= c$, $r \mathrel{+}= x_2$, $r \mathrel{*}= d$, $y_2 \mathrel{+}= r$. It is immediate to check that the linear map computed by this program is the $2 \times 2$ matrix-vector product presented in the introduction. The transposed program is $r \mathrel{+}= y_2$, $r \mathrel{*}= d$, $x_2 \mathrel{+}= r$, $y_2 \mathrel{*}= c$, $x_1 \mathrel{+}= y_2$, $r = 0$, $r \mathrel{+}= y_1$, $r \mathrel{*}= b$, $x_2 \mathrel{+}= r$, $y_1 \mathrel{*}= a$, $x_1 \mathrel{+}= y_1$.

We could have used more classical instructions such as $p = q + r$ and $p = aq$, $a \in R$. It is not difficult to check that such programs can be rewritten in our model within the same space complexity but a constant increase of time complexity. For such programs, Tellegen's principle would be stated with no increase in space complexity, but a constant increase in time complexity. This is why we use straight-line programs for our complexity estimates in Section 4.6.

Open Problem 6 in [124] asks for a transposition theorem without space complexity swell. The above Proposition sheds new light on this problem: In the present computational model, it is immediate to observe that memory consumption is left unchanged under transposition.

**Comments** The above models compute functions of fixed input and output size: in the sequel, we will actually write families of programs, one for each size. We also use control instructions as **for** and **if**, and calls to previously defined subroutines. Last, we will consider algorithms mixing linear and non-linear precomputations; the transposition principle leaves the latter unchanged.

## 4.4 Polynomial multiplication

In this section $a$ is a fixed polynomial of degree $m$. For an integer $n \geq 0$, consider the multiplication map by $a$:

$$\begin{array}{rccc} \mathrm{mul}(a,.): & R[x]_n & \to & R[x]_{m+n} \\ & b & \mapsto & ab. \end{array}$$

The transposed map is denoted by $\mathrm{mul}^t(n,a,.)$; to write our pseudo-code, it is necessary to consider $n$ as an argument of the transposed function. The next subsection shows that this map is:

$$\begin{array}{rccc} \mathrm{mul}^t(n,a,.): & R[x]_{m+n} & \to & R[x]_n \\ & c & \mapsto & [\mathrm{rev}(m,a)c]_m^{n+m+1}. \end{array}$$

We adopt the following convention: if $\deg(c) > m+n$ then $\mathrm{mul}^t(n,a,c)$ returns an error.

The above formula explains why the transposed multiplication is also called the *middle product* [115]. Performing this operation fast is the first task to accomplish before transposing higher level algorithms. Observe that computing $\mathrm{mul}^t(n,a,c)$ by multiplying $\mathrm{rev}(m,a)$ by $c$ before extracting the middle part requires to multiply two polynomials of degrees $m$ and $m+n$.

Tellegen's principle implies that this transposed computation can be performed for the cost of the multiplication of two polynomials of degrees $m$ and $n$ only. In what follows, we make this explicit for the plain, Karatsuba and Fast Fourier Transform multiplications.

### 4.4.1 Plain multiplication

In the canonical monomial bases, the matrix of the linear map $\mathrm{mul}(a,.): R_n[x] \to R_{m+n}[x]$ is the following Toeplitz matrix $T$ with $m+n+1$ rows and $n+1$ columns:

$$T = \begin{bmatrix} a_0 & 0 & \dots & 0 \\ a_1 & a_0 & \ddots & \vdots \\ \vdots & a_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 \\ a_m & \vdots & \ddots & a_1 \\ 0 & a_m & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_m \end{bmatrix}.$$

Basically, the plain multiplication algorithm corresponds to performing the matrix vector product $c = Tb$ naively using the following sequence of instructions:

$c \leftarrow 0$;
**for** $i$ **from** $0$ **to** $m+n$ **do**
  **for** $j$ **from** $\max(0, i-n)$ **to** $\min(m,i)$ **do**
    $c_i \leftarrow c_i + a_j b_{i-j}$;

Reversing the computation flow, $b = \mathrm{mul}^t(n, a, c)$ is computed by the following program:

$b \leftarrow 0$;
**for** $i$ **from** $m+n$ **downto** $0$ **do**
  **for** $j$ **from** $\min(m, i)$ **downto** $\max(0, i-n)$ **do**
    $b_{i-j} \leftarrow b_{i-j} + a_j c_i$;

Observing that the entries of $b$ are given by:

$$b_i = \sum_{j=i}^{i+m} a_{j-i} c_j, \quad i \in \{0, \ldots, n\},$$

we deduce that the map $\mathrm{mul}^t(n, a, .)$ reformulates in terms of the *middle product* [115] by the relation

$$\mathrm{mul}^t(n, a, c) = [\mathrm{rev}(m, a)c]_m^{n+m+1},$$

as announced above.

### 4.4.2 Karatsuba's algorithm

The next paragraphs are devoted to the transposition of Karatsuba's multiplication algorithm. Concretely, we consider the NTL implementation and present the transpose version we made of it. Figures 4.3 and 4.4 describe Karatsuba's algorithm and its transpose. As for the $\mathrm{mul}^t$ function, the transposed Karatsuba multiplication takes $n$ as an additional argument.

In order to prepare the proof of Algorithm TKarMul we first decompose Algorithm KarMul into linear maps. Following Algorithm KarMul we enter the procedure with polynomials $a$ of degree $m$ and $b$ of degree $n$. We let $\mu = \lfloor m/2 \rfloor + 1$, $\nu = \lfloor n/2 \rfloor + 1$ and $h = \max(\mu, \nu)$ and distinguish three exclusive cases:

*Normal case:* ($m < n$ and $\nu \leq m$) or ($m \geq n$ and $\mu \leq n$). Note that $\deg(\lfloor a \rfloor_h) = m - h$, let $\rho = \deg(\lceil a \rceil^h)$ and $\lambda = \deg(\lceil a \rceil^h + \lfloor a \rfloor_h)$. Let $M_l : R[x]_{h-1} \to R[x]_{\rho+h-1}$, $M_h : R[x]_{n-h} \to R[x]_{m+n-2h}$ and $M_f : R[x]_{h-1} \to R[x]_{\lambda+h-1}$ be the linear maps of multiplication by resp. $\lceil a \rceil^h$, $\lfloor a \rfloor_h$ and $\lceil a \rceil^h + \lfloor a \rfloor_h$. Noticing that $n - h \leq h - 1$, we construct the following block matrix $M$, where we take $r = m + n$ and $q = \rho + h$:

$$M = \left[\begin{array}{c|c} \begin{array}{c|c} M_l & 0_{q,n-h+1} \\ \hline 0_{r-2h+1,h} & M_h \end{array} \\ \hline M_f \cdot \left[\begin{array}{c|c} 1_h & \begin{array}{c} 1_{n-h+1} \\ \hline 0_{2h-n-1,n-h+1} \end{array} \end{array}\right] \end{array}\right].$$

Remark that $(r, s, t) = Mb$, when considering that $r \in R[x]_{\rho+h-1}$, $s \in R[x]_{m+n-2h}$ and $t \in R[x]_{\lambda+h-1}$. In order to recover the product $c = ab$ it remains to compute $c = (N_+ - N_-)(r, s, t)$, with

$$N_+ = \left[\begin{array}{c|c|c} \begin{array}{c} 1_{\rho+h} \\ \hline 0_{r+1-q,q} \end{array} & \begin{array}{c} 0_{2h,r-2h+1} \\ \hline 1_{r-2h+1} \end{array} & \begin{array}{c} 0_{h,\lambda+h} \\ \hline 1_{\lambda+h} \\ \hline 0_{r+1-\lambda-2h,\lambda+h} \end{array} \end{array}\right],$$

$m \leftarrow \deg(a)$;
$n \leftarrow \deg(b)$;
**if** $n = 0$ **then return** $b_0 a$;
**if** $m \leq 0$ **then return** $a_0 b$;
$\mu \leftarrow \lfloor m/2 \rfloor + 1$;
$\nu \leftarrow \lfloor n/2 \rfloor + 1$;
**if** $\mu > n$ **then**
  $u \leftarrow \mathrm{KarMul}(\lceil a \rceil^{\mu}, b)$;
  $v \leftarrow \mathrm{KarMul}(\lfloor a \rfloor_{\mu}, b)$;
  **return** $u + x^{\mu} v$;
**if** $\nu > m$ **then**
  $u \leftarrow \mathrm{KarMul}(a, \lceil b \rceil^{\nu})$;
  $v \leftarrow \mathrm{KarMul}(a, \lfloor b \rfloor_{\nu})$;
  **return** $u + x^{\nu} v$;
$h \leftarrow \max(\mu, \nu)$;
$r \leftarrow \mathrm{KarMul}(\lceil a \rceil^{h}, \lceil b \rceil^{h})$;
$s \leftarrow \mathrm{KarMul}(\lfloor a \rfloor_{h}, \lfloor b \rfloor_{h})$;
$t \leftarrow \mathrm{KarMul}(\lceil a \rceil^{h} + \lfloor a \rfloor_{h}, \lceil b \rceil^{h} + \lfloor b \rfloor_{h})$;
**return** $r + x^{h}(t - s - r) + x^{2h} s$;

Figure 4.3: KarMul$(a, b)$

$$N_{-} = \left[\begin{array}{c|c|c} \begin{array}{c} 0_{h,\rho+h} \\ \hline 1_{\rho+h} \\ \hline 0_{r-\rho-2h+1,\rho+h} \end{array} & \begin{array}{c} 0_{h,r-2h+1} \\ \hline 1_{r-2h+1} \\ \hline 0_{h,r-2h+1} \end{array} & 0_{r+1,\lambda+h} \end{array}\right].$$

*Degenerate case 1:* $\mu > n$. We let $\rho = \deg(\lceil a \rceil^{\mu})$ and consider $u \in R[x]_{\rho+n}$, $v \in R[x]_{m+n-\mu}$ so that we have $ab = N_1(u, v)$, where $N_1$ is:

$$N_1 = \left[\begin{array}{c|c} \begin{array}{c} 1_{\rho+n+1} \\ \hline 0_{m-\rho,\rho+n+1} \end{array} & \begin{array}{c} 0_{\mu,r-\mu+1} \\ \hline 1_{r-\mu+1} \end{array} \end{array}\right].$$

*Degenerate case 2:* $\nu > m$. We consider $u \in R[x]_{m+\nu-1}$, $v \in R[x]_{m+n-\nu}$ so that we have $ab = N_2(u, v)$, with

$$N_2 = \left[\begin{array}{c|c} \begin{array}{c} 1_{m+\nu} \\ \hline 0_{n-\nu+1,m+\nu} \end{array} & \begin{array}{c} 0_{\nu,r-\nu+1} \\ \hline 1_{r-\nu+1} \end{array} \end{array}\right].$$

**Proposition 3** *Algorithm* TKarMul *is correct.*

*Proof.* We use the notation of Figure 4.4 and distinguish the same three cases. The normal

$m \leftarrow \deg(a)$;
**if** $\deg(c) > m + n$ **then** Error;
**if** $n = 0$ **then return** $\sum_{k=0}^{m} a_k c_k$;
**if** $m \leq 0$ **then return** $a_0 c$;
$\mu \leftarrow \lfloor m/2 \rfloor + 1$;
$\nu \leftarrow \lfloor n/2 \rfloor + 1$;
$\rho \leftarrow \deg(\lceil a \rceil^h)$;
**if** $\mu > n$ **then**
  $u \leftarrow \text{TKarMul}(n, \lceil a \rceil^{\mu}, \lceil c \rceil^{n+\rho})$;
  $v \leftarrow \text{TKarMul}(n, \lfloor a \rfloor_{\mu}, \lfloor c \rfloor_{\mu})$;
  **return** $u + v$;
**if** $\nu > m$ **then**
  $u \leftarrow \text{TKarMul}(\nu - 1, a, \lceil c \rceil^{m+\nu})$;
  $v \leftarrow \text{TKarMul}(n - \nu, a, \lfloor c \rfloor_{\nu})$;
  **return** $u + v$;
$h \leftarrow \max(\mu, \nu)$;
$\lambda \leftarrow \deg(\lceil a \rceil^h + \lfloor a \rfloor_h)$;
$r \leftarrow \text{TKarMul}(h - 1, \lceil a \rceil^h, \lceil c \rceil^{\rho+h} - [c]_h^{\rho+2h})$;
$s \leftarrow \text{TKarMul}(n - h, \lfloor a \rfloor_h, \lfloor c \rfloor_{2h} - [c]_h^{m+n-h+1})$;
$t \leftarrow \text{TKarMul}(h - 1, \lceil a \rceil^h + \lfloor a \rfloor_h, [c]_h^{\lambda+2h})$;
**return** $r + t + x^h(s + \lceil t \rceil^{n-h+1})$;

Figure 4.4: TKarMul$(n, a, c)$

case follows from the equalities

$$\begin{aligned} N_+^t c &= (\lceil c \rceil^{\rho+h}, \lfloor c \rfloor_{2h}, [c]_h^{\lambda+2h}), \\ N_-^t c &= ([c]_h^{\rho+2h}, [c]_h^{m+n-h+1}, 0). \end{aligned}$$

In Degenerate cases 1 and 2, we respectively compute

$$N_1^t c = (\lceil c \rceil^{n+\rho}, \lfloor c \rfloor_{\mu}) \text{ and } N_2^t c = (\lceil c \rceil^{m+\nu}, \lfloor c \rfloor_{\nu}).$$

□

Some refinements may be done when implementing Algorithm TKarMul. First observe that it saves memory to compute $\lceil c \rceil^{m+n-h+1} - \lfloor c \rfloor_h$ in order to deduce $r$ and $-s$ and then one propagates this change of sign by returning $r + t + x^h(\lceil t \rceil^{n-h+1} - s)$. Another observation is that Karatsuba's multiplication can be done in a different manner using the identity $\lceil a \rceil^h \lfloor b \rfloor_h + \lfloor a \rfloor_h \lceil b \rceil^h = \lceil a \rceil^h \lceil b \rceil^h + \lfloor a \rfloor_h \lfloor b \rfloor_h - (\lceil a \rceil^h - \lfloor a \rfloor_h)(\lceil b \rceil^h - \lfloor b \rfloor_h)$. When transposing this slightly different version, we obtain the middle product algorithm presented in [115].

### 4.4.3 The Fast Fourier Transform

Multiplication algorithms using the Fast Fourier Transform are quite easy to transpose since the matrices involved are symmetric. We only give a brief presentation and refer to [255] for more details about the discrete Fourier transform.

Let $l \in \mathbb{N}$ such that $m+n+1 \leq 2^l$ and that $R$ contains a primitive $2^l$-th root $\omega$ of unity. The discrete Fourier transform $\mathrm{DFT}(\omega, a)$ of the polynomial $a$ is the vector

$$\big(a(1), a(\omega), \ldots, a\big(\omega^{2^l-1}\big)\big) \in R^{2^l}.$$

Let $\mathrm{DFT}^{-1}(\omega, .) : R^{2^l} \to R[x]_{2^l-1}$ be the inverse map of $\mathrm{DFT}(\omega, .)$ and $H$ the diagonal matrix of diagonal $\mathrm{DFT}(\omega, a)$. Then we have the equality

$$ab = \mathrm{DFT}^{-1}(\omega, H\,\mathrm{DFT}(\omega, b)).$$

Since $\mathrm{DFT}(\omega, .)$ and $H$ are symmetric, we deduce that:

$$\mathrm{mul}^t(n, a, c) = \big\lceil \mathrm{DFT}(\omega, H\,\mathrm{DFT}^{-1}(\omega, c)) \big\rceil^{n+1},$$

for any polynomial $c$ of degree at most $m+n$. Letting $\widetilde{H}$ be the diagonal $\mathrm{DFT}(\omega, \mathrm{rev}(m, a))$ matrix and using

$$\mathrm{DFT}^{-1}(\omega, .) = \frac{1}{2^l}\mathrm{DFT}(\omega^{-1}, .),$$

we deduce the equalities

$$\begin{aligned} \mathrm{mul}^t(n, a, c) &= \Big\lceil \mathrm{DFT}^{-1}(\omega, \omega^{-m}\widetilde{H}\,\mathrm{DFT}(\omega, c)) \Big\rceil^{n+1} \\ &= \Big[ \mathrm{DFT}^{-1}(\omega, \widetilde{H}\,\mathrm{DFT}(\omega, c)) \Big]_m^{m+n+1}, \end{aligned}$$

which can also be obtained from the middle product formulation.

## 4.5 Polynomial Division

We come now to the transposition of the Euclidean division. In this section we are given a polynomial $a \neq 0$ of degree $m$ whose leading coefficient $a_m$ is invertible. For a polynomial $b$, we write the division of $b$ by $a$ as $b = aq + r$ with $\deg(r) < m$ and define the maps

$$\begin{array}{rccc} \mathrm{quo}(a, .): & R[x]_n & \to & R[x]_{n-m} \\ & b & \mapsto & q, \end{array}$$

$$\begin{array}{rccc} \mathrm{rem}(a, .): & R[x]_n & \to & R[x]_{m-1} \\ & b & \mapsto & r, \end{array}$$

$$\begin{array}{rccc} \mathrm{quorem}(a, .): & R[x]_n & \to & R[x]_{n-m} \times R[x]_{m-1} \\ & b & \mapsto & (q, r). \end{array}$$

The transposed operations are written $\mathrm{quorem}^t(n,a,q,r)$, $\mathrm{quo}^t(n,a,q)$ and $\mathrm{rem}^t(n,a,r)$, with the convention that these functions return an error if $\deg(r) \geq \deg(a)$ or $\deg(q) > n - \deg(a)$. The next paragraphs are devoted to transpose the quorem map through the plain and Sieveking-Kung's division algorithms. We will prove that the transposed remainder is given by:

$$\begin{array}{rccc} \mathrm{rem}^t(n,a,.): & R^m & \to & R^{n+1} \\ & (r_0,\ldots,r_{m-1}) & \mapsto & (b_0,\ldots,b_n), \end{array}$$

where the $b_j$ are defined by the linear recurrence

$$(*) \quad b_j = -\frac{1}{a_m}\Big(a_{m-1}b_{j-1} + \cdots + a_0 b_{j-m}\Big), \quad m \leq j$$

with initial conditions $b_j = r_j, \quad j \in \{0,\ldots,m-1\}$, so that linear recurrence sequence extension is dual to remainder computation.

### 4.5.1 Plain division

We enter the plain division procedure with the two polynomials $a$ and $b$ and compute $q$ and $r$ by the following algorithm

```
q ← 0;
r ← b;
for i from 0 to n − m do
    q ← xq + r_{n−i}/a_m;
    r ← r − r_{n−i}/a_m x^{n−m−i} a;
```

To transpose this algorithm we introduce the sequences

$$q^{\{i\}} \in R[x]_{i-1},\ r^{\{i\}} \in R[x]_{n-i} \quad i = 0,\ldots,n-m+1.$$

They are defined by $q^{\{0\}} = 0, r^{\{0\}} = b$ and for $i \geq 1$

$$\begin{array}{rcl} q^{\{i+1\}} & = & xq^{\{i\}} + r^{\{i\}}_{n-i}/a_m, \\ r^{\{i+1\}} & = & r^{\{i\}} - r^{\{i\}}_{n-i}/a_m x^{n-m-i} a, \end{array}$$

so that the relation $b = ax^{n-m+1-i}q^{\{i\}} + r^{\{i\}}$ holds. For $i = n-m+1$, we have $q = q^{\{n-m+1\}}$ and $r = r^{\{n-m+1\}}$.

In order to formulate the algorithm in terms of linear maps we introduce $v^{\{i\}} = (q^{\{i\}}, r^{\{i\}}) \in R^{n+1}$. Then we have $v^{\{i+1\}} = Mv^{\{i\}}$, where

$$M = \left[\begin{array}{c|c} 0_{1,n} & 1/a_m \\ \hline & 0_{n-m,1} \\ & -a_0/a_m \\ 1_n & \vdots \\ & -a_{m-1}/a_m \end{array}\right].$$

Now reverse the flow of calculation: we start with a vector $v^{\{n-m+1\}} = (q^{\{n-m+1\}}, r^{\{n-m+1\}}) \in R^{n+1}$ with $q^{\{n-m+1\}} \in R[x]_{n-m}$ and $r^{\{n-m+1\}} \in R[x]_{m-1}$. Then we compute $v^{\{i\}} = M^t v^{\{i+1\}}$ by the formulae

$$\begin{aligned} q^{\{i\}} &= \left\lfloor q^{\{i+1\}} \right\rfloor_1, \\ r^{\{i\}} &= r^{\{i+1\}} + \frac{1}{a_m} x^{n-i} \Big( q_0^{\{i+1\}} - \sum_{j=0}^{m-1} a_{m-1-j} r_{n-i-j-1}^{\{i+1\}} \Big). \end{aligned}$$

We deduce the following transposed algorithm for computing $b = \mathrm{quorem}^t(n, a, q, r)$:

$b \leftarrow r$;
**for** $i$ **from** $m$ **to** $n$ **do**
  $b_i \leftarrow \left( q_{i-m} - \sum_{j=0}^{m-1} a_{m-1-j} b_{i-j-1} \right) / a_m$;

The maps $\mathrm{quo}(a, .)$ and $\mathrm{rem}(a, .)$ can be obtained by composing a projection after $\mathrm{quorem}(a, .)$ but in practice it is better to implement specific optimized procedures for each. It is easy to implement these optimizations; we refer to our NTL implementation for details.

Since $b = \mathrm{rem}^t(n, a, r) = \mathrm{quorem}^t(n, a, 0, r)$, the coefficients of $b$ satisfy the linear recurrence relation with constant coefficients $(*)$, as announced above.

### 4.5.2 Sieveking-Kung's division

Sieveking-Kung's algorithm is based on the formula

$$\mathrm{rev}(n, b) = \mathrm{rev}(n - m, q)\mathrm{rev}(m, a) + x^{n-m+1}\mathrm{rev}(m - 1, r),$$

see [255] for details. This yields

$$q = \mathrm{rev}(n - m, \lceil \mathrm{mul}(\alpha, \lceil \mathrm{rev}(n, b) \rceil^{n-m+1}) \rceil^{n-m+1}),$$

where $\alpha = \lceil \mathrm{rev}(m, a)^{-1} \rceil^{n-m+1}$. Then we deduce $r$ from $q$ using $r = b - aq$.

Transposing these equalities, we see that for $q \in R[x]_{n-m}$

$$\mathrm{quo}^t(n, a, q) = \mathrm{rev}(n, \mathrm{mul}^t(n - m, \alpha, \mathrm{rev}(n - m, q))).$$

For $r \in R[x]_{m-1}$, it follows:

$$\mathrm{rem}^t(n, a, r) = r - \mathrm{quo}^t(n, a, \mathrm{mul}^t(n - m, a, r)).$$

Let $s = \mathrm{rem}^t(n, a, r)$ and $p = \mathrm{mul}^t(n-m, a, r)$. Using the middle product formula to express this last $\mathrm{quo}^t$ expression yields

$$s = r - \mathrm{rev}(n, [\mathrm{rev}(n - m, \alpha)\mathrm{rev}(n - m, p)]_{n-m}^{2(n-m)+1}),$$

which simplifies to

$$s = r - \mathrm{rev}(n, \mathrm{rev}(n-m, \lceil \alpha p \rceil^{n-m+1})).$$

Last we obtain

$$\mathrm{rem}^t(n, a, r) = r - x^m \left\lceil \alpha \mathrm{mul}^t(n-m, a, r) \right\rceil^{n-m+1},$$

which can also be rewritten this way, using the middle product formula again:

$$\mathrm{rem}^t(n, a, r) = r - x^m \left\lceil \alpha \left[ \mathrm{rev}(m, a) r \right]_m^{n+1} \right\rceil^{n-m+1}.$$

This actually coincides with the algorithm given in [223] for extending linear recurrence sequences.

**Remark** More generally, the following formula holds:

$$\mathrm{quorem}^t(n, a, q, r) = r - x^m \left\lceil \alpha(\left[ \mathrm{rev}(m, a) r \right]_m^{n+1} - q) \right\rceil^{n-m+1}.$$

### 4.5.3 Modular multiplication

As a byproduct, our algorithms enable to transpose the *modular multiplication*. Given a monic polynomial $a$ of degree $m$ with invertible leading coefficient and a polynomial $b$ of degree at most $m-1$, consider the composed map

$$\begin{array}{ccccc} R[x]_{m-1} & \to & R[x]_{2m-2} & \to & R[x]_{m-1} \\ c & \mapsto & bc & \mapsto & bc \mod a. \end{array}$$

An ad hoc algorithm for the transpose map is detailed in [227]. Using our remark on the transpose of polynomial remaindering, it is seen to essentially coincide with the one obtained by composing the algorithms for transposed multiplication and transposed remainder. A constant factor is lost in [227], as no middle product algorithm is used to transpose the first map; this was already pointed out in [115].

## 4.6 Transposed Vandermonde

We now focus on algorithms for Vandermonde matrices; we assume that $R = k$ is a field and consider $m+1$ pairwise distinct elements $a_0, \ldots, a_m$ in $k$. Even if not required by the algorithms, we take $m = 2^l - 1$, with $l \in \mathbb{N}$, in order to simplify the complexity estimates.

The Vandermonde matrix $V_a$ is the square matrix:

$$V_a = \begin{bmatrix} 1 & a_0 & a_0^2 & \ldots & a_0^m \\ 1 & a_1 & a_1^2 & \ldots & a_1^m \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & a_m & a_m^2 & \ldots & a_m^m \end{bmatrix}.$$

If $b$ is in $k[x]_m$ then $V_a b$ is the vector $(b(a_0), \ldots, b(a_m))$. This computation is commonly referred to as *multipoint evaluation*. The inverse problem is *interpolation*: for $c \in k^{m+1}$, $V_a^{-1} c$ corresponds to the polynomial $b \in k[x]_m$ satisfying $b(a_i) = c_i$ for $i \in \{0, \ldots, m\}$.

We first recall the notion of *subproduct tree* and refer to [255, §10.1] for details and historical notes. Then we design a fast algorithm for transposed evaluation. We transpose it backwards to obtain an improved evaluation algorithm, and deduce similar improvements for interpolation and its transpose.

For complexity analysis we use the straight-line program model [47, Chapter 4]. The function $\mathsf{M}(n)$ denotes the complexity of multiplying a polynomial of degree less than $n$ by a fixed polynomial of degree less than $n$, restricting to linear straight-line programs only. This way Proposition 1 implies that $\mathsf{M}(n)$ is also the complexity of the transpose of this multiplication. Restricting to linear straight-line programs is actually not embarrassing since all known multiplication algorithms [255] for the straight-line program model fit into this setting.

As in [255, §8.3] we assume that $\mathsf{M}(n_1 + n_2) \geq \mathsf{M}(n_1) + \mathsf{M}(n_2)$ for any positive integers $n_1$ and $n_2$. For the sake of simplicity we also assume that $n \log(n) \in O(\mathsf{M}(n))$.

### 4.6.1 Going up the subproduct tree

A common piece of the following algorithms is the computation of the *subproduct tree* $T$ with leaves $x - a_0, x - a_1, \ldots, x - a_m$. It is defined recursively, together with the sequence of integers $m_i$, by

$$T_{0,j} = x - a_j, \text{ for } j \in \{0, \ldots, m\}, \quad m_0 = m + 1,$$

and for $i \geq 1$ by

$$T_{i,j} = T_{i-1,2j} T_{i-1,2j+1}, \text{ for } j < h_i = \lfloor m_i/2 \rfloor .$$

If $m_i = 2h_i + 1$ we let $T_{i,h_i} = T_{i-1,m_i-1}$ and $m_{i+1} = h_i + 1$, otherwise we just let $m_{i+1} = h_i$. Let $d$ be the smallest integer such that $m_d = 1$; we get $T_{d,0} = \prod_{j=0}^{m}(x - a_j)$. By [255, Exercise 10.3], $T$ can be computed within $1/2\,\mathsf{M}(m)\log(m) + O(m \log(m))$ operations.

The following algorithm describes our basic use of $T$. On input $c = (c_0, \ldots, c_m)$, it computes the polynomial $b = \sum_{j=0}^{m} c_j \frac{T_{d,0}}{x - a_j}$.

```
UpTree(c)
  b ← c;
  for i ← 0 to d − 1 do
    for j ← 0 to h_i − 1 do
      b_j ← T_{i,2j+1} b_{2j} + T_{i,2j} b_{2j+1};
    if m_i = 2h_i + 1 then b_{h_i} ← b_{m_i−1};
  return b_0;
```

We deduce its transpose as follows:

```
TUpTree(b)
  c_0 ← b;
  for i ← d − 1 downto 0 do
    if m_i = 2h_i + 1 then c_{m_i−1} ← c_{h_i};
    for j ← h_i − 1 downto 0 do
      n ← deg(T_{i,j}) − 1;
      c_{2j+1} ← mul^t(n, T_{i,2j}, c_j);
      c_{2j} ← mul^t(n, T_{i,2j+1}, c_j);
  return c;
```

On input $b \in k[x]_m$, the transposed algorithm outputs the coefficients of $x^m$ in the polynomial products $\mathrm{rev}(m, b)\frac{T_{d,0}}{x-a_j}$, for $j = 0, \ldots, m$.

Once $T$ is computed, using Proposition 1 and [255, Th. 10.10], both algorithms require $\mathsf{M}(m)\log(m) + O(m\log(m))$ operations.

### 4.6.2 Multipoint evaluation

We first treat the transposed problem. Let $c_0, \ldots, c_m$ be in $k$. A direct computation shows that the entries of $b = V_a^t c$ are the first $m+1$ coefficients of the Taylor expansion of $S(x) =$

$$\sum_{j=0}^{m} \frac{c_j}{1-a_j x} = \frac{1}{\mathrm{rev}(m+1, T_{d,0})} \sum_{j=0}^{m} \frac{c_j \mathrm{rev}(m+1, T_{d,0})}{1-a_j x}.$$

The last sum is obtained by computing $\mathrm{UpTree}(c)$ and reversing the result. Computing the Taylor expansion of $S$ requires one additional power series inversion and one multiplication.

```
α ← 1/rev(m + 1, T_{d,0})  mod x^{m+1};
s ← UpTree(c);
t ← rev(m, s);
b ← mul(α, t)  mod x^{m+1};
```

We deduce the following algorithm for evaluating a polynomial $b$:

```
α ← 1/rev(m + 1, T_{d,0})  mod x^{m+1};
t ← mul^t(m, α, b);
s ← rev(m, t);
c ← TUpTree(s);
```

By the above complexity results, these algorithms require $3/2\,\mathsf{M}(m)\log(m) + O(\mathsf{M}(m))$ operations, since the additional operations have negligible cost. We gain a constant factor on the usual algorithm of repeated remaindering, of complexity $7/2\,\mathsf{M}(m)\log(m) + O(\mathsf{M}(m))$, see [171] or [255, Exercise 10.9]. We also gain on $13/6\,\mathsf{M}(m)\log(m) + O(\mathsf{M}(m))$ given in [171, §3.7] for base fields $k$ allowing Fast Fourier Transform in $k[x]$.

Moreover, if many evaluations at the same set of points $a_i$ have to be performed, then all data depending only on the evaluation points (the tree $T$ and $\alpha$) may be precomputed and stored, and the cost of evaluation drops to essentially $\mathsf{M}(m)\log(m)+O(\mathsf{M}(m))$. This improves [255, Exercise 10.11] by a factor of 2.

As a final remark, our interpretation of the transposed remainder shows that the algorithm in [223] for transposed evaluation is the exact transposition of the classical evaluation algorithm as given in [255, §10.1], so ours is faster. The algorithm of [49] is still slower by a constant factor, since it uses an interpolation routine.

### 4.6.3 Interpolation

The fast interpolation algorithm $c = V_a^{-1}b$ proceeds this way, see [255, §10.2]:

$p \leftarrow dT_{d,0}/dx$;
$z \leftarrow (p(a_0), \ldots, p(a_m))$;
$c \leftarrow (b_0/z_0, \ldots, b_m/z_m)$;
$c \leftarrow \mathrm{UpTree}(c)$;

Here, $z$ is computed using fast multipoint evaluation. At the end, $c$ contains the interpolating polynomial. Reversing the computation flow, we get the transposed algorithm for computing $b = (V_a^{-1})^t c$:

$p \leftarrow dT_{d,0}/dx$;
$z \leftarrow (p(a_0), \ldots, p(a_m))$;
$b \leftarrow \mathrm{TUpTree}(c)$;
$b \leftarrow (b_0/z_0, \ldots, b_m/z_m)$;

Using the results given above, these algorithms require $5/2\,\mathsf{M}(m)\log(m) + O(\mathsf{M}(m))$ operations. This improves again the complexity results of [255].
The algorithm of [127] for transposed interpolation does the same precomputation as ours, but the call to TUpTree is replaced by (mainly) a multipoint evaluation. Using precomputations and our result on evaluation, that algorithm also has complexity $5/2\,\mathsf{M}(m)\log(m) + O(\mathsf{M}(m))$.

## 4.7 Conclusion, future work

A first implementation of our improved evaluation algorithm gains a factor of about 1.2 to 1.5 over the classical one [255, §10.1], for degrees about 10000. But let us mention that the crossover point between repeating Horner's rule and the classical fast evaluation we have implemented is about 32. In consequence, it seems interesting to explore the constant factors hidden behind the above quantities $O(\mathsf{M}(m))$.

Our results of Section 4.6 can be generalized to solve the *simultaneous modular reduction* and the *Chinese remainder* problems faster than in [255, Chapter 10]. Theoretical and practical developments related to these problems are work in progress.

Transposed computations with Vandermonde matrices are the basis of fast multiplication algorithms for sparse and dense multivariate polynomials [19, 49, 267] and power series [248, 152]. Up to logarithmic factors, over a base field of characteristic zero, these multiplications have linear complexities in the size of the output. Their implementation is the subject of future work.

# Chapter 5

# Polynomial evaluation and interpolation on special sets of points

We give complexity estimates for the problems of evaluation and interpolation on various polynomial bases. We focus on the particular cases when the sample points form an arithmetic or a geometric sequence. We improve the known algorithms for evaluation and interpolation in the monomial basis in both the arithmetic and the geometric case. We discuss applications, respectively to computations with differential operators and symbolic summation, and polynomial matrix multiplication.

This chapter is joint work with É. Schost [38].

## Contents

## 5.1 Introduction

Let $k$ be a field and $x_0, \ldots, x_{n-1}$ be $n$ pairwise distinct points in $k$. Given arbitrary values $v_0, \ldots, v_{n-1}$, there exists a unique polynomial $F$ in $k[x]$ of degree less than $n$ such that $F(x_i) = v_i$, $i = 0, \ldots, n-1$. Having fixed a basis of the vector space of polynomials of degree at most $n-1$, interpolation and evaluation questions consist in computing the coefficients of $F$ on this basis from the values $(v_i)$, and conversely.

Well-known problems are that of interpolation and evaluation in the *monomial basis* $1, x, \ldots, x^{n-1}$:

- Given $x_0, \ldots, x_{n-1}$ and $v_0, \ldots, v_{n-1}$, *monomial interpolation* consists in determining the unique coefficients $f_0, \ldots, f_{n-1}$ such that the polynomial $F = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$ satisfies $F(x_i) = v_i$, for $i = 0, \ldots, n-1$.
- Given $x_0, \ldots, x_{n-1}$ and $f_0, \ldots, f_{n-1}$, *monomial evaluation* consists in computing the values $v_0 = F(x_0), \ldots, v_{n-1} = F(x_{n-1})$, where $F$ is the polynomial $f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$.

The *Newton basis* associated to the points $x_0, \ldots, x_{n-1}$ provides with an alternative basis of degree $n-1$ polynomials, which is defined as

$$1, (x - x_0), (x - x_0)(x - x_1), \ldots, (x - x_0) \cdots (x - x_{n-2}).$$

An important particular case is the *falling factorial basis*

$$1,\ x^{\underline{1}} = x,\ x^{\underline{2}} = x(x-1),\ x^{\underline{3}} = x(x-1)(x-2), \ldots,$$

which is a used in many algorithms for symbolic summation, see for instance [3, 193, 191].

Accordingly, Newton interpolation and Newton evaluation are defined as the problems of interpolation and evaluation with respect to the Newton basis:

- Given $x_0, \ldots, x_{n-1}$ and $v_0, \ldots, v_{n-1}$, *Newton interpolation* consists in determining the unique coefficients $f_0, \ldots, f_{n-1}$ such that the polynomial

$$F = f_0 + f_1(x - x_0) + f_2(x - x_0)(x - x_1) + \cdots + f_{n-1}(x - x_0) \cdots (x - x_{n-2}) \quad (5.1)$$

  satisfies $F(x_i) = v_i$, for $i = 0, \ldots, n-1$.
- Given $x_0, \ldots, x_{n-1}$ and $f_0, \ldots, f_{n-1}$, *Newton evaluation* consists in computing the values $v_0 = F(x_0), \ldots, v_{n-1} = F(x_{n-1})$, where $F$ is the polynomial given by Formula (5.1).

Fast algorithms for evaluation and interpolation in the monomial basis have been discovered in the seventies and have complexity in $O(\mathsf{M}(n)\log(n))$, where $\mathsf{M}(n)$ denotes the complexity of univariate polynomial multiplication in degree $n$. Using the FFT-based multiplication algorithms of [214, 210, 50], for which $\mathsf{M}(n)$ can be taken in $O(n\log(n)\log(\log(n)))$, this complexity is nearly optimal, up to logarithmic factors (note that naive algorithms are quadratic in $n$). These fast algorithms are nowadays classical topics covered by most of the computer algebra textbooks [25, 47, 255] and their practical relevance is recognized.

In contrast, fast algorithms for Newton evaluation and interpolation are quite recent, despite a potentially vast field of applications. The standard algorithms use divided differences and have a complexity quadratic in $n$ [133]. Yet, using a divide-and-conquer approach, the complexity of Newton evaluation and interpolation becomes essentially linear in $n$: the algorithms suggested in [25, Ex. 15, p. 67] have complexity in $O(\mathsf{M}(n)\log(n))$. Such fast algorithms rely on an additional task: the basis conversion between the monomial basis and the Newton basis. These conversion algorithms are detailed in Theorems 2.4 and 2.5 in [92].

**Our contribution.** The first goal of this chapter is to make explicit the constants hidden behind the Big-Oh notation in the running-time estimates for the three tasks mentioned up to now:

1. conversions between Newton and monomial bases,
2. monomial evaluation and interpolation,
3. Newton evaluation and interpolation.

Our second and main objective is to obtain better algorithms for special cases of evaluation points $x_i$. Indeed, it is well known that divided difference formulas simplify when the points form an *arithmetic* or a *geometric* progression; we will show how to obtain improved complexity estimates based on such simplifications. We also discuss applications, to computations with differential operators, symbolic summation and polynomial matrix multiplication.

Table 5.1 summarizes the best results known to us on the three questions mentioned above; we will now review its columns in turn. In what follows, all results of type $O(\mathsf{M}(n)\log(n))$ are valid when $n$ is a power of 2. The results for the arithmetic progression case require that the base field has characteristic 0 or larger than $n$.

**The general case.** The first column gives estimates for arbitrary sample points. In this case, the conversion algorithms and the monomial evaluation/interpolation algorithms are designed first, and Newton evaluation/interpolation algorithms are deduced by composition.

The results on monomial evaluation and interpolation appeared in [35] and improve the classical results by Borodin, Moenck [169, 30], Strassen [235] and Montgomery [171]. The other complexity results in that column were already known to lie in the class $O(\mathsf{M}(n)\log(n))$, see for instance [25]; the conversion algorithms were detailed in [92], who also gives their bit complexity when the base field is $\mathbb{Q}$. Our only contribution here is to apply an idea by Montgomery [171] to save a constant factor in the conversion from the monomial to the Newton basis.

**The arithmetic progression case.** In the arithmetic progression case, a sharp complexity statement was derived in [92, Th. 3.2 and Th. 3.4], which proved that Newton evaluation and interpolation at an arithmetic progression can be done within $\mathsf{M}(n) + O(n)$ operations. That article also analyzes the bit complexity when the base field is $\mathbb{Q}$.

| Question | general case |
|---|---|
| Newton to monomial basis (**NtoM**) | $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [25, 92] |
| Monomial to Newton basis (**MtoN**) | $5/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [25, 92] |
| Monomial evaluation (**MtoV**) | $3/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [35] |
| Monomial interpolation (**VtoM**) | $5/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [35] |
| Newton evaluation (**NtoV**) | $2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [25] |
| Newton interpolation (**VtoN**) | $9/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [25] |

| Question | arithmetic case | geometric case |
|---|---|---|
| **NtoM** | $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ [25] | $\mathsf{M}(n) + O(n)$ |
| **MtoN** | $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ | $\mathsf{M}(n) + O(n)$ |
| **MtoV** | $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ | $2\,\mathsf{M}(n) + O(n)$ [198, 27, 5] |
| **VtoM** | $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ | $2\,\mathsf{M}(n) + O(n)$ |
| **NtoV** | $\mathsf{M}(n) + O(n)$ [92] | $\mathsf{M}(n) + O(n)$ |
| **VtoN** | $\mathsf{M}(n) + O(n)$ [92] | $\mathsf{M}(n) + O(n)$ |

Table 5.1: Complexity results on conversions between Newton and monomial bases, monomial evaluation and interpolation, Newton evaluation and interpolation.

Using this result, we improve (by constant factors) many of the other tasks. First, using the Newton basis for intermediate computations, we obtain a new, faster algorithm for monomial interpolation in the arithmetic progression case. Using Tellegen's transposition theorem [239, 47, 124, 35] and an idea from [49], we deduce an improvement of the complexity of monomial evaluation as well. This finally enables us to perform the conversion from

the monomial basis to the Newton basis faster, using monomial evaluation and Newton interpolation as intermediate steps; this improves the result of [92, Th. 2.4] by a constant factor.

We apply these algorithms to conversions between the monomial and the falling factorial bases. This yields fast algorithms for computing Stirling numbers, which in turn are the basis for symbolic summation of polynomials and fast computation with linear differential operators. Also, we discuss the transpose of the algorithm of [92], which is shown to be closely related to an algorithm of [5] for polynomial shift.

**The geometric case.** In the geometric progression case, we show that the complexities of Newton evaluation and interpolation drop to $\mathsf{M}(n)+O(n)$ as well. The improvements are obtained by (mainly) translating into equalities of generating series the formulas for divided $q$-differences, similarly to what is done in [92] for the arithmetic case. By considering the transposed problems, we deduce that the conversions between the Newton and the monomial bases can be done with the same asymptotic complexity of $\mathsf{M}(n)+O(n)$ operations in the base field.

These results have consequences for evaluation and interpolation in the monomial basis. It is known [198, 27, 5] that evaluating a polynomial of degree less than $n$ on $n$ points in geometric progression has cost $2\mathsf{M}(n)+O(n)$ (the algorithm given in [5] actually has complexity more than $2\mathsf{M}(n)+O(n)$, but using the operation of middle product [115] yields the announced complexity bound).

An analogue result for the inverse problem — that is, interpolation on the monomial basis at points in geometric progression — was previously not known. Using the Newton basis for intermediate computations, we show that this can also be done using $2\mathsf{M}(n)+O(n)$ operations.

Thus, this allows to exhibit special sequences of points, lying in the base field, for which both evaluation and interpolation are cheaper by a logarithmic factor than in the general case. Many algorithms using evaluation and interpolation for intermediate computations can benefit from this. We exemplify this by improving the complexity of polynomial matrix multiplication.

**Organization of the chapter.** In Section 5.2 we recall and/or analyze known algorithms for evaluation and interpolation in the monomial and Newton bases, as well as conversion algorithms, in the general case of arbitrary sample points. The operations performed by these algorithms are linear in the entries of the input: Section 5.3 is devoted to present some transposed versions of these algorithms, which will be needed in the sequel.

In Section 5.4, we focus on the case when the sample points form an arithmetic sequence, and present applications to computations with differential operators and symbolic summation. Section 5.5 is devoted to the special case of evaluation points in a geometric progression; we conclude this section by an application to polynomial matrices multiplication.

We suppose that the *multiplication time* function $\mathsf{M}$ verifies the inequality $\mathsf{M}(d_1)+\mathsf{M}(d_2) \leq \mathsf{M}(d_1+d_2)$ for all positive integers $d_1$ and $d_2$; in particular, the inequality $\mathsf{M}(d/2) \leq 1/2\,\mathsf{M}(d)$ holds for all $d \geq 1$. We also make the hypothesis that $\mathsf{M}(cd)$ is in $O(\mathsf{M}(d))$, for all $c > 0$. The basic examples we have in mind are *classical* multiplication, for which $\mathsf{M}(n) \in O(n^2)$, Karatsuba's multiplication [129] with $\mathsf{M}(n) \in O(n^{1.59})$ and the FFT-based multiplication [214, 210, 50], which have $\mathsf{M}(n) \in O(n\log(n)\log(\log(n)))$. Our references for matters related to polynomial arithmetic are the books [25, 255].

## 5.2 The general case

In this section, we treat the questions of monomial and Newton evaluation and interpolation, and conversion algorithms, for an arbitrary choice of sample points $(x_0, \ldots, x_{n-1})$. In what follows, we suppose that $n$ is a power of 2.

We first introduce the *subproduct tree* $\mathcal{T}$ associated to the points $x_0, \ldots, x_{n-1}$, which is used in all the sequel. In Subsections 5.2.2 and 5.2.3 we recall fast algorithms for evaluation and interpolation in the monomial basis and for conversions between monomial and Newton bases. In Subsection 5.2.4 we deduce fast algorithms for Newton evaluation and interpolation.

Again, let us mention that the algorithms involving the Newton basis were already suggested or described in [25, 189, 92]. Yet, our more precise complexity analysis is needed later.

### 5.2.1 The subproduct tree

The subproduct tree is a binary tree, all whose nodes contain polynomials. Our assumption that $n$ is a power of 2 makes the definition of this tree straightforward: let $n = 2^m$; then the tree $\mathcal{T}$ associated to a sequence $x_0, \ldots, x_{2^m-1} \in k^{2^m}$ is defined as follows:

- If $m = 0$, $\mathcal{T}$ reduces to a single node, containing the polynomial $x - x_0$.
- For $m > 0$, let $\mathcal{T}_0$ and $\mathcal{T}_1$ be the trees associated to respectively $x_0, \ldots, x_{2^{m-1}-1}$ and $x_{2^{m-1}}, \ldots, x_{2^m-1}$. Let $M_0$ and $M_1$ be the polynomials at the roots of $\mathcal{T}_0$ and $\mathcal{T}_1$. Then $\mathcal{T}$ is the tree whose root contains the product $M_0M_1$ and whose children are $\mathcal{T}_0$ and $\mathcal{T}_1$.

Alternately, one can represent the subproduct tree as a 2-dimensional array $T_{i,j}$, with $0 \leq i \leq m$, $0 \leq j \leq 2^{m-i} - 1$. Then

$$T_{i,j} = \prod_{k=2^ij}^{2^i(j+1)-1} (x - x_k).$$

For instance, if $m = 2$ (and thus $n = 4$), the tree associated to $x_0, x_1, x_2, x_3$ is given by

$$T_{0,0} = x - x_0,\ T_{0,1} = x - x_1,\ T_{0,2} = x - x_2,\ T_{0,3} = x - x_3$$

$$T_{1,0} = (x - x_0)(x - x_1),\ T_{1,1} = (x - x_2)(x - x_3)$$

$$T_{2,0} = (x - x_0)(x - x_1)(x - x_2)(x - x_3)$$

In terms of complexity, the subproduct tree associated to $x_0, \ldots, x_{n-1}$ can be computed within $1/2\,\mathsf{M}(n)\log(n) + O(n\log(n))$ base field operations, see [255].

### 5.2.2 Evaluation and interpolation on the monomial basis

Let $F \in k[x]$ be a polynomial of degree less than $n$, let $x_0, \ldots, x_{n-1}$ be $n$ pairwise distinct points in $k$ and denote $v_i = F(x_i)$. The questions of multipoint evaluation and interpolation in the monomial basis consists in computing the coefficients of $F$ in the monomial representation from the values $(v_i)$, and conversely.

Fast algorithms for these tasks were given by Borodin and Moenck [169, 30], then successively improved by Strassen [235] and Montgomery [171]. All these algorithms are based on (recursive) polynomial remaindering and have complexity $O(\mathsf{M}(n)\log(n))$. Recently, different algorithms, based on the (recursive) use of transposed operations, have been designed in [35, Section 6] and led to improved complexity bounds, by constant factors. For the sake of completeness, we summarize the corresponding results in [35] in the theorem below:

**Theorem 2** *Let $k$ be a field, let $x_0, \ldots, x_{n-1}$ be pairwise distinct elements of $k$ and let $F \in k[x]$ of degree less than $n$. Suppose that $n$ is a power of 2 and that the subproduct tree associated to the points $(x_i)$ has been precomputed. Then:*

- *the evaluation of $F$ at the points $(x_i)$ can be done using $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*
- *the interpolation of $F$ at the points $(x_i)$ can be done using $2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*

Taking into account the complexity of computing the subproduct tree, which is within $1/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations, we obtain the estimates given in the first column, middle rows, in Table 5.1.

**Remark.** Interpolation requires to evaluate the derivative of $\prod_{i=0}^{n-1}(x - x_i)$ on all points $x_i$, which contributes for $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ in the above estimate. In the case of an arithmetic or a geometric progression, these values can be computed in linear time, see for instance [34], so the complexity of interpolation drops to

$$(1/2 + 1)\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n)) = 3/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$$

in these cases. In Sections 5.4 and 5.5 we show that one can actually do better in these two special cases.

### 5.2.3 Conversions between Newton basis and monomial basis

We now estimate the complexity for the conversions between the monomial and the Newton bases. The results are summarized in the theorem below:

**Theorem 3** *Let $k$ be a field, let $x_0, \ldots, x_{n-1}$ be pairwise distinct elements of $k$ and let $F \in k[x]$ of degree less than $n$. Suppose that $n$ is a power of 2 and that the subproduct tree associated to the points $(x_i)$ has been precomputed. Then:*

- *given the coefficients of $F$ in the Newton basis, one can recover the coefficients of $F$ in the monomial basis using $1/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*
- *given the coefficients of $F$ in the monomial basis, one can recover the coefficients of $F$ in the Newton basis using $2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*

Taking into account the complexity of computing the subproduct tree, we obtain the estimates given in the first column, first two rows, in Table 5.1.

**From Newton basis to monomial basis.** Let $F$ be a polynomial of degree less than $n$ and $f_0, \ldots, f_{n-1}$ its coefficients in the Newton basis. Given the coefficients $(f_i)$, we want to recover the coefficients of $F$ in the monomial basis. To this effect, we write the equality

$$\begin{aligned} F &= F_0 + (x - x_0)\cdots(x - x_{n/2-1})F_1, \quad \text{with} \\ F_0 &= f_0 + f_1(x - x_0) + \cdots + f_{n/2-1}(x - x_0)\cdots(x - x_{n/2-2}), \\ F_1 &= f_{n/2} + f_{n/2+1}(x - x_{n/2}) + \cdots + f_{n-1}(x - x_{n/2})\cdots(x - x_{n-2}). \end{aligned}$$

Using this decomposition, the following recursive conversion algorithm can be deduced. On input the coefficients $f_0, \ldots, f_{n-1}$ and the points $x_0, \ldots, x_{n-1}$, it outputs the coefficients of $F$ on the monomial basis.

**NewtonToMonomial**$(f_0, \ldots, f_{n-1}, x_0, \ldots, x_{n-1})$

- if $n = 1$ return $f_0$.
- let $F_0 = \text{NewtonToMonomial}(f_0, \ldots, f_{n/2-1}, x_0, \ldots, x_{n/2-1})$.
- let $F_1 = \text{NewtonToMonomial}(f_{n/2}, \ldots, f_{n-1}, x_{n/2}, \ldots, x_{n-1})$.
- return $F_0 + (x - x_0)\cdots(x - x_{n/2-1})F_1$.

Let us assume that the subproduct tree associated to the points $x_0, \ldots, x_{n-1}$ has been precomputed. Since all polynomials of the form $(x - x_0)\cdots(x - x_{n/2-1})$ that are used in the recursive steps of the algorithms belong to the subproduct tree, no additional operation is required to obtain them. Denoting by $\mathsf{C}(n)$ the number of necessary operations on inputs of size $n$ and using the inequality $\mathsf{M}(n/2) \leq 1/2\,\mathsf{M}(n)$, we obtain the following estimate:

$$\mathsf{C}(n) \leq 2\,\mathsf{C}\left(\frac{n}{2}\right) + \frac{1}{2}\,\mathsf{M}(n) + O(n),$$

so Lemma 8.2 in [255] yields $\mathsf{C}(n) = 1/2\,\mathsf{M}(n)\log(n) + O(n\log(n))$, proving the first part of Theorem 3. Again, the asymptotic estimate $O(\mathsf{M}(n)\log(n))$ was already obtained in [25, Ex. 15, p. 67] and [92, Th. 2.4], but the more precise form given here is needed in the sequel.

**From monomial basis to Newton basis.** We now treat the inverse problem. Let $F$ be a polynomial of degree less than $n$, whose coefficients on the monomial basis are known; we want to recover the coefficients of $F$ on the Newton basis $\{1, x-x_0, \ldots, (x-x_0)\cdots(x-x_{n-2})\}$. The key fact is that if $F$ writes

$$F = F_0 + (x - x_0)\cdots(x - x_{n/2-1})F_1,$$

then it is enough to recover the coefficients of $F_0$ and $F_1$ on the Newton bases $\{1, x - x_0, \ldots, (x - x_0)\cdots(x - x_{n/2-2})\}$ and $\{1, x - x_{n/2}, \ldots, (x - x_{n/2})\cdots(x - x_{n-2})\}$ respectively.

Using this remark, we deduce as above a recursive conversion algorithm. It takes as input the coefficients of $F$ in the monomial basis and it returns the coefficients of $F$ on the Newton basis associated to the points $x_0, \ldots, x_{n-1}$.

**MonomialToNewton**$(F,\, x_0, \ldots, x_{n-1})$

- if $n = 1$ return $F$.
- let $\mathcal{L}_1 = \text{MonomialToNewton}\big(F \mod (x - x_0)\cdots(x - x_{n/2-1}),\, x_0, \ldots, x_{n/2-1}\big)$.
- let $\mathcal{L}_2 = \text{MonomialToNewton}\big(F \text{ div } (x - x_0)\cdots(x - x_{n/2-1}),\, x_{n/2}, \ldots, x_{n-2}\big)$.
- return $\mathcal{L}_1 \cup \mathcal{L}_2$.

The cost analysis is similar to that in the previous subsection. Let us assume that the subproduct tree associated to the points $x_0, \ldots, x_{n-1}$ has been precomputed. Since at all recursion levels we do division (with quotient and remainder) by (the half of) the polynomials belonging to this subproduct tree, we use Montgomery's idea [171, p. 22-24], to save constant factors. Thus, we precompute the power series inverses of the polynomials $\text{rev}(T_{i,j})$ at precision $2^i$, for a cost of $\mathsf{M}(n)\log(n) + O(n\log(n))$ operations, see [255, Ex. 10.9, (i)].

