Efficient computation with structured matrices and arithmetic expressions

Christophe Mouilleron

▶ **To cite this version:**

Christophe Mouilleron. Efficient computation with structured matrices and arithmetic expressions. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2011. English. NNT : 2011ENSL0652 . tel-00688388

HAL Id: tel-00688388
https://theses.hal.science/tel-00688388

Submitted on 17 Apr 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

Christophe Mouilleron

pour l'obtention du grade de

Docteur de l'Université de Lyon – École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'École Doctorale de mathématiques et d'informatique fondamentale de Lyon

Efficient computation with structured matrices and arithmetic expressions

Directeur de thèse :	Gilles Villard
Co-directeur de thèse :	Claude-Pierre Jeannerod
Après avis de :	Markus Püschel Lihong Zhi

Devant la commission d'examen formée de :

Dario Bini	Membre
Claude-Pierre Jeannerod	Membre
Bernard Mourrain	Membre
Markus Püschel	Membre/Rapporteur
Gilles Villard	Membre
Lihong Zhi	Membre/Rapporteur

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Today, computers are heavily used to carry out all sorts of computations. The question of designing efficient code is thus crucial, and it occurs at several levels. First, it is important to design efficient algorithms, relying on the appropriate data structures. Second, one needs to turn algorithms into programs, which typically implies to make several implementation choices, that may strongly impact the running time in practice. Third, the compiler should provide further optimizations, depending on the target architecture.

This thesis addresses two situations where we aim at efficient code. First, we will focus on computations involving dense structured matrices. Such matrices appear frequently in computer algebra, in coding theory for error correction, and in signal and image processing. In this case, the structure implies that the matrices, while being dense, can be stored more efficiently than an arbitrary dense matrix. This allows for the design of fast algorithms, like those that we will present in Part I. Second, we present some work on the evaluation of arithmetic expressions in Part II. Implementing efficiently the evaluation of a given arithmetic expression on a target architecture may be quite difficult, all the more as the architectures tend to become more complex. Optimization of one or several criteria (like latency or numerical accuracy) may require to explore a set of possible implementations in order to extract the relevant ones. Part II presents a general framework that allows for such a search process, as well as other analyses that can be used to gain some insight into the properties of the set of implementations.

Part I – Improving computations with structured matrices

In many cases, the structure of a matrix A, when there is one, can be exposed by applying an appropriate linear map $\mathcal{L}$ to it so that the rank α of the resulting matrix $\mathcal{L}(\mathsf{A})$ is small, in a sense that depends on the context. Then, A can be represented by two matrices G and H having only α columns and such that $\mathcal{L}(\mathsf{A}) = \mathsf{GH}^T$. Classical choices for displacement operator $\mathcal{L}$ are Sylvester's operator $\nabla[\mathsf{M},\mathsf{N}] : \mathsf{A} \mapsto \mathsf{MA} - \mathsf{AN}$ and Stein's operator $\Delta[\mathsf{M},\mathsf{N}] : \mathsf{A} \mapsto \mathsf{A} - \mathsf{MAN}$. By taking M and N among diagonal matrices, and unit φ-circulant matrices and their transposes, several well-known types of matrices are covered: Toeplitz, Hankel, Vandermonde, and Cauchy matrices. Moreover, using this displacement technique, generalizations of these matrices are easily defined. The main point with these structured matrices is that, since the generator (G,H) of A has only $\alpha(m+n)$ elements instead of mn, one can achieve several operations faster than with dense, unstructured matrices, like multiplication by a vector, transposition, addition or multiplication of structured matrices, and inversion. Note that, for all the above operations whose result is a matrix, it is known that this matrix is structured, and so it is also represented using a generator.

In Part I, we first focus on the problem of inverting a structured matrix: given a generator (G, H) for a regular matrix A, compute efficiently a generator (Y, Z) for A^{-1}. Subsequently, since fast inversion algorithms rely on the problem of multiplying a structured matrix of displacement rank α by α vectors, we also consider the slightly more general problem of computing the multiplication of a structured matrix by a matrix.

Main contributions

Compression-free algorithms for inversion. One issue with the classical divide-and-conquer algorithm for structured matrix inversion lies in the need to control the size of the generators for intermediate quantities. This is usually achieved thanks to so-called *compression steps*, whose purpose is to reduce the size of computed generators having a larger size than expected. We propose in this thesis a general, compression-free algorithm for inversion, that extends the algorithm presented in [Car99, Car00] for Cauchy-like matrices. Here, the control of the size of the generators is achieved thanks to explicit recursive formulas, which give already compressed generators. Then, we deduce an algorithm for Cauchy- and Vandermonde-like matrices, that we extend to cover Hankel-like matrices. Even if suppressing compression steps does not yield a better asymptotic cost, this leads to small dominant terms and thus allows for speed-ups up to a factor of about 7 compared to the classical approach, both in theory and in practice.

Algorithms for fast multiplication of a structured matrix by a matrix. An algorithm for the fast multiplication of an $n \times n$ Toeplitz-like matrix A of displacement rank α by an $n \times \alpha$ matrix B is proposed in [Bos10, page 210] as an improvement of [BJS07, BJS08]. In this thesis, we extend this approach to Toeplitz-, Cauchy-, and Vandermonde-like matrices in the rectangular case $m \times n$. Moreover, we analyze the cost of this approach in the case where B has β columns, for some arbitrary positive integer β.

Outline of Part I

The first part of this document is organized into three chapters as follows.

Chapter 1 – Computing with structured matrices. This chapter serves as an introduction for this part. It starts with a short reminder on matrix arithmetic, before introducing several types fo matrices for which basic operations like multiplication by a vector can be carried out in quasi-linear time using polynomial arithmetic. Then, we present the basics of the displacement rank approach, which aims at providing a general framework that covers many matrices like for instance Toeplitz, Vandermonde, and Cauchy matrices. Next, we list the main properties of these structured matrices, and, in particular, issues related to structured matrix inversion are discussed. We conclude this first chapter with the detailed list of our contributions to the domain of structured matrices.

Chapter 2 – Compression-free inversion of structured matrices. This chapter presents an extension of an algorithm for Cauchy-like matrix inversion by Cardinal [Car99, Car00] to a broader class of structured matrices. Two new inversion algorithms are introduced, one covering both the Cauchy-like and Vandermonde-like structures, and one dedicated to the Hankel-like structure. A detailed cost analysis is provided for these three cases. Finally, experimental results confirm this analysis and show that our new approach is up to 6.7 times faster than the classical inversion algorithm in practice.

Chapter 3 – Fast multiplication of a structured matrix by a matrix. In this chapter, we tackle the problem of multiplying a structured matrix A by an matrix B. A recent work in [Bos10, page 210] proposes an asymptotically fast algorithm in the case of square Toeplitz-like matrices, that we extend here to several other structures. Thus, we give a general polynomial interpretation for the product AB, and we show how to solve efficiently the polynomial problem that appears from this interpretation by extending the algorithm of [Bos10, page 210]. Finally, we obtain an asymptotically fast, general algorithm for the multiplication of a structured matrix by a matrix, that we can use together with the material from the previous chapter to obtain fast inversion algorithms in practice.

Part II – Analyzing the implementations of arithmetic expressions

Arithmetic expressions are formulas made of sums and products of variables. Implementing efficiently such expressions for a given architecture raises several questions. When parallelism is available, one should use a mathematical writing of the expression which exposes as much parallelism as possible. On the other hand, when computations are carried out using finite-precision arithmetic, one may want to analyze the numerical properties of the computed result, and to show that the error entailed by the successive roundings is below some given bound. One example of a situation where both questions arise is the design of efficient code for the evaluation of a polynomial approximation within a library of mathematical functions (libm). One may also come with more theoretical questions, like counting the number of implementations or determining the minimum number of operations required to evaluate some arithmetic expression in a given computational model. The main challenge addressed by Part II is to design and implement a general framework allowing to provide answers to such questions.

Main contributions

Framework for analyzing the implementations of a given arithmetic expression. Generalizing the generation algorithm in [Rev09, §6.1], we propose to model implementations by the notion of evaluation scheme. Provided we have an algorithmic way to deduce how a given arithmetic expression can split into smaller ones, we can analyze in a divide-and-conquer manner the set of all its evaluation schemes. This approach yields three types of analyses depending on the problem we want to tackle: generation, counting, and optimization.

Applications to code generation for polynomial evaluation. The aforementioned framework is used in two situations involving polynomial evaluations. First, it allows us to significantly improve a code generator for fast and accurate-enough polynomial evaluation on VLIW architectures, that is used in the context of a libm. In particular, we measure an average gain of about 50% in the overall generation time. Second, we address the issue of evaluating a polynomial with scalar coefficients at a matrix point with a minimum number of matrix-matrix multiplications. The explorative aspect of our framework is then used to study the set of optimal schemes, and to deduce the optimality of some known schemes for small polynomial degrees.

Outline of Part II

The second part of this document is organized into five chapters as follows.

Chapter 4 – On the evaluation of arithmetic expressions. This chapter serves as an introduction to our work on the evaluation of arithmetic expressions. The implementation of a given arithmetic expression raises several questions in various fields of computer science. Our initial context, that is, the implementation of floating-point operators for integer VLIW processors based on polynomial approximation, also provides several questions, some of them being classical, while others are more specific. We first present these issues along with some background material. Then we conclude this fourth chapter with the detailed list of our contributions to the domain of arithmetic expression evaluation.

Chapter 5 – How to model and analyze implementations of arithmetic expressions. This chapter introduces a way to model and analyze the different implementations of an arithmetic expression using *evaluation schemes* and *decompositions*. Together, they allow for the design of divide-and-conquer analysis algorithms. This approach is first illustrated with the description of an exhaustive generator of evaluation schemes. Furthermore, a way to model optimization criteria is discussed, and the previous generator is extended so that only the schemes achieving a value below a given threshold for some criterion are generated.

Chapter 6 – On the combinatorics of evaluation schemes. This chapter focuses on combinatorics issues related to sets of evaluation schemes. We will see how to count the total number of schemes, and how to get the distribution of these schemes according to some given criterion. Furthermore, we will present an asymptotic study of several sequences that appear when counting the number of schemes for some parameterized families of expressions.

Chapter 7 – Optimization. This chapter is dedicated to optimization issues. We first introduce a simple optimization algorithm that works well with some criteria like the latency on unbounded parallelism. Then, several more complex algorithms are given in order to tackle more optimization problems, like the minimization of the number of multiplications. Finally, we address the question of multicriteria optimization and, in particular, we propose an algorithm that computes the best trade-offs between two given criteria, like latency and accuracy.

Chapter 8 – Application examples. Finally, we detail here two concrete situations that benefit from the material presented so far in this second part of the thesis. First, our approach leads us to review the architecture of CGPE, a software tool for generating fast and accurate-enough codes for polynomial evaluation on a VLIW architecture. By adding more constraints within the computation of the schemes to be further analyzed by the tool, we obtain significant speed-ups for the whole code generation. Second, we tackle the issue of minimizing the number of matrix-matrix multiplication in the evaluation of $p(\mathsf{A})$ where p is a univariate polynomial and A is an $n \times n$ matrix. In this case, our approach allows us both to automatically retrieve known schemes due to Paterson and Stockmeyer [PS73] and, in the context of the matrix exponential, due to Higham [Hig08, §10], and to prove their optimality for small degrees of p.

Part I

Improving computations with structured matrices

Chapter 1

Computing with structured matrices

This chapter serves as an introduction for the first part of this document. After a short reminder on matrix arithmetic in Section 1.1, we examine in Section 1.2 several special matrices, for which the problems of multiplication by a vector and linear system solving can be solved in quasi-linear time using polynomial arithmetic. Then, we introduce in Section 1.3 the basics of the structured matrix theory, which is a general framework that covers all the special matrices listed in Section 1.2, along with generalizations of them. Next, we explain in Section 1.4 how to perform basic operations on structured matrices, like multiplication by a vector, transposition, addition or multiplication, and inversion. Most of the material presented in these first four sections is taken from [vzGG03] and [BP94, Pan01]. Finally, in Section 1.5, we discuss issues on structured matrix inversion and on the multiplication of a structured matrix by a matrix, and we detail our contributions to this domain.

1.1 Preliminaries on dense matrices

This section aims at recalling a few results about dense matrix multiplication and block Gaussian elimination. Matrices considered in this first part of the document will have coefficients in a field $\mathbb{K}$, and the set of the $m \times n$ matrices with coefficients in $\mathbb{K}$ will be denoted by $\mathbb{K}^{m\times n}$.

1.1.1 Notation

Here and hereafter, $\mathbb{I}_n$ is the identity matrix of order n, and $\mathbb{J}_n$ is the reflexion matrix of order n, whose (i, j) entry is 1 if $i + j = n + 1$, and 0 otherwise:

$$\mathbb{I}_n = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} \qquad \text{and} \qquad \mathbb{J}_n = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}.$$

We will also write $\mathsf{e}_{n,i}$ for the ith unit vector of $\mathbb{K}^n$, and e_n for the vector of $\mathbb{K}^n$ whose all entries are equal to 1. Moreover, for any matrix A, we denote its (i, j) entry by a_{ij} and its jth column by a_j.

Furthermore, when $\mathsf{A} \in \mathbb{K}^{m\times n}$ and when m_1, m_2, n_1, and n_2 are positive integers such that

$$m = m_1 + m_2 \qquad \text{and} \qquad n = n_1 + n_2,$$

we can partition the matrix A into blocks as

$$\mathsf{A} = \begin{bmatrix} \mathsf{A}_{11} & \mathsf{A}_{12} \\ \mathsf{A}_{21} & \mathsf{A}_{22} \end{bmatrix}, \quad \text{where} \quad \mathsf{A}_{ij} \in \mathbb{K}^{m_i \times n_j}. \tag{1.1}$$

1.1.2 Matrix multiplication

Given two matrices A and B in $\mathbb{K}^{n\times n}$, the naive way to compute the product matrix AB takes n^3 multiplications and $n^3 - n^2$ additions in $\mathbb{K}$. Yet, one can perform this operation asymptotically faster than $O(n^3)$. The first algorithm that achieves a subcubic cost for matrix multiplication is due to Strassen. He proposes in [Str69] an algorithm in $O(n^{\log_2 7})$, where the exponent is $\log_2 7 \approx 2.807$. Several variants of Strassen's algorithm were designed to decrease this exponent (see [vzGG03, Note 12.1] and the references therein), and the most powerful variant up to now is one by Coppersmith and Winograd [CW87], which achieves a complexity in $O(n^{2.376})$.

In this document, we shall write $O(n^\omega)$ for the cost of the multiplication of two $n \times n$ matrices. Coppersmith and Winograd's method shows that one can achieve $\omega \leq 2.376$. While it is not known yet whether the multiplication of two $n\times n$ matrices can be achieved in quasi-quadratic time with respect to n, we will assume in this document that $\omega > 2$.

Matrix multiplication is at the heart of algorithms in linear algebra. As a consequence, many other problems involving $n \times n$ matrices over a field (including matrix inversion, solving a linear system, and computing the determinant or the rank) can be solved at the same asymptotic cost $O(n^\omega)$ as matrix multiplication [IMH82].

1.1.3 Matrix inversion using block Gaussian elimination

Matrix inversion can be achieved thanks to Gaussian elimination with pivoting. We recall here a block version of Gaussian elimination without pivoting, which relies on matrix multiplication and thus achieves a cost in $O(n^\omega)$ provided all the matrices that need to be inverted are nonsingular.

Let $\mathsf{A} \in \mathbb{K}^{n\times n}$ be a nonsingular matrix, partitioned as in (1.1) with $m_1 = n_1$ so that A_{11} is in turn a square matrix. Supposing that A_{11} is also nonsingular, we can define the Schur complement of A_{11} in A, which we shall denote by S:

$$\mathsf{S} = \mathsf{A}_{22} - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{A}_{12}.$$

Let also E and F be the block matrices defined by

$$\mathsf{E} = \begin{bmatrix} \mathbb{I}_{n_1} & \\ -\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathbb{I}_{n_2} \end{bmatrix} \qquad \text{and} \qquad \mathsf{F} = \begin{bmatrix} \mathbb{I}_{n_1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12} \\ & \mathbb{I}_{n_2} \end{bmatrix}.$$

By noting that E and F are nonsingular, and that $\mathsf{EAF} = \left[\begin{smallmatrix} \mathsf{A}_{11} & \\ & \mathsf{S} \end{smallmatrix}\right]$, we see that S is nonsingular if A_{11} and A are nonsingular. Hence, we deduce the following classical recursive factorization of the inverse of A [Pan01, p. 157]:

$$\mathsf{A}^{-1} = \mathsf{F} \begin{bmatrix} \mathsf{A}_{11}^{-1} & \\ & \mathsf{S}^{-1} \end{bmatrix} \mathsf{E}. \tag{1.2}$$

If we want to compute the inverse of A recursively using the above formula, we need the upper-left blocks of A_{11} and S to be in turn invertible, and so forth. The correct hypothesis on A is to assume that it is *strongly regular*,[1] that is, all the square upper-left blocks of A have to be nonsingular. If A is strongly regular then so are A_{11} and S, and (1.2) gives a recursive algorithm to compute A^{-1} whose cost $C(n)$ satisfies

$$C(n) = 2 \cdot C\Big(\frac{n}{2}\Big) + O(n^\omega),$$

which leads to $C(n) \in O(n^\omega)$.

Note that not all nonsingular matrices are strongly regular. In order to invert any nonsingular matrix in $O(n^\omega)$, one can use an inversion algorithm based on LUP decomposition [BH74]. Yet, matrix inversion using block Gaussian elimination plays an important role in our context since it serves as a basis for the inversion of structured matrices (see Section 1.4.3).

1.2 Special matrices and fast polynomial arithmetic

1.2.1 Toeplitz matrices and polynomial multiplication

Consider the following matrix-vector product:

$$\begin{bmatrix} a_0 & & \\ a_1 & a_0 & \\ a_2 & a_1 & a_0 \\ & a_2 & a_1 \\ & & a_2 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_0b_2 + a_1b_1 + a_2b_0 \\ a_1b_2 + a_2b_1 \\ a_2b_2 \end{bmatrix}. \tag{1.3}$$

Let $a(x) = \sum_{i=0}^2 a_i \cdot x^i$ and $b(x) = \sum_{i=0}^2 b_i \cdot x^j$. One can see that the resulting vector is made of the coefficients of the polynomial $c = a \cdot b$. Using naive matrix-vector product, we get the coefficients of c in $O(n^2)$. Yet, supposing that $\mathbb{K} = \mathbb{C}$ for instance, we can compute c using $O(n \log n)$ operations in $\mathbb{C}$ thanks to the FFT (Fast Fourier Transform) algorithm. Actually, not all fields $\mathbb{K}$ support the FFT. However, the multiplication of two polynomials with degree less than n can still be achieved using $O(n \log n \log\log n)$ operations in $\mathbb{K}$ [vzGG03, Theorem 8.22]. This remains asymptotically better than the naive matrix-vector product.

The gain obtained in the previous example reflects the fact that we have a very specific matrix in the matrix-vector product. In fact, this matrix is a special case of Toeplitz matrix:

[1] We may also equally say *strongly nonsingular*.

Definition 1.1. *Let m and n be two positive integers, and $\mathsf{v} \in \mathbb{K}^{m+n-1}$ be a vector indexed from $1-n$ to $m-1$. The Toeplitz matrix $\mathbb{T}(\mathsf{v}, m, n)$ is the $m \times n$ matrix whose entry (i, j) is v_{i-j}. In particular, when $m = n$, we have the following square Toeplitz matrix*

$$\mathbb{T}(\mathsf{v}) = \mathbb{T}(\mathsf{v}, n, n) = \begin{bmatrix} v_0 & v_{-1} & \dots & v_{1-n} \\ v_1 & v_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & v_{-1} \\ v_{n-1} & \dots & v_1 & v_0 \end{bmatrix}.$$

Similarly, given a vector $v \in \mathbb{K}^n$, one can define the $n \times n$ lower-triangular Toeplitz matrix $\mathbb{L}(\mathsf{v})$ and the $n \times n$ upper-triangular Toeplitz matrix $\mathbb{U}(\mathsf{v})$ by

$$\mathbb{L}(\mathsf{v}) = \begin{bmatrix} v_0 & & & \\ v_1 & v_0 & & \\ \vdots & \ddots & \ddots & \\ v_{n-1} & \dots & v_1 & v_0 \end{bmatrix} \quad \text{and} \quad \mathbb{U}(\mathsf{v}) = \begin{bmatrix} v_0 & v_1 & \dots & v_{n-1} \\ & v_0 & \ddots & \vdots \\ & & \ddots & v_1 \\ & & & v_0 \end{bmatrix}.$$

One noticeable property of these triangular Toeplitz matrices is that multiplying one of them by a vector can be achieved using a product of two polynomials. Let $\mathsf{u}, \mathsf{v} \in \mathbb{K}^n$, $u(x) = \sum_{i=0}^{n-1} u_i \cdot x^i$, $\widetilde{u}(x) = \sum_{i=0}^{n-1} u_{n-1-i} \cdot x^i$, and $v(x) = \sum_{i=0}^{n-1} v_i \cdot x^i$. We can indeed deduce by looking at (1.3) that:

- the ith entry of the vector $\mathbb{L}(\mathsf{u})\mathsf{v}$ is the coefficient of x^{i-1} in $u(x)v(x)$,
- the ith entry of the vector $\mathbb{U}(\mathsf{u})\mathsf{v}$ is the coefficient of x^{n-2+i} in $\widetilde{u}(x)v(x)$.

Similarly, if we now have $\mathsf{u} \in \mathbb{K}^{m+n-1}$ and $u(x) = \sum_{i=0}^{m+n-2} u_{i+1-n} \cdot x^i$ then the ith entry of $\mathbb{T}(\mathsf{u}, n, n)\mathsf{v}$ is the coefficient of x^{n-2+i} in $u(x)v(x)$.

Cost of polynomial multiplication and of "(Toeplitz matrix) × vector" product

Following [vzGG03, p. 242], we write $\mathsf{M}(n)$ for the cost of multiplying two polynomials over $\mathbb{K}[x]$ with degree less than n, and we assume that:

- $\mathsf{M}(n)$ is "superlinear," that is, the function $n \mapsto \mathsf{M}(n)/n$ is nondecreasing;
- $\mathsf{M}(n)$ is at most quadratic, that is, for all $n, m \in \mathbb{N}_{>0}$, we have $\mathsf{M}(mn) \leq m^2\mathsf{M}(n)$.

Moreover, the superlinearity property implies that, for all $n, m \in \mathbb{N}_{>0}$:

$$\mathsf{M}(mn) \geq m \cdot \mathsf{M}(n), \quad \mathsf{M}(m+n) \geq \mathsf{M}(m) + \mathsf{M}(n), \quad \text{and} \quad \mathsf{M}(n) \geq n. \tag{1.4}$$

We can now sum up our previous remark:

Property 1.1. *Given $\mathsf{u}, \mathsf{v} \in \mathbb{K}^n$, one can compute $\mathbb{L}(\mathsf{u})\mathsf{v}$ and $\mathbb{U}(\mathsf{u})\mathsf{v}$ using $\mathsf{M}(n)$ operations in $\mathbb{K}$. In addition, given $\mathsf{u} \in \mathbb{K}^{m+n-1}$ and $\mathsf{v} \in \mathbb{K}^n$, one can compute $\mathbb{T}(\mathsf{u}, m, n)\mathsf{v}$ using $\mathsf{M}(m+n)$ operations in $\mathbb{K}$. If $m = n$ then it becomes $\mathsf{M}(2n) \in O(\mathsf{M}(n))$ operations in $\mathbb{K}$.*

Note that, using the techniques in [Mul00], one can actually avoid to compute all the polynomial products mentioned above, thus saving some work. Yet, all the costs remain in $O(\mathsf{M}(n))$.