This way, the cost of the division by $(x - x_0)\cdots(x - x_{n/2-1})$ becomes $2\,\mathsf{M}(n/2) + O(n)$. Denoting by $\mathsf{C}(n)$ the number of operations required by the previous algorithm on inputs of size $n$ (without counting precomputations) and using the inequality $2\,\mathsf{M}(n/2) \leq \mathsf{M}(n)$, we obtain:

$$\mathsf{C}(n) \leq 2\,\mathsf{C}\left(\frac{n}{2}\right) + \mathsf{M}(n) + O(n),$$

so $\mathsf{C}(n) = \mathsf{M}(n)\log(n) + O(n\log(n))$, by [255, Lemma 8.2]. This concludes the proof of Theorem 3.

### 5.2.4 Newton evaluation and interpolation

Combining the results of Sections 5.2.2 and 5.2.3, we deduce the following corollary concerning the complexities of Newton evaluation and interpolation on an arbitrary set of evaluation points.

**Theorem 4** *Let $k$ be a field, let $x_0, \ldots, x_{n-1}$ be pairwise distinct elements of $k$ and let $F \in k[x]$ of degree less than $n$. Suppose that $n$ is a power of 2 and that the subproduct tree associated to the points $(x_i)$ has been precomputed. Then:*

- *Newton evaluation of $F$ at the points $(x_i)$ can be done using $3/2\,\mathsf{M}(n)\log(n)+O(\mathsf{M}(n))$ operations in $k$.*
- *Newton interpolation of $F$ at the points $(x_i)$ can be done using $4\,\mathsf{M}(n)\log(n)+O(\mathsf{M}(n))$ operations in $k$.*

Taking into account the complexity of computing the subproduct tree, this completes the entries of the first column of Table 5.1.

## 5.3 Transposed conversion algorithms

The conversion algorithms described in the previous section all compute base change maps, which are linear in $F$. For instance, the algorithm **NewtonToMonomial** computes the matrix-vector product between a matrix built on the points $x_i$ and a vector whose entries are the coefficients $(f_i)$ of $F$. In this section we are interested in computing the transpose of this map. The results presented here are used in Section 5.4.

An algorithmic theorem called the *Transposition principle*, or *Tellegen's principle* [239] states that given any algorithm that performs a $(m+n)\times m$ matrix-vector product using only linear operations, one can deduce an algorithm that performs the transposed matrix-vector product, with the same complexity, up to $O(n)$: see [47] for a precise statement and [124] for historical notes and further comments on this question.

Fast algorithms for transposed monomial evaluation and interpolation are presented in [35]. We now inspect the transpose of the algorithm **NewtonToMonomial** given in the preceding section, since this will be used in the next section.

Computing the subproduct tree associated to the points $(x_i)$ is a precomputation, which does not depend on the polynomial $F$, and is left unchanged by transposition. As to the recursive part of the algorithm, we need to introduce the *transposed multiplication*: Let us denote by $k[x]_i$ the vector space of polynomials of degree at most $i$; then given $a\in k[x]$ of degree $m$, we denote by $\mathrm{mul}^t(n,a,.) : k[x]_{m+n}\to k[x]_n$ the transpose of the multiplication-by-$a$ map $k[x]_n\to k[x]_{m+n}$.

Various algorithms for computing the transposed multiplication are detailed in [115, 35]. Tellegen's principle implies that whatever the algorithm used for polynomial multiplication is, the cost of the direct and of the transposed multiplication are equal, up to $O(m)$ operations in $k$; for instance, if $m=n$, the complexity of $\mathrm{mul}^t(n,a,.)$ is $\mathsf{M}(n)+O(n)$.

Using this operation, we obtain by a mechanical transformation the following transposed conversion algorithm. The direct version takes a list of coefficients as input and gives its output in the form of a polynomial; the transposed algorithm takes a polynomial as input and outputs a list of coefficients.

**TNewtonToMonomial**$(F=\sum_{i=0}^{n-1} f_i x^i, x_0,\dots,x_{n-1})$

- if $n=1$ return $[f_0]$.
- let $A=\mathrm{mul}^t\left(n/2-1,(x-x_0)\cdots(x-x_{n/2-1}),F\right)$.

- let $[B_0, \ldots, B_{n/2-1}] = \text{TNewtonToMonomial}(F \bmod x^{n/2}, x_0, \ldots, x_{n/2-1})$.
- let $[C_0, \ldots, C_{n/2-1}] = \text{TNewtonToMonomial}(A, x_0, \ldots, x_{n/2-1})$.
- return $[B_0, \ldots, B_{n/2-1}, C_0, \ldots, C_{n/2-1}]$.

It follows from either a direct analysis or the transposition principle that, including the precomputation of the subproduct tree, this algorithm requires $\mathsf{M}(n)\log(n) + O(n\log(n))$ base field operations. If the subproduct tree is already known, the complexity drops to $1/2\,\mathsf{M}(n)\log(n) + O(n\log(n))$.

The same transformation techniques can be applied in order to exhibit the transposes of all the algorithms in Section 5.2, within the same costs as the direct algorithms, up to linear factors in $n$. Since we do not need these transposed algorithms in the sequel, we will not give further details and leave to the reader the formative task of deriving them by herself.

## 5.4 Special case of an arithmetic progression

In this section we focus on the special case of evaluation points in arithmetic progression, and show that many of the above complexity estimates can be improved in this case.

We begin by recalling a result taken from [92, Section 3], which shows that the complexities of Newton evaluation and interpolation drop to $\mathsf{M}(n) + O(n)$ in this case, and we point out the link between these algorithms and the algorithm for shift of polynomials of [5]. Next, using the transposed algorithm of Section 5.3, we show how to improve (by constant factors) the complexities of evaluation and interpolation in the *monomial* basis on an arithmetic progression. Finally, we give estimates for the cost of conversions between Newton and monomial bases, and present some applications.

**Newton interpolation and evaluation.** We first recall the algorithm of [92, Section 3]: this gives the last two entries of the second column, in Table 5.1.

**Proposition 1** *Suppose that $k$ is a field of characteristic 0 or larger than $n$. Let $h$ be a non-zero element in $k$. Then, Newton interpolation and evaluation of a polynomial of degree $n$ on the arithmetic sequence $x_i = x_0 + ih$, for $i = 0, \ldots, n-1$ can be done using $\mathsf{M}(n) + O(n)$ operations in $k$.*

*Proof.* Let $F$ be a polynomial of degree less than $n$, $(v_i)$ the values $(F(x_i))$ and $(f_i)$ the coefficients of $F$ on the Newton basis associated to the points $(x_i)$. Evaluating Formula (5.1) at $x_i$, we deduce the following equalities relating the values $v_i$ and the coefficients $f_i$:

$$\begin{aligned} v_0 &= f_0 \\ v_1 &= f_0 + hf_1 \\ v_2 &= f_0 + 2hf_1 + (2h \cdot h)f_2 \\ v_3 &= f_0 + 3hf_1 + (3h \cdot 2h)f_2 + (3h \cdot 2h \cdot h)f_3 \quad \ldots \end{aligned}$$

This suggests to introduce the auxiliary sequence $w_i$ defined by

$$w_i = \frac{v_i}{i!h^i}, \quad i = 0, \ldots, n-1. \tag{5.2}$$

Note that the sequences $(v_i)_{i \leq n-1}$ and $(w_i)_{i \leq n-1}$ can be deduced from one another for $O(n)$ base field operations. Using the sequence $(w_i)_{i \leq n-1}$, the above relations become

$$w_i = \sum_{j+k=i} \frac{1}{h^k k!} f_j.$$

Introducing the generating series

$$W = \sum_{i=0}^{n-1} w_i x^i, \quad F = \sum_{i=0}^{n-1} f_i x^i, \quad S = \sum_{i=0}^{n-1} \frac{1}{i!h^i} x^i, \tag{5.3}$$

all above relations are summarized in the equation $W = FS$ modulo $x^n$. Since $S$ is the truncation of $\exp(x/h)$, its inverse $S^{-1}$ is the truncation of $\exp(-x/h)$, so multiplying or dividing by $S$ modulo $x^n$ can be done in $\mathsf{M}(n) + O(n)$ base field operations. We deduce that $W$ and $F$ can be computed from one another using $\mathsf{M}(n) + O(n)$ base field operations. This proves the proposition. □

Let us make a few comments regarding the previous algorithm. The problem of Newton evaluation on the arithmetic sequence $x_i = x_0 + ih$ is closely related to that of Taylor shift by $1/h$. More precisely, the matrix $\mathrm{Newton}_h$ of Newton evaluation is equal, up to multiplication by diagonal matrices, to the transpose of the matrix $\mathrm{Shift}_{1/h}$ representing the map $F(x) \mapsto F(x + 1/h)$ in the monomial basis. Indeed, the following matrix equality is easy to infer:

$$\mathrm{Newton}_h = \mathrm{Diag}\Big(1, h, h^2, \ldots, h^{n-1}\Big) \cdot \mathrm{Shift}_{1/h}^t \cdot \mathrm{Diag}\Big(0!, 1!, \ldots, (n-1)!\Big). \tag{5.4}$$

In the same vein, one can also interpret Newton interpolation as the transpose of Taylor shift by $-1/h$ (up to diagonal matrices). A simple way to see this is to take the inverse of Equation (5.4) and to use the equality between $\mathrm{Shift}_{1/h}^{-1}$ and $\mathrm{Shift}_{-1/h}$. In what follows, we also need a transposed version of Newton interpolation. Transposing and taking the inverse in Equation (5.4), we obtain the equation

$$\mathrm{Newton}_h^{-t} = \mathrm{Diag}\Big(1, h, h^2, \ldots, h^{n-1}\Big)^{-1} \cdot \mathrm{Shift}_{-1/h} \cdot \mathrm{Diag}\Big(0!, 1!, \ldots, (n-1)!\Big)^{-1}. \tag{5.5}$$

Now, a classical algorithm of Aho, Steiglitz and Ullman [5] solves the Taylor shift problem within $\mathsf{M}(n) + O(n)$ operations. Given a degree $n-1$ polynomial $F(x) = \sum_{i=0}^{n-1} f_i x^i$, the algorithm in [5] computes the coefficients of $\mathrm{Shift}_{1/h}(F) = F(x + 1/h)$ by exploiting Taylor's formula

$$\mathrm{Shift}_{1/h}(F) = \sum_{j=0}^{n-1} F^{(j)}(1/h) \, \frac{x^j}{j!}$$

and the fact that $F^{(j)}(1/h)$ is the coefficient of $x^{n-j-1}$ in the product

$$\left(\sum_{i=0}^{n-1} i!\, f_i x^{n-i-1}\right) \cdot \left(\sum_{i=0}^{n-1} \frac{x^i}{i!\, h^i}\right).$$

In view of Equation (5.4), it is actually immediate to show that the algorithm for Newton evaluation on an arithmetic progression presented in Proposition 1 can be interpreted as the *transposition* of the algorithm in [5] (up to diagonal matrices multiplications) and thus could have been deduced automatically from that algorithm using the effective transposition tools in [35].

**Evaluation and interpolation on the monomial basis.** A surprising consequence of Proposition 1 is the following improvement on the complexity of evaluation and interpolation on the *monomial basis*, for an arithmetic sequence. Indeed, the following corollary shows that in this case, one can speed up both evaluation and interpolation using the Newton basis for intermediate computations. This gives the middle entries of the second column in Table 5.1.

**Corollary 1** *Let $n$ be a power of 2 and let $k$ be a field of characteristic 0 or larger than $n$. Let $F \in k[x]$ of degree less than $n$ and let $x_0, \ldots, x_{n-1}$ be an arithmetic progression in $k$. Then:*

- *Given the coefficients of $F$ on the monomial basis, all values $F(x_0), \ldots, F(x_{n-1})$ can be computed in $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ base field operations.*
- *Given the values $F(x_0), \ldots, F(x_{n-1})$, all coefficients of $F$ on the monomial basis can be computed in $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ base field operations.*

*Proof.* We first treat the interpolation. To this effect, we first perform a Newton interpolation, which takes $\mathsf{M}(n)+O(n)$ base field operations by Proposition 1. Then, as in Section 5.2.3, we compute the subproduct tree associated to the sample points $(x_i)$, and apply the conversion algorithm **NewtonToMonomial** to deduce the coefficients of $F$ on the monomial basis. The estimates of Sections 5.2.1 and 5.2.3 give the complexity bound $1/2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ for both the subproduct tree step and for the conversion step. Summing up, this entails a cost of $\mathsf{M}(n)\log(n)+O(\mathsf{M}(n))$. Let us show that in the present arithmetic setting, a factor of $1/4\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ can be saved, thus yielding the announced running time bound.

This improvement comes from the observation that in the conversion step, only the polynomials $T_{i,2j}$ from the subproduct tree are necessary. We show now that under the hypotheses of Corollary 1, we are able to build the subproduct tree for roughly halving the cost of precomputing all its nodes. The crucial point is that at every level $i$, the nodes $T_{i,j}$ can be deduced from one another by performing a polynomial shift. Thus, our strategy is to first determine the left-hand nodes $T_{i,0}$, then to obtain the desired nodes by polynomial shifts performed at every level $i$.

The left-hand nodes $T_{i,0}$, for $0 \le i \le m-1$, can all be determined by a recursive procedure based on the equality

$$T_{i+1,0}(x) = T_{i,0}(x) \cdot T_{i,0}(x + 2^i h), \quad 0 \le i \le m-2.$$

Using the algorithm for the polynomial shift in [5] (recalled on page 103), this can be done within $O(\mathsf{M}(n))$ operations in $k$.

Starting from the $T_{i,0}$, the remaining desired nodes can be determined using the equality

$$T_{i,2j}(x) \; = \; T_{i,0}(x + 2^{i+1} j h), \quad 0 \le i \le m-2, \; 0 \le j \le 2^{m-i-1},$$

by performing one shift in degree $n/4$, two shifts in degree $n/8$, etc., thus for a total cost of $1/4\,\mathsf{M}(n)\log(n) + O(n\log(n))$ operations in $k$. This finishes the proof of the first part of Corollary 1.

Let us turn to evaluation. Let $V$ be the Vandermonde matrix associated to the points $x_0, \ldots, x_{n-1}$ and $V^t$ its transpose. Following [49], we use the reduction of a multipoint evaluation to a transposed evaluation. This is done with the help of the equality

$$(V^t)V = H, \quad \text{with} \quad H_{i,j} = \sum_{k=0}^{n-1} x_k^{i+j-2},$$

which rewrites as $V = (V^t)^{-1} H$. Let us see how to derive from this equality an algorithm for multipoint evaluation, whose cost is within the requested complexity bound.

First, we compute the subproduct tree associated to the points $(x_i)$, so in particular we have at our disposal the polynomial $P = (x - x_0) \cdots (x - x_{n-1})$. The matrix $H$ is the $n \times n$ Hankel matrix whose entries are the Newton sums of $P$; since $P$ is known, they can be computed in time $O(\mathsf{M}(n))$, see [212]. Then the product by $H$ also has complexity $O(\mathsf{M}(n))$, see [25] (actually, the cost is precisely $\mathsf{M}(n) + O(n)$, see [115]).

We next consider the multiplication by $(V^t)^{-1}$. The above algorithm for interpolation performs the multiplication by $V^{-1}$ using Newton interpolation followed by the conversion algorithm of Section 5.2.3. Transposing it is immediate: we first apply the transposed conversion algorithm **TNewtonToMonomial** of Section 5.3, followed by the transposed Newton interpolation mentioned after Proposition 1. The complexity estimates from Sections 5.2.1, Section 5.3 and Proposition 1 conclude the proof. □

**Conversions between monomial and Newton bases.** A straightforward consequence of Proposition 1 and Corollary 1 is an improvement of conversion from the monomial to the Newton basis in the special case of an arithmetic progression; this presents an improvement on [92, Th. 2.4] by a constant factor. In the converse direction, we use the same algorithm as in the general case, which, we recall, was already given in [92, Th. 2.5]. This completes the second column of Table 5.1.

**Corollary 2** *Let $n$ be a power of 2, let $k$ be a field of characteristic 0 or larger that $n$ and let $x_0, \ldots, x_{n-1}$ be an arithmetic progression in $k$. Then the conversions between the monomial and Newton bases associated to the points $(x_i)$ can be done using $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*

*Proof.* The conversion from the monomial to the Newton basis is done by an evaluation on the points $(x_i)$ followed by a Newton interpolation. □

**Applications.** Our initial interest in improving evaluation and interpolation on the points of an arithmetic progression was motivated by the study of linear recurrences with polynomial coefficients presented in [59, 34]: the algorithms therein can benefit from any improvement on evaluation and interpolation on an arithmetic progression. The cryptographic-sized record obtained in [34] requires to work in degree several tens of thousands, and gaining even a constant factor is interesting in such sizes.

We conclude this section by describing another two applications of Corollary 2. The first one comes from the domain of exact computations with linear differential operators, while the second one is a basic item in effective difference algebra.

While computing with linear differential operators, it is sometimes easier to work with the derivation $\delta = x\frac{d}{dx}$ instead of the usual derivation $D = \frac{d}{dx}$. For instance, the coefficients of a power series solution $\sum_{i\geq 0} s_i x^i$ of a linear differential operator $\mathcal{L}$ satisfy a linear recurrence, whose coefficients *can be read off* the coefficients of $\mathcal{L}$ when it is written in $\delta$. More precisely, if $\mathcal{L} = \sum_{i=0}^{n} p_i(x)\delta^i$ has coefficients $p_i(x) = \sum_{j=0}^{m} p_{ij}x^j$, then letting $\widetilde{p_j}(x) = \sum_{i=0}^{n} p_{ij}x^i$ for $0 \leq j \leq m$, the recurrence satisfied by the $s_i$ writes

$$\widetilde{p_m}(i)s_i + \cdots + \widetilde{p_0}(i+m)s_{i+m} = 0, \qquad \text{for all } i \geq 0.$$

Converting from the representation in $\delta$ to that in $D$, or backwards, amounts to compute several matrix-vector products $Sv$ or $S^{-1}v$, where $S$ is the $n \times n$ matrix

$$S = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & S_{1,n} \\ 0 & 1 & 3 & 7 & \dots & S_{2,n} \\ 0 & 0 & 1 & 6 & \dots & S_{3,n} \\ 0 & 0 & 0 & 1 & \dots & S_{4,n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & S_{n,n} \end{bmatrix}.$$

Indeed, one has the equalities, seemingly known by Stirling [230], see also [201]:

$$\begin{aligned} \delta^1 &= xD, \\ \delta^2 &= xD + x^2D^2, \\ \delta^3 &= xD + 3\,x^2D^2 + x^3D^3, \\ \delta^4 &= xD + 7\,x^2D^2 + 6\,x^3D^3 + x^4D^4, \\ \delta^5 &= xD + 15\,x^2D^2 + 25\,x^3D^3 + 10\,x^4D^4 + x^5D^5 \dots \end{aligned}$$

Writing $\mathcal{L} = \sum_{i=0}^{n} p_i(x)\delta^i$ in matrix form, as the product

$$\begin{bmatrix} 1 & x & \dots & x^m \end{bmatrix} \cdot \begin{bmatrix} p_{00} & p_{10} & \cdots & p_{n0} \\ \vdots & \vdots & \vdots & \vdots \\ p_{0m} & p_{1m} & \cdots & p_{nm} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \delta \\ \vdots \\ \delta^n \end{bmatrix}$$

the previous equalities show that rewriting $\mathcal{L}$ in $D$ amounts to $m$ matrix-vector products by the Stirling matrix $S$.

Conversely, let $\mathcal{L} = \sum_{i=0}^{n} q_i(x) D^i$ be a linear differential operator of order $n$ in $D$, whose coefficients $q_i(x) = \sum_{j=0}^{k} q_{ji} x^j$ are polynomials of degree at most $k$. Expressing $\mathcal{L}$ in the matricial form:

$$\mathcal{L} = \begin{bmatrix} x^{-n} & x^{-(n-1)} & \dots & x^k \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & \dots & q_{n0} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & q_{10} & \dots & q_{nk} \\ q_{00} & q_{11} & \dots & \vdots \\ \vdots & \vdots & & \\ \vdots & q_{1k} & & \vdots \\ q_{0k} & 0 & \dots & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ xD \\ \vdots \\ x^n D^n \end{bmatrix}$$

shows that the problem of expressing $\mathcal{L}$ as an operator with rational function coefficients in $\delta$ is reduced to the $n + k$ matrix-vector multiplications by the inverse of the Stirling matrix.

The entries $(S_{i,j})$ of the matrix $S$ are the *Stirling numbers of the second kind*; they satisfy the recurrence $S_{i,j+1} = S_{i-1,j} + iS_{i,j}$. These numbers also represent the coordinates of ordinary powers $1, x, \dots, x^{n-1}$ in the falling factorial basis $1, x^{\underline{1}} = x, \dots, x^{\underline{n-1}} = x(x-1)\cdots(x-n+2)$. For instance, for $j = 1, \dots, 5$ these relations write

$$\begin{aligned} x^1 &= x^{\underline{1}}, \\ x^2 &= x^{\underline{1}} + x^{\underline{2}}, \\ x^3 &= x^{\underline{1}} + 3\,x^{\underline{2}} + x^{\underline{3}}, \\ x^4 &= x^{\underline{1}} + 7\,x^{\underline{2}} + 6\,x^{\underline{3}} + x^{\underline{4}}, \\ x^5 &= x^{\underline{1}} + 15\,x^{\underline{2}} + 25\,x^{\underline{3}} + 10\,x^{\underline{4}} + x^{\underline{5}} \quad \dots \end{aligned}$$

Hence, the entries of the vector $Sv$ represent the coefficients of the polynomial $\sum_{i=0}^{n-1} v_i x^i$ in the Newton basis $1, x^{\underline{1}}, x^{\underline{2}}, \dots$ Similarly, computing $S^{-1}v$ amounts to converting a polynomial from its Newton representation (in the falling factorial basis) to the monomial one. Classically, these conversions are done by a direct quadratic algorithm, but using Corollary 2, they can be done in complexity $\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$.

This remark yields a fast algorithm to convert a differential equation of order $n$ in $D$ with polynomial coefficients of degree at most $m$ to the recurrence satisfied by a power series solution. By the previous considerations, its complexity is $O(\mathsf{M}(n)\max(n, m)\log(n))$. In comparison, the classical method uses the recurrence

$$\sum_{j=0}^{n} \sum_{k=0}^{m} p_{k,j}(i-k+1)(i-k+2)\cdots(i-k+j)s_{n+j-k} = 0$$

and amounts to compute the $mn$ polynomials $(x-k+1)(x-k+2)\cdots(x-k+j)$, which can be done in complexity of $O(mn\,\mathsf{M}(n))$. If $m$ and $n$ are of the same order, our method saves a factor of $n$.

The previous considerations on the Stirling matrix also yields a fast algorithm for the definite summation problem for polynomials. This should be compared with the classical quadratic algorithm described in [75] which was probably used by Faulhaber [134] in the 17th century, see also [255, Section 23.1]. We precise this result in the following corollary.

**Corollary 3** *Let $n$ be a power of 2, $k$ a field of characteristic zero or larger than $n$, $g$ a polynomial in $k[x]$, of degree less than $n$ and $m$ a positive integer. The coefficients of the definite sum $\sum_{i=0}^{m} g(i)$ (as a polynomial in $m$) can be determined using $2\,\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $k$.*

*Proof.* Let $S$ be the $(n+1) \times (n+1)$ Stirling matrix and let $T$ be the $(n+1) \times (n+1)$ matrix obtained by bordering the $n \times n$ Stirling matrix with first row $[1, 0, \ldots, 0]$ and first column $[1, 0, \ldots, 0]^t$. If $\widetilde{g}$ denotes the $(n+1) \times 1$ matrix whose entries are the coefficients of $g$, then the required coefficients of $\sum_{i=0}^{m} g(i)$ are the entries of the vector $S^{-1}DT\widetilde{g}$, where $D$ is the diagonal matrix with diagonal entries $[1, 1/2, \ldots, 1/(n+1)]$. The complexity estimates follow at once. □

## 5.5 The geometric progression case

Simplifications for Newton evaluation and interpolation also arise when the sample points form a geometric progression; this was already pointed out in [208] and references therein, but without mention to complexity.

In this section we show that the complexities of Newton evaluation and interpolation on a geometric progression of size $n$ drop to $\mathsf{M}(n) + O(n)$. By transposition, we deduce that the conversion between monomial and Newton bases have the same asymptotic cost. Last, as in the previous section, we obtain as corollaries fast algorithms for evaluation and interpolation on the monomial basis: the complexities of both tasks is shown to be in $O(\mathsf{M}(n))$, *i.e.* better by a logarithmic factor than in the case of arbitrary samples points.

Thus, geometric progressions should be considered as interesting choices for algorithms relying on evaluation and interpolation techniques. We illustrate this in the case of polynomial matrix multiplication algorithms.

In all what follows, we actually assume for simplicity that the geometric progression we consider has the form $x_i = q^i$, $i = 0, \ldots, n-1$. Treating the general case $x_i = x_0 q^i$, with arbitrary $x_0$, does not alter the asymptotic estimates, and only burdens the notation. Finally, we mention that many formulas presented below can be thought as $q$-analogues of those presented in the previous section.

**Newton interpolation and evaluation.** Our first question is that of Newton interpolation and evaluation: the following proposition proves the estimates of the last entry in the third column of Table 5.1.

**Proposition 2** *Let $k$ be a field and let $q \in k$ such that the elements $x_i = q^i$ are different from 1, for $i = 0, \ldots n-1$. Then Newton interpolation and evaluation on the geometric sequence $1, \ldots, q^{n-1}$ can be done using $\mathsf{M}(n) + O(n)$ base field operations.*

*Proof.* Let $F$ be a polynomial of degree less than $n$, let $(v_i)$ be the values $(F(x_i))$ and $(f_i)$ the coefficients of $F$ on the Newton basis associated to the points $(x_i)$. As in the previous section, we evaluate Formula (5.1) on the points $x_i$, yielding

$$\begin{aligned}
v_0 &= f_0 \\
v_1 &= f_0 + (q-1)f_1 \\
v_2 &= f_0 + (q^2-1)f_1 + (q^2-1)(q^2-q)f_2 \\
v_3 &= f_0 + (q^3-1)f_1 + (q^3-1)(q^3-q)f_2 + (q^3-1)(q^3-q)(q^3-q^2)f_3 \quad \dots
\end{aligned}$$

Let us introduce the triangular numbers $t_i = 1 + 2 + \cdots + (i-1) = i(i-1)/2$, for $i \geq 0$ and the modified sequence $g_i = q^{t_i} f_i$, for $i = 0, \dots, n-1$. Note that all coefficients $q^{t_i}$ can be computed in $O(n)$ base field operations, since $q^{t_{i+1}} = q^i q^{t_i}$. Thus, the coefficients $g_i$ and $f_i$ can be computed from one another for $O(n)$ base field operations. With this data, the above relations become

$$\begin{aligned}
v_0 &= g_0 \\
v_1 &= g_0 + (q-1)g_1 \\
v_2 &= g_0 + (q^2-1)g_1 + (q^2-1)(q-1)g_2 \\
v_3 &= g_0 + (q^3-1)g_1 + (q^3-1)(q^2-1)g_2 + (q^3-1)(q^2-1)(q-1)g_3 \quad \dots
\end{aligned}$$

Next, we introduce the numbers $w_i$ defined by

$$w_0 = v_0, \quad w_i = \frac{v_i}{(q-1)\cdots(q^i-1)}, \quad i = 1, \dots, n-1. \tag{5.6}$$

As above, $w_i$ and $v_i$ can be computed from one another for $O(n)$ base field operations. Using the modified values $w_i$, the above relations become

$$w_i = g_i + \sum_{j=0}^{i-1} \frac{1}{(q-1)\cdots(q^{i-j}-1)} g_j.$$

We conclude as in the arithmetic case. We introduce the generating series

$$W = \sum_{i=0}^{n-1} w_i x^i, \quad G = \sum_{i=0}^{n-1} g_i x^i, \quad T = 1 + \sum_{i=1}^{n-1} \frac{1}{(q-1)\cdots(q^i-1)} x^i, \tag{5.7}$$

so that the above relations become $W = GT$ modulo $x^n$. All coefficients of the series $T$ can be obtained in $O(n)$ base field relations, and its inverse modulo $x^n$ is given by

$$1 + \sum_{i=1}^{n-1} \frac{q^{\frac{i(i-1)}{2}}(-1)^i}{(q-1)\cdots(q^i-1)} x^i,$$

whose coefficients can also be obtained in $O(n)$ operations. The conclusion follows. □

**Conversions between monomial and Newton bases.** Our next step is to study the complexity of conversion between monomial and Newton bases. We prove the following result, which completes the first two entries in the last column of Table 5.1.

**Proposition 3** *Let $k$ be a field and let $q \in k$ such that the elements $x_i = q^i$ are different from 1, for $i = 0, \dots n-1$. Then the conversion between the Newton basis associated to $1, q, \dots, q^{n-1}$ and the monomial basis can be done using $\mathsf{M}(n) + O(n)$ base field operations.*

The proof comes from considering the transposed of the Newton evaluation and interpolation. Indeed, the following lemma relates these questions to those of conversions between monomial and Newton bases.

**Lemma 3** *Let $k$ be a field, let $q \in k \setminus \{0\}$ and let $r = 1/q$. Suppose that $1, q, \dots, q^{n-1}$ are pairwise distinct and define the following matrices:*

- *Let $A$ be the matrix of base change from the Newton basis associated to $1, q, \dots, q^{n-1}$ to the monomial basis.*
- *Let $B$ the matrix of Newton evaluation at $\{1, r, \dots, r^{n-1}\}$.*
- *Let $D_1$ and $D_2$ be the $n \times n$ diagonal matrices*

$$D_1 = \mathrm{Diag}\left[\frac{q^{i(i-1)/2}}{\prod_{k=0}^{i-1}(q^k-1)}\right]_{i=1}^{n} \quad \text{and} \quad D_2 = \mathrm{Diag}\left[(-1)^{j-1} q^{\frac{(j-1)(j-2)}{2}}\right]_{j=1}^{n}.$$

*Then the matrix equality $A = D_1\, B^t\, D_2$ holds.*

*Proof.* Given two integers $n$ and $k$, the *$q$-binomial coefficient* [202, 117, 91] is defined as

$$\left[\begin{array}{c} n \\ k \end{array}\right]_q = \begin{cases} \frac{1-q^n}{1-q} \cdot \frac{1-q^{n-1}}{1-q^2} \cdots \frac{1-q^{n-k+1}}{1-q^k}, & \text{for} \quad n \geq k \geq 1, \\ 0, & \text{for} \quad n < k \ \text{ or } \ k = 0. \end{cases}$$

The following generalization of the usual binomial formula holds:

$$\prod_{k=1}^{n} \left(1 + q^{k-1} x\right) \;=\; \sum_{k=0}^{n} \left[\begin{array}{c} n \\ k \end{array}\right]_q q^{\frac{k(k-1)}{2}}\, x^k. \tag{5.8}$$

From Equation (5.8), it is then easy to deduce that the entries of the matrix $A$ are

$$A_{i,j} \;=\; (-1)^{j-i} \left[\begin{array}{c} j-1 \\ i-1 \end{array}\right]_q q^{(j-i)(j-i-1)/2}.$$

On the other hand, the $(i,j)$ entry of the matrix representing Newton evaluation with respect to $\{x_0, \dots, x_{n-1}\}$ is zero if $j < i$ and equals $\prod_{k=1}^{j-1} \left(x_{i-1} - x_{k-1}\right)$ for all $j \geq i \geq 1$. Applying this to $x_i = 1/q^i$, we get

$$B_{i,j} = (-1)^{j-1} \cdot \prod_{k=1}^{j-1} \frac{q^{i-k}-1}{q^{i-1}}, \quad \text{for all} \quad j \geq i \geq 1.$$

Having the explicit expressions of the entries of $A$ and $B$ allows to write the equality

$$\frac{B^t_{i,j}}{A_{i,j}} = (-1)^{j-1} \frac{q^{\frac{i(i-1)}{2}+\frac{(j-1)(j-2)}{2}}}{(q-1)\cdots(q^{i-1}-1)},$$

from which the lemma follows. □

Thus, up to multiplications by diagonal matrices, the conversion maps between monomial and Newton bases are the transposes of those of Newton evaluation and interpolation, at the cost of replacing $q$ by $1/q$. Using Tellegen's theorem, the proof of Proposition 3 is now immediate, since the two diagonal matrices involved can be computed in time $O(n)$.

For the sake of completeness, we describe below an algorithm for the transposed evaluation on a geometric sequence. On input the $n$ values $v_0, \ldots, v_{n-1}$, it does the following:

- Compute the values $w_i$ defined in Equation (5.6).
- Compute the transposed product $\sum_{i=0}^{n-1} g_i x^i$ of the series $T$ defined in Equation (5.7) by the series $\sum_{i=0}^{n-1} w_i x^i$.
- Return the values $q^{t_i} g_i$, for $0 \leq i \leq n-1$, where $t_i = i(i-1)/2$.

The algorithm for transposed interpolation would be deduced in a similar manner.

**Evaluation and interpolation on the monomial basis** We now treat the question of fast monomial evaluation and interpolation on a geometric progression. As before, we take $x_i = q^i$, $i = 0, \ldots, n-1$, where $q \in k$ is such that the elements $1, q, \ldots, q^{n-1}$ be pairwise distinct.

It is known that evaluating a polynomial $P = p_0 + p_1 x + \cdots + p_n x^n$ on the geometric progression $\{1, q, \ldots, q^{n-1}\}$ can be done using $O(\mathsf{M}(n))$ operations. This operation, generalizing the discrete Fourier transform, is called the *chirp transform* and has been independently studied by Rabiner, Schafer and Rader [198] and by Bluestein [27], see also [5]. In contrast, to the best of our knowledge, no algorithm for the inverse operation – interpolation at a geometric progression – has ever been given. Our aim is now to show that the inverse chirp transform can be performed in a similar asymptotic complexity. These results are gathered in the following proposition, which completes the entries of Table 5.1.

**Proposition 4** *Let $k$ be a field, let $n \geq 0$ and let $q \in k$ such that the elements $x_i = q^i$, for $i = 0, \ldots n-1$, are pairwise distinct. If $F \in k[x]$ has degree less than $n$ then:*

- *Given the coefficients of $F$ on the monomial basis, all values $F(x_0), \ldots, F(x_{n-1})$ can be computed in $2\mathsf{M}(n) + O(n)$ base field operations.*
- *Given the values $F(x_0), \ldots, F(x_{n-1})$, all coefficients of $F$ on the monomial basis can be computed in $2\mathsf{M}(n) + O(n)$ base field operations.*

*Proof.* We briefly recall how the direct chirp transform works. It is based on the equalities:

$$P(q^i) = \sum_{j=0}^{n} p_j q^{ij} = q^{-i^2/2} \cdot \sum_{j=0}^{n} p_j q^{-j^2/2} q^{(i+j)^2/2}.$$

Suppose first that $q$ is a square in $k$. Computing the values $b_i = q^{i^2/2}$, for $0 \leq i \leq 2n$ and $c_j = p_j q^{-j^2/2}$, for $0 \leq j \leq n$, takes linear time in $n$, using the recurrence $q^{(i+1)^2} = q^{i^2} q^{2i} q$. Then, the preceding formula shows that the values $P(q^i)$ are, up to constant factors, given by the *middle part* of the polynomial product

$$\Big(b_0 + b_1 x + \cdots + b_{2n} x^{2n}\Big)\Big(c_n + c_{n-1} x + \cdots + c_0 x^n\Big).$$

Using standard polynomial multiplication, this algorithm requires $2\mathsf{M}(n)$ operations, but the complexity actually drops to $\mathsf{M}(n) + O(n)$, using the middle product of [115].

In the general case when $q$ is not a square, several possibilities are available. The first idea is to introduce a square root for $q$ by computing in $K = k[T]/(T^2 - q)$. Another choice is to use the algorithms described previously: performing first a change of base to the Newton representation, and then a Newton evaluation. This way, we obtain the estimate of $2\mathsf{M}(n) + O(n)$ operations for the chirp transform.

Let us now focus on the computation of the inverse chirp transform. As above, we use the Newton basis for intermediate computations: first perform a Newton interpolation, then perform a conversion from the Newton basis to the monomial basis. Both steps have complexities $\mathsf{M}(n) + O(n)$, which gives the estimate of $2\mathsf{M}(n) + O(n)$ operations for the inverse chirp transform. □

For completeness, we give in Figure 5.1 and Figure 5.2 below our new algorithms for evaluation and interpolation on points in a geometric progression.

**EvalGeom**$(p_0, \ldots, p_{n-1}, F)$
  $q_0 \leftarrow 1$; $s_0 \leftarrow 1$; $u_0 \leftarrow 1$; $z_0 \leftarrow 1$; $g_0 \leftarrow 1$;
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $q_i \leftarrow q_{i-1} \cdot p_{i-1}$;
    $s_i \leftarrow s_{i-1} \cdot (p_i - 1)$;
    $u_i \leftarrow u_{i-1} \cdot p_i/(1 - p_i)$;
    $z_i \leftarrow (-1)^i u_i / q_i$;
  $G \leftarrow \mathrm{mul}^t(n-1, \sum_{i=0}^{n-1} z_i x^i, \sum_{i=0}^{n-1} \mathtt{Coeff}(F, i)/z_i\, x^i)$;
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $g_i \leftarrow (-1)^i u_i\, \mathtt{Coeff}(G, i)$;
  $W \leftarrow (\sum_{i=0}^{n-1} g_i x^i) \cdot (\sum_{i=0}^{n-1} s_i^{-1} x^i)$;
**return** $s_0\, \mathtt{Coeff}(W, 0), \ldots, s_{n-1}\, \mathtt{Coeff}(W, n-1)$;

Figure 5.1: Polynomial evaluation at the points $p_0 = 1, p_1 = q, \ldots, p_{n-1} = q^{n-1}$.

**InterpGeom**$(p_0, p_1, \ldots, p_{n-1}, v_0, \ldots, v_{n-1})$
  $q_0 \leftarrow 1$; $s_0 \leftarrow 1$; $u_0 \leftarrow 1$; $z_0 \leftarrow 1$; $w_0 \leftarrow v_0$;
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $q_i \leftarrow q_{i-1} \cdot p_{i-1}$;
    $s_i \leftarrow s_{i-1} \cdot (p_i - 1)$;
    $u_i \leftarrow u_{i-1} \cdot p_i/(1-p_i)$;
    $z_i \leftarrow (-1)^i u_i/q_i$;
  $H \leftarrow (\sum_{i=0}^{n-1} v_i/s_i x^i) \cdot (\sum_{i=0}^{n-1} (-x)^i q_i/s_i)$;
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $w_i \leftarrow (-1)^i \texttt{Coeff}(H, i)/u_i$;
  $G \leftarrow \mathrm{mul}^t(n-1, \sum_{i=0}^{n-1} u_i x^i, \sum_{i=0}^{n-1} w_i x^i)$;
**return** $\sum_{i=0}^{n-1} z_i \, \texttt{Coeff}(G, i)\, x^i$;

Figure 5.2: Polynomial interpolation at the points $p_0 = 1, p_1 = q, \ldots, p_{n-1} = q^{n-1}$.

**Application to polynomial matrix multiplication.** We now apply the above results to improve the complexity of polynomial matrix multiplication. This problem is important, since polynomial matrix multiplication is a primitive of linear algebra algorithms dealing with polynomial matrices (determinant, inversion, system solving, column reduction, integrality certification, normal forms), see for instance [251, 233, 97]. It also occurs during computations of matrix Padé-type approximants [17, 18, 241, 242, 97], recurrences with polynomial coefficients [59, 34] and linear differential operators.

Let $\mathsf{MM}(n, d)$ represent the number of base field operations required to multiply two $n \times n$ matrices with polynomial entries of degree at most $d$. For simplicity, the cost $\mathsf{MM}(n, 0)$ of scalar $n \times n$ matrix multiplication will be denoted $\mathsf{MM}(n)$.

Cantor and Kaltofen [50] described an algorithm for multiplying degree $d$ polynomials with coefficients from an arbitrary (possibly non commutative) algebra using $O(\mathsf{M}(d))$ algebra operations. Viewing polynomial matrices as polynomials with scalar matrix coefficients, the result in [50] implies that $\mathsf{MM}(n, d) = O\left(\mathsf{M}(d)\,\mathsf{MM}(n)\right)$. Over base fields of cardinality larger than $2d$, the use of an evaluation/interpolation scheme allows to uncouple polynomial and matrix products and yields the better bound

$$\mathsf{MM}(n, d) = O\left(\mathsf{MM}(n)\, d + n^2\, \mathsf{M}(d) \log(d)\right). \tag{5.9}$$

An important remark [241, 242] (see also [18]) is that if the base field supports FFT, then choosing the roots of unity as sample evaluation points improves the previous estimate to

$$\mathsf{MM}(n, d) = O\left(\mathsf{MM}(n)\, d + n^2\, d \log(d)\right). \tag{5.10}$$

However, the algorithm in [241, 242] is dependent on the specific use of FFT, which might not be pertinent for polynomials of moderate degrees.

In contrast, using evaluation and interpolation at a geometric progression enables us to obtain the following result.

**Theorem 5** *Let $n, d \geq 0$ and let $k$ be a field of characteristic 0, or a finite field of cardinality at least $2d+1$. Then we have the estimate*

$$\mathsf{MM}(n,d) = 2\,d\,\mathsf{MM}(n) + 6\,n^2\,\mathsf{M}(2d) + O(n^2 d).$$

*Proof.* In both cases, we use multipoint evaluation and interpolation on a geometric progression $1, q, \ldots, q^{2d-1}$ of size $2d$. In characteristic 0, we can take $q = 2$. If $k$ is finite, we take for $q$ a generator of the multiplicative group $k^*$ (for practical purposes, we might as well choose $q$ at random, if $k$ has a large enough cardinality). □

Theorem 5 may be seen as an improvement by a log factor of the bound (5.9), generalizing the bound (5.10) to an arbitrary multiplication time $\mathsf{M}$ function satisfying the hypotheses described at page 97. Still, for polynomial matrices of high degrees, the method in [241, 242] is better by a constant factor than ours, since the polynomial multiplication uses FFT, and thus itself requires evaluating and interpolating at the roots of unity.

To conclude this chapter, Figure 5.3 and Figure 5.4 display the speed-up obtained using our polynomial matrix multiplication algorithm, versus a naive product (thus, a larger number means a more significant improvement). The matrix sizes vary from 1 to 120, the polynomial degrees vary from 0 to 200, and the base field is $\mathbb{Z}/p\mathbb{Z}$, where $p$ is a 32 bit prime. The time ratios are given in the table of Figure 5.3 and displayed graphically in Figure 5.4.

The implementation is made using Shoup's NTL C++ library [226]; we used a naive matrix multiplication of cubic complexity, and NTL's built-in polynomial arithmetic (for polynomials in the range 0–200, naive, Karatsuba and FFT multiplication algorithms are successively used). The timings are obtained on an Intel Pentium 4 CPU at 2GHz.

| | 15 | 35 | 55 | 75 | 95 | 115 | 135 | 155 | 175 | 195 |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.6 | 0.8 | 1.4 | 1.4 | 1.6 | 1.6 | 2.2 | 2.1 | 1.7 | 1.7 |
| 30 | 1.1 | 1.2 | 2.0 | 2.0 | 2.4 | 2.3 | 3.3 | 3.2 | 2.5 | 2.5 |
| 40 | 1.2 | 1.6 | 2.6 | 2.7 | 3.2 | 3.0 | 4.3 | 4.1 | 3.3 | 3.2 |
| 50 | 1.4 | 2.0 | 3.2 | 3.3 | 3.9 | 3.6 | 5.3 | 5.1 | 4.1 | 4.0 |
| 60 | 1.7 | 2.3 | 3.8 | 3.9 | 4.6 | 4.4 | 6.3 | 6.1 | 4.8 | 4.7 |
| 70 | 1.9 | 2.6 | 4.3 | 4.5 | 5.3 | 5.0 | 7.2 | 6.9 | 5.6 | 5.4 |
| 80 | 2.1 | 2.9 | 4.8 | 4.9 | 6.0 | 5.6 | 8.1 | 7.7 | 6.2 | 5.9 |
| 90 | 2.3 | 3.3 | 5.5 | 5.7 | 6.6 | 6.2 | 9.0 | 8.6 | 6.9 | 6.7 |
| 100 | 2.5 | 3.5 | 6.0 | 6.2 | 7.3 | 6.8 | 9.8 | 9.3 | 7.5 | 7.3 |
| 110 | 2.6 | 3.9 | 6.3 | 6.6 | 7.8 | 7.3 | 10.6 | 10.1 | 8.1 | 7.9 |

Figure 5.3: Time ratios between classical and improved polynomial matrix multiplication algorithms. Rows are indexed by the matrix size (20—110); columns are indexed by the matrix degree (15–195).

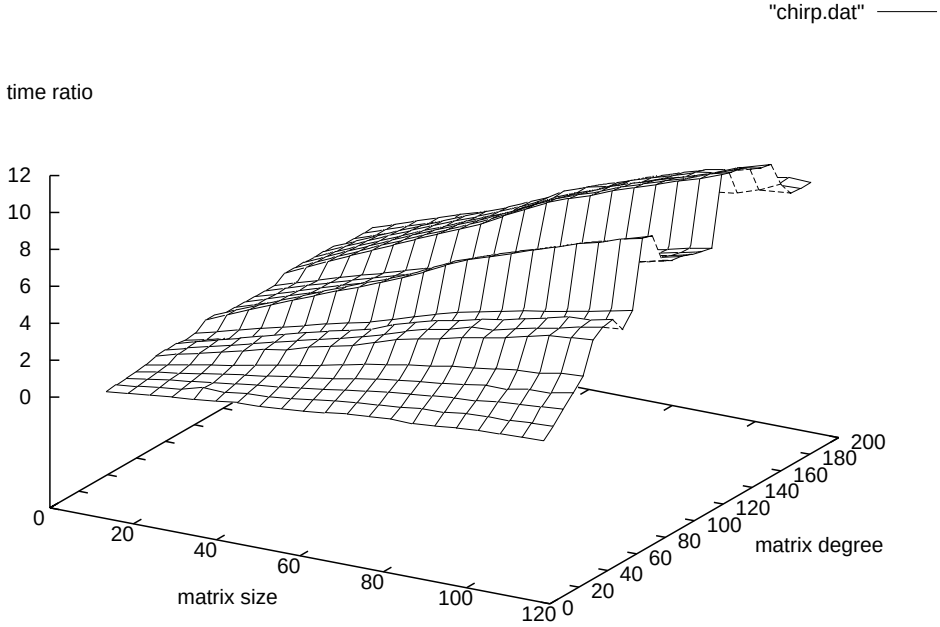



Figure 5.4: Speed-up between classical and improved polynomial matrix multiplication.

## 5.6 Appendix: Fast conversions between monomial and Bernstein basis

Let $k$ be a field and let $n \in \mathbb{N}$. The *Bernstein basis* of the $k$-vector space of polynomials of degree at most $n$ is defined by:

$$B_k(x) = \binom{n}{k} x^k (1-x)^{n-k}.$$

In this short appendix, we address the question of converting a polynomial from the Bernstein basis to the monomial basis and vice-versa. The direct question translates in matrix terms as the problem of matrix-vector multiplication by the lower triangular *Bernstein matrix* $B$ defined as follows:

$$B_{i,j} = (-1)^{i-j} \binom{n}{j-1} \binom{n-j+1}{i-j}, \quad \text{for } 1 \leq j \leq i \leq n+1.$$

For instance, the $6 \times 6$ Bernstein matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -5 & 5 & 0 & 0 & 0 & 0 \\ 10 & -20 & 10 & 0 & 0 & 0 \\ -10 & 30 & -30 & 10 & 0 & 0 \\ 5 & -20 & 30 & -20 & 5 & 0 \\ -1 & 5 & -10 & 10 & -5 & 1 \end{bmatrix}.$$

Spencer [229] proposed an algorithm using divided differences, whose complexity is quadratic in the degree of the polynomial. That algorithm amounts to constructing the entries of the matrix $B$ above. Using generating series, we show that the complexity of both conversion problems is in $2\,\mathsf{M}(n) + O(n)$.

The complexity estimate is given in terms of number of operations in the base field (*e.g.* the rational field or any other field of characteristic zero or larger than $n$). When the base field is $\mathbb{Q}$, we do not consider the bit complexity of these computations.

## Algorithms

Let $q_0, \ldots, q_n$ be the coefficients of $P(x)$ in the Bernstein basis. Denoting

$$d_i = q_i - \binom{i}{1} q_{i-1} + \cdots + (-1)^i q_0 \quad \text{and} \quad c_i = \binom{n}{i},$$

it is easy to show that the coefficients of $P(x)$ in the monomial basis are given by the formula

$$p_i = c_i d_i.$$

The terms $c_i$ can be computed in $O(n)$ using the recurrence $c_i = c_{i-1}\frac{n-i+1}{i}$ for $1 \leq i \leq n$, with initial condition $c_0 = 1$. This shows that our conversion problems reduce to computing the terms $d_i$ from the coefficients of $P$ in the monomial basis and conversely.

Our fast solution relies on the observation that the generating series $\sum_{i \geq 0} d_i x^i$ is rational and equals $N(x)/D(x)$, where $D(x) = (x+1)^{n+1}$ and

$$N(x) = \sum_{i=0}^{n} q_i x^i (x+1)^{n-i}.$$

Denoting $Q = \sum_{i=0}^{n} q_i x^{n-i}$, we deduce that $N(x)$ is the reversal polynomial of $Q(x+1)$.

The conversion algorithms follow easily from this remark.

**From Bernstein basis to monomial basis.** Starting from $Q = \sum_{i=0}^{n} q_i x^{n-i}$, compute $R = Q(1+x)$, then multiply the first $n$ terms of the power series $\mathrm{rev}(Q)/(1+x)^{n+1}$ by $c_0, \ldots, c_n$. This gives $P$.

**From the monomial basis to Bernstein basis.** Starting from $P = \sum_{i=0}^{n} p_i x^i$, compute the product $S$ between $(1+x)^{n+1}$ and $\sum_{i=0}^{n} p_i/c_i x^i$ modulo $x^{n+1}$, then evaluate the reversal $\mathrm{rev}(S)$ on $(x-1)$ to obtain the coefficients $q_i$ of $P$ in Bernstein basis.

Let us estimate the complexities of these algorithms. It is well known that the coefficients of either $(1+x)^{n+1}$ or its inverse modulo $x^{n+1}$ can be obtained in $O(n)$ operations. Indeed, denoting

$$(1+x)^{n+1} = \sum_{j=0}^{n} a_j x^j \mod x^{n+1} \quad \text{and} \quad 1/(1+x)^{n+1} = \sum_{j=0}^{n} b_j x^j \mod x^{n+1},$$

the coefficients $a_j$ and $b_j$ satisfy the recurrences

$$a_0 = 1,\ a_{i+1} = \frac{n+1-i}{i+1} a_i \quad \text{and} \quad b_0 = 1,\ b_{i+1} = -\frac{i+n+1}{i+1} b_i, \quad \text{for } 0 \le i \le n-1.$$

Thus, the previous algorithms both require two polynomial multiplications in degree $n$ and $O(n)$ additional operations. This proves our claims on the complexity of the operations with Bernstein matrix.

# Chapter 6

# Equivalence between polynomial evaluation and interpolation

We compare the complexities of multipoint polynomial evaluation and interpolation. We show that, over fields of characteristic zero, both questions have equivalent complexities, up to a constant number of polynomial multiplications.

The results of this chapter are joint work with É. Schost [37].

## Contents

## 6.1 Introduction

Multipoint polynomial evaluation and interpolation are ubiquitous problems. They can be stated as follows:

**Evaluation:** Given some evaluation points $x_0, \ldots, x_n$ and the coefficients $p_0, \ldots, p_n$ of a polynomial $P$, compute the values $P(x_i) = \sum_{j=0}^{n} p_j x_i^j$, for $i = 0, \ldots, n$.

**Interpolation:** Given distinct interpolation points $x_0, \ldots, x_n$ and values $q_0, \ldots, q_n$, compute the unique coefficients $p_0, \ldots, p_n$ such that $\sum_{j=0}^{n} p_j x_i^j = q_i$ holds for $i = 0, \ldots, n$.

It is known that the complexities of these operations are closely related: for instance, the interpolation algorithms of Lipson [156], Fidducia [80], Borodin and Moenck [169, 30] and Strassen [235] all require to perform a multipoint evaluation as a subtask. Thus in this note, rather than describing particular algorithms, we focus on comparing the complexities of both questions, that is, on reductions of one question to the other.

Close links appear when one puts *program transposition* techniques into use. Roughly speaking, such techniques prove that an algorithm that performs a matrix-vector product can be transformed into an algorithm with essentially the same complexity, and which performs the transposed matrix product. These techniques are particularly relevant here, as many relations exist between Vandermonde matrices, their transposes, and other structured matrices such as Hankel matrices.

Using such relations, reductions of interpolation to evaluation, and conversely, have been proposed in (or can be deduced from) [127, 49, 184, 83, 104, 105, 25, 189]. Nevertheless, to our knowledge, no equivalence theorem has been established for these questions. All results we are aware of involve the following additional operation: given $x_0, \ldots, x_n$, compute the coefficients of $\prod_{i=0}^{n}(T - x_i)$, that is, the elementary symmetric functions in $x_0, \ldots, x_n$. If we denote by $\mathsf{E}(n)$, $\mathsf{I}(n)$ and $\mathsf{S}(n)$ the complexities of multipoint evaluation, interpolation and elementary symmetric functions computation on inputs of size $n+1$, then the above references yield

$$\mathsf{I}(n) \in O(\mathsf{E}(n) + \mathsf{S}(n)) \quad \text{and} \quad \mathsf{E}(n) \in O(\mathsf{I}(n) + \mathsf{S}(n)).$$

The best currently known result gives $\mathsf{S}(n) \in O(\mathsf{M}(n)\log(n))$, where $\mathsf{M}(n)$ is the cost of degree $n$ polynomial multiplication, see [255, Ch. 10].

Our purpose in this note is to reduce the gap, replacing the terms $\mathsf{S}(n)$ by $\mathsf{M}(n)$ in the above estimates. This is all the more relevant as the best currently known algorithms for evaluation and interpolation have complexities in $O(\mathsf{M}(n)\log(n))$ [169, 30, 235, 35], and obtaining algorithms of complexity $O(\mathsf{M}(n))$ for any of these questions are long-standing open problems.

Actually, we prove a sharper statement: it is known that evaluation or interpolation simplifies for particular families of points (e.g., geometric progressions), see for instance [5, 38] and the comments below. We take this specificity into account; roughly speaking, we prove that:

- given an algorithm that performs evaluation on some distinguished families of points, one can deduce an algorithm that performs interpolation on the same families of points, and with essentially the same complexity, up to a constant number of polynomial multiplications.

- given an algorithm that performs interpolation on some distinguished families of points, one can deduce an algorithm that performs evaluation on the same families of points, and with essentially the same complexity, up to a constant number of polynomial multiplications.

We can infer two corollaries from these results: first, we deduce the estimates

$$\mathsf{I}(n) \in O(\mathsf{E}(n) + \mathsf{M}(n)) \quad \text{and} \quad \mathsf{E}(n) \in O(\mathsf{I}(n) + \mathsf{M}(n)),$$

as claimed above.

Our second corollary relates to results from [5]. That article studies the families of $n$ points in $\mathbb{C}$ on which any degree $n$ polynomial can be evaluated in time $O(\mathsf{M}(n))$. The above results show that these are precisely the families of points on which any degree $n$ polynomial can interpolated in time $O(\mathsf{M}(n))$. For instance, it is proved in [5] that given any $a, b, c, z \in \mathbb{C}^4$, any degree $n$ polynomial can be evaluated on the sequence $a + bz^i + cz^{2i}$ in time $O(\mathsf{M}(n))$. We deduce that as soon as all these points are distinct, any degree $n$ polynomial can be interpolated on this sequence in time $O(\mathsf{M}(n))$ as well.

Our approach closely follows the ideas given in the references mentioned above, notably [127, 49]. In particular, we borrow from these two references the reductions of one problem to the other. In both cases, we have to compute the symmetric functions in the sample points $x_0, \ldots, x_n$. Technically, our contribution is to point out that the cost of this operation reduces to that of either interpolation or evaluation, up to a constant number of polynomial multiplications. The main ideas are the following:

- Suppose that an algorithm that performs interpolation at $x_0, \ldots, x_n$ is given. We cannot use it to deduce the polynomial $F = \prod_{i=0}^{n}(T - x_i)$ directly, since $F$ has degree $n+1$. Nevertheless, we can recover the polynomial $\prod_{i=1}^{n}(T - x_i)$ by interpolation, since it has degree $n$, and its values at $x_0, \ldots, x_n$ are easy to compute. Then, recovering $F$ is immediate.

- Suppose that an algorithm that performs evaluation at $x_0, \ldots, x_n$ is given. By transposition, this algorithm can be used to compute the power sums of the polynomial $F = \prod_{i=0}^{n}(T - x_i)$. Then one can deduce the coefficients of $F$ from its power sums using the fast exponentiation algorithm of [42, 212].

The rest of this chapter is devoted to give a rigorous version of these considerations. In the next section, we first precise our computational model, then state our results in this model. The final two sections give the proofs.

## 6.2 Computational model, main result

Our basic computational objects are *straight-line programs* (allowing divisions), which are defined as follows: Let $\mathbf{A} = A_0, \ldots, A_r$ be a family of indeterminates over a field $k$ of characteristic 0. Let us define $F_{-r} = A_0, \ldots, F_0 = A_r$. A *straight-line program* $\gamma$ is a sequence $F_1, \ldots, F_L \subset k(\mathbf{A})$ such that for $1 \leq \ell \leq L$, one of the following holds:

- $F_\ell = \lambda$, with $\lambda \in k$;
- $F_\ell = \lambda \star F_i$, with $\lambda \in k$, $\star \in \{+, -, \times, \div\}$ and $-r \leq i < \ell$;
- $F_\ell = F_i \star F_j$, with $\star \in \{+, -, \times, \div\}$ and $-r \leq i, j < \ell$.

The *size* of $\gamma$ is $L$; the *output* of $\gamma$ is a subset of $\{F_{-r}, \ldots, F_L\}$. For $\mathbf{a} = a_0, \ldots, a_r \in k^{r+1}$, $\gamma$ is *defined* at $\mathbf{a}$ if $\mathbf{a}$ cancels no denominator in $\{F_1, \ldots, F_L\}$.

In the sequel, we will consider algorithms that take as input both the sample points $\mathbf{x} = x_0, \ldots, x_n$ and the coefficients (resp. values) of a polynomial $P$. We will allow arbitrary operations on the sample points. On the other hand, since we compute linear functions of the coefficients (resp. values) of $P$, we will only allow *linear* operations on them; this is actually not a limitation, because any non-linear step can be simulated by at most 3 linear steps, see [236] and [47, Th. 13.1].

Formally, we will thus consider straight-line programs taking as input two families of indeterminates $\mathbf{A}$ and $\mathbf{B}$, allowing only linear operations on the second family of indeterminates. The straight-line programs satisfying these conditions are called $\mathbf{B}$-*linear straight-line programs* and are defined as follows (compare with [47, Ch. 13]).

Let $\mathbf{A} = A_0, \ldots, A_r$ and $\mathbf{B} = B_0, \ldots, B_s$ be two families of indeterminates over a field $k$ of characteristic 0. Let us define $F_{-r} = A_0, \ldots, F_0 = A_r$ and $G_{-s} = B_0, \ldots, G_0 = B_s$. A $\mathbf{B}$-*linear straight-line program* $\Gamma$ is the data of two sequences $F_1, \ldots, F_L \subset k(\mathbf{A})$ and $G_1, \ldots, G_M \subset k(\mathbf{A})[\mathbf{B}]$ such that:

- $F_1, \ldots, F_L$ satisfy the axioms of straight-line programs given above.
- For $1 \leq m \leq M$, one of the following holds:
  - $G_m = \lambda G_i$, with $\lambda \in k \cup \{F_{-r}, \ldots, F_L\}$ and $-s \leq i < m$;
  - $G_m = \pm G_i \pm G_j$, with $-s \leq i, j < m$.

In particular, $G_1, \ldots, G_M$ are linear forms in $\mathbf{B}$, as requested. Here, the *size* of $\Gamma$ is $L + M$. The *output* of $\Gamma$ is a subset of $\{G_{-s}, \ldots, G_M\}$. For $\mathbf{a} = a_0, \ldots, a_r \in k^{r+1}$, $\Gamma$ is *defined* at $\mathbf{a}$ if $\mathbf{a}$ cancels no denominator in $\{F_1, \ldots, F_L\} \cup \{G_1, \ldots, G_M\}$.

**Composition rules.** In the sequel, we use estimates for the behaviour of the size of straight-line programs and $\mathbf{B}$-linear straight-line programs under various compositions. We now state them explicitly for completeness; later, we will use them in an implicit manner.

Let $\mathbf{A}$ and $\mathbf{B}$ be as above. Let the cost function $C$ assign to a sequence $\mathbf{F} \subset k(\mathbf{A})$ the minimal size of a straight-line program that computes $\mathbf{F}$; define $C_{\text{lin}}$ similarly for sequences of linear forms in $k(\mathbf{A})[\mathbf{B}]$, using $\mathbf{B}$-linear straight-line programs.

Let $\mathbf{F}, \mathbf{F}'$ be sequences of rational functions in $k(\mathbf{A})$ and let $\mathbf{G}, \mathbf{G}'$ be sequences of linear forms in $k(\mathbf{A})[\mathbf{B}]$. The rational functions $\mathbf{F} \circ \mathbf{F}'$ and the linear forms $\mathbf{G} \circ \mathbf{G}'$ are defined in an obvious way, as soon as dimensions match. We next define $\mathbf{G} \circ \mathbf{F} \in k(\mathbf{A})$ as the evaluation of $\mathbf{G}$ on $\mathbf{F}$, and $\mathbf{G} \diamond \mathbf{F} \in k(\mathbf{A})[\mathbf{B}]$ as the linear form where all coefficients of $\mathbf{G}$ are evaluated at $\mathbf{F}$ (again, as soon as dimensions match). Then all our results rely on the following straightforward properties:

$$C(\mathbf{F} \circ \mathbf{F}') \leq C(\mathbf{F}) + C(\mathbf{F}'), \quad C(\mathbf{F} \cup \mathbf{F}') \leq C(\mathbf{F}) + C(\mathbf{F}'),$$
$$C_{\text{lin}}(\mathbf{G} \circ \mathbf{G}') \leq C_{\text{lin}}(\mathbf{G}) + C_{\text{lin}}(\mathbf{G}'), \quad C_{\text{lin}}(\mathbf{G} \cup \mathbf{G}') \leq C_{\text{lin}}(\mathbf{G}) + C_{\text{lin}}(\mathbf{G}'),$$
$$C(\mathbf{G} \circ \mathbf{F}) \leq C_{\text{lin}}(\mathbf{G}) + C(\mathbf{F}), \quad C_{\text{lin}}(\mathbf{G} \diamond \mathbf{F}) \leq C_{\text{lin}}(\mathbf{G}) + C(\mathbf{F}).$$

**Multiplication.** We will use a function denoted by $\mathsf{M}(n)$, which represents the complexity of univariate polynomial multiplication. It is defined as follows: For any $n \geq 0$, let us introduce the indeterminates $\mathbf{A} = A_0, \ldots, A_n$, $\mathbf{B} = B_0, \ldots, B_n$, and let us define the polynomials $C_0, \ldots, C_{2n}$ in $k[\mathbf{A}, \mathbf{B}]$ by the relation

$$\left(\sum_{i=0}^{n} A_i T^i\right)\left(\sum_{i=0}^{n} B_i T^i\right) = \sum_{i=0}^{2n} C_i T^i$$

in $k[\mathbf{A}, \mathbf{B}][T]$. The polynomials $C_i$ are *linear* in $\mathbf{B}$ (they are of course actually bilinear in $\mathbf{A}, \mathbf{B}$); then, we require that they can be computed by a $\mathbf{B}$-*linear* straight-line program of size $\mathsf{M}(n)$. Theorem 13.1 in [47] shows again that restricting to such linear operations is no limitation, since allowing arbitrary operations in $\mathbf{B}$ would at best gain a constant factor.

**Main results.** With this definition, our results are the following. Roughly speaking, Theorem 6 shows that, up to a constant number of polynomial multiplications, evaluation is not harder than interpolation, and Theorem 7 shows the converse. As mentioned above, we want to take into account the possibility of specialized algorithms, which may give the result only for some distinguished families of sample points: this justifies putting hypotheses on the points $\mathbf{x}$ in the theorems.

**Theorem 6** *Let $\Gamma$ be a $\mathbf{Q}$-linear straight-line program of size $L$, taking as input $\mathbf{X} = X_0, \ldots, X_n$ and $\mathbf{Q} = Q_0, \ldots, Q_n$, and let $G_0, \ldots, G_n \in k(\mathbf{X})[\mathbf{Q}]$ be the output of $\Gamma$. Then there exists a $\mathbf{P}$-linear straight-line program $\Delta$ of size $2L + O(\mathsf{M}(n))$, taking as input $\mathbf{X}$ and $\mathbf{P} = P_0, \ldots, P_n$, and with the following property.*

*Let* $\mathbf{x} = x_0, \ldots, x_n$ *be pairwise distinct points such that* $\Gamma$ *is defined at* $\mathbf{x}$ *and such that the vector* $G_j(\mathbf{x}, \mathbf{Q})$ *satisfies*

$$\sum_{j=0}^{n} G_j(\mathbf{x}, \mathbf{Q}) x_i^j = Q_i, \quad \text{for } i = 0, \ldots, n.$$

*Then* $\Delta$ *is defined at* $\mathbf{x}$ *and the output* $H_0, \ldots, H_n$ *of* $\Delta$ *satisfies*

$$H_i(\mathbf{x}, \mathbf{P}) = \sum_{j=0}^{n} P_j x_i^j, \quad \text{for } i = 0, \ldots, n.$$

**Theorem 7** *Let* $\Delta$ *be a* $\mathbf{P}$*-linear straight-line program of size* $L$*, taking as input* $\mathbf{X} = X_0, \ldots, X_n$ *and* $\mathbf{P} = P_0, \ldots, P_n$*, and let* $H_0, \ldots, H_n \in k(\mathbf{X})[\mathbf{P}]$ *be the output of* $\Delta$*. Then there exists a* $\mathbf{Q}$*-linear straight-line program* $\Gamma$ *of size* $3L + O(\mathsf{M}(n))$*, taking as input* $\mathbf{X}$ *and* $\mathbf{Q} = Q_0, \ldots, Q_n$*, and with the following property.*

*Let* $\mathbf{x} = x_0, \ldots, x_n$ *be pairwise distinct points such that* $\Delta$ *is defined at* $\mathbf{x}$ *and such that the vector* $H_j(\mathbf{x}, \mathbf{P})$ *satisfies*

$$H_i(\mathbf{x}, \mathbf{P}) = \sum_{j=0}^{n} P_j x_i^j, \quad \text{for } i = 0, \ldots, n.$$

*Then* $\Gamma$ *is defined at* $\mathbf{x}$ *and the output* $G_0, \ldots, G_n$ *of* $\Gamma$ *satisfies*

$$\sum_{j=0}^{n} G_j(\mathbf{x}, \mathbf{Q}) x_i^j = Q_i, \quad \text{for } i = 0, \ldots, n.$$

## 6.3 Program transposition

Inspired by [127, 49, 189], we will use the following idea: any algorithm that performs multipoint evaluation (resp. evaluation) can be transformed into one that performs the transposed operation. Originating from [239], *Tellegen's principle* precisely gives this kind of result, and predicts the difference of complexity induced by the transposition operation; see [47] for a proof and [124] for a detailed discussion. In our context, we easily obtain the following result:

**Lemma 4** *Let* $\Gamma$ *be* $\mathbf{P}$*-linear straight line program of size* $L$*, taking as input* $\mathbf{X} = X_0, \ldots, X_n$ *and* $\mathbf{P} = P_0, \ldots, P_n$ *and let* $G_0, \ldots, G_n \in k(\mathbf{X})[\mathbf{P}]$ *be the output of* $\Gamma$*. Then there exists a* $\mathbf{Q}$*-linear straight line program* $\Gamma^\dagger$ *of size* $L + O(n)$*, taking as input* $\mathbf{X}$ *and* $\mathbf{Q} = Q_0, \ldots, Q_n$*, and with the following property.*

*Let* $\mathbf{x} \in k^{n+1}$ *be such that* $\Gamma$ *is defined at* $\mathbf{x}$ *and let* $\varphi$ *be the linear map* $\mathbf{p} \mapsto (G_i(\mathbf{x}, \mathbf{p}))$*. Then* $\Gamma^\dagger$ *is defined at* $\mathbf{x}$ *and* $\mathbf{q} \mapsto (H_i(\mathbf{x}, \mathbf{q}))$ *is the transposed map* $\varphi^t$*.*

## 6.4 From interpolation to evaluation

We now prove Theorem 6: given an algorithm that performs interpolation, possibly on some distinguished families of points only, one deduces an algorithm which performs evaluation, on the same families of points, and with essentially the same complexity.

The reduction follows from the following matrix identity, which appeared in [49]:

$$\begin{bmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ x_0^n & \dots & x_n^n \end{bmatrix} \begin{bmatrix} 1 & \dots & x_0^n \\ \vdots & & \vdots \\ 1 & \dots & x_n^n \end{bmatrix} = \begin{bmatrix} s_0 & \dots & s_n \\ \vdots & & \vdots \\ s_n & \dots & s_{2n} \end{bmatrix},$$

where $s_i = \sum_{j=0}^{n} x_j^i$ is the $i$th Newton sum of $F = \prod_{i=0}^{n}(T - x_i)$. We rewrite this identity as $(V^t)V = H$, where $H$ is the Hankel matrix made upon $s_0, \dots, s_{2n}$, which in turn yields $V = (V^t)^{-1}H$.

Using this last equality, we deduce the following algorithm to evaluate a polynomial $P$ on the points $\mathbf{x}$; this algorithm appeared originally in [49] (in a "transposed" form).

1. Compute the power sums $s_0, \dots, s_{2n}$ of the polynomial $F = \prod_{i=0}^{n}(T - x_i)$.
2. Compute $\mathbf{p}' = H\mathbf{p}$, where $H$ is as above and $\mathbf{p}$ is the vector of coefficients of $P$.
3. Compute $(V^t)^{-1}\mathbf{p}'$.

Our contribution is the remark that the first step can be essentially reduced to perform a suitable interpolation. Consider indeed $\widetilde{F} = \prod_{i=1}^{n}(T - x_i)$. Then we have the equalities

$$\widetilde{F}(x_0) = \prod_{i=1}^{n}(x_0 - x_i) \quad \text{and} \quad \widetilde{F}(x_i) = 0,\ i > 0.$$

All these values can be computed in time $O(n)$. It suffices to interpolate these values at $x_0, \dots, x_n$ to recover the coefficients of $\widetilde{F}$, since this polynomial has degree $n$. Then, the coefficients of $F$ can be deduced for $O(n)$ additional operations. Finally, we can compute the first $2n+1$ power sums of $F$ for $O(\mathsf{M}(n))$ additional operations using power series expansion, see [212]; this concludes the description of Step 1.

On input the power sums $s_0, \dots, s_{2n}$ and the coefficients of $P$, Step 2 can be done in time $O(\mathsf{M}(n))$ since $H$ is a Hankel matrix, see [25]. It then suffices to perform a transposed interpolation to conclude Step 3.

Formally, let $\Gamma$ be a $\mathbf{Q}$-linear straight-line program of size $L$ as in Theorem 6. Using the composition rules given in the introduction, we deduce from the above discussion that Step 1 can be performed by a straight-line program that takes $\mathbf{X}$ as input, and has size $L+O(\mathsf{M}(n))$. Step 2 can be done by a $\mathbf{P}$-linear straight-line program that takes $s_0, \dots, s_{2n}$ and $\mathbf{P}$ as input, and has size $O(\mathsf{M}(n))$. By Lemma 4, Step 3 can be done by a $\mathbf{P}'$-linear straight-line program of size $L + O(n)$ that takes $\mathbf{X}$ and $\mathbf{P}'$ as input. We conclude the proof by composing these programs.

## 6.5 From evaluation to interpolation

We finally prove Theorem 7: given an algorithm that performs evaluation, possibly on some distinguished families of points only, one deduces an algorithm which performs interpolation, on the same families of points, and with essentially the same complexity. Consider the matrix-vector product

$$\begin{bmatrix} 1 & \dots & x_0^n \\ \vdots & & \vdots \\ 1 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} p_0 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} q_0 \\ \vdots \\ q_n \end{bmatrix}.$$

Our goal is to compute $\mathbf{p}$ on input $\mathbf{q}$. To do so, we first consider the transposed problem, that is, computing $\mathbf{p}'$ on input $\mathbf{q}$, where $\mathbf{p}'$ is defined by

$$\begin{bmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ x_0^n & \dots & x_n^n \end{bmatrix} \begin{bmatrix} p'_0 \\ \vdots \\ p'_n \end{bmatrix} = \begin{bmatrix} q_0 \\ \vdots \\ q_n \end{bmatrix}. \tag{6.1}$$

To solve this question, we use a reduction that appeared in [127] (see also [189] for an alternative formula originating from [104], which requires essentially the same operations). The generating series $\mathcal{Q} = \sum_{i=0}^{n} q_i T^i$ satisfies the following identity:

$$\mathcal{Q} \cdot \prod_{i=0}^{n} (1 - x_i T) = \sum_{i=0}^{n} \left( p'_i \prod_{0 \le j \le n}^{j \neq i} (1 - x_j T) \right) \mod T^{n+1}.$$

Let us next define

$$I = \sum_{i=0}^{n} \left( p'_i \prod_{0 \le j \le n}^{j \neq i} (T - x_j) \right) \quad \text{and} \quad F = \prod_{i=0}^{n} (T - x_i).$$

Then we have $p'_i = I(x_i)/F'(x_i)$. We deduce the following algorithm for recovering $p'_0, \dots, p'_n$ from $q_0, \dots, q_n$. This originally appeared in [127] and follows [267].

1. Compute $F = \prod_{i=0}^{n} (T - x_i)$ and $G = T^{n+1} F(1/T) = \prod_{i=0}^{n} (1 - x_i T)$.
2. Compute $H = \mathcal{Q} \cdot G \mod T^{n+1}$.
3. Evaluate $I = T^n H(1/T)$ and $F'$ on $x_0, \dots, x_n$ and output $I(x_i)/F'(x_i)$.

As in the previous section, our contribution lies in Step 1: we show that computing $F$ is not more costly than performing an evaluation and some polynomial multiplications.

Indeed, let us compute the transposed evaluation on the set of points $x_0, \dots, x_n$ with input values $x_0, \dots, x_n$: this gives the first $n+2$ Newton sums of $F$. Then we can recover the coefficients of the polynomial $F$ using the exponentiation algorithm of [42, 212], which requires $O(\mathsf{M}(n))$ operations.

This concludes the description of Step 1. Step 2 can then be done for $\mathsf{M}(n)$ operations, and Step 3 for two multipoint evaluations plus $n+1$ scalar divisions.

Formally, let $\Delta$ be a $\mathbf{P}$-linear straight-line program of size $L$ as in Theorem 7. Using Lemma 4 and the composition rules, the above discussion shows that the coefficients of $\mathbf{F}$ can be computed by a straight-line program that takes $\mathbf{X}$ as input and has size $L+O(\mathsf{M}(n))$. Step 2 can be done by a $\mathbf{Q}$-linear straight-line program of size $\mathsf{M}(n)$ that takes the coefficients of $F$ and $\mathcal{Q}$ as input. Finally, Step 3 can be done by a $\mathbf{H}$-linear straight-line program that takes $\mathbf{X}$ and the coefficients of $F$ and $H$ as input, and has size $L+O(n)$.

By composition, we obtain a $\mathbf{Q}$-linear straight-line program $\Gamma$ that takes $\mathbf{X}$ and $\mathbf{Q}$ as input and has size $3L+O(\mathsf{M}(n))$; on an input $\mathbf{x}$ that satisfies the assumptions of Theorem 7, $\Gamma$ computes the values $\mathbf{p}'$ satisfying Equation (6.1), as requested. Applying Lemma 4 to $\Gamma$ concludes the proof.

# Part III

# Fast Algorithms for Algebraic Numbers

# Chapter 7

# Fast Computation with Two Algebraic Numbers

We propose fast algorithms for computing *composed products* and *composed sums*, as well as *diamond products* of univariate polynomials. These operations correspond to special resultants, that we compute using power sums of roots of polynomials, by means of their generating series. This chapter is joint work with Ph. Flajolet, B. Salvy and É. Schost.

## Contents

---

## 7.1 Introduction

Let $k$ be a field and let $f$ and $g$ be monic polynomials in $k[T]$, of degrees $m$ and $n$ respectively. We are interested in computing efficiently their *composed sum* $f \oplus g$ and *composed product* $f \otimes g$. These are polynomials of degree $D = mn$ defined by

$$f \oplus g = \prod_{\alpha,\beta} \big(T - (\alpha + \beta)\big) \quad \text{and} \quad f \otimes g = \prod_{\alpha,\beta} (T - \alpha\beta),$$

the products running over all the roots $\alpha$ of $f$ and $\beta$ of $g$, counted with multiplicities, in an algebraic closure $\overline{k}$ of $k$.

More generally, given a bivariate polynomial $H \in k[X, Y]$, of degree less than $m$ in $X$ and of degree less than $n$ in $Y$, we study the fast computation of the *diamond product* $f \diamond_H g$ of $f$ and $g$, which is the polynomial of degree $D = mn$ defined by

$$f \diamond_H g = \prod_{\alpha,\beta} \big(T - H(\alpha, \beta)\big), \tag{7.1}$$

the product running over all the roots $\alpha$ of $f$ and $\beta$ of $g$, counted with multiplicities.

The operation $\diamond_H$ was introduced by Brawley and Carlitz in [39]. They showed that if $k$ is finite, for large families of polynomials $H$, the diamond product enjoys the following remarkable property: $f \diamond_H g$ is irreducible if and only if both $f$ and $g$ are irreducible and their degrees are co-prime. Consequently, diamond products are used for constructing irreducible polynomials of large degree over finite fields, see [39, 40, 222, 224].

These operations, in particular composed sums and composed products, actually appear as basic subroutines in many other algorithms, including computations with algebraic numbers, symbolic summation and study of linear recurrent sequences. We present some of these applications in Section 7.5.

The polynomials $f \oplus g$ and $f \otimes g$ can be expressed in terms of resultants, see for instance [159]:

$$\begin{aligned} (f \oplus g)(x) &= \mathrm{Res}_y(f(x - y), g(y)) \\ (f \otimes g)(x) &= \mathrm{Res}_y(y^m f(x/y), g(y)). \end{aligned} \tag{7.2}$$

A similar (less well-known) formula also holds for $f \diamond_H g$:

$$(f \diamond_H g)(x) = \mathrm{Res}_y\Big(\mathrm{Res}_z\big(x - H(z, y), f(z)\big), g(y)\Big). \tag{7.3}$$

Formulas (7.2) and (7.3) already show that $f \otimes g$, $f \oplus g$ and $f \diamond_H g$ have coefficients in $k$. They also provide a way of computing these polynomials. Still, the efficiency of the resulting algorithms is not entirely satisfactory. For instance, if $f$ and $g$ have degrees of order $\sqrt{D}$, the fastest available algorithms for multivariate resultants [218, 154, 199, 255, 155] based on formulas (7.2) have complexity of order $O_{\log}\big(D\,\mathsf{M}(\sqrt{D}))\big)$ field operations, while that exploiting formula (7.3) has complexity $O_{\log}\big(D^2\,\mathsf{M}(\sqrt{D})\big)$.

Indeed, suppose that $m$ and $n$ are of order $\sqrt{D}$. We estimate the complexity of computing $f \diamond_H g$ using formula (7.3) and an evaluation-interpolation method. We treat only the simpler case when the base field $k$ has at least $D+1$ distinct elements $x_0, \ldots, x_D$. Denote $R_j(y) = \mathrm{Res}_z\big(x_j - H(y,z), f(z)\big)$, for $0 \leq j \leq D$. Then evaluating $f$ at the points $x_j$ amounts to computing the values $\delta_j = \mathrm{Res}_y(R_j(y), g(y))$. It is easy to see that each polynomial $R_j$ has degree at most $D$, so that the computation of $\delta_0, \ldots, \delta_D$ requires $O(D\,\mathsf{M}(D)\log D)$ operations in $k$. Since interpolating $f \diamond_H g$ at the points $x_j$ has cost $O(\mathsf{M}(D)\log D)$, it remains to determine the complexity of computing the polynomials $R_j$. This can be done again by evaluation-interpolation at the points $x_0, \ldots, x_D$, using the equality $R_j(x_i) = \mathrm{Res}_z\big(x_j - H(x_i,z), f(z)\big)$. The polynomials $H(x_i,z)$ can be computed naively in $O(D^2)$, so the cost of computing $R_j(x_i)$ is dominated by the $mn = D$ resultants of polynomials of degree $\sqrt{D}$, that is, $O_{\log}(D^2\,\mathsf{M}(\sqrt{D}))$.

In this chapter $\mathsf{M}(D)$ stands for the number of base field operations required to perform the product of two polynomials of degree $D$ and, also, the first $D$ coefficients in the product of two formal power series given at precision $D$. The symbol $O_{\log}$ indicates the presence of logarithmic terms in $D$.

Over fields of characteristic zero, a different approach for computing composed sums and products was suggested by Dvornicich and Traverso [74]. The key idea is to represent polynomials by the power sums of their roots; for convenience, this will be called *Newton representation*. Dvornicich and Traverso gave formulas expressing $f \oplus g$ and $f \otimes g$ in terms of $f$ and $g$ in this alternative representation. However, a direct application of their formulas, combined with the use of Newton formulas for conversions between Newton representation and the monomial one, lead them to algorithms using $O(D^2)$ operations in $k$.

Brawley, Gao and Mills proposed in [40] several algorithmic solutions for the composed product and sum over a finite field. Apart from the resultant method described below, their most efficient solution has quadratic complexity in the degree $D$ of the output and works only under the assumption of an irreducible output.

In [40, Section 3], Brawley, Gao and Mills also considered the problem of computing the diamond product. Their algorithm works over a finite field with $q$ elements and has complexity $O_{\log}\big(D\log(q) + D^3\big)$, if $f$ and $g$ have degrees of order $\sqrt{D}$.

### Our contribution

In this chapter our aim is to show that better can be done, both in characteristic zero and in positive characteristic. One of the algorithmic keys of our approach is the use of fast algorithms in [212, 188], for converting a polynomial from the classical monomial representation to its Newton representation and backwards.

Another crucial ingredient is our reformulation, in terms of generating series, of some formulas in [74] expressing $f \otimes g$ and $f \oplus g$ in their Newton representation, in characteristic zero or large enough. This enables us to give *nearly optimal* algorithms for the composed product and the composed sum, if the characteristic $\mathrm{char}(k)$ of the base field is zero or large enough. Our algorithms use mainly multiplications, inversions and exponentiations of power series, for which nearly optimal algorithms are known [228, 138, 42], see also [118, Section 13.9], [25], [47, Chapter 2], [255, Section 9.1]. In this chapter, "nearly optimal"

means that the number of operations in the base field $k$ is linear in the size of the result, up to logarithmic factors.

Our algorithm for the composed product can be very slightly modified so as to work in arbitrary characteristic, but the situation is different for the composed sum. By introducing a new combinatorial idea, we reduce the computation of composed sums in small characteristic $p$ to multiplying two multivariate power series of degree less than $p$ in each variable. The resulting algorithm is faster than the existing methods. However, it is not nearly optimal, since no nearly optimal algorithms for such multivariate power series multiplication are known to us. Any improvement on the latter problem would have a positive impact on our algorithm.

We also propose a fast algorithm for computing the general diamond operation. The key point of our method consists in relating the Newton representation of $f \diamond_H g$ to the traces of multiplication by successive powers of $H$ in the quotient algebra $k[X,Y]/(f(X), g(Y))$. This way, the computation of $f \diamond_H g$ reduces roughly to solving a particular instance of the *power projection problem* in $k[X,Y]/(f(X), g(Y))$. For the latter problem, we propose an explicit algorithm using $O\left(\sqrt{D}\big(\mathsf{M}(D) + D^{\omega/2}\big)\right)$ operations in $k$. Here $\omega$ denotes a feasible exponent of matrix multiplication over the field $k$, that is, a positive real number such that any two $n \times n$ matrices over $k$ can be multiplied using $O(n^\omega)$ operations in $k$.

Combining our algorithm for the power projection problem in $k[X,Y]/(f(X), g(Y))$ with the fast conversion techniques mentioned above, we obtain an algorithm for the diamond product of complexity $O\left(\sqrt{D}\big(\mathsf{M}(D) + D^{\omega/2}\big)\right)$. Taking the best upper bound known to this date $\omega < 2.376$ [69], and using Fast Fourier Transform for the power series multiplication, for which $\mathsf{M}(D) = O\big(D \log(D) \log\log(D)\big)$, see [255, Section 8.2], the theoretical complexity of our algorithm is in $O(D^{1.688})$ [1]. Note that even using naive matrix multiplication ($\omega = 3$) this improves the previously known complexity results for the computation of the diamond product roughly by a factor of $\sqrt{D}$.

It should be noted that the same complexity result for the composed product is already (implicitly) given in [224], in a slightly more general context, see also [124]. Nevertheless, the result in [224] is an existence theorem, as no explicit algorithm with this complexity is given. On the contrary, our algorithm is completely explicit; we refer to Section 7.4 for more developped historical notes on this subject.

One of our aims has been to demonstrate the practical relevance of fast algorithms, in particular, that of Strassen's matrix multiplication [234], whose exponent is $\omega = \log_2(7) \simeq 2.808$. In Section 7.4 we also report the practical achievement of our sub-quadratic algorithm for the diamond product.

**Complexity statements.**

We encapsulate our complexity results in the following theorem. Our algorithms for the composed sums and products and for the diamond product work under no additional assumption

[1] The exponent 1.688 may be slightly improved to 1.667 by using the faster algorithms for rectangular matrix multiplication by Huang and Pan [121].

if the base field has characteristic zero or large enough. Over fields of small positive characteristic, they require a mild assumption, which is satisfied, for instance, if the output is an irreducible polynomial.

**Theorem 8** *Let $k$ be a field of characteristic $p$ and let $f$ and $g$ be two monic polynomials in $k[T]$ of degrees $m$ and $n$. Let $D = mn$.*

1. *If $p = 0$ or $p > D$, then the composed operations $f \otimes g$ and $f \oplus g$ can be performed using $O(\mathsf{M}(D))$ operations in $k$;*
2. *If $p$ is larger than all the multiplicities of the roots of $f \otimes g$, then the composed product $f \otimes g$ can be computed within $O\left(p\,\mathsf{M}\left(\frac{D}{p}\right)\log\left(\frac{D}{p}\right) + \mathsf{M}(D)\right)$ operations in $k$;*
3. *If $p$ is larger than all the multiplicities of the roots of $f \oplus g$, then the composed sum $f \oplus g$ can be computed within $O\left(p\,\mathsf{M}\left(\frac{D}{p}\right)\log\left(\frac{D}{p}\right) + \mathsf{M}\left(D^{1+\frac{1}{\log(p)}}\right)\right)$ operations in $k$;*

*Suppose that $H \in k[X, Y]$ has degree less than $m$ in $X$ and degree less than $n$ in $Y$. Then one can compute:*

4. *the diamond product $f \diamond_H g$ using $O\left(\sqrt{D}\left(\mathsf{M}(D) + D^{\omega/2}\right)\right)$ operations in $k$, if $p$ is zero or larger than all the multiplicities of the roots of $f \diamond_H g$ (see comments below).*

We stress the fact that taking $\mathsf{M}(D) \in O_{\log}(D)$ justifies our claims on the near optimality of our algorithms for the composed product and sum.

Table 7.1 aims at clarifying our contributions to the state of the art on the questions addressed in this chapter. The complexities are stated in terms of the degree $D$ of the output. The entries in the case of the arbitrary characteristic are valid under the assumptions of Theorem 8.

Finally, let us mention again that the existence of an algorithm computing the diamond product within the above complexity bound was proved in [224], our contribution being a first explicit and practical algorithm. The star in Table 7.1 refers to this fact.

### Outline of the chapter

- In Section 7.2, we recall fast algorithms for the translation between Newton representation and classical representation of univariate polynomials. Depending on the characteristic of the base field, we split the presentation into two cases, which are detailed in Subsections 7.2.1 and 7.2.2.

- In Section 7.3 we use these results to compute the composed product and sum, and we present the experimental behavior of the resulting algorithms.

| $\mathrm{char}(k)$ | $f \otimes g$ | $f \oplus g$ |
|---|---|---|
| $\substack{p=0 \\ p>D}$ | $O\big(\mathsf{M}(D)\big)$ | $O\big(\mathsf{M}(D)\big)$ |
| $p > 0$ | $O\left(p\,\mathsf{M}\left(\frac{D}{p}\right)\log\left(\frac{D}{p}\right) + \mathsf{M}(D)\right)$ | $O\left(p\,\mathsf{M}\left(\frac{D}{p}\right)\log\left(\frac{D}{p}\right) + \mathsf{M}\big(D^{1+\frac{1}{\log(p)}}\big)\right)$ |

| $\mathrm{char}(k)$ | $f \diamond_H g$ |
|---|---|
| any | $O\left(\sqrt{D}\big(\mathsf{M}(D) + D^{\omega/2}\big)\right)$ $\star$ |

Table 7.1: A bird's eye-view on our contribution. See comments after Theorem 8 concerning the starred entry.

- In Section 7.4 we address the problem of computing the diamond product $f \diamond_H g$. We show that it amounts to evaluating the trace form on the successive powers of $H$ in the quotient algebra $k[X,Y]/(f(X), g(Y))$. This is a particular instance of the power projection problem; we solve it using an effective bivariate version of a "baby-step / giant-step" algorithm of Shoup's. We conclude the section by experimental results.

- Section 7.5 presents several applications of these composed operations and describes two related questions: the fast computation of resolvents and of Graeffe polynomials.

**Notation**

In the rest of this chapter, we make use of the following notation:

- $N_s(h)$ denotes the $s$th *power sum* of the roots of a polynomial $h \in k[T]$, that is, the sum $\sum_\gamma \gamma^s$, taken over all the roots of $h$ in $\overline{k}$, counted with multiplicities.

- The *Newton series* $\mathrm{Newton}(h)$ of $h \in k[T]$ is the power series $\sum_{s\geq 0} N_s(h)T^s$.

- If $P$ is a polynomial in $k[T]$ of degree at most $n$, we write $\mathrm{rev}(n, P)$ for its $n$th *reversal*, namely $P\big(\frac{1}{T}\big)T^n$.

- For $h > l \geq 0$, we introduce the operations $\lceil . \rceil^h$ and $[.]_l^h$ on $P = \sum_i p_i T^i$:

$$\lceil P \rceil^h = \sum_{i=0}^{h-1} p_i T^i,\ [P]_l^h = \sum_{i=0}^{h-l-1} p_{i+l} T^i.$$

- Given a power series $S = \sum_{i \geq 0} s_i T^i \in k[[T]]$ and an integer $m \geq 1$, we write $S \bmod T^m$ for the truncated power series $\sum_{i \geq 0}^{m-1} s_i T^i$.
- By $\lfloor q \rfloor$ we denote the integer part of a rational number $q$.
- For a $k$-vector space $V$, we denote by $\widehat{V}$ its dual, that is, the $k$-vector space of $k$-linear maps $\ell : V \to k$.
- For any integer $m \geq 1$, we write $k[T]_{<m}$ for the set of polynomials of degree less than $m$ and $k[[T]]_{<m}$ for the set of power series truncated at $T^m$. They are dual $k$-vector spaces.

## 7.2 Fast Conversion Algorithms between Polynomials and Power Sums

As mentioned in the Introduction, our acceleration in computing composed and diamond operations is based on the use of an alternative representation for univariate polynomials, the *Newton representation* by power sums of roots.

The use of the Newton representation for polynomials is certainly not our innovation. It is already present in [147], but also in [74, 144, 244, 102, 240, 213, 108, 106, 107, 204, 250, 44], ... Following [74], our contribution is to demonstrate that it provides the appropriate data structure for the efficient computation of composed sums and products and of diamond products.

A polynomial of degree $D$ is uniquely determined by the first $D$ power sums of its roots, at least in characteristic zero or larger than $D$. Moreover, Newton formulas provide a straightforward algorithm to make these conversions, which has quadratic complexity in the degree $D$.

Fortunately, faster conversion methods exist. As far as we know, Schönhage [212] was the first to propose such methods, in the context of devising numerical root-finders for univariate polynomials. Over fields of characteristic zero, his algorithms extend to an exact setting as well, see [185, Appendix A] and [25, Problem 4.8]. Moreover, Schönhage's algorithm for translating a polynomial to its Newton representation remains valid over fields of any characteristic, while his algorithm for the converse direction works over fields of characteristic large enough.

The question of converting power sums of roots to coefficients for polynomials over fields of small characteristic is more delicate and many efforts have been done to bypass its difficulty. Historically, two kinds of approaches were proposed: on the one hand, the techniques of *recursive triangulation* originating in [125], on the other hand, those using *fundamental sets of power sums* of [213, 25, 186, 188]. The best currently known solution is that of [188].

Thus, the results of this section are not new. The conversions algorithms described hereafter will be used as basic algorithmic bricks in the rest of our chapter. For the sake of completeness, we gathered them together, under the shape of ready-to-implement pseudo-code.

The structure of this section is as follows: we begin by recalling an algorithm for the direct conversion (from a polynomial to its Newton representation), which works in arbitrary characteristic. Next, we detail the algorithm in [212] for the inverse conversion in characteristic zero or large enough. We conclude the section by presenting an algorithm for the inverse conversion in the positive characteristic setting. This algorithm is due to Pan [188]. It inherits mathematical ideas of Schönhage [213] and successive algorithmic improvements [25, 186, 187].

## From monomial to Newton representation

The translation from a polynomial to the power sums of its roots is quite simple and is based on the following basic result.

**Lemma 5** *Let $h$ be a monic polynomial in $k[T]$, of degree $D$. Then, the series* $\mathrm{Newton}(h)$ *is rational; moreover, the following formula holds:*

$$\mathrm{Newton}(h) = \frac{\mathrm{rev}(D-1, h')}{\mathrm{rev}(D, h)}.$$

*Proof.* Let $\gamma_1, \ldots, \gamma_D$ be the roots of $h$ in $\overline{k}$. Since $h = \prod_{i=1}^{D}(T - \gamma_i)$, we have

$$\mathrm{Newton}(h) = \sum_{s \geq 0} \Big( \sum_{i=1}^{D} \gamma_i^s \Big) T^s = \sum_{i=1}^{D} \Big( \sum_{s \geq 0} \gamma_i^s T^s \Big) = \sum_{i=1}^{D} \frac{1}{1 - \gamma_i T} = \frac{\mathrm{rev}(D-1, h')}{\mathrm{rev}(D, h)}.$$

□

**Proposition 5** *If $h$ is a polynomial of degree $D$ in $k[T]$ and if $N \geq D$, then the first $N$ power sums of the roots of $h$ can be computed within $O\big(\frac{N}{D}\mathsf{M}(D)\big)$ operations in $k$.*

*Proof.* By the preceding Lemma, it is sufficient to prove that if a polynomial $A$ has degree at most $D-1$ and if a polynomial $B$ has degree $D$, then the first $N \geq D$ coefficients of the rational series $A/B$ can be computed within the announced running time bound. The idea is to proceed by *slices* of size $D$. We first compute the first $D$ coefficients of $1/B$, using Sieveking-Kung's algorithm [228, 138], for a cost of $O(\mathsf{M}(D))$ operations in $k$. We denote $B_0$ the corresponding polynomial, of degree $D-1$. We let $C_0 = \lceil AB_0 \rceil^D$ and recursively define the polynomials

$$C_{j+1} = -\left\lceil \lfloor BC_j \rfloor_D B_0 \right\rceil^D, \quad \text{for } 0 \leq j \leq \lfloor N/D \rfloor .$$

Then, it is a simple matter to verify that $\frac{A}{B} = C_0 + T^D C_1 + T^{2D} C_2 + \ldots$ and the result follows. □

We summarize the corresponding algorithm in Figure 7.1.

**Computing the Newton series of a polynomial**

**Input:** a polynomial $h$ of degree $D$.
**Output:** the series Newton($h$) at precision $N$.

$A \leftarrow \mathrm{rev}(D-1, h')$
$B \leftarrow \mathrm{rev}(D, h)$
$B_0 \leftarrow \left\lceil \frac{1}{B} \right\rceil^D$
$C_0 \leftarrow \lceil AB_0 \rceil^D$
$l \leftarrow \left\lfloor \frac{N}{D} \right\rfloor$
`for` $j$ `from` $0$ `to` $l$ `do`
    $C_{j+1} \leftarrow -\left\lceil \lfloor BC_j \rfloor_D B_0 \right\rceil^D$
`return` $\sum_{i=0}^{l} C_i T^{Di} + O(T^N)$

Figure 7.1: Computing the Newton series of a polynomial

## From Newton representation to the monomial one

The converse direction is more difficult to handle: while in characteristic zero the Newton formulas give a one-to-one correspondence between power sums and elementary symmetric sums, in the positive characteristic case distinct monic polynomials of the same degree may have equal power sums of roots. Consequently, the treatment of this question should take into account the characteristic of the base field. The results of the next subsections are summarized in the following proposition.

**Proposition 6** *Let $h$ be a polynomial of degree $D$ in $k[T]$.*

1. *If $k$ has characteristic zero or greater than $D$, then the polynomial $h$ can be computed from the first $D$ power sums of its roots within $O\big(\mathsf{M}(D)\big)$ operations in $k$.*

2. *Suppose that $k$ has positive characteristic $p$ and that all the roots of $h$ have multiplicities less than $p$. Then, the polynomial $h$ can be computed from the first $2D$ power sums of its roots within*
$$O\left(\mathsf{M}(D) + p\,\mathsf{M}\Big(\frac{D}{p}\Big)\log\Big(\frac{D}{p}\Big)\right)$$
*operations in $k$.*

As we already mentioned in the preamble of this section, the results of Proposition 6 are not new; the first part is commonly attributed to Schönhage [212], while the second one is due to Pan [188].

We are now going to discuss these results in some detail and present the corresponding algorithms. In Subsection 7.2.1 we treat the case of characteristic zero or large enough, since we consider that the ideas involved in that case are important and help understanding the extension to the arbitrary positive characteristic case. The latter case is addressed in Subsection 7.2.2, where technical details are intentionally omitted. Instead we preferred to put down a complete pseudo-code, to simplify the task of reading Pan's original paper [188].

### 7.2.1 The case of characteristic zero or large enough

The *exponential* of a power series $F$ with positive valuation over a field $k$ of characteristic zero is given by

$$\exp(F) = \sum_{s \geq 0} \frac{F^s}{s!}.$$

If the base field $k$ has positive characteristic $p$, we define, as an analogue to the exponential of a power series $F \in k[[T]]$ with positive valuation, the series

$$\exp(F) = \sum_{s=0}^{p-1} \frac{F^s}{s!}.$$

Note that if $F$ and $G$ are power series with positive valuation over a field of characteristic $p > 0$, one has the formula $\exp(F+G) = \exp(F) \cdot \exp(G) \mod T^p$.

The next lemma expresses the reverse of a monic polynomial as the exponential of a series involving its Newton series. The subsequent corollary proves the first part of Proposition 6.

**Lemma 6** *Let $h$ be a monic polynomial of degree $D$ in $k[T]$, where $k$ is a field of characteristic zero or larger than $D$. Then the following formula holds:*

$$\mathrm{rev}(D,h) = \exp\left(\int \frac{1}{T} \cdot \Big(D - \mathrm{Newton}(h)\Big)\right).$$

*Proof.* Let $\gamma_1, \ldots, \gamma_D$ be the roots of $h$ in $\overline{k}$. By definition, it follows that:

$$\frac{1}{T} \cdot \Big(D - \mathrm{Newton}(h)\Big) = -\sum_{s \geq 0} N_{s+1}(h) T^s.$$

On the other hand, an immediate calculation shows that:

$$\frac{\mathrm{rev}(D,h)'}{\mathrm{rev}(D,h)} = \sum_i \frac{-\gamma_i}{1-\gamma_i T} = -\sum_{s \geq 0} \Big(\sum_i \gamma_i^{s+1}\Big) T^s. \tag{7.4}$$

As one can easily verify, for a polynomial $P \in k[T]$ with constant coefficient 1, the formula $P = \exp(\int P'/P)$ holds as soon as $k$ has characteristic zero. The same equality remains valid modulo $T^p$ if $\mathrm{char}(k)$ is larger than the degree of $P$. By applying this fact to $P = \mathrm{rev}(D,h)$ in conjunction with the previous two formulas, we conclude the proof of the lemma. □

**Corollary 4** *A monic polynomial $h$ of degree $D$ over a field of characteristic zero or larger than $D$ can be computed from the first $D$ power sums of its roots within $O(\mathsf{M}(D))$ base field operations.*

*Proof.* (Compare [25, p. 34–35]) By assumption, we know the series $\mathrm{Newton}(h)$ at precision $D$, from which we deduce the series $\int \frac{1}{T} \cdot \big[D - \mathrm{Newton}(h)\big]$ at precision $D$ in linear time. By Lemma 6, exponentiating the latter series gives the polynomial $\mathrm{rev}(D, h)$. The exponential of a power series can be computed within $O(\mathsf{M}(D))$ field operations, using a Newton iteration; this was pointed out for the first time by Brent in [42]. Finally, we recover the polynomial $h = \mathrm{rev}\big(D, \mathrm{rev}(D, h)\big)$. □

At this time, the first part of Proposition 6 is proved. We present the resulting algorithm in Figure 7.2. For a polynomial $P$, we denote by $\texttt{Coeff}(P, i)$ the coefficient of $T^i$ in $P$. We use a slightly modified Newton iteration, whose complexity has a better constant factor and which is similar to that suggested in [187, Appendix A].

**Recovering a monic polynomial from its Newton series in characteristic zero**

**Input:** the first $D$ terms of the series $\mathrm{Newton}(h)$.
**Output:** the polynomial $h$.

$S \leftarrow (\mathrm{Newton}(h) - D)/T$
$R \leftarrow 1 - \texttt{Coeff}(S, 0)T$
$n \leftarrow 2$
`while` $n \leq D$ `do`
    $M' \leftarrow -\left\lceil \frac{R'}{R} + S \right\rceil^{2n-1}$
    $M \leftarrow 1 + \sum_i \texttt{Coeff}(M', i)\frac{T^i}{i}$
    $R \leftarrow \lceil RM \rceil^{2n}$
    $n \leftarrow 2n$
$R \leftarrow \lceil R \rceil^{D+1}$
`return` $\mathrm{rev}(D, R)$

Figure 7.2: Recovering a polynomial from its Newton series in characteristic zero

### 7.2.2 The small positive characteristic case – Schönhage-Pan's algorithm

Let $h = \prod_{i=1}^{D}(T - \lambda_i)$ be a polynomial over a field of positive characteristic $p < D$. Schönhage [213] suggested the following method for recovering the polynomial $h$ from its first $2D$ power sums $N_j = \sum_{i=1}^{D} \lambda_i^j$. Write $H$ for $\mathrm{rev}(D, h)$ and decompose it as $\sum_{j=0}^{p-1} H_j(T^p)T^j$. Next, consider the auxiliary power series $Q(T) = \frac{H(T)}{H_0(T^p)} = \sum_{i\geq 0} q_i T^i$. Then, $Q'/Q$ equals $H'/H$ and, by Equation (7.4), the latter coincides with $-\sum_{i\geq 0} N_{i+1}T^i$. Moreover, $Q$ and $H$ coincide up to precision $p$ and $q_0 = 1$, $q_{pi} = 0$, for all $i \geq 1$.

Based on these facts, Schönhage's strategy consists of the following two stages:

1. From the first $2D$ power sums of $h$, determine the first $2D$ coefficients of $Q$.
2. Starting from $Q$, recover the polynomial $h$.

In [213] the first stage was completed using a reduction to a triangular linear system of size $\lfloor D/p \rfloor$, for a total cost of $O\big(\mathsf{M}(D) + (D/p)^2\big)$ operations in $k$. Subsequently, Pan [186] gave an improved solution for the computation of $Q$, by adapting Newton's iteration in the algorithm of Corollary 4 to the positive characteristic setting, thus leading to a cost of $O\left(\mathsf{M}(D)\right)$ operations in $k$ for this first stage.

Concerning the second stage, Schönhage proposed in [213] a solution based on the resolution of a linear system of equations of size $\lfloor D/p \rfloor$. This step was accelerated by Pan in [187], who showed that it amounts to solving $p - 1$ Padé approximation problems of sizes at most $(D/p, D/p)$.

For further technical details, we refer to the original papers [213, 186, 187, 188]. To facilitate the reader's task, we give in Figure 7.3 the algorithm we extracted from [188]. This algorithm takes as input the first $2D$ terms of the series Newton$(h)$ and returns the polynomial $h$. We use the notation $\mathtt{lcm}(f_i)$ for the least common multiple of a family of polynomials $(f_i)$ and $\mathtt{Pade}(S, a, b)$ for the Padé approximant $(A, B)$ of a power series (polynomial) $S$, that is the (unique) pair of polynomials $A$ and $B$ of minimal degrees such that $B(0) = 1$ and such that the following relations hold:

$$A - BS = 0 \mod T^{a+b+1}, \ \deg(A) \leq a, \ \deg(B) \leq b.$$

## 7.3 Two Useful Resultants that Can Be Computed Fast

The results of the preceding section will help us design optimal algorithms for the computation of the composed product $f \otimes g$ and composed sum $f \oplus g$ of two monic polynomials $f$ and $g$ of degrees $m$ and $n$. These are particular instances of resultants

$$(f \otimes g)(x) = \prod_{g(\beta)=0} \beta^m f(x/\beta) = \mathrm{Res}_y\big(y^m f(x/y), g(y)\big),$$

**Schönhage-Pan's algorithm**

**Input:** the first $2D$ terms of Newton$(h)$.
**Output:** the polynomial $h$.

$S \leftarrow (\text{Newton}(h) - D)/T$
$R \leftarrow 1 - \texttt{Coeff}(S, 0)T$
$n \leftarrow 2$
`while` $n \leq 2D$ `do`
    $M' \leftarrow -\left\lceil \frac{R'}{R} + S \right\rceil^{2n-1}$
    $M \leftarrow 1 + \sum_{p \nmid i} \texttt{Coeff}(M', i)\frac{T^i}{i}$
    $g \leftarrow \lfloor \frac{2n-1}{p} \rfloor$
    `if` $gp > n$ `then`
        $A \leftarrow [R]_1^{gp-n+1}\,[M]_n^{gp}$
        $M \leftarrow M - \sum_{i=\lceil \frac{n}{p} \rceil}^{g} \texttt{Coeff}(A, ip - n - 1)T^{ip}$
    $R \leftarrow \lceil RM \rceil^{2n}$
    $n \leftarrow 2n$
`for` $i$ `from` 1 `to` $p - 1$ `do`
    $d_i \leftarrow \left\lfloor \frac{D-i}{p} \right\rfloor$
    $Q_i \leftarrow \sum_{j=0}^{d+d_i} \texttt{Coeff}(R, jp + i)T^j$
    $(A_i, B_i) \leftarrow \texttt{Pade}(Q_i, d_i, d)$
    $H_0 \leftarrow \texttt{lcm}(B_i)$
    $H_i \leftarrow \lceil Q_i H_0 \rceil^{d_i+1}$
$H \leftarrow H_0(T^p) + \sum_{i=1}^{p-1} H_i(T^p)T^i$
`return` rev$(D, H)$

Figure 7.3: Recovering a monic polynomial from its Newton series in small characteristic

and

$$(f \oplus g)(x) = \prod_{g(\beta)=0} f(x - \beta) = \text{Res}_y\big(f(x - y), g(y)\big)$$

that we compute faster than the general bivariate resultants. Our algorithms are based on formulas expressing the Newton series of $f \otimes g$ and of $f \oplus g$ in terms of those of $f$ and $g$. Note that designing nearly optimal algorithms for general bivariate resultants is still an open problem, see [255, Research problem 11.10].

### 7.3.1 Computing the composed product

Our algorithm for the composed product $f \otimes g$ is based on the following lemma:

**Lemma 7** *Let $f$ and $g$ be two polynomials in $k[T]$. Then, the following formula holds:*

$$\mathrm{Newton}(f \otimes g) = \mathrm{Newton}(f) \odot \mathrm{Newton}(g),$$

*where $\odot$ denotes the Hadamard (term-wise) product of power series.*

*Proof.* For $s \geq 0$, the $s$th power sum of the roots of $f \otimes g$ is $\sum_{\alpha,\beta}(\alpha\beta)^s$, the sum running over all the roots $\alpha$ of $f$ and $\beta$ of $g$. This sum can be rewritten as $\big(\sum_\alpha \alpha^s\big) \cdot \big(\sum_\beta \beta^s\big)$, which is the product of the $s$th power sums of the roots of $f$ and of $g$. This proves that the series $\mathrm{Newton}(f \otimes g)$ is the Hadamard product of $\mathrm{Newton}(f)$ and $\mathrm{Newton}(g)$. □

As a corollary, we obtain the following algorithm for $f \otimes g$. Given two monic polynomials $f$ and $g$ of degrees $m$ and $n$, we first compute the power series $\mathrm{Newton}(f)$ and $\mathrm{Newton}(g)$ up to precision $D = mn$. Using Proposition 5, this step requires $O\left(\frac{D}{m}\mathsf{M}(m) + \frac{D}{n}\mathsf{M}(n)\right)$ operations in $k$. Then we perform, for a cost linear in $D$, the Hadamard product of $\mathrm{Newton}(f)$ and $\mathrm{Newton}(g)$. By the preceding lemma, we thus obtain the Newton series of $f \otimes g$ at precision $D$. We recover the polynomial $f \otimes g$ by applying the conversion algorithms in Proposition 6. Summing up the costs of each stage proves the complexity results concerning $f \otimes g$ in the first two assertions in Theorem 8.