Linear systems involving Toeplitz matrices

We have just seen that the product of a square Toeplitz matrix by a vector can be achieved faster than in $O(n^2)$ using polynomial multiplication. In fact, a similar conclusion holds for solving linear systems that involve Toeplitz matrices [Pan01, §2.5 and §2.11]:

Property 1.2. *Let $n \in \mathbb{N}_{>0}$ and $\mathsf{u}, \mathsf{v} \in \mathbb{K}^n$ with $u_0 \neq 0$. Matrices $\mathbb{L}(\mathsf{u})$ and $\mathbb{U}(\mathsf{u})$ are then nonsingular, and $\mathbb{L}(\mathsf{u})^{-1}\mathsf{v}$ and $\mathbb{U}(\mathsf{u})^{-1}\mathsf{v}$ can be computed using $O(\mathsf{M}(n))$ operations in $\mathbb{K}$.*

Moreover, let $n \in \mathbb{N}_{>0}$, $\mathsf{u} \in \mathbb{K}^{2n-1}$, and $\mathsf{v} \in \mathbb{K}^n$. If $\mathbb{T}(\mathsf{u})$ is nonsingular then the computation of $\mathbb{T}(\mathsf{u})^{-1}\mathsf{v}$ can be achieved using $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{K}$.

1.2.2 Vandermonde matrices, multipoint evaluation and interpolation

Multipoint evaluation

The problem of multipoint evaluation is the following: Given $\mathsf{x} \in \mathbb{K}^n$ and $p \in \mathbb{K}[x]$ with $\deg(p) = m$, compute $p(x_i)$ for $1 \leq i \leq n$. As stated in [vzGG03, Corollary 10.8], this problem can be solved using $O(\mathsf{M}(n)\log n)$ in $\mathbb{K}$ when $m < n$. Like for polynomial multiplication, we can rewrite this problem into a product of a special matrix by a vector.

Definition 1.2. *Let $\mathsf{x} \in \mathbb{K}^m$. The Vandermonde matrix $\mathbb{V}(\mathsf{x}, n)$ is the $m \times n$ matrix whose entry (i, j) is equal to $\mathsf{x}_i^{(j-1)}$:*

$$\mathbb{V}(\mathsf{x}, n) = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix}.$$

Moreover, when $m = n$, we shall write $\mathbb{V}(\mathsf{x})$ instead of $\mathbb{V}(\mathsf{x}, n)$.

If $\mathsf{x} \in \mathbb{K}^m$, p is a polynomial of degree less than n, and $\mathsf{v} \in \mathbb{K}^n$ denotes the vector whose ith entry is the coefficient of x^{i-1} in p then the problem of multipoint evaluation is equivalent to computing the product $\mathbb{V}(\mathsf{x}, n)\mathsf{v}$. As a consequence, we deduce for $m = n$ that multiplying a square Vandermonde matrix by a vector can be achieved in $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{K}$.

Interpolation

The problem of interpolation is the opposite of multipoint evaluation: Given a repetition-free vector of points $\mathsf{x} \in \mathbb{K}^n$ (that is, $x_i \neq x_j$ for all $i \neq j$) and a vector of values $\mathsf{y} \in \mathbb{K}^n$, find the unique polynomial $p \in \mathbb{K}[x]$ of degree less than n such that $p(x_i) = y_i$ for all $1 \leq i \leq n$. The matricial form for this problem is to find the vector v such that

$$\mathbb{V}(\mathsf{x})\mathsf{v} = \mathsf{y}.$$

Since $x_i \neq x_j$ for $i \neq j$, we have that the Vandermonde matrix $\mathbb{V}(\mathsf{x})$ is invertible, so the solution for the interpolation problem is $\mathsf{v} = \mathbb{V}(\mathsf{x})^{-1}\mathsf{y}$.

Solving this linear system using one of the usual methods for dense matrices would yield a cost in $O(n^\omega)$ operations in $\mathbb{K}$. Yet, interpolation can be achieved in $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{K}$ (see for instance [vzGG03, Corollary 10.12]).

Summary

We can sum up the two facts mentioned above with the following property by:

Property 1.3. *Let* $\mathsf{x}, \mathsf{y} \in \mathbb{K}^n$*. One can compute* $\mathbb{V}(\mathsf{x})\mathsf{y}$ *using* $O(\mathsf{M}(n)\log n)$ *operations in* $\mathbb{K}$*. Moreover, when* x *is repetition-free,* $\mathbb{V}(\mathsf{x})$ *is invertible and one can compute* $\mathbb{V}(\mathsf{x})^{-1}\mathsf{y}$ *using* $O(\mathsf{M}(n)\log n)$ *operations in* $\mathbb{K}$*.*

1.2.3 Other links between special matrices and polynomials

There exist several other special matrices for which some linear algebra operations can be achieved asymptotically faster than for general dense matrices. We cite here two of them: Hankel and Cauchy matrices.

Hankel matrix

Definition 1.3. *Let* m *and* n *be two positive integers, and* $\mathsf{v} \in \mathbb{K}^{m+n-1}$ *be a vector indexed from* $1-n$ *to* $m-1$*. The Hankel matrix* $\mathbb{H}(\mathsf{v}, m, n)$ *is the* $m \times n$ *matrix whose entry* (i,j) *is* $\mathsf{v}_{i+j-1-n}$*. In particular, when* $m = n$*, we have the following square Hankel matrix:*

$$\mathbb{H}(\mathsf{v}, n, n) = \begin{bmatrix} v_{1-n} & \cdots & v_{-1} & v_0 \\ \vdots & \iddots & v_0 & v_1 \\ v_{-1} & \iddots & \iddots & \vdots \\ v_0 & v_1 & \cdots & v_{n-1} \end{bmatrix}.$$

Since $\mathbb{H}(\mathsf{v}, m, n) = \mathbb{T}(\mathsf{v}, m, n)\mathbb{J}_n$, linear algebra operations involving Hankel matrices will essentially have the same cost as their Toeplitz counterpart.

Cauchy matrix

Definition 1.4. *Let* m *and* n *be two positive integers, and let* $\mathsf{x} \in \mathbb{K}^m$ *and* $\mathsf{y} \in \mathbb{K}^n$ *be such that* $x_i \neq y_j$ *for all* (i,j)*. The Cauchy matrix* $\mathbb{C}(\mathsf{x}, \mathsf{y})$ *is the* $m \times n$ *matrix whose entry* (i,j) *is equal to* $\frac{1}{x_i - y_j}$*:*

$$\mathbb{C}(\mathsf{x}, \mathsf{y}) = \begin{bmatrix} \dfrac{1}{x_1 - y_1} & \dfrac{1}{x_1 - y_2} & \cdots & \dfrac{1}{x_1 - y_n} \\ \dfrac{1}{x_2 - y_1} & \dfrac{1}{x_2 - y_2} & \cdots & \dfrac{1}{x_2 - y_n} \\ \vdots & \vdots & & \vdots \\ \dfrac{1}{x_m - y_1} & \dfrac{1}{x_m - y_2} & \cdots & \dfrac{1}{x_m - y_n} \end{bmatrix}.$$

Multiplication of the Cauchy matrix $\mathbb{C}(\mathsf{x}, \mathsf{y})$ by a vector v is related to the multipoint evaluation of the rational function $\sum_{j=1}^{n} \dfrac{v_j}{x - y_j}$ at $x_1, \ldots, x_m$, and it can be achieved when $m = n$ using $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{K}$ [Pan01, page 88]. Moreover, if x and y are two repetition-free vectors of size n then $\mathbb{C}(\mathsf{x}, \mathsf{y})$ is nonsingular and $\mathbb{C}(\mathsf{x}, \mathsf{y})^{-1}\mathsf{v}$ can also be computed using $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{K}$ [Pan01, page 92].

Summary

To conclude this section, we present in Table 1.1 a summary of all the costs of matrix-vector multiplication and linear system solving for all the special matrices we have mentioned so far. We add to it the Vandermonde transposed matrix, for which the costs of multiplication by a vector and linear system solving are also in $O(\mathsf{M}(n) \log n)$ (see [vzGS92, Theorem 10.4] and [Pan89, §10]). Notice that we do not need to consider the transpose for the other matrices since we have $\mathbb{L}(\mathsf{u})^T = \mathbb{U}(\mathsf{u})$, $\mathbb{T}(\mathsf{u}, m, n)^T = \mathbb{T}(\mathbb{J}_{m+n-1}\mathsf{u}, n, m)$, $\mathbb{H}(\mathsf{u}, m, n)^T = \mathbb{H}(\mathsf{u}, n, m)$, and $\mathbb{C}(\mathsf{x}, \mathsf{y})^T = -\mathbb{C}(\mathsf{y}, \mathsf{x})$.

Table 1.1: Costs of matrix-vector multiplication and linear system solving for several types of matrix.

A	Av	$\mathsf{A}^{-1}\mathsf{v}$
dense, unstructured	$O(n^2)$	$O(n^\omega)$
triangular Toeplitz	$O(\mathsf{M}(n))$	$O(\mathsf{M}(n))$
Toeplitz	$O(\mathsf{M}(n))$	$O(\mathsf{M}(n) \log n)$
Hankel	$O(\mathsf{M}(n))$	$O(\mathsf{M}(n) \log n)$
Vandermonde	$O(\mathsf{M}(n) \log n)$	$O(\mathsf{M}(n) \log n)$
Vandermonde transposed	$O(\mathsf{M}(n) \log n)$	$O(\mathsf{M}(n) \log n)$
Cauchy	$O(\mathsf{M}(n) \log n)$	$O(\mathsf{M}(n) \log n)$

Except for the dense matrix case, all the costs mentioned in Table 1.1 are quasi-linear in n. In the sequel, we will use the notation $O\tilde{}(n)$, which means $O(n \log^k n)$ for some constant $k \in \mathbb{N}$. In other words, we add a tilde in Landau's "Big-O" notation to indicate that logarithmic factors have been dropped. The conclusion is therefore that, for all the special matrices mentioned in this section, multiplication by a vector can be achieved in $O\tilde{}(n)$ instead of $O(n^2)$ and linear system solving can be achieved in $O\tilde{}(n)$ instead of $O(n^\omega)$.

1.3 Matrices with displacement structure

In the previous section, we have seen several types of matrices for which the costs of multiplication by a vector and linear system solving can be achieved in quasi-linear time with respect to the dimension n. All these matrices share the particularity that only $O(n)$ coefficients are needed to represent them. In this section, we present an approach that was introduced by [KKM79] in order to put these matrices together in the same framework.

1.3.1 Displacement operators and displacement rank

Toeplitz, Hankel, Vandermonde, and Cauchy matrices of size $n \times m$ can be represented by $O(n+m)$ elements in $\mathbb{K}$. Therefore, there exist strong relations between the mn coefficients in these matrices. Let A be one of such matrices. The idea is to exploit these relations in order to compress A. More precisely, by applying an appropriate linear operator $\mathcal{L} : \mathbb{K}^{m\times n} \to \mathbb{K}^{m\times n}$ called *displacement operator*, we aim at turning A into a matrix of low rank $\mathcal{L}(\mathsf{A})$ [KKM79]. Two families of displacement operators are commonly used:

Definition 1.5. *Let* $\mathsf{M} \in \mathbb{K}^{m\times m}$ *and* $\mathsf{N} \in \mathbb{K}^{n\times n}$*. The linear operator*

$$\begin{array}{rccl} \nabla[\mathsf{M},\mathsf{N}] : & \mathbb{K}^{m\times n} & \to & \mathbb{K}^{m\times n} \\ & \mathsf{A} & \mapsto & \mathsf{MA}-\mathsf{AN}, \end{array}$$

is called **Sylvester's displacement operator** *associated to* M *and* N*, and the linear operator*

$$\begin{array}{rccl} \Delta[\mathsf{M},\mathsf{N}] : & \mathbb{K}^{m\times n} & \to & \mathbb{K}^{m\times n} \\ & \mathsf{A} & \mapsto & \mathsf{A}-\mathsf{MAN}, \end{array}$$

is called **Stein's displacement operator** *associated to* M *and* N*.*

Let us introduce a few more definitions before giving examples:

Definition 1.6. *Let* $\mathcal{L} : \mathbb{K}^{m\times n} \to \mathbb{K}^{m\times n}$ *and* $\mathsf{A} \in \mathbb{K}^{m\times n}$*. We will call hereafter* **displacement rank** *of* A *with respect to operator* $\mathcal{L}$ *the rank of the matrix* $\mathcal{L}(\mathsf{A})$*. If this rank is* α *then for any* $\beta \geq \alpha$ *there exist pairs of matrices* (G,H) *such that*

$$\mathsf{G} \in \mathbb{K}^{m\times\beta}, \qquad \mathsf{H} \in \mathbb{K}^{n\times\beta}, \qquad \mathcal{L}(\mathsf{A}) = \mathsf{GH}^T.$$

Any such pair will be called a **generator of length** β *for* A *with respect to operator* $\mathcal{L}$*.*

A given matrix A is considered to be structured when its displacement rank α with respect to some displacement operator $\mathcal{L}$ is "small" (in a sense that depends on the context) compared to m and n. Indeed, (G,H) contains only $\alpha(m+n)$ elements instead of mn for A. So, when α is small, (G,H) can thus be viewed as a compressed form the structured matrix A.

1.3.2 Main examples

Toeplitz and Hankel structures

The structure in Toeplitz and Hankel matrices comes from the repetition of coefficients along the diagonals. The idea to obtain a low displacement rank is to subtract the given matrix by a shifted version of it in order to produce lots of zeros. For this purpose, we introduce the family of so-called unit φ-circulant matrices $\mathbb{Z}_{n,\varphi} \in \mathbb{K}^{n\times n}$, where $n \in \mathbb{N}_{>0}$, $\varphi \in \mathbb{K}$ and the (i,j) entry of $\mathbb{Z}_{n,\varphi}$ is φ in position $(1,n)$, ones in positions $(i+1,i)$, and zeros everywhere else:

$$\mathbb{Z}_{n,\varphi} = \begin{bmatrix} & & \varphi \\ 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}.$$

When $\varphi = 0$ the matrix $\mathbb{Z}_{n,0}$ is called a shift matrix.

One can easily check that, for instance:

$$\Delta[\mathbb{Z}_{m,0}, \mathbb{Z}_{n,0}^T]\,(\mathbb{T}(\mathsf{v}, m, n)) = \begin{bmatrix} v_0 & v_{-1} \dots v_{1-n} \\ v_1 & \\ \vdots & \\ v_{m-1} & \end{bmatrix} = \begin{bmatrix} v_0 & 1 \\ v_1 & \\ \vdots & \\ v_{m-1} & \end{bmatrix} \cdot \begin{bmatrix} 1 & \\ & v_{-1} \\ & \vdots \\ & v_{1-n} \end{bmatrix}^T,$$

so that a Toeplitz matrix has a displacement rank at most equal to 2 for operator $\Delta[\mathbb{Z}_{m,0}, \mathbb{Z}_{n,0}^T]$. The same result holds for operators $\Delta[\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}^T]$, $\Delta[\mathbb{Z}_{m,\varphi}^T, \mathbb{Z}_{n,\psi}]$, $\nabla[\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}]$ and $\nabla[\mathbb{Z}_{m,\varphi}^T, \mathbb{Z}_{n,\psi}^T]$, where φ and ψ are any elements of $\mathbb{K}$.

Similarly, a Hankel matrix has a displacement rank at most equal to 2 for operators $\Delta[\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}]$, $\Delta[\mathbb{Z}_{m,\varphi}^T, \mathbb{Z}_{n,\psi}^T]$, $\nabla[\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}^T]$ and $\nabla[\mathbb{Z}_{m,\varphi}^T, \mathbb{Z}_{n,\psi}]$, where φ and ψ are any elements of $\mathbb{K}$.

Vandermonde structure

In a Vandermonde matrix, a given column is equal to the previous column pre-multiplied by the diagonal matrix $\mathbb{D}(\mathsf{x})$ defined by

$$\mathbb{D}(x) = \begin{bmatrix} x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_m \end{bmatrix}.$$

We have for instance

$$\nabla[\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,0}] \left(\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix} \right) = \begin{bmatrix} & & & x_1^n \\ & & & x_2^n \\ & & & \vdots \\ & & & x_m^n \end{bmatrix} = \begin{bmatrix} x_1^n \\ x_2^n \\ \vdots \\ x_m^n \end{bmatrix} \cdot \begin{bmatrix} \\ \\ \\ 1 \end{bmatrix}^T,$$

so that $\mathbb{V}(\mathsf{x}, n)$ has a displacement rank of 1 (assuming $\mathsf{x} \neq 0$) for the displacement operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,0}]$.

Cauchy structure

For Cauchy matrices, we will use two diagonal matrices for the displacement operator. Let $\mathsf{x} \in \mathbb{K}^m$ and $\mathsf{y} \in \mathbb{K}^n$. We have

$$\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]\,(\mathbb{C}(\mathsf{x}, \mathsf{y})) = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{bmatrix} = \mathsf{e}_m \cdot \mathsf{e}_n^T,$$

so that $\mathbb{C}(\mathsf{x}, \mathsf{y})$ has a displacement rank of 1 for the displacement operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$.

Generalization

Here and hereafter, we will restrict ourselves to displacement operators $\nabla[\mathsf{M}, \mathsf{N}]$ and $\Delta[\mathsf{M}, \mathsf{N}]$ with *displacement matrices* M and N such that

$$\begin{aligned} &\mathsf{M} \in \{\mathbb{D}(\mathsf{x}),\, \mathbb{Z}_{m,\varphi},\, \mathbb{Z}_{m,\varphi}^T\}, \quad \mathsf{x} \in \mathbb{K}^m, \quad \varphi \in \mathbb{K}; \\ &\mathsf{N} \in \{\mathbb{D}(\mathsf{y}),\, \mathbb{Z}_{n,\psi},\, \mathbb{Z}_{n,\psi}^T\}, \quad \mathsf{y} \in \mathbb{K}^n, \quad \psi \in \mathbb{K}. \end{aligned} \tag{1.5}$$

As we have seen above, this covers all the special matrices mentioned in Section 1.2.

In addition, each time we encounter a matrix A whose displacement rank α according to $\nabla[\mathsf{M},\mathsf{N}]$ or $\Delta[\mathsf{M},\mathsf{N}]$ is small, we will say that this matrix A is "like" the special matrix that also has a small displacement rank for this operator. This generalization is important because of observations like the following one: the product of two Toeplitz matrices is not a Toeplitz matrix in general, but it is always a Toeplitz-like matrix. Tables 1.2 and 1.3 sum up all the structures that we shall consider in the following of this part.

Table 1.2: Structure for $\nabla[\mathsf{M},\mathsf{N}]$ when M,N are diagonal or unit circulant matrices.

N M	$\mathbb{D}(\mathsf{y})$	$\mathbb{Z}_{n,\psi}$	$\mathbb{Z}_{n,\psi}^T$
$\mathbb{D}(\mathsf{x})$	Cauchy-like	Vandermonde-like	Vandermonde-like
$\mathbb{Z}_{m,\varphi}$	Vandermonde-transposed-like	Toeplitz-like	Hankel-like
$\mathbb{Z}_{m,\varphi}^T$	Vandermonde-transposed-like	Hankel-like	Toeplitz-like

Table 1.3: Structure for $\Delta[\mathsf{M},\mathsf{N}]$ when M,N are diagonal or unit circulant matrices.

N M	$\mathbb{D}(\mathsf{y})$	$\mathbb{Z}_{n,\psi}$	$\mathbb{Z}_{n,\psi}^T$
$\mathbb{D}(\mathsf{x})$	Cauchy-like	Vandermonde-like	Vandermonde-like
$\mathbb{Z}_{m,\varphi}$	Vandermonde-transposed-like	Hankel-like	Toeplitz-like
$\mathbb{Z}_{m,\varphi}^T$	Vandermonde-transposed-like	Toeplitz-like	Hankel-like

1.4 Basic properties of structured matrices

1.4.1 Recovering a structured matrix from its generators

One may wonder whether it is possible to explicitly build a structured matrix A given some generator (G,H) for it. To answer this question, a study of the invertibility of the operators $\nabla[\mathsf{M},\mathsf{N}]$ and $\Delta[\mathsf{M},\mathsf{N}]$ has been carried out in [Pan01, Theorem 4.3.2].

We present here several cases where the displacement operator is indeed invertible, and where we have formulas to achieve the recovery of A from (G,H). First, we recall two well-known formulas for the Cauchy-like structure. Then, we present two general reconstruction formulas that we illustrate on two examples that we will reuse in Section 2.3: Vandermonde-like and Hankel-like structures.

Reconstruction for Cauchy-like structure

For $\mathsf{x} \in \mathbb{K}^m$ and $\mathsf{y} \in \mathbb{K}^n$, assume

$$\mathsf{M} = \mathbb{D}(\mathsf{x}), \quad \mathsf{N} = \mathbb{D}(\mathsf{y}), \quad x_i \neq y_j \text{ for all } (i,j). \tag{1.6a}$$

Then $\nabla[\mathsf{M},\mathsf{N}]$ is invertible and it is known [GO94a] (see also [Pan01, page 8]) that $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ is equivalent to

$$\mathsf{A} = \textstyle\sum_{j=1}^{\alpha} \mathbb{D}(\mathsf{g}_j)\, \mathbb{C}(\mathsf{x},\mathsf{y})\, \mathbb{D}(\mathsf{h}_j). \tag{1.6b}$$

Similarly, assuming now that $x_i y_j \neq 1$ for all (i,j), $\Delta[\mathsf{M},\mathsf{N}]$ is invertible and $\Delta[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ is equivalent to

$$\mathsf{A} = \textstyle\sum_{j=1}^{\alpha} \mathbb{D}(\mathsf{g}_j)\, \widetilde{\mathbb{C}}(\mathsf{x},\mathsf{y})\, \mathbb{D}(\mathsf{h}_j),$$

with $\widetilde{\mathbb{C}}(\mathsf{x},\mathsf{y})$ the m by n matrix $[1/(1-x_i \cdot y_j)]_{i,j}$ (see [Pan01, Example 4.4.7]).

General reconstruction formulas

For the other structures mentioned in Section 1.3.2, we will rely on two general reconstruction formulas (one for Sylvester's displacement operator, and one for Stein's displacement operator). These formulas involve Krylov matrices:

Definition 1.7. *For* $\mathsf{M} \in \mathbb{K}^{m\times m}$, $\mathsf{u} \in \mathbb{K}^m$, *and* $\ell \in \mathbb{N}_{>0}$, *the Krylov matrix* $\mathcal{K}_\ell(\mathsf{M},\mathsf{u})$ *is the* $m \times \ell$ *matrix whose* j*th column equals* $\mathsf{M}^{j-1}\mathsf{u}$.

Lemma 1.1 (see [PW02])**.** *If* $\Delta[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ *then for any positive integer* ℓ, *we have*

$$\mathsf{A} - \mathsf{M}^\ell \mathsf{A} \mathsf{N}^\ell = \sum_{j\le\alpha} \mathcal{K}_\ell(\mathsf{M},\mathsf{g}_j)\mathcal{K}_\ell(\mathsf{N}^T,\mathsf{h}_j)^T.$$

The special shape of Krylov matrices implies, when M is invertible, that $\mathcal{K}_\ell(\mathsf{M}^{-1},\mathsf{M}^{-1}\mathsf{u}) = \mathsf{M}^{-\ell}\mathcal{K}_\ell(\mathsf{M},\mathsf{u})\mathbb{J}_\ell$. Using this identity, an analog for Sylvester's displacement of the previous lemma follows easily:

Lemma 1.2. *If* $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A})$ *and if* $(\det \mathsf{M}, \det \mathsf{N}) \neq (0,0)$ *then*

$$\mathsf{M}^\ell \mathsf{A} - \mathsf{A}\mathsf{N}^\ell = \sum_{j\le\alpha} \mathcal{K}_\ell(\mathsf{M},\mathsf{g}_j)\mathbb{J}_\ell\mathcal{K}_\ell(\mathsf{N}^T,\mathsf{h}_j)^T$$

for any positive integer ℓ.

Since we have already dealt with the Cauchy-like structure, we have that either M or N is a shift matrix $\mathbb{Z}_{*,*}$ or $\mathbb{Z}_{*,*}^T$. Using an appropriate value of ℓ, we thus obtain either $\mathsf{M}^\ell = \lambda\mathbb{I}_m$ or $\mathsf{N}^\ell = \lambda\mathbb{I}_n$, so that we can deduce from the above lemmas a formula for A.

Example of the Vandermonde-like structure (operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,0}^T]$)

For $\mathsf{x} \in \mathbb{K}^m$, assume now

$$\mathsf{M} = \mathbb{D}(\mathsf{x}), \quad \mathsf{N} = \mathbb{Z}_{n,0}^T, \quad x_i \neq 0 \text{ for all } i. \tag{1.7a}$$

Then $\det \mathsf{M} \neq 0$ and Lemma 1.2 with $\ell = n$ tells us that A can be recovered as follows:

$$\mathsf{A} = \mathbb{D}(\mathsf{x})^{-n} \sum_{j=1}^{\alpha} \mathcal{K}_n(\mathbb{D}(\mathsf{x}),\mathsf{g}_j)\mathbb{J}_n\mathcal{K}_n(\mathbb{Z}_{n,0},\mathsf{h}_j)^T.$$

Now, $\mathcal{K}_n(\mathbb{D}(\mathsf{x}), \mathsf{u}) = \mathbb{D}(\mathsf{u})\mathbb{V}(\mathsf{x}, n)$ and $\mathcal{K}_n(\mathbb{Z}_{n,0}, \mathsf{u}) = \mathbb{L}(u)$ for all $\mathsf{x} \in \mathbb{K}^m$ and $\mathsf{u} \in \mathbb{K}^n$. Hence, noticing that $\mathbb{D}(\mathsf{x})^{-n}\mathbb{V}(\mathsf{x}, n)\mathbb{J}_n = \mathbb{D}(\mathsf{x})^{-1}\mathbb{V}(\mathsf{x}^{-1}, n)$, we get

$$\mathsf{A} = \mathbb{D}(\mathsf{x})^{-1} \sum_{j=1}^{\alpha} \mathbb{D}(\mathsf{g}_j)\, \mathbb{V}(\mathsf{x}^{-1}, n)\, \mathbb{U}(\mathsf{h}_j). \tag{1.7b}$$

Example of the Hankel-like structure (operator $\nabla[\mathbb{Z}_{m,1}, \mathbb{Z}_{n,0}^T]$)

Assume that

$$\mathsf{M} = \mathbb{Z}_{m,1}, \quad \mathsf{N} = \mathbb{Z}_{n,0}^T. \tag{1.8a}$$

Since $\det \mathsf{M} \neq 0 = \det \mathsf{N}$, Lemma 1.2 with $\ell = n$ tells us that $\nabla[\mathsf{M}, \mathsf{N}]$ is invertible and that we can recover A as follows:

$$\begin{aligned}\mathsf{A} &= \mathbb{Z}_{m,1}^{-n} \sum_{j=1}^{\alpha} \mathcal{K}_n(\mathbb{Z}_{m,1}, \mathsf{g}_j)\mathbb{J}_n\mathcal{K}_n(\mathbb{Z}_{n,0}, \mathsf{h}_j)^T \\ &= \sum_{j=1}^{\alpha} \mathcal{K}_n(\mathbb{Z}_{m,1}^{-1}, \mathbb{Z}_{m,1}^{-1}\mathsf{g}_j)\mathcal{K}_n(\mathbb{Z}_{n,0}, \mathsf{h}_j)^T.\end{aligned}$$

Now, $\mathcal{K}_n(\mathbb{Z}_{n,0}, \mathsf{u}) = \mathbb{L}(u)$ and $\mathbb{L}(u)^T = \mathbb{J}_n\mathbb{L}(u)\mathbb{J}_n$ for all $\mathsf{u} \in \mathbb{K}^n$. Moreover, if $\widetilde{\mathsf{g}}_j \in \mathbb{K}^{m+n-1}$ denotes the vector in $\mathbb{K}^{m+n-1}$ whose ℓth entry is equal to $g_{j,1+(\ell \bmod m)}$, we have [JM10a, Appendix A] that $\mathcal{K}_n(\mathbb{Z}_{m,1}^{-1}, \mathbb{Z}_{m,1}^{-1}\mathsf{g}_j) = \mathbb{T}(\widetilde{\mathsf{g}}_j, m, n)\mathbb{J}_n$. Therefore, we obtain

$$\mathsf{A} = \sum_{j=1}^{\alpha} \mathbb{T}(\widetilde{\mathsf{g}}_j, m, n)\, \mathbb{L}(\mathsf{h}_j)\, \mathbb{J}_n. \tag{1.8b}$$

Remarks

For the other structures mentioned in Section 1.3.2, we refer to [Pan01, §4.4] and [PW02], where several reconstruction formulas (and their hypotheses) resulting from the two aforementioned lemmas are given. Now, let $\mathsf{A} \in \mathbb{K}^{m\times n}$ be a structured matrix according to $\mathcal{L} = \Delta[\mathsf{M}, \mathsf{N}]$ or $\nabla[\mathsf{M}, \mathsf{N}]$ with M and N as in (1.5). Given a length-α generator (G, H) for A, and assuming that the hypotheses for the reconstruction formula associated to $\mathcal{L}$ holds, we have that A is equal to a sum of α products involving a constant number of special matrices from Section 1.2. Therefore, given a vector $\mathsf{b} \in \mathbb{K}^n$, we can compute Ab from G, H, and b in $O\tilde{~}(\alpha(m+n))$: First, we distribute b in the sum; Then we compute each of the α terms within this sum in $O\tilde{~}(n)$ as seen in Table 1.1; And finally we perform the sum in $O(\alpha m)$.

1.4.2 Basic computations with structured matrices

Let us now review a few rules that allow one to carry out basic operations involving structured matrices. The main point here is to show that the concept of structured matrix is preserved in some way when adding, transposing, multiplying, and extracting structured matrices. Therefore, these operations can be performed by computations on generators, so that the structured matrices are never built explicitly. This offers the advantage that quasi-linear algorithms with respect to the dimensions of the involved matrices can be derived for all these operations.

The following properties are extracted from [Pan01, §1.5]. One can find the corresponding proofs therein.

Addition of two structured matrices

If two matrices share the same structure, we can add them and obtain another matrix, which also has this structure. This actually comes from the choice of a linear operator as the displacement operator $\mathcal{L}$:

Property 1.4. *Let* $\mathsf{A}, \mathsf{B} \in \mathbb{K}^{m\times n}$, $\mathsf{G}_\mathsf{A} \in \mathbb{K}^{m\times\alpha}$, $\mathsf{H}_\mathsf{A} \in \mathbb{K}^{n\times\alpha}$, $\mathsf{G}_\mathsf{B} \in \mathbb{K}^{m\times\beta}$, *and* $\mathsf{H}_\mathsf{B} \in \mathbb{K}^{n\times\beta}$. *If* $\mathcal{L}(\mathsf{A}) = \mathsf{G}_\mathsf{A}\mathsf{H}_\mathsf{A}^T$ *and* $\mathcal{L}(\mathsf{B}) = \mathsf{G}_\mathsf{B}\mathsf{H}_\mathsf{B}^T$ *then*

$$\mathcal{L}(\mathsf{A}+\mathsf{B}) = \mathsf{G}\mathsf{H}^T,$$

where $\mathsf{G} = [\mathsf{G}_\mathsf{A}|\mathsf{G}_\mathsf{B}]$ *and* $\mathsf{H} = [\mathsf{H}_\mathsf{A}|\mathsf{H}_\mathsf{B}]$.

First, note that computing the generator (G, H) for $\mathsf{A}+\mathsf{B}$ described above requires no arithmetic operations in $\mathbb{K}$. Next, notice that, if A and B have respectively a displacement rank of α and β, we deduce that the displacement rank of $\mathsf{A}+\mathsf{B}$ is at most $\alpha+\beta$. While it is easy to find cases where $\mathsf{A}+\mathsf{B}$ has actually a displacement rank equal to $\alpha+\beta$, it may happen that this displacement rank is smaller. For instance, think of the case where $\mathsf{B} = -\mathsf{A}$, where the displacement rank will drop to zero.

Transposition of a structured matrix

Property 1.5. *Let* $\mathsf{A}, \mathsf{G}, \mathsf{H}$ *be such that* $\nabla[\mathsf{M}, \mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$. *By transposing the identity* $\mathsf{MA} - \mathsf{AN} = \mathsf{G}\mathsf{H}^T$, *we obtain*

$$\nabla[\mathsf{N}^T, \mathsf{M}^T](\mathsf{A}^T) = -\mathsf{H}\mathsf{G}^T,$$

so that the pair $(-\mathsf{H}, \mathsf{G})$ *is a* $\nabla[\mathsf{N}^T, \mathsf{M}^T]$*-generator for* A^T.

Similarly, let $\mathsf{A}, \mathsf{G}, \mathsf{H}$ *be such that* $\Delta[\mathsf{M}, \mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$. *By transposing the identity* $\mathsf{A} - \mathsf{MAN} = \mathsf{G}\mathsf{H}^T$, *we obtain*

$$\Delta[\mathsf{N}^T, \mathsf{M}^T](\mathsf{A}^T) = \mathsf{H}\mathsf{G}^T,$$

so that the pair (H, G) *is a* $\Delta[\mathsf{N}^T, \mathsf{M}^T]$*-generator for* A^T.

In the second case, computing the proposed generator for A^T requires no operations in $\mathbb{K}$. For the first case, we need to compute the opposite of all the elements in H, so $\alpha n \in O(\alpha(m+n))$ operations in $\mathbb{K}$ are needed.

Multiplication of two structured matrices

Property 1.6. *Let* $\mathsf{A} \in \mathbb{K}^{m \times p}$ *and* $\mathsf{B} \in \mathbb{K}^{p \times n}$. *One has the following equality:*

$$\nabla[\mathsf{M}, \mathsf{N}](\mathsf{AB}) = \nabla[\mathsf{M}, \mathsf{P}](\mathsf{A})\,\mathsf{B} + \mathsf{A}\,\nabla[\mathsf{P}, \mathsf{N}](\mathsf{B}).$$

Moreover, if $(\mathsf{G_A}, \mathsf{H_A})$ *is a length-*α *generator of* A *for operator* $\nabla[\mathsf{M}, \mathsf{P}]$ *and if* $(\mathsf{G_B}, \mathsf{H_B})$ *is a length-*β *generator of* B *for operator* $\nabla[\mathsf{P}, \mathsf{N}]$, *we get*

$$\nabla[\mathsf{M}, \mathsf{N}](\mathsf{AB}) = \mathsf{GH}^T,$$

where $\mathsf{G} = [\mathsf{G_A} | \mathsf{AG_B}] \in \mathbb{K}^{m \times (\alpha+\beta)}$ *and* $\mathsf{H} = [\mathsf{B}^T\mathsf{H_A} | \mathsf{H_B}] \in \mathbb{K}^{n \times (\alpha+\beta)}$. *Therefore, the displacement rank of* AB *is at most* $\alpha + \beta$.

The multiplication rule for Stein-type displacement operators is a bit more complicated. The equivalent of the first formula in Property 1.6 is

$$\Delta[\mathsf{M}, \mathsf{N}](\mathsf{AB}) = \Delta[\mathsf{M}, \mathsf{P}](\mathsf{A})\,\mathsf{B} + \mathsf{MA}\,\nabla[\mathsf{P}, \mathsf{N}](\mathsf{B}).$$

In order to obtain a formula with Stein-type displacement operators only, we need to make some extra assumption. For instance, if P is invertible, we deduce that

$$\Delta[\mathsf{M}, \mathsf{N}](\mathsf{AB}) = \Delta[\mathsf{M}, \mathsf{P}](\mathsf{A})\,\mathsf{B} + \mathsf{MAP}\,\Delta[\mathsf{P}^{-1}, \mathsf{N}](\mathsf{B}).$$

In this case, given a $\Delta[\mathsf{M}, \mathsf{P}]$-generator $(\mathsf{G_A}, \mathsf{H_A})$ of length α for A and a $\Delta[\mathsf{P}^{-1}, \mathsf{N}]$-generator $(\mathsf{G_B}, \mathsf{H_B})$ of length β for B, the matrices $\mathsf{G} = [\mathsf{G_A} | \mathsf{MAP}\,\mathsf{G_B}]$ and $\mathsf{H} = [\mathsf{B}^T\mathsf{H_A} | \mathsf{H_B}]$ form a $\Delta[\mathsf{M}, \mathsf{N}]$-generator of length $\alpha + \beta$ for AB. We refer to [Pan01, Theorem 1.5.4] for a more complete result, and to [Kal94, Proposition 2] for the product of two Toeplitz-like matrices (operator $\Delta[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T]$), where the length achieved is[2] $\alpha + \beta + 1$.

Computing the proposed generator for AB requires that we compute the products like $\mathsf{AG_B}$, that is, products of the form "structured matrix × vectors". A naive way to perform these products is to successively multiply the structured matrix by each vector like in Section 1.4.1. This yields a cost of $O\tilde{~}(\alpha\beta(m + p + n))$ operations in $\mathbb{K}$. When $\beta \in O(\alpha)$ and $m, p \in O(n)$, this amounts to $O\tilde{~}(\alpha^2 n)$.

Extraction of a submatrix

Given positive integers n_1 and n_2 such that $n_1 + n_2 = n$, we partition A, G, H, M, N into $n_i \times n_j$ or $n_i \times \alpha$ blocks as

$$\begin{gathered}\mathsf{A} = \begin{bmatrix} \mathsf{A}_{11} & \mathsf{A}_{12} \\ \mathsf{A}_{21} & \mathsf{A}_{22} \end{bmatrix}, \quad \mathsf{G} = \begin{bmatrix} \mathsf{G}_1 \\ \mathsf{G}_2 \end{bmatrix}, \quad \mathsf{H} = \begin{bmatrix} \mathsf{H}_1 \\ \mathsf{H}_2 \end{bmatrix}, \\ \mathsf{M} = \begin{bmatrix} \mathsf{M}_{11} & \mathsf{M}_{12} \\ \mathsf{M}_{21} & \mathsf{M}_{22} \end{bmatrix}, \quad \mathsf{N} = \begin{bmatrix} \mathsf{N}_{11} & \mathsf{N}_{12} \\ \mathsf{N}_{21} & \mathsf{N}_{22} \end{bmatrix}.\end{gathered} \tag{1.9}$$

[2] In this case, $\mathsf{P} = \mathbb{Z}_{n,0}^T$ is not invertible, so the previous reasoning leading to a generator of length $\alpha + \beta$ does not hold.

We shall write μ and ν for the rank of, respectively, M_{12} and N_{21}. Consequently, those two matrices can be written

$$\mathsf{M}_{12} = \mathsf{U}_1\mathsf{V}_2^T, \qquad \mathsf{N}_{21} = \mathsf{U}_2\mathsf{V}_1^T \tag{1.10}$$

for some full column rank matrices $\mathsf{U}_1 \in \mathbb{K}^{n_1\times\mu}$, $\mathsf{V}_2 \in \mathbb{K}^{n_2\times\mu}$, $\mathsf{U}_2 \in \mathbb{K}^{n_2\times\nu}$, and $\mathsf{V}_1 \in \mathbb{K}^{n_1\times\nu}$. Similarly, let us write $\widetilde{\mu}$ and $\widetilde{\nu}$ for the rank of, respectively, M_{21} and N_{12} so that

$$\mathsf{M}_{21} = \widetilde{\mathsf{U}}_2\widetilde{\mathsf{V}}_1^T, \qquad \mathsf{N}_{12} = \widetilde{\mathsf{U}}_1\widetilde{\mathsf{V}}_2^T$$

for some full column rank matrices $\mathsf{U}_2 \in \mathbb{K}^{n_2\times\widetilde{\mu}}$, $\mathsf{V}_1 \in \mathbb{K}^{n_1\times\widetilde{\mu}}$, $\mathsf{U}_1 \in \mathbb{K}^{n_1\times\widetilde{\nu}}$, and $\mathsf{V}_2 \in \mathbb{K}^{n_2\times\widetilde{\nu}}$.

Property 1.7. *From $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ and the partitioning into blocks we deduce that, for $i, j \in \{1,2\}$, submatrix A_{ij} satisfies the following matrix equation*

$$\nabla[\mathsf{M}_{ij},\mathsf{N}_{ij}](\mathsf{A}_{ij}) = \mathsf{G}_{ij}\mathsf{H}_{ij}^T,$$

where (see for example [Pan00, Proposition 4.4]):

$$\mathsf{G}_{11} = \left[\ \mathsf{G}_1 \;\middle|\; -\mathsf{U}_1 \;\middle|\; \mathsf{A}_{12}\mathsf{U}_2\ \right] \in \mathbb{K}^{n_1\times(\alpha+\mu+\nu)}, \tag{1.11a}$$

$$\mathsf{H}_{11} = \left[\ \mathsf{H}_1 \;\middle|\; \mathsf{A}_{21}^T\mathsf{V}_2 \;\middle|\; \mathsf{V}_1\ \right] \in \mathbb{K}^{n_1\times(\alpha+\mu+\nu)}, \tag{1.11b}$$

$$\mathsf{G}_{12} = \left[\ \mathsf{G}_1 \;\middle|\; -\mathsf{U}_1 \;\middle|\; \mathsf{A}_{11}\widetilde{\mathsf{U}}_1\ \right] \in \mathbb{K}^{n_1\times(\alpha+\mu+\widetilde{\nu})},$$

$$\mathsf{H}_{12} = \left[\ \mathsf{H}_2 \;\middle|\; \mathsf{A}_{22}^T\mathsf{V}_2 \;\middle|\; \widetilde{\mathsf{V}}_2\ \right] \in \mathbb{K}^{n_2\times(\alpha+\mu+\widetilde{\nu})},$$

$$\mathsf{G}_{21} = \left[\ \mathsf{G}_2 \;\middle|\; -\widetilde{\mathsf{U}}_2 \;\middle|\; \mathsf{A}_{22}\mathsf{U}_2\ \right] \in \mathbb{K}^{n_2\times(\alpha+\widetilde{\mu}+\nu)},$$

$$\mathsf{H}_{21} = \left[\ \mathsf{H}_1 \;\middle|\; \mathsf{A}_{11}^T\widetilde{\mathsf{V}}_1 \;\middle|\; \mathsf{V}_1\ \right] \in \mathbb{K}^{n_1\times(\alpha+\widetilde{\mu}+\nu)},$$

$$\mathsf{G}_{22} = \left[\ \mathsf{G}_2 \;\middle|\; -\widetilde{\mathsf{U}}_2 \;\middle|\; \mathsf{A}_{21}\widetilde{\mathsf{U}}_1\ \right] \in \mathbb{K}^{n_2\times(\alpha+\widetilde{\mu}+\widetilde{\nu})},$$

$$\mathsf{H}_{22} = \left[\ \mathsf{H}_2 \;\middle|\; \mathsf{A}_{12}^T\widetilde{\mathsf{V}}_1 \;\middle|\; \widetilde{\mathsf{V}}_2\ \right] \in \mathbb{K}^{n_2\times(\alpha+\widetilde{\mu}+\widetilde{\nu})}.$$

It should be noted that, for all the structures mentioned in Section 1.3.2, the ranks μ, ν, $\widetilde{\mu}$ and $\widetilde{\nu}$ are always at most 1. Consequently, the property above tells us that a submatrix of a structured matrix is almost as structured as the initial matrix, for any structure of interest here.

In order to effectively compute G_{ij} and H_{ij} as above, we need to perform products like $\mathsf{A}_{12}\mathsf{U}_2$, where the matrix on the left is a submatrix of A. We do not know yet a generator for A_{12}, but we can use the ones of A instead and compute $\mathsf{A}\cdot\left[\begin{smallmatrix}0\\ \mathsf{U}_2\end{smallmatrix}\right]$, from which we extract the first n_1 rows. Using successive "structured matrix × vector" products, the computation of $\mathsf{A}_{12}\mathsf{U}_2$ can then be achieved using $\tilde{O}(\alpha(m+n)\nu)$ operations in $\mathbb{K}$. Assuming that μ, ν, $\widetilde{\mu}$ and $\widetilde{\nu}$ are small constants, and using the same method for the other products appearing in the above definitions of G_{ij} and H_{ij}, we conclude that generators for A_{ij} can be deduced from the length-α generator (G,H) of A using $\tilde{O}(\alpha(m+n))$ operations in $\mathbb{K}$.

1.4.3 Inversion of a structured matrix

One of the most important properties of structured matrices is that the inverse of a structured matrix is also structured. This is summarized by the following theorem [Pan01, Theorem 1.5.3]:

Property 1.8. *If* $\mathsf{A} \in \mathbb{K}^{n\times n}$ *is nonsingular and* $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ *with* $\mathsf{G},\mathsf{H} \in \mathbb{K}^{n\times\alpha}$ *then*

$$\nabla[\mathsf{N},\mathsf{M}](\mathsf{A}^{-1}) = -\mathsf{A}^{-1}\ \nabla[\mathsf{M},\mathsf{N}](\mathsf{A})\ \mathsf{A}^{-1} = \mathsf{Y}\mathsf{Z}^T,$$

where $\mathsf{Y} := -\mathsf{A}^{-1}\mathsf{G}$ *and* $\mathsf{Z} := \mathsf{A}^{-T}\mathsf{H}$.

Moreover, let $\mathsf{A} \in \mathbb{K}^{n\times n}$ *be nonsingular and* $\Delta[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ *with* $\mathsf{G},\mathsf{H} \in \mathbb{K}^{n\times\alpha}$*:*

- *if* M *is invertible then*

  $$\Delta[\mathsf{N},\mathsf{M}](\mathsf{A}^{-1}) = \mathsf{A}^{-1}\mathsf{M}^{-1}\ \Delta[\mathsf{M},\mathsf{N}](\mathsf{A})\ \mathsf{A}^{-1}\mathsf{M} = \mathsf{Y}\mathsf{Z}^T,$$

  where $\mathsf{Y} := \mathsf{A}^{-1}\mathsf{M}^{-1}\mathsf{G}$ *and* $\mathsf{Z} := \mathsf{M}^T\mathsf{A}^{-T}\mathsf{H}$*;*

- *if* N *is invertible then*

  $$\Delta[\mathsf{N},\mathsf{M}](\mathsf{A}^{-1}) = \mathsf{N}\mathsf{A}^{-1}\ \Delta[\mathsf{M},\mathsf{N}](\mathsf{A})\ \mathsf{N}^{-1}\mathsf{A}^{-1} = \mathsf{Y}\mathsf{Z}^T,$$

  where $\mathsf{Y} := \mathsf{N}\mathsf{A}^{-1}\mathsf{G}$ *and* $\mathsf{Z} := \mathsf{A}^{-T}\mathsf{N}^{-T}\mathsf{H}$.

Like for the multiplication, the situation is simpler with a Sylverster-type displacement operator $\nabla[\mathsf{M},\mathsf{N}]$. In this case, we can conclude without any assumption on M or N that the $\nabla[\mathsf{N},\mathsf{M}]$-displacement rank of A^{-1} is equal to the $\nabla[\mathsf{M},\mathsf{N}]$-displacement rank of A. This is also true for Stein-type operators given the invertibility of M or N, or in some special cases like the Toeplitz-like structure (see [KKM79, Theorem 1]).

Therefore, each time we consider the inverse A^{-1} of a structured matrix A, we will assume that A^{-1} is actually represented by a generator (Y,Z). For instance, suppose we want to solve a structured linear system, that is, to compute $\mathsf{A}^{-1}\mathsf{b}$ for some structured matrix A and some input vector b. This computation will be carried out by first computing a generator for A^{-1} from the one of A, and then applying the reconstruction formula corresponding to the displacement operator associated to A^{-1} in order to compute the product $\mathsf{A}^{-1}\cdot\mathsf{b}$.

One classical algorithm to compute a generator for A^{-1}, given a generator (G,H) for the Toeplitz-like matrix A, was introduced independently by Morf [Mor80], and Bitmead and Anderson [BA80]. Their recursive approach, sketched in Algorithm 1.1, can be viewed as a structured version of the matrix inversion using block Gaussian elimination mentioned in Section 1.1.2. Indeed, instead of manipulating matrices themselves, `MBA` produces generators for the involved matrices using the rules on addition and multiplication of structured matrices mentioned in Section 1.4.2. Thus, a generator for the Schur complement S is computed at line 3, for instance.