### 7.3.2 Computing the composed sum in characteristic zero or large enough

Let $k$ be a field of arbitrary characteristic and let $\mathrm{E} \in k[[T]]$ denote the power series

$$\mathrm{E} = \exp(T),$$

where exp denotes the exponential defined in Section 7.2.1. Then our algorithm for $f \oplus g$ is based on the following lemma:

**Lemma 8** *Let $f$ and $g$ be two polynomials in $k[T]$. Then*

1. *If the characteristic of $k$ is zero, the following formula holds:*

$$\mathrm{Newton}(f \oplus g) \ \odot\ \mathrm{E} = \big(\mathrm{Newton}(f) \ \odot\ \mathrm{E}\big) \cdot \big(\mathrm{Newton}(g) \ \odot\ \mathrm{E}\big);$$

2. *If $p > 0$ is the characteristic of $k$, the following formula holds:*

$$\mathrm{Newton}(f \oplus g) \ \odot\ \mathrm{E} = \big(\mathrm{Newton}(f) \ \odot\ \mathrm{E}\big) \cdot \big(\mathrm{Newton}(g) \ \odot\ \mathrm{E}\big) \mod T^p.$$

*Proof.* We give the proof in the zero characteristic case; the same arguments apply *mutatis mutandis* in the second case, by truncating the series involved judiciously.

The definition of Newton implies that

$$\mathrm{Newton}(f \oplus g) = \sum_{s \geq 0} \left( \sum_{\alpha,\beta} (\alpha + \beta)^s \right) T^s,$$

the second sum running over all the roots $\alpha$ of $f$ and $\beta$ of $g$. The conclusion now reads

$$\sum_{s\geq 0} \frac{\sum_{\alpha,\beta} (\alpha+\beta)^s}{s!} T^s = \left(\sum_{s\geq 0} \frac{\sum_{\alpha} \alpha^s}{s!} T^s\right) \cdot \left(\sum_{s\geq 0} \frac{\sum_{\beta} \beta^s}{s!} T^s\right)$$

and we are done, as the latter equality simply translates the fact that

$$\sum_{\alpha,\beta} \exp\big((\alpha+\beta)T\big) = \Big(\sum_{\alpha} \exp\big(\alpha T\big)\Big) \cdot \Big(\sum_{\beta} \exp\big(\beta T\big)\Big).$$

□

As a corollary, we obtain an algorithm for computing the composed sum of two monic polynomials $f$ and $g$: first, compute Newton($f$) and Newton($g$) to precision $D$, perform their Hadamard product with E, then compute the product of power series in Lemma 8 to recover Newton($f \oplus g$) at precision $D$ and finally convert the last Newton series to the polynomial $f \oplus g$. Using Proposition 5 and Proposition 6, we complete the proof of the first assertion in Theorem 8.

### 7.3.3 Computing the composed sum in small characteristic

In this section we generalize the technique of the preceding paragraphs to the computation of composed sums over fields of small characteristic. Recall that our algorithm in the case of large positive characteristic relies on a combinatorial identity involving the exponential series. Because of factorials, the definition of the latter series is not straightforward in the present case. In what follows we overcome this difficulty by using some multivariate exponential generating series.

We begin by giving the intuition behind our approach on a particular case. Suppose that $f$ and $g$ are polynomials over a field of characteristic $p > 0$; our aim is express the first $p^2$ power sums of the roots $\sum_{\alpha,\beta}(\alpha+\beta)^\ell$ in terms of $\sum_\alpha \alpha^\ell$ and $\sum_\beta \beta^\ell$. The first $p$ of these sums can be determined using the method in the previous section, which amounts to exploit the identity $\exp((\alpha+\beta)T) = \exp(\alpha T) \cdot \exp(\beta T)$. For $\ell$ between $p$ and $p^2-1$, this method fails, since one is not able to divide by $p$ in $k$. In contrast, if we write $\ell$ as $i+pj$ with $i$ and $j$ less than $p$, then the equality $(\alpha+\beta)^\ell = (\alpha+\beta)^i(\alpha^p+\beta^p)^j$ suggests the use of the bivariate identity

$$\exp\big((\alpha+\beta)T + (\alpha+\beta)^p U\big) = \exp(\alpha T + \alpha^p U) \cdot \exp(\beta T + \beta^p U),$$

which translates into the following equality modulo $(T^p, U^p)$

$$\left(\sum_{i,j=0}^{p-1} \Big(\sum_\alpha \alpha^{i+pj}\Big) \frac{T^i U^j}{i!j!}\right) \cdot \left(\sum_{i,j=0}^{p-1} \Big(\sum_\beta \beta^{i+pj}\Big) \frac{T^i U^j}{i!j!}\right) = \sum_{i,j=0}^{p-1} \Big(\sum_{\alpha,\beta} (\alpha+\beta)^{i+pj}\Big) \frac{T^i U^j}{i!j!},$$

helping to find the first $p^2$ power sums of $f \oplus g$ by means of a single multiplication of bivariate power series. We are going now to formalize this idea in the general case.

Let $k$ be a field of characteristic $p > 0$. For any $i \in \mathbb{N}$, we write $\mathbf{i_p} = (i_0, \ldots, i_s)$ for its $p$-adic expansion, that is, the (unique) sequence of integers $0 \leq i_\ell < p$ such that $i =$

$i_0 + i_1 p + \cdots + i_s p^s$. Let $\mathbf{T}$ be an infinite set of indeterminates $(T_i)_{i\geq 0}$. We define *the p-exponential* $\mathrm{E}_p \in k[[\mathbf{T}]]$ as the multivariate power series

$$\mathrm{E}_p \;=\; \sum_{i\geq 0} \frac{\mathbf{T}^{\mathbf{i_p}}}{\mathbf{i_p}!}$$

where for $\mathbf{i_p} = (i_0, \ldots, i_s)$, we denote $\mathbf{i_p}! = i_0! \cdots i_s!$ and $\mathbf{T}^{\mathbf{i_p}} = T_0^{i_0} \cdots T_s^{i_s}$.

For a univariate polynomial $f \in k[T]$, we call the *p-Newton series of f* the series

$$\mathrm{Newton}_p(f) \;=\; \sum_{i\geq 0} N_i(f)\mathbf{T}^{\mathbf{i_p}}.$$

By definition, the degree of $\mathrm{Newton}_p(f)$ is smaller than $p$ in each variable.

With these notation, our algorithm for $f \oplus g$ in small characteristic is based on the following results, which can be seen as a generalization of Lemma 8.

**Lemma 9** *Let $k$ a field of characteristic $p$ and let $f$ be a polynomial in $k[T]$. Then:*

$$\mathrm{Newton}_p(f) \odot \mathrm{E}_p \;=\; \sum_{s\geq 0} \sum_{f(\alpha)=0} \left( \prod_{\ell=0}^{s} \exp(\alpha^{p^\ell} T_\ell) \right),$$

*where $\odot$ denotes the term-wise product of multivariate power series.*

*Proof.* By definition, the left-hand side equals

$$\sum_{\substack{i\geq 0 \\ \mathbf{i_p}=(i_0,\ldots,i_s)}} \frac{N_i(f)}{i_0! \cdots i_s!}\, T_0^{i_0} \cdots T_s^{i_s} \;=\; \sum_{s\geq 0} \sum_{f(\alpha)=0} \left( \sum_{\substack{0\leq i_0<p \\ \cdots \\ 0\leq i_s<p}} \frac{\alpha^{i_0+i_1 p+\cdots+i_s p^s}}{i_0! \cdots i_s!}\, T_0^{i_0} \cdots T_s^{i_s} \right).$$

Since, for any $\ell \geq 0$, the summation indices $i_\ell$ vary independently and since

$$\sum_{0\leq i_\ell<p} \frac{\alpha^{i_\ell p^\ell}}{i_\ell!} T_\ell^{i_\ell} = \exp(\alpha^{p^\ell} T_\ell),$$

the lemma follows. □

**Lemma 10** *Let $k$ be a field of characteristic $p > 0$ and let $f$ and $g$ be two monic polynomials in $k[T]$. Then the following identity holds:*

$$\mathrm{Newton}_p(f \oplus g) \odot \mathrm{E}_p \;=\; \big(\mathrm{Newton}_p(f) \odot \mathrm{E}_p\big) \cdot \big(\mathrm{Newton}_p(g) \odot \mathrm{E}_p\big) \mod (\mathbf{T}^p),$$

*where $(\mathbf{T}^p)$ denotes the ideal generated by the monomials $T_0^p, \ldots T_s^p, \ldots$ in $k[[\mathbf{T}]]$.*

*Proof.* By Lemma 9, we can rewrite the left-hand side as

$$\sum_{s\geq 0} \sum_{\substack{f(\alpha)=0\\ g(\beta)=0}} \exp\big((\alpha+\beta)T_0\big)\cdots\exp\big((\alpha+\beta)^{p^s}T_s\big)$$

and the right-hand side as

$$\sum_{s\geq 0} \sum_{\substack{f(\alpha)=0\\ g(\beta)=0}} \exp(\alpha T_0)\exp(\beta T_0)\cdots\exp(\alpha^{p^s}T_s)\exp(\beta^{p^s}T_s).$$

Using the fact that the series

$$\exp\big((\alpha+\beta)^{p^\ell}T_\ell\big) \;=\; \exp\big(\alpha^{p^\ell}T_\ell+\beta^{p^\ell}T_\ell\big)$$

is equal to $\exp(\alpha^{p^\ell}T_\ell)\exp(\beta^{p^\ell}T_\ell)$ modulo $T_\ell^p$ for any $\ell\geq 0$, the conclusion of the lemma follows. □

Via the fast conversion algorithms in Section 7.2, the preceding lemma enables us to reduce the computation of the composed sum of characteristic $p>0$ to a single multiplication of multivariate series involving a *finite number of variables* and of degree less than $p$ in each variable. More precisely, to recover the polynomial $f\oplus g$, only $2D$ terms of its Newton series suffice, where $D=\deg(f\oplus g)$, and this means that in the $p$-Newton series of $f\oplus g$, all we need to know are coefficients of the monomials containing $T_0,\ldots,T_s$, where $s=\lfloor\log(2D)/\log(p)\rfloor$. By Lemma 10, this can be done by multiplying two multivariate power series involving at most

$$\log(2D)/\log(p)$$

variables and of degree less than $p$ in each variable.

Given two multivariate power series $f$, $g$ in $k[[T_0,\ldots,T_s]]$ of degree less than $p$ in each variable, a naive multiplication routine has quadratic complexity in the size $p^s$ of the output $fg$. Improving this running time bound is an important question in itself. At this time, no nearly optimal algorithm is available, contrary to the univariate case. We refer to [25, Section 1.8] for more details.

Our solution relies on Kronecker's substitution (see [137] and [255, Section 8.4]), which allows to find the product of multivariate power series by performing a multiplication of univariate power series of degree less than $N=2^s p^{s+1}$. The key point is the use of the *substitution* map

$$\begin{array}{rccc} \psi: & k[[T_0,\ldots,T_s]] & \to & k[T]_{<N} \\ & T_i & \mapsto & T^{(2p-1)^i}, \end{array}$$

which provides an encoding of multivariate power series into univariate ones and which behaves well under multiplication. Indeed, the coefficients of a multivariate product $fg$ can be read off the coefficients of the univariate product $\psi(f)\psi(g)$ of the images of $f$ and $g$ under this substitution. The cost of this multiplication method is $O\big(\mathsf{M}(2^s p^{s+1})\big)$ operations in $k$.

In our case, we deal with series with at most $\log(2D)/\log(p)$ variables of degree less than $p$ in each variable, so their product can be performed using Kronecker's substitution within

$$O\big(\mathsf{M}(2^s p^{s+1})\big) = O\Big(\mathsf{M}\big(D^{1+\frac{1}{\log(p)}}\big)\Big)$$

operations in $k$.

We summarize our algorithm in Figure 7.4. To simplify notation, we write

$$[a]_p^q \;=\; a_0 + a_1 q + \cdots + a_s q^s,$$

where $\sum_{i=0}^{s} a_i p^i$ is the $p$-adic expansion of the integer $a$. `NewtonToPol` denotes the procedure described in Section 7.2.2, which outputs a polynomial, given its Newton representation.

**Computing the composed sum in small characteristic**

**Input:** $f$, $g$ in $k[T]$, $\mathrm{char}(k) = p > 0$.
**Output:** the polynomial $f \oplus g$.

$D \leftarrow \deg(f)\deg(g)$
$A_1 \leftarrow \displaystyle\sum_{i=0}^{2D} \frac{N_i(f)}{\mathbf{i_p}!} T^{[i]_p^{2p-1}}$
$A_2 \leftarrow \displaystyle\sum_{i=0}^{2D} \frac{N_i(g)}{\mathbf{i_p}!} T^{[i]_p^{2p-1}}$
$A \leftarrow A_1 \cdot A_2$
$R \leftarrow \displaystyle\sum_{i=0}^{2D} \mathtt{Coeff}(A, [i]_p^{2p-1})\ \mathbf{i_p}!\ T^i$
`return` $\mathtt{NewtonToPol}(R)$

Figure 7.4: Our algorithm for the composed sum in small characteristic

The first $2D$ terms $\frac{1}{\mathbf{i_p}!}$ can be computed iteratively using $O(D)$ operations. Taking into account the cost of conversions between coefficients and power sums, this concludes the proof of the third part in Theorem 8.

### 7.3.4 Experimental results

We have implemented the algorithms for the composed sum and product, in their versions for large or zero characteristic. We used the NTL C++ package as a basis [226].

Since our complexity estimates are stated in terms of number of operations in the base field, we have chosen to experiment on finite fields of the form $\mathbb{Z}/p\mathbb{Z}$, with $p$ prime. For such base fields, NTL implements polynomial arithmetic using classical, Karatsuba and Fast Fourier Transform multiplications.

For the tests presented in Figure 7.5, the input polynomials have equal degrees $m = n$ and their coefficients are chosen randomly; the output has degree $D = m^2$. We let $m$ vary from 1 to 500 by steps of 8, so that $D$ varies from 1 to 250000; the base field is defined by a 32 bit prime. We stress the fact that such output degrees are met in applications, see Section 7.5.

All the tests have been performed on the computers of the MEDICIS resource center[2], using a 2 GB, 2200+ AMD Athlon processor.

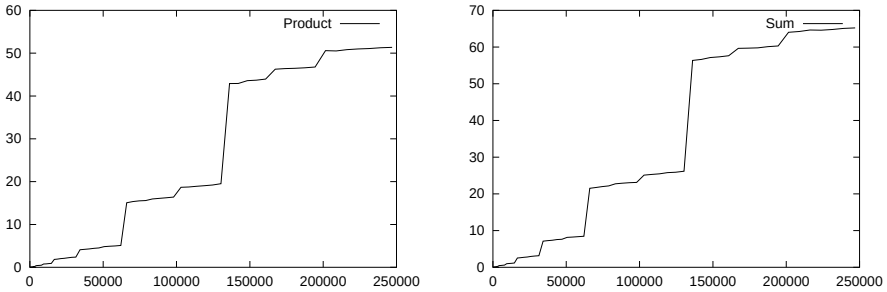



Figure 7.5: Composed product and sum. (Time in sec. vs output degree)

In both cases (composed product and composed sum), the running time presents an abrupt jump when the output degree $D$ passes a power of 2. This feature is actually already present in the polynomial multiplication, and is intrinsic to NTL's Fourier Transform: see Figure 7.6, which displays the time for one polynomial multiplication, in the same degree range as Figure 7.5. We also plot the ratios between the times of composed sum (resp. product) and polynomial multiplication. For large degrees, the ratios do not exceed 5.

We finally give the timings given by NTL's `Resultant` computation. NTL implements both quadratic and fast resultant algorithms, the latter being used for $m$ larger than 180, i.e. $D$ larger than 32400. The experimental times are given in Figure 7.7; in large degrees, the resultant computations take several hours, whereas our algorithms take approximately one minute.

## 7.4 Computing the Diamond Product

We finally address the general case: computing the *diamond product* of $f$ and $g$. From the data of a polynomial $H(X, Y)$, of degree less than $m$ in $X$ and less than $n$ in $Y$, the diamond

[2]http://www.medicis.polytechnique.fr

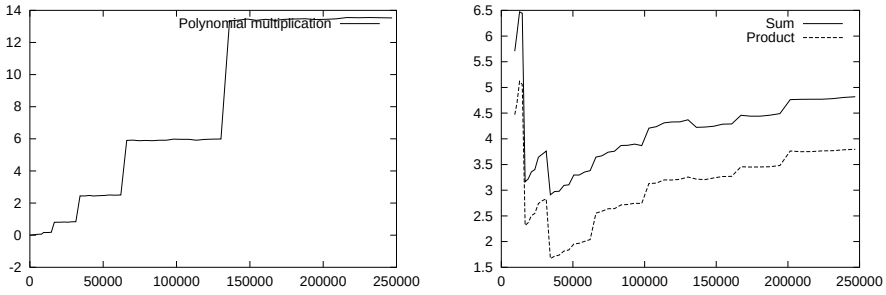



Figure 7.6: Left: polynomial multiplication (Time in sec. vs output degree). Right: (Composed product or sum time) / (Multiplication time) vs output degree.

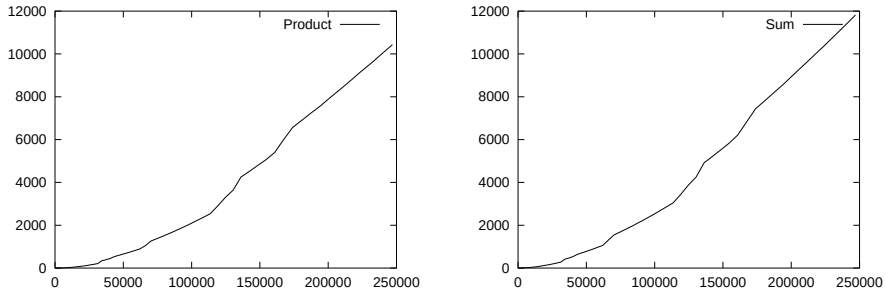



Figure 7.7: Composed sum and product by resultant computation. (Time in sec. vs output degree)

product is defined as the polynomial of degree $D = mn$

$$f \diamond_H g = \prod_{\alpha,\beta} \big(T - H(\alpha, \beta)\big),$$

where the product runs over all the roots $\alpha$ of $f$ and $\beta$ of $g$, counted with multiplicities. In this section, we give an algorithm that computes $f \diamond_H g$ using

$$O\left(\sqrt{D}\big(\mathsf{M}(D) + D^{\omega/2}\big)\right)$$

operations in $k$, where $\omega$ is the exponent of matrix multiplication [47, Chapter 15].

Anticipating the following section, $f \diamond_H g$ is the characteristic polynomial of $H$ in the quotient algebra $k[X, Y]/(f(X), g(Y))$. This characterization of the diamond product enables us to reduce its computation to that of the *power projections* of $H$ under the *trace* map, followed by a linearly generated sequence recovery.

The origins of this approach trace back at least to Le Verrier [147] (see also [213]) who proposed a method for computing characteristic polynomials of matrices by means of traces of matrix powers and using Newton identities for the recovery step. The idea of using power

projections for computing minimal polynomials in quotient algebras appears in [240, 200] in the one variable case $k[X]/(f(X))$. A breakthrough was achieved in [224] by Shoup, who was the first to notice that the power projection problem is *dual* to the *modular composition problem*, see also Kaltofen [124]. Moreover, combining a complexity result of Brent & Kung [43] for the latter problem with a general algorithmic theorem, called *Tellegen's principle*, Shoup proved the *theoretical* existence of an algorithm solving the power projection problem within the same complexity as ours, even for the more general algebra $k[X,Y]/(f(X),g(X,Y))$. Still, Shoup gave no explicit algorithm. In [225, Section 4.1 and 7.5] he partially filled in this gap and proposed a "baby step/giant step" algorithm for the power projection in the univariate case, yet only in the FFT polynomial multiplication setting; in a subsequent paper [227, Section 2.2], Shoup extended his algorithm to the bivariate case, and independently of the polynomial multiplication model. Yet, the complexity of algorithms in [225, 227] is higher, the term $D^{\omega/2}$ being replaced by $D^{3/2}$.

In what follows, we *explicitly* solve the power projection problem for our special quotient algebra $Q = k[X,Y]/(f(X),g(Y))$ within the complexity predicted by Tellegen's principle. This is done by applying some effective transposition techniques [35] to Brent & Kung's algorithm for modular composition algorithm in $Q$. We point out that similar ideas apply to the case $k[X,Y]/(f(X),g(X,Y))$, yielding a *practical* algorithm of complexity $O\big(\sqrt{D}(\mathsf{M}(D)+D^{\omega/2})\big)$ for the power projection in this more general algebra. Finally, we refer to [36] for a description of the general multivariate power projection problem, and its applications to the context of polynomial system solving.

### 7.4.1 Computations in the quotient algebra

Let $Q$ be the quotient algebra $k[X,Y]/(f(X),g(Y))$. In the rest of this section, we repeatedly use the *trace*, which is a linear form defined on $Q$: the trace of $A \in Q$ is defined as the trace of the endomorphism of multiplication by $A$ in $Q$.

Our algorithm for the diamond product is based on the following fundamental fact, which is sometimes referred to as Stickelberger's Theorem, see [70, Proposition 2.7]: for any $A$ in $Q$, the characteristic polynomial of $A$ equals $\prod_{\alpha,\beta}\big(T - A(\alpha,\beta)\big)$, where the product runs over all the roots of $f$ and $g$ counted with multiplicities. As a corollary, we have the following:

**Lemma 11** *The polynomial $f \diamond_H g$ is the characteristic polynomial of $H$ in $Q$. The $s$th power sum of the roots of $f \diamond_H g$ is the trace of $H^s$ in $Q$.*

Indeed, the second part of Lemma 11 in conjunction with fast conversion algorithms of Section 7.2 shows that proving the final part of Theorem 8 amounts to giving a fast computation scheme for the first traces of $H^s$ in $Q$. This is the object of the following proposition.

**Proposition 7** *Given $N \geq 1$, the sequence*

$$\mathrm{trace}(1), \mathrm{trace}(H), \mathrm{trace}(H^2), \ldots, \mathrm{trace}(H^{N-1})$$

*can be computed within $O\Big(\sqrt{N}\,\mathsf{M}(D) + DN^{(\omega-1)/2}\Big)$ base field operations.*

We immediately deduce the proof of the complexity estimates in Theorem 8: from Proposition 6, the number of traces to be computed is at most $2D$ and by Proposition 7, this has complexity $O\big(\sqrt{D}(\mathsf{M}(D) + D^{\omega/2})\big)$. Then Proposition 6 and the obvious inequality $p\,\mathsf{M}(\frac{D}{p})\log(\frac{D}{p}) \leq \mathsf{M}(D)\log(D)$ show that the cost of recovering $f \diamond_H g$ from the power sums of its roots is negligible.

Thus, we now concentrate on proving Proposition 7.

### 7.4.2 Power projection

Computing the traces of the first $N$ powers of $H$ is a particular instance of the *power projection* problem: given a linear form $\ell$ on the $k$-algebra $Q$, "compute" the linear map

$$\begin{array}{rcl} \widehat{Q} & \to & k[[T]]_{<N} \\ \ell & \mapsto & \displaystyle\sum_{i=0}^{N-1} \ell(H^i)T^i + O(T^N), \end{array}$$

where $\widehat{Q}$ is the dual $k$-vector space of $Q$, while $k[[T]]_{<N}$ denotes the $k$-vector space of power series truncated at $T^N$.

It is useful to notice that $k[[T]]_{<N}$ naturally identifies, as a $k$-vector space, with the dual of the space $k[T]_{<N}$ formed by polynomials of degree at most $N-1$. Under this identification, the linear map defining the power projection is the transposed map of the *modular composition* (*polynomial evaluation*) by $H$

$$\begin{array}{rcl} k[T]_{<N} & \to & Q \\ p & \mapsto & p(H). \end{array}$$

Now, an algorithmic theorem called *transposition principle*, or *Tellegen's principle*, states, roughly speaking, that for any algorithm computing a linear map there exists an algorithm that computes the transposed map using almost the same number of arithmetic operations (see the next proposition for a precise statement). In our situation, this principle establishes a computational equivalence between the dual problems of modular composition and of power projection. This fact was pointed out for the first time by Shoup in [224], see also Kaltofen [124].

We recall Tellegen's principle in terms of *linear straight-line programs*; the latter can be thought as "ordinary" straight-line programs, but using only linear operations, see [47, Chapter 13] for precise definitions. The complexity of a linear straight-line program is measured by its size, that is, the number of operations it uses.

**Proposition 8** [47, Th. 13.20] *Let* $\mathbf{M}$ *be a* $m \times n$ *matrix with* $z_r$ *zero rows and* $z_c$ *zero columns. For every linear straight-line program of size* $L$ *that computes the matrix-vector product* $\mathbf{Mv}$ *there exists a linear straight-line program of size* $L-n+m-z_r+z_c$ *that computes the transposed matrix-vector product* $\mathbf{M}^{\mathrm{tr}}\mathbf{v}$.

In [190], Paterson & Stockmeyer proposed a "baby-step / giant-step" algorithm for the modular composition by $H$, requiring the computation of *only* $\sqrt{N}$ powers of $H$. The key

idea is to see $p(H)$ as a polynomial in $H^{\sqrt{N}}$ of degree $\sqrt{N}$. Computing its coefficients amounts to $\sqrt{N}$ modular compositions of polynomials of degree at most $\sqrt{N}$ by *the same element* $H$. Brent & Kung [43, Algorithm 2.1] (see also [255, Section 12.2]) remarked that these *simultaneous* modular compositions can be done using $D/\sqrt{N}$ products of pairs of $\sqrt{N} \times \sqrt{N}$ matrices. Adding the $O(\sqrt{N})$ multiplications in $Q$, the total cost of this algorithm is

$$O\big(\sqrt{N}\,\mathsf{M}(D) + DN^{(\omega-1)/2}\big).$$

Tellegen's principle implies that the power projection problem can also be solved within $O\big(\sqrt{N}\,\mathsf{M}(D) + DN^{(\omega-1)/2}\big)$ operations in $k$. This has been noted by Shoup, see the historical notes at the beginning of this section.

We now propose a practical algorithm within this complexity. As we show thereafter, it is obtained by making explicit and then transposing the maps involved in the algorithm for modular composition described above. In order to simplify the notation, we let $r = \left\lfloor \sqrt{N} \right\rfloor$ and $G = H^r$. For a polynomial $p = p_0 + \cdots + p_{N-1}T^{N-1}$, we let $\widetilde{p}_i = p_{(i-1)r} + \cdots + p_{ir-1}T^{r-1}$. The modular composition map $p \mapsto p(H)$ decomposes as follows.

$$\begin{array}{ccccccc}
k[T]_{<N} & \to & k[T]_{<r} \times \cdots \times k[T]_{<r} & \to & Q \times \cdots \times Q & \to & Q \\
p(T) & \mapsto & \big(\widetilde{p}_1(T), \ldots, \widetilde{p}_r(T)\big) & & & & \\
 & & (q_1, \ldots, q_r) & \mapsto & \big(q_1(H), \ldots, q_r(H)\big) & & \\
 & & & & (A_1, \ldots, A_r) & \mapsto & \sum_{i=1}^{r} A_i G^{i-1}
\end{array}$$

After precomputing the elements $1, H, \ldots, H^r$, Brent & Kung's algorithm computes the three linear maps above, as follows:

- The first one is a splitting map, so no arithmetic operation is required.
- The second map is $M \mapsto \mathcal{H}M$, where $\mathcal{H}$ is the $r \times D$ matrix whose columns contain the coordinates of the first $r-1$ powers of $H$.
- The third map is computed using a Horner-like method.

We now proceed to inspect the transposed maps. To write down the transpose of the last one, we first note that, by definition, the transpose of the usual product map $B \mapsto AB$ on $Q$ is the map $\widehat{Q} \to \widehat{Q}$ that associates to $\ell \in \widehat{Q}$ the linear form

$$\begin{array}{rcl}
A \circ \ell : Q & \to & k \\
B & \mapsto & \ell(AB).
\end{array}$$

This is why, from now on, we will use the classical denomination *transposed product* for the operation $A \circ \ell$. An important property of this operation is that it endows the dual $\widehat{Q}$ with a natural $Q$-module structure. In particular, for any $A, C$ in $Q$ and any $\ell$ in $\widehat{Q}$, the following equation holds:

$$(AC) \circ \ell = A \circ (C \circ \ell). \tag{7.5}$$

With these notation, reversing the arrows in the previous diagram gives the following decomposition of the power projection map; recall that $1, H, \ldots, H^{r-1}$ and $G = H^r$ are precomputed.

$$\begin{array}{ccccccc} k[[T]]_{<N} & \leftarrow & k[[T]]_{<r} \times \cdots \times k[[T]]_{<r} & \leftarrow & \widehat{Q} \times \cdots \times \widehat{Q} & \leftarrow & \widehat{Q} \\ \sum_{i=1}^{r} S_i T^{(i-1)r} & \mapsto & (S_1, \ldots, S_r) & & & & \\ & & \left( \sum_{i=0}^{r-1} \ell_j(H^i) T^i \right)_{1 \leq j \leq r} & \mapsto & (\ell_1, \ldots, \ell_r) & & \\ & & & & (\ell, \ldots, G^{r-1} \circ \ell) & \mapsto & \ell \end{array}$$

Let us make a few comments concerning the way to compute each of these maps. For the first one, we transpose Horner's rule: by Equation (7.5), this amounts to computing $G \circ \ell$, then $G \circ (G \circ \ell)$ and so on. The second map is simply given by $M \mapsto \mathcal{H}^{\mathrm{tr}} M$ and the last map can be computed for free.

Summarizing these considerations, our algorithm for the power projection works as described in Figure 7.4.2 below; therein we denoted by $M[i, j]$ the $(i, j)$th entry of a matrix $M$ and by $1_Q$ the unity of the algebra $Q$.

**Power projection**

**Input:** $H$ in $Q$, $\ell$ in $\widehat{Q}$, $N \geq 1$ in $\mathbb{N}$.
**Output:** the series $\sum_{i=0}^{N-1} \ell(H^i) T^i + O(T^N)$.

$r \leftarrow \lfloor \sqrt{N} \rfloor, s \leftarrow \lceil N/r \rceil$
$H_0 \leftarrow 1_Q$
`for` $p$ `from` 1 `to` $r$ `do`
$H_p \leftarrow H \cdot H_{p-1}$
$M_1 \leftarrow$ the $D \times r$ matrix whose $p$th column is $H_{p-1}$
$\ell_0 \leftarrow \ell$
`for` $q$ `from` 1 `to` $s - 1$ `do`
$\ell_q \leftarrow H_r \circ \ell_{q-1}$
$M_2 \leftarrow$ the $s \times D$ matrix whose $q$th row is $\ell_{q-1}$
$M \leftarrow M_2 M_1$
`return` $\sum_{\substack{1 \leq i \leq s \\ 1 \leq j \leq r}} M[i, j] T^{(i-1)r+j-1} + O(T^N)$

Figure 7.8: Bivariate power projection

Apart from the computation of a product of two rectangular matrices $M_1$ and $M_2$ of sizes $\sqrt{N} \times D$ and $D \times \sqrt{N}$, this algorithm requires $\sqrt{N}$ multiplications in $Q$ and $\sqrt{N}$ transposed

products. Decomposing the matrices $M_1$ and $M_2$ into $D/\sqrt{N}$ square matrices of size $\sqrt{N}$ allows to compute their product $M_2M_1$ within $O\big(DN^{(\omega-1)/2}\big)$ operations in $k$. In the next sections, we give explicit algorithms for the product and the transposed product in $Q$, which have complexity $O(\mathsf{M}(D))$. This concludes the proof of Proposition 7.

We stress the fact that, in contrast with Shoup's algorithm for the power projection in [227], our algorithm uses fast matrix arithmetic. Interestingly, in [126, Algorithm AP], a related question, the *automorphism evaluation problem*, is solved in a similar fashion as the above algorithm for the power projection problem.

### 7.4.3 Representing the linear forms

The quotient algebra $Q$ has a canonical monomial basis: since $f$ has degree $m$ and $g$ has degree $n$, then

$$\mathcal{M} = \{x^i y^j,\ 0 \le i \le m-1, 0 \le j \le n-1\}$$

forms a monomial basis of $Q$, where $x$ and $y$ are the images of $X$ and $Y$ in $Q$.

The linear forms will be given by their coefficients in the dual basis of $\mathcal{M}$, that is, by the list of their values on the elements in $\mathcal{M}$. Then the cost of a single evaluation is $mn = D$ operations in the base field.

As a preamble to our algorithm, it is also necessary to compute the trace of all elements in the basis $\mathcal{M}$.

Let us thus consider $i$ in $0, \ldots, m-1$ and $j$ in $0, \ldots, n-1$. By Stickelberger's theorem, the trace of $x^i y^j$ is $\sum_{\alpha,\beta} \alpha^i \beta^j$; then Lemma 7 shows that this trace is the product of the coefficients of $T^i$ in Newton$(f)$ and $T^j$ in Newton$(g)$.

The series Newton$(f)$ and Newton$(g)$ can be computed at precision respectively $m$ and $n$ in $O(\mathsf{M}(\max(m,n)))$ base field operations. Then by the above reasoning, the value of the trace form on the canonical basis $\mathcal{M}$ can be computed for $mn = D$ additional multiplications.

### 7.4.4 Complexity of the product in $Q$

Due to the very specific form of the ideal defining $Q$, one can design the following algorithm for the product in $Q$. It takes as input two elements $A, B$ in $Q$. To obtain $AB$ in $Q$, we first compute their product as plain polynomials in $k[X,Y]$, then reduce this product modulo $(f(X), g(Y))$.

We use again Kronecker's substitution $X \leftarrow T, Y \leftarrow T^{2m-1}$ (see [137] and [255, Section 8.4]) to reduce the computation of the product of $A$ and $B$ as polynomials in $k[X,Y]$ to a univariate multiplication of polynomials of degree at most $2mn - m - n < 2D$. This can be done with complexity $\mathsf{M}(2D)$, which is clearly bounded by $4\mathsf{M}(D)$. The resulting product $P = AB$ is a bivariate polynomial of degree at most $2m-2$ in $X$ and at most $2n-2$ in $Y$. We next proceed to reduce it modulo the ideal $(f(X), g(Y))$; this can be done in two steps.

We first consider $P$ as a polynomial in $k[X][Y]$ and we reduce all its coefficients $P_j(X)$ modulo $f$, using the variant of Sieveking-Kung's algorithm [228, 138] described in [255, Algorithm

9.5]: given $P_j$, compute $S_j = \lceil u \cdot \mathrm{rev}(2m-2, P_j) \rceil^{m-1}$, where $u$ denotes the power series $1/\mathrm{rev}(m, f)$ at precision $m$; then $P_j \mod f$ is given by $P_j - \mathrm{rev}(m-2, S_j) \cdot f$. Besides the precomputation of $u$, whose cost is $3\mathsf{M}(m) + O(m)$ base field operations, see [255, Theorem 9.4], this algorithm uses at most $2n(2\mathsf{M}(m) + O(m)) = 4n\mathsf{M}(m) + O(D)$ operations in $k$. We obtain a bivariate polynomial of degree at most $m-1$ in $X$ and at most $2n-2$ in $Y$, which we now view in $k[Y][X]$. The final step consists in reducing its coefficients modulo $g(Y)$. This can be done using again Sieveking-Kung's algorithm, within $2m\mathsf{M}(n) + O(D)$ operations in $k$, plus the precomputation of $1/\mathrm{rev}(n, g)$, of cost $3\mathsf{M}(n) + O(n)$. As, by assumption, both $m\mathsf{M}(n)$ and $n\mathsf{M}(m)$ are at most $\mathsf{M}(mn) = \mathsf{M}(D)$, our algorithm for the product in $Q$ uses $O(\mathsf{M}(D))$ operations in $k$.

We give the detailed corresponding pseudo-code in Figure 7.9 below. The function `Coeff` extracts coefficients of univariate polynomials.

**Product in $Q$**

**Input:** $A$ and $B$ in $Q = k[X,Y]/(f(X), g(Y))$,
$m = \deg(f)$, $n = \deg(g)$.
**Output:** the product $AB$ in $Q$.

$C(T) \leftarrow A(T, T^{2m-1}) \cdot B(T, T^{2m-1})$
$u \leftarrow \lceil 1/\mathrm{rev}(m, f) \rceil^{m-1}$
`for` $j$ `from` $0$ `to` $2n-2$ `do`
    $P_j \leftarrow \sum_{i=0}^{2m-2} \mathtt{Coeff}(C, (2m-1)j + i) X^i$
    $S_j \leftarrow \lceil u \cdot \mathrm{rev}(2m-2, P_j) \rceil^{m-1}$
    $P_j \leftarrow P_j - \mathrm{rev}(m-2, S_j) \cdot f$
$v \leftarrow \lceil 1/\mathrm{rev}(n, g) \rceil^{n-1}$
`for` $i$ `from` $0$ `to` $m-1$ `do`
    $Q_i \leftarrow \sum_{j=0}^{2n-2} \mathtt{Coeff}(P_j, i) Y^j$
    $R_i \leftarrow \lceil v \cdot \mathrm{rev}(2n-2, Q_i) \rceil^{n-1}$
    $Q_i \leftarrow Q_i - \mathrm{rev}(n-2, R_i) \cdot g$
`return` $\displaystyle\sum_{i=0}^{m-1} Q_i(Y) X^i$

Figure 7.9: Bivariate modular multiplication

Note that in [256, Lemma 2.2], a similar algorithm has been given for the more general case of multiplication in a quotient algebra of type $k[X,Y]/(f(X), g(X,Y))$.

### 7.4.5 Complexity of the transposed product

In this section we explicit an algorithm for the transposed product in $Q$, whose complexity is the same as that of the multiplication in $Q$. As noticed by Shoup [224], Proposition 8 already implies that such an algorithm exists, our contribution is to exhibit a simple, ready to implement one. We derive it from the algorithm of Section 7.4.4, by using some *effective* program transformation techniques described in [35]. Roughly, they work as follows. Given a program, we decompose it into "elementary" blocks of instructions, then go through from the bottom to the top and *transpose* each block. In this process, the ascending `for` loops are transformed into descending ones, and the input and the output are swapped. We refer to [35] for more details.

In our case, two main procedures have to be transposed: the bivariate polynomial multiplication using Kronecker's substitution on one hand, and the Sieveking-Kung's algorithm on the other hand. The latter is explicitly transposed in [35]. Since Kronecker's substitution is the identity map in the canonical bases, transposing it is immediate, so it remains to transpose the univariate polynomial product involved in the bivariate multiplication. In [115], the following operation is defined: given two polynomials, $P$ of degree $m$ and $Q$ of degree at most $m+n$, their *middle product* is the polynomial $[\mathrm{rev}(m,P)\cdot Q]_m^{m+n+1}$, denoted $\texttt{MidProd}(n,P,Q)$. When viewed as a linear map $: k[T]_{<m+n+1} \to k[T]_{<n+1}$, the middle product is the transposed map of the classical polynomial multiplication by $P$, so by Proposition 8 it can be computed for the cost of one multiplication of two polynomials of degree $m$ and $n$, up to $O(m)$ operations, see [115, 35] for explicit algorithms. Note that the transposed Sieveking-Kung's algorithm described in [35] also uses middle products.

With these specifications in mind, our algorithm for the transposed product in $Q$ goes as in Figure 7.4.5.

We stress the fact that this program has been constructed by operating some transformation on the instructions of the program treating the dual question in Section 7.4.4. Its correctness is guaranteed by the validity of these transformations techniques. However, we conclude this section by giving the interpretation of what is computed.

The algorithm takes as input a linear form $\ell$ in $\widehat{Q}$. Since $f(x)=0$ in $Q$, for any integer $j\geq 0$, the sequence $\ell(x^iy^j)_{i\geq 0}$ satisfies a linear recurrence with constant coefficients, of characteristic polynomial $f$. The problem of extending such a linear recurrent sequence is dual to the division with remainder by $f$, see [35, Section 5], so in the first `for` loop, the algorithm computes the values $\ell(x^iy^j)$, for $0\leq j\leq n-1$ and $m\leq i\leq 2m-2$. Similarly, after the pass through the second `for` loop, all the values taken by $\ell$ on the monomials in the set $\mathcal{M}^2=\{x^iy^j, 0\leq i\leq 2m-2, 0\leq j\leq 2n-2\}$ are computed; these values are encoded in the polynomial $P(X,Y)$.

By definition of the middle product, one can finally check that the algorithm outputs the part supported by $\mathcal{M}$ in the product $A(\frac{1}{X},\frac{1}{Y})\cdot P(X,Y)$, that is, in the product

$$A\Big(\frac{1}{X},\frac{1}{Y}\Big)\cdot \sum_{x^iy^j\in\mathcal{M}^2} \ell(x^iy^j)X^iY^j.$$

**Transposed product in $Q$**

**Input:** $A$ in $Q = k[X,Y]/(f(X), g(Y))$, $\ell$ in $\widehat{Q}$,
$m = \deg(f)$, $n = \deg(g)$.
**Output:** the transposed product $A \circ \ell$.

$v \leftarrow \lceil 1/\mathrm{rev}(n, g) \rceil^{n-1}$
`for` $i$ `from` $m-1$ `downto` $0$ `do`
  $Q_i \leftarrow \sum_{j=0}^{n-1} \ell(x^i y^j) Y^j$
  $R_i \leftarrow \mathtt{MidProd}(n-2, g, Q_i)$
  $Q_i \leftarrow Q_i - X^n \lceil R_i \cdot v \rceil^{n-1}$
$u \leftarrow \lceil 1/\mathrm{rev}(m, f) \rceil^{m-1}$
`for` $j$ `from` $2n-2$ `downto` $0$ `do`
  $P_j \leftarrow \sum_{i=0}^{m-1} \mathtt{Coeff}(Q_i, j) X^i$
  $S_j \leftarrow \mathtt{MidProd}(m-2, f, P_j)$
  $P_j \leftarrow P_j - X^m \lceil S_j \cdot u \rceil^{m-1}$
$P(X,Y) \leftarrow \sum_{j=0}^{2n-2} P_j(X) Y^j$
$C \leftarrow \mathtt{MidProd}\big(2mn - m - n, A(T, T^{2m-1}), P(T, T^{2m-1})\big)$
`return` $\displaystyle\sum_{\substack{0 \le i \le m-1 \\ 0 \le j \le n-1}} \mathtt{Coeff}(C, (2m-1)j + i) X^i Y^j$

Figure 7.10: Bivariate transposed product

Since [36, Proposition 1] shows that this part gives the coefficients of $A \circ \ell$ in the dual basis of $\mathcal{M}$, this finishes an alternative proof of the correctness of our algorithm.

### 7.4.6 Experimental results

We have implemented our diamond product algorithm on the basis on the NTL C++ package [226]; as for the composed sum and product, we implemented the version for large or zero characteristic, for base fields of the form $\mathbb{Z}/p\mathbb{Z}$, with $p$ prime. Our implementation uses Winograd's variant of Strassen's matrix multiplication algorithm [234]; it also uses the implementation of transposed polynomial multiplication presented in [35].

The data used to test the diamond product are similar to those used in Section 7.3.4: the input polynomials $f(X)$ and $g(Y)$ have equal degrees $m = n$ and their coefficients are chosen randomly; the output has degree $D = m^2$. The polynomial $H$ is a randomly chosen polynomial of degree less than $m$ in both $X$ and $Y$. We let $m$ vary from 1 to 425 by steps

of 8; the base field is defined by a 32 bit prime. As in Section 7.3.4, the tests were made on the computers of the MEDICIS resource, using a 2 GB, 2200+ AMD Athlon processor.

Figure 7.11 gives the time of the diamond product computation. Again, the abrupt time jumps that occur at powers of 2 come from the Fourier transform algorithm. In Figure 7.12, we separate the times used in respectively the polynomial multiplication and transposed multiplication step, and the linear algebra step. The polynomial multiplication step is predominant, but the ratio between this step and the linear algebra one actually reduces as the degree grows.

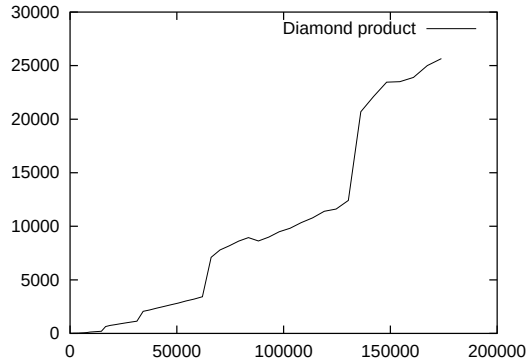



Figure 7.11: Diamond product. (Time in sec. vs output degree)

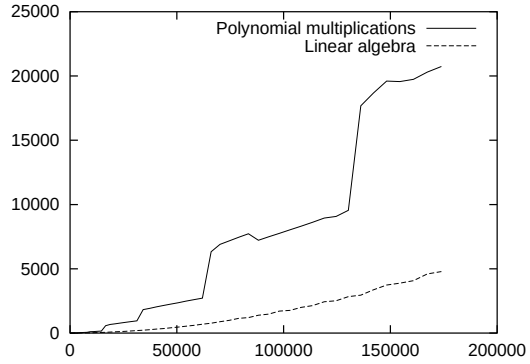



Figure 7.12: Respective times for polynomial multiplications & linear algebra. (Time in sec. vs output degree)

For completeness, we compare in Figure 7.13 the times for one matrix multiplication, using classical and Strassen's multiplication algorithms. Recall that the diamond product algorithm handles matrices of size $\sqrt{D}$, which equals $m$ here, and thus varies from 1 to 425. It appears that for such problem sizes, using a fast matrix multiplication algorithm does have a practical significance on the whole diamond product computation time. Indeed we save a factor of up to 3 on the linear algebra phase, and thus more than 1/4 on the whole computation time.

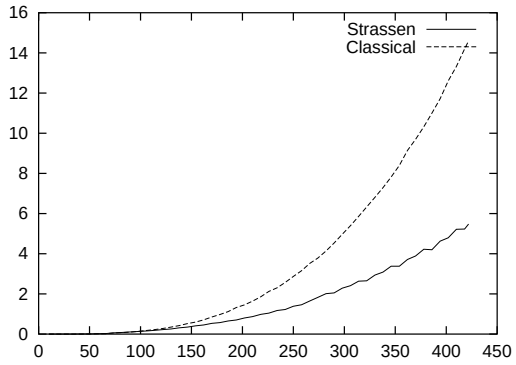



Figure 7.13: Classical and Strassen's matrix multiplications. (Time in sec. vs size)

## 7.5 Applications and Related Questions

To conclude this chapter, we present several situations where composed operations, notably composed sums and products, are used. We also mention two questions somehow similar to composed operations, but for which no optimal algorithms are known to us.

### 7.5.1 Applications

#### Algebraic numbers

Algebraically, one may represent an algebraic number by its minimal polynomial (to distinguish between conjugates, one can use numerical approximates).

If $\alpha$ and $\beta$ are two algebraic numbers represented by their minimal polynomials $f(x)$ and $g(y)$, the sum $\alpha+\beta$ is represented by one of the irreducible factors of the composed sum $f \oplus g$. Similarly, the product $\alpha\beta$ is represented by one of the irreducible factors of the composed product $f \otimes g$. Thus the resultant methods described in [64, 159] can be replaced by our faster solutions, even if factoring the output remains necessary and possibly costly.

We mention that [124] presents an alternative solution for this question: it consists in factoring $f$ in the algebraic extension $\mathbb{Q}[y]/(g(y))$ beforehand, and then computing a power projection modulo a system of the form $g(y) = 0, h(x, y) = 0$. Thus, the factorization in degree $\deg(f)\deg(g)$ over $\mathbb{Q}$ is avoided, at the cost of a factorization in degree $\deg(f)$ in a number field of degree $\deg(g)$.

#### Algebraic functions

The algorithms described in this chapter also adapt to operations over algebraic functions and series as it suffices to operate with a base field of the form $k(z)$. As a matter of fact, formal ideas similar to the ones developed in Section 7.3 prove useful in determining the generating series of walks over the half-line determined by a fixed finite set of allowed jumps; see the "Platypus Algorithm" [sic] and the discussion of [13, p. 56–58]. (There the problem

is to calculate the minimal polynomial satisfied by a product $\alpha_1 \cdots \alpha_k$ of $k$ distinct branches of an algebraic function defined by $P(z, \alpha) = 0$.)

### Dispersion set of polynomials

In many algorithms for symbolic summation (e.g., [2, 109, 3, 193, 191]) one has to compute *the dispersion set* of two polynomials, that is, the set of the (integer) distances between their roots. Classically, this is done by computing the polynomial whose roots are the elements of the dispersion set. The latter polynomial is again a resultant of the particular form discussed in Section 7.3 and can be computed fast using our algorithms. We point out that alternative methods for determining the dispersion set have been recently designed [162, 93]. It would be interesting to carefully compare the (bit) complexities and the practical performances of the resulting algorithms.

### Irreducible polynomials

Constructing irreducible polynomials of prescribed degree over finite fields is a difficult and useful task. It is used, for instance, to implement arithmetic in extension fields. The most efficient algorithm is due to Shoup [222, 224]: it consists in first constructing irreducible polynomials of prime power degree, then combining them to form an irreducible polynomial, using composed products.

In [224], the second step is achieved by a minimal polynomial computation, which has complexity $O\big(D^{(\omega+1)/2}\big)$, where $D$ is the degree of the output. Using our algorithm for the composed product, the cost of this step becomes linear in $D$, up to logarithmic factors.

### Point counting

Designing genus 2 hyperelliptic cryptosystems requires the ability to compute the cardinality of the Jacobian of genus 2 curves defined over finite fields. When the base field is a prime field of the form $\mathbb{Z}/p\mathbb{Z}$, the most commonly used solution is an extension of Schoof's algorithm for elliptic curves [215, 88], which requires to compute torsion subgroups of the Jacobian.

Working out the details, one is led to compute the solutions of systems of the form

$$\frac{f(x_1)}{g(x_1)} = \frac{f(x_2)}{g(x_2)}, \quad \frac{h(x_1)}{g(x_1)} = \frac{h(x_2)}{g(x_2)},$$

where $f, g, h$ are univariate polynomials. Taking into account the symmetry in $x_1, x_2$, we wish to compute an eliminating polynomial for $x_1 + x_2$. This can be done through a suitable resultant computation, but the denominators $g(x_1)$ and $g(x_2)$ create high-degree parasitic factors in this resultant, which should be computed and removed. The parasites are powers of the composed sum of $g$ with itself; in the cryptographic-size record presented in [89], they have degrees several hundreds of thousands. To treat problems of such sizes, the use of our fast algorithms for composed sums becomes necessary.

**Linear recurrent sequences with infinitely many zeros**

A classical result [24] asserts that a linear recurrent sequence has infinitely many zero terms if its minimal polynomial $f$ has a unitary pair, that is, if it has two roots whose ratio is a root of unity.

In [263], Yokoyama, Li and Nemes give algorithms to test this condition, and if so, to find the order of the multiplicative group generated by the corresponding roots of unity. The most time-consuming part of their algorithm is the computation of a polynomial whose roots are the ratios of all pairs of roots of $f$. This directly reduces to the computation of a composed product.

**Shift of polynomials**

In [254], six algorithms for computing shifts of polynomials are proposed and their complexity is analyzed. A seventh algorithm can be deduced as a straightforward application of our algorithm for the composed sums, since $f \oplus (T + a)$ is the shift polynomial of $f$ by $a$. In characteristic zero, the complexity of this algorithm is linear (up to logarithmic factors) in the degree of $f$, in terms of base field operations. Yet, the convolution method of Aho, Steiglitz and Ullman [5] is better by a constant factor. In small characteristic, the comparative analysis has yet to be done.

### 7.5.2 Related questions and open problems

**Resolvents**

Resolvents are an important tool in Galois theory, notably for the *direct problem* of determining the Galois group of an irreducible polynomial $f$ of degree $m$. Their factorization patterns help determine the Galois group of $f$.

For $h \leq m$, an example of such a resolvent is the polynomial $f^{+h}$ of degree $N = \binom{m}{h}$, whose roots are the sums $\alpha_{i_1} + \cdots + \alpha_{i_h}$, with $1 \leq i_1 < \cdots < i_h \leq m$, where $(\alpha_i)_{1 \leq i \leq m}$ are the roots of $f$. This differs from the $h$th iterated composed sum, since repetitions of roots are not allowed here. Yet, the methods we have presented can help answer some simple cases, as illustrated in the following example.

Let $f(T) = T^7 - 7T + 3$ be the Cartier polynomial introduced in [102], and $F = f^{+3}$ the polynomial whose roots are all sums of $h = 3$ distinct roots of $f$. To prove that the Galois group of $f$ is not the symmetric group $S_7$, it is enough to check that $F$ is not irreducible. The polynomial $F$ has degree 35, so knowing its Newton series to order 35 suffices to recover it. To do this, we first decompose $f^{\oplus 3} = f \oplus f \oplus f$ as

$$f^{\oplus 3} = \prod_{\alpha} (T - 3\alpha) \cdot \prod_{\alpha \neq \beta} \big(T - (\alpha + 2\beta)\big)^3 \cdot \prod_{\alpha \neq \beta \neq \gamma \neq \alpha} \big(T - (\alpha + \beta + \gamma)\big)^6.$$

Then, using the definition $F = \prod_{\alpha \neq \beta \neq \gamma \neq \alpha} \big(T - (\alpha + \beta + \gamma)\big)$ and the equalities

$$\prod_{\alpha} (T - 3\alpha) = f \otimes (T - 3) \text{ and } \prod_{\alpha \neq \beta} \big(T - (\alpha + 2\beta)\big) = \frac{f \oplus \big(f \otimes (T-2)\big)}{f \otimes (T-3)}$$

enables to express Newton($F$) as

$$\frac{1}{6}\Big(\text{Newton}\big(f^{\oplus 3}\big) + 2\ \text{Newton}\big(f \otimes (T-3)\big) - 3\ \text{Newton}\ \big(f \oplus \big(f \otimes (T-2)\big)\big)\ \Big).$$

Using Lemmas 7 and 8, the last series can be computed from the series Newton($f$) and $\exp(T)$ to order 35. The polynomial $F$ is then recovered from its Newton series, using the algorithms in Section 7.2. The CPU time used in the whole computation is about 300 times as fast as a direct resultant computation.

A straightforward generalization of this approach for an arbitrary $h$ is not satisfactory, due to the combinatorial explosion of the number of terms involved. A faster method is presented in [53] and has complexity $O_{\log}(h^2N + N^2)$. It is based on the following recurrence relation, expressing $f^{+h}$ in terms of $f^{+j}$, for $j < h$:

$$\big(f^{+h}\big)^h = \prod_{i=1}^{h} \big(\big(f \otimes (T-i)\big) \oplus f^{+(h-i)}\big)^{(-1)^{i+1}}.$$

Using this formula and the fast conversion algorithms presented in Section 7.2 reduces the complexity to $O_{\log}(hN)$. Nevertheless, the degree of the output is $N$, so an optimal algorithm for this question has yet to be found.

### Graeffe polynomials

Let $f$ be a monic polynomial of degree $m$ and $N$ be a positive integer. We call $N$th *Graeffe polynomial* of $f$ the polynomial of degree $m$ whose roots are the $N$th powers of the roots of $f$.