One technical issue with this approach lies in the growth in length of the intermediate generators, resulting from the successive additions and multiplications of structured matrices. In order to achieve a cost of $O\tilde{~}(\alpha^2\,n)$ with this approach, where α and n are the dimensions of matrices at input, one must ensure that both the generator $(\mathsf{G}_\mathsf{S},\mathsf{H}_\mathsf{S})$ used

Algorithm 1.1: Sketch of the Morf/Bitmead-Anderson (`MBA`) algorithm for divide-and-conquer inversion of Toeplitz-like matrices [Mor80, BA80].

```
Input      : $\mathsf{G}, \mathsf{H} \in \mathbb{K}^{n\times\alpha}$ such that $\Delta[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$.
Assumption : $\mathsf{A}$ is strongly regular.
Output     : $\mathsf{Y}$ and $\mathsf{Z}$ such that $\Delta[\mathbb{Z}_{n,0}^T, \mathbb{Z}_{n,0}](\mathsf{A}^{-1}) = \mathsf{Y}\mathsf{Z}^T$.
1 Compute a length-$\alpha$ generator $(\mathsf{G}_{11}, \mathsf{H}_{11})$ for $A_{11}$
2 $(\mathsf{Y}_{11}, \mathsf{Z}_{11}) \leftarrow$ MBA$(\mathsf{G}_{11}, \mathsf{H}_{11})$
3 Compute a generator $(\widetilde{\mathsf{G}_\mathsf{S}}, \widetilde{\mathsf{H}_\mathsf{S}})$ for $\mathsf{S} = \mathsf{A}_{22} - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$
4 Deduce from $(\widetilde{\mathsf{G}_\mathsf{S}}, \widetilde{\mathsf{H}_\mathsf{S}})$ a length-$\alpha$ generator $(\mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S})$ for $\mathsf{S}$
5 $(\mathsf{Y}_\mathsf{S}, \mathsf{Z}_\mathsf{S}) \leftarrow$ MBA$(\mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S})$
6 Compute generators for $-\mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}$, $-\mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$ and $\mathsf{A}_{11}^{-1} + \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$
7 Deduce a generator $(\widetilde{\mathsf{Y}}, \widetilde{\mathsf{Z}})$ for $\mathsf{A}^{-1} = \begin{bmatrix} \mathsf{A}_{11}^{-1}+\mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1} \\ -\mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathsf{S}^{-1} \end{bmatrix}$
8 Deduce from $(\widetilde{\mathsf{Y}}, \widetilde{\mathsf{Z}})$ a length-$\alpha$ generator $(\mathsf{Y}, \mathsf{Z})$ for $\mathsf{A}^{-1}$
9 return $\mathsf{Y}$ and $\mathsf{Z}$
```

for the recursive call and the output generator (Y, Z) are of length α. We have already discussed the size of a generator for the inverse when commenting Property 1.8 above, and seen that size α is to be expected. In fact, the same conclusion holds for the Schur complement, which was one of the main contributions from [Mor80] and [BA80]. Yet, the arithmetic on structured matrices provides larger generator matrices. To cope with this length issue, one uses a generator compression stage[3] which, given matrices $\mathsf{G}, \mathsf{H} \in \mathbb{K}^{n\times\beta}$ such that $\mathsf{G}\mathsf{H}^T$ has rank $\alpha \leq \beta$, computes matrices $\mathsf{G}_c, \mathsf{H}_c$ that satisfy $\mathsf{G}_c\mathsf{H}_c^T = \mathsf{G}\mathsf{H}^T$ but now have exactly α columns. We refer to [Pan92b, Pan92a, Pan93] and [Pan01, §4.6] for more details about this compression stage, and to [Kal94] for a complete study of the length of intermediate generators in `MBA` and a proof that it can be implemented so as to achieve a total cost in $O\tilde{~}(\alpha^2\, n)$.

Since its first statement in 1980, `MBA` has been improved in several ways. Kaltofen [Kal94, Kal95] proposed to add a probabilistic preconditioning in order to get rid of the strong-regularity assumption on A. Pan and Zheng [PZ00] gave a version of `MBA` for Cauchy-like matrices (with Sylvester-type operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$). Moreover, using an explicit formula for a generator of the Schur complement pointed out in [GO94b, Theorem 2.3], they managed to avoid the compression stage (line 4 in Algorithm 1.1) preceding the second recursive call. Cardinal [Car99, Car00] presented an algorithm, again for the Cauchy-like structure, where both compression stages are avoided. One version of `MBA` for Vandermonde-like matrices can be found in [OS03]. Finally, a uniform approach covering all the structures mentioned in Section 1.3.2 is described in [Pan01, §5]. Furthermore, more recent works [BJS07, BJS08] show that it is possible to use polynomial matrix multiplication in order to speed-up the multiplications of type "Toeplitz-like matrix × matrix". Then, a cost of $O\tilde{~}(\alpha^{\omega-1}\, n)$ for inversion of Toeplitz-like matrices, as well as for the other structures via a reduction to the Toeplitz-like case.

To conclude, let us mention a few alternatives to `MBA` for the problem of structured

[3] This stage is explicitly written in Algorithm 1.1 (see lines 4 and 8).

matrix inversion. First, several iterative algorithms [GO94b, GKO95], [KS99, §1.10] that can be viewed as the structured version of Gaussian elimination or LU decomposition have been developed. These algorithms are usually called *fast algorithms* since they achieve a complexity in $O(\alpha\, n^2)$, which is asymptotically faster than $O(n^3)$ for naive dense matrix inversion, but for small α asymptotically slower than the complexity in $O\tilde{}(\alpha^2\, n)$ achieved by *superfast algorithms* like `MBA`. The main advantage of these fast algorithms lies in the usage of pivoting, so that the assumption on the strong regularity of the matrix in input is no more needed, and the numerical quality can be improved (see for instance [GKO95]). Second, more specific algorithms have been designed for particular structures. For instance, one can cite the works in [LCC90, LBC95] where the inversion of mosaic Hankel matrices[4] is achieved through matrix-Padé approximations. This gives a deterministic algorithm whose complexity for block Hankel matrices is in $O\tilde{}(\alpha^{\omega-1}\, n)$, using the complexity announced in [GJV03] for matrix-Padé approximation. Furthermore, it may happen that the generator of a structured matrix is also structured. This occurs for instance with a block Toeplitz matrix where the blocks themselves are Toeplitz matrices. In this special cases, algorithms that are faster than `MBA` can be designed [Kha08, §4 and 5].

While all these alternatives of `MBA` are interesting, we aim at asymptotically fast inversion for all the structures mentioned in Section 1.3.2. Therefore, we will focus on variations of `MBA` in the next chapters.

1.5 Contributions of this thesis

In this work, we will focus on structured matrices as in Table 1.2, thus considering only Sylvester's displacement operators. However, the results presented in Chapter 3 have been adapted to Stein's displacement operators [BJMS11].

The contributions to this first part of the document are;

- a general, compression-free approach for structured matrix inversion, extending the algorithm for Cauchy-like matrices in [Car99, Car00];
- a new approach for the fast computation of multiplications of the form "structured matrix × matrix", extending the approach for Toeplitz-like matrices (with Stein's displacement operator) presented in [Bos10, page 210];
- some software developments, and experimental results.

Let us now detail these three points.

1.5.1 Compression-free approach for structured matrix inversion

As mentioned in Section 1.4.3, a technical issue in Algorithm `MBA` and most of its variants lies in the control of the length of the intermediate generators, that is usually achieved through compression steps. In Chapter 2, we present a general, compression-free approach

[4] This is a variant of the block Hankel structure, where the blocks can have different sizes.

for structured matrix inversion. While suppressing the compression steps leaves the asymptotic cost of structured matrix inversion unchanged, it yields smaller dominant terms in the overall cost.

Reviewing Cardinal's algorithm

A compression-free algorithm for the Cauchy-like structure was introduced by Cardinal [Car99, Car00]. The key point in his algorithm is to compute recursively the specified generator for A^{-1}:

$$\mathsf{Y} = -\mathsf{A}^{-1}\mathsf{G}, \qquad \mathsf{Z} = \mathsf{A}^{-T}\mathsf{H}.$$

By analyzing Cardinal's algorithms, we deduce in Section 2.1.3 the following recursive formulas that express the specified generator for A^{-1} from the specified generators $(\mathsf{Y}_{11}, \mathsf{Z}_{11})$ and $(\mathsf{Y}_{\mathsf{S}}, \mathsf{Z}_{\mathsf{S}})$ of A_{11}^{-1} and S^{-1}, respectively:

$$\mathsf{Y} = \mathsf{F}\begin{bmatrix}\mathsf{Y}_{11}\\ \mathsf{Y}_{\mathsf{S}}\end{bmatrix} \qquad \text{and} \qquad \mathsf{Z} = \mathsf{E}^{T}\begin{bmatrix}\mathsf{Z}_{11}\\ \mathsf{Z}_{\mathsf{S}}\end{bmatrix},$$

where $\mathsf{E} = \left[\begin{smallmatrix}\mathbb{I}_{n_1} & \\ -\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathbb{I}_{n_2}\end{smallmatrix}\right]$ and $\mathsf{F} = \left[\begin{smallmatrix}\mathbb{I}_{n_1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12} \\ & \mathbb{I}_{n_2}\end{smallmatrix}\right]$.

In fact, this formula holds for all the structures defined by an operator $\nabla[\mathsf{M}, \mathsf{N}]$ with M and N^T lower triangular. This fact is a consequence of a general formula, holding for $\nabla[\mathsf{M}, \mathsf{N}]$ with arbitrary matrices M and N, that we shall prove in Section 2.2.1.

Algorithm `GenInvLT`

Combining the formula in [GO94b, Theorem 2.3] for a compression-free generation of the Schur complement S and the aforementioned recursive formula for a compression-free generation of A^{-1}, we obtain when M and N^T are lower triangular an inversion algorithm without compression, that we call `GenInvLT` (see Section 2.3.1).

Cost for the Cauchy-like structure. In Section 2.3.2, we prove that `GenInvLT` for the Cauchy-like structure as defined in (1.6a) requires at most

$$3\log(n)\,\mathsf{MM}_{\mathsf{C}}(\alpha, n) + O(\alpha\, n\log(n))$$

field operations, where $\mathsf{MM}_{\mathsf{C}}(\alpha, n)$ denotes the cost of a product "Cauchy-like matrix × vectors" where the displacement rank of the Cauchy-like matrix and the number of vectors are both equal to α. Moreover, assuming in addition that x and y are repetition free, we can use a simplification pointed out by Cardinal, leading to a cost of at most

$$2\log(n)\,\mathsf{MM}_{\mathsf{C}}(\alpha, n) + O(\alpha\, n\log(n))$$

field operations.

Cost for the Vandermonde-like structure. In Section 2.3.3, we prove that `GenInvLT` for the Vandermonde-like structure as defined in (1.7a) requires at most

$$3\log(n)\,\mathsf{MM}_{\mathsf{V}}(\alpha,n)+O(\alpha\,\mathsf{M}(n)\log^2(n))$$

field operations, where $\mathsf{MM}_{\mathsf{V}}(\alpha,n)$ denotes the cost of a product "Vandermonde-like matrix × vectors" where the displacement rank of the Vandermonde-like matrix and the number of vectors are both equal to α. Moreover, assuming in addition that x is repetition free, we can extend Cardinal's simplification for the Cauchy-like case, leading to a cost of at most

$$2\log(n)\,\mathsf{MM}_{\mathsf{V}}(\alpha,n)+O(\alpha\,\mathsf{M}(n)\log^2(n)).$$

field operations.

Algorithm `GenInvHL`

For the Hankel-like structure, we want to consider the displacement operator $\nabla[\mathbb{Z}_{n,0},\mathbb{Z}_{n,0}^T]$ in order to have $\mathsf{M}=\mathsf{N}=\mathbb{Z}_{n,0}$ lower triangular. However, the operator is not invertible, meaning that the matrix A cannot be recovered from its generator (G,H) only. To cope with this difficulty, we need to deal with additional data called the *irregularity set* [Pan01, p. 136]. In Section 2.3.4, we explain how to extend Algorithm `GenInvLT` in order to handle the Hankel-like structure with its irregularity set. This yields Algorithm `GenInvHL`. Moreover, we show that inverting a Hankel-like matrix using `GenInvHL` requires at most

$$2\log(n)\,\mathsf{MM}_{\mathsf{H}}(\alpha,n)+O(\alpha\,\mathsf{M}(n)\log(n)).$$

field operations, where $\mathsf{MM}_{\mathsf{H}}(\alpha,n)$ is the counterpart of $\mathsf{MM}_{\mathsf{C}}(\alpha,n)$ and $\mathsf{MM}_{\mathsf{V}}(\alpha,n)$ for Hankel-like matrices.

Comparison with the classical `MBA` algorithm

Finally, a study of the cost of the classical `MBA` approach yields a cost dominated by $14\log(n)\,\mathsf{MM}_*(\alpha,n)$ with $*\in\{\mathsf{C},\mathsf{V},\mathsf{H}\}$ depending on the structure. Therefore, we conclude that our approach leads to a theoretical speed-up of $14/3\approx 4.67$ in general, and of $14/2=7$ in some situations.

1.5.2 Fast multiplication of a structured matrix by a matrix

The other part of our work on structured matrices, which is presented in Chapter 3, deals with the multiplication of a structured matrix by a matrix. Given an $m\times n$ structured matrix A with a displacement rank α and an $n\times\beta$ matrix B, the problem is to compute efficiently the product AB.

Extension of the approach in [Bos10, page 210]

For $\alpha=\beta$ and for square Toeplitz-like matrices (with Stein's displacement operator), the approach in [Bos10, page 210] gives a multiplication by a matrix that can be achieved in $O\tilde{~}(\alpha^{\omega-1}\,n)$ field operations using fast multiplication of polynomial matrices. We extend it in this thesis to all the structures in Table 1.2, and to the rectangular case.

Polynomial expression of AB. In Section 3.2, we propose a polynomial expression for the matrix product AB, generalizing the one proposed in [Bos10, page 210] for the Toeplitz-like case. This points out the problem of computing the polynomial row vector $\mathsf{R} = \mathsf{U}^T(\mathsf{VW}^T \bmod P)$ given $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V} \in \mathbb{K}[x]_n^{\alpha\times 1}$, and $\mathsf{W} \in \mathbb{K}[x]_n^{\beta\times 1}$, and where P is either $x^n - \psi$ or $\prod_{i=1}^n (x - y_i)$.

Computation of $\mathsf{R} = \mathsf{U}^T(\mathsf{VW}^T \bmod P)$. In Section 3.3, we provide two algorithms for the computation of R, assuming that $\beta = \alpha$. First, we explain how one can use the material in [Bos10, page 210] in order to solve the case where $P = x^n - \psi$. This yields Algorithm `ComputeRx`, for which we prove that it requires at most $\tilde{O}(\alpha^{\omega-1}(n+m))$ field operations (see Theorem 3.2 for a more precise result). Then, we propose an extension of the material in [Bos10, page 210] so as to deal with the case where $P = \prod_{i=1}^n (x - x_i)$. This yields Algorithm `ComputeRy`, whose cost is also in $\tilde{O}(\alpha^{\omega-1}(n+m))$ (see Theorem 3.4 for a more precise result).

Resulting costs for the multiplication of a structured matrix by a matrix. Combining the two previous points, we obtain a general algorithm for the multiplication "structured matrix × vectors," in the case where $\alpha = \beta$. In Section 3.4.1, we obtain the following costs, depending on the structure and the field in use, and for $N = m + n$:

	arbitrary field	field of characteristic zero, or finite field of cardinality at least $2N$
Toeplitz-like or Hankel-like	$O(\alpha^{\omega-1}\,\mathsf{M}(N))$	$O(\alpha^{\omega-1}\,N + \alpha \log \alpha\, \mathsf{M}(N))$
Vandermonde-like or Cauchy-like	$O(\alpha^{\omega-1}\,\mathsf{M}(N) + \alpha\,\mathsf{M}(N) \log N)$	$O(\alpha^{\omega-1}\,N + \alpha\,\mathsf{M}(N) \log N)$

Application to structured matrix inversion

Using the aforementioned costs for the functions MM_* yields the following costs for our inversion algorithms introduced in Chapter 2 depending on the structure and the field in use:

	arbitrary field	field of characteristic zero, or finite field of cardinality at least $4n$
Hankel-like	$O(\alpha^{\omega-1}\,\mathsf{M}(n) \log n)$	$O(\alpha^{\omega-1}\,n \log n + \alpha \log \alpha\, \mathsf{M}(n) \log n)$
Vandermonde- or Cauchy-like	$O(\alpha^{\omega-1}\,\mathsf{M}(n) \log n + \alpha\,\mathsf{M}(n) \log^2 n)$	$O(\alpha^{\omega-1}\,n \log n + \alpha\,\mathsf{M}(n) \log^2 n)$

Combining [BJS08] and [Bos10, page 210] leads to a slightly better cost for the Cauchy- and Vandermonde-like structures. Nevertheless, our approach has the advantage to be direct for these two structures, contrary to the approach in [BJS08] which reduces them to the Toeplitz-like case.

1.5.3 Software development

In addition to our theoretical results above, we have implemented the algorithms introduced in Chapters 2 and 3 within the `C++` library SLA[5] (Structured Linear Algebra). This library already provided general support for Stein's displacement operator, as well as a compression routine. We have added to it about 3000 lines of code, offering the following features:

- some support for the Cauchy-like structure (operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$) and the Hankel-like structure (operator $\nabla[\mathbb{Z}_{m,0}, \mathbb{Z}_{n,0}^T]$);
- an implementation for five inversion algorithms (three variants of `MBA`, `GenInvLT` from Section 2.3.1, and `GenInvHL` from Section 2.3.4);
- extra support for polynomial matrices;
- an implementation of Algorithms `mpx`, `mpy`, and `ComputeRy` (see Section 3.3).

This code is used for three kinds of experiments:

- First, we compare for the Cauchy-like structure (with a naive implementation of the multiplication "Cauchy-like matrix × vectors") the costs of the various inversion algorithms implemented. The theoretical speed-ups announced above
- Second, we measure the time spent on irregularity-set handling within Algorithm `GenInvHL`, so as to confirm experimentally that, compared to the overall cost, it is negligible in practice.
- Third, we measure the impact of using a fast multiplication "Cauchy-like matrix × vectors" in `GenInvLT`. This yields significant speed-ups.

[5] `https://gforge.inria.fr/projects/sla/`

Chapter 2

Compression-free inversion of structured matrices

The goal of this chapter is to extend a compression-free algorithm by Cardinal [Car99, Car00] for Cauchy-like matrix inversion to a broader class of structured matrices. In a first section, we will discuss the idea underlying Cardinal's approach, that is, the computation of a specified generator for the inverse of a Cauchy-like matrix, rather than just an arbitrary one of length α. Then, we will show in a second section that, using specified generators, we can design a general compression-free version of `MBA`, and that we only need to deal with three structures that are Cauchy-, Vandermonde-, and Hankel-like in order to obtain a specified generator for any invertible matrix with one of the nine structures mentioned in Table 1.2. Next, we introduce in a third section two new compression-free algorithms: `GenInvLT` for the inversion of Cauchy- and Vandermonde-like matrices, and `GenInvHL` which is a slight extension of the previous algorithm in order to cover the Hankel-like case. A detailed cost analysis will be provided for each of these three cases. Finally, experimental results are reported in a fourth section. For the Cauchy-like structure, for instance, the speed-ups compared to `MBA` are by a factor from 4.6 to 6.7. This suggests that our extension of Cardinal's compression-free approach may yield algorithms that are not only simpler but also significantly faster in practice.

Most of the work presented in this chapter has been published in [JM10b].

2.1 Techniques to avoid compression stages

As mentioned in Section 1.4.3, classical algorithms for structured matrix inversion are often based on two compression stages to cope with a growth in length of the intermediate generators. We review here two techniques that make it possible to avoid this compression:

1. When the displacement operator is $\nabla[\mathsf{M},\mathsf{N}]$, and when M and N^T are lower triangular, one can use an explicit formula in order to obtain a generator for the Schur complement that has already the expected length. Thus, the first compression stage can be avoided;

2. For the Cauchy-like case with displacement operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$, an algorithm by Cardinal managed to also suppress the second compression stage.

2.1.1 Generation of the Schur complement without compression

Let us partition A, M, N, G, and H as in Section 1.4.2. The authors of [GO94b, Theorem 2.3] (see also [OP98, Lemma 3.1] and [Pan01, §5.4]) remarked that, when $\mathsf{M}_{12} = 0$ and $\mathsf{N}_{21} = 0$, one can perform the block Gauss elimination described in Section 1.1.2 directly on the generator. More precisely, starting with $\mathsf{MA} - \mathsf{AN} = \mathsf{GH}^T$, we may premultiply by $\mathsf{E} = \begin{bmatrix} \mathbb{I}_{n_1} & \\ -\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathbb{I}_{n_2} \end{bmatrix}$, postmultiply by $\mathsf{F} = \begin{bmatrix} \mathbb{I}_{n_1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12} \\ & \mathbb{I}_{n_2} \end{bmatrix}$, and then look at the bottom-right blocks in order to obtain the following property:

Property 2.1. *Let $\mathsf{A} \in \mathbb{K}^{n\times n}$ be such that its top-left block $\mathsf{A}_{11} \in \mathbb{K}^{n_1\times n_1}$ is invertible, and let (G, H) be a $\nabla[\mathsf{M}, \mathsf{N}]$-generator for A. If M_{12} and N_{21} are zero matrices then one has*

$$\mathsf{M}_{22}\mathsf{S} - \mathsf{S}\mathsf{N}_{22} = \mathsf{G}_\mathsf{S}\mathsf{H}_\mathsf{S}^T,$$

where $\mathsf{G}_\mathsf{S} = \mathsf{G}_2 - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{G}_1$ and $\mathsf{H}_\mathsf{S} = \mathsf{H}_2 - \mathsf{A}_{12}^T\mathsf{A}_{11}^{-T}\mathsf{H}_1$.

Therefore, using the last two equations and given any generators for A_{11}^{-1}, A_{12} and A_{21}, one can compute a $\nabla[\mathsf{M}_{22}, \mathsf{N}_{22}]$-generator for the Schur complement S whose length is α. As it is the expected length for a generator of S in `MBA`, the compression stage before the recursive call on the Schur complement is no more needed. Notice that, since we perform recursive calls, we may want to suppress the aforementioned compression stage for each of these calls. This can be achieved if M_{11}, M_{22}, N_{11}^T and N_{22}^T are also block lower triangular, and if their respective diagonal blocks are in turn lower triangular, and so on recursively. As a consequence, the correct assumption for a complete suppression of the first compression stage becomes that M and N^T are lower triangular.

More general formulas

In fact, one does not need to assume that M_{12} and N_{21} are zero matrices in order to get a direct formula for G_S and H_S. Indeed, let μ and ν be the rank of M_{12} and N_{21}, respectively. Hence, as already mentioned in (1.10), we can write $\mathsf{M}_{12} = \mathsf{U}_1\mathsf{V}_2^T$ and $\mathsf{N}_{21} = \mathsf{U}_2\mathsf{V}_1^T$ with $\mathsf{U}_1 \in \mathbb{K}^{n_1\times\mu}$, $\mathsf{V}_2 \in \mathbb{K}^{n_2\times\mu}$, $\mathsf{U}_2 \in \mathbb{K}^{n_2\times\nu}$, and $\mathsf{V}_1 \in \mathbb{K}^{n_1\times\nu}$. Then, [Pan00, Proposition 4.5] gives us the following general description of the structure of the Schur complement S of A_{11} in A:

$$\nabla[\mathsf{M}_{22}, \mathsf{N}_{22}](\mathsf{S}) = \mathsf{G}_\mathsf{S}\mathsf{H}_\mathsf{S}^T,$$

with G_S and H_S the two matrices in $\mathbb{K}^{n_2\times(\alpha+\mu+\nu)}$ given by

$$\mathsf{G}_\mathsf{S} = \left[\ \mathsf{G}_2 - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{G}_1 \ \middle|\ \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{U}_1 \ \middle|\ -\mathsf{S}\mathsf{U}_2\ \right], \tag{2.1a}$$

$$\mathsf{H}_\mathsf{S} = \left[\ \mathsf{H}_2 - \mathsf{A}_{12}^T\mathsf{A}_{11}^{-T}\mathsf{H}_1 \ \middle|\ \mathsf{S}^T\mathsf{V}_2 \ \middle|\ \mathsf{A}_{12}^T\mathsf{A}_{11}^{-T}\mathsf{V}_1\ \right]. \tag{2.1b}$$

Notice that the displacement rank of S for operator $\nabla[\mathsf{M}_{22}, \mathsf{N}_{22}]$ is not equal to α in all generality. Yet, the previous equations tell us that it is at most $\alpha + \mu + \nu$.