This polynomial can be obtained using $O(\mathsf{M}(mN))$ operations in $k$, by computing the composed product of $f$ and $X^N - 1$. Note that the same complexity result is announced in [118, Section 13.8]. This is nearly optimal with respect to $m$, but not to $N$. On the other hand, the $N$th Graeffe polynomial of $f$ is the characteristic polynomial of $X^N$ modulo $f$. Computing $X^N \mod f$ has complexity $O(\mathsf{M}(m)\log(N))$, which is optimal in $N$, but then the characteristic polynomial computation has complexity more than linear in $m$.

In [252], Graeffe polynomials are computed by means of determinants of so-called *decimation matrices*; these are structured (Toeplitz, quasi-circulant) $N \times N$ matrices with polynomial entries of degree at most $\frac{m+N}{N}$. Using an evaluation-interpolation scheme, the cost of this method is dominated by the evaluation of $m$ determinants of $N \times N$ scalar Toeplitz matrices. Thus, this method is slower than ours.

Writing $P(X) = P_0(X^N) + XP_1(X^N) + \cdots + X^{N-1}P_{N-1}(X^N)$, the entries of the decimation matrix $M$ of $P$ are $M[i,j] = P_{i-j}$ if $i \geq j$ and $M[i,j] = XP_{N+i-j}$ if $i < j$.

$$M = \begin{bmatrix} P_0(X) & XP_{N-1}(X) & \cdots & XP_1(X) \\ P_1(X) & P_0(X) & \ddots & XP_2(X) \\ \vdots & \ddots & \ddots & \vdots \\ P_{N-1}(X) & \cdots & P_1(X) & P_0(X) \end{bmatrix}.$$

Is there a way of reducing the whole cost to $O(\mathsf{M}(m)\log(N))$? If $N$ is a power of 2, this can be achieved using binary powering, but the general case remains open.

# Chapter 8

# Fast Algorithms for Zero-Dimensional Polynomial Systems using Duality

Many questions concerning a zero-dimensional polynomial system can be reduced to linear algebra operations in the quotient algebra $A = k[X_1, \ldots, X_n]/\mathcal{I}$, where $\mathcal{I}$ is the ideal generated by the input system. Assuming that the multiplicative structure of the algebra $A$ is (partly) known, we address the question of speeding up the linear algebra phase for the computation of minimal polynomials and rational parametrizations in $A$.

We present new formulæ for the rational parametrizations, extending those of Rouillier, and algorithms extending ideas introduced by Shoup in the univariate case. Our approach is based on the $A$-module structure of the dual space $\widehat{A}$. An important feature of our algorithms is that we do not require $\widehat{A}$ to be free and of rank 1.

The complexity of our algorithms for computing the minimal polynomial and the rational parametrizations are $O(2^n D^{5/2})$ and $O(n2^n D^{5/2})$ respectively, where $D$ is the dimension of $A$. For fixed $n$, this is better than algorithms based on linear algebra except when the complexity of the available matrix product has exponent less than 5/2.

This chapter is joint work with B. Salvy and É. Schost [36].

## Contents

---

## 8.1 Introduction

Many questions concerning zero-dimensional polynomial systems can be reduced to linear algebra operations in some quotient algebra. Assuming that the multiplicative structure of this algebra is (partly) known, we address the question of speeding up the linear algebra phase for two questions.

Specifically, let $k$ be a field, let $\overline{k}$ be its algebraic closure and let $\mathcal{I}$ be a zero-dimensional ideal of $k[X_1, \ldots, X_n]$. Let $\mathcal{V}(\mathcal{I}) \subset \overline{k}^n$ be the zero-set of the polynomial system defined by $\mathcal{I}$. Given an element $u$ of $A = k[X_1, \ldots, X_n]/\mathcal{I}$, we consider the following problems:

1. compute its *minimal polynomial* $m_u$, that is, the (unique) monic univariate polynomial of minimal degree such that $m_u(u) = 0$ in $A$;
2. if $u$ *separates* the points of $\mathcal{V}(\mathcal{I})$ (see definition below), compute *parametrizations* expressing the coordinates of these points in terms of $u$.

We suppose that $k$ is a *perfect* field. This discards many pathologies such as algebraic field extensions of $k$ without a primitive element. In most applications we have in mind, $k$ is finite or of characteristic zero, so this assumption is satisfied.

The computation of minimal polynomials of elements in such quotient algebras is of particular interest when $A$ is a field or a product of fields. This question appears as a basic subroutine for the computation of triangular sets [145], for the study of the intermediate fields between $k$ and $A$ [146], in Galois theory [9], ... For instance, starting from a description of a quotient algebra by means of a Gröbner basis, Lazard's algorithm `Triangular` [145] produces a "triangular description" of the input ideal through repeated minimal polynomial computations.

In the noncommutative setting of the effective theory of $\mathcal{D}$-modules, an important role is played by the *b-function* of a holonomic system of linear partial differential equations. Algorithm 5.1.5 in [205] reduces the computation of the $b$-function to that of the minimal polynomial of an element in a quotient algebra of the type we consider here.

Another of our initial motivations is the study of algebraic curves and cryptosystems built upon them. Factorization patterns of the minimal polynomials of well-chosen elements help determine the cardinality of the Jacobian of hyperelliptic curves over finite fields, see [55, 86, 216, 90]. In such situations, the element $u$ will typically not be primitive for $k \to A$. The polynomial $m_u$ has degree less than the dimension of $A$, and of course we want to make use of this fact.

Our second interest is the determination of a parametrization of the coordinates of the solutions of $\mathcal{I}$. To this effect, we say that $u \in A$ *separates* the points of $\mathcal{V}(\mathcal{I})$, or is a *separating element* for $\mathcal{I}$ if for all points $P \neq P'$ in $\mathcal{V}(\mathcal{I})$, $u$ takes distinct values on $P$ and $P'$ (see [6, 204]). Since $k$ is a perfect field, this is the case if and only if $u$ is a primitive element of the reduced algebra $A_{\mathrm{red}} = k[X_1, \ldots, X_n]/\sqrt{\mathcal{I}}$, where $\sqrt{\mathcal{I}}$ is the radical of $\mathcal{I}$, see [16]. In this situation, the coordinates of the points in $\mathcal{V}(\mathcal{I})$ can be expressed as *rational* functions of $u$. We call *rational parametrization* of the coordinates of the points in the zero-set $\mathcal{V}(\mathcal{I})$ the data of a separating element $u$, its minimal polynomial $m_u$, and rational functions $f_1, \ldots, f_n$ such that $X_i = f_i(u)$ holds in $A_{\mathrm{red}}$.

Such representations, which go back to Kronecker [137], are well suited to many purposes such as effective computation in the reduced algebra $A_{\mathrm{red}}$ or counting and isolation of real or complex roots. Introduced in Computer algebra by Chistov and Grigoriev [56], then constantly revisited [135, 48, 94, 141, 99, 6], this kind representation bears the name *Geometric Resolution* in [101, 100, 103, 149]. Using the characteristic polynomial of $u$ instead of its minimal polynomial, this representation is called a *Rational Univariate Representation* of the roots of $\mathcal{I}$, using the denomination introduced in [204].

In this chapter, we present some structure theorems related to the two questions mentioned above, then show how algorithmic ideas introduced in the univariate case by Shoup [224, 227] fit into this context. Our algorithms require precomputations, either of some multiplication matrices in $A$, or of the whole multiplication table. These objects may be obtained from the computation of a Gröbner basis [45, 79, 77]. We do not address the difficult question of the complexity of these precomputations.

**Computing a minimal polynomial.**

Let $u$ be an element in $A$ and $\delta$ a bound on the degree of its minimal polynomial $m_u$. A natural algorithm for the computation of $m_u$ consists in expressing the first $\delta$ powers of $u$ on a basis of the $k$ vector space $A$ and then looking for a linear dependency between them. This last step has complexity $O(D^\omega)$, where $\omega$ is the exponent of the complexity of matrix multiplication, and $D$ is the dimension of $A$ over $k$ [47]. Thus $\omega = 3$ for the naive product, and the best result known to this date is $\omega < 2.376$ [69]. However, the fastest widely available implementation we are aware of is based on Strassen's algorithm [234] of exponent $\log_2(7) \simeq 2.808$, in the computer algebra system Magma [32].

A first improvement consists in considering the values taken by a linear form $\ell$ on the powers of $u$. The sequence $(\ell(u^i))_{i\geq 0}$ admits a minimal linear recurrence relation, which coincides, for a random choice of $\ell$, with the minimal polynomial of $u$, and which can be computed efficiently. This suggests the following algorithm: compute the powers of $u$, evaluate $\ell$ on them, and recover the minimal polynomial. This requires the ability to multiply by $u$. The input of this first algorithm will thus be the multiplication matrix of $u$ in $A$.

In the context of polynomial factorization over finite fields, Shoup showed in [224, 227] how to speed up these computations in the univariate case when $A = k[X]/(f)$. His idea is to adapt Paterson and Stockmeyer's fast evaluation algorithm [190] using an $A$-module structure on the dual space $\widehat{A}$. The clever use of this structure avoids the computation of all the powers of the element $u$.

We demonstrate here that this idea extends to multivariate situations, and yields another method for computing a minimal polynomial. The main difficulty lies in obtaining an efficient implementation of the operations in $\widehat{A}$. For the moment, our solution requires a stronger input than above: the whole multiplication table of $A$. This input is also used for instance in the algorithms of [6, 204].

These results are presented in a precise fashion in the following theorem. The algorithms require an *a priori* bound $\delta$ on the degree of the minimal polynomial we want to compute.

A trivial bound is the dimension $D$ of $A$. Problem-specific bounds are often available, as for instance in [55, 86, 90, 216].

**Theorem 9** *Let $D$ be the dimension of $A$ as a $k$-vector space, and let $u$ be in $A$, with minimal polynomial $m_u$. Suppose that $\delta$ is an* a priori *bound on the degree of $m_u$.*

1. *If the matrix of multiplication by $u$ is known, then $m_u$ can be computed by a probabilistic algorithm in $O(\delta D^2)$ operations in $k$.*

2. *If the multiplication table of $A$ is known, then $m_u$ can be computed by a probabilistic algorithm in $O(2^n \delta^{1/2} D^2)$ operations in $k$.*

*In both cases, the algorithm chooses $D$ values in $k$. If these values are chosen in a finite subset $\Gamma$ of $k$, all choices except at most $\delta|\Gamma|^{D-1}$ assure success.*

For $\delta \approx D$, the complexity is $O(D^3)$ in the first case and $O(2^n D^{5/2})$ in the second case. If the number of variables $n$ is fixed, the gain in complexity is of order $\sqrt{D}$, typical of the baby step/giant step techniques which underlie the second approach.

The probabilistic aspect comes from the choice of a linear form over $A$. For unlucky choices, the output of our algorithms is a strict divisor of the actual minimal polynomial. If the degree of the output coincides with the upper bound $\delta$, then this output is necessarily correct. Otherwise, we can either estimate the probability of an unlucky choice, or evaluate the candidate minimal polynomial on $u$.

**Computing parametrizations.**

In the discussion leading to the proof of Theorem 9, we introduce some generating series, depending on both the element $u$ and a linear form over $A$. If $u$ is separating, we show that such series allow to compute rational parametrizations of the points of $\mathcal{V}(\mathcal{I})$. This yields our formulæ in Theorem 11, that extend those of Rouillier [204].

Our formulæ are satisfied if $\mathcal{I}$ is a radical ideal. In the general case, they remain valid under an additional hypothesis, given in Theorem 10 below, and explained in more detail in §8.3.2. In short, the minimal polynomial of $u$ must have the maximal possible degree, and the characteristic of the base field must be zero or large enough.

To use these formulæ in practice, the computational task is quite similar to that required to compute a minimal polynomial: evaluating some linear forms on the powers of $u$. So in a similar fashion, we propose two methods: the direct approach, which requires only multiplication matrices, or its refinement based on Shoup's idea, using the whole multiplication table.

The first approach has the same complexity as the algorithm of [204], at most $O(D^3)$, but our input is weaker. The second approach takes the same input as [204]. Its complexity is at most $O(n2^n D^{5/2})$. This becomes better when the number $n$ of variables is kept constant, whereas the dimension of the quotient algebra becomes large. As above, the gain is then of order $\sqrt{D}$.

**Theorem 10** *Let $D$ be the dimension of $A = k[X_1, \ldots, X_n]/\mathcal{I}$ as a $k$-vector space, and let $u$ be a separating element in $A$, with minimal polynomial $m_u$. Assume that*

- *the characteristic of $k$ is zero or greater than $\min\{s \mid \sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}\}$;*
- *the degree of the minimal polynomial of $u$ is the degree of the minimal polynomial of a generic element in $A$.*

*If $\delta$ is an* a priori *bound on the degree of $m_u$, then the following holds:*

1. *If the matrices of multiplication by $u$ and $x_1, \ldots, x_n$ are known, then a rational parametrization of the zero-set $\mathcal{V}(\mathcal{I})$ can be computed in $O(\delta D^2 + nD^2)$ operations in $k$.*
2. *If the multiplication table of $A$ is known, then a parametrization can be computed in $O(n2^n\delta^{1/2}D^2)$ operations in $k$.*

*The algorithms are probabilistic. In both cases, the algorithm chooses $D$ values in $k$. If these values are chosen in a finite subset $\Gamma$ of $k$, all choices except at most $\delta|\Gamma|^{D-1}$ assure success.*

The probabilistic aspect lies, as in Theorem 9, in the choice of a linear form over $A$. If $\mathcal{I}$ is a radical ideal, it is straightforward to check the correctness of the output, see Section 8.3.1. Otherwise, the last assertion in the theorem makes it possible to estimate the probability of choosing an unlucky linear form.

The algorithms mentioned in Theorems 9 and 10 are easily implemented in a computer algebra system such as Magma [32]. Our experiments show their good practical behavior (see Section 8.5).

**Related results.**

The $A$-module $\widehat{A}$ is called the *canonical module* [207, 139, 76], and has been used in a variety of applications. In particular, the case when the dual $\widehat{A}$ is a free $A$-module of rank 1 has led to new geometric and arithmetic forms of the Nullstellensatz [101, 100], a new proof of the Eisenbud-Levine formula [15], or fast algorithms for isolating roots of complete intersection multivariate systems [173, 174, 175, 176].

One of our main contributions is to propose algorithms using this module structure whenever the operations in $A$ and $\widehat{A}$ are effective, even if the dual is not free and of rank 1.

We have focused on the case when the structure of the algebra $A$ is explicitly given. Our ideas also apply if $\mathcal{I}$ is given by $n$ generators without zeros at infinity. Indeed, in this context, the basis of the results in [174, 175, 176] are fast multiplication algorithms in $A$. It might be possible to extend these results so as to obtain similar complexity estimates for the operations in $\widehat{A}$, which would lead to improved complexity algorithms in this case. More generally, any efficient algorithm for the operations in $A$ and $\widehat{A}$ can be used in conjunction with the ideas presented here.

In a different context, the *geometric resolution* algorithm of [103] solves polynomial systems of dimension zero without multiplicities. Its complexity is quadratic in a geometric quantity attached to the input system, and linear in its *complexity of evaluation*, that is, the number of arithmetic operations necessary to evaluate the system. Recently, this algorithm has been extended so as to handle arbitrary systems, see [149, 150, 151]. An important issue is to extend our algorithmic ideas to this context.

Finally, let us mention that F. Rouillier informed us of an improvement of the second result given in Theorem 10, where a factor of order $n$ is saved.

**Outline of the chapter.**

In Section 8.2, we define the module structure on the dual of $A$, and some useful generating series. In Section 8.3, we show how both a minimal polynomial and some parametrizations can be read out from such series. A direct approach to compute these series yields at once the first assertions in Theorems 9 and 10. In Section 8.4, we show how to improve the crucial step: the evaluation of a linear form on the successive powers of an element in $A$. This will prove the second parts of Theorems 9 and 10. In Section 8.5 we present the experimental behavior of our algorithms. The last section gives the proof of a key proposition in Section 8.3.

**Notation.**

We use the following notation:

- The radical of an ideal $\mathcal{I}$ of $k[X_1, \ldots, X_n]$ is denoted by $\sqrt{\mathcal{I}}$.
- The algebra $A$ is the quotient $k[X_1, \ldots, X_n]/\mathcal{I}$; the images of the variables $X_1, \ldots, X_n$ in $A$ are denoted by $x_1, \ldots, x_n$. We denote by $D$ the dimension of the $k$-vector space $A$, by $\Omega = \{\omega_i\}_{i=1,\ldots,D}$ a monomial basis of $A$ and by $E \subset \mathbb{N}^n$ the set of exponents of the elements in $\Omega$.
- Given $\alpha = (\alpha_1, \ldots, \alpha_n)$ in $\mathbb{N}^n$, we write $X^\alpha$ for the monomial $X_1^{\alpha_1} \cdots X_n^{\alpha_n}$, and $x^\alpha$ for the product $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$.
- The minimal polynomial of any element $t$ in a finite-dimensional algebra is denoted by $m_t$.
- For two subsets $E \subset \mathbb{N}^n$ and $F \subset \mathbb{N}^n$, we let $E + F$ be their Minkowski sum, that is, the set $\{e + f,\ e \in E, f \in F\}$. We use the abbreviation $2E$ for $E + E \subset \mathbb{N}^n$.
- $\widehat{A}$ designates the dual space $\mathrm{Hom}_k(A, k)$ of the $k$-linear forms on $A$. The set $\widehat{\Omega} = \{\widehat{\omega_i}\}_{i=1,\ldots,D}$ represents the dual basis of $\Omega$.
- For a polynomial $P \in k[U]$, we write $\mathrm{rec}(P)$ for its reciprocal $U^{\deg(P)} P(\frac{1}{U})$.

## 8.2 On the Dual of the Quotient Algebra

Most results in this chapter involve linear forms defined over the algebra $A$. We frequently use the following operation, which makes the dual $\widehat{A}$ a $A$-module:

$$\begin{aligned} \circ : A \times \widehat{A} &\to \widehat{A} \\ (u, \ell) &\mapsto u \circ \ell : v \mapsto \ell(vu). \end{aligned}$$

This section is devoted to basic results related to this operation. As mentioned in the introduction, the case when $\widehat{A}$ is a free $A$-module of rank 1 is of particular interest, but this assumption is not required here.

The following lemma (see also [227, 175]) justifies the terminology *transposed product* for the $A$-module operation on $\widehat{A}$.

**Lemma 12** *Let $u$ be in $A$. The matrix of the linear operator*

$$\begin{aligned} \widehat{A} &\to \widehat{A} \\ \ell &\mapsto u \circ \ell \end{aligned}$$

*in the dual basis $\widehat{\Omega}$ is the transposed of the matrix of multiplication by $u$ in the basis $\Omega$.*

*Proof.* Let $\omega$ be in $\Omega$. The value $(u \circ \ell)(\omega)$ is $\ell(\omega u)$. It is given by the product between the row-vector of the coefficients of $\omega u$ on the basis $\Omega$ and the vector representing $\ell$ on the dual basis. This implies that the vector representing $u \circ \ell$ is the product $\mathbf{M}_u^t \ell$, where $\mathbf{M}_u^t$ is the transposed of the matrix $\mathbf{M}_u$ representing the multiplication by $u$ in the basis $\Omega$. □

This result has a strong consequence in terms of complexity, based on the *transposition principle*, or *Tellegen's principle*. This principle is actually a theorem about arithmetic circuits, which originates from linear circuit design and analysis [239, 29, 192, 8] and was introduced in computer algebra in [80, 81, 119, 128]. The proof can be found in [47, Theorem 13.20], see also [124, Problem 6] for more comments.

**Transposition principle.** Let $\mathbf{M}$ be a $n \times n$ matrix, with no zero row nor column, and suppose that the product $\mathbf{v} \mapsto \mathbf{M}\mathbf{v}$ can be computed by an arithmetic circuit of size $\mathcal{C}$. Then there exists an arithmetic circuit of size $\mathcal{C}$ that computes the transposed product $\mathbf{w} \mapsto \mathbf{M}^t\mathbf{w}$.

In most applications, the multiplication matrix $\mathbf{M}_u$ is not known, and its determination might be quite costly. Nevertheless, the transposition principle implies that, whatever the algorithm used for multiplication, there exists an algorithm for transposed multiplication with the same cost, as long as arithmetic circuits are used.

Yet, the algorithms used for (fast) multiplication may not be given by arithmetic circuits. Moreover, even if the proof of the transposition principle is constructive, it is far from obvious how to put it to practice in a computer algebra environment. Therefore, particular attention must be given to design explicit versions of transposed algorithms. In [35], the transposes of some basic algorithms for univariate polynomials are described. In what follows, we will give algorithms for the transposed product in the algebra $A$.

**Generating series.**

We associate to every element $\ell$ of $\widehat{A}$ a multivariate formal power series, denoted $S(\ell)$. For a subset $F \subset \mathbb{N}^n$ we also define a truncated series $S(\ell, F)$. These series are given by:

$$S(\ell) := \sum_{\alpha \in \mathbb{N}^n} \ell(x^\alpha) X^\alpha, \quad S(\ell, F) := \sum_{\alpha \in F} \ell(x^\alpha) X^\alpha.$$

Since $E$ is the set of exponents of the monomial basis $\Omega$, a linear form $\ell$ in $\widehat{A}$ is uniquely determined by $S(\ell, E)$. Given $u$ in $A$ and $\ell$ in $\widehat{A}$, we also introduce the univariate Laurent series

$$R(u, \ell) := \sum_{i \geq 0} \frac{\ell(u^i)}{U^{i+1}}.$$

The series $S(\ell)$ and particularly $R(u, \ell)$ are used repeatedly in this chapter. Similar representations appear in [227, 174, 175], and in [204] for specific linear forms. The following proposition gathers the results we will need when using these generating series. The first point is folklore, similar arguments can be found in [174, 175] and [227]. Let us also mention that results very similar to the second point below can be found in [158], which describes the use of duality-based techniques in coding theory.

**Proposition 9** *Let $\ell$ be in $\widehat{A}$.*

- *Let $u = \sum_{\alpha \in E} u_\alpha x^\alpha$ be in $A$, let $F$ be a subset of $\mathbb{N}^n$ and let $T$ be the Laurent series*

$$T = \sum_{\alpha \in \mathbb{Z}^n} t_\alpha X^\alpha := \left( \sum_{\alpha \in E} \frac{u_\alpha}{X^\alpha} \right) \cdot S(\ell, E + F).$$

  *Then the series $S(u \circ \ell, F)$ is $\sum_{\alpha \in F} t_\alpha X^\alpha$.*

- *For $i$ in $1, \ldots, n$, let $m_i \in k[X_i]$ be the minimal polynomial of $x_i$, and let $\delta_i$ be its degree. Then there exists a polynomial $H_\ell \in k[X_1, \ldots, X_n]$ of partial degree in each variable $X_i$ less than $\delta_i$, such that the following holds:*

$$S(\ell) = \frac{H_\ell}{\mathrm{rec}(m_1) \cdots \mathrm{rec}(m_n)}.$$

- *Let $u$ be in $A$, with minimal polynomial $m_u \in k[U]$ of degree $\delta_u$. Then there exists a polynomial $G_{u,\ell} \in k[U]$ of degree less than $\delta_u$ such that the following holds:*

$$R(u, \ell) = \frac{G_{u,\ell}}{m_u}.$$

  *Moreover, $G_{u,\ell}$ is the quotient of $m_u \sum_{i=0}^{\delta_u - 1} \ell(u^i) U^{\delta_u - i - 1}$ by $U^{\delta_u}$.*

- *There exists a nonzero polynomial $r_u \in k[L_1, \ldots, L_D]$ of total degree at most $\delta_u$, such that $G_{u,\ell}$ is coprime to $m_u$ if and only if $r_u(\ell_1, \ldots, \ell_D) \neq 0$, where $(\ell_1, \ldots, \ell_D)$ are the coordinates of $\ell$ on the dual basis $\widehat{\Omega}$.*

*Proof.* For $\alpha'$ in $F$, the value $(u \circ \ell)(x^{\alpha'})$ is $\ell(ux^{\alpha'}) = \sum_{\alpha \in E} u_\alpha \ell(x^{\alpha+\alpha'})$. The series $T$ can be written

$$T = \left( \sum_{\alpha \in E} u_\alpha X^{-\alpha} \right) \left( \sum_{\beta \in E+F} \ell(x^\beta) X^\beta \right) = \sum_{\alpha' \in E+F-E} \left( \sum_{\alpha \in E} u_\alpha \ell(x^{\alpha+\alpha'}) \right) X^{\alpha'}.$$

The coefficient of $X^{\alpha'}$ in $T$ coincides with $\ell(ux^{\alpha'})$, which proves the first point.

We turn to the second point. Taking $F = \mathbb{N}^n$ shows that for any $u$ in $A$, the series $S(u \circ \ell)$ is the restriction of $u(1/X_1, \ldots, 1/X_n)S(\ell)$ to the set of monomials with exponent in $\mathbb{N}^n$. Let $i$ be in $1, \ldots, n$. Since $m_i(X_i)$ is zero in $A$, the series $S(m_i(x_i) \circ \ell)$ is zero. Consequently, all the monomials in $m_i(1/X_i)S(\ell)$ have degree in $X_i$ between $-\delta_i$ and $-1$. This means that all monomials in $\mathrm{rec}(m_i)(X_i)S(\ell) = X_i^{\delta_i} m_i(1/X_i)S(\ell)$ have degree in $X_i$ between 0 and $\delta_i - 1$. Taking all $i$ into account shows that the series $\mathrm{rec}(m_1)(X_1) \cdots \mathrm{rec}(m_n)(X_n)S(\ell)$ is a polynomial, whose partial degree in each variable $X_i$ is less than $\delta_i$.

Next, we prove the third part. The linear form $\ell$ induces a linear form on the algebra $k[U]/m_u$. The previous point shows that $\mathrm{rec}(m_u) \sum_{i \geq 0} \ell(u^i) U^i$ is a polynomial of degree less than $\delta_u$. Evaluating it at $1/U$ and multiplying the result by $U^{\delta_u - 1}$ shows that $m_u R(u, \ell) = m_u \sum_{i \geq 0} \ell(u^i)/U^{i+1}$ is also a polynomial of degree less than $\delta_u$, denoted by $G_{u,\ell}$. For degree reasons, only a finite number of terms in $R(u, \ell)$ contribute at the product $m_u R(u, \ell)$ defining $G_{u,\ell}$. More exactly, the polynomial $G_{u,\ell}$ equals the polynomial part of the series $m_u \sum_{i=0}^{\delta_u - 1} \ell(u^i)/U^{i+1}$ containing only nonnegative powers of $U$; on the other hand, this polynomial part is obviously the quotient of the division of $m_u \sum_{i=0}^{\delta_u - 1} \ell(u^i) U^{\delta_u - i - 1}$ by $U^{\delta_u}$.

Let us finally prove the last point. For $\omega_i$ in $\Omega$, we let $G_{u,i} \in k[U]$ be $m_u R(u, \widehat{\omega_i})$. If $\ell_1, \ldots, \ell_D$ are the coordinates of $\ell$ on the dual basis, then $G_{u,\ell}$ is $\sum_{1 \leq i \leq D} \ell_i G_{u,i}$. Let now $r_u \in k[L_1, \ldots, L_D]$ be the resultant of $\sum_{1 \leq i \leq D} L_i G_{u,i}$ and $m_u$ with respect to $U$. Then, using [255, Lemma 6.25], we see that $G_{u,\ell}$ and $m_u$ are coprime if and only if $r_u(\ell_1, \ldots, \ell_D) \neq 0$.

For any polynomial $G$ of degree less than $\delta_u$, we now prove that there exists $\ell \in \widehat{A}$ such that $G = G_{u,\ell}$. This suffices to show that $r_u$ is a nonzero polynomial. Since $r_u$ has total degree at most $\delta_u$, this will prove the proposition.

The system $m_u R(u, \ell) = G$ is linear in $(\ell(1), \ldots, \ell(u^{\delta_u - 1}))$, of triangular form with diagonal entries equal to 1 as $m_u$ is monic. Since $(1, \ldots, u^{\delta_u - 1})$ are linearly independent, it is always possible to find $\ell$ which takes prescribed values on these powers of $u$. □

This proposition shows that for a generic choice of $\ell$, the irreducible form of the rational series $R(u, \ell)$ has the minimal polynomial $m_u$ for denominator. This will be used repeatedly in the rest of this chapter.

**An algorithm for the transposed product.**

The first point in the previous proposition suggests the following algorithm for the transposed product: given $\ell$ and $u$, first compute $S(\ell, 2E)$, taking $F = E$; then perform a power series multiplication, and read off the coefficients of $S(u \circ \ell, E)$.

The main difficulty lies in determining the truncated series $S(\ell, 2E)$ from its first terms $S(\ell, E)$. The second point of Proposition 9 shows that the series $S(\ell)$ is rational. When there is only one variable, the quotient $A$ is given as $k[X]/(f)$, so the denominator of $S(\ell)$ is known *a priori*, as it is the reciprocal polynomial of $f$. It is then straightforward to recover the numerator from the first terms $S(\ell, E)$, which in turns gives the next terms of $S(\ell, 2E)$ by Taylor expansion. This is the basis of Shoup's algorithm for the univariate transposed product [227].

In the general case, the denominator is not known in advance. At the moment, we are unable to make an algorithmic use of the rationality of the series $S(\ell)$ with good complexity, or even of the stronger form given in the second part of Proposition 9.

## 8.3 Computing Minimal Polynomials and Rational Parametrizations

We now describe our first algorithms solving the questions mentioned in the introduction: computing the minimal polynomial of an element $u$ in $A$, and the corresponding parametrization, if $u$ is separating. These algorithms are derived from the study of the generating series introduced in the previous section, and yield the first parts of Theorems 9 and 10.

Similar considerations to those presented in Subsection 8.3.1 can be found in the literature, for instance in [258, 224, 227, 123]. The main new result is Theorem 11 in Subsection 8.3.2: it provides a generalization of Rouillier's formulæ [204], which does not require the use of a specific linear form to compute parametrizations. In [204], this specific form, the *trace*, is computed from the multiplication table of $A$. Here, we avoid this precomputation, as we show that almost any form can be used. Consequently, the algorithms presented in Subsection 8.3.3 only require multiplication matrices as input.

All these algorithms are based on the same basic subroutine, the evaluation of a linear form on the successive powers of an element in $A$. Thus their complexity is fundamentally dependent on the cost of this particular task; reducing this cost will be the object of Section 8.4.

### 8.3.1 Computing a minimal polynomial

Our method to compute a minimal polynomial in $A$ is based on the following property: if $\ell$ is an arbitrary linear form on $A$, then the scalar sequence $(\ell(u^i))_{i \geq 0}$ is linearly recurrent, that is, it can be defined by a linear recurrence relation with constant coefficients. The relation of minimal degree is called its *minimal polynomial*; if $\ell$ is a "generic" linear form, then this polynomial equals the the minimal polynomial of $u$.

This principle has been used in a variety of settings. It underlies Wiedemann's algorithm [258] for solving sparse — or rather, easy-to-evaluate — linear systems, and is the basis of Li's and Shoup's algorithms [240, 224, 227] to compute minimal polynomials in the univariate case $A = k[X]/(f)$.

Given an upper bound $\delta$ on its degree, the minimal polynomial of a sequence of scalars $\mathcal{L}$ satisfying a linear recurrence can be computed by Berlekamp-Massey's algorithm, see [20, 165] and [255, Section 12.3]. This algorithm requires the first $2\delta$ values of $\mathcal{L}$, and amounts to the computation of a $(\delta, \delta)$ Padé approximant for the generating series $\sum_{i\geq 0} \mathcal{L}_i U^i$. This is denoted by `MinimalPolynomial`$(\mathcal{L})$ in the algorithm below.

**Computing the minimal polynomial**

**Input:** $u$ in $A$, $\ell$ in $\widehat{A}$, a bound $\delta$ on the degree of $m_u$.
**Output:** a polynomial $m_{u,\ell}$ in $k[U]$.

$\mathcal{L} \leftarrow [\ell(1), \ell(u), \ldots, \ell(u^{2\delta-1})]$;
$m_{u,\ell} \leftarrow$ `MinimalPolynomial`$(\mathcal{L})$;
`return`$(m_{u,\ell})$;

The next proposition encapsulates the cost and correctness analysis of this algorithm. Similar considerations for Wiedemann's algorithm can be found in [123].

**Proposition 10** *Let $u$ be in $A$ and let $m_u$ be its minimal polynomial. If $\delta$ is a bound on the degree of $m_u$, then besides the evaluation of the sequence $[\ell(1), \ell(u), \ldots, \ell(u^{2\delta-1})]$, the previous algorithm requires $O(\delta^2)$ operations in $k$. Its output is the polynomial $m_u$ if and only if the polynomial $G_{u,\ell}$ from Proposition 12 and $m_u$ are coprime. Otherwise, the output $m_{u,\ell}$ is a strict divisor of $m_u$.*

*Proof.* Using a naive version of the extended Euclidean algorithm, the running time of Berlekamp-Massey's algorithm is quadratic in $\delta$. This proves the complexity estimate.

Let $m_{u,\ell}$ be the (monic) minimal polynomial of the sequence $(\ell(u^i))_{i\geq 0}$. The polynomial $m_u$ cancels this sequence, since $\sum_i a_i u^i = 0$ implies that the equality $\sum_i a_i \ell(u^{i+j}) = 0$ holds for all $j$. Consequently, $m_{u,\ell}$ divides $m_u$. Let us show that they coincide if and only if the polynomials $G_{u,\ell}$ and $m_u$ are coprime, where $G_{u,\ell}$ is defined in Proposition 12:

$$R(u, \ell) := \sum_{i\geq 0} \frac{\ell(u^i)}{U^{i+1}} = \frac{G_{u,\ell}}{m_u}. \tag{8.1}$$

To this effect, we recall the following result from [110, Lemma 1]: the generating series $R(u, \ell)$ has the rational form

$$R(u, \ell) = \frac{H_{u,\ell}}{m_{u,\ell}}, \tag{8.2}$$

the polynomials $H_{u,\ell}$ and $m_{u,\ell}$ *being coprime.*

The two rational expressions of $R(u, \ell)$ in equations (8.1) and (8.2) show that if $m_{u,\ell}$ and $m_u$ coincide, then $G_{u,\ell}$ and $H_{u,\ell}$ coincide, so $G_{u,\ell}$ and $m_u$ are coprime. For the converse direction, we first notice that, by equations (8.1) and (8.2), $m_u$ divides $m_{u,\ell}G_{u,\ell}$. Therefore, if $G_{u,\ell}$ and $m_u$ are coprime, then $m_u$ divides $m_{u,\ell}$. Since $m_{u,\ell}$ always divides $m_u$, it follows that $m_{u,\ell}$ and $m_u$ coincide. This finishes the proof. □

Using a fast extended Euclidean algorithm [255, Section 11.1], the complexity of Berlekamp-Massey's algorithm drops to $O(\delta \log^2 \delta \log\log \delta)$. The polynomial $G_{u,\ell}$ can be computed as a byproduct without affecting the complexity. In any case, the limiting factor in this algorithm is the computation of the sequence $[\ell(1), \ell(u), \ldots, \ell(u^{2\delta-1})]$.

If the degree of the output coincides with the known upper bound for $\deg m_u$, the output is necessarily correct. A trivial upper bound is the dimension of $A$: if the degree of the output reaches this upper bound, then $u$ is primitive for $k \to A$, and the result of the algorithm is correct. Otherwise, Proposition 10 states that the output $m_{u,\ell}$ is correct if and only if $m_{u,\ell}(u)$ is zero.

### 8.3.2 Computing parametrizations

If $u$ is a separating element for $\mathcal{I}$, we want to compute parametrizations giving the values of the variables $X_j$ on $\mathcal{V}(\mathcal{I})$ as functions of $u$, that is, rational functions $f_j(u)$ such that the relations $x_j = f_j(u)$ hold in the reduced algebra $A_{\text{red}} = k[X_1, \ldots, X_n]/\sqrt{\mathcal{I}}$. Following the ideas of Kronecker [137] and Macaulay [160], we propose a method to compute *rational* parametrizations of the form

$$x_j = \frac{g_j(u)}{g(u)}.$$

Our method requires the following assumptions:

1. the characteristic of $k$ is zero or larger than $\min\{s, \sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}\}$;
2. the degree of the minimal polynomial $m_u$ of $u$ is the degree of the minimal polynomial of a generic element in $A$.

A *generic element* in $A$ is defined as $\sum_{i=1}^{D} T_i\omega_i$ in $A \otimes_k k(T_1, \ldots, T_D)$. This element depends on the choice of the basis $\Omega$, but the degree of its minimal polynomial over $k(T_1, \ldots, T_D)$ depends only on $A$, as a standard linear algebra fact [140, Section 62] ensures that two similar matrices have the same minimal polynomial. As an illustration, consider the case $A = \mathbb{Q}[X_1, X_2]/(X_1^2, X_2^2)$. The minimal polynomial of a generic element has degree 3, but $x_1$, even though separating, has $U^2$ for minimal polynomial. The possible defects can be measured using the nil-indices of the local factors of $A$, see Section 8.6.

If $\mathcal{I}$ is a radical ideal, assumption 1 is obviously satisfied. Since $k$ is perfect, a separating element is also primitive, so assumption 2 is also satisfied in this case.

Taking the above assumptions for granted, our main result is the following theorem:

**Theorem 11** *Let $u$ in $A$ be a separating element of $\mathcal{I}$, such that the above assumptions are satisfied. Let $v$ be in $A$, $\ell$ in $\widehat{A}$, and let $G_{u,\ell}$ and $G_{u,v\circ\ell}$ be the polynomials in $k[U]$ of degree less than that of $m_u$ such that*

$$R(u,\ell) = \frac{G_{u,\ell}}{m_u}, \quad R(u, v\circ\ell) = \frac{G_{u,v\circ\ell}}{m_u}.$$

*Then if $m_u$ and $G_{u,\ell}$ are coprime, the following equality holds:*

$$v = \frac{G_{u,v\circ\ell}(u)}{G_{u,\ell}(u)} \quad \textit{in} \quad A_{\mathrm{red}}.$$

This proposition requires a few comments:

- If the condition on the degree of $m_u$ is not satisfied, then the conclusion may become false for a generic linear form. Consider again $A = \mathbb{Q}[X_1, X_2]/(X_1^2, X_2^2)$ with basis $(1, x_1, x_2, x_1x_2)$, $u = x_1$, $v = x_2$, and let $\ell_1, \ell_{x_1}, \ell_{x_2}, \ell_{x_1x_2}$ be the coordinates of $\ell$ on the dual basis. A short calculation shows that

  $$m_u = U^2, \quad R(x_1,\ell) = \frac{\ell_1 U + \ell_{x_1}}{U^2}, \quad R(x_1, x_2\circ\ell) = \frac{\ell_{x_2}U + \ell_{x_1x_2}}{U^2};$$

  so our formulæ would wrongly give the value $\ell_{x_1x_2}/\ell_{x_2}$ for $x_2$ instead of 0.

- In [204, Theorem 3.1], a similar result is proved for a particular linear form, the *trace*, which associates to any element $v$ in $A$ the trace of the multiplication map by $v$. For this particular form, the hypothesis on the degree of $m_u$ is not required.

- If $\mathcal{I}$ is a radical ideal, a direct proof of Theorem 11 is the following: since $k$ is a perfect field, the trace form generates $\widehat{A}$ as a $A$-module [15, 206]. The conclusion follows from [204, Theorem 3.1].

We defer the somewhat lengthy proof of Theorem 11 to the last section of the chapter and we directly present our algorithm for computing rational parametrizations. It takes as input a linear form $\ell$ on $A$, an element $u$ in $A$, its minimal polynomial $m_u$ of degree $\delta_u$, as well as the polynomial $G_{u,\ell}$ defined in Proposition 9.

**Computing the parametrizations**

**Input:** $u$ in $A$, $\ell$ in $\widehat{A}$, $m_u$ and $G_{u,\ell}$ in $k[U]$.
**Output:** a rational parametrization of the coordinates.

`for` $j$ in $1, \ldots, n$ `do`
  $c^{(j)} \leftarrow [(x_j\circ\ell)(1), (x_j\circ\ell)(u), \ldots, (x_j\circ\ell)(u^{\delta_u-1})]$;
  $C_j \leftarrow \sum_{i=0}^{\delta_u-1} c_i^{(j)} U^{\delta_u-i-1}$;
  $G_{u,x_j\circ\ell} \leftarrow m_u \cdot C_j \text{ div } U^{\delta_u}$;
`return` $[\frac{G_{u,x_1\circ\ell}}{G_{u,\ell}}, \ldots, \frac{G_{u,x_n\circ\ell}}{G_{u,\ell}}]$;

**Proposition 11** *Under the hypotheses of Theorem 11, the output of the previous algorithm is a rational parametrization of the points in $\mathcal{V}(\mathcal{I})$. Besides the evaluation of the sequences*

$$[(x_j \circ \ell)(1), (x_j \circ \ell)(u), \ldots, (x_j \circ \ell)(u^{\delta_u - 1})], \qquad j \in \{1, \ldots, n\},$$

*this algorithm requires at most $O(nD^2)$ additional operations in $k$.*

*Proof.* We begin by recalling that the polynomial $G_{u,x_j \circ \ell}$ can be obtained as the quotient of $m_u \sum_{i=0}^{\delta_u - 1} (x_j \circ \ell)(u^i) U^{\delta_u - i - 1}$ by $U^{\delta_u}$, where $\delta_u$ is the degree of $m_u$. We proved this fact in the third part of Proposition 9. The correctness of the above algorithm then follows from the formulæ in Theorem 11, applied to $v = x_j$, for $j = 1, \ldots, n$. The cost analysis is straightforward, since each polynomial multiplication has complexity at most quadratic in the degree $\delta_u \leq D$. □

We point out that fast algorithms for polynomial multiplication would yield a linear complexity in $D$, up to logarithmic factors, but the bottleneck of this algorithm is the computation of the sequences $[(x_j \circ \ell)(1), (x_j \circ \ell)(u), \ldots, (x_j \circ \ell)(u^{\delta_u - 1})]$. We stress the fact that the probabilistic aspect of the output relies only on the correct computation of the minimal polynomial of $u$; see the previous subsection for more comments on this point.

### 8.3.3 Complexity estimates for the first approach

To put the algorithms of the previous subsections to practice, we must specify the operations in $A$. In this subsection, we assume that the *matrices of multiplication* by $u$ and $x_1, \ldots, x_n$ are known and prove the first parts of Theorems 9 and 10.

The algorithm for a minimal polynomial is given in Subsection 8.3.1. The main task lies in computing the values

$$[\ell(1), \ell(u), \ldots, \ell(u^{2\delta - 1})],$$

$\delta$ being an *a priori* bound on the degree of $m_u$ and $\ell$ a linear form on $A$. To compute the parametrizations corresponding to a separating element $u$, we first compute its minimal polynomial as above, then evaluate

$$[(x_j \circ \ell)(1), (x_j \circ \ell)(u), \ldots, (x_j \circ \ell)(u^{\delta_u - 1})], \quad j = 1, \ldots, n,$$

where $\delta_u \leq \delta$ is the degree of the minimal polynomial of $u$.

The other necessary operations and their complexity are given in Propositions 10 and 11, so we just need to detail the cost of the successive evaluations of respectively $\ell$ and $x_1 \circ \ell, \ldots, x_n \circ \ell$. For the moment, we follow a direct approach. All powers of $u$ are computed, then the linear forms are evaluated on all of them. A more refined method is introduced in the next section.

- Using its multiplication matrix, one multiplication by $u$ has cost $O(D^2)$ operations in $k$. Consequently, all the requested powers of $u$ can be computed within $O(\delta D^2)$ operations in $k$.

- Given the linear form $\ell$, each linear form $x_j \circ \ell$ can be computed using Lemma 12 since the matrix of multiplication by $x_j$ is known. The total cost is thus within $O(nD^2)$ operations in $k$.

- The evaluation of a single linear form takes $O(D)$ operations in $k$. Evaluating all the linear forms on the powers of $u$ requires respectively $O(\delta D)$ or $O(n\delta_u D)$ operations in $k$.

This gives respectively $O(\delta D^2)$ operations in $k$ for the minimal polynomial, and $O(\delta D^2+nD^2)$ for the parametrizations. The additional costs are given in Propositions 10 and 11. They fit into the complexity bounds $O(\delta D^2)$ and $O(\delta D^2 + nD^2)$. This concludes the complexity analysis.

Propositions 10 and 11 show that the output is correct whenever the polynomials $G_{u,\ell}$ and $m_u$ are coprime. The last point in Proposition 9 shows that this is the case if and only if the coefficients of $\ell$ on the dual basis cancel a nonzero polynomial $r_u$ of degree at most $\delta_u$. Zippel-Schwartz's lemma (see [266, 218] and [255, Lemma 6.44]) concludes the probability analysis.

## 8.4 Speeding up the Power Projection

The algorithms presented in the previous section share the same basic subroutine: the evaluation of a linear form on the successive powers of an element in $A$. Their complexity fundamentally relies on the cost of this particular operation, called *power projection*.

**Power Projection Problem.** Let $u$ be in $A$, $\ell$ in $\widehat{A}$ and $N > 0$. Compute the sequence $[\ell(1), \ell(u), \ldots, \ell(u^{N-1})]$.

The naive solution to this question used in the previous section requires to evaluate all the powers of $u$. In this section, we present a result given by Shoup in the univariate case [224, 227], which shows how to avoid the computations of all those powers, by a "transposition" of Paterson and Stockmeyer's fast evaluation algorithm [190]. This brings a speed-up of order $\sqrt{N}$ over the naive version.
This approach requires other operations than mere multiplications by $u$ or $x_i$. Thus, we first state the complexity results in terms of the cost of product and transposed product in $A$, denoted respectively by $\mathcal{M}(A)$ and $\mathcal{M}^t(A)$. Next, we put these ideas to practice. For the time being, our effective version of the transposed product requires the whole multiplication table of the algebra $A$.

### 8.4.1 Baby step / giant step techniques

It is noted in [224, 227, 124] that the power projection problem itself is a transposition of the question of polynomial evaluation in $A$:

**Polynomial Evaluation Problem.** Let $p$ be a polynomial in $k[T]$ of degree $N-1$, and $u$ in $A$. Compute $p(u)$.

For both questions, the point is to avoid the computation of *all* powers $u^i$, which would lead to a complexity of $O(N\mathcal{M}(A))$ operations in $k$. In [190], Paterson and Stockmeyer propose an algorithm for the polynomial evaluation problem (see also [43]) which saves a factor $\sqrt{N}$ using a baby step / giant step technique.

The idea underlying this process also applies to the power projection problem and yields the following algorithm, initially presented in [227] for the case $A = k[X]/(f)$. As in Paterson and Stockmeyer's, this algorithm takes as input two parameters $k$ and $k'$, which must satisfy $kk' \geq N$.

**Power projection**

**Input:** $u$ in $A$, $\ell$ in $\widehat{A}$, $N$, $k$, $k'$.
**Output:** the sequence $[\ell(1), \ell(u), \ldots, \ell(u^{N-1})]$.

$u_i \leftarrow u^i, \;\; i = 0, \ldots, k$
`for` $i \leftarrow 0, \ldots, k'-1$ `do`
    $c_{ik+j} \leftarrow \ell(u_j), \;\; j = 0, \ldots, k-1$
    $\ell \leftarrow u_k \circ \ell$
`return` $[c_0, \ldots, c_{N-1}]$;

We encapsulate the complexity of this algorithm in the following proposition. A similar result is presented in [227].

**Proposition 12** *Let $u$ be in $A$, let $\ell$ be in $\widehat{A}$ and let $N > 0$. Then, the sequence*

$$[\ell(1), \ell(u), \ldots, \ell(u^{N-1})]$$

*can be computed within $O\left(N^{1/2}(\mathcal{M}(A) + \mathcal{M}^t(A)) + ND\right)$ operations in $k$.*

*Proof.* We take $k$ and $k'$ of the same magnitude, that is

$$k = \lfloor \sqrt{N} \rfloor, \quad k' = \lceil N/k \rceil,$$

where $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively denote the largest integer less than or equal to $x$, and the smallest integer larger than or equal to $x$.

The precomputation of the first $k$ powers of $u$ requires $O(N^{1/2})$ multiplications in $A$. Each of the $k'$ passes through the `for` loop requires the evaluation of $k$ linear forms, plus a transposed multiplication. Since $kk' = O(N)$, the overall cost is thus $O(ND)$ operations for the evaluation of the linear forms and $O(N^{1/2})$ transposed multiplications. This proves the proposition. □

**Corollary 5** *Let $D$ be the dimension of $A$ as a $k$-vector space, and let $u$ be in $A$. Let $\delta$ be a bound on the degree of the minimal polynomial of $u$. Then:*

- *The minimal polynomial of $u$ can be computed by a probabilistic algorithm in $O\left(\delta^{1/2}(\mathcal{M}(A) + \mathcal{M}^t(A)) + \delta D\right)$ operations in $k$.*

- *If $u$ is a separating element of $\mathcal{V}(\mathcal{I})$ such that the assumptions of Subsection 8.3.2 are satisfied, a parametrization of the algebraic variables can be computed in*

$$O\left(n\delta^{1/2}(\mathcal{M}(A)+\mathcal{M}^t(A))+nD^2\right)$$

*operations in $k$.*

*In both cases, the algorithm chooses $D$ values in $k$. If these values are chosen in a finite subset $\Gamma$ of $k$, all choices except at most $\delta|\Gamma|^{D-1}$ assure success.*

*Proof.* The proof is similar to that of Subsection 8.3.3, the difference lies in the complexity analysis of the power projection. Proposition 12 brings the result, taking respectively $N = 2\delta$ for the minimal polynomial computation, and $N = \delta_u \leq \delta$ for the parametrizations. □
Using the transposition principle, these complexity results could be rewritten in terms of $\mathcal{M}(A)$ only, but our explicit version reflects the underlying algorithm more closely.

### 8.4.2 Complexity estimates for the second approach

To put such algorithms to practice, we need an effective version of the transposed product. To this effect we suppose that the structure of the algebra $A$ is given by a monomial basis *and* the corresponding multiplication tensor. This makes it possible to estimate the cost of the product and transposed product, which will conclude the proofs of Theorems 9 and 10.

More precisely, in the following paragraphs, we show that the costs of multiplication and transposed multiplication, denoted by $\mathcal{M}(A)$ and $\mathcal{M}^t(A)$ up to now, are in $O(2^nD^2)$ operations in $k$. With these results, the complexity estimates of Corollary 5 become respectively $O(2^n\delta^{1/2}D^2)$ and $O(n2^n\delta^{1/2}D^2)$ operations in $k$, which concludes the proof of Theorems 9 and 10.

**A note on Rouillier's algorithm.**

The input is now the same as that of [204]. Yet, Rouillier's algorithm uses a particular linear form, the trace. In the present context, computing the trace is straightforward, since we have precomputed the whole multiplication table. Thus, we can apply our baby step/giant step techniques to speed up the deterministic algorithm of [204]. Still, using random linear forms has its benefits; for instance, we may choose forms with many coefficients equal to zero.

To prove the estimates on the complexity of the operations in $A$ and $\widehat{A}$, we recall and introduce some notation.

- We recall that $\Omega = \{\omega_i\}_{i=1,\dots,D}$ is a monomial basis of $A$, and that $E \subset \mathbb{N}^n$ is the corresponding set of exponents, so that $\Omega = x^E$.

- We denote by $\Omega\cdot\Omega$ the set of products $\{\omega_i\omega_j \mid \omega_i \in \Omega, \omega_j \in \Omega\}$. The corresponding set of exponents is denoted by $2E$, and is the Minkowski sum $E+E \subset \mathbb{N}^n$. Its cardinality is bounded by $2^n|E| = 2^nD$.

- We assume that the sets $\Omega$ and $\Omega \cdot \Omega$ are ordered; the elements of $A$ will be given by their coefficients on the basis $\Omega$. The multiplication tensor in $A$ is given by a $|E| \times |2E|$ matrix $\mathbf{M}$, with rows indexed by the elements in $\Omega$ and columns indexed by the elements of $\Omega \cdot \Omega$. The columns of $\mathbf{M}$ give the coordinates of the element in $\Omega \cdot \Omega$ on the basis $\Omega$.

Introducing the matrix $\mathbf{M}$ is a convenient way to describe the operations in $A$ and $\widehat{A}$ and bound their complexity.

**Multiplication in the quotient.**

We first give the cost of the multiplication in $A$. This operation is done in a straightforward manner. Two elements $u$ and $v$ in $A$ are multiplied as polynomials in $k[X_1, \ldots, X_n]$, then reduced using the matrix $\mathbf{M}$.

In the algorithm below, $u$ and $v$ are given by the vectors $\mathbf{u}$ and $\mathbf{v}$ of their coefficients on the basis $\Omega$. Given a vector $\mathbf{u}$ of size $D$ and a monomial $\omega$ in $\Omega$, $\mathbf{u}[\omega]$ denotes the entry of $\mathbf{u}$ corresponding to $\omega$. The function $\texttt{Coefficients}(W, \Omega \cdot \Omega)$ returns the vector of the coefficients of $W$ on the monomial family $\Omega \cdot \Omega$.

**Multiplication in the quotient**

**Input:** the coefficients of $u, v$ in $A$, the matrix $\mathbf{M}$.
**Output:** the coefficients of the product $uv$ in $A$.

$U \leftarrow \sum_{\omega \in \Omega} \mathbf{u}[\omega]\omega$;
$V \leftarrow \sum_{\omega \in \Omega} \mathbf{v}[\omega]\omega$;
$R \leftarrow UV$; # the multiplication is done in $k[X_1, \ldots, X_n]$
$\mathbf{c}_W \leftarrow \texttt{Coefficients}(W, \Omega \cdot \Omega)$;
`return` $\mathbf{M}\mathbf{c}_W$;

Given $u$ and $v$ in $A$, the previous algorithm computes the product $uv$ in $A$ within $O(2^n D^2)$ operations in $k$. Indeed, the naive multiplication of two polynomials with support in $E$ requires $O(D^2)$ operations. The reduction of the product is done by the matrix-vector product, which requires $|E||2E| \leq 2^n |E|^2 = 2^n D^2$ operations in $k$.

**Transposed multiplication.**

Our effective version of the transposed product was described at the end of Section 8.2. There, we reduced the transposed multiplication $u \circ \ell$ to two steps. First computing $S(\ell, 2E)$, that is, the values of $\ell$ on the elements of $\Omega \cdot \Omega$, then performing a multivariate series multiplication and extracting the required coefficients.

For any $\eta$ in $\Omega \cdot \Omega$, the value $\ell(\eta)$ is the product between the row $\mathbf{c}_\ell$ of the coefficients of $\ell$ on the dual basis and the column of the coefficients of $\eta$ on the basis $\Omega$. In other words, the coefficients of $S(\ell, 2E)$ are the entries of the product $\mathbf{c}_\ell \mathbf{M}$.

This property yields the following algorithm for the transposed product. The linear form $\ell$ is given as the row-vector $\mathbf{c}_\ell$ of its coefficients on the dual basis. The other notation was introduced above.