2.1.2 Cardinal's algorithm for Cauchy-like matrix inversion

When the displacement operator considered is $\nabla[\mathbb{D}(x), \mathbb{D}(y)]$, that is, when considering Cauchy-like matrices, Cardinal [Car99, Car00] proposed an inversion algorithm that completely avoids the compression stages of `MBA` (yet, the asymptotic cost remains unchanged). His approach is based on two mutually-recursive routines (see Algorithms 2.1 and 2.2):

- `GenSchur` returns the generator for the Schur complement as in Property 2.1, along with a generator for the submatrix A_{11}^{-1};
- `Invert` returns the generator for A^{-1} as defined in Property 1.8, that is, $Y = -A^{-1}G$ and $Z = A^{-T}H$, where (G, H) is the generator provided as input for A.

Algorithm 2.1: `GenSchur` (from [Car00])

```
Input      : $x, y \in \mathbb{K}^n$ and $G, H \in \mathbb{K}^{n\times\alpha}$ such that $\nabla[\mathbb{D}(x), \mathbb{D}(y)](A) = GH^T$, and
             two positive integers $n_1$ and $n_2$ such that $n_1 + n_2 = n$.
Assumption : $A$ is strongly regular, and $x_1, \dots, x_n, y_1, \dots, y_n$ are pairwise distinct.
Output     : $Y_{11} = -A_{11}^{-1}G_1$, $Z_{11} = A_{11}^{-T}H_1$, $G_S = G_2 - A_{21}A_{11}^{-1}G_1$, and
             $H_S = H_2 - A_{12}^T A_{11}^{-T}H_1$.
1 $\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \leftarrow G$ with $G_i \in \mathbb{K}^{n_i\times\alpha}$;   $\begin{bmatrix} H_1 \\ H_2 \end{bmatrix} \leftarrow H$ with $H_i \in \mathbb{K}^{n_i\times\alpha}$
2 $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leftarrow x$ with $x_i \in \mathbb{K}^{n_i}$;   $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \leftarrow y$ with $y_i \in \mathbb{K}^{n_i}$
3 if $n_1 \le \alpha$ then
4     Evaluate $G_1H_1^T$ and deduce the matrix $A$
5     $Y_{11} \leftarrow -A_{11}^{-1}G_1$;   $Z_{11} \leftarrow A_{11}^{-T}H_1$
6 else
7     $(Y_{11}, Z_{11}) \leftarrow$ Invert$(x_1, y_1, G_1, H_1)$
8 $G_S \leftarrow G_2 + A_{21}Y_{11}$;   $H_S \leftarrow H_2 - A_{12}^T Z_{11}$
9 return $Y_{11}$, $Z_{11}$, $G_S$, and $H_S$
```

The key point for this approach to work lies in the precise specification of the output for `Invert`. Instead of computing any length-α generator for A^{-1}, we get a specific one that is precisely related to the input of the routine. This has several consequences:

- First, the formulas from Property 2.1 can be simplified as shown at line 8 in Algorithm 2.1, since we know that $Y_{11} = -A_{11}^{-1}G_1$ and $Z_{11} = A_{11}^{-T}H_1$;
- Second, it can be checked (see [Car00, Proposition 6]) that the pair $(\widetilde{G}, \widetilde{H})$ computed at line 4 in `Invert` satisfies

  $$\nabla[\mathbb{D}(\widetilde{x}), \mathbb{D}(\widetilde{y})](\widetilde{A}) = \widetilde{G}\widetilde{H}^T, \qquad \text{where} \qquad \widetilde{A} = \begin{bmatrix} S & A_{21}A_{11}^{-1} \\ -A_{11}^{-1}A_{12} & A_{11}^{-1} \end{bmatrix},$$

  so that it can be viewed as a generator for the specific Cauchy-like matrix $\widetilde{A}$;

Algorithm 2.2: `Invert` (from [Car00])

```
Input      : $x, y \in \mathbb{K}^n$ and $G, H \in \mathbb{K}^{n \times \alpha}$ such that $\nabla[\mathbb{D}(x), \mathbb{D}(y)](A) = GH^T$.
Assumption: $A$ is strongly regular, and $x_1, \ldots, x_n, y_1, \ldots, y_n$ are pairwise distinct.
Output     : $Y = -A^{-1}G$ and $Z = A^{-T}H$, so that $\nabla[\mathbb{D}(y), \mathbb{D}(x)](A^{-1}) = YZ^T$.

Choose two positive integers $n_1$ and $n_2$ such that $n = n_1 + n_2$
$Y_{11}$, $Z_{11}$, $G_S$, $H_S \leftarrow$ GenSchur($x, y, G, H, n_1, n_2$)
$\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] \leftarrow x$ with $x_i \in \mathbb{K}^{n_i}$;     $\left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right] \leftarrow y$ with $y_i \in \mathbb{K}^{n_i}$
$\widetilde{G} \leftarrow \left[\begin{smallmatrix} G_S \\ Y_{11} \end{smallmatrix}\right]$;   $\widetilde{H} \leftarrow \left[\begin{smallmatrix} H_S \\ Z_{11} \end{smallmatrix}\right]$;   $\widetilde{x} \leftarrow \left[\begin{smallmatrix} x_2 \\ y_1 \end{smallmatrix}\right]$;   $\widetilde{y} \leftarrow \left[\begin{smallmatrix} y_2 \\ x_1 \end{smallmatrix}\right]$
$\widetilde{Y}_{11}$, $\widetilde{Z}_{11}$, $\widetilde{G}_S$, $\widetilde{H}_S \leftarrow$ GenSchur($\widetilde{x}, \widetilde{y}, \widetilde{G}, \widetilde{H}, n_2, n_1$)
$Y \leftarrow \begin{bmatrix} \widetilde{G}_S \\ \widetilde{Y}_{11} \end{bmatrix}$;   $Z \leftarrow \begin{bmatrix} \widetilde{H}_S \\ \widetilde{Z}_{11} \end{bmatrix}$
return $Y$ and $Z$
```

- Third, using the same reasoning as in the second point, we can prove that the pair (Y, Z) computed at the end of `Invert` is actually a generator for A^{-1}.

Notice that, while the minimal assumption for Cauchy-like matrix reconstruction (for Sylvester's displacement operator) is that $x_i \neq y_j$ for all $1 \leq i, j \leq n$, Cardinal's approach also asks that $x_i \neq x_j$ and $y_i \neq y_j$ for $i \neq j$. This unusual assumption is a consequence of the entries of x and y being mixed in the definitions of $\widetilde{x}$ and $\widetilde{y}$. A way to get rid of this stronger assumption, along with the gain we can obtain when it holds, will be discussed in Section 2.3.2.

2.1.3 Formulas for the generator of a Cauchy-like matrix inverse

The precise specification of the output plays a central role in Cardinal's algorithm. Therefore, let us introduce the following definition:

Definition 2.1. *Let $A \in \mathbb{K}^{n \times n}$ be invertible, and (G, H) be a $\nabla[M, N]$-generator for A of length α. We will call* **specified generator** *of A^{-1} associated to (G, H) the pair (Y, Z) defined by*

$$Y = -A^{-1}G \qquad \textit{and} \qquad Z = A^{-T}H.$$

In particular, (Y, Z) is a $\nabla[N, M]$-generator for A^{-1} of length α, and we will usually refer to it as the specified generator for A^{-1} when G and H can be determined by the context.

It should be noted that this formula, which is a straightforward consequence of a remark in [HR84], was explicitly stated in [Pan89]. Yet, superfast algorithms for structured matrix inversion did not really exploit it, except Cardinal's one.

Our goal in the sequel of this chapter will be to use specified generators as much as possible, and to design algorithms similar to `MBA` but, following Cardinal's approach, without the compression stages. To achieve this, let us first reconsider this approach so as to deduce more explicit formulas for the generator (Y, Z) returned by the routine `Invert`.

The first call to `GenSchur` gives us the specified generator (Y_{11}, Z_{11}) for A_{11}^{-1} and a generator (G_S, H_S) for the Schur complement S. Similarly, the second call to `GenSchur` gives us the specified generator $(\widetilde{\mathsf{Y}_{11}}, \widetilde{\mathsf{Z}_{11}})$ for $\widetilde{\mathsf{A}_{11}}^{-1} = \mathsf{S}^{-1}$, which we shall denote by $(\mathsf{Y}_\mathsf{S}, \mathsf{Z}_\mathsf{S})$ hereafter for consistency. Moreover, we obtain

$$\begin{aligned}\widetilde{\mathsf{G}}_\mathsf{S} &= \widetilde{\mathsf{G}}_2 - \widetilde{\mathsf{A}}_{21}\widetilde{\mathsf{A}}_{11}^{-1}\widetilde{\mathsf{G}}_1 \\ &= \mathsf{Y}_{11} + \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}\mathsf{G}_\mathsf{S} \\ &= \mathsf{Y}_{11} - \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{Y}_\mathsf{S}\end{aligned}$$

and

$$\begin{aligned}\widetilde{\mathsf{Z}}_\mathsf{S} &= \widetilde{\mathsf{H}}_2 - \widetilde{\mathsf{A}}_{12}^T\widetilde{\mathsf{A}}_{11}^{-T}\widetilde{\mathsf{H}}_1 \\ &= \mathsf{Z}_{11} - \mathsf{A}_{11}^{-T}\mathsf{A}_{21}^T\mathsf{S}^{-T}\mathsf{H}_\mathsf{S} \\ &= \mathsf{Z}_{11} - \mathsf{A}_{11}^{-T}\mathsf{A}_{21}^T\mathsf{Z}_\mathsf{S}.\end{aligned}$$

Therefore, we deduce the following new formulas for Y and Z:

$$\mathsf{Y} = \begin{bmatrix}\mathsf{Y}_{11} - \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{Y}_\mathsf{S} \\ \mathsf{Y}_\mathsf{S}\end{bmatrix} \qquad \text{and} \qquad \mathsf{Z} = \begin{bmatrix}\mathsf{Z}_{11} - \mathsf{A}_{11}^{-T}\mathsf{A}_{21}^T\mathsf{Z}_\mathsf{S} \\ \mathsf{Z}_\mathsf{S}\end{bmatrix}.$$

Using the matrices $\mathsf{E} = \begin{bmatrix}\mathbb{I}_{n_1} & \\ -\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathbb{I}_{n_2}\end{bmatrix}$ and $\mathsf{F} = \begin{bmatrix}\mathbb{I}_{n_1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12} \\ & \mathbb{I}_{n_2}\end{bmatrix}$, we can rewrite the last two equations as:

$$\mathsf{Y} = \mathsf{F}\begin{bmatrix}\mathsf{Y}_{11} \\ \mathsf{Y}_\mathsf{S}\end{bmatrix} \qquad \text{and} \qquad \mathsf{Z} = \mathsf{E}^T\begin{bmatrix}\mathsf{Z}_{11} \\ \mathsf{Z}_\mathsf{S}\end{bmatrix}.$$

As a consequence, the absence of compression for the generator of A^{-1} in Cardinal's approach appears to be an opposite process of the one from Property 2.1: In order to avoid the need for compression before the second recursive call in `MBA`, we performed an elimination on the generator; Here, the compression stage following the second recursive call is suppressed by some kind of recursive reconstruction formulas.

2.2 Computations with specified generators

This section presents two results involving the concept of specified generators:

- First, we formalize and generalize the analysis performed in the previous section. We then obtain a new, compression-free formulation of the `MBA` algorithm;
- Second, we recall some classical reductions that we will use to restrict our future analysis to only three structures, and we show that, as soon as these three structures are handled correctly, we can compute specified generators for all the nine structures mentioned in Section 1.3.2.

Before we start, let us just introduce a useful notation:

Notation 2.1. *For $\mathsf{X} \in \mathbb{K}^{m\times n}$ and $\alpha \leq n$, we write $\mathsf{X}^{\mapsto\alpha}$ for the matrix $[\mathsf{x}_1|\cdots|\mathsf{x}_\alpha] \in \mathbb{K}^{m\times\alpha}$ made of the first α columns of X.*

2.2.1 Recursive factorization formula

At the end of Section 2.1.3, we have deduced from Cardinal's approach two recursive formulas for the specified generator of a Cauchy-like matrix inverse. Yet, the Cauchy-like structure did not play a major role in the reasoning, and the formulas can actually be proved for any structure defined by a displacement operator $\nabla[\mathsf{M},\mathsf{N}]$ where M and N are such that M_{12} and N_{21} are zero matrices. In fact, like for Property 2.1, the condition on M and N is not mandatory, and a more general result can be stated without it. This yields the following theorem:

Theorem 2.1. *Let* $\mathsf{A} \in \mathbb{K}^{n\times n}$ *be nonsingular and* (G,H) *be a* $\nabla[\mathsf{M},\mathsf{N}]$*-generator for* A. *Assume that* A_{11} *is nonsingular as well, that it is generated by* G_{11} *and* H_{11} *as in (1.11), and let*

$$\mathsf{Y}_{11} = -\mathsf{A}_{11}^{-1}\mathsf{G}_{11}, \qquad \mathsf{Z}_{11} = \mathsf{A}_{11}^{-T}\mathsf{H}_{11}.$$

Assume further that the Schur complement S *of* A_{11} *in* A *is generated by* G_S *and* H_S *as in (2.1), and let*

$$\mathsf{Y}_\mathsf{S} = -\mathsf{S}^{-1}\mathsf{G}_\mathsf{S}, \qquad \mathsf{Z}_\mathsf{S} = \mathsf{S}^{-T}\mathsf{H}_\mathsf{S}.$$

Then the matrices Y *and* Z *of the specified generator of* A^{-1} *satisfy*

$$\mathsf{Y} = \mathsf{F}\begin{bmatrix}\mathsf{Y}_{11}^{\mapsto\alpha}\\ \mathsf{Y}_\mathsf{S}^{\mapsto\alpha}\end{bmatrix}, \qquad \mathsf{Z} = \mathsf{E}^T\begin{bmatrix}\mathsf{Z}_{11}^{\mapsto\alpha}\\ \mathsf{Z}_\mathsf{S}^{\mapsto\alpha}\end{bmatrix},$$

where $\mathsf{E} = \begin{bmatrix}\mathbb{I}_{n_1} & \\ -\mathsf{A}_{21}\mathsf{A}_{11}^{-1} & \mathbb{I}_{n_2}\end{bmatrix}$ *and* $\mathsf{F} = \begin{bmatrix}\mathbb{I}_{n_1} & -\mathsf{A}_{11}^{-1}\mathsf{A}_{12}\\ & \mathbb{I}_{n_2}\end{bmatrix}$ *are the classical block Gaussian elimination matrices.*

Proof. Since $\mathsf{A}^{-1} = \mathsf{F}\begin{bmatrix}\mathsf{A}_{11}^{-1} & \\ & \mathsf{S}^{-1}\end{bmatrix}\mathsf{E}$, we obtain

$$-\mathsf{A}^{-1}\mathsf{G} = \mathsf{F}\begin{bmatrix}-\mathsf{A}_{11}^{-1}\mathsf{G}_1\\ -\mathsf{S}^{-1}(\mathsf{G}_2 - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{G}_1)\end{bmatrix}.$$

It follows from (1.11a) and (2.1a) that $\mathsf{G}_1 = \mathsf{G}_{11}^{\mapsto\alpha}$ and that $\mathsf{G}_2 - \mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathsf{G}_1 = \mathsf{G}_\mathsf{S}^{\mapsto\alpha}$. The expression claimed for $\mathsf{Y} = -\mathsf{A}^{-1}\mathsf{G}$ then follows from applying the rule $\mathsf{A}(\mathsf{B}^{\mapsto\alpha}) = (\mathsf{AB})^{\mapsto\alpha}$ twice, and from the definitions of Y_{11} and Y_S. The expression for Z can be obtained in a similar way, using (1.11b) and (2.1b). □

A first consequence of this theorem is a "compressed" analogue of the classical recursive factorization formula (1.2):

$$\mathsf{Y}\,\mathsf{Z}^T = \mathsf{F}\begin{bmatrix}\mathsf{Y}_{11}^{\mapsto\alpha}\\ \mathsf{Y}_\mathsf{S}^{\mapsto\alpha}\end{bmatrix}\begin{bmatrix}\mathsf{Z}_{11}^{\mapsto\alpha}\\ \mathsf{Z}_\mathsf{S}^{\mapsto\alpha}\end{bmatrix}^T\mathsf{E}.$$

Indeed, this recursive formula allows to factor a specified generator of the inverse for A in terms of specified generators for the inverse of its upper-left block A_{11} and for the inverse of the Schur complement of A_{11} in A.

A second consequence of Theorem 2.1 is that, for A strongly regular, we immediately get a recursive algorithm *à la* `MBA` whose key steps are the computation of generators

Figure 2.1: General compression-free approach for structured matrix inversion.

Given a generator (G,H) of length α for A,

1. Compute a generator $(\mathsf{G}_{11},\mathsf{H}_{11})$ for A_{11} using (1.11);
2. Recursively, compute $(\mathsf{Y}_{11},\mathsf{Z}_{11}) = (-\mathsf{A}_{11}^{-1}\mathsf{G}_{11},\mathsf{A}_{11}^{-T}\mathsf{H}_{11})$;
3. Compute a generator $(\mathsf{G}_{\mathsf{S}},\mathsf{H}_{\mathsf{S}})$ for S using (2.1);
4. Recursively, compute $(\mathsf{Y}_{\mathsf{S}},\mathsf{Z}_{\mathsf{S}}) = (-\mathsf{S}^{-1}\mathsf{G}_{\mathsf{S}},\mathsf{S}^{-T}\mathsf{H}_{\mathsf{S}})$;
5. Compute $(-\mathsf{A}^{-1}\mathsf{G},\mathsf{A}^{-T}\mathsf{H})$ from the first α columns of Y_{11}, Y_{S}, Z_{11}, and Z_{S} using Theorem 2.1.

$(\mathsf{G}_{11},\mathsf{H}_{11})$ and $(\mathsf{G}_{\mathsf{S}},\mathsf{H}_{\mathsf{S}})$, and their associated specified generators. This approach is illustrated in Figure 2.1.

Since we have replaced arithmetic on structured matrices with explicit formulas to compute intermediate generators, our approach is "compression-free." Yet, we have seen in Section 1.4.3 that in order to achieve an asymptotic cost of $\tilde{O}(\alpha^2\, n)$ with this approach, we need to ensure that recursive calls are performed on generators of length α, which is guaranteed by neither (1.11) nor (2.1) in general. However, this holds when M and N^T are lower triangular. Therefore, our next step will be to see how we can come down to this situation.

2.2.2 Reduction to M and N^T lower triangular, and A strongly regular

We want to compute specified generators for any invertible matrix A structured with respect to the displacement operator $\nabla[\mathsf{M},\mathsf{N}]$ where $\mathsf{M},\mathsf{N} \in \{\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\varphi}, \mathbb{Z}_{n,\psi}^T\}$. The purpose of this section is:

1. to prove that we can come down to the three following types of structure:

$$(\mathsf{M},\mathsf{N}) \in \Big\{(\mathbb{D}(\mathsf{x}),\mathbb{D}(\mathsf{y})),\ (\mathbb{D}(\mathsf{x}),\mathbb{Z}_{n,0}),\ (\mathbb{Z}_{n,0},\mathbb{Z}_{n,0}^T)\Big\}, \tag{2.2}$$

   that is, to Cauchy-, Vandermonde- and Hankel-like structures;

2. to show that the classical technique used to ensure the strong regularity of the matrix in input is compatible with the concept of specified generator.

We will see that, in both cases, the initial matrix A we want to invert will be replaced with a matrix $\widetilde{\mathsf{A}} \in \{\mathsf{A}^T, \mathsf{P}_1\mathsf{A}\mathsf{P}_2\}$, where P_1 and P_2 are two $n \times n$ matrices. Therefore, let us first see how one can deduce the specified generator for A^{-1} given the specified generator for $\widetilde{\mathsf{A}}^{-1}$.

Recovery after matrix transformations

Applying twice the rule for the product of two structured matrices mentioned in Property 1.6, we can straightforwardly deduce explicit formulas for generating products of three matrices:

Lemma 2.1. *Let* $\mathsf{A}, \mathsf{G}, \mathsf{H}$ *be such that* $\nabla[\mathsf{M}, \mathsf{P}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ *and, for two matrices* P_1 *and* P_2*, let* $\widetilde{\mathsf{A}} = \mathsf{P}_1\mathsf{A}\mathsf{P}_2$*. If* $\nabla[\widetilde{\mathsf{M}}, \mathsf{M}](\mathsf{P}_1) = \mathsf{G}_{\mathsf{P}_1}\mathsf{H}_{\mathsf{P}_1}^T$ *and* $\nabla[\mathsf{N}, \widetilde{\mathsf{N}}](\mathsf{P}_2) = \mathsf{G}_{\mathsf{P}_2}\mathsf{H}_{\mathsf{P}_2}^T$ *then*

$$\nabla[\widetilde{\mathsf{M}}, \widetilde{\mathsf{N}}](\widetilde{\mathsf{A}}) = \widetilde{\mathsf{G}}\widetilde{\mathsf{H}}^T,$$

where $\mathsf{G} = [\mathsf{P}_1\mathsf{G}|\mathsf{G}_{\mathsf{P}_1}|\mathsf{P}_1\mathsf{A}\mathsf{G}_{\mathsf{P}_2}]$ *and* $\mathsf{H} = [\mathsf{P}_2^T\mathsf{H}|\mathsf{P}_2^T\mathsf{A}^T\mathsf{H}_{\mathsf{P}_1}|\mathsf{H}_{\mathsf{P}_2}]$.

As an example, let us mention three special cases which we will use later: assuming $\mathsf{M}, \mathsf{N} \in \{\mathbb{Z}_{n,\varphi}, \mathbb{Z}_{n,\varphi}^T\}$, let first

$$(\mathsf{P}_1, \mathsf{P}_2) = (\mathbb{I}_n, \mathbb{J}_n) \quad \text{and} \quad (\widetilde{\mathsf{M}}, \widetilde{\mathsf{N}}) = (\mathsf{M}, \mathsf{N}^T).$$

Then obviously $\nabla[\widetilde{\mathsf{M}}, \mathsf{M}](\mathsf{P}_1)$ is zero and, using the facts that $\mathbb{J}_n^2 = \mathbb{I}_n$ and $\mathbb{J}_n\mathbb{Z}_{n,\varphi}\mathbb{J}_n = \mathbb{Z}_{n,\varphi}^T$ (see [Pan01, p. 24]), we deduce that $\nabla[\mathsf{N}, \widetilde{\mathsf{N}}](\mathsf{P}_2)$ is zero as well. Consequently, since $\mathbb{J}_n$ is symmetric, applying Lemma 2.1 above yields

$$\nabla[\mathsf{M}, \mathsf{N}^T](\mathsf{A}\,\mathbb{J}_n) = \mathsf{G}(\mathbb{J}_n\mathsf{H})^T. \tag{2.3a}$$

Similarly, exchanging the roles of P_1 and P_2 yields

$$\nabla[\mathsf{M}^T, \mathsf{N}](\mathbb{J}_n\mathsf{A}) = (\mathbb{J}_n\mathsf{G})\mathsf{H}^T, \tag{2.3b}$$

while taking $\mathsf{P}_1 = \mathsf{P}_2 = \mathbb{J}_n$ gives

$$\nabla[\mathsf{M}^T, \mathsf{N}^T](\mathbb{J}_n\mathsf{A}\,\mathbb{J}_n) = (\mathbb{J}_n\mathsf{G})(\mathbb{J}_n\mathsf{H})^T. \tag{2.3c}$$

Property 1.5 in Section 1.4.2 and Lemma 2.1 above provide formulas for generating the matrix $\widetilde{\mathsf{A}} \in \{\mathsf{A}^T, \mathsf{P}_1\mathsf{A}\mathsf{P}_2\}$ from some generators of the matrix A. Conversely, we give in the theorem below some formulas for recovering specified generators of the inverse of A from specified generators of the inverse of $\widetilde{\mathsf{A}}$.

Theorem 2.2. *Let* $\mathsf{A} \in \mathbb{K}^{n\times n}$ *be invertible,* (G, H) *be a* $\nabla[\mathsf{M}, \mathsf{N}]$*-generator of length* α *for* A*, and* (Y, Z) *be the specified generator for* A^{-1} *associated to* (G, H)*. Let* $\widetilde{\mathsf{A}} \in \mathbb{K}^{n\times n}$ *be invertible and, for* $\widetilde{\mathsf{G}}, \widetilde{\mathsf{H}} \in \mathbb{K}^{n\times\beta}$*,* $\beta \geq \alpha$*, define*

$$\widetilde{\mathsf{Y}} = -\widetilde{\mathsf{A}}^{-1}\widetilde{\mathsf{G}}, \qquad \widetilde{\mathsf{Z}} = \widetilde{\mathsf{A}}^{-T}\widetilde{\mathsf{H}}.$$

Then

- *for* $\widetilde{\mathsf{A}} = \mathsf{A}^T$ *and* $(\widetilde{\mathsf{G}}, \widetilde{\mathsf{H}}) = (-\mathsf{H}, \mathsf{G})$*, one has*

$$\mathsf{Y} = -\widetilde{\mathsf{Z}}, \qquad \mathsf{Z} = \widetilde{\mathsf{Y}};$$

- *for* $\widetilde{\mathsf{A}} = \mathsf{P}_1\mathsf{A}\mathsf{P}_2$ *with* $\mathsf{P}_1, \mathsf{P}_2 \in \mathbb{K}^{n\times n}$ *invertible, and for* $\widetilde{\mathsf{G}}, \widetilde{\mathsf{H}}$ *as in Lemma 2.1, one has*
$$\mathsf{Y} = \mathsf{P}_2\widetilde{\mathsf{Y}}^{\mapsto\alpha}, \qquad \mathsf{Z} = \mathsf{P}_1^T\widetilde{\mathsf{Z}}^{\mapsto\alpha}.$$

Proof. In the first case, $\widetilde{\mathsf{Y}} = -(\mathsf{A}^{-T})(-\mathsf{H}) = \mathsf{A}^{-T}\mathsf{H} = \mathsf{Z}$ and $\widetilde{\mathsf{Z}} = (\mathsf{A}^T)^{-T}\mathsf{G} = \mathsf{A}^{-1}\mathsf{G} = -\mathsf{Y}$. Now, in the case where $\widetilde{\mathsf{A}} = \mathsf{P}_1\mathsf{A}\mathsf{P}_2$ Lemma 2.1 implies that the first α columns of $\widetilde{\mathsf{Y}}$ are $\widetilde{\mathsf{Y}}^{\mapsto\alpha} = -(\mathsf{P}_1\mathsf{A}\mathsf{P}_2)^{-1}\mathsf{P}_1\mathsf{G} = \mathsf{P}_2^{-1}\mathsf{Y}$. Similarly, the first α columns of $\widetilde{\mathsf{Z}}$ are $\widetilde{\mathsf{Z}}^{\mapsto\alpha} = (\mathsf{P}_1\mathsf{A}\mathsf{P}_2)^{-T}\mathsf{P}_2^T\mathsf{H} = \mathsf{P}_1^{-T}\mathsf{Z}$. □

For example, when $\mathsf{P}_1, \mathsf{P}_2 \in \{\mathbb{I}_n, \mathbb{J}_n\}$, it follows from Lemma 2.1 that $\beta = \alpha$. Consequently, Theorem 2.2 yields

$$(\mathsf{Y},\ \mathsf{Z}) = (\mathbb{J}_n\widetilde{\mathsf{Y}},\ \widetilde{\mathsf{Z}}) \qquad \text{if} \quad \widetilde{\mathsf{A}} = \mathsf{A}\,\mathbb{J}_n, \tag{2.4a}$$
$$(\mathsf{Y},\ \mathsf{Z}) = (\widetilde{\mathsf{Y}},\ \mathbb{J}_n\widetilde{\mathsf{Z}}) \qquad \text{if} \quad \widetilde{\mathsf{A}} = \mathbb{J}_n\mathsf{A}, \tag{2.4b}$$
$$(\mathsf{Y},\ \mathsf{Z}) = (\mathbb{J}_n\widetilde{\mathsf{Y}},\ \mathbb{J}_n\widetilde{\mathsf{Z}}) \qquad \text{if} \quad \widetilde{\mathsf{A}} = \mathbb{J}_n\mathsf{A}\,\mathbb{J}_n. \tag{2.4c}$$

Reduction to basic displacements

A first consequence of Theorem 2.2, when it comes to computing specified inverse generators, is that the nine possible displacements defined in (1.5) can be reduced to the three basic ones in (2.2).

First, it follows from (2.3a) and (2.4a) that the case $\mathsf{N} = \mathbb{Z}_{n,\psi}$ reduces to the case $\mathsf{N} = \mathbb{Z}_{n,\psi}^T$. Similarly, (2.3b) and (2.4b) imply that the case $\mathsf{M} = \mathbb{Z}_{n,\varphi}^T$ reduces to the case $\mathsf{M} = \mathbb{Z}_{n,\varphi}$. We thus are left with the four cases defined by

$$\mathsf{M} \in \{\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\varphi}\} \quad \text{and} \quad \mathsf{N} \in \{\mathbb{D}(\mathsf{y}), \mathbb{Z}_{n,\psi}^T\}.$$

Using the transposition rule from Property 1.5 allows to further reduce the case where $\mathsf{M} = \mathbb{Z}_{n,\varphi}$ and $\mathsf{N} = \mathbb{D}(\mathsf{y})$ to the case where $\mathsf{M} = \mathbb{D}(\mathsf{y})$ and $\mathsf{N} = \mathbb{Z}_{n,\varphi}^T$. Due to the nature of the transformations applied to the $n\times\alpha$ generators (sign changes, permutations), the three reductions done so far imply an extra cost of only $O(\alpha\, n)$ operations in $\mathbb{K}$. To reach (2.2) it remains to zero out the scalars φ and ψ. This can be done without transforming A, but only its displacement: for example, by combining the obvious identity

$$\mathbb{Z}_{n,\varphi} = \mathbb{Z}_{n,0} + \varphi\, \mathsf{e}_{n,1}\, \mathsf{e}_{n,n}^T, \tag{2.5}$$

with $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\psi}^T](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ and $\nabla[\mathbb{Z}_{n,\varphi}, \mathbb{Z}_{n,\psi}^T](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$, we arrive at, respectively,

$$\nabla[\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,0}^T](\mathsf{A}) = \widetilde{\mathsf{G}}\widetilde{\mathsf{H}}^T \qquad (i)$$

with $\widetilde{\mathsf{G}} = [\mathsf{G}|\mathsf{A}\mathsf{e}_{n,n}]$ and $\widetilde{\mathsf{H}} = [\mathsf{H}|\psi\, \mathsf{e}_{n,1}]$, and

$$\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T](\mathsf{A}) = \widetilde{\mathsf{G}}\widetilde{\mathsf{H}}^T \qquad (ii)$$

with $\widetilde{\mathsf{G}} = [\mathsf{G}| - \varphi\, \mathsf{e}_{n,1}|\mathsf{A}\mathsf{e}_{n,n}]$ and $\widetilde{\mathsf{H}} = [\mathsf{H}|\mathsf{A}^T\mathsf{e}_{n,n}|\psi\mathsf{e}_{n,1}]$. The last column or row of A needed to set up the matrices $\widetilde{\mathsf{G}}$ and $\widetilde{\mathsf{H}}$ can be computed in $O(\alpha\, \mathsf{M}(n)\log(n))$ —case (i)— or $O(\alpha\, \mathsf{M}(n))$ —case (ii)— field operations from the explicit bilinear expressions of A given in [Pan01, Examples 4.4.4 and 4.4.6(d)]. Due to the shape of $\widetilde{\mathsf{G}}$, $\widetilde{\mathsf{H}}$ above, extracting the first α columns of $\widetilde{\mathsf{Y}} = \mathsf{A}^{-1}\widetilde{\mathsf{G}}$ and $\widetilde{\mathsf{Z}} = \mathsf{A}^{-T}\widetilde{\mathsf{H}}$ in time $O(\alpha\, n)$ then yields the desired specified inverse generator (Y, Z).

Probabilistic reduction to strong regularity

Theorem 2.2 further allows to restrict to matrices that are not only invertible but strongly regular. Strong regularity, which is needed to apply Theorem 2.1 recursively, is classically obtained by preconditioning A into $\widetilde{A} = P_1AP_2$ with two random structured matrices P_1 and P_2 (see [Pan01, §5.6]). Thus, one may generate $\widetilde{A}$ as in Lemma 2.1, then compute an associated specified generator $(\widetilde{Y}, \widetilde{Z})$ of its inverse, and finally recover via Theorem 2.2 a specified generator (Y, Z) of the inverse of A.

Let r_1 and r_2 be two random vectors in $\mathbb{K}^n$ and whose first entry equals 1. Then, applying the rules of [Pan01, p. 167], possible preconditioners for each of the three basic displacements of (2.2) are as follows (with $\widetilde{x}, \widetilde{y}$ in $\mathbb{K}^n$ and such that $\widetilde{x}_i \neq x_i$ and $\widetilde{y}_i \neq y_i$ for all i):

M, N	P_1	P_2
$\mathbb{D}(x)$, $\mathbb{D}(y)$	$\mathbb{C}(\widetilde{x}, x)\,\mathbb{D}(r_1)$	$\mathbb{C}(y, \widetilde{y})\,\mathbb{D}(r_2)$
$\mathbb{D}(x)$, $\mathbb{Z}_{n,0}^T$	$\mathbb{C}(\widetilde{x}, x)\,\mathbb{D}(r_1)$	$\mathbb{L}(r_2)$
$\mathbb{Z}_{n,0}$, $\mathbb{Z}_{n,0}^T$	$\mathbb{U}(r_1)$	$\mathbb{L}(r_2)$

For all these cases, one may check that the structure of A, P_1, and P_2 allows to prepare $(\widetilde{G}, \widetilde{H})$ in Lemma 2.1 and to recover (Y, Z) in Theorem 2.2 in time $O(\alpha\, M(n))$ or $O(\alpha\, M(n) \log(n))$.

2.3 Compression-free structured matrix inversion

In order to cover simultaneously the three displacements in (2.2) to which we have previously reduced, we assume in this section that both operator matrices M and N^T are lower triangular.

2.3.1 Algorithms for lower triangular operator matrices M and N^T

Since M and N^T are lower triangular, we have in particular that the blocks M_{12} and N_{21} are zero, so that their respective ranks μ and ν satisfy $\mu = \nu = 0$. From (1.11) it then follows that the submatrix A_{11} satisfies

$$\nabla[M_{11}, N_{11}](A_{11}) = G_1 H_1^T. \tag{2.6}$$

Thus, some generators of length at most α for A_{11} can be read off the first n_1 rows of some generators of length at most α for A.

Assuming that A_{11} is invertible, consider now the associated specified generator of A_{11}^{-1}, that is,

$$Y_{11} = -A_{11}^{-1} G_1, \qquad Z_{11} = A_{11}^{-T} H_1. \tag{2.7}$$

Combining the two identities in (2.7) with the explicit Schur complement generation formulas from Property 2.1 yields

$$\nabla[M_{22}, N_{22}](S) = (G_2 + A_{21} Y_{11})(H_2 - A_{12}^T Z_{11})^T. \tag{2.8}$$

In other words, the precise specification of the above generator of the inverse of A_{11} can be exploited to simplify even further the generator of the Schur complement. In [Car99, Proposition 1], Cardinal had already noted this formula but only for the Cauchy-like structure (M and N diagonal).

Now, assuming further that A is strongly regular (which, if randomization is allowed, makes sense in view of the probabilistic reductions to strong regularity shown in Section 2.2.2), we obtain Algorithm 2.3.

Algorithm 2.3: `GenInvLT`

```
Input      : $\mathsf{M}, \mathsf{N} \in \mathbb{K}^{n \times n}$ and $\mathsf{G}, \mathsf{H} \in \mathbb{K}^{n \times \alpha}$ such that $\mathsf{M}$ and $\mathsf{N}^T$ are lower
             triangular, and $\nabla[\mathsf{M}, \mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$.
Assumption: $\mathsf{A}$ is strongly regular, and $m_{ii} \neq n_{jj}$ for all $(i, j)$.
Output     : $\mathsf{Y} = -\mathsf{A}^{-1}\mathsf{G}$ and $\mathsf{Z} = \mathsf{A}^{-T}\mathsf{H}$, so that $\nabla[\mathsf{N}, \mathsf{M}](\mathsf{A}^{-1}) = \mathsf{Y}\mathsf{Z}^T$.

if $n = 1$ then
    Evaluate the dot product $\mathsf{G}\mathsf{H}^T$
    Deduce the scalar $\mathsf{A}$
    $\mathsf{Y} \leftarrow -\mathsf{A}^{-1}\mathsf{G}$;    $\mathsf{Z} \leftarrow \mathsf{A}^{-T}\mathsf{H}$
else
    $n_1 \leftarrow \lceil \frac{n}{2} \rceil$;    $n_2 \leftarrow \lfloor \frac{n}{2} \rfloor$
    $\mathsf{G}_{11} \leftarrow \mathsf{G}_1$;    $\mathsf{H}_{11} \leftarrow \mathsf{H}_1$
    $(\mathsf{Y}_{11}, \mathsf{Z}_{11}) \leftarrow$ GenInvLT($\mathsf{M}_{11}, \mathsf{N}_{11}, \mathsf{G}_{11}, \mathsf{H}_{11}$)
    $\mathsf{G}_\mathsf{S} \leftarrow \mathsf{G}_2 + \mathsf{A}_{21}\mathsf{Y}_{11}$;    $\mathsf{H}_\mathsf{S} \leftarrow \mathsf{H}_2 - \mathsf{A}_{12}^T\mathsf{Z}_{11}$
    $(\mathsf{Y}_\mathsf{S}, \mathsf{Z}_\mathsf{S}) \leftarrow$ GenInvLT($\mathsf{M}_{22}, \mathsf{N}_{22}, \mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S}$)
    $\mathsf{Y} \leftarrow \begin{bmatrix} \mathsf{Y}_{11} - \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{Y}_\mathsf{S} \\ \mathsf{Y}_\mathsf{S} \end{bmatrix}$;    $\mathsf{Z} \leftarrow \begin{bmatrix} \mathsf{Z}_{11} - \mathsf{A}_{11}^{-T}\mathsf{A}_{21}^T\mathsf{Z}_\mathsf{S} \\ \mathsf{Z}_\mathsf{S} \end{bmatrix}$
return $\mathsf{Y}$ and $\mathsf{Z}$
```

Theorem 2.3. *Algorithm* `GenInvLT` *is correct.*

Proof. When $n = 1$, the assumption on M and N implies that A is the scalar $\dfrac{\sum_{i=1}^{\alpha} g_{1i}h_{1i}}{m_{11} - n_{11}}$. Correctness then follows immediately in this case. Assume now that $n > 1$ and, in order to proceed by induction, assume correctness for $n' < n$. The matrix A_{11} is strongly regular (since A is) and it satisfies (2.6), where, by assumption M_{11} and N_{11}^T are both lower triangular and with disjoint diagonals. Since $n_1 < n$, the induction assumption then implies that the pair $(\mathsf{Y}_{11}, \mathsf{Z}_{11})$ returned by the first recursive call is precisely $(-\mathsf{A}_{11}^{-1}\mathsf{G}_1, \mathsf{A}_{11}^{-T}\mathsf{H}_1)$. Therefore, the computed pair $(\mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S})$ satisfies (2.8), where, by assumption, S is strongly regular (since A is) and where M_{22} and N_{22}^T are both lower triangular and have disjoint diagonals. Since $n_2 < n$, the induction assumption implies that the pair $(\mathsf{Y}_\mathsf{S}, \mathsf{Z}_\mathsf{S})$ returned by the second recursive call is exactly $(-\mathsf{S}^{-1}\mathsf{G}_\mathsf{S}, \mathsf{S}^{-T}\mathsf{H}_\mathsf{S})$. The conclusion then follows from Theorem 2.1. □

To implement Algorithm `GenInvLT` and bound its cost, all we need is to be able to evaluate the four matrix products

$$\mathsf{A}_{21}\mathsf{Y}_{11}, \quad \mathsf{A}_{12}^T\mathsf{Z}_{11}, \quad \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{Y}_\mathsf{S}, \quad \mathsf{A}_{11}^{-T}\mathsf{A}_{21}^T\mathsf{Z}_\mathsf{S}. \tag{2.9}$$

In the next subsections, we study the evaluation of those expressions for each of three basic structures of the Cauchy, Vandermonde, and Hankel types. That requires in each case a detailed analysis of the structure of the matrices A_{11}^{-1}, A_{12}, A_{21}, and their transposes. Since in (2.9) there are two ways of parenthesizing the products of three matrices, we will also study the structure of $\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$ and $(\mathsf{A}_{21}\mathsf{A}_{11}^{-1})^T$. The parenthesizations $(\mathsf{A}_{11}^{-1}\mathsf{A}_{12})\mathsf{Y}_\mathsf{S}$ and $(\mathsf{A}_{21}\mathsf{A}_{11}^{-1})^T\mathsf{Z}_\mathsf{S}$ will be referred to as "*Cardinal's trick*" later on, as they have been initially used in [Car99] for the Cauchy-like case.

Cost functions

Algorithm `GenInvLT` essentially requires the ability to efficiently evaluate products of the form AB and $\mathsf{A}^T\mathsf{B}$, where A is a structured matrix, and B consists of one or several vectors. In order to precisely describe its cost in the case of Cauchy-, Vandermonde-, and Hankel-like structures, we introduce the following functions. For the Cauchy-like structure (1.6a), let $\mathsf{MM}_\mathsf{C} : \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{N}_{>0} \to \mathbb{R}_{\geq 0}$ be such that, for $\mathsf{A} \in \mathbb{K}^{n\times n}$ given by the right-hand side of (1.6b) and $\mathsf{B} \in \mathbb{K}^{n\times\beta}$, the products AB and $\mathsf{A}^T\mathsf{B}$ can be computed using at most $\mathsf{MM}_\mathsf{C}(\alpha, n, \beta)$ operations in $\mathbb{K}$. We define the functions MM_V and MM_H in a similar way for, respectively, the Vandermonde-like and Hankel-like structures. Also, when $\beta = \alpha$ we shall simply write $\mathsf{MM}_*(\alpha, n)$, for $* = \mathsf{C}, \mathsf{V}, \mathsf{H}$.

As mentioned in Table 1.1, we have that multiplying $\mathbb{C}(\mathsf{x}, \mathsf{y})^T = -\mathbb{C}(\mathsf{y}, \mathsf{x})$, $\mathbb{C}(\mathsf{x}, \mathsf{y})$, $\mathbb{V}(\mathsf{x}, n)$, or $\mathbb{V}(\mathsf{x}, n)^T$ by a vector can be done in time $O(\mathsf{M}(n) \log(n))$. Hence by a straightforward application of the summation formulas (1.6b), (1.7b), and (1.8b), one has

$$\mathsf{MM}_*(\alpha, n, 1) \in O(\alpha\, \mathsf{M}(n) \log(n)) \quad \text{for } * = \mathsf{C}, \mathsf{V},$$

$$\mathsf{MM}_\mathsf{H}(\alpha, n, 1) \in O(\alpha\, \mathsf{M}(n)).$$

We shall also use the three properties given below:

Lemma 2.2. *Let $k, \ell \in O(1)$. Then*

$$\mathsf{MM}_\mathsf{V}(\alpha + k, n, \alpha) \in \mathsf{MM}_\mathsf{V}(\alpha, n) + O(\alpha\, \mathsf{M}(n) \log(n)), \tag{2.10a}$$

$$\mathsf{MM}_\mathsf{H}(\alpha + k, n, \alpha) \in \mathsf{MM}_\mathsf{H}(\alpha, n) + O(\alpha\, \mathsf{M}(n)), \tag{2.10b}$$

and, for $* = \mathsf{C}, \mathsf{V}, \mathsf{H}$,

$$\mathsf{MM}_*(k\alpha, n, \ell\alpha) \in k\ell\, \mathsf{MM}_*(\alpha, n) + O(\alpha\, n). \tag{2.11}$$

Proof. To get (2.10) note that, for all $*$, $\mathsf{MM}_*(\alpha{+}k, n, \alpha)$ is in $\mathsf{MM}_*(\alpha, n, \alpha){+}\mathsf{MM}_*(k, n, \alpha){+}$ $O(\alpha n)$. Indeed, one can evaluate our sum of $\alpha + k$ products by adding the first α terms and the last k terms separately, and then combining the two intermediate results. Since moreover $\mathsf{MM}_*(k, n, \alpha) \leq \alpha \mathsf{MM}_*(k, n, 1)$, (2.10a) and (2.10b) follow from the complexities of $\mathsf{MM}_\mathsf{V}(\alpha, n)$ and $\mathsf{MM}_\mathsf{H}(\alpha, n)$ mentioned above. To establish (2.11), notice that a sum of $k\alpha$ terms for $\ell\alpha$ vectors can be evaluated via k sums of α terms for α vectors plus a final sum in $O(\alpha\, n)$, repeated ℓ times. □

Finally, we assume as for $\mathsf{M}(n)$ that the functions $\mathsf{MM}_*(\cdot, n)$ are superlinear. This assumption will allow us to simplify the cost bounds of the algorithms of Section 2.3.1 and can be easily supported by "naive" implementations in $O\tilde{~}(\alpha^2 n)$ as those used in Section 2.4.1.

2.3.2 Application to Cauchy-like matrices

We consider here the specialization of Algorithm `GenInvLT` to the Cauchy-like structure defined in (1.6a). Partitioning the two vectors x and y conformally with A yields

$$\mathsf{x} = \begin{bmatrix} \mathsf{x}_1 \\ \mathsf{x}_2 \end{bmatrix}, \quad \mathsf{y} = \begin{bmatrix} \mathsf{y}_1 \\ \mathsf{y}_2 \end{bmatrix}, \quad \mathsf{x}_1, \mathsf{y}_1 \in \mathbb{K}^{n_1}, \quad \mathsf{x}_2, \mathsf{y}_2 \in \mathbb{K}^{n_2}. \tag{2.12}$$

Lemma 2.3. *Let the matrices* $\mathsf{A}, \mathsf{G}, \mathsf{H}, \mathsf{Y}_{11}, \mathsf{Z}_{11}, \mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S}$ *be as in Algorithm* `GenInvLT`. *Then*

- $\nabla[\mathbb{D}(\mathsf{x}_i), \mathbb{D}(\mathsf{y}_j)](\mathsf{A}_{ij}) = \mathsf{G}_i \mathsf{H}_j^T$ *for* $1 \le i, j \le 2$,
- $\nabla[\mathbb{D}(\mathsf{y}_1), \mathbb{D}(\mathsf{x}_1)](\mathsf{A}_{11}^{-1}) = \mathsf{Y}_{11} \mathsf{Z}_{11}^T$,
- $\nabla[\mathbb{D}(\mathsf{y}_1), \mathbb{D}(\mathsf{y}_2)](\mathsf{A}_{11}^{-1} \mathsf{A}_{12}) = -\mathsf{Y}_{11} \mathsf{H}_\mathsf{S}^T$,
- $\nabla[\mathbb{D}(\mathsf{x}_2), \mathbb{D}(\mathsf{x}_1)](\mathsf{A}_{21} \mathsf{A}_{11}^{-1}) = \mathsf{G}_\mathsf{S} \mathsf{Z}_{11}^T$.