**Transposed multiplication in the quotient**

**Input:** $u$ in $A$, $\ell$ in $\widehat{A}$, the matrix $\mathbf{M}$.
**Output:** $u \circ \ell$ in $\widehat{A}$.

$\mathbf{d}_\ell \leftarrow \mathbf{c}_\ell \mathbf{M}$;
$S \leftarrow \sum_{\eta \in \Omega \cdot \Omega} \mathbf{d}_\ell[\eta] X^\eta$;
$T \leftarrow u(1/X_1, \ldots, 1/X_n) \cdot S$;
`return Coefficients`$(T, \Omega)$;

Given $u$ in $A$ and $\ell$ in $\widehat{A}$, the previous algorithm computes the transposed product $u \circ \ell$ within $O(2^n D^2)$ operations in $k$. Indeed, the matrix-vector product requires $|E||2E| \leq 2^n D^2$ operations in $k$. Using a naive series multiplication routine, the Laurent series product also requires $2^n D^2$ operations in $k$.

## 8.5 Experimental Results

The algorithms underlying Theorems 9 and 10 have been implemented in the Magma computer algebra system [32]. In this section, we compare the methods presented respectively in Subsections 8.3.3 and 8.4.2, for the computation of a parametrization of the solutions of a polynomial system. Recall that the two methods differ by their input, respectively some multiplication matrices or the whole multiplication table, and by the computation of the power projection.

Since our complexity estimates are stated in terms of operations in the base field, we insist on computations on a finite field, where such operations have almost constant cost. Our base field is thus the finite field with 9001 elements.

The systems we have chosen are presented in Figure 8.1. All of them are complete intersection zero-dimensional systems. Systems 1 and 2 were proposed by S. Mallat for the design of foveal wavelets [161]. Systems 3 and 4 are the Cyclic systems [26] for $n = 6$ and $n = 7$. Systems 5 and 6 are sparse systems, with about 10 monomials of degree at most 4 per equation, and a single higher-degree monomial. Systems 7 and 8 are obtained by applying a linear change of variables on the previous systems.

- The first lines indicate the number of variables and the maximum degree of the input equations, then the dimension of the quotient algebra, that is the number of solutions counted with multiplicities.

| System | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Variables | 3 | 4 | 6 | 7 | 3 | 4 | 3 | 4 |
| Max. Degree | 12 | 12 | 6 | 7 | 12 | 6 | 12 | 6 |
| Solutions | 30 | 192 | 156 | 962 | 1728 | 1296 | 1728 | 1296 |
| Gröbner basis | 1 | 4 | 4.5 | 309 | 0.2 | 0.2 | 6.2 | 170 |
| Reconstruction | 0.2 | 0.1 | 0.1 | 0.5 | 4 | 6 | 7 | 8 |

Algorithms of Section 8.3.3:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Mult. Matrices | 0.1 | 2 | 1 | 6 | 3 | 4 | 5 | 30 |
| Power Projection | 0.4 | 3.6 | 3 | 57 | 695 | 763 | 700 | 1220 |
| Total | 0.5 | 5.6 | 4 | 63 | 698 | 767 | 705 | 1250 |

Algorithms of Section 8.4.2:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Mult. Table | 0.2 | 2.5 | 1.5 | 80 | 24 | 54 | 403 | 1330 |
| Power Projection | 0.3 | 2.1 | 2.2 | 20 | 164 | 250 | 290 | 370 |
| Total | 0.5 | 4.6 | 3.7 | 100 | 188 | 304 | 693 | 1700 |

Figure 8.1: Experimental Data; times are given in seconds

- For all systems, the separating element is a randomly chosen linear combination of the variables, and the linear form has only 5 nonzero coefficients on the dual basis. In all cases, we find a minimal polynomial of degree the dimension of the quotient, so the output is correct.

- A basis for the quotient algebra is computed using Magma's `GroebnerBasis` function for a Graded Reverse Lexicographical order. Its computation time is given in the line labelled "Gröbner Basis". The line labelled "Reconstruction" gives the time necessary to perform all reconstruction operations, that is, Berlekamp Massey's algorithm and univariate polynomial multiplications. Their cost is detailed in Propositions 10 and 11, and is the same for both approaches.

- The computation times are next given for both approaches. For the algorithm of Section 8.3.3, this includes the computation of some multiplication matrices (using Magma's `RepresentationMatrix` function), then the naive version of the power projection. For the algorithm of Section 8.4.2, this includes the computation of the whole multiplication table, which enables a faster version of the power projection.

As was to be expected, the baby steps/giant steps techniques bring a consequent speed up

over the naive version of the power projection. On the other hand, the precomputation of the whole multiplication table obviously affects this speed-up.

Systems 5 and 6 were chosen such that the Gröbner basis and the multiplication table were fast to compute. The advantage of using baby step/giant step techniques appears clearly for such examples.

Remark that the algorithm in [204] first requires to compute the whole multiplication table, then computes a power projection using the slower technique, i.e. without using the baby steps / giant steps techniques. The solutions we present here are certainly competitive with this approach.

## 8.6 Proof of Theorem 11

In this section, we prove Theorem 11. The data is a finite dimensional quotient algebra $A = k[X_1, \ldots, X_n]/\mathcal{I}$ over a perfect field $k$, a separating element $u$ in $A$ and a linear form $\ell$ in $\widehat{A}$. Our assumptions are as follows:

**Assumption 1** *The following conditions hold:*

- *the characteristic of $k$ is zero or greater than* $\min\{s, \sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}\}$*;*
- *the degree of the minimal polynomial $m_u$ of $u$ equals the degree of the minimal polynomial of a generic element in $A$ (see definition below);*
- *$\ell$ and $u$ are such that*

$$R(u, \ell) := \sum_{i \geq 0} \frac{\ell(u^i)}{U^{i+1}} = \frac{G_{u,\ell}}{m_u},$$

  *with $G_{u,\ell}$ and $m_u$ coprime (the definitions of the series $R$ and the polynomial $G_{u,\ell}$ are given in Section 8.2).*

Note that if $\mathcal{I}$ is a radical ideal, then the first two assumptions are satisfied as soon as $u$ is a separating element, since in this case the degree of the minimal polynomial of $u$ equals the dimension $D$ of $A$. The number $\min\{s, \sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}\}$ is called the *exponent* of $\mathcal{I}$; it equals 1 if $\mathcal{I}$ is radical.

Our goal is to show that for every $v$ in $A$ and for every $\alpha \in \mathcal{V}(\mathcal{I})$,

$$\left(\frac{G_{u,v\circ\ell}}{G_{u,\ell}}\right)(u(\alpha)) = v(\alpha).$$

Recall the in the particular case when $\mathcal{I}$ is radical, we indicated, in the comments following Theorem 11, a quick proof of these formulæ. The rest of the chapter is devoted to the proof in the general case. Since the arguments are a little involved, we divide their exposition in three parts. In Subsection 8.6.1 we relate the factorization of $m_u$ and the exponents of the primary components of $\mathcal{I}$; the main result is Proposition 13, which is an analogue for

minimal polynomials of a classical result on characteristic polynomials, sometimes referred to as Stickelberger's theorem [70, Proposition 2.7].

In Subsection 8.6.2 we rewrite the series $R(u, v \circ \ell)$ using a description of $\widehat{A}$ by differential conditions on the local factors of $A$. Finally, our knowledge of the factorization of $m_u$ will make it possible to read out the required result on the new expression of $R(u, v \circ \ell)$ and to conclude in Subsection 8.6.3 the proof of Theorem 11.

### 8.6.1 Minimal polynomials of generic elements and local factors

Given the $k$-algebra $A = k[X_1, \ldots, X_n]/\mathcal{I}$ and its basis $\Omega = (\omega_1, \ldots, \omega_D)$, we recall that we call the *generic element* in $A$ the element $T := \sum_{i=1}^{D} T_i \omega_i$ in $A \otimes_k k(T_1, \ldots, T_D)$. We denote by $m_T$ the minimal polynomial of $T$ and by $\delta(A)$ the degree of $m_T$. The polynomial $m_T$ depends on the choice of the basis $\Omega$, but its degree depends only on $A$. The numbers $\delta(A_\alpha)$ will be used in the next paragraph, for some algebras $A_\alpha$ to be introduced. They are defined in the same manner.

**Reduction to the case $k$ algebraically closed.**

This first section encloses a result on transfer properties of ideals in polynomial algebras under extension from $k$ to its algebraic closure $\overline{k}$. This result will serve to reduce the proof of Theorem 11 to the case when $k$ algebraically closed. In the lemma below, if $\mathcal{J}$ is an ideal in $k[X_1, \ldots, X_n]$, we denote by $\overline{\mathcal{J}}$ the ideal it generates in $\overline{k}[X_1, \ldots, X_n]$, that is, the set of all finite sums $\sum a_i f_i$, where $a_i \in \overline{k}[X_1, \ldots, X_n]$ and $f_i \in \mathcal{J}$. We will particularly focus on the ideal $\overline{\mathcal{I}}$, and we will denote $\overline{A} = \overline{k}[X_1, \ldots, X_n]/\overline{\mathcal{I}}$.

**Lemma 13** *The following results hold:*

- *The ideal $\overline{\mathcal{I}}$ is zero-dimensional in $\overline{k}[X_1, \ldots, X_n]$ and $\dim_k A$ equals $\dim_{\overline{k}} \overline{A}$.*
- *The minimal polynomial over $k$ of an element $u$ in $A$ coincides with the minimal polynomial of $u$ as an element of $\overline{A}$ over $\overline{k}$.*
- *The degree of the minimal polynomial of a generic element in $A$ equals the degree of the minimal polynomial of a generic element in $\overline{A}$.*
- *The exponent of $\overline{\mathcal{I}}$ equals the exponent of $\mathcal{I}$.*

Before starting the proof, we stress the fact that the first three points do not require that $k$ is a perfect field, while for the last point, this hypothesis is crucial, as showed by the following example. Let $k$ be the field $\mathbb{F}_p(Y)$ of rational functions over the finite field with $p$ elements; then the polynomial $X^p - Y$ is square-free over $k$ but not over $\overline{k}$, therefore the ideal it generates in $k[X]$ is radical, while its extension to $\overline{k}[X]$ is not.

*Proof.* The first assertion is a classical one, we refer to [136, Corollary 3.7.3] for a proof. The second and the third assertions are direct consequences of the fact that minimal polynomials are invariant under change of base ring, see for instance [143, Chapter XIV, Corollary 2.2].

It remains to prove the last assertion. We begin by showing that the operations of extending an ideal and taking the radical of an ideal commute, that is the ideals $\overline{\sqrt{\mathcal{I}}}$ and $\sqrt{\overline{\mathcal{I}}}$ are equal. Since $\sqrt{\mathcal{I}}$ contains $\mathcal{I}$ and extending ideals preserves inclusion, we have that $\overline{\mathcal{I}} \subset \overline{\sqrt{\mathcal{I}}}$. Since $k$ is a perfect field, and $\sqrt{\mathcal{I}}$ is radical, [136, Proposition 3.7.18] shows that its extension $\overline{\sqrt{\mathcal{I}}}$ is also radical, so taking again radicals in $\overline{\mathcal{I}} \subset \overline{\sqrt{\mathcal{I}}}$, we obtain the first inclusion $\sqrt{\overline{\mathcal{I}}} \subset \overline{\sqrt{\mathcal{I}}}$.

Let us now justify the converse inclusion $\overline{\sqrt{\mathcal{I}}} \subset \sqrt{\overline{\mathcal{I}}}$. Since $\overline{\mathcal{I}}$ contains $\mathcal{I}$ and taking radicals preserves inclusion, we have that $\sqrt{\mathcal{I}} \subset \sqrt{\overline{\mathcal{I}}}$. Thus any element $y$ in $\overline{\sqrt{\mathcal{I}}}$ may be written as a finite sum $\sum_i a_i f_i$, for some polynomials $a_i$ with coefficients in $\overline{k}$ and some $f_i$ belonging to $\sqrt{\overline{\mathcal{I}}}$, so $y \in \sqrt{\overline{\mathcal{I}}}$. Thus, the equality of $\overline{\sqrt{\mathcal{I}}}$ and $\sqrt{\overline{\mathcal{I}}}$ is proved.

We finally prove the last assertion concerning the exponent preservation under extension to $\overline{k}$. By definition of the exponent, it is enough to show that $\sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}$ if and only if $\sqrt{\overline{\mathcal{I}}}^{\,s} \subset \overline{\mathcal{I}}$. For the direct assertion, suppose that $\sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}$. Taking extensions and using the property proved in the previous paragraph, we deduce $\sqrt{\overline{\mathcal{I}}}^{\,s} \subset \overline{\mathcal{I}}$.

Conversely, suppose that $\sqrt{\overline{\mathcal{I}}}^{\,s} \subset \overline{\mathcal{I}}$. Intersecting both sides with $k[X_1, \ldots, X_n]$ (this operation is called *contraction*), we assert that we recover $\sqrt{\mathcal{I}}^{\,s} \subset \mathcal{I}$. In order to justify this, we use the fact that in polynomial algebras, extension followed by contraction of an ideal returns the initial ideal, see for instance [142, Chapter III, Proposition 7]. Indeed, this fact, in conjunction with the previous arguments implies the equalities $\mathcal{I} = k[X_1, \ldots, X_n] \cap \overline{\mathcal{I}}$ and $\sqrt{\mathcal{I}}^{\,s} = \sqrt{\overline{\mathcal{I}}}^{\,s} \cap k[X_1, \ldots, X_n]$, and this concludes the proof of our lemma. □

**Minimal polynomials of generic elements.**

The following lemma shows that over an algebraically closed field, the degree of the minimal polynomial of a generic element in $A$ equals the maximal degree of all minimal polynomials of elements in $A$. We point out that this result applies to any algebra of finite dimension, and will be used for the algebras $A_\alpha$ introduced in the next paragraph.

**Lemma 14** *For every $t$ in $A$, $\deg m_t \leq \delta(A)$, and there exists $t$ in $A$ such that $\deg m_t = \delta(A)$. In other words, $\delta(A) = \max_{t \in A}(\deg m_t)$.*

*Proof.* Let $B$ be $A \otimes_k k(T_1, \ldots, T_D)$ and let $T \in B$ be $\sum_{i=1}^{D} T_i \omega_i$. The $k$-basis $\Omega$ of $A$ is also a $k(T_1, \ldots, T_D)$-basis of $B$. We define $\mathbf{M}_T$ as the matrix of multiplication by $T$ in this basis; then $m_T(\mathbf{M}_T) = 0$.

Let $t$ be in $A$; $t$ can be written $\sum_{i=1}^{D} t_i \omega_i$. Both $\mathbf{M}_T$ and $m_T$ have their coefficients in $k[T_1, \ldots, T_D]$, so the equality $m_T(\mathbf{M}_T) = 0$ can be specialized at $(t_1, \ldots, t_D)$. The matrix $\mathbf{M}_T$ specializes into the multiplication matrix of $t$ in $A$, which shows that $\deg m_t \leq \deg m_T = \delta(A)$.

Consider now the $D \times \delta(A)$ matrix whose columns contain the coefficients of $T^0, \ldots, T^{\delta(A)-1}$ on the basis $\Omega$. This matrix has entries that are polynomial in $(T_1, \ldots, T_D)$, and has maximal rank, so admits a $\delta(A) \times \delta(A)$ submatrix with nonzero determinant $\mathcal{D} \in k[T_1, \ldots, T_D]$.

Since $k$ is algebraically closed, there exists a $D$-tuple $(t_1, \ldots, t_D)$ which does not cancel $\mathcal{D}$. Then the first $\delta(A) - 1$ powers of $t = \sum_{i=1}^{D} t_i \omega_i$ are independent over $k$, so the minimal polynomial of $t$ has degree $\delta(A)$. □

**Minimal polynomials and local factors.**

Let $u \in A$ be an element of $A$, whose minimal polynomial $m_u$ has degree $\delta(A)$, the degree of the minimal polynomial of a generic element in $A$. The aim of the rest of this section is to describe the factorization properties of the polynomial $m_u$.

Since $k$ is algebraically closed, each zero $\alpha$ of $\mathcal{I}$ is in $k^n$. Moreover, if we let $\mathfrak{m}_\alpha \subset k[X_1, \ldots, X_n]$ be the maximal ideal at $\alpha$, then the primary decomposition of the zero-dimensional ideal $\mathcal{I}$ has the form:

$$\mathcal{I} = \bigcap_{\alpha \in \mathcal{V}(\mathcal{I})} \mathcal{I}_\alpha,$$

where $\mathcal{I}_\alpha$ is a $\mathfrak{m}_\alpha$-primary ideal.

We write $A_\alpha$ for the local algebra $k[X_1, \ldots, X_n]/\mathcal{I}_\alpha$ and denote by $N_\alpha$ the exponent of $\mathcal{I}_\alpha$, that is the minimal $s$ such that $\mathfrak{m}_\alpha^s \subset \mathcal{I}_\alpha$. This is also the nil-index of the local algebra $A_\alpha$.

The main result of this section shows that under Assumption 1, the minimal polynomial of $u$ equals

$$m_u = \prod_{\alpha \in \mathcal{V}(\mathcal{I})} \big(U - u(\alpha)\big)^{N_\alpha}.$$

This fact is crucial in proving Theorem 11; we divide its proof into several lemmas.

**Lemma 15** *Suppose $u \in A$ has minimal polynomial $m_u$ of degree $\delta(A)$. Then the minimal polynomial of $u$ is given by*

$$m_u = \prod_{\alpha \in \mathcal{V}(\mathcal{I})} \big(U - u(\alpha)\big)^{\delta(A_\alpha)}.$$

*Proof.* By the Chinese Remainder Theorem, $A$ is isomorphic to the product $\prod_\alpha A_\alpha$. We denote by $u_\alpha$ the images of $u$ in $A_\alpha$ under this isomorphism. Let us show that the minimal polynomial of $u$ equals the least common multiple of the minimal polynomials $m_{u_\alpha}$.

For any polynomial $P$, the image in $A_\alpha$ of the element $P(u)$ under the Chinese isomorphism is $P(u_\alpha)$. Since $m_u(u) = 0$, this implies that $m_u(u_\alpha) = 0$ for all $\alpha$, therefore all $m_{u_\alpha}$ divide $m_u$. Conversely, let $m$ be a polynomial divisible by all $m_{u_\alpha}$. It follows that $m(u_\alpha) = 0$ for all $\alpha$, so $m(u) = 0$. Thus, $m_u$ divides $m$ and this proves that $m_u$ is the lcm of $m_{u_\alpha}$. As a consequence, we have the inequality

$$\delta(A) \leq \sum_\alpha \deg m_{u_\alpha} \leq \sum_\alpha \delta(A_\alpha). \tag{8.3}$$

We next show that for all $\alpha$, the polynomial $m_{u_\alpha}$ has the form $\big(T - u(\alpha)\big)^{s_\alpha}$, for some integer $1 \leq s_\alpha \leq N_\alpha$. Since it vanishes on $\alpha$, the element $u_\alpha - u(\alpha)$ belongs to the radical $\mathfrak{m}_\alpha$ of $\mathcal{I}_\alpha$. It follows that the element $\big(u_\alpha - u(\alpha)\big)^{N_\alpha}$ belongs to $\mathcal{I}_\alpha$, thus is zero in the quotient $A_\alpha$. Therefore, $m_{u_\alpha}$ divides $\big(T - u(\alpha)\big)^{N_\alpha}$, hence it has the form $\big(U - u(\alpha)\big)^{s_\alpha}$. Since $m_u$ equals their lcm, it has the form $m_u = \prod_\alpha \big(U - u(\alpha)\big)^{r_\alpha}$.

We show now that $r_\alpha = \delta(A_\alpha)$, for all $\alpha$. Using Lemma 14 for each $\alpha$ in $\mathcal{V}(\mathcal{I})$, we choose elements $t_\alpha$ in $A_\alpha$ such that the degree of the minimal polynomial of $t_\alpha$ is $\delta(A_\alpha)$. The

previous paragraph shows that, up to adding well-chosen constants to the $t_\alpha$, we can assure that their minimal polynomials are pairwise coprime. Let $t \in A$ be such that its images in the local algebras $A_\alpha$ are the elements $t_\alpha$. Then the minimal polynomial of $t$ is the product $\prod_\alpha m_{t_\alpha}$, so its degree is $\sum_\alpha \delta(A_\alpha)$. Thus:

$$\sum_\alpha \delta(A_\alpha) \leq \delta(A). \tag{8.4}$$

Combining the inequalities (8.3) and (8.4) with the fact that $\delta(A) = \deg m_u$ equals $\sum_\alpha r_\alpha$, we conclude that $r_\alpha = \delta(A_\alpha)$, for all $\alpha$, so $m_u$ has the desired form. □

The next lemma relates the degree $\delta(A_\alpha)$ to the local exponents $N_\alpha$. We point out that this result depends on the characteristic of the base field $k$.

**Lemma 16** *Let $a = (a_1, \ldots, a_n) \in k^n$, let $\mathcal{J}$ be a $(X_1 - a_1, \ldots, X_n - a_n)$-primary ideal of $k[X_1, \ldots, X_n]$, let $N_\mathcal{J}$ be the exponent of $\mathcal{J}$ and let $A_\mathcal{J}$ be $k[X_1, \ldots, X_n]/\mathcal{J}$. If the characteristic of $k$ is zero or greater than $N_\mathcal{J} - 1$ then $\delta(A_\mathcal{J}) = N_\mathcal{J}$.*

*Proof.* Up to a translation, we may assume that the point $a$ is the origin of $k^n$ and that the ideal $\mathcal{J}$ is $(X_1, \ldots, X_n)$-primary.

Let $D_\mathcal{J}$ be the dimension of $A_\mathcal{J}$ and $\beta_1, \ldots, \beta_{D_\mathcal{J}}$ be a monomial basis of $A_\mathcal{J}$. We suppose that $\beta_1 = 1$. By Lemma 14, we can choose $t := \sum_{i=1}^{D_\mathcal{J}} t_i \beta_i$ such that $\deg m_t = \delta(A_\mathcal{J})$. Then $t - t_1$ is in $(X_1, \ldots, X_n)$, so $(t - t_1)^{N_\mathcal{J}} = 0$. This shows that the degree of the minimal polynomial of $t$ is at most $N_\mathcal{J}$, i.e. $\delta(A_\mathcal{J}) \leq N_\mathcal{J}$.

By assumption, there exists a monomial $M$ of total degree $N_\mathcal{J} - 1$ which is not in $\mathcal{J}$. Without loss of generality, $M$ can be written $\prod_{i=1}^d X_i^{\alpha_i}$ for some integer $1 \leq d \leq D_\mathcal{J}$ and some positive integers $\alpha_i$, of sum $N_\mathcal{J} - 1$. We let $t$ be $\sum_{i=1}^d X_i$. The coefficient of $M$ in $t^{N_\mathcal{J}-1}$ is

$$\frac{(N_\mathcal{J} - 1)!}{\alpha_1! \cdots \alpha_d!},$$

which is well-defined and nonzero since the characteristic of $k$ is either zero or greater than $N_\mathcal{J} - 1$. Consequently, $t^{N_\mathcal{J}-1}$ is not zero, so the minimal polynomial of $t$ is $T^{N_\mathcal{J}}$. This shows that $N_\mathcal{J} \leq \delta(A_\mathcal{J})$. The converse inequality follows from the first part of Lemma 15. This concludes the proof. □

To apply this result to each local factor, we need to ensure that the characteristic of $k$ is indeed greater than the exponents of the local factors. This is the objective of the next lemma.

**Lemma 17** *The exponent of $\mathcal{I}$ equals* $\max_{\alpha \in \mathcal{V}(\mathcal{I})} N_\alpha$.

*Proof.* Let $S$ be the exponent of $\mathcal{I}$ and $N$ be $\max_{\alpha \in \mathcal{V}(\mathcal{I})} N_\alpha$. Then $\sqrt{\mathcal{I}}^N$ is $\prod_\alpha \mathfrak{m}_\alpha^N$, which is contained in $\prod_\alpha \mathcal{I}_\alpha = \mathcal{I}$, so $S \leq N$. Conversely, for any $\alpha$ in $\mathcal{V}(\mathcal{I})$, we have

$$\mathcal{I}_\alpha + \prod_{\alpha' \neq \alpha} \mathfrak{m}_{\alpha'}^S = (1).$$

Multiplying both sides by $\mathfrak{m}_\alpha^S$ yields

$$\mathfrak{m}_\alpha^S \mathcal{I}_\alpha + \prod_{\alpha' \in \mathcal{V}(\mathcal{I})} \mathfrak{m}_{\alpha'}^S = \mathfrak{m}_\alpha^S.$$

Now $S$ is such that $\sqrt{\mathcal{I}}^S \subset \mathcal{I}$, so $\prod_{\alpha' \in \mathcal{V}(\mathcal{I})} \mathfrak{m}_{\alpha'}^S \subset \mathcal{I} \subset \mathcal{I}_\alpha$. The previous equality then shows that $\mathfrak{m}_\alpha^S \subset \mathcal{I}_\alpha$, for each $\alpha$, hence $S \geq N$. □

The following proposition summarizes the results of this section.

**Proposition 13** *Let $u$ be in $A$, such that the degree of its minimal polynomial $m_u$ equals the degree of the minimal polynomial of a generic element in $A$. If furthermore the characteristic of $k$ is zero or greater than the exponent of $\mathcal{I}$, then the polynomial $m_u$ factorizes as*

$$m_u = \prod_{\alpha \in \mathcal{V}(\mathcal{I})} \big(U - u(\alpha)\big)^{N_\alpha}.$$

*Proof.* By assumption and using Lemma 17, we are in position to apply Lemma 16 on each local factor $A_\alpha$. Together with Lemma 15, this gives the result. □

### 8.6.2 High order derivations, dual spaces and generating series

In this section, we recall the notion of *high order derivations* and exhibit their connection with the dual spaces of quotient algebras. We also give a description of some generating series of the type $R(u, \ell)$ which are built upon such derivations.

#### Basic facts.

We start by recalling the notion of high order derivation over an algebra, introduced in [181, 178]. Let $k$ be an arbitrary field and $R$ be a $k$-algebra. A $k$-linear map $d : R \to R$ is called a *$k$-derivation of order 1* if $d(xy) = xd(y) + yd(x)$, for all $x$ and $y$ in $R$. High order derivations are defined recursively. A $k$-linear map $d : R \to R$ is called a *$k$-derivation of order $N > 1$* if the map $[d, x] : y \mapsto d(xy) - xd(y) - yd(x)$ is a $k$-derivation of order $N - 1$ for all $x \in R$. For $N \geq 1$, we write $\mathrm{Der}_k^N(R)$ for the $k$-vector space of all $k$-derivations of order $N$, and we take $\mathrm{Der}_k^0(R) = k \cdot 1_R$. One can easily show that $d(1) = 0$ for any derivation $d$ of order at least 1 and that $\mathrm{Der}_k^N(R) \subset \mathrm{Der}_k^{N+1}(R)$ for all $N \geq 1$, see [28, Section 1]. These two basic properties will be implicitly used in the proofs below.

In the particular case $R = k[X_1, \ldots, X_n]$, the $k$-linear map $\delta^v : R \to R$ defined on the monomial basis by:

$$\delta^v : X_1^{\mu_1} \cdots X_n^{\mu_n} \mapsto \binom{\mu_1}{v_1} \cdots \binom{\mu_n}{v_n} X_1^{\mu_1 - v_1} \cdots X_n^{\mu_n - v_n}$$

is a $k$-derivation in $\mathrm{Der}^{|v|}(R)$, with $|v| = v_1 + \cdots + v_n$. Remark that the binomial coefficient $\binom{\beta}{\alpha}$ is defined over any field, for instance as the coefficient of $Y^\alpha$ in $(1 + Y)^\beta$. If $k$ has characteristic zero, then we recover the well-known definition of differential operators:

$$\delta^v(P) = \frac{1}{v_1! \cdots v_n!} \frac{\partial^{v_1 + \cdots + v_n}(P)}{\partial_{X_1}^{v_1} \cdots \partial_{X_n}^{v_n}}.$$

**Dual spaces and high order derivations.**

We next exhibit the connection between high order derivations and dual spaces of quotient algebras. The idea to characterize primary ideals by differential conditions in characteristic zero is due to Gröbner [114]. Similar or more general treatment can be found in [164, 172, 28, 180]. For the sake of completeness, we gather in the following lemma the needed facts, in *arbitrary* characteristic. Our proof is inspired by that of [28, Proposition 3.2].

**Lemma 18** *Let $a = (a_1, \ldots, a_n) \in k^n$, let $\mathcal{J}$ be a $(X_1 - a_1, \ldots, X_n - a_n)$-primary ideal of $R = k[X_1, \ldots, X_n]$ and let $N_{\mathcal{J}}$ be the exponent of $\mathcal{J}$. Then there exists a $k$-basis of the dual $\widehat{R/\mathcal{J}}$ consisting of elements*

$$L_i : P + \mathcal{J} \longmapsto (D_i P)(a),$$

*where $D_1$ is the identity map and with $D_i$ in $\mathrm{Der}_k^{N_{\mathcal{J}}-1}(R)$ for $i > 1$.*

*Proof.* Up to a translation, we assume, without loss of generality, that the point $a$ is the origin of $k^n$ and that the ideal $\mathcal{J}$ is $(X_1, \ldots, X_n)$-primary. If $v$ is a multi-index with $|v| < N_{\mathcal{J}}$, the $k$-linear map $R \to k$ given by $P \mapsto (\delta^v P)(0)$ factors to a $k$-linear map $\delta_*^v : R/(X_1, \ldots, X_n)^{N_{\mathcal{J}}} \to k$ and the induced maps $\{\delta_*^v\}_{|v|<N_{\mathcal{J}}}$ form the dual $k$-basis of the monomial basis $\{x^\mu\}_{|\mu|<N_{\mathcal{J}}}$ of $R/(X_1, \ldots, X_n)^{N_{\mathcal{J}}}$.

The dual of $R/\mathcal{J}$ is a $k$-linear subspace of the dual of $R/(X_1, \ldots, X_n)^{N_{\mathcal{J}}}$, which contains $\delta_*^0$. Thus, it admits a $k$-basis whose elements are of the form $L_1 = \delta_*^0$ and $L_i = \sum_{0<|v|<N_{\mathcal{J}}} b_v^{(i)} \delta_*^v$ for $i > 1$. We take $D_1$ as the identity map and, for $i > 1$, $D_i = \sum_{0<|v|<N_{\mathcal{J}}} b_v^{(i)} \delta^v$, so that $D_i \in \mathrm{Der}_k^{N_{\mathcal{J}}-1}(R)$. This proves the lemma. □

**High order derivations and generating series.**

The following result makes a link between the poles of the rational series $R(u, \ell)$ introduced in Proposition 12 and the order of a derivation.

**Lemma 19** *Let $N \geq 0$, $R$ be a $k$-algebra, $u \in R$ and $D$ in $\mathrm{Der}^N(R)$. Then there exists $c$ in $R$ such that, for every $v \in R$, there exist $N$ elements $c_j$ in $R$ such that the following equality holds in $R[[U^{-1}]]$:*

$$\sum_{i \geq 0} \frac{D(vu^i)}{U^{i+1}} = \frac{cv}{(U-u)^{N+1}} + \sum_{j=1}^{N} \frac{c_j}{(U-u)^j}.$$

*Proof.* We proceed by induction on $N$. We begin by considering the case $N = 0$, that is, $D$ is the multiplication map by a certain element $r$ in $R$. We have that

$$\sum_{i \geq 0} \frac{D(vu^i)}{U^{i+1}} = rv \sum_{i \geq 0} \frac{u^i}{U^{i+1}} = \frac{rv}{U-u},$$

so this series has the desired form.

We treat now the inductive step. Let $N \geq 1$; we suppose the lemma is true for index $N-1$ and we prove it for index $N$. Let thus $D$ be an arbitrary derivation in $\mathrm{Der}^N(R)$. By definition, we have the formula $D(vu^i) = [D, v](u^i) + vD(u^i) + u^i D(v)$, so

$$\sum_{i\geq 0}\frac{D(vu^i)}{U^{i+1}}=\sum_{i\geq 0}\frac{[D,v](u^i)}{U^{i+1}}+v\sum_{i\geq 0}\frac{D(u^i)}{U^{i+1}}+D(v)\sum_{i\geq 0}\frac{u^i}{U^{i+1}}. \tag{8.5}$$

We analyze each term in this sum separately. Since $[D,v]$ belongs to $\mathrm{Der}^{N-1}(R)$, the induction hypothesis shows that

$$\sum_{i\geq 0}\frac{[D,v](u^i)}{U^{i+1}}=\frac{c'}{(U-u)^N}+\sum_{j=1}^{N-1}\frac{c'_j}{(U-u)^j}$$

for some elements $c'$ and $c'_j$ in $R$. Using the fact that $D(u^i)=[D,u](u^{i-1})+uD(u^{i-1})+u^{i-1}D(u)$, it is easy to derive the formula

$$\sum_{i\geq 0}\frac{D(u^i)}{U^{i+1}}=\frac{1}{U-u}\sum_{i\geq 0}\frac{[D,u](u^i)}{U^{i+1}}+\frac{D(u)}{(U-u)^2}.$$

By the inductive hypothesis, the second term in the sum (8.5) is thus equal to

$$\frac{v}{U-u}\left(\frac{c''}{(U-u)^N}+\sum_{j=1}^{N-1}\frac{c''_j}{(U-u)^j}+\frac{D(u)}{(U-u)}\right),$$

for some elements $c''$ and $c''_j$ in $R$ depending *only* on $D$ and $u$. Finally, the third term in the sum (8.5) obviously equals

$$D(v)\sum_{i\geq 0}\frac{u^i}{U^{i+1}}=\frac{D(v)}{U-u}.$$

Putting these pieces all together in sum (8.5) completes the proof. □

### 8.6.3 Conclusion

The final step of the proof consists in rewriting the series $R(u,v\circ\ell)$ so as to exhibit its dependence with respect to $v$. Lemma 18 shows that for each $\alpha\in\mathcal{V}(\mathcal{I})$ there exists a family of derivations $\Delta^\alpha=\{D_j^\alpha\}_{j=1,\ldots,\dim_k(A_\alpha)}$, such that the functionals

$$L_j^\alpha : P+\mathcal{I}_\alpha\mapsto D_j^\alpha(P)(\alpha)$$

form a $k$-basis of $\widehat{A_\alpha}$. Furthermore, $D_1^\alpha=1$ and for $j>1$, $D_j^\alpha$ belongs in $\mathrm{Der}^{N_\alpha-1}(k[X_1,\ldots,X_n])$. Using Lemma 19 and evaluating at $\alpha$, we see that there exist $c_j^\alpha$ in $k$, and, for every $v\in k[X_1,\ldots,X_n]$, some elements $(c_{j,i}^\alpha)_{1\leq i<N}$ in $k$ such that

$$R(u,v\circ L_1^\alpha)=\sum_{i\geq 0}\frac{(vu^i)(\alpha)}{U^{i+1}}=\frac{v(\alpha)}{U-u(\alpha)} \tag{8.6}$$

and, for $j>1$,

$$R(u,v\circ L_j^\alpha)=\sum_{i\geq 0}\frac{D_j^\alpha(vu^i)(\alpha)}{U^{i+1}}=\frac{v(\alpha)c_j^\alpha}{(U-u(\alpha))^{N_\alpha}}+\sum_{i=1}^{N_\alpha-1}\frac{c_{j,i}^\alpha}{(U-u(\alpha))^j} \tag{8.7}$$

hold in $k[[U^{-1}]]$.

Let now $\ell$ be in $\widehat{A}$. Since the union $\cup_\alpha \Delta^\alpha$ forms a $k$-basis of $\widehat{A}$, and using the linearity of $R(u, v \circ \ell)$ with respect to $\ell$, equations (8.6) and (8.7) show that for every $v$ the equality

$$R(u, v \circ \ell) = \sum_{\alpha \in \mathcal{V}(\mathcal{I})} \frac{v(\alpha) c_\alpha}{(U - u(\alpha))^{N_\alpha}} + \sum_{\alpha \in \mathcal{V}(\mathcal{I})} \sum_{j=1}^{N_\alpha - 1} \frac{c_j^\alpha}{(U - u(\alpha))^j} \tag{8.8}$$

holds, where $c_\alpha$ and $c_j^\alpha$ belong to $k$, and $c_\alpha$ does not depend on $v$. If one of the coefficients $c_\alpha$ were zero, then for any $v$, $R(u, v \circ \ell)$ could be written with a denominator of degree less than $\sum_\alpha N_\alpha$, that is, of degree less than $\deg m_u$, by Proposition 13. In particular, for $v = 1$, $R(u, \ell)$ would admit a denominator of degree less than $\deg m_u$. Since, by Assumption 1, $\ell$ is such that

$$R(u, \ell) = \frac{G_{u,\ell}}{m_u},$$

with $G_{u,\ell}$ and $m_u$ coprime, none of the coefficients $c_\alpha$ can be zero.

Recall that by Proposition 13, the polynomial $m_u$ writes as

$$m_u = \prod_{\alpha \in \mathcal{V}(\mathcal{I})} \big(U - u(\alpha)\big)^{N_\alpha}.$$

Let $Q_\alpha$ be the quotient of $m_u$ by $(U - u(\alpha))^{N_\alpha}$, so that $Q_\alpha$ takes a nonzero value on $u(\alpha)$. Using equation (8.8), we deduce that for any $v$, there exists a polynomial $V_v \in k[U]$ such that

$$G_{u,v\circ\ell} = m_u R(u, v \circ \ell) = \sum_{\alpha \in \mathcal{V}(\mathcal{I})} v(\alpha) c_\alpha Q_\alpha(U) + V_v(U) \prod_{\alpha \in \mathcal{V}(\mathcal{I})} (U - u(\alpha)).$$

This implies that $G_{u,v\circ\ell}(u(\alpha))$ equals $v(\alpha) c_\alpha Q_\alpha(u(\alpha))$. Since $c_\alpha Q_\alpha(u(\alpha))$ is not zero and is independent from $v$, this shows that $\frac{G_{u,v\circ\ell}}{G_{u,\ell}}$ takes the value $v(\alpha)$ at $u(\alpha)$. This proves the proposition.

# Part IV

# Fast Algorithms for Linear Recurrences and Linear Differential Operators

# Chapter 9

# Linear Recurrences with Polynomial Coefficients

We improve an algorithm originally due to Chudnovsky and Chudnovsky for computing one selected term in a linear recurrent sequence with polynomial coefficients. Using baby-steps / giant-steps techniques, the $n$th term in such a sequence can be computed in time proportional to $\sqrt{n}$, instead of $n$ for a naive approach.

As an intermediate result, we give a fast algorithm for computing the values taken by an univariate polynomial $P$ on an arithmetic progression, taking as input the values of $P$ on a translate on this progression.

We apply these results to the computation of the Cartier-Manin operator of a hyperelliptic curve. If the base field has characteristic $p$, this enables us to reduce the complexity of this computation by a factor of order $\sqrt{p}$. We treat a practical example, where the base field is an extension of degree 3 of the prime field with $p = 2^{32} - 5$ elements.

This chapter is joint work with É. Schost and P. Gaudry [34].

## Contents

## 9.1 Introduction

In this chapter, we investigate some complexity questions related to linear recurrent sequences. Specifically, we concentrate on recurrences with polynomial coefficients; our main focus is on the complexity of computing one selected term in such a recurrence.

A well-known particular case is that of recurrences with constant coefficients, where the $n$th term can be computed with a complexity that is logarithmic in $n$, using binary powering techniques.

In the general case, there is a significant gap, as for the time being no algorithm with complexity polynomial in $\log(n)$ is known. Yet, in [59], Chudnovsky and Chudnovsky proposed an algorithm that allows to compute one selected term in such a sequence without computing all intermediate ones. This algorithm appears as a generalization of those of Pollard [195] and Strassen [237] for integer factorization; using baby-steps / giant-steps techniques, it requires a number of operations which is roughly linear in $\sqrt{n}$ to compute the $n$th term in the sequence.

Our main contribution is an improvement of the algorithm of [59]; for simplicity, we only give the details in the case when all coefficients are polynomials of degree 1, as the study in the general case would follow in the same manner. The complexity of our algorithm is still (roughly) linear in $\sqrt{n}$; Chudnovsky and Chudnovsky actually suggested that this bound might be essentially optimal. We improve the time and space complexities by factors that are logarithmic in $n$; in practice, this is far from negligible, since in the application detailed below, $n$ has order $2^{32}$. A precise comparison with Chudnovsky and Chudnovsky's algorithm is made in Section 9.3.

Along the way, we also consider a question of basic polynomial arithmetic: given the values taken by a univariate polynomial $P$ on a set of points, how fast can we compute the values taken by $P$ on a translate of this set of points? An obvious solution is to make use of fast interpolation and evaluation techniques, but we show that one can do better when the evaluation points form an arithmetic sequence.

**Computing the Cartier-Manin operator.**

Our initial motivation is an application to point-counting procedures in hyperelliptic curve cryptography, related to the computation of the Cartier-Manin operator of curves over finite fields. We now present these matters in more detail.

The Cartier-Manin operator of a curve defined over a finite field, together with the Hasse-Witt matrix, are useful tools to study the arithmetic properties of the Jacobian of that curve. Indeed, the supersingularity, and more generally the $p$-rank, can be read from the invariants of the Hasse-Witt matrix. In the case of hyperelliptic curves, this matrix was used in [88, 166] as part of a point-counting procedure for cryptographic-oriented applications.

Indeed, thanks to a result of Manin, computing the Cartier-Manin operator gives the coefficients of the Zeta function modulo $p$; this partial information can then be completed by some other algorithms. However, in [88] and [166], the method used to compute the Hasse-Witt matrix has a complexity which is essentially linear in $p$.

It turns out that one can do better. The entries of the Hasse-Witt matrix of a hyperelliptic curve $y^2 = f(x)$ defined over a finite field of characteristic $p$ are coefficients of the polynomial $h = f^{(p-1)/2}$, so they satisfy a linear recurrence with rational function coefficients. Using our results on linear recurrences, this remark yields an algorithm to compute the Hasse-Witt matrix whose complexity now grows like $\sqrt{p}$, up to logarithmic factors, instead of $p$.

We demonstrate the interest of these techniques by a point-counting example, for a curve of genus 2 defined over a finite field whose characteristic just fits in one 32-bit machine word; this kind of fields have an interest for efficiency reasons [10].

Note finally that other point-counting algorithms, such as the $p$-adic methods used in Kedlaya's algorithm [131], also provide efficient point-counting procedures in small characteristic, but their complexity remains at least linear in $p$ [87]. On the other hand, Kedlaya's algorithm outputs the whole Zeta function and should be preferred if available. Therefore, the range of application of our algorithm is when the characteristic is too large for Kedlaya's algorithm to be run.

**Organization of the chapter.**

We start in Section 9.2 with our algorithm for shifting a polynomial given by its values on some evaluation points. This building block is used in Section 9.3 to describe our improvement on Chudnovsky and Chudnovsky's algorithm. In Section 9.4 we apply these results to the computation of the Cartier-Manin operator of a hyperelliptic curve. We conclude in Section 9.5 with a numerical example.

**Notation.**

In what follows, we give complexity estimates in terms of number of base ring operations (additions, subtractions, multiplications and inversions of unit elements) and of storage requirements; this last quantity is measured in terms of number of elements in the ring. We pay particular attention to polynomial and matrix multiplications and use the following notation.

- Let $R$ be a commutative ring; we suppose that $R$ is unitary, its unit element being denoted by $1_R$, or simply 1. Let $\varphi$ be the map $\mathbb{N} \to R$ sending $n$ to $n \cdot 1_R = 1_R + \cdots + 1_R$ ($n$ times); the map $\varphi$ extends to a map $\mathbb{Z} \to R$. When the context is clear, we simply denote the ring element $\varphi(n)$ by $n$.

- We denote by $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ a function that represents the complexity of univariate polynomial multiplication, *i.e.* such that over any ring $R$, the product of two degree $d$ polynomials can be computed within $\mathsf{M}(d)$ base ring operations. Using the algorithms of [214, 210, 50], $\mathsf{M}(d)$ can be taken in $O(d \log(d) \log(\log(d)))$.

  We suppose that the function $\mathsf{M}$ verifies the inequality $\mathsf{M}(d_1) + \mathsf{M}(d_2) \leq \mathsf{M}(d_1 + d_2)$ for all positive integers $d_1$ and $d_2$; in particular, the inequality $\mathsf{M}(d) \leq \frac{1}{2}\mathsf{M}(2d)$ holds for

all $d \geq 1$. On the other hand, we make the (natural) hypothesis that $\mathsf{M}(cd) \in O(\mathsf{M}(d))$ for all $c \geq 1$.

We also assume that the product of two degree $d$ polynomials can be computed in space $O(d)$; this is the case for all classical algorithms, such as naive, Karatsuba and Schönhage-Strassen multiplications.

- We let $\omega$ be a real number such that for every commutative ring $R$, all $n \times n$ matrices over $R$ can be multiplied within $O(n^\omega)$ operations in $R$. The classical multiplication algorithm gives $\omega = 3$. Using Strassen's algorithm [234], we can take $\omega = \log_2(7) \simeq 2.81$. We assume that the product of two $n \times n$ matrices can be computed in space $O(n^2)$, which is the case for classical as well as Strassen's multiplications.

In the sequel, we need the following classical result on polynomial arithmetic over $R$. The earliest references we are aware of are [169, 30], see [255] for a detailed account. We also refer to [35] for a solution that is in the same complexity class, but where the constant hidden in the $O(\ )$ notation is actually smaller than that in [255].

**Multipoint evaluation.** If $P$ is a polynomial of degree $d$ in $R[X]$ and $r_0, \ldots, r_d$ are points in $R$, then the values $P(r_0), \ldots, P(r_d)$ can be computed using $O(\mathsf{M}(d)\log(d))$ operations in $R$ and $O(d\log(d))$ space.

## 9.2 Shifting evaluation values

In this section, we address a particular case of the question of *shifting evaluation values* of a polynomial. The question reads as follows: Let $P$ be a polynomial of degree $d$ in $R[X]$, where $R$ is a commutative unitary ring. Let $a$ and $r_0, \ldots, r_d$ be in $R$. Given $P(r_0), \ldots, P(r_d)$, how fast can we compute $P(r_0 + a), \ldots, P(r_d + a)$?

A reasonable condition for this question to make sense is that all differences $r_i - r_j$, $i \neq j$, are units in $R$; otherwise, uniqueness of the answer might be lost. Under this assumption, using fast interpolation and fast multipoint evaluation, the problem can be answered within $O(\mathsf{M}(d)\log(d))$ operations in $R$. We now show that the cost reduces to $\mathsf{M}(2d) + O(d)$ operations in $R$, in the particular case when $r_0, \ldots, r_d$ are in arithmetic progression, so we gain a logarithmic factor.

Our solution reduces to the multiplication of two suitable polynomials of degree at most $2d$; $O(d)$ additional operations come from additional pre- and post-processing operations. As mentioned in Section 9.1, all operations made below on integer values actually take place in $R$.

The algorithm underlying Proposition 14 is given in Figure 9.1; we use the notation $\texttt{coeff}(Q, k)$ to denote the coefficient of degree $k$ of a polynomial $Q$. We stress the fact that the polynomial $P$ is *not* part of the input of our algorithm.

**Input** $P(0), \ldots, P(d)$ and $a$ in $R$
**Output** $P(a), \ldots, P(a+d)$

- Compute

$$\delta(0,d) = \prod_{j=1}^{d}(-j), \quad \delta(i,d) = \frac{i}{i-d-1}\delta(i-1,d) \quad i = 1, \ldots, d$$

$$\Delta(a,0,d) = \prod_{j=0}^{d}(a-j), \quad \Delta(a,k,d) = \frac{a+k}{a+k-d-1}\Delta(a,k-1,d) \quad k = 1, \ldots, d$$

- Let

$$\widetilde{P} = \sum_{i=0}^{d} \frac{P(i)}{\delta(i,d)} X^i, \quad S = \sum_{i=0}^{2d} \frac{1}{a+i-d} X^i, \quad Q = \widetilde{P}S.$$

- Return the sequence $\Delta(a,0,d)\cdot\texttt{coeff}(Q,d), \ldots, \Delta(a,d,d)\cdot\texttt{coeff}(Q,2d)$.

Figure 9.1: Shifting evaluation values

**Proposition 14** *Let $R$ be a commutative ring with unity, and $d \in \mathbb{N}$ such that $1, \ldots, d$ are units in $R$. Let $P$ be in $R[X]$ of degree $d$, such that the sequence*

$$P(0), \ldots, P(d)$$

*is known. Let $a$ be in $R$, such that $a-d, \ldots, a+d$ are units in $R$. Then the sequence*

$$P(a), \ldots, P(a+d)$$

*can be computed within $\mathsf{M}(2d) + O(d)$ base ring operations, using space $O(d)$.*

*Proof.* Our assumption on $R$ enables to write the Lagrange interpolation formula:

$$P = \sum_{i=0}^{d} P(i) \frac{\prod_{j=0, j\neq i}^{d}(X-j)}{\prod_{j=0, j\neq i}^{d}(i-j)}.$$

From now on, we denote by $\delta(i,d)$ the denominator $\prod_{j=0,j\neq i}^{d}(i-j)$ and by $\widetilde{P}_i$ the ratio $P(i)/\delta(i,d)$.

First note that all $\delta(i,d), i = 0, \ldots, d$, can be computed in $O(d)$ operations in $R$. Indeed, computing the first value $\delta(0,d) = \prod_{j=1}^{d}(-j)$ takes $d$ multiplications. Then for $i = 1, \ldots, d$, $\delta(i,d)$ can be deduced from $\delta(i-1,d)$ for two ring operations using the formula

$$\delta(i,d) = \frac{i}{i-d-1}\delta(i-1,d),$$

so their inductive computation requires $O(d)$ multiplications as well. Thus the sequence $\widetilde{P}_i, i = 0, \ldots, d$, can be computed in admissible time and space $O(d)$ from the input sequence $P(i)$. Accordingly, we rewrite the above formula as

$$P = \sum_{i=0}^{d} \widetilde{P}_i \prod_{j=0, j\neq i}^{d} (X - j).$$

For $k$ in $0, \ldots, d$, let us evaluate $P$ at $a + k$:

$$P(a+k) = \sum_{i=0}^{d} \widetilde{P}_i \prod_{j=0, j\neq i}^{d} (a + k - j).$$

Using our assumption on $a$, we can complete each product by the missing factor $a + k - i$:

$$P(a+k) = \sum_{i=0}^{d} \widetilde{P}_i \frac{\prod_{j=0}^{d}(a+k-j)}{a+k-i} = \left(\prod_{j=0}^{d}(a+k-j)\right) \cdot \left(\sum_{i=0}^{d} \widetilde{P}_i \frac{1}{a+k-i}\right). \tag{9.1}$$

Just as we introduced the sequence $\delta(i, d)$ above, we now introduce the sequence $\Delta(a, k, d)$ defined by $\Delta(a, k, d) = \prod_{j=0}^{d}(a + k - j)$. In a parallel manner, we deduce that all $\Delta(a, k, d), k = 0, \ldots, d$ can be computed in time and space $O(d)$, using the formulas:

$$\Delta(a, 0, d) = \prod_{j=0}^{d}(a - j), \quad \Delta(a, k, d) = \frac{a+k}{a+k-d-1}\Delta(a, k-1, d).$$

Let us denote $Q_k = P(a+k)/\Delta(a, k, d)$. We now show that knowing $\widetilde{P}_i, i = 0, \ldots, d$, we can compute $Q_k, k = 0, \ldots, d$ in $\mathsf{M}(2d)$ base ring operations and space $O(d)$; this is enough to conclude, by the above reasoning.

Using the coefficients $\Delta(a, k, d)$, Equation (9.1) reads

$$Q_k = \sum_{i=0}^{d} \widetilde{P}_i \frac{1}{a+k-i}. \tag{9.2}$$

Let $\widetilde{P}$ and $S$ be the polynomials:

$$\widetilde{P} = \sum_{i=0}^{d} \widetilde{P}_i X^i, \quad S = \sum_{i=0}^{2d} \frac{1}{a+i-d} X^i;$$

then by Equation (9.2), for $k = 0, \ldots, d$, $Q_k$ is the coefficient of degree $k + d$ in the product $\widetilde{P}S$. This concludes the proof. □

We will conclude this section by an immediate corollary of this proposition; we first give a few comments.

- An alternative $O(\mathsf{M}(d))$ algorithm which does *not* require any inversibility hypotheses can be designed in the special case when $a = d+1$. The key fact is that for any degree $d$ polynomial $P$, the sequence $P(0), P(1), \ldots$ is linearly recurrent, of characteristic polynomial $Q(X) = (1-X)^{d+1}$. Thus, if the first terms $P(0), \ldots, P(d)$ are known, the next $d+1$ terms $P(d+1), \ldots, P(2d+1)$ can be recovered in $O(\mathsf{M}(d))$ using the algorithm in [223, Theorem 3.1].

- The general case when the evaluation points form an arbitrary arithmetic progression reduces to the case treated in the above proposition. Indeed, suppose that $r_0, \ldots, r_d$ form an arithmetic progression of difference $\delta$, that $P(r_0), \ldots, P(r_d)$ are known and that we want to compute the values $P(r_0+a), \ldots, P(r_d+a)$, where $a \in R$ is divisible by $\delta$. Introducing the polynomial $Q(X) = P(\delta X + r_0)$, we are under the hypotheses of the above proposition, and it suffices to determine the shifted evaluation values of $Q$ by $a/\delta$.

- The reader may note the similarity of our problem with the question of computing the Taylor expansion of a given polynomial $P$ at a given point in $R$. The algorithm of [5] solves this question with a complexity of $\mathsf{M}(d)+O(d)$ operations in $R$ and space $O(d)$. The complexity results are thus quite similar; it turns out that analogous generating series techniques are used in that algorithm.

- In [115], an operation called *middle product* is defined: Given a ring $R$, and $A, B$ in $R[X]$ of respective degrees $d$ and $2d$, write $AB = C_0 + C_1 X^{d+1} + C_2 X^{2d+2}$, with all $C_i$ of degree at most $d$; then the middle product of $A$ and $B$ is the polynomial $C_1$. This is precisely what is needed in the above algorithm.

  Up to considering the reciprocal polynomial of $A$, the middle product by $A$ can be seen as the transpose of the map of multiplication by $A$. General program transformation techniques [47, 115] then show that it can be computed in time $\mathsf{M}(d)+O(d)$, but with a possible loss in space complexity. In [35], it is shown how to keep the same space complexity, at the cost of a constant increase in time complexity. Managing both requirements remains an open question, already stated in [124, Problem 6].

**Corollary 6** *Let $R$ be a commutative ring with unity, and $d \in \mathbb{N}$ such that $1, \ldots, 2d+1$ are units in $R$. Let $P$ be a degree $d$ polynomial in $R[X]$ such that the sequence*

$$P(0), \ldots, P(d)$$

*is known. For any $s$ in $\mathbb{N}$, the sequence*

$$P(0), P(2^s), \ldots, P(2^s d)$$

*can be computed in time $s\,\mathsf{M}(2d) + O(sd) \in O(s\,\mathsf{M}(d))$ and space $O(d)$.*

*Proof.* For any $s \in \mathbb{N}$, let us denote by $P_s(X)$ the polynomial $P(2^s X)$. We prove by induction that all values $P_s(0), \ldots, P_s(d)$ can be computed in time $s\,\mathsf{M}(2d) + O(sd)$ and space $O(d)$, which is enough to conclude. The case $s = 0$ is obvious, as there is nothing to compute. Suppose then that $P_s(0), \ldots, P_s(d)$ can be computed in time $s\,\mathsf{M}(2d) + O(sd)$ and using $O(d)$ temporary space allocation.