Proof. Since $\mathbb{D}(\mathsf{x})$ and $\mathbb{D}(\mathsf{y})$ are diagonal matrices, their off-diagonal blocks are zero, and the first identity follows from Property 1.7. To obtain the second identity, it suffices to pre- and postmultiply by A_{11}^{-1} both sides of the first identity for $(i, j) = (1, 1)$, and then to use the specification of Y_{11} and Z_{11}. Using the multiplication rule from Property 1.6, we deduce further from the first identity for $(i, j) = (1, 2)$ and from the second one that

$$\begin{aligned} \nabla[\mathbb{D}(\mathsf{y}_1), \mathbb{D}(\mathsf{y}_2)](\mathsf{A}_{11}^{-1} \mathsf{A}_{12}) &= \mathsf{Y}_{11} \mathsf{Z}_{11}^T \mathsf{A}_{12} + \mathsf{A}_{11}^{-1} \mathsf{G}_1 \mathsf{H}_2^T \\ &= \mathsf{Y}_{11} (\mathsf{Z}_{11}^T \mathsf{A}_{12} - \mathsf{H}_2^T), \end{aligned}$$

which by definition of H_S equals $-\mathsf{Y}_{11} \mathsf{H}_\mathsf{S}^T$. Similarly,

$$\begin{aligned} \nabla[\mathbb{D}(\mathsf{x}_2), \mathbb{D}(\mathsf{x}_1)](\mathsf{A}_{21} \mathsf{A}_{11}^{-1}) &= \mathsf{G}_2\, \mathsf{H}_1^T \mathsf{A}_{11}^{-1} + \mathsf{A}_{21} \mathsf{Y}_{11} \mathsf{Z}_{11}^T \\ &= (\mathsf{G}_2 + \mathsf{A}_{21} \mathsf{Y}_{11}) \mathsf{Z}_{11}^T, \end{aligned}$$

which by definition of G_S equals $\mathsf{G}_\mathsf{S}\, \mathsf{Z}_{11}^T$. □

Theorem 2.4. *Let n be a power of two and* $\mathsf{M}, \mathsf{N} \in \mathbb{K}^{n \times n}$ *be as in (1.6a). Then Algorithm* `GenInvLT` *requires at most*

$$3 \log(n)\, \mathsf{MM}_\mathsf{C}(\alpha, n) + O(\alpha\, n \log(n))$$

field operations. If the set $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ *has cardinality $2n$ then this bound drops to*

$$2 \log(n)\, \mathsf{MM}_\mathsf{C}(\alpha, n) + O(\alpha\, n \log(n)).$$

Proof. When $n = 1$, $\mathsf{A} = (\sum_{i=1}^{\alpha} g_{1i} h_{1i})/(m_{11} - n_{11})$. Hence A^{-1} can be computed using $2\alpha + 1$ operations in $\mathbb{K}$, and the cost for $n = 1$ is $C(\alpha, 1) := 4\alpha + 2$. Consider now the case $n \ge 2$. Using Lemma 2.3 together with Property 1.5, we see that the matrices A_{11}^{-1}, A_{12}, A_{21}, and their transposes are all of the Cauchy-like structure defined in (1.6a). Furthermore, for each of them a generator of length at most α can deduced in time $O(\alpha\, n)$ from the quantities computed by Algorithm `GenInvLT`. Consequently, one can compute

$\mathsf{A}_{21}\mathsf{Y}_{11}$, $\mathsf{A}_{12}^T\mathsf{Z}_{11}$, $\mathsf{A}_{11}^{-1}(\mathsf{A}_{12}\mathsf{Y}_\mathsf{S})$, and $\mathsf{A}_{11}^{-T}(\mathsf{A}_{21}^T\mathsf{Z}_\mathsf{S})$ via six applications, in dimension $n/2$, of the reconstruction formula (1.6b) to α vectors in $\mathbb{K}^{n/2}$. Finally, Algorithm `GenInvLT` uses $2\,\alpha\, n$ additions to deduce G_S, H_S, and the upper parts of Y and Z. Overall, the cost for $n \geq 2$ thus satisfies

$$C(\alpha, n) \leq 2\,C(\alpha, n/2) + 6\,\mathsf{MM}_\mathsf{C}(\alpha, n/2) + k\,\alpha\, n$$

for some constant k. The superlinearity of $\mathsf{MM}_\mathsf{C}(\cdot, n)$ then yields our first bound.

Assume now that the x_i and y_i are $2n$ pairwise distinct values. From Lemma 2.3 the reconstruction formula (1.6b) can then be applied directly to $\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$ and to the transpose of $\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$, in order to compute $(\mathsf{A}_{11}^{-1}\mathsf{A}_{12})\mathsf{Y}_\mathsf{S}$ and $(\mathsf{A}_{21}\mathsf{A}_{11}^{-1})^T\mathsf{Z}_\mathsf{S}$. This reduces the number of reconstructions from six to four, whence the second cost bound. □

2.3.3 Application to Vandermonde-like matrices

Let us now focus on the cost of Algorithm `GenInvLT` when M and N correspond to the Vandermonde-like structure defined in (1.7a). We assume x to be partitioned as in (2.12).

Lemma 2.4. *Let the matrices* $\mathsf{A}, \mathsf{G}, \mathsf{H}, \mathsf{Y}_{11}, \mathsf{Z}_{11}, \mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S}$ *be as in Algorithm* `GenInvLT`. *Let also* w_{11} *be the last column of* A_{11} *and* v_{12}^T *be the first row of* $\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$. *Then*

- $\nabla[\mathbb{D}(\mathsf{x}_1), \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{12}) = \mathsf{G}_1\mathsf{H}_2^T + \mathsf{w}_{11}\,\mathsf{e}_{n_2,1}^T$,
- $\nabla[\mathbb{D}(\mathsf{x}_2), \mathbb{Z}_{n_1,0}^T](\mathsf{A}_{21}) = \mathsf{G}_2\mathsf{H}_1^T$,
- $\nabla[\mathbb{Z}_{n_1,0}^T, \mathbb{D}(\mathsf{x}_1)](\mathsf{A}_{11}^{-1}) = \mathsf{Y}_{11}\mathsf{Z}_{11}^T$,
- $\nabla[\mathbb{D}(\mathsf{x}_2), \mathbb{D}(\mathsf{x}_1)](\mathsf{A}_{21}\mathsf{A}_{11}^{-1}) = \mathsf{G}_\mathsf{S}\mathsf{Z}_{11}^T$,
- $\nabla[\mathbb{Z}_{n_1,1}^T, \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{11}^{-1}\mathsf{A}_{12}) = -\mathsf{Y}_{11}\mathsf{H}_\mathsf{S}^T + \mathsf{e}_{n_1,n_1}(\mathsf{e}_{n_2,1} + \mathsf{v}_{12})^T$.

Proof. In this case, the upper-right block of N satisfies $\mathsf{N}_{12} = \mathsf{e}_{n_1,n_1}\mathsf{e}_{n_2,1}^T$. Hence we deduce from $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ that $\nabla[\mathbb{D}(\mathsf{x}_1), \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{12}) = \mathsf{G}_1\mathsf{H}_2^T + \mathsf{A}_{11}\mathsf{e}_{n_1,n_1}\mathsf{e}_{n_2,1}^T$ and the first identity follows from the definition of vector w_{11}. The second to fourth identities are obtained in the same way as in the proof of Lemma 2.3. Let us now verify the last identity, which displays the structure of the product $\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$. First, applying the techniques of Lemma 2.3, we deduce that

$$\nabla[\mathbb{Z}_{n_1,0}^T, \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{11}^{-1}\mathsf{A}_{12}) = -\mathsf{Y}_{11}\mathsf{H}_\mathsf{S}^T + \mathsf{e}_{n_1,n_1}\mathsf{e}_{n_2,1}^T.$$

Then, using (2.5) with $(\varphi, n) = (1, n_1)$ together with the definition of v_{12} yields the announced expression. □

Theorem 2.5. *Let n be a power of two and* $\mathsf{M}, \mathsf{N} \in \mathbb{K}^{n\times n}$ *be as in (1.7a). Then Algorithm* `GenInvLT` *requires at most*

$$3\log(n)\,\mathsf{MM}_\mathsf{V}(\alpha, n) + O(\alpha\,\mathsf{M}(n)\log^2(n))$$

field operations. If, in addition, the set $\{x_1, \ldots, x_n\}$ has cardinality n then this bound drops to

$$2\log(n)\,\mathsf{MM}_\mathsf{V}(\alpha, n) + O(\alpha\,\mathsf{M}(n)\log^2(n)).$$

Proof. When $n = 1$, $\mathsf{A}^{-1} = x_1/(\sum_{i=1}^{\alpha} g_{1i}h_{1i})$, so that the cost is $C(\alpha, 1) := 4\alpha + 1$. Assume now that $n \geq 2$. Lemma 2.4 implies that A_{12}, A_{21}, and A_{11}^{-T} share the same Vandermonde-like structure (1.7a) as A and A_{11}. However, A_{12} has displacement rank bounded by $\alpha + 1$ and computing its generator can be done at cost $O(\alpha\,\mathsf{M}(n)\log(n))$ by applying (1.7b) to A_{11}. Hence, for $n \geq 2$,

$$\begin{aligned} C(\alpha, n) \quad \leq \quad & 2\,C(\alpha, n/2) + 4\,\mathsf{MM}_{\mathsf{V}}(\alpha, n/2) \\ & + 2\mathsf{MM}_{\mathsf{V}}(\alpha + 1, n/2, \alpha) + k\,\alpha\,\mathsf{M}(n)\log(n), \end{aligned}$$

for some constant k. From (2.10a), and the superlinearity of $\mathsf{M}(n)$ and $\mathsf{MM}_{\mathsf{V}}(\cdot, n)$, we then deduce the first cost bound.

If all the x_i are distinct then, for $\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$, we proceed as for the Cauchy-like case. For $\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$, note that $\mathbb{J}_{n_1}\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$ is Hankel-like in the sense of (1.8a). Hence, one may first generate the latter matrix in time $O(\alpha\,\mathsf{M}(n)\log(n))$ by obtaining the vector v_{12} after two applications of (1.7b), then multiply by Y_{S} using (1.8b), and re-apply a reflexion. Thus,

$$\begin{aligned} C(\alpha, n) \quad \leq \quad & 2\,C(\alpha, n/2) + \mathsf{MM}_{\mathsf{V}}(\alpha, n/2) \\ & + \mathsf{MM}_{\mathsf{V}}(\alpha + 1, n/2, \alpha) + \ \mathsf{MM}_{\mathsf{C}}(\alpha, n/2) \\ & + \mathsf{MM}_{\mathsf{H}}(\alpha + 1, n/2, \alpha) + k\,\alpha\,\mathsf{M}(n)\log(n), \end{aligned}$$

for some constant k, and the conclusion follows as before. □

Note that unlike for the Cauchy-like case, if α is small enough then in the cost bounds of Theorem 2.5 both summands have the same order of magnitude.

2.3.4 Extension to Hankel-like matrices

Let us now consider the Hankel-like structure defined by $\mathsf{M} = \mathbb{Z}_{n,0}$ and $\mathsf{N} = \mathbb{Z}_{n,0}^T$. Although M and N^T are lower triangular, Algorithm `GenInvLT` cannot be used directly in this case as the operator $\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T]$ is not invertible. Covering such a structure, however, is interesting in particular as it yields an immediate extension to some Toeplitz-like matrices (see [Pan01, Remark 5.4.4] and our Section 2.2.2).

To cope with the singularity of the displacement operator, some additional data, called *irregularity set* in [Pan01, p. 136], are needed, which typically consist in "a few" entries of A. An irregularity set for $\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T]$ is given by the last row of A. Indeed, for $\mathsf{u}^T = \mathsf{e}_{n,n}^T\mathsf{A}$ we see that $\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ and (2.5) imply

$$\nabla[\mathbb{Z}_{n,1}, \mathbb{Z}_{n,0}^T](\mathsf{A}) = [\mathsf{G} \,|\, \mathsf{e}_{n,1}]\,[\mathsf{H} \,|\, \mathsf{u}]^T, \tag{2.13}$$

so that the matrix A is Hankel-like in the sense of (1.8a), with displacement rank $\alpha + 1$. Consequently, the reconstruction formula (1.8b) can be used.

We need to exhibit an irregularity set for $\nabla[\mathbb{Z}_{n,0}^T, \mathbb{Z}_{n,0}]$ too, because we shall multiply with inverses of Hankel-like matrices. A suitable choice here is $\mathsf{v}^T = \mathsf{e}_{n,1}^T\mathsf{A}^{-1}$, the first row of the inverse of A: indeed, if $\nabla[\mathbb{Z}_{n,0}^T, \mathbb{Z}_{n,0}](\mathsf{A}^{-1}) = \mathsf{Y}\mathsf{Z}^T$ then, recalling (2.3c), we may check that $\mathbb{J}_n\mathsf{A}^{-1}\mathbb{J}_n$ satisfies an identity similar to (2.13); it is thus fully determined by, up to reflexions, Y, Z, and its last row $\mathsf{v}^T\mathbb{J}_n$.

The resulting adaptation of Algorithm `GenInvLT` to the Hankel-like operator $\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T]$ is as follows:

Algorithm 2.4: `GenInvHL`

```
Input      : $G, H \in \mathbb{K}^{n\times\alpha}$ such that $\nabla[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T](A) = GH^T$, and $u = A^T e_{n,n}$ (the
             last row of $A$).
Assumption: $A$ is strongly regular.
Output     : $Y = -A^{-1}G$, $Z = A^{-T}H$, and $v = A^{-T}e_{n,1}$ (the first row of $A^{-1}$).

if $n = 1$ then
    $Y \leftarrow -u^{-1}G$;    $Z \leftarrow u^{-1}H$;    $v \leftarrow u^{-1}$
else
    $n_1 \leftarrow \lceil \frac{n}{2} \rceil$;    $n_2 \leftarrow \lfloor \frac{n}{2} \rfloor$
    $\left[\begin{smallmatrix} u_{11} \\ u_{12} \end{smallmatrix}\right] \leftarrow A^T e_{n,n_1}$;    $\left[\begin{smallmatrix} u_{21} \\ u_{22} \end{smallmatrix}\right] \leftarrow u$
    $(Y_{11}, Z_{11}, v_{11}) \leftarrow$ GenInvHL($G_1, H_1, u_{11}$)
    $G_S \leftarrow G_2 + A_{21}Y_{11}$;    $H_S \leftarrow H_2 - A_{12}^T Z_{11}$
    $u_S \leftarrow u_{22} - A_{12}^T A_{11}^{-T} u_{21}$
    $(Y_S, Z_S, v_S) \leftarrow$ GenInvHL($G_S, H_S, u_S$)
    $Y \leftarrow \left[\begin{smallmatrix} Y_{11} - (A_{11}^{-1}A_{12})Y_S \\ Y_S \end{smallmatrix}\right]$;    $Z \leftarrow \left[\begin{smallmatrix} Z_{11} - (A_{11}^{-T}A_{21}^T)Z_S \\ Z_S \end{smallmatrix}\right]$
    $w \leftarrow -S^{-T}A_{12}^T v_{11}$;    $v \leftarrow \left[\begin{smallmatrix} v_{11} - A_{11}^{-T}A_{21}^T w \\ w \end{smallmatrix}\right]$
return $Y$, $Z$, and $v$
```

Theorem 2.6. *Algorithm* `GenInvHL` *is correct.*

Proof. When $n = 1$, both A and u are reduced to the scalar $a_{1,1}$ and correctness is then straightforward. Assume now that $n > 1$ and, in order to proceed by induction, assume correctness for $n' < n$. The vector u is split into $u_{21} \in \mathbb{K}^{n_1}$ and $u_{22} \in \mathbb{K}^{n_2}$. Similarly, the vector of coefficients of row n_1 of A is split into $u_{11} \in \mathbb{K}^{n_1}$ and $u_{12} \in \mathbb{K}^{n_2}$. Hence u_{11} equals $A_{11}^T e_{n_1,n_1}$ (that is, the vector of coefficients of the last row of A_{11}), $u_{22} = A_{22}^T e_{n_2,n_2}$, and $u_{21} = A_{21}^T e_{n_2,n_2}$. Recalling that $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$, we deduce that the vector u_S computed by Algorithm `GenInvHL` satisfies $u_S = S^T e_{n_2,n_2}$ and thus is the vector of coefficients of the last row of S. Since the computation of Y and Z is unchanged in comparison to Algorithm `GenInvLT`, we still have $Y = -A^{-1}G$ and $Z = A^{-T}H$. All that remains is to prove that v is actually the vector of coefficients of the first row of A^{-1}. By induction, v_{11}^T and v_S^T correspond to the first rows of A_{11}^{-1} and S^{-1}, respectively. Using the factorization of A^{-1} seen in (1.2) and letting $w^T = -v_{11}^T A_{12} S^{-1}$, we get:

$$\begin{aligned} e_{n,1}^T A^{-1} &= \begin{bmatrix} e_{n_1,1}^T & -e_{n_1,1}^T A_{11}^{-1} A_{12} \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & \\ & S^{-1} \end{bmatrix} E \\ &= \begin{bmatrix} v_{11}^T & w^T \end{bmatrix} E \\ &= \begin{bmatrix} v_{11}^T - w^T A_{21} A_{11}^{-1} & w^T \end{bmatrix}, \end{aligned}$$

which is exactly the way the vector v is computed. □

Lemma 2.5. *Let* $A, G, H, Y_{11}, Z_{11}, G_S, H_S, u_{11}$ *be as in Algorithm* `GenInvHL`. *Recall that* u_{11} *is the last row of the matrix* A_{11} *and let* w_{11} *be its last column. Then*

- $\nabla[\mathbb{Z}_{,n_1}0, \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{12}) = \mathsf{G}_1\mathsf{H}_2^T + \mathsf{w}_{11}\,\mathsf{e}_{n_2,1}^T$,
- $\nabla[\mathbb{Z}_{n_2,0}, \mathbb{Z}_{n_1,0}^T](\mathsf{A}_{21}) = \mathsf{G}_2\mathsf{H}_1^T - \mathsf{e}_{n_2,1}\,\mathsf{u}_{11}^T$,
- $\nabla[\mathbb{Z}_{n_1,0}^T, \mathbb{Z}_{n_1,0}](\mathsf{A}_{11}^{-1}) = \mathsf{Y}_{11}\mathsf{Z}_{11}^T$,
- $\nabla[\mathbb{Z}_{n_1,0}^T, \mathbb{Z}_{n_2,0}^T](\mathsf{A}_{11}^{-1}\mathsf{A}_{12}) = -\mathsf{Y}_{11}\mathsf{H}_{\mathsf{S}}^T + \mathsf{e}_{n_1,n_1}\mathsf{e}_{n_2,1}^T$,
- $\nabla[\mathbb{Z}_{n_2,0}, \mathbb{Z}_{n_1,0}](\mathsf{A}_{21}\mathsf{A}_{11}^{-1}) = \mathsf{G}_{\mathsf{S}}\mathsf{Z}_{11}^T - \mathsf{e}_{n_2,1}\,\mathsf{e}_{n_1,n_1}^T$.

Proof. Proceed as for Lemma 2.3 and Lemma 2.4. □

Theorem 2.7. *Let n be a power of two and $\mathsf{M}, \mathsf{N} \in \mathbb{K}^{n\times n}$ be as in (1.8a). Then Algorithm* `GenInvHL` *requires at most*

$$2\log(n)\,\mathsf{MM}_{\mathsf{H}}(\alpha, n) + O(\alpha\,\mathsf{M}(n)\log(n)).$$

field operations.

Proof. When $n = 1$, u is a scalar and the algorithm has cost $C(\alpha, 1) := 2\alpha + 2$. Assume now $n \geq 2$. Given G, H, and u, one has (2.13) and thus (1.8b) yields $[\mathsf{u}_{11}^T, \mathsf{u}_{12}^T]$ in time $O(\alpha\,\mathsf{M}(n))$. From Lemma 2.5, all the blocks involved have the same structure as A, up to transposition and row/column reflexion, and with sometimes a displacement rank $\alpha + 1$ instead of α. Generating theses blocks requires the knowledge of the vectors u_{11} (already computed) and w_{11} (computable as u_{11}), which has cost $O(\alpha\,\mathsf{M}(n))$. Now, one may check that the irregularity sets of A_{12}, A_{21}, $\mathbb{J}_{n_1}\mathsf{A}_{11}^{-1}\mathbb{J}_{n_1}$, $\mathbb{J}_{n_1}\mathsf{A}_{11}^{-1}\mathsf{A}_{12}$, $\mathsf{A}_{21}\mathsf{A}_{11}^{-1}\mathbb{J}_{n_1}$, and $\mathbb{J}_{n_2}\mathsf{S}^{-1}\mathbb{J}_{n_2}$ are, respectively, u_{12}, u_{21}, v_{11}, $\mathsf{v}_{11}^T\mathsf{A}_{12}$, $\mathsf{u}_{21}\mathsf{A}_{11}^{-1}\mathbb{J}_{n_1}$, and v_{S}. The vector u_{12} has already been computed, u_{21} is part of the input, v_{11} and v_{S} are computed recursively, and the two remaining vectors can be recovered in time $O(\alpha\,\mathsf{M}(n))$ from u_{21}, v_{11}, and the generators of A_{11}^{-1} and A_{12}. Consequently, all the products that appear in Algorithm `GenInvHL` can be produced by applications of (1.8b). Finally, Algorithm `GenInvHL` still uses $O(\alpha\,n)$ additions, so that the total cost bound is given by

$$C(\alpha, n) \quad \leq \quad 2\,C(\alpha, n/2) + 4\,\mathsf{MM}_{\mathsf{H}}(\alpha + 1, n/2, \alpha) + k\,\alpha\,\mathsf{M}(n),$$

for some constant k. The conclusion follows from (2.10b) and the superlinearity assumptions. □

2.4 Experimental results and concluding remarks

2.4.1 Experimental results

For our experiments, we have implemented several algorithms and structures within the `C++` library SLA[1] (Structured Linear Algebra). This library already provided, among other features, some general support for Stein's displacement operator, a routine for generator compression, one implementation of `MBA` for the Toeplitz-like structure (operator $\Delta[\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}^T]$) and a framework for testing and timing purpose. We have added to it:

[1] Available at `https://gforge.inria.fr/projects/sla/`.

- some support for the Cauchy-like structure (operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$);
- some support for the Hankel-like structure (operator $\nabla[\mathbb{Z}_{m,0}, \mathbb{Z}_{n,0}^T]$);
- an implementation of Pan and Zheng's algorithm [PZ00];
- an implementation of `GenInvLT` and `GenInvHL`;
- and finally, two implementations of `MBA`, the first one being based on the description of `MBA` in [Kal94], and the second one being a slight variation with additional compression stages.

We estimate the size of the code produced to be around 2000 lines.

For all our experiments, we take for $\mathbb{K}$ the finite field $\mathbb{F}_p$ with $p = 999999937$ elements. For basic operations in $\mathbb{K}$, we use the library NTL[2] developed by Shoup, which also provides fast, FFT-based polynomial arithmetic over $\mathbb{K}[x]$. Since operations in $\mathbb{K}$ are achieved in constant time, we are thus able to measure and compare the algebraic costs for the different algorithms. All the computations are carried out on a desktop machine with an Intel® Core™ 2 Duo processor at 2.66 GHz.

Comparison of the different approaches for the Cauchy-like structure

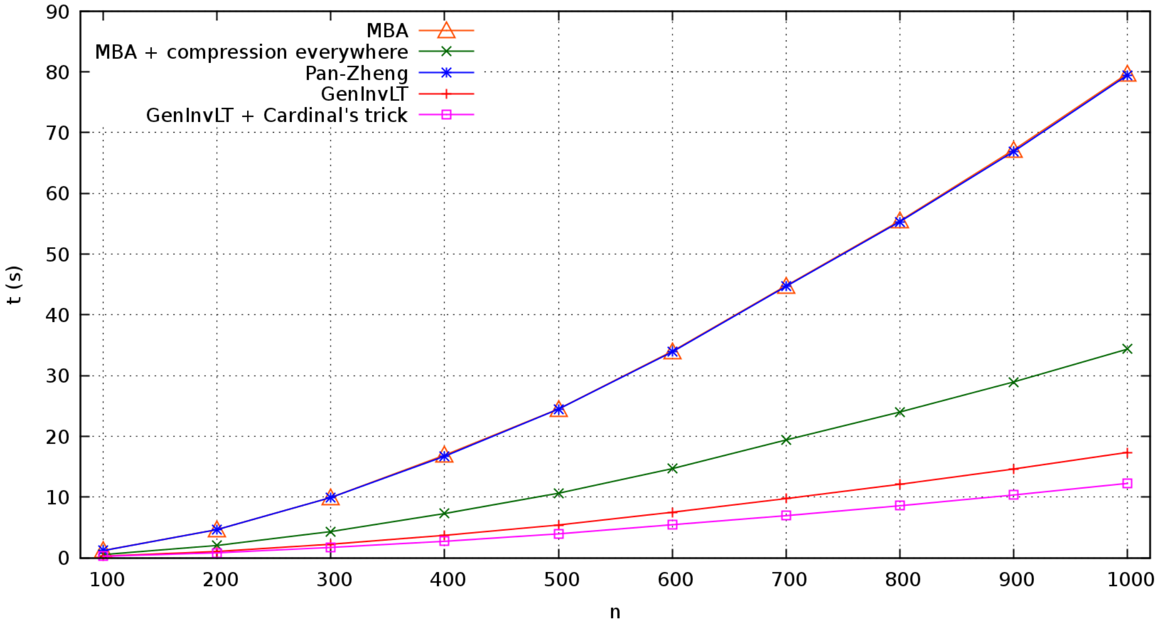
As a first experiment, we measure the algebraic costs of several algorithms for the Cauchy-like structure. Generators (G, H) in input are picked randomly, while operator matrices $\mathbb{D}(\mathsf{x})$, $\mathbb{D}(\mathsf{y})$ are chosen in order to satisfy all the assumptions made on the algorithms. Figure 2.2 shows computing times for inverting Cauchy-like matrices of displacement rank $\alpha = 10$ when n increases.

First, we can remark in Figure 2.2(a) that we obtain a computing time that is quasi-linear with respect to n for each method, as expected. Moreover, as shown by Figure 2.2(b), the time spent in the compression routine for the two variants of `MBA`, and for Pan and Zheng's algorithm confirms that compression is really negligible in practice, compared to the total cost. Yet, compression-free algorithms perform better than algorithms using compression. The main difference explaining the various performances lies in the number and the size of products of the form "Cauchy-like matrix × vectors." We have already seen in Theorem 2.4 that the choice in the parenthesizations leads to one variant in $3\log(n)\,\mathsf{MM}_{\mathsf{C}}(\alpha, n)$ and, up to stronger conditions on the input, to another variant in $2\log(n)\,\mathsf{MM}_{\mathsf{C}}(\alpha, n)$. Let us now estimate this cost for the three other algorithms in Figure 2.2(a).

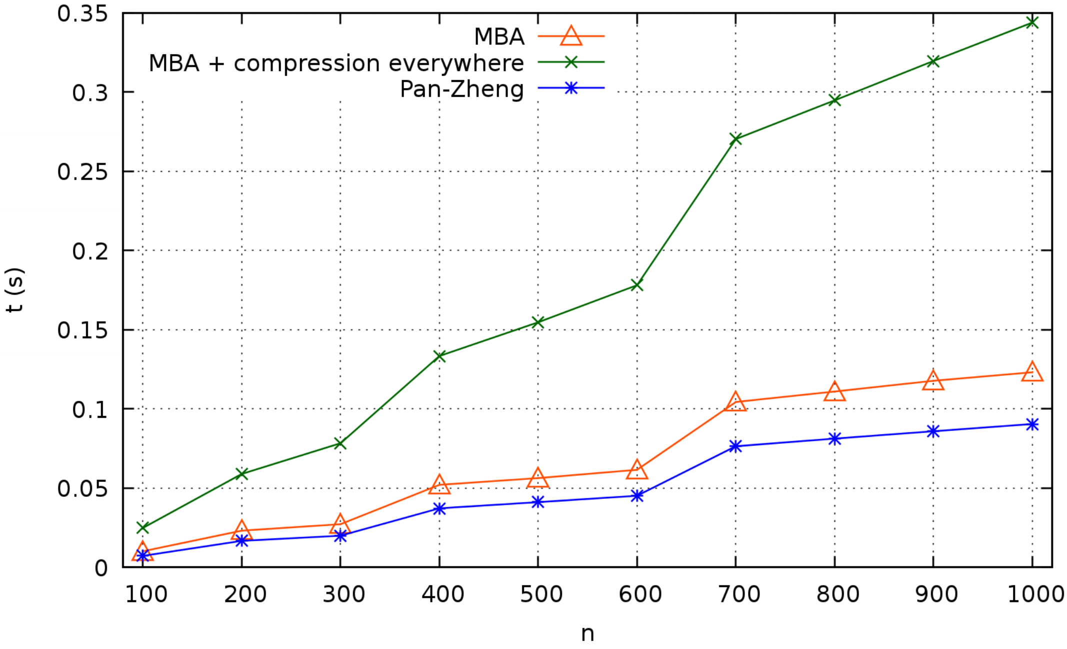
From Property 1.6, we deduce that multiplying an $n \times n$ Cauchy-like matrix of displacement rank α with another $n \times n$ Cauchy-like matrix of displacement rank β costs essentially $\mathsf{MM}_{\mathsf{C}}(\alpha, n, \beta) + \mathsf{MM}_{\mathsf{C}}(\beta, n, \alpha)$. Such multiplications are used in `MBA` in order to compute generators for the Schur complement S and the inverse $\mathsf{A}^{-1} = \left[\begin{smallmatrix} \mathsf{B}_{11} & -\mathsf{B}_{12} \\ -\mathsf{B}_{21} & \mathsf{S}^{-1} \end{smallmatrix}\right]$, where $\mathsf{B}_{11} = \mathsf{A}_{11}^{-1} + \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$, $\mathsf{B}_{12} = \mathsf{A}_{11}^{-1}\mathsf{A}_{12}\mathsf{S}^{-1}$, and $\mathsf{B}_{21} = \mathsf{S}^{-1}\mathsf{A}_{21}\mathsf{A}_{11}^{-1}$. The parenthesization we have chosen for our implementation, along with the dominant cost for each operation and the length of intermediate generators, are presented in Table 2.1.

[2] See `http://www.shoup.net/ntl/`.

Figure 2.2: Cost (in seconds) of Cauchy-like matrix inversion for $\alpha = 10$ and increasing values of n.



(a) Total time.



(b) Compression time only.

Counting the costs of all these products using (2.11) and the superlinearity of $\mathsf{MM}_{\mathsf{C}}(\cdot, n)$ leads to a bound of $14\,\mathsf{MM}_{\mathsf{C}}(\alpha, n)$ in the recurrence equation for the cost of `MBA`, which gives a total cost dominated by $14 \log(n)\,\mathsf{MM}_{\mathsf{C}}(\alpha, n)$. In Figure 2.2(a), we observe a speed-up around $4.6 \approx 14/3$ between `MBA` and our first variant of `GenInvLT`, and around

$6.7 \approx 14/2$ between `MBA` and `GenInvLT` using Cardinal's trick, which is in agreement with our analysis above.

Table 2.1: Dominant cost in our implementation of the `MBA` algorithm.

	operation	resulting generator length	dominant cost
S	$\mathsf{X}_1 = \mathsf{A}_{11}^{-1}\mathsf{A}_{12}$	2α	$2\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2})$
	$\mathsf{S} = \mathsf{A}_{22} - \mathsf{A}_{21}\mathsf{X}_1$	4α	$\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2}, 2\alpha) + \mathsf{MM}_\mathsf{C}(2\alpha, \frac{n}{2}, \alpha)$
A^{-1}	$\mathsf{X}_2 = \mathsf{A}_{21}\mathsf{A}_{11}^{-1}$	2α	$2\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2})$
	$\mathsf{B}_{12} = \mathsf{X}_1\mathsf{S}^{-1}$	3α	$\mathsf{MM}_\mathsf{C}(2\alpha, \frac{n}{2}, \alpha) + \mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2}, 2\alpha)$
	$\mathsf{B}_{21} = \mathsf{S}^{-1}\mathsf{X}_2$	3α	$\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2}, 2\alpha) + \mathsf{MM}_\mathsf{C}(2\alpha, \frac{n}{2}, \alpha)$
	$\mathsf{B}_{11} = \mathsf{A}_{11}^{-1} + \mathsf{B}_{12}\mathsf{X}_2$	6α	$\mathsf{MM}_\mathsf{C}(3\alpha, \frac{n}{2}, 2\alpha) + \mathsf{MM}_\mathsf{C}(2\alpha, \frac{n}{2}, 3\alpha)$

The `Pan-Zheng` variant of `MBA` replaces structured matrix multiplications for the generation of S with the direct application of the formulas $\mathsf{G}_\mathsf{S} = \mathsf{G}_2 - \mathsf{A}_{21}\big(\mathsf{A}_{11}^{-1}\mathsf{G}_1\big)$ and $\mathsf{H}_\mathsf{S} = \mathsf{H}_2 - \mathsf{A}_{12}^T\big(\mathsf{A}_{11}^{-T}\mathsf{H}_1\big)$ from Property 2.1. Thus, they avoid compression for the computation of a generator for S. The pair $(\mathsf{G}_\mathsf{S}, \mathsf{H}_\mathsf{S})$ is obtained in $4\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2})$. Later, the generator for A^{-1} is deduced by generating successively X_1, X_2, B_{12}, B_{21}, and B_{11}, as in Table 2.1. Again, using (2.11) and the superlinearity of $\mathsf{MM}_\mathsf{C}(\cdot, n)$ yields a bound of $14\,\mathsf{MM}_\mathsf{C}(\alpha, n)$ in the recurrence equation for the cost of `Pan-Zheng`, so that the total cost is also dominated by $14\log(n)\,\mathsf{MM}_\mathsf{C}(\alpha, n)$ for this method. Therefore, suppressing the compression stage for the Schur complement is not enough to decrease the constant in the dominating part of the cost, and this is reflected in practice by Figure 2.2(a), where the costs of the `MBA` and `Pan-Zheng` algorithms are overlaid.

In fact, the main factor to achieve speed-ups lies in keeping intermediate generator lengths as small as possible, so as to reduce the costs of the products of the form "Cauchy-like matrix × vectors." This motivates for a second implementation of `MBA`, where a compression stage is systematically added after each generator computation involving an addition or a multiplication of Cauchy-like matrices. Indeed, we have seen in Lemma 2.3 that at least X_1 and X_2 in Table 2.1 possess length-α generators. It appears in practice that it also holds for B_{12} and B_{21}. Hence, with additional compression stages, we obtain a variant of `MBA` where each dominant cost from Table 2.1 drops to $2\mathsf{MM}_\mathsf{C}(\alpha, \frac{n}{2})$. This yields an overall cost for this new method in $6\log(n)\,\mathsf{MM}_\mathsf{C}(\alpha, n) + O(\alpha\, n\, \log(n))$, which is 14/6 times faster than the original `MBA`, but still 3 times slower than the fastest version of `GenInvLT`.

To conclude with this experiment, the main part of the cost in `MBA` and its variations comes from the products "Cauchy-like matrix × vectors." Completely avoiding compression implies that generators for these Cauchy-like matrices is always kept as small as possible, which in turn allows us to obtain speed-ups up to ≈ 6.7 in practice. Finally, it should be noted that all the approaches based on compression are using as intermediate quantities the matrices X_1 and X_2. As remarked in Section 2.3.2, computation with these two matrices can only be carried out under the assumption that $\{\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{y}_1, \ldots, \mathsf{y}_n\}$

has cardinality $2n$. Therefore, the first variant of `GenInvLT`, which is the second fastest method, is also the only one which works under the weaker assumption than $\mathsf{x}_i \neq \mathsf{y}_j$ for all $1 \leq i, j \leq n$. This was also made possible thanks to the usage of direct formulas for the generation of S and A^{-1} in order to suppress the compression stages.

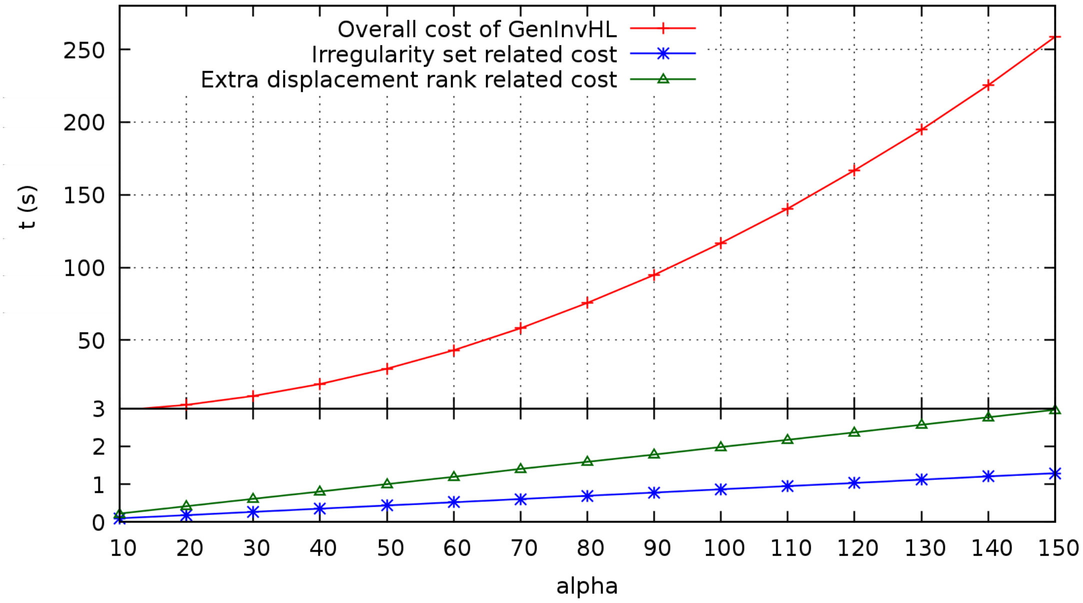
Estimation of the cost for `GenInvHL`

As a second experiment, we measure the computation time of `GenInvHL`, along with:

- the time spent to compute the irregularity sets of all intermediate quantities, that is, additional rows that where needed in order to use the reconstruction formula (1.8b) for the Hankel-like structure;
- the additional time due to subblocks having displacement rank $\alpha + 1$ instead of α like in the Cauchy-like case.

The matrices G and H in input are again picked randomly, and we set $n = 400$ and take increasing values of α. Figure 2.3 shows the result of our experimentation.

Figure 2.3: Cost (in seconds) of Hankel-like matrix inversion using `GenInvHL`, for $n = 400$ and increasing values of α.



First, we can see that the overall cost of `GenInvLT` is quadratic with respect to α, which is in agreement with the cost announced in Theorem 2.7 and the fact that we used a routine for the products “Hankel-like matrix × vectors” such that $\mathsf{MM}_\mathsf{H}(\alpha, n) \in O(\alpha^2\,\mathsf{M}(n))$. Second, the cost of the additional computations for the Hankel-like structure compared to Cauchy-like appears to be negligible. Indeed, the two sources of this extra work have a cost which is linear in α, while we just have seen that the overall cost grows like α^2. In fact, the linear shape of the curve for the cost related to irregularity set issues confirms what we have said in the proof of Theorem 2.7, while the linear shape of the cost due to extra displacement rank for some subblocks was predicted by Equation (2.10b) from Lemma 2.2.

2.4.2 Concluding remarks on our new approach

Let us conclude this chapter with three remarks on our new approach for structured matrix inversion:

1. We have presented in this chapter a general compression-free approach that we have specialized to three structures (Cauchy-, Vandermonde-, and Hankel-like). Yet, some reductions exist among these three structures, see for instance [Pan00, §6]. Therefore, it would be interesting to compare our approach for the three aforementioned structures, and see whether a reduction to the best specialization can improve performance for the two other structures.

2. At the end of Section 2.2.1, we have introduced a general version (see Figure 2.1) of our new approach. We have then remarked that having M and N^T lower triangular is preferable, and so brought down to this situation in Section 2.2.2. However, the reductions we propose therein do not work for the (Toeplitz+Hankel)-like structure, which corresponds to the displacement operator $\nabla[\mathsf{M}, \mathsf{N}]$ with $\mathsf{M} = \mathsf{N} = \mathbb{Z}_{n,0} + \mathbb{Z}_{n,0}^T$. In this case, we can still apply the general approach in Figure 2.1. Indeed, Since the displacement ranks for A_{11} and S are at most $\alpha + 2$ (see (1.11) and (2.1), respectively), we obtain the recursive equation cost

   $$C(\alpha, n) \leq 2\, C(\alpha + 2, n/2) + O(\mathsf{MM}_{\mathsf{T+H}}(\alpha, n/2)).$$

   Assuming that $C(\cdot, n)$ is increasing, and that the product of an $n\times n$ (Toeplitz+Hankel)-like matrix of displacement rank α by an $n\times\alpha$ matrix can be done in $\mathsf{MM}_{\mathsf{T+H}}(\alpha, n) \in O\tilde{~}(\alpha^2\, n)$, this yields a total cost in $O(\mathsf{MM}_{\mathsf{T+H}}(\alpha + 2\log_2(n), n)\ \log_2(n)) \in O\tilde{~}(\alpha^2\, n)$.

3. When analyzing the cost of our new approach for the Cauchy-, Vandermonde-, and Hankel-like structures, it appears that what dominates is the cost of products "structured matrix × vectors," where the number of vectors is of the order of α, the displacement rank of the structured matrix. Moreover, this was confirmed by our experiments. Until now, we have assumed that the cost for this operation satisfies $\mathsf{MM}_*(\alpha, n) \in O\tilde{~}(\alpha^2\, n)$ for each structure. Yet, a better asymptotic of $O\tilde{~}(\alpha^{\omega-1}\, n)$ was achieved in [BJS07, BJS08, Bos10] for the Toeplitz-like structure. The next chapter will extend this result to the other structures.

Chapter 3

Fast multiplication of a structured matrix by a matrix

In this chapter, we study the problem of multiplying an $m \times n$ structured matrix A by an $n \times \beta$ matrix B. In the case of square Toeplitz-like matrices (with Stein's operator), recent works in [BJS07, BJS08] and [Bos10, page 210] have shown that, when β is equal to the displacement rank α of the structured matrix, the product AB can be computed using $O\tilde{~}(\alpha^{\omega-1}\, n)$ field operations. We aim here at generalizing this result to all the structures mentioned in Table 1.2, and for rectangular matrices. After some preliminaries in Section 3.1, we introduce in Section 3.2 a general polynomial expression for the product AB, which generalizes the one proposed in [BJS07, BJS08] for the Toeplitz-like structure. The evaluation of the polynomial expression obtained requires to compute the polynomial row vector $\mathsf{R} = \mathsf{U}^T(\mathsf{VW}^T \bmod P)$, for some $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V} \in \mathbb{K}[x]_n^{\alpha\times 1}$, and $\mathsf{W} \in \mathbb{K}[x]_n^{\beta\times 1}$, and where P is either $x^n - \psi$ or $\prod_{i=1}^n (x - y_i)$. When $\alpha = \beta$ and $P = x^n$, an efficient way to compute R is provided by [Bos10, page 210]. We extend this work in Section 3.3 to the case of $P = x^n - \psi$ and $P = \prod_{i=1}^n (x - y_i)$. Next, we propose in Section 3.4 a new algorithm for the computation of AB. We show that its cost when $\alpha = \beta$ is in $O\tilde{~}(\alpha^{\omega-1}\,(n+m))$, and propose a cost for the general case $\alpha \neq \beta$. Finally, we study the impact of this fast "structured matrix × matrix" routine on the inversion algorithms presented in the previous chapter.

While we focus here on Sylvester's displacement operator, we refer to [BJMS11] for an adaptation to Stein's displacement operator.

3.1 Preliminaries

In this first section, we introduce some material that we will use in the sequel of this chapter.

Multiplication of polynomial matrices

Let us first recall the cost of the multiplication of polynomial matrices.

Property 3.1. *Let* $\mathsf{MM} : \mathbb{N}_{>0} \times \mathbb{N}_{>0} \to \mathbb{R}_{>0}$ *be such that two matrices in* $\mathbb{K}[x]_d^{n\times n}$ *can be multiplied using at most* $\mathsf{MM}(n,d)$ *operations in* $\mathbb{K}$*. By [CK91], one has*

$$\mathsf{MM}(n,d) \in O\big(n^\omega \mathsf{M}(d)\big). \tag{3.1a}$$

If $\mathbb{K}$ *is a field of characteristic zero, or a finite field of cardinality at least* $2d$ *then it was shown in [BS05] that*

$$\mathsf{MM}(n,d) \in O\big(n^\omega d + n^2 \mathsf{M}(d)\big). \tag{3.1b}$$

Cost function $f_{\mathbb{K}}$

The next function is introduced in order to make the intermediate complexity results concise. Let $f_{\mathbb{K}} : \mathbb{N}_{>0} \times \mathbb{N}_{>0} \to \mathbb{R}_{>0}$ be defined as follows: if $\mathbb{K}$ is a field of characteristic zero, or a finite field of cardinality at least $2d$ then

$$f_{\mathbb{K}}(n,d) = n^{\omega-1} d + n \log n \, \mathsf{M}(d), \tag{3.2a}$$

otherwise,

$$f_{\mathbb{K}}(n,d) = n^{\omega-1} \mathsf{M}(d). \tag{3.2b}$$

The operators Pol and rev

Now, let us introduce two operators that we will heavily use when presenting polynomial expressions for the matrix product AB, where A is a structured matrix.

For $m \in \mathbb{N}_{>0}$ and $\mathsf{a} = [a_1|\cdots|a_m]^T \in \mathbb{K}^m$, we write $\mathsf{Pol}_m(\mathsf{a})$ for the polynomial $\sum_{0\le i<m} a_{i+1}x^i \in \mathbb{K}[x]_m$. Conversely, given $a \in \mathbb{K}[x]_m$, the vector of its m coefficients in the monomial basis will be written $\mathsf{Pol}_m^{-1}(a)$.

We shall also use reversals of polynomials [vzGG03, page 254]: for $n \in \mathbb{N}_{>0}$ and $a \in \mathbb{K}[x]_n$, we write $\mathsf{rev}_n(a)$ for the polynomial in $\mathbb{K}[x]_n$ given by $\mathsf{rev}_n(a) = x^{n-1}a(1/x)$. Three properties will be useful in the sequel: For $\mathsf{a} \in \mathbb{K}^n$,

$$\mathsf{Pol}_n(\mathbb{J}_n \mathsf{a}) = \mathsf{rev}_n(\mathsf{Pol}_n(\mathsf{a})). \tag{3.3a}$$

Second, one may check that for $a \in \mathbb{K}[x]_m$ and $b \in \mathbb{K}[x]_n$,

$$a \, \mathsf{rev}_n(b) = \mathsf{rev}_{m+n-1}(\mathsf{rev}_m(a)\, b). \tag{3.3b}$$

Third, one may check that for $a, b \in \mathbb{K}[x]_n$, the quotient in the division of ab by x^n satisfies

$$ab \text{ div } x^n = \mathsf{rev}_n(x \, \mathsf{rev}_n(a) \, \mathsf{rev}_n(b) \bmod x^n). \tag{3.3c}$$

The operators Pol_m, Pol_m^{-1}, and rev_n are linear maps. Also, most of the time we will apply them in a componentwise fashion to sets of vectors or polynomials: for example, for $\mathsf{A} = [\mathsf{a}_1|\cdots|\mathsf{a}_n] \in \mathbb{K}^{m\times n}$ with jth column a_j we write $\mathsf{Pol}_m(\mathsf{A})$ to denote the polynomial row vector $[\mathsf{Pol}_m(\mathsf{a}_1)|\cdots|\mathsf{Pol}_m(\mathsf{a}_n)]$ in $\mathbb{K}[x]_m^{1\times n}$.

Polynomial P_{x} and matrix $\mathbb{W}(\mathsf{x})$

For $\mathsf{x} \in \mathbb{K}^n$, let P_{x} denote the monic polynomial of degree n

$$P_{\mathsf{x}}(x) = \prod_{i=1}^{n}(x - x_i).$$

Let P'_{x} denote the first derivative of P_{x}. If x is repetition free then $P'_{\mathsf{x}}(x_i)$ is nonzero for all i, and we define

$$\mathbb{W}(\mathsf{x}) = \mathbb{D}(P'_{\mathsf{x}}(\mathsf{x}))^{-1}\mathbb{V}(\mathsf{x}).$$

Displacement matrices considered in this chapter

For this chapter, we will assume that:

$$\begin{gathered}\mathsf{M} \in \{\mathbb{D}(\mathsf{x}),\, \mathbb{Z}_{m,\varphi},\, \mathbb{Z}_{m,\varphi}^T\},\ \mathsf{N} \in \{\mathbb{D}(\mathsf