Under our assumption on $R$, Proposition 14 shows that the values $P_s(d+1), \ldots, P_s(2d+1)$ can be computed in time $\mathsf{M}(2d) + O(d)$, using again $O(d)$ temporary space allocation. The values $P_s(0), P_s(2), \ldots, P_s(2d)$ coincide with $P_{s+1}(0), P_{s+1}(1), \ldots, P_{s+1}(d)$, so the corollary is proved. □

## 9.3 Computing one selected term of a linear sequence

In this section, we recall and improve the complexity of an algorithm due to Chudnovsky and Chudnovsky [59] for computing selected terms of linear recurrent sequences with polynomial coefficients. The results of the previous section are used as a basic subroutine for these questions.

As in the previous section, $R$ is a commutative ring with unity. Let $A$ be a $n \times n$ matrix of polynomials in $R[X]$. For simplicity, in what follows, we only treat the case of degree 1 polynomials, since this is what is needed in the sequel. Nevertheless, all results extend *mutatis mutandis* to arbitrary degree.

For $r$ in $R$, we denote by $A(r)$ the matrix over $R$ obtained by specializing all coefficients of $A$ at $r$. In particular, for $k$ in $\mathbb{N}$, $A(k \cdot 1_R)$ is simply denoted by $A(k)$, following the convention used up to now. Given a vector of initial conditions $U_0 = [u_1, \ldots, u_n]^t \in R^n$ and given $k$ in $\mathbb{N}$, we consider the question of computing the $k$th term of the linear sequence defined by the relation $U_i = A(i)U_{i-1}$ for $i > 0$, that is, the product

$$U_k = A(k)A(k-1)\cdots A(1)U_0.$$

For simplicity, we write

$$U_k = \left(\prod_{i=1}^{k} A(i)\right) U_0,$$

performing all successive matrix products, $i = 1, \ldots, k$, on the left side. We use this convention hereafter.

In the particular case when $A$ is a matrix of constant polynomials, and taking only the dependence on $k$ into account, the binary powering method gives a time complexity of order $O(\log(k))$ base ring operations.

In the general case, the naive solution consists in evaluating all matrices $A(i)$ and performing all products. With respect to $k$ only, the complexity of this approach is of order $O(k)$ base ring operations. In [59], Chudnovsky and Chudnovsky propose an algorithm that reduces this cost to essentially $O\big(\sqrt{k}\big)$. We first recall the main lines of this algorithm; we then present some improvements in both time and space complexities.

**The algorithm of Chudnovsky and Chudnovsky.**

The original algorithm uses baby-step / giant-step techniques, so for simplicity we assume that $k$ is a square in $\mathbb{N}$. Let $C$ be the $n \times n$ matrix over $R[X]$ defined by

$$C = \prod_{i=1}^{\sqrt{k}} A(X+i),$$

where $A(X+i)$ denotes the matrix $A$ with all polynomials evaluated at $X+i$. By assumption on $A$, the entries of $C$ have degree at most $\sqrt{k}$. For $r$ in $R$, we denote by $C(r)$ the matrix $C$ with all entries evaluated at $r$. Then the requested output $U_k$ can be obtained by the equation

$$U_k = \left( \prod_{j=0}^{\sqrt{k}-1} C(j\sqrt{k}) \right) U_0. \tag{9.3}$$

Here are the main steps of the algorithm underlying Equation (9.3), originally due to [59].

*Baby steps.* The "baby steps" part of the algorithm consists in computing the polynomial matrix $C$. In [59], this is done within $O(n^\omega \mathsf{M}(\sqrt{k}))$ base ring operations, as products of polynomial matrices with entries of degree $O(\sqrt{k})$ are required.

*Giant steps.* In the second part the matrix $C$ is evaluated on the arithmetic progression $0, \sqrt{k}, 2\sqrt{k}, \ldots, (\sqrt{k}-1)\sqrt{k}$ and the value of $U_k$ is obtained using Equation (9.3). Using fast evaluation techniques, all evaluations are done within $O(n^2\,\mathsf{M}(\sqrt{k})\log(k))$ base ring operations, while performing the $\sqrt{k}$ successive matrix-vector products in Equation (9.3) adds a negligible cost of $O(n^2\sqrt{k})$ operations in $R$.

Summing all the above costs gives an overall complexity bound of

$$O\big(n^\omega \mathsf{M}\big(\sqrt{k}\big) + n^2\,\mathsf{M}\big(\sqrt{k}\big)\log(k)\big)$$

base ring operations for computing a selected term of a linear sequence. Due to the use of fast evaluation algorithms in degree $\sqrt{k}$, the space complexity is $O(n^2\sqrt{k} + \sqrt{k}\log(k))$.
In the particular case when $A$ is the $1\times 1$ matrix $[X]$, the question reduces to the computation of $\prod_{j=1}^{k} j$ in the ring $R$. For this specific problem, note that the ideas presented above were already used in [195, 237], for the purpose of factoring integers.

**Avoiding multiplications of polynomial matrices.**

In what follows, we show how to avoid the multiplication of polynomial matrices, and reduce the cost of the above algorithm to $O(n^\omega\sqrt{k} + n^2\,\mathsf{M}(\sqrt{k})\log(k))$ base ring operations, storing only $O(n^2\sqrt{k})$ elements of $R$.

Our improvements are obtained through a modification of the baby steps phase; the underlying idea is to work with the *values* taken by the polynomial matrices instead of their representation on the monomial basis. This idea is encapsulated in the following proposition.

**Proposition 15** *Let $A$ be a $n \times n$ matrix with entries in $R[X]$, of degree at most 1. Let $N \geq 1$ be an integer and let $C$ be the $n \times n$ matrix over $R[X]$ defined by*

$$C = \prod_{i=1}^{N} A(X+i).$$

*Then one can compute all scalar matrices $C(0), C(1), \ldots, C(N)$ within $O(n^\omega N)$ operations in $R$ and with a memory requirement of $O(n^2 N)$ elements in $R$.*

*Proof.* We first compute the scalar matrices $[A(1), A(2), \ldots, A(2N)]$. Since all entries of $A$ are linear in $X$, the complexity of this preliminary step is $O(n^2 N)$, both in time and space. Then, we construct the matrices $(C'_j)_{0 \leq j \leq N}$ and $(C''_j)_{0 \leq j \leq N}$, which are defined as follows: we let $C'_0$ and $C''_0$ equal the identity matrix $I_n$ and we recursively define

$$\begin{array}{rcll} C'_j & = & A(N+j)C'_{j-1} & \text{for} \quad 1 \leq j \leq N, \\ C''_j & = & C''_{j-1}A(N-j+1) & \text{for} \quad 1 \leq j \leq N. \end{array}$$

Explicitly, for $0 \leq j \leq N$, we have

$$C'_j = A(N+j) \cdots A(N+1)$$

and

$$C''_j = A(N) \cdots A(N-j+1),$$

thus

$$C''_{N-j} = A(N) \cdots A(j+1).$$

Computing all the scalar matrices $(C'_j)$ and $(C''_j)$ requires $2N$ matrix multiplications with entries in $R$; their cost is bounded by $O(n^\omega N)$ in time and by $O(n^2 N)$ in space. Lastly, the formula

$$C(j) = A(N+j) \cdots A(N+1)A(N) \cdots A(j+1) = C'_j C''_{N-j}, \quad 0 \leq j \leq N$$

enables to recover $C(0), C(1), \ldots, C(N)$ in time $O(n^\omega N)$ and space $O(n^2 N)$. □

From this proposition, we deduce the following corollary, which shows how to compute the scalar matrices used in the giant steps.

**Corollary 7** *Let $A$ and $C$ be polynomial matrices as in Proposition 15. If the elements $1, \ldots, 2N+1$ are units in $R$, then for any integer $s \geq 1$, the sequence*

$$C(0), C(2^s), \ldots, C(2^s(N-1))$$

*can be computed using $O(n^\omega N + n^2 s\, \mathsf{M}(N))$ operations in $R$ and $O(n^2 N)$ memory space.*

*Proof.* This is an immediate consequence of Proposition 15 and Corollary 6. □

The above corollary enables us to perform the "giant steps" phase of Chudnovsky and Chudnovsky's algorithm in the special case when $N = 2^s$; this yields the $4^s$th term in the recurrent sequence. Using this intermediate result, the following theorem shows how to compute the $k$th term, for arbitrary $k$, using the 4-adic expansion of $k$.

**Theorem 12** *Let $A$ be a $n \times n$ matrix with linear entries in $R[X]$ and let $U_0$ be in $R^n$. Suppose that $(U_i)$ is the sequence of elements in $R^n$ defined by the linear recurrence*

$$U_{i+1} = A(i+1)U_i, \quad \text{for all } i \geq 0.$$

*Let $k > 0$ be an integer and suppose that $1, \ldots, 2\lceil\sqrt{k}\rceil + 1$ are units in $R$. Then the vector $U_k$ can be computed within $O\big(n^\omega\sqrt{k} + n^2\,\mathsf{M}\big(\sqrt{k}\big)\log(k)\big)$ operations in $R$ and using memory space $O(n^2\sqrt{k})$.*

The proof of Theorem 12 is divided in two steps. We begin by proving the proposition in the particular case when $k$ is a power of 4, then we treat the general case.

**The case $k$ is a power of 4.**

Let us suppose that $N = 2^s$ and $k = N^2$, so that $k = 4^s$. With this choice of $k$, Corollary 7 shows that the values $C(0), C(N), \ldots, C((N-1)N)$ can be computed within the required time and space complexities. Then we go on to the giant step phase described at the beginning of the section, and summarized in Equation (9.3). It consists in performing $\sqrt{k}$ successive matrix-vector products, which has a cost in both time and space of $O(n^2\sqrt{k})$.

**The general case.**

We now consider the general case. Let $k = \sum_{i=0}^{s} k_i 4^i$ be the 4-adic expansion of $k$, with $k_i \in \{0, 1, 2, 3\}$ for all $i$. Given any $t$, we will denote by $\lceil k \rceil^t$ the integer $\sum_{i=0}^{t-1} 4^i k_i$. Using this notation, we define a sequence $(V_t)_{0 \leq t \leq s}$ as follows: we let $V_0 = U_0$ and, for $0 \leq t \leq s$ we set

$$V_{t+1} = A(\lceil k \rceil^t + 4^t k_t) \cdots A(\lceil k \rceil^t + 1)V_t. \tag{9.4}$$

It is easy to verify that $V_{s+1} = U_k$. Therefore, it suffices to compute the sequence $(V_t)$ within the desired complexities.

Supposing that the term $V_t$ has been determined, we estimate the cost of computing the next term $V_{t+1}$. If $k_t$ is zero, we have nothing to do. Otherwise, we let $V_{t+1}^{(0)} = V_t$, and, for $1 \leq j \leq k_t$, we let $A^{(j)}(X) = A\left(X + \lceil k \rceil^t + 4^t(j-1)\right)$. Then we define $V_{t+1}^{(j)}$ by

$$V_{t+1}^{(j)} = A^{(j)}(4^t) \cdots A^{(j)}(1) V_{t+1}^{(j-1)}, \quad j = 1, \ldots, k_t.$$

By Equation (9.4), we have $V_{t+1}^{(k_t)} = V_{t+1}$. Thus, passing from $V_t$ to $V_{t+1}$ amounts to computing $k_t$ selected terms of a linear recurrence of the special form treated in the previous paragraph. Using the complexity result therein and the fact that all $k_t$ are bounded by 3, the total cost of the general case is thus

$$O\left(\sum_{t=0}^{s}\left(n^\omega 2^t + n^2 t\,\mathsf{M}(2^t)\right)\right) = O\left(n^\omega 2^s + n^2 s\left(\sum_{t=0}^{s}\mathsf{M}(2^t)\right)\right).$$

Using the fact that $2^s \leq \sqrt{k} \leq 2^{s+1}$ and the assumptions on the function $\mathsf{M}$, we easily deduce that the whole complexity fits into the bound $O\big(n^\omega\sqrt{k} + n^2\,\mathsf{M}\big(\sqrt{k}\big)\log(k)\big)$, as claimed. Similar considerations also yield the bound concerning the memory requirements. This concludes the proof of Theorem 12.

**Comments.**

The question of a lower time bound for computing $U_k$ is still open. The simpler question of reducing the cost to $O\big(n^\omega\sqrt{k}+n^2\,\mathsf{M}\big(\sqrt{k}\big)\big)$ base ring operations, that is gaining a logarithmic factor, already raises challenging problems.

As the above paragraphs reveal, this improvement could be obtained by answering the following question: Let $P$ be a polynomial of degree $d$ in $R[X]$. Given $r$ in $R$, how fast can we compute $P(0), P(r), \ldots, P(rd)$ from the data of $P(0), P(1), \ldots, P(d)$? A complexity of order $O(\mathsf{M}(d))$ would immediately give the improved bound mentioned above. We leave it as an open question.

## 9.4 The Cartier-Manin operator on hyperelliptic curves

We finally show how to apply the above results to the computation of the Cartier-Manin operator, and start by reviewing some known facts on this operator.

Let $\mathcal{C}$ be a hyperelliptic curve of genus $g$ defined over the finite field $\mathbb{F}_{p^d}$ with $p^d$ elements, where $p$ is the characteristic of $\mathbb{F}_{p^d}$. We suppose that $p > 2$ and that the equation of $\mathcal{C}$ is of the form $y^2 = f(x)$, where $f \in \mathbb{F}_{p^d}[X]$ is a monic squarefree polynomial of degree $2g+1$. The generalization to hyperelliptic curves of the Hasse invariant for elliptic curves is the so-called Hasse-Witt matrix, which is defined as follows:

**Definition 2** *Let $h_k$ be the coefficient of degree $k$ in the polynomial $f^{(p-1)/2}$. The Hasse-Witt matrix is the $g \times g$ matrix with coefficients in $\mathbb{F}_{p^d}$ given by*

$$H = (h_{ip-j})_{1 \le i,j \le g}.$$

This matrix was introduced in [116]; in a suitable basis, it represents the operator on differential forms that was introduced by Cartier in [52]. Manin then showed in [163] that this matrix is strongly related to the action of the Frobenius endomorphism on the $p$-torsion part of the Jacobian of $\mathcal{C}$. The article [264] provides a complete survey about those facts; they can be summarized by the following theorem:

**Theorem 13 (Manin)** *Let $\mathcal{C}$ be a hyperelliptic curve of genus $g$ defined over $\mathbb{F}_{p^d}$. Let $H$ be the Hasse-Witt matrix of $\mathcal{C}$ and let $H_\pi = HH^{(p)} \cdots H^{(p^{d-1})}$, where the notation $H^{(q)}$ means element-wise raising to the power $q$. Let $\kappa(t)$ be the characteristic polynomial of the matrix $H_\pi$ and let $\chi(t)$ be the characteristic polynomial of the Frobenius endomorphism of the Jacobian of $\mathcal{C}$. Then*

$$\chi(t) \equiv (-1)^g t^g \kappa(t) \mod p.$$

This result provides a quick method to compute the characteristic polynomial of the Frobenius endomorphism and hence the group order of the Jacobian of $\mathcal{C}$ modulo $p$, when $p$ is not too large. Combined with a Schoof-like algorithm and / or a baby-step / giant-step

algorithm, it can lead to a full point-counting algorithm, in particular for genus 2 curves, as was demonstrated in [88, 166].

The obvious solution consists in expanding the product $f^{(p-1)/2}$. Using balanced multiplications, and taking all products modulo $X^{gp}$ this can be done in $O(\mathsf{M}(gp))$ base field operations, whence a time complexity within $O(\mathsf{M}(p))$, if $g$ is kept constant. In what follows, regarding the dependence in $p$ only, we show how to obtain a complexity of $O(\mathsf{M}(\sqrt{p})\log(p))$ base field operations, using the results of the previous sections.

We will make the assumption that the constant term of $f$ is not zero. Note that if it is zero, the problem is actually simpler: writing $f = Xf_1$, the coefficient of degree $ip-j$ in $f^{(p-1)/2}$ is the coefficient of degree $ip-j-(p-1)/2$ in $f_1^{(p-1)/2}$. Hence we can work with a polynomial of degree $2g$ instead of $2g+1$ and the required degrees are slightly less.

Furthermore, for technical reasons, we assume that $g < p$. This is not a true restriction since for $g \geq p$, all the coefficients of $f^{(p-1)/2}$ up to degree $g(p-1)$ are needed to fill in the matrix $H$.

**Introduction of a linear recurrent sequence.**

In [85], Flajolet and Salvy already treat the question of computing a selected coefficient in a high power of some given polynomial, as an answer to a SIGSAM challenge. The key point of their approach is that $h = f^{(p-1)/2}$ satisfies the following first-order linear differential equation

$$fh' - \frac{p-1}{2} f'h = 0.$$

From this, we deduce that the coefficients of $h$ satisfy a linear recurrence of order $2g+1$, with coefficients that are rational functions of degree 1.

Explicitly, let us denote by $h_k$ the coefficient of degree $k$ of the polynomial $h$, and for convenience, set $h_k = 0$ for $k < 0$. Similarly, the coefficient of degree $k$ of $f$ is denoted by $f_k$. From the above differential equation, for all $k$ in $\mathbb{Z}$, we deduce that

$$\sum_{i=0}^{2g+1} \left( k+1 - \frac{(p+1)i}{2} \right) f_i \, h_{k+1-i} \; = \; 0.$$

We set $U_k = [h_{k-2g}, h_{k-2g+1}, \ldots, h_k]^t$, and let $A(k)$ be the $(2g+1) \times (2g+1)$ companion matrix:

$$A(k) = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & 1 \\ \frac{((2g+1)(p+1)/2-k)f_{2g+1}}{kf_0} & \cdots & \cdots & \cdots & \frac{((p+1)/2-k)f_1}{kf_0} \end{bmatrix}.$$

The initial vector $U_0 = [0, \ldots, 0, f_0^{(p-1)/2}]^t$ can be computed using binary powering techniques in $O(\log(p))$ base field operations; then for $k \geq 0$, we have $U_{k+1} = A(k+1)U_k$. Thus, to

answer our specific question, it suffices to note that the vector $U_{ip-j}$ gives the coefficients $h_{ip-j}$ for $j = 1, \ldots, g$ that form the $i$th row of the Hasse-Witt matrix of $\mathcal{C}$.

Yet, Theorem 12 cannot be directly applied to this sequence, because $A(k)$ has entries that are rational functions, not polynomials. Though the algorithm could be adapted to handle the case of rational functions, we rather use the very specific form of the matrix $A(k)$, so only a small modification is necessary. Let us define a new sequence $V_k$ by the relation

$$V_k = f_0^k k! U_k.$$

Then, this sequence is linearly generated and we have $V_{k+1} = B(k+1)V_k$, where

$$B(k) = f_0 k A(k).$$

Therefore, the entries of the matrix $B(k)$ are polynomials of degree at most 1. Note also that the denominators $f_0^k k!$ satisfy the recurrence relation

$$f_0^{k+1}(k+1)! = (f_0(k+1)) \cdot (f_0^k k!).$$

Thus, we will compute separately, first $V_{p-1}, V_{2p-1}, \ldots, V_{gp-1}$ and then the denominators $f_0^{p-1}(p-1)!, \ldots, f_0^{gp-1}(gp-1)!$.

To this effect, we proceed iteratively. Let us for instance detail the computation of the sequence $V_{p-1}, V_{2p-1}, \ldots, V_{gp-1}$. Knowing $V_0$, we compute $V_{p-1}$ using Theorem 12. Then we shift all entries of $B$ by $p$, so another application of Theorem 12 yields $V_{2p-1}$. Iterating $g$ times, we obtain $V_{p-1}, V_{2p-1}, \ldots, V_{gp-1}$ as requested; the same techniques are used to compute $f_0^{p-1}(p-1)!, \ldots, f_0^{gp-1}(gp-1)!$. Then the vectors $U_{p-1}, U_{2p-1}, \ldots, U_{gp-1}$ are deduced from

$$U_k = \frac{1}{f_0^k k!} V_k.$$

**Lifting to characteristic zero.**

A difficulty arises from the fact that the characteristic is too small compared to the degrees we are aiming to, so $p!$ is zero in $\mathbb{F}_{p^d}$. The workaround is to do computations in the unramified extension $K$ of $\mathbb{Q}_p$ of degree $d$, whose residue class field is $\mathbb{F}_{p^d}$. The ring of integers of $K$ will be denoted by $O_K$; any element of $O_K$ can be reduced modulo $p$ to give an element of $\mathbb{F}_{p^d}$. On the other hand, $K$ has characteristic 0, so $p$ is invertible in $K$.

We consider an arbitrary lift of $f$ to $O_K[X]$. The reformulation in terms of linear recurrent sequence made in the above paragraph can be performed over $K$; the coefficients of $f^{(p-1)/2}$ are computed as elements of $K$ and then projected back onto $\mathbb{F}_{p^d}$. This is possible, as they all belong to $O_K$.

Using the iteration described above, we separately compute the values in $K$ of the vectors $V_{ip-1}$ and the denominators $f_0^{ip-1}(ip-1)!$, for $i = 1, \ldots, g$. To this effect, we apply $g$ times the result given in Theorem 12; this requires to perform

$$O\left(g^{\omega+1}\sqrt{p} + g^3 \mathsf{M}(\sqrt{p}) \log(p)\right),$$

operations in $K$ and to store $O(g^2\sqrt{p})$ elements of $K$.

**Computing at fixed precision.**

Of course, we do not want to compute in the field $K$ at arbitrary precision: for our purposes, it suffices to truncate all computations modulo a suitable power of $p$. To evaluate the required precision of the computation, we need to check when the algorithm operates a division by $p$.

To compute the vectors $V_{ip-1}$ and the denominators $f_0^{ip-1}(ip-1)!$, for $i = 1, \ldots, g$, we use Theorem 12. This requires that all integers up to $2\lceil\sqrt{p}\rceil + 1$ are invertible, which holds as soon as $p \geq 11$.

Then, for all $i = 1, \ldots, g$, to deduce $U_{ip-1}$ from $V_{ip-1}$, we need to divide by $f_0^{ip-1}(ip-1)!$. The element $f_0$ is a unit in $O_K$, so the only problem comes from the factorial term. With our assumption that $g < p$, we have $i < p$ and then the $p$-adic valuation of $(ip-1)!$ is exactly $i-1$. Therefore the worst case is $i = g$, for which we have to divide by $p^{g-1}$. Hence computing the vectors $V_{ip-1}$ modulo $p^g$ is enough to know the vectors $U_{ip-1}$ modulo $p$, and then to deduce the Hasse-Witt matrix.

**Overall complexity.**

Storing an element of $O_K/p^gO_K$ requires $O(dg\log(p))$ bits, and multiplying two such elements can be done with $O(\mathsf{M}(dg\log(p)))$ bit-operations. From the results of Section 9.3, we then deduce the following theorem on the complexity of computing the Hasse-Witt matrix.

**Theorem 14** *Let $p$ a prime, $d \geq 1$ and $\mathcal{C}$ a hyperelliptic curve defined over $\mathbb{F}_{p^d}$ by the equation $y^2 = f(x)$, with $f$ of degree $2g+1$. Then, assuming $g < p$, one can compute the Hasse-Witt matrix of $\mathcal{C}$ with a complexity of*

$$O\left(\left(g^{\omega+1}\sqrt{p} + g^3\mathsf{M}(\sqrt{p})\log(p)\right)\ \mathsf{M}(dg\log(p))\right)$$

*bit-operations and $O\left(dg^3\sqrt{p}\ \log(p)\right)$ storage.*

The matrix $H$ by itself gives some information on the curve $\mathcal{C}$, for instance $H$ is invertible if and only if the Jacobian of $\mathcal{C}$ is ordinary [264, Corollary 2.3]. However, as stated in Theorem 13, the matrix $H_\pi$ and in particular its characteristic polynomial $\chi(t)$ tell much more and are required if the final goal is point-counting. Thus, we finally concentrate on the cost of computing the characteristic polynomial of $H_\pi$.

The matrix $H_\pi$ is the "norm" of $H$ and as such can be computed with a binary powering algorithm. For simplicity, we assume that $d$ is a power of 2, then denoting

$$H_{\pi,i} = HH^{(p)} \cdots H^{\ p^{2^i-1}}\ .$$

we have

$$H_{\pi,i+1} = H_{\pi,i} \cdot (H_{\pi,i})^{\ p^{2^i}}\ .$$

Hence the computation of $H_{\pi,i+1}$ from $H_{\pi,i}$ costs one matrix multiplication and $2^i$ matrix conjugations. A matrix conjugation consists in raising all the entries to the power $p$, therefore

it costs $O(g^2 \log(p))$ operations in $\mathbb{F}_{p^d}$. The matrix we need to compute is $H_\pi = H_{\pi,\log_2(d)}$. Hence the cost of computing $H_\pi$ is

$$O\left(dg^2 \log(p) + g^\omega \log(d)\right)$$

operations in $\mathbb{F}_{p^d}$. The general case where $d$ is not a power of 2 is handled by adjusting the recursive step according to the binary expansion of $d$ and yields the same complexity up to a constant factor.

The cost of the characteristic polynomial computation is bounded by the cost of a matrix multiplication [132] and is therefore negligible compared to the other costs.

If we are interested only in the complexity in $p$ and $d$, *i.e.* if we assume that the genus is fixed, we get a time complexity for computing $\chi(t) \mod p$ in

$$O\left((\mathsf{M}(\sqrt{p}) + d)\, \mathsf{M}(d \log(p)) \log(p)\right).$$

**Case of large genus.**

In case of large genus, the algorithm of Theorem 12 is asymptotically not the fastest. In this paragraph, we assume that the function $\mathsf{M}$ is essentially linear and we do not take into account the logarithmic factors; adding appropriate epsilons in the exponents would yield a rigorous analysis. The cost in bit-operations of Theorem 14 is at least $g^4\sqrt{p}d$ whereas the cost of the naive algorithm is linear in $gpd$. If $g > p^{1/6}$, then $g^4\sqrt{p} > gp$, and therefore the naive algorithm is faster.

## 9.5 Point-counting numerical example

We have implemented our algorithm using Shoup's NTL C++ library [226]. NTL does not provide any arithmetic of local fields or rings, but allows to work in finite extensions of rings of the form $\mathbb{Z}/p^g\mathbb{Z}$, as long as no division by $p$ occur; the divisions by $p$ are well isolated in the algorithm, so we could handle them separately. Furthermore, NTL multiplies polynomials defined over this kind of structure using an asymptotically fast FFT-based algorithm.

To illustrate that our method can be used as a tool in point-counting algorithms, we have computed the Zeta function of a (randomly chosen) genus 2 curve defined over $\mathbb{F}_{p^3}$, with $p = 2^{32} - 5$. Such a Jacobian has therefore about $2^{192}$ elements and should be suitable for cryptographic use if the group order has a large prime factor. Note that previous computations were limited to $p$ of order $2^{23}$ [166].

The characteristic polynomial $\chi$ of the Frobenius endomorphism was computed modulo $p$ in 3 hours and 41 minutes, using 1 GB of memory, on an AMD Athlon MP 2200+. Then we used the Schoof-like algorithms of [88] and [89] to compute $\chi$ modulo $128 \times 9 \times 5 \times 7$, and finally we used the modified baby-step / giant-step algorithm of [166] to finish the computation. These other parts were implemented in Magma [32] and were performed in about 15 days of computation on an Alpha EV67 at 667 MHz. We stress that this computation was meant as an illustration of the possible use of our method, so little time was spent optimizing our code. In particular, the Schoof-like part and the final baby-step / giant-step computations are done using a generic code that is not optimized for extension fields.

**Numerical data.**

The irreducible polynomial $P(t)$ that was used to define $\mathbb{F}_{p^3}$ as $\mathbb{F}_p[t]/(P(t))$ is

$$t^3 + 1346614179t^2 + 3515519304t + 3426487663.$$

The curve $\mathcal{C}$ has equation $y^2 = f(x)$ where $f$ is given by

$$\begin{aligned} f(x) \;=\; & x^5 + (2697017539t^2 + 1482222818t + 3214703725)x^3 + \\ & (676673546t^2 + 3607548185t + 1833957986)x^2 + \\ & (1596634951t^2 + 3203023469t + 2440208439)x + \\ & 2994361233t^2 + 3327339023t + 862341251. \end{aligned}$$

Then the polynomial characteristic $\chi(T)$ of the Frobenius endomorphism is given by $T^4 - s_1T^3 + s_2T^2 - p^3s_1T + p^6$, where

$$s_1 = 332906835893875, \quad s_2 = 142011235215638946167187570235.$$

The group order of the Jacobian is then

$$\begin{aligned} & 627710169154160539591778508077182588386018946581362599397 7 \\ = \;\; & 3^3 \times 13 \times 67 \times 639679 \times 41726806872753637081001017234423602545593395313 9. \end{aligned}$$

This number has a large prime factor of size $2^{158}$, therefore that curve is cryptographically secure.

**Measure of the complexity in $p$.**

To check the practical asymptotic behaviour of our algorithm, we ran our implementation on a genus 2 curve defined over $\mathbb{F}_{p^3}$ with $p = 2^{34} - 41$. We performed only the Cartier-Manin step, and not the full point-counting algorithm. As the characteristic is about 4 times larger than in the previous example, a complexity linear in $\sqrt{p}$ means a runtime multiplied by about 2. On the same computer, the runtime is 8 hours and 48 minutes. Hence the ratio of the runtimes is about 2.39. The defect of linearity can be explained by taking into account the logarithmic factors. Assuming that $\mathsf{M}(n)$ is $O(n\log(n)\log(\log(n)))$, and neglecting the multi-logarithmic factors, the complexity announced in Theorem 14 is in $O(\sqrt{p}(\log(p))^3)$. With this estimate, the expected ratio between the runtimes becomes about 2.40, that is very close to the measure. This validates our analysis.

## 9.6 Conclusion

In this chapter, we have presented an improvement of an algorithm by Chudnovsky and Chudnovsky to compute selected terms in a linear sequence with polynomial coefficients. This algorithm is then applied to the computation of the Cartier-Manin operator of hyperelliptic curves, thus leading to improvements in the point-counting problems that occur in cryptography.

This strategy extends readily to curves of the form $y^r = f(x)$ with $r > 2$, for which the Hasse-Witt matrix has a similar form. For more general curves, Mike Zieve pointed to us the work of Stöhr and Voloch [231] that gives formulas that still fit in our context in some cases.

Finally, Mike Zieve pointed out to us the work of Wan [257] that relates Niederreiter's polynomial factorization algorithm to the computation of the Cartier-Manin operator of some variety. The link with our work is not immediate, as that variety has dimension zero. Nevertheless, this remains intriguing, especially if we think of Pollard-Strassen's integer factoring algorithm as a particular case of Chudnovsky and Chudnovsky's algorithm.

# Chapter 10

# Fast Algorithms for Linear Differential Operators

In this chapter we address the problem of fast computation with linear differential operators. We propose a unified strategy, which can be viewed as the non-commutative analogue of semi-numerical computations with algebraic numbers based on LLL [153] algorithm. Our methods rely on a change of representation: instead of dealing with coefficients of differential operators, we work with their (local) power series solutions. This allows to treat various operations on linear differential operators including least common left multiples and tensor products. This chapter is an extended abstract of an article in preparation.

## Contents

## 10.1 Introduction

Suppose that we are given two linear differential operators $L_1$ and $L_2$, and that we want to compute $L_1 \star L_2$, where $\star$ is a certain *linear algebra operation*, defined in terms of (local) solution spaces of $L_1$ and $L_2$. Classical examples of such operations are the least common (left) multiple $L_1 \oplus L_2$ and the tensor product $L_1 \otimes L_2$; these are the linear differential operators whose solution spaces are spanned by the sums $y_1+y_2$, respectively by the products $y_1y_2$, of solutions of $L_1y_1 = 0$ and $L_2y_2 = 0$.

**Specificity of our approach.** The aim of this chapter is a feasibility study of a new method for computing $L = L_1 \star L_2$, which extends the usual *evaluation / interpolation* strategy to a non-commutative framework. The rôle of points of evaluation is played by formal power series, on which differential operators are evaluated.

To compute $L = L_1 \star L_2$, our *generic* algorithm consists in performing the following stages:

1. compute truncated power series solutions $S_1$ and $S_2$ of both input operators $L_1$ and $L_2$;
2. combine $S_1$ and $S_2$ by means of the defining operation $\star$ and deduce a (truncated) solution $S$ of the output operator $L$;
3. reconstruct the operator $L$ from its truncated power series solution $S$.

**Similarity with computations with algebraic numbers.** One should notice the similarity between this method and the classical semi-numerical approach for computing with algebraic numbers, which uses the lattice reduction algorithm LLL [153]. If $\alpha$ and $\beta$ are two algebraic numbers represented by their minimal polynomials $m_\alpha$ and $m_\beta$, the LLL-based method computes the minimal polynomial of $\alpha + \beta$ as follows: first determine (sufficiently accurate) numerical approximations $\widetilde{\alpha}$ of $\alpha$ and $\widetilde{\beta}$ of $\beta$, then form $\widetilde{\gamma} = \widetilde{\alpha} + \widetilde{\beta}$ and search for integers $\ell_0, \ell_1, \ldots$, such that $\sum_i \ell_i \widetilde{\gamma}^i$ be *small enough*.

In our present differential setting, (floating point) real numbers are replaced by (truncated) power series (with exact coefficients), integers $\ell_i$ by polynomials and numerical approximations of real numbers by truncation of power series.

**Linear operators with constant coefficients.** This similarity is not fortuitous, since operations on algebraic numbers can be viewed as operations on differential operators with constant coefficients; for instance, the sum of two algebraic numbers corresponds to lclm of differential operators with constant coefficients.

Indeed, if the operators are represented by polynomials $P$ and $Q$, then $\{\exp(\alpha X)\}_{P(\alpha)=0}$ and $\{\exp(\beta X)\}_{Q(\beta)=0}$ are bases of their solution spaces, so $\{\exp((\alpha + \beta)X)\}_{\alpha,\beta}$ is a basis of the tensor product of $P$ and $Q$, therefore the polynomial representing it is the minimal polynomial of $\alpha + \beta$. Thus, our results in Chapter 7 yield nearly optimal algorithms for the tensor product of operators with constant coefficients.

**Efficiency issues.** Stages (1) and (3) perform conversions between both representations of differential operators. In stage (1), the input is the monomial representation of a differential operator, while the output is its representation in terms of (local) solution spaces. Stage (3) performs the converse direction: on input one or several truncated power series, it computes the differential operator of smallest order that admits these power series as solutions.

Thus, the efficiency of the method depends on the ability of doing these conversions fast. In the case of algebraic numbers, the equivalent of stage (1) is performed in a nearly-optimal fashion using a Newton iterator. In our more general case, we are still able to perform nearly optimally the conversion in stage (1): given an operator $L$ with polynomial coefficients, we show that the first $\sigma$ terms of the power solution $S$ at an ordinary point can be computed using $O_{\log}(\sigma)$ operations.

In contrast, the fastest solution for stage (3) uses LLL algorithm, whose complexity is not optimal. Similar difficulties occur for the operator reconstruction stage (3), which may be viewed as a computation of *differential Padé-Hermite approximants*. Indeed, the coefficients of the output operator $\mathcal{L}$ are given by a Padé-Hermite approximant of $(y, y', \ldots, y^{(D)})$ of type $(N, N, \ldots, N)$, where $D$ is the order of $\mathcal{L}$ and $N$ is a bound on the degrees of the coefficients of $\mathcal{L}$.

**Bounds on the results.** This suggests that the a priori knowledge of tight bounds on the degree of the coefficients of $\mathcal{L}$ in terms of similar bounds on the input operators should contribute to improve efficiency of our approach.

Since we consider only *linear algebra constructions* on differential operators with polynomial coefficients, such bounds are available by means of Cramer's rule. However, due to analytic phenomena like the appearance of *apparent singularities* during such constructions, the bounds provided by Cramer's rule are not tight. For this reason, different methods need to be designed in order to yield more realistic bounds. We exemplify two methods based respectively on differential resultants and (generalized) Fuchs relation, in the case of lclm and tensor power.

## 10.2 From differential operators to power series solutions

Let $\mathcal{P}$ be a linear differential operator of order $n$, with polynomial coefficients $p_i(x) = \sum_{j=0}^{m} p_{ij} x^j$ of degree at most $m$. We suppose for simplicity that $x = 0$ is an ordinary point for $\mathcal{P}$, that is $p_n(0) \neq 0$. Performing a generic translation of the coefficients of $\mathcal{P}$ ensures that this hypothesis can be taken for granted.

We want to compute the first $\sigma$ coefficients of a power series solution $S = \sum_{i \geq 0} s_i x^i$ of $\mathcal{P}$ at $x = 0$, with given initial conditions. We propose a method which makes use of fast polynomial evaluation methods and which is faster by an order of magnitude than the classical algorithm (obtained by identification of coefficients) which uses the one-to-one correspondence $x^k S(x) \leftrightarrow s_{i-k}$, $xS'(x) \leftrightarrow i s_i$.

The sequence $(s_i)$ satisfies a linear recurrence of order at most $m+n$, whose coefficients are polynomials in $i$ of degree at most $n$ [1]. This recurrence writes

$$\widetilde{p_0}(i)s_i + \cdots + \widetilde{p_{n+m}}(i)s_{i-m-n} = 0,$$

where $\widetilde{p_i}$ are polynomials of degree at most $n$.

Determining the coefficients of the polynomials $\widetilde{p_i}$ amounts to expressing $\mathcal{P}$ in terms of the Euler operator $\delta = xD$. This can be done efficiently using the method explained in Chapter 5. This method requires $O\big(\mathsf{M}(n)\max(m,n)\log(n)\big)$ base field operations. Then, we use the Equation (2.3) on page 56 of this thesis in the following way: we perform the evaluation of the polynomials $\widetilde{p_i}$ at the points in arithmetic progression $0, 1, \ldots, \sigma$, then we compute iteratively the coefficients $s_i$.

Since the polynomials $\widetilde{p_i}$ have degree at most $n$, each one of these $m+n$ multipoint evaluations can be done using our algorithms in Chapter 5, for a cost of

$$\frac{\sigma}{n}\Big(\mathsf{M}(n)\log(n) + O\big(\mathsf{M}(n)\big)\Big)$$

operations. Then, each value $s_i$ is obtained in an on-line manner using $O(m+n)$ operations.

Putting these arguments together, we have proved

**Theorem 15** *Let $\mathcal{P} = \sum_{i=0}^{n} p_i(x)D^i$ be a linear differential operator of order $n$, with polynomial coefficients $p_i(x) = \sum_{j=0}^{m} p_{ij}x^j$ of degree at most $m$. Suppose that $x = 0$ is an ordinary point for $\mathcal{P}$, that is $p_n(0) \neq 0$. Then, the first $\sigma$ coefficients of a power series solution $S = \sum_{i\geq 0} s_i x^i$ of $\mathcal{P}$ at $x = 0$ can be computed using*

$$O\left(\mathsf{M}(n)\max(m,n)\log(n)\big(1+\sigma/n\big)\right)$$

*base field operations.*

**Remark.** The same complexity estimates are valid for the problem of verifying that a given power series $S$ is a solution of $\mathcal{P}S = 0$, or, more generally, to evaluate the differential operator $\mathcal{P}$ on a given power series $S$ at precision $\sigma$.

## 10.3 Apparent singularities and bounds on the coefficients

Given two linear differential homogeneous operators $L_1$ et $L_2$ of orders $m$, respectively $n$, we address the problem of bounding the degrees of the coefficients of their tensor product $L$ in terms of the corresponding bounds for $L_1$ and $L_2$ and of the orders $m$ and $n$.

By definition, the solution space of $L$ consist of all the products $y_1y_2$ of solutions of $L_1$, respectively $L_2$. Therefore, a singular point for one of its solutions is a singular point for a

[1] The exact order of the recurrence is $\max_{a_{i,j}\neq 0}(j-i) - \min_{a_{i,j}\neq 0}(j-i)$.

solution of either $L_1$ or $L_2$ and this has an impact on the leading coefficients of $L$, $L_1$ and $L_2$; more precisely, if $L_1 = p_0(x)D^m + p_1(x)D^{m-1} + \cdots + p_m(x)$ and $L_2 = q_0(x)D^n + q_1(x)D^{n-1} + \cdots + q_n(x)$, then the leading coefficient of $L$ is a multiple of $p_0(x)^n q_0(x)^m$.

The presence of an extra factor of this coefficient is due to the so-called *apparent singularities* of $L$, which are by definition [122, 196], regular singular points of $L$ where the general solution is holomorphic.

**Theorem 16** *[196, 122] Suppose that the roots $\rho_1, \ldots, \rho_n$ of the indicial equation of $L$ at $x = 0$ are distinct nonnegative integers and let $N = \rho_n - \rho_1$ be the difference between the largest and the smallest exponent. The following assertions are equivalent:*

1. *$x = 0$ is an apparent singularity of $L$.*
2. *the Wronskian determinant of $L$ is an analytic function and vanishes at $x = 0$.*
3. *for all power series solution $\sum_n c_n x^n$ of $L$, the $n$ coefficients of the powers $x^{\rho_i}$ are arbitrary.*
4. *the rank of the system expressing the compatibility of $N+1$ recurrence formulas satisfied by the coefficients of $(x^{\rho_1}, x^{\rho_1+1}, \ldots, x^{\rho_n})$ is $N - n + 1$.*
5. *arbitrary values cannot be assigned to $y(0), y'(0), \ldots, y^{(n-1)}(0)$ for any power series solution $y$ of $L$.*
6. *there exists a nonzero power series solution $y$ of $L$ such that $y(0) = \cdots = y^{(n-1)}(0) = 0$.*

Here is a simple example showing how apparent singularities can occur in a tensor product. Let $L_1 = D^2 + 2x - 1$ and $L_2 = (x-1)D^2 + 1$. Their tensor product $L = L_1 \otimes L_2$ equals

$$L = x(x-1)^3(2x-3)D^4 - (x-1)^2(2x^2+3-4x)D^3 + x(x-1)^2(2x-3)(4x^2-6x+4)D^2 + (x-1)(-14x^3+25x-2x^2+4x^4-12)D + (43x^3-85x^4-44x^6-20x^2+90x^5+10x-3+8x^7).$$

Thus, the points $x = 0$ and $x = 3/2$ are ordinary points for $L_1$ and $L_2$, but apparent singularities for their tensor product $L$. Their presence is reflected in the leading coefficient of $L$. A direct verification goes by simple inspection on the exponents of a basis of power series solutions of $L$. Starting from the bases $\{f_1, f_2\}$ of $L_1$ and $\{g_1, g_2\}$ of $L_2$, where

$$\begin{array}{rcl} f_1 & = & -1 - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{24}x^4 + \frac{1}{15}x^5 - \frac{17}{720}x^6 + \frac{1}{280}x^7 + O(x^8) \\ f_2 & = & x + \frac{1}{6}x^3 - \frac{1}{6}x^4 + \frac{1}{120}x^5 - \frac{1}{60}x^6 + \frac{41}{5040}x^7 + O(x^8), \\ g_1 & = & -1 - \frac{1}{2}x^2 - \frac{1}{6}x^3 - \frac{1}{8}x^4 - \frac{1}{12}x^5 - \frac{43}{720}x^6 - \frac{5}{112}x^7 + O(x^8) \quad \text{and} \\ g_2 & = & x + \frac{1}{6}x^3 + \frac{1}{12}x^4 + \frac{7}{120}x^5 + \frac{1}{24}x^6 + \frac{157}{5040}x^7 + O(x^8). \end{array}$$

the following basis for the tensor product $L$ at the origin is obtained

$$\begin{aligned} h_1 &= 1 + x^2 - \tfrac{1}{6}x^3 + \tfrac{5}{12}x^4 - \tfrac{1}{15}x^5 + \tfrac{1}{9}x^6 + \tfrac{37}{2520}x^7 + O(x^8), \\ h_2 &= -x - \tfrac{2}{3}x^3 + \tfrac{1}{4}x^4 - \tfrac{11}{60}x^5 + \tfrac{7}{180}x^6 - \tfrac{53}{840}x^7 + O(x^8), \\ h_3 &= -x - \tfrac{2}{3}x^3 - \tfrac{13}{60}x^5 - \tfrac{1}{90}x^6 - \tfrac{41}{630}x^7 + O(x^8) \quad \text{and} \\ h_4 &= x^2 + \tfrac{1}{3}x^4 - \tfrac{1}{12}x^5 + \tfrac{17}{180}x^6 + \tfrac{1}{90}x^7 + O(x^8). \end{aligned}$$

Since the exponents of the basis $\{h_1, h_2, h_4, h_2 - h_3\}$ are $\{0, 1, 2, 4\}$, the point $x = 0$ is an apparent singularity of $L$. This example suggests a general way of verifying whether a given point is an apparent singularity or not. One simply computes a local basis of solutions at the given point and look for a linear combination of valuation greater than $n - 1$.

**Computing apparent singularities of the tensor product** $L = L_1 \otimes L_2$ We now consider the problem of finding the apparent singularities of the tensor product at $x = 0$, supposed an ordinary point for both $L_1$ and $L_2$.

1. consider the shifted operator $L_1(a)$, obtained by replacing $x$ by $x+a$ in the coefficients of $L_1$. Similarly construct $L_2(a)$.
2. compute two bases $f_1(a), \ldots, f_m(a)$ and $g_1(a), \ldots, g_n(a)$ of power series solutions of $L_1(a)$ and $L_2(a)$ at $x = 0$.
3. compute the first $mn$ coefficients of the $mn$ cross products $h_{ij}(a) = f_i(a)g_j(a)$; form the $mn \times mn$ matrix $W(a)$ whose entries are these coefficients.
4. the desired points are the zeros of the rational function $\det(W(a))$.

**Continued example** The translations of $L_1$ and $L_2$ are the operators

$$L_1(a) = D^2 + 2x + 2a - 1 \quad \text{and} \quad L_2(a) = (x + a - 1)D^2 + 1.$$

Bases of local solutions at $x = 0$ of $L_1(a)$ and $L_2(a)$ with initial conditions $(1, 0)$ and $(0, 1)$ are formed respectively by the power series

$$\begin{aligned} &1 + \left(-a + \frac{1}{2}\right)x^2 - \frac{1}{3}x^3 + O(x^4), \quad x + \left(-\frac{1}{3}a + \frac{1}{6}\right)x^3 + O(x^4) \\ &1 - \frac{1}{2(a-1)}x^2 + \frac{1}{6(a-1)^2}x^3 + O(x^4), \quad x - \frac{1}{6(a-1)}x^3 + O(x^4) \end{aligned}$$

Thus, the mobile wronskian of $L$ at $a$ is the following matrix

$$W(a) = \begin{bmatrix} 1 & 0 & -\frac{(2a^2-3a+2)}{2(a-1)} & -\frac{(2a^2-4a+1)}{6(a-1)^2} \\ 0 & 1 & 0 & -\frac{(6a^2-9a+4)}{6(a-1)} \\ 0 & 1 & 0 & -\frac{(2a^2-3a+4)}{6(a-1)} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The determinant $\det(W(a))$ is the rational fraction $-\frac{a(2a-3)}{3(a-1)}$; its zeros are the apparent singularities of $L$ and its pole is the the unique true singularity of $L$. This remark extends to a general setting and yields the following result.

**Theorem 17** $\det(W(a))$ *is a rational function; its zeros encode the apparent singularities of* $L$ *and its poles are the true singularities of* $L$.

**Theorem 18** *Suppose* $L_1$ *and* $L_2$ *have orders* $d_1$ *and* $d_2$ *and polynomial coefficients of degrees bounded by* $k_1$ *and* $k_2$. *Then*

1. *the tensor product of* $L_1$ *and* $L_2$ *has order at most* $d_1 d_2$ *and coefficients of degree at most* $(d_1 d_2 - d_1 - d_2 + 2)(d_2 k_1 + d_1 k_2)$.
2. *the least common left multiple of* $L_1$ *and* $L_2$ *has order at most* $d_1 + d_2$ *and coefficients of degree at most* $k_1(d_2 + 1) + k_2(d_1 + 1)$.

## 10.4 From power series solution to differential operators

We show how to efficiently recover a linear differential operator $\mathcal{L}$ with polynomial coefficients from the local information provided by the coefficients of a power-series solution of $\mathcal{L}$ at an ordinary point.

### 10.4.1 Pade-Hermite approximation

Given $\mathbf{F}$ an $(m+1)$-tuple of formal power series $(f_0, \ldots, f_m)$, and given $\mathbf{d}$ an $(m+1)$-tuple of integers $(d_0, \ldots, d_m)$, one classically defines *a Padé-Hermite approximant* for $\mathbf{F}$ of type $\mathbf{d}$ as a nontrivial tuple $\mathbf{P}$ of polynomials $P_j$ over $\mathbb{K}$ having degrees bounded by $d_j$ such that:

$$\mathbf{P}(z) \cdot \mathbf{F}(z) = \sum_j P_j(z) f_j(z) = O\big(z^\sigma\big),$$

where $\sigma := d_0 + \cdots + d_m + m$.

The Padé-Hermite approximation problem was introduced in 1873 by Hermite and has been widely studied for several authors [245] [17], [18], [11], [12]. The case $m = 1$ corresponds to the Padé approximation problem for a single power series, see also Section 3.3.4 of this thesis. Padé-Hermite approximation includes as important sub-cases the algebraic approximants $(f_j = f^j)$ [219], [220], [148].

In recent years several matrix generalizations have also been studied; a uniform approach to the various aspects of Padé-Hermite approximation is presented in [18] and in [245]. Beckermann and Labahn proposed a fast, as well as a "super-fast" algorithm for computing Padé-Hermite approximates. Recently, an "ultra-fast" version of their algorithm has been designed [148, 97]. For the sake of completeness, we recall the complexity result in [148, 97].

**Theorem 19** *Let* $\mathbf{F} = (f_0, \ldots, f_m)$ *be a* $(m+1)$*-tuple of formal power series and let* $d \geq 0$. *Then one can compute a Padé-Hermite approximant for* $\mathbf{F}$ *of type* $(d, \ldots, d)$ *within*

$$O\big(\mathsf{MM}(m, d) \log(d)\big) = O_{\log}(m^\omega d)$$

*base field operations.*

Let us notice that this algorithm computes a *basis* of approximants, whose size is $O(m^2 d)$, thus it is nearly optimal for this task. However, we use this algorithm in the very special case where the input series are the successive derivatives of a fixed power series. For the moment we are unable to adapt this result so as to exploit the special structure of the differential approximant problem. However, we proceed with this approach, since we feel that such efficient *structured Padé-Hermite approximants* should exist; the, the rest of our results will yield very efficient algorithms.

**Theorem 20** *Suppose that $\mathcal{P}$ is an unknown linear differential operator, whose order $p$ and maximum degree $k$ of coefficients are known. Then starting from a generic power series solution given at precision $(k+2)(p+1)$ around an ordinary point, $\mathcal{P}$ can be recovered using*

$$O_{\log}\big(p^{\omega}(k+1)\big)$$

*base field operations.*

**Continued example** Suppose the bound $d = 7$ is apriori known for degree of the coefficients of $L$. We compute the first $5\times 9-1 = 44$ terms of power series solutions $y_1$, respectively $y_2$ for $L_1$ and $L_2$ at the ordinary point $x = 0$, then compute the product $y = y_1 y_2$ and his $m = 4$ successive derivatives and then apply the Padé-Hermite routine to it. The output of this computation is, as expected, a vector collinear with that of the coefficients of $L$.

$$\begin{bmatrix} x^7 - \frac{11}{2}x^6 + \frac{45}{4}x^5 - \frac{85}{8}x^4 + \frac{43}{8}x^3 - \frac{5}{2}x^2 + \frac{5}{4}x - \frac{3}{8} \\ \frac{1}{2}x^5 - \frac{9}{4}x^4 + \frac{3}{2}x^3 + \frac{27}{8}x^2 - \frac{37}{8}x + \frac{3}{2} \\ x^6 - 5x^5 + \frac{41}{4}x^4 - 11x^3 + \frac{25}{4}x^2 - \frac{3}{2}x \\ -\frac{1}{4}x^4 + x^3 - \frac{13}{8}x^2 + \frac{5}{4}x - \frac{3}{8} \\ \frac{1}{4}x^5 - \frac{9}{8}x^4 + \frac{15}{8}x^3 - \frac{11}{8}x^2 + \frac{3}{8}x \end{bmatrix}.$$

### 10.4.2 Wronskians

An alternative algorithm can be obtained if one is given as input a basis of power series solutions at $x = 0$ of the shifted operator $L(a)$. The complexity result is formulated in the following theorem.

**Theorem 21** *Let $\mathcal{P}$ be an unknown linear differential operator, whose order $p$ and maximum degree $k$ of coefficients are known. Suppose that a basis of local solutions at $x = 0$ of the shifted operator $\mathcal{P}(a)$ is given at precision $p+1$:*

$$y_i = \sum_{j\geq 0} \frac{u_{i,j}(a)}{v_{i,j}(a)} \frac{x^j}{j!} + O(x^{p+1}),$$

*where the polynomials $u_{i,j}$ and $v_{i,j}$ have degrees bounded by $N$. Then, one can recover the operator $\mathcal{P}$ using*

$$O_{\log}\big(p^{\omega} N\big)$$

*base field operations.*

Indeed, the *mobile wronskian matrix* $W(a)$ containing the rational fraction coefficients $y_{i,j}$ of solutions

$$y_i = \sum_{j \geq 0} y_{i,j}(a) x^j + O(x^{p+1}), \quad (i = 1 \dots p)$$

verifies the equality:

$$\begin{bmatrix} y_1(a) & y_1'(a) & \dots & y_1^{(p-1)}(a) \\ \vdots & \vdots & & \vdots \\ y_p(a) & y_p'(a) & \dots & y_p^{(p-1)}(a) \end{bmatrix} \cdot \begin{bmatrix} \frac{a_0}{a_p}(a) \\ \vdots \\ \frac{a_{p-1}}{a_p}(a) \end{bmatrix} = \begin{bmatrix} -y_1^{(p)}(a) \\ \vdots \\ -y_p^{(p)}(a) \end{bmatrix}.$$

This system can be solved using Storjohann's algorithm described in Section 3.4.2 within the announced running time bound.

**Remark** Storjohann's result is nearly optimal for generic polynomial matrices. In our case, we apply it to matrices $W(a)$ which have particular structures. For such matrices, an improved version of Storjohann's algorithm has yet to be found. Typically, specific bounds as those presented for the lclm or tensor product help to speed-up only half of Storjohann's algorithm, roughly by an order of magnitude. Unfortunately, this is not enough to improve the complexity of the whole computation, since the *high-order lifting* step cannot take benefit of tight bounds in a direct fashion, so it continues to work as in the generic case. We leave the problem of designing efficient algorithms for *structured* polynomial matrices as an important open question.

## 10.5 Application to lclm and tensor product

Let us summarize in the table below the complexity results for the computation of lclm and of tensor products, corresponding to the conversion algorithms described in the preceding sections. Since only the conversion from power series to operators may differ, we indicate only the algorithm used for that conversion.

To simplify the notation, we assume that both $L_1$ and $L_2$ have orders $n$ and polynomial coefficients of degree at most $n$.

| algorithm | $L_1 \otimes L_2$ | $\mathrm{LCLM}(L_1, L_2)$ |
|---|---|---|
| `Padé-Hermite` | $O_{\log}\left(n^{2\omega+4}\right)$ | $O_{\log}\left(n^{\omega+2}\right)$ |
| `mobile Wronskian` | $O_{\log}\left(n^{2\omega+3}\right)$ | $O_{\log}\left(n^{\omega+2}\right)$ |
| size of output | $O(n^6)$ | $O(n^3)$ |

Figure 10.1: Complexity results for lclm and tensor products.

## 10.6 Conclusions

In this chapter, we proposed a fast method for computing with linear differential operators. This method is inspired by the usual *evaluation / interpolation* strategy in the commutative polynomial case. It yields algorithms for lclm and tensor product, which are sub-quadratic in the size of the output. However, these algorithms are not optimal, due to the reconstruction step, performed using methods (like Padé-Hermite approximant algorithms or Storjohann's algorithm) which do not take into account the special (differential) structure of the problem. We feel that fast, structure sensitive, versions of these methods should exist, and we leave this as a direction of further research. With the notations in Table 10.1, such improved structured versions would decrease the actual complexities by one order of magnitude, that is, down to $O(n^{2\omega+2})$ for the tensor product and down to $O(n^{\omega+1})$ for the lclm.

# Au lieu de conclusion

## Mise en contexte

L'un des aspects du travail de cette thèse a porté sur la conception d'algorithmes efficaces pour le calcul des opérations de base avec les opérateurs différentiels linéaires à coefficients polynomiaux. Ceci fait partie d'un programme plus vaste de recherche dans le cadre de la théorie de l'élimination dans les $\mathcal{D}$-*modules* et ses applications aux *fonctions spéciales*.

Mon intérêt pour le cadre différentiel linéaire a une motivation multiple. D'une part, les opérateurs différentiels linéaires fournissent une structure de données très flexible pour représenter de nombreuses fonctions spéciales ; ce point de vue est similaire à celui permettant de manipuler des nombres algébriques via leurs polynômes minimaux. Par ailleurs, le cadre différentiel linéaire suffit pour englober des applications algorithmiques importantes à la sommation et à l'intégration définie des fonctions spéciales, domaine initié par D. Zeilberger au début des années '90, [265].

Dans la lignée des travaux fondateurs de Zeilberger, F. Chyzak [61] a montré la possibilité d'algorithmiser l'intégration et la sommation des fonctions spéciales définies par des systèmes d'équations différentielles, via des calculs de bases de Gröbner dans des anneaux non-commutatifs. Le sujet est cependant loin d'être clos car de nombreux problèmes d'efficacité, tant théoriques que pratiques, demeurent. Ces problèmes sont liés à la structure de données utilisée – la représentation développée des polynômes – et, a posteriori, à l'utilisation des techniques de réécriture, dont la complexité est intrinsèquement élevée.

Par ailleurs, le même problème est présent dans les méthodes classiques de résolution des systèmes polynomiaux. Or, depuis quelques années, une alternative aux méthodes de réécriture a été proposée par G. Lecerf [149], dans la continuation de travaux récents du groupe de recherche international TERA [1]. L'approche repose crucialement sur la *représentation en évaluation des polynômes*. Ainsi, il a été montré que les problèmes d'élimination en géométrie algébrique classique peuvent être ramenés dans une classe de complexité polynomiale si on adopte la représentation en évaluation des polynômes.

Un objectif important est d'importer les techniques d'évaluation dans le cadre non-commutatif des polynômes différentiels à plusieurs dérivations. Pour que cela puisse se réaliser, il faudra :

- comprendre la géométrie du processus d'élimination sous-jacent aux méthodes de sommation et d'intégration symboliques, par exemple, dans la méthode du *creative telescoping* [265].

- développer une algorithmique efficace adaptée à ce type d'élimination, en essayant d'importer les techniques qui ont déjà permis de réduire la complexité de l'élimination dans le cadre de la géométrie algébrique effective.

## Détail de quelques perspectives de recherche

La *complexité* est à la frontière entre les mathématiques et l'informatique. En calcul formel et en calcul numérique, les meilleurs résultats de complexité reposent souvent sur des algorithmes dont la conception parvient à exploiter la *géométrie* du problème à résoudre.

En partant de résultats récents sur la complexité de la *résolution géométrique* de systèmes polynomiaux [149] et des acquis algorithmiques décrits dans cette thèse, il est envisageable de procéder au défrichage du champ algorithmique des opérateurs différentiels linéaires – du point de vue de la complexité.

Dans ce mémoire, j'ai proposé des algorithmes efficaces concernant principalement les polynômes commutatifs, à une ou plusieurs variables. La ressemblance formelle entre les calculs sur les polynômes et les calculs sur les opérateurs différentiels linéaires est évidente. Cependant, la mise en pratique d'idées exploitant cette ressemblance est difficile : la non-commutativité du différentiel linéaire rend la traduction directe des algorithmes existants en commutatif soit inopérante soit trop complexe.

Un petit nombre de principes méthodologiques (*diviser pour régner*, *évaluation–interpolation*, *pas de bébés–pas de géants*) sont à la base de la plupart des algorithmes fondamentaux les plus efficaces en calcul formel. Mon approche repose sur l'établissement de liens entre ces principes et des notions géométriques ou algébriques (*réductibilité, fibres, dualité*), et l'exploitation de ces liens pour la conception d'algorithmes efficaces.

Les résultats attendus sont de nature algorithmique et logicielle. En particulier, je vise la mise au point de nouveaux algorithmes d'intégration définie et de sommation définie de fonctions spéciales. La méthodologie que j'envisage consiste à *importer* dans l'univers non-commutatif les progrès de complexité obtenus par une approche géométrique en commutatif.

**Calculs rapides sur des opérateurs linéaires univariés.** Dans le domaine univarié, la mise en évidence d'un cadre commun pour les opérateurs linéaires et les polynômes remonte à Ore dans les années 1930. En particulier, Ore a explicité un algorithme d'Euclide étendu permettant de calculer le pgcd et le ppcm d'opérateurs linéaires. Dans le cadre commutatif univarié, la complexité de ces calculs se réduit classiquement à celle du produit de polynômes, elle-même essentiellement optimale via la transformée de Fourier rapide. Par comparaison, le produit rapide d'opérateurs différentiels n'est connu que depuis peu [247] comme étant équivalent au produit de matrices. En ce qui concerne le calcul du pgcd, le seul progrès connu par rapport à l'algorithme d'Euclide naïf est une version non-commutative des sous-résultants, dont la complexité est loin d'être optimale.

Dans le dernier chapitre de cette thèse, j'ai proposé une méthode de type évaluation-interpolation pour certaines opérations usuelles sur les opérateurs différentiels linéaires à coefficients polynomiaux, permettant d'aborder le ppcm et en même temps que le produit symétrique. Le rôle des points d'évaluation est joué par des séries formelles sur lesquelles on évalue des opérateurs différentiels, tandis que l'interpolation consiste à reconstruire un opérateur à partir de ses solutions séries (tronquées).

Cependant, si pour l'étape d'*évaluation* on dispose maintenant d'algorithmes satisfaisants, je ne connais pas d'algorithme optimal pour l'étape de *reconstruction*. Ainsi, pour l'instant,

la solution adoptée passe par l'emploi d'algorithmes généraux de calculs d'*approximants de Padé-Hermite*, qui n'arrivent à exploiter que partiellement la structure spéciale du problème. C'est pourquoi, je me propose comme objectif à court terme de chercher un algorithme adapté à ce cadre. Je vise également un traitement de ces questions pour les opérateurs aux ($q$–) différences, et plus généralement dans le cas des opérateurs de Ore.

**Élimination multivariée non-commutative.** Une extension naturelle de ces opérations dans le cadre multivarié consiste à considérer des idéaux d'opérateurs linéaires (différentiels, ou plus généralement, de Ore) dans des algèbres appropriées. Les idéaux pertinents sont dits $\mathcal{D}$-finis ; ils sont engendrés par des systèmes d'opérateurs dont les solutions forment un espace vectoriel de dimension finie. Les mêmes questions qu'en univarié peuvent alors être abordées : produit symétrique, somme ou union d'idéaux $\mathcal{D}$-finis. F. Chyzak et B. Salvy [63, 61] ont récemment développé une algorithmique de ces systèmes fondée sur des calculs de bases de Gröbner non-commutatives. Un programme à moyen terme que je voudrais démarrer consiste à améliorer l'efficacité de ces calculs, soit en exploitant les progrès récents de calculs de bases de Gröbner commutatives [78], soit en amorçant une approche de type résolution géométrique non-commutative.

**Création téléscopique.** Une opération d'élimination très importante, permettant le calcul des intégrales définies, est la *création télescopique* (*creative telescoping* en anglais) due à D. Zeilberger. Cette opération pose un problème d'élimination spécifique au non-commutatif : il s'agit d'éliminer une variable dans la somme d'un idéal à gauche et d'un idéal à droite (cette somme n'est pas un idéal). Plus généralement, une résolution algorithmique satisfaisante de cette question permettrait de traiter efficacement d'autres problèmes importants comme : le calcul du polynôme de Bernstein-Sato en théorie des $\mathcal{D}$-modules ; la recherche de solutions particulières d'équations fonctionnelles linéaires inhomogènes ; les calculs d'*eigenring* en théorie de Galois différentielle ou aux différences.

Dans le cadre purement différentiel, Zeilberger a proposé un algorithme qui repose sur le calcul d'un sous-idéal de l'idéal cherché [265]. Cet algorithme ne fonctionne qu'à condition de disposer d'une description d'un *module holonome* (au sens de la théorie des $\mathcal{D}$-modules) correspondant à l'idéal considéré. Par ailleurs, F. Chyzak a donné récemment un algorithme rapide [62], généralisant un résultat de H. Wilf et D. Zeilberger [259], qui calcule également un sous-idéal de l'idéal cherché, mais dont la terminaison n'est assurée qu'en cas d'holonomie.

D'ailleurs, l'hypothèse d'holonomie est à ce jour requise par tous les algorithmes connus d'intégration définie de fonctions spéciales et cette limitation semble artificielle.

C'est pourquoi, il est naturel de rechercher des solutions algorithmiques permettant de calculer la création télescopique *indépendamment* du caractère holonome de l'idéal considéré. Je prévois d'attaquer cette question en appliquant et en étendant des résultats récents de H. Tsai [243] aux algèbres de Ore.

# List of Figures

# Bibliography


[1] http://tera.medicis.polytechnique.fr/.

[2] Abramov, S. A. On the summation of rational functions. *Ž. Vyčisl. Mat. i Mat. Fiz. 11*, 4 (1971), 1071–1075. English translation in U.S.S.R. Comp. Maths. Math. Phys., 324–330.

[3] Abramov, S. A. Rational solutions of linear differential and difference equations with polynomial coefficients. *Zh. Vychisl. Mat. i Mat. Fiz. 29*, 11 (1989), 1611–1620, 1757. English translation in U.S.S.R. Comp. Maths. Math. Phys., 7–12.

[4] Abramowitz, M., and Stegun, I. A., Eds. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Dover Publications Inc., New York, 1992. Reprint of the 1972 edition.

[5] Aho, A. V., Steiglitz, K., and Ullman, J. D. Evaluating polynomials at fixed sets of points. *SIAM Journal on Computing 4*, 4 (1975), 533–539.

[6] Alonso, M.-E., Becker, E., Roy, M.-F., and Wörmann, T. Zeros, multiplicities, and idempotents for zero-dimensional systems. In *Algorithms in algebraic geometry and applications (Santander, 1994)*. Birkhäuser, Basel, 1996, pp. 1–15.

[7] Andrews, G. E., Askey, R., and Roy, R. *Special functions*, vol. 71 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1999.

[8] Antoniou, A. *Digital Filters: Analysis and Design*. McGraw-Hill Book Co., 1979.

[9] Arnaudiès, J.-M., and Valibouze, A. Lagrange resolvents. *Journal of Pure and Applied Algebra 117/118* (1997), 23–40. Algorithms for algebra (Eindhoven, 1996).

[10] Bailey, D., and Paar, C. Optimal extension fields for fast arithmetic in public-key algorithms. In *Advances in Cryptology – CRYPTO '98* (1998), H. Krawczyk, Ed., vol. 1462 of *LNCS*, Springer-Verlag, pp. 472–485.

[11] Baker, Jr., G. A., and Graves-Morris, P. *Padé approximants. Part I*. Addison-Wesley Publishing Co., Reading, Mass., 1981. Basic theory, With a foreword by Peter A. Carruthers.

[12] Baker, Jr., G. A., and Graves-Morris, P. *Padé approximants. Part II.* Addison-Wesley Publishing Co., Reading, Mass., 1981. Extensions and applications, With a foreword by Peter A. Carruthers.

[13] Banderier, C., and Flajolet, P. Basic analytic combinatorics of directed lattice paths. *Theoretical Computer Science 281*, 1-2 (2002), 37–80.

[14] Baur, W., and Strassen, V. The complexity of partial derivatives. *Theoretical Computer Science 22* (1983), 317–330.

[15] Becker, E., Cardinal, J. P., Roy, M.-F., and Szafraniec, Z. Multivariate Bezoutians, Kronecker symbol and Eisenbud-Levine formula. In *Algorithms in algebraic geometry and applications (Santander, 1994)*. Birkhäuser, Basel, 1996, pp. 79–104.

[16] Becker, E., and Wörmann, T. Radical computations of zero-dimensional ideals and real root counting. *Mathematics and Computers in Simulation 42*, 4-6 (1996), 561–569. Symbolic computation, new trends and developments (Lille, 1993).

[17] Beckermann, B., and Labahn, G. A uniform approach for Hermite Padé and simultaneous Padé approximants and their matrix-type generalizations. *Numerical Algorithms 3*, 1-4 (1992), 45–54.

[18] Beckermann, B., and Labahn, G. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM Journal on Matrix Analysis and Applications 15*, 3 (1994), 804–823.

[19] Ben-Or, M., and Tiwari, P. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of STOC'88* (1988), ACM Press, pp. 301–309.

[20] Berlekamp, E. R. *Algebraic coding theory.* McGraw-Hill Book Co., New York, 1968.

[21] Bernstein, D. J. Multidigit multiplication for mathematicians. Preprint, available from `http://cr.yp.to/papers.html`.

[22] Bernstein, D. J. Removing redundancy in high precision Newton iteration. Preprint, available from `http://cr.yp.to/papers.html`.

[23] Bernstein, D. J. Composing power series over a finite ring in essentially linear time. *Journal of Symbolic Computation 26*, 3 (1998), 339–341.

[24] Berstel, J., and Mignotte, M. Deux propriétés décidables des suites récurrentes linéaires. *Bull. Soc. Math. France 104*, 2 (1976), 175–184.

[25] Bini, D., and Pan, V. Y. *Polynomial and matrix computations. Vol. 1.* Birkhäuser Boston Inc., Boston, MA, 1994. Fundamental algorithms.

[26] Björck, G. Functions of modulus 1 on $Z_n$ whose Fourier transforms have constant modulus, and "cyclic $n$-roots". In *Recent advances in Fourier analysis and its applications (Il Ciocco, 1989)*, vol. 315 of *NATO Advance Science Institutes Series C: Mathematical and Physical Sciences*. Kluwer Academic Publishers, Dordrecht, 1990, pp. 131–140.

[27] Bluestein, L. I. A linear filtering approach to the computation of the discrete Fourier transform. *IEEE Trans. Electroacoustics AU-18* (1970), 451–455.

[28] Bommer, R. High order derivations and primary ideals to regular prime ideals. *Archiv der Mathematik 46*, 6 (1986), 511–521.

[29] Bordewijk, J. L. Inter-reciprocity applied to electrical networks. *Appl. Sci. Res. B. 6* (1956), 1–74.

[30] Borodin, A., and Moenck, R. T. Fast modular transforms. *Comput. System Sci. 8*, 3 (1974), 366–386.

[31] Borodin, A., and Munro, I. *The computational complexity of algebraic and numeric problems*. American Elsevier Publishing Co., Inc., New York-London-Amsterdam, 1975. Elsevier Computer Science Library; Theory of Computation Series, No. 1.

[32] Bosma, W., Cannon, J., and Playoust, C. The Magma algebra system. I. The user language. *Journal of Symbolic Computation 24*, 3-4 (1997), 235–265. See also `http://www.maths.usyd.edu.au:8000/u/magma/`.

[33] Bostan, A., Flajolet, P., Salvy, B., and Schost, É. Fast computation with two algebraic numbers. Research Report 4579, Institut National de Recherche en Informatique et en Automatique, Oct. 2002. 20 pages.

[34] Bostan, A., Gaudry, P., and Schost, É. Linear recurrences with polynomial coefficients and computation of the Cartier-Manin operator on hyperelliptic curves. In *Finite fields and their applications (Toulouse, 2003)* (2003), Springer–Verlag. To appear.

[35] Bostan, A., Lecerf, G., and Schost, É. Tellegen's principle into practice. In *Proceedings of ISSAC'03* (2003), ACM Press, pp. 37–44.

[36] Bostan, A., Salvy, B., and Schost, É. Fast algorithms for zero-dimensional polynomial systems using duality. *Applicable Algebra in Engineering, Communication and Computing 14*, 4 (2003), 239–272.

[37] Bostan, A., and Schost, É. On the complexities of multipoint evaluation and interpolation. Tech. rep., École polytechnique, 2003.

[38] Bostan, A., and Schost, É. Polynomial evaluation and interpolation on special sets of points. Tech. rep., École polytechnique, 2003.

[39] Brawley, J. V., and Carlitz, L. Irreducibles and the composed product for polynomials over a finite field. *Discrete Mathematics 65*, 2 (1987), 115–139.

[40] Brawley, J. V., Gao, S., and Mills, D. Computing composed products of polynomials. In *Finite fields: theory, applications, and algorithms (Waterloo, ON, 1997)*. Amer. Math. Soc., Providence, RI, 1999, pp. 1–15.

[41] Brent, R. P. Algorithms for matrix multiplication. Tech. Rep. CS-157, Stanford University, 1970.

[42] Brent, R. P. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic computational complexity (Proc. Sympos., Carnegie-Mellon Univ., Pittsburgh, Pa., 1975)*. Academic Press, New York, 1976, pp. 151–176.

[43] Brent, R. P., and Kung, H. T. Fast algorithms for manipulating formal power series. *J. Assoc. Comput. Mach. 25*, 4 (1978), 581–595.

[44] Briand, E., and González-Vega, L. Multivariate Newton sums: identities and generating functions. *Communications in Algebra 30*, 9 (2002), 4527–4547.

[45] Buchberger, B. Gröbner bases: An algorithmic method in polynomial ideal theory. In *Multidimensional System Theory*. Reidel, Dordrecht, 1985, pp. 374–383.

[46] Bunch, J., and Hopcroft, J. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation 28* (1974), 231–236.

[47] Bürgisser, P., Clausen, M., and Shokrollahi, M. A. *Algebraic complexity theory*, vol. 315 of *Grundlehren Math. Wiss.* Springer-Verlag, 1997.

[48] Canny, J. Some algebraic and geometric problems in PSPACE. In *Proceedings of STOC'88* (1988), ACM Press, pp. 460–467.

[49] Canny, J., Kaltofen, E., and Yagati, L. Solving systems of non-linear polynomial equations faster. In *Proceedings of ISSAC'89* (1989), ACM Press, pp. 121–128.

[50] Cantor, D. G., and Kaltofen, E. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica 28*, 7 (1991), 693–701.

[51] Capocelli, R. M., Cerbone, G., Cull, P., and Holloway, J. L. Fibonacci facts and formulas. In *Sequences (Naples/Positano, 1988)*. Springer, New York, 1990, pp. 123–137.

[52] Cartier, P. Une nouvelle opération sur les formes différentielles. *C. R. Acad. Sci. Paris 244* (1957), 426–428.

[53] Casperson, D., and McKay, J. Symmetric functions, $m$-sets, and Galois groups. *Mathematics of Computation 63*, 208 (1994), 749–757.

[54] Cerlienco, L., Mignotte, M., and Piras, F. Suites récurrentes linéaires: propriétés algébriques et arithmétiques. *Enseign. Math. (2) 33*, 1-2 (1987), 67–108.

[55] Charlap, L. S., Coley, R., and Robbins, D. Enumeration of rational points on elliptic curves over finite fields. Preprint, 1991.

[56] Chistov, A. L., and Grigoriev, D. Y. Polynomial-time factoring of multivariable polynomials over a global field. LOMI preprint E-5-82, Steklov Institute, Leningrad, 1982.

[57] Chudnovsky, D. V., and Chudnovsky, G. V. On expansion of algebraic functions in power and Puiseux series. I. *J. Complexity 2*, 4 (1986), 271–294.

[58] Chudnovsky, D. V., and Chudnovsky, G. V. On expansion of algebraic functions in power and Puiseux series. II. *J. Complexity 3*, 1 (1987), 1–25.

[59] Chudnovsky, D. V., and Chudnovsky, G. V. Approximations and complex multiplication according to Ramanujan. In *Ramanujan revisited (Urbana-Champaign, Ill., 1987)*. Academic Press, Boston, MA, 1988, pp. 375–472.

[60] Chudnovsky, D. V., and Chudnovsky, G. V. Computer algebra in the service of mathematical physics and number theory. In *Computers in Mathematics (Stanford, CA, 1986) (New York)*, vol. 125. Dekker, 1990, pp. 109–232.

[61] Chyzak, F. Gröbner bases, symbolic summation and symbolic integration. In *Gröbner bases and applications (Linz, 1998)*. Cambridge University Press, Cambridge, 1998, pp. 32–60.

[62] Chyzak, F. An extension of Zeilberger's fast algorithm to general holonomic functions. *Discrete Mathematics 217*, 1-3 (2000), 115–134. Formal power series and algebraic combinatorics (Vienna, 1997).

[63] Chyzak, F., and Salvy, B. Non-commutative elimination in Ore algebras proves multivariate identities. *Journal of Symbolic Computation 26*, 2 (1998), 187–227.

[64] Cohen, H. *A course in computational algebraic number theory.* Springer-Verlag, Berlin, 1993.

[65] Cook, S. A. *On the minimum computation time of functions.* PhD thesis, Harvard, 1966.

[66] Cooley, J. W. The re-discovery of the Fast Fourier Transform algorithm. *Mikrochimica Acta 3* (1987), 33–45.

[67] Cooley, J. W. How the FFT gained acceptance. In *A history of scientific computing (Princeton, NJ, 1987)*, ACM Press Hist. Ser. ACM Press, New York, 1990, pp. 133–140.

[68] Coppersmith, D. Rectangular matrix multiplication revisited. *Journal of Complexity 13*, 1 (1997), 42–49.

[69] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation 9*, 3 (Mar. 1990), 251–280.

[70] Cox, D., Little, J., and O'Shea, D. *Using algebraic geometry*. Springer-Verlag, New York, 1998.

[71] Cull, P., and Holloway, J. L. Computing Fibonacci numbers quickly. *Information Processing Letters 32*, 3 (1989), 143–149.

[72] Dongarra, J., and Sullivan, F. Top Ten Algorithms. *Computing in Science & Engineering 2*, 1 (2000).

[73] Dornstetter, J.-L. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Trans. Inform. Theory 33*, 3 (1987), 428–431.

[74] Dvornicich, R., and Traverso, C. Newton symmetric functions and the arithmetic of algebraically closed fields. In *Proceedings of AAECC-5*, vol. 356 of *LNCS*. Springer, Berlin, 1989, pp. 216–224.

[75] Edwards, A. W. F. A quick route to sums of powers. *American Mathematical Monthly 93* (1986), 451–455.

[76] Eisenbud, D. *Commutative algebra, with a view toward algebraic geometry*. Graduate Texts in Mathematics. Springer-Verlag, New York, 1995.

[77] Faugère, J.-C. A new efficient algorithm for computing Gröbner bases ($F_4$). *Journal of Pure and Applied Algebra 139*, 1-3 (1999), 61–88. Proceedinds of MEGA'98.

[78] Faugère, J.-C. A new efficient algorithm for computing Gröbner bases without reduction to zero ($F5$). In *Proceedings of ISSAC'02* (2002), ACM Press.

[79] Faugère, J.-C., Gianni, P., Lazard, D., and Mora, T. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *Journal of Symbolic Computation 16*, 4 (1993), 329–344.

[80] Fiduccia, C. M. On obtaining upper bounds on the complexity of matrix multiplication. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*. Plenum, New York, 1972, pp. 31–40.

[81] Fiduccia, C. M. *On the algebraic complexity of matrix multiplication*. PhD thesis, Brown Univ., Providence, RI, Center Comput. Inform. Sci., Div. Engin., 1973.

[82] Fiduccia, C. M. An efficient formula for linear recurrences. *SIAM Journal on Computing 14*, 1 (1985), 106–112.

[83] Finck, T., Heinig, G., and Rost, K. An inversion formula and fast algorithms for Cauchy-Vandermonde matrices. *Linear Algebra and its Applications 183* (1993), 179–191.

[84] Fischer, P. C. Further schemes for combining matrix algorithms. In *LNCS* (1974), J. Loeck, Ed., vol. 14, pp. 428–436.

[85] Flajolet, P., and B. Salvy, B. The SIGSAM challenges: Symbolic asymptotics in practice. *SIGSAM Bull. 31*, 4 (1997), 36–47.

[86] Gaudry, P. *Algorithmique des courbes hyperelliptiques et applications à la cryptologie.* PhD thesis, École polytechnique, 2000.

[87] Gaudry, P., and Gürel, N. Counting points in medium characteristic using Kedlaya's algorithm. Preprint.

[88] Gaudry, P., and Harley, R. Counting points on hyperelliptic curves over finite fields. In *Proceedings of ANTS-IV* (2000), W. Bosma, Ed., vol. 1838 of *LNCS*, Springer-Verlag, pp. 313–332.

[89] Gaudry, P., and Schost, É. Cardinality of a genus 2 hyperelliptic curve over GF($5 \cdot 10^{24} + 41$). e-mail to the `NMBRTHRY` mailing list. Sept. 2002.

[90] Gaudry, P., and Schost, É. Modular equations for hyperelliptic curves. Tech. rep., École polytechnique, 2002.

[91] Gauss, C. F. Summatio quarundam serierum singularium. *Opera, Vol. 2, Göttingen: Gess. d. Wiss.* (1863), 9–45.

[92] Gerhard, J. Modular algorithms for polynomial basis conversion and greatest factorial factorization. In *Proceedings of RWCA'00* (2000), pp. 125–141.

[93] Gerhard, J., Giesbrecht, M., Storjohann, A., and Zima, E. V. Shiftless decomposition and polynomial-time rational summation. In *Proceedings of ISSAC'03* (2003), ACM Press, pp. 119–126.

[94] Gianni, P., and Mora, T. Algebraic solution of systems of polynomial equations using Gröbner bases. In *Proceedings of AAECC-5* (1989), vol. 356 of *LNCS*, Springer-Verlag, pp. 247–257.

[95] Giesbrecht, M. Nearly optimal algorithms for canonical matrix forms. *SIAM Journal on Computing 24*, 5 (1995), 948–969.

[96] Gilbert, J.-C., Le Vey, G., and Masse, J. La différentiation automatique de fonctions représentées par des programmes. Tech. rep., RR INRIA 1557, 1991.

[97] Giorgi, P., Jeannerod, C.-P., and Villard, G. On the complexity of polynomial matrix computations. In *Proceedings of ISSAC'03* (2003), ACM Press, pp. 135–142.

[98] Giusti, M. Géométrie effective. Cours de DEA, 1999/2000. Université Paris 6.

[99] Giusti, M., and Heintz, J. La détermination des points isolés et de la dimension d'une variété algébrique peut se faire en temps polynomial. In *Computational Algebraic Geometry and Commutative Algebra* (1993), D. Eisenbud and L. Robbiano, Eds., vol. XXXIV of *Symposia Matematica*, Cambridge University Press, pp. 216–256.

[100] Giusti, M., Heintz, J., Hägele, K., Morais, J. E., Pardo, L. M., and Montaña, J. L. Lower bounds for Diophantine approximations. *Journal of Pure and Applied Algebra 117/118* (1997), 277–317. Algorithms for algebra (Eindhoven, 1996).

[101] Giusti, M., Heintz, J., Morais, J. E., Morgenstern, J., and Pardo, L. M. Straight-line programs in geometric elimination theory. *Journal of Pure and Applied Algebra 124* (1998), 101–146.

[102] Giusti, M., Lazard, D., and Valibouze, A. Algebraic transformations of polynomial equations, symmetric polynomials and elimination. In *Proceedings of ISSAC'88*, P. Gianni, Ed., vol. 358 of *LNCS*. Springer-Verlag, 1989, pp. 309–314.

[103] Giusti, M., Lecerf, G., and Salvy, B. A Gröbner free alternative for polynomial system solving. *Journal of Complexity 17*, 1 (2001), 154–211.

[104] Gohberg, I., and Olshevsky, V. Complexity of multiplication with vectors for structured matrices. *Linear Algebra and its Applications 202* (1994), 163–192.

[105] Gohberg, I., and Olshevsky, V. Fast algorithms with preprocessing for matrix-vector multiplication problems. *Journal of Complexity 10*, 4 (1994), 411–427.

[106] González-López, M.-J., and González-Vega, L. Newton identities in the multivariate case: Pham systems. In *Gröbner bases and applications (Linz, 1998)*, vol. 251 of *London Math. Soc. Lecture Note Ser.* Cambridge University Press, Cambridge, 1998, pp. 351–366.

[107] González-Vega, L., and Trujillo, G. Implicitization of parametric curves and surfaces by using symmetric functions. In *Proceedings of ISSAC'95* (1995), ACM Press, pp. 180–186.

[108] González-Vega, L., and Trujillo, G. Using symmetric functions to describe the solution set of a zero-dimensional ideal. In *Proceedings of AAECC-11*, vol. 948 of *LNCS*. Springer, Berlin, 1995, pp. 232–247.

[109] Gosper, Jr., R. W. Decision procedure for indefinite hypergeometric summation. *Proc. Nat. Acad. Sci. U.S.A. 75*, 1 (1978), 40–42.

[110] Göttfert, R., and Niederreiter, H. On the minimal polynomial of the product of linear recurring sequences. *Finite Fields and their Applications 1*, 2 (1995), 204–218. Special issue dedicated to Leonard Carlitz.

[111] Gourdon, X., and Salvy, B. Effective asymptotics of linear recurrences with rational coefficients. *Discrete Mathematics 153*, 1-3 (1996), 145–163.

[112] Graham, R. L., Knuth, D. E., and Patashnik, O. *Concrete mathematics*, second ed. Addison-Wesley Publishing Company, Reading, MA, 1994. A foundation for computer science.

[113] Gries, D., and Levin, G. Computing Fibonacci numbers (and similarly defined functions) in log time. *Information Processing Letters 11*, 2 (1980), 68–69.

[114] Gröbner, W. La théorie des idéaux et la géométrie algébrique. In *Deuxième Colloque de Géométrie Algébrique, Liège, 1952*. Georges Thone, Liège, 1952, pp. 129–144.

[115] Hanrot, G., Quercia, M., and Zimmermann, P. The middle product algorithm, I. Speeding up the division and square root of power series. Tech. rep., RR INRIA 4664, 2002.

[116] Hasse, H., and Witt, E. Zyklische unverzweigte Erweiterungskörper vom primzahlgrade $p$ über einem algebraischen Funktionenkörper der Charakteristik $p$. *Monatsch. Math. Phys. 43* (1936), 477–492.

[117] Heine, E. Untersuchungen über die Reihe $1+\frac{(1-q^\alpha)(1-q^\beta)}{(1-q)(1-q^\gamma)}\cdot x+\frac{(1-q^\alpha)(1-q^{\alpha+1})(1-q^\beta)(1-q^{\beta+1})}{(1-q)(1-q^2)(1-q^\gamma)(1-q^{\gamma+1})}\cdot x^2+\dots$. *J. reine angew. Math. 34* (1847), 285–328.

[118] Henrici, P. *Applied and computational complex analysis. Vol. 3*. John Wiley & Sons Inc., New York, 1986. Discrete Fourier analysis—Cauchy integrals—construction of conformal maps—univalent functions, A Wiley-Interscience Publication.

[119] Hopcroft, J., and Musinski, J. Duality applied to the complexity of matrix multiplication and other bilinear forms. *SIAM Journal on Computing 2* (1973), 159–173.

[120] Hopcroft, J. E., and Kerr, L. R. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM Journal on Applied Mathematics 20*, 1 (1971), 30–36.

[121] Huang, X., and Pan, V. Y. Fast rectangular matrix multiplication and applications. *J. Complexity 14*, 2 (Jun 1998), 257–299.

[122] Ince, E. L. *Ordinary Differential Equations*. New York: Dover Publications, 1956.

[123] Kaltofen, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation 64*, 210 (1995), 777–806.

[124] Kaltofen, E., Corless, R. M., and Jeffrey, D. J. Challenges of symbolic computation: my favorite open problems. *Journal of Symbolic Computation 29*, 6 (2000), 891–919.

[125] Kaltofen, E., and Pan, V. Y. Parallel solution of Toeplitz and Toeplitz-like linear systems over fields of small positive characteristic. In *Proceedings of PASCO '94*, vol. 5 of *Lecture Notes Ser. Comput.* World Sci. Publishing, River Edge, NJ, 1994, pp. 225–233.

[126] Kaltofen, E., and Shoup, V. Subquadratic-time factoring of polynomials over finite fields. *Mathematics of Computation 67*, 223 (1998), 1179–1197.

[127] Kaltofen, E., and Yagati, L. Improved sparse multivariate polynomial interpolation algorithms. In *Proceedings of ISSAC'88*, P. Gianni, Ed., vol. 358 of *LNCS*. Springer-Verlag, 1989, pp. 467–474.

[128] Kaminski, M., Kirkpatrick, D. G., and Bshouty, N. H. Addition requirements for matrix and transposed matrix products. *Journal of Algorithms 9*, 3 (1988), 354–364.

[129] Karatsuba, A., and Offman, Y. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady 7* (1963), 595–596.

[130] Karp, A. H., and Markstein, P. High-precision division and square root. *ACM Transactions on Mathematical Software 23*, 4 (1997), 561–589.

[131] Kedlaya, K. Countimg points on hyperelliptic curves using Monsky-Washnitzer. *J. Ramanujan Math. Soc. 16* (2001), 323–338.

[132] Keller-Gehrig, W. Fast algorithms for the characteristic polynomial. *Theoretical Computer Science 36*, 2-3 (1985), 309–317.

[133] Knuth, D. E. *The art of computer programming. Vol. 2: Seminumerical algorithms.* Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont, 1969.

[134] Knuth, D. E. Johann Faulhaber and sums of powers. *Mathematics of Computation 61* (1993), 277–294.

[135] Kobayashi, H., Moritsugu, S., and Hogan, R. W. On solving systems of algebraic equations. In *Proceedings of ISSAC 88* (1988), no. 358 in LNCS, Springer-Verlag, pp. 139–149.

[136] Kreuzer, M., and Robbiano, L. *Computational commutative algebra. 1.* Springer-Verlag, Berlin, 2000.

[137] Kronecker, L. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *J. Reine Angew. Math. 92* (1882), 1–122.

[138] Kung, H. T. On computing reciprocals of power series. *Numerische Mathematik 22* (1974), 341–348.

[139] Kunz, E. *Kähler differentials.* Vieweg advanced lectures in Mathematics. Friedr. Vieweg & Sohn, Braunschweig, 1986.

[140] Kurosh, A. *Cours d'algèbre supérieure*. Éditions Mir, Moscou, 1973.

[141] Lakshman, Y. N., and Lazard, D. On the complexity of zero-dimensional algebraic systems. In *Effective methods in algebraic geometry*, vol. 94 of *Progress in Mathematics*. Birkhäuser, 1991, pp. 217–225.

[142] Lang, S. *Introduction to algebraic geometry*. Interscience Publishers, New York, 1958.

[143] Lang, S. *Algebra*, third ed., vol. 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2002.

[144] Lascoux, A. La résultante de deux polynômes. In *Séminaire d'algèbre Paul Dubreil et Marie-Paule Malliavin, 37ème année (Paris, 1985)*, vol. 1220 of *LNM*. Springer, Berlin, 1986, pp. 56–72.

[145] Lazard, D. Solving zero-dimensional algebraic systems. *Journal of Symbolic Computation 13* (1992), 117–133.

[146] Lazard, D., and Valibouze, A. Computing subfields: reverse of the primitive element problem. In *Computational algebraic geometry (Nice, 1992)*. Birkhäuser Boston, Boston, MA, 1993, pp. 163–176.

[147] Le Verrier, U. J. J. Sur les variations séculaires des éléments elliptiques des sept planètes principales : Mercure, Venus, La Terre, Mars, Jupiter, Saturne et Uranus. *J. Math. Pures Appli. 4* (1840), 220–254.

[148] Lecerf, G. An ultrafast algorithm for Padé-Hermite approximants. Personal communication, 2001.

[149] Lecerf, G. *Une alternative aux méthodes de réécriture pour la résolution des systèmes algébriques*. PhD thesis, École polytechnique, 2001.

[150] Lecerf, G. Quadratic Newton iteration for systems with multiplicity. *Journal of FoCM 2*, 3 (2002), 247–293.

[151] Lecerf, G. Computing the equidimensional decomposition of an algebraic closed set by means of lifting fibers. *Journal of Complexity 19* (2003), 564–596.

[152] Lecerf, G., and Schost, É. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research 5*, 1 (2003), 1–10.

[153] Lenstra, A. K., Lenstra, H. W., and Lovász, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261* (1982), 515–534.

[154] Lickteig, T., and Roy, M.-F. Cauchy index computation. *Calcolo 33*, 3-4 (1996), 337–351. Toeplitz matrices: structures, algorithms and applications (Cortona, 1996).

[155] Lickteig, T., and Roy, M.-F. Sylvester-Habicht sequences and fast Cauchy index computation. *Journal of Symbolic Computation 31*, 3 (2001), 315–341.

[156] Lipson, J. D. Chinese remainder algorithm and interpolation algorithms. In *Proceedings 2nd ACM Symposium of Symbolic and Algebraic Manipulation* (1971), S. R. Petrick, Ed., ACM Press, pp. 372–391.

[157] Lipson, J. D. Newton's Method: A Great Algebraic Algorithm. In *Proceedings ACM Symposium of Symbolic and Algebraic Computation* (1976), ACM Press, pp. 260–270.

[158] Little, J. A key equation and the computation of error values for codes from order domains. Available at http://arXiv.org/math.AC/0303299, 2003.

[159] Loos, R. Computing in algebraic extensions. In *Computer algebra*. Springer, Vienna, 1983, pp. 173–187.

[160] Macaulay, F. S. *The Algebraic Theory of Modular Systems*. Cambridge University Press, 1916.

[161] Mallat, S. Foveal detection and approximation for singularities. *Applied and Computational Harmonic Analysis* (2003). To appear.

[162] Man, Y.-K., and Wright, F. J. Fast polynomial dispersion computation and its application to indefinite summation. In *Proceedings of ISSAC'94* (1994), ACM Press, pp. 175–180.

[163] Manin, J. I. The Hasse-Witt matrix of an algebraic curve. *Trans. Amer. Math. Soc. 45* (1965), 245–264.

[164] Marinari, M. G., Mora, T., and Möller, H. M. Gröbner bases of ideals defined by functionals with an application to ideals of projective points. *Applicable Algebra in Engineering, Communication and Computing 4*, 103–145 (1993).

[165] Massey, J. L. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory IT-15* (1969), 122–127.

[166] Matsuo, K., Chao, J., and Tsujii, S. An improved baby step giant step algorithm for point counting of hyperelliptic curves over finite fields. In *Proceedings of ANTS-V* (2002), C. Fiecker and D. Kohel, Eds., vol. 2369 of *LNCS*, Springer-Verlag, pp. 461–474.

[167] Meunier, L., and Salvy, B. ESF: An automatically generated encyclopedia of special functions. In *Proceedings of ISSAC'03* (2003), ACM Press, pp. 199–205.

[168] Moenck, R., and Allen, J. Efficient division algorithms in Euclidean domains. Tech. Rep. CS-75-18, Univ. of Waterloo, 1975.

[169] Moenck, R. T., and Borodin, A. Fast modular transforms via division. *Thirteenth Annual IEEE Symposium on Switching and Automata Theory (Univ. Maryland, College Park, Md., 1972)* (1972), 90–96.

[170] Moenck, R. T., and Carter, J. H. Approximate algorithms to derive exact solutions to systems of linear equations. In *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*, vol. 72 of *LNCS*. Springer, Berlin, 1979, pp. 65–73.

[171] Montgomery, P. L. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, University of California, Los Angeles CA, 1992.

[172] Mourrain, B. Isolated points, duality and residues. *Journal of Pure and Applied Algebra 117/118* (1997), 469–493. Algorithms for algebra (Eindhoven, 1996).

[173] Mourrain, B., and Pan, V. Y. Solving special polynomial systems by using structured matrices and algebraic residues. In *Foundations of computational mathematics (Rio de Janeiro, 1997)*. Springer, Berlin, 1997, pp. 287–304.

[174] Mourrain, B., and Pan, V. Y. Asymptotic acceleration of solving multivariate polynomial systems of equations. In *Proceedings of STOC'88* (1998), ACM Press, pp. 488–496.

[175] Mourrain, B., and Pan, V. Y. Multivariate polynomials, duality, and structured matrices. *Journal of Complexity 16*, 1 (2000), 110–180.

[176] Mourrain, B., Pan, V. Y., and Ruatta, O. Accelerated solution of multivariate polynomial systems of equations. *SIAM Journal on Computing 32*, 2 (2003), 435–454.

[177] Munro, I. Problems related to matrix multiplication. In *Courant Institute Symposium on Computational Complexity* (1973), R. Rustin, Ed., Algorithmics Press, New York, pp. 137–152.

[178] Nakai, Y. High order derivations. I. *Osaka Journal of Mathematics 7* (1970), 1–27.

[179] Newton, I. *La méthode des fluxions et des suites infinies*. Librairie Scientifique Albert Blanchard, 1966. 150 p.

[180] Oberst, U. The construction of Noetherian operators. *Journal of Algebra 222*, 2 (1999), 595–620.

[181] Osborn, H. Modules of differentials. II. *Mathematische Annalen 175* (1968), 146–158.

[182] Pan, V. Y. *How to multiply matrices faster*. No. 179 in LNCS. Springer, 1984.

[183] Pan, V. Y. Computing the determinant and the characteristic polynomial of a matrix via solving linear systems of equations. *Information Processing Letters 28* (1988), 71–75.

[184] Pan, V. Y. On computations with dense structured matrices. In *Proceedings of ISSAC'89* (1989), ACM Press, pp. 34–42.

[185] Pan, V. Y. Parallel least-squares solution of general and Toeplitz-like linear systems. In *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architecture* (1990), ACM Press, pp. 244–253.

[186] Pan, V. Y. Parallel computation of polynomial GCD and some related parallel computations over abstract fields. *Theoretical Computer Science 162*, 2 (1996), 173–223.

[187] Pan, V. Y. Faster solution of the key equation for decoding BCH error-corecting codes. In *Proceedings of STOC'97* (1997), ACM Press, pp. 168–175.

[188] Pan, V. Y. New techniques for the computation of linear recurrence coefficients. *Finite Fields and their Applications 6*, 1 (2000), 93–118.

[189] Pan, V. Y. *Structured matrices and polynomials*. Birkhäuser Boston Inc., Boston, MA, 2001. Unified superfast algorithms.

[190] Paterson, M. S., and Stockmeyer, L. J. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing 2*, 1 (Mar. 1973), 60–66.

[191] Paule, P. Greatest factorial factorization and symbolic summation. *Journal of Symbolic Computation 20*, 3 (1995), 235–268.

[192] Penfield, Jr., P., Spence, R., and Duinker, S. *Tellegen's theorem and electrical networks*. The M.I.T. Press, Cambridge, Mass.-London, 1970.

[193] Petkovšek, M. Hypergeometric solutions of linear recurrences with polynomial coefficients. *Journal of Symbolic Computation 14*, 2-3 (1992), 243–264.

[194] Pettorossi, A. Derivation of an $O(k^2 \log\ n)$ algorithm for computing order-$k$ Fibonacci numbers from the $O(k^3 \log\ n)$ matrix multiplication method. *Information Processing Letters 11*, 4-5 (1980), 172–179.

[195] Pollard, J. M. Theorems on factorization and primality testing. *Proc. Cambridge Philos. Soc. 76* (1974), 521–528.

[196] Poole, E. G. C. *Introduction to the Theory of Linear Differential Equations*. New York: Dover, 1960.

[197] Probert, R. On the complexity of matrix multiplication. Tech. Rep. CS-73-27, Univ. of Waterloo, 1973.

[198] Rabiner, L. R., Schafer, R. W., and Rader, C. M. The chirp $z$-transform algorithm and its application. *Bell System Tech. J. 48* (1969), 1249–1292.

[199] Reischert, D. Asymptotically fast computation of subresultants. In *Proceedings of ISSAC'97* (1997), ACM Press, pp. 233–240.

[200] Rifà, J., and Borrell, J. Improving the time complexity of the computation of irreducible and primitive polynomials in finite fields. In *Proceedings of AAECC-9* (1991), vol. 539, pp. 352–359.

[201] Roman, S. *The umbral calculus*, vol. 111 of *Pure and Applied Mathematics*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], New York, 1984.

[202] Rothe, H. A. Formulae de serierum reversione demonstratio universalis signis localibus combinatorico-analyticorum vicariis exhibita. Leipzig, 1793.

[203] Rouillier, F. *Algorithmes efficaces pour l'étude des zéros réels des systèmes polynomiaux*. PhD thesis, Université de Rennes I, may 1996.

[204] Rouillier, F. Solving zero-dimensional systems through the Rational Univariate Representation. *Applicable Algebra in Engineering, Communication and Computing 9*, 5 (1999), 433–461.

[205] Saito, M., Sturmfels, B., and Takayama, N. *Gröbner deformations of hypergeometric differential equations*. Springer-Verlag, Berlin, 2000.

[206] Samuel, P. *Théorie algébrique des nombres*. Hermann, 1971.

[207] Scheja, G., and Storch, U. Über Spurfunktionen bei vollständigen Durchschnitten. *J. Reine Angew. Math. 278-279* (1975), 174–190.

[208] Schoenberg, I. J. On polynomial interpolation at the points of a geometric progression. *Proc. Roy. Soc. Edinburgh Sect. A 90*, 3-4 (1981), 195–207.

[209] Schönhage, A. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Numerische Mathematik 20* (1973), 409–417.

[210] Schönhage, A. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica 7* (1977), 395–398.

[211] Schönhage, A. Partial and total matrix multiplication. *SIAM Journal on Computing 10* (1981), 434–455.

[212] Schönhage, A. The fundamental theorem of algebra in terms of computational complexity. Tech. rep., Univ. Tübingen, 1982. 73 pages.

[213] Schönhage, A. Fast parallel computation of characteristic polynomials by Leverrier's power sum method adapted to fields of finite characteristic. In *Automata, languages and programming (Lund, 1993)*, vol. 700 of *LNCS*. Springer, Berlin, 1993, pp. 410–417.

[214] Schönhage, A., and Strassen, V. Schnelle Multiplikation großer Zahlen. *Computing 7* (1971), 281–292.

[215] Schoof, R. Elliptic curves over finite fields and the computation of square roots mod $p$. *Mathematics of Computation 44* (1985), 483–494.

[216] Schost, É. *Sur la résolution des systèmes polynomiaux à paramètres.* PhD thesis, École polytechnique, 2000.

[217] Schost, É. *Résolution des systèmes polynomiaux.* École des Jeunes Chercheurs en Algorithmique et Calcul formel, Université de Marne-la-Vallée, 2003, pp. 237–277.

[218] Schwartz, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the Association for Computing Machinery 27*, 4 (1980), 701–717.

[219] Shafer, R. E. On quadratic approximation. *SIAM J. Numer. Anal. 11* (1974), 447–460.

[220] Shafer, R. E. On quadratic approximation. II. *Univ. Beograd. Publ. Elektrotehn. Fak. Ser. Mat. Fiz.*, 602-633 (1978), 163–170 (1979).

[221] Shortt, J. An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations. *Information Processing Letters 7*, 6 (1978), 299–303.

[222] Shoup, V. New algorithms for finding irreducible polynomials over finite fields. *Mathematics of Computation 54*, 189 (1990), 435–447.

[223] Shoup, V. A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In *Proceedings of ISSAC'91* (1991), ACM Press, pp. 14–21.

[224] Shoup, V. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation 17*, 5 (1994), 371–391.

[225] Shoup, V. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation 20*, 4 (1995), 363–397.

[226] Shoup, V. NTL: A library for doing number theory. `http://www.shoup.net`, 1996–2003.

[227] Shoup, V. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceedings of ISSAC'99* (New York, 1999), ACM Press, pp. 53–58.

[228] Sieveking, M. An algorithm for division of powerseries. *Computing 10* (1972), 153–156.

[229] Spencer, M. *Polynomial real root finding in Bernstein form.* PhD thesis, Dept. Civil Eng., Brigham Young University, 1994.

[230] Stirling, J. *Methodus Differentialis: sive Tractatus de Summatione et Interpolatione Serierum Infinitarum.* Gul. Bowyer, London, 1730. English translation by Holliday, J. The Differential Method: A Treatise of the Summation and Interpolation of Infinite Series. 1749.

[231] Stöhr, K.-O., and Voloch, J. A formula for the Cartier operator on plane algebraic curves. *J. Reine Angew. Math. 377* (1987), 49–64.

[232] Storjohann, A. *Algorithms for matrix canonical forms.* PhD thesis, Department of Computer Science, ETH, Zurich, 2000.

[233] Storjohann, A. High-order lifting. In *Proceedings of ISSAC'02* (2002), ACM Press, pp. 246–254.

[234] Strassen, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356.

[235] Strassen, V. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numerische Mathematik 20* (1972/73), 238–251.

[236] Strassen, V. Vermeidung von Divisionen. *J. Reine Angew. Math. 264* (1973), 184–202.

[237] Strassen, V. Einige Resultate über Berechnungskomplexität. *Jber. Deutsch. Math.-Verein. 78*, 1 (1976/77), 1–8.

[238] Takahashi, D. A fast algorithm for computing large Fibonacci numbers. *Information Processing Letters 75*, 6 (2000), 243–246.

[239] Tellegen, B. A general network theorem, with applications. *Philips Research Reports 7* (1952), 259–269.

[240] Thiong Ly, J.-A. Note for computing the minimum polynomial of elements in large finite fields. In *Coding theory and applications (Toulon, 1988)*, vol. 388 of *LNCS*. Springer, New York, 1989, pp. 185–192.

[241] Thomé, É. Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In *Proceedings of ISSAC'01* (2001), ACM Press, pp. 323–331.

[242] Thomé, É. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of Symbolic Computation 33*, 5 (2002), 757–775.

[243] Tsai, H. *Algorithms for algebraic analysis.* PhD thesis, University of California at Berkeley, Spring 2000.

[244] Valibouze, A. Fonctions symétriques et changements de bases. In *Proc. EUROCAL–87* (1989), vol. 378 of *LNCS*, pp. 323–332.

[245] Van Barel, M., and Bultheel, A. A general module-theoretic framework for vector M-Padé and matrix rational interpolation. *Numerical Algorithms 3*, 1-4 (1992), 451–461. Extrapolation and rational approximation (Puerto de la Cruz, 1992).

[246] van der Hoeven, J. Fast evaluation of holonomic functions near and in regular singularities. *Journal of Symbolic Computation 31*, 6 (2001), 717–743.

[247] VAN DER HOEVEN, J. FFT-like multiplication of linear differential operators. *Journal of Symbolic Computation 33*, 1 (2002), 123–127.

[248] VAN DER HOEVEN, J. Relax, but don't be too lazy. *Journal of Symbolic Computation 34*, 6 (2002), 479–542.

[249] VAN DER POORTEN, A. J. Some facts that should be better known, especially about rational functions. In *Number theory and applications (Banff, AB, 1988)*, vol. 265 of *NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci.* Kluwer Acad. Publ., Dordrecht, 1989, pp. 497–528.

[250] VAN HOEIJ, M. Factoring polynomials and the knapsack problem. *Journal of Number Theory 95*, 2 (2002), 167–189.

[251] VILLARD, G. Computing Popov and Hermite forms of polynomial matrices. In *Proceedings of ISSAC'96* (1996), ACM Press, pp. 251–258.

[252] VON HAESELER, F., AND JÜRGENSEN, W. Irreducible polynomials generated by decimations. In *Finite fields and their applications (Augsburg, 1999)*. Springer, Berlin, 2001, pp. 224–231.

[253] VON ZUR GATHEN, J. Functional decomposition of polynomials: the wild case. *Journal of Symbolic Computation 10*, 5 (1990), 437–452.

[254] VON ZUR GATHEN, J., AND GERHARD, J. Fast algorithms for Taylor shifts and certain difference equations. In *Proceedings of ISSAC'97* (New York, 1997), ACM Press, pp. 40–47.

[255] VON ZUR GATHEN, J., AND GERHARD, J. *Modern computer algebra*. Cambridge University Press, New York, 1999.

[256] VON ZUR GATHEN, J., AND SHOUP, V. Computing Frobenius maps and factoring polynomials. *Computational Complexity 2*, 3 (1992), 187–224.

[257] WAN, D. Computing zeta functions over finite fields. *Contemp. Math. 225* (1999), 131–141.

[258] WIEDEMANN, D. Solving sparse linear equations over finite fields. *IEEE Transactions on informations theory IT-32* (1986), 54–62.

[259] WILF, H. S., AND ZEILBERGER, D. An algorithmic proof theory for hypergeometric (ordinary and "$q$") multisum/integral identities. *Inventiones Mathematicae 108* (1992), 575–633.

[260] WILSON, T. C., AND SHORTT, J. An $O(\log\ n)$ algorithm for computing general order-$k$ Fibonacci numbers. *Information Processing Letters 10*, 2 (1980), 68–75.

[261] WINOGRAD, S. A new algorithm for inner products. *IEEE Trans. Comp.* (1968), 693–694.

[262] Winograd, S. On multiplication of $2 \times 2$ matrices. *Linear Algebra and Appl. 4* (1971), 381–388.

[263] Yokoyama, K., Li, Z., and Nemes, I. Finding roots of unity among quotients of the roots of an integral polynomial. In *Proceedings of ISSAC'95* (1995), ACM Press, pp. 85–89.

[264] Yui, N. On the Jacobian varietes of hyperelliptic curves over fields of characteristic $p > 2$. *J. Algebra 52* (1978), 378–410.

[265] Zeilberger, D. A holonomic systems approach to special functions identities. *Journal of Computational and Applied Mathematics 32*, 3 (1990), 321–368.

[266] Zippel, R. Probabilistic algorithms for sparse polynomials. In *Symbolic and algebraic computation* (Berlin, 1979), no. 72 in LNCS, Springer, pp. 216–226. Proceedings EUROSAM '79, Marseille, 1979.

[267] Zippel, R. Interpolating polynomials from their values. *Journal of Symbolic Computation 9*, 3 (1990), 375–403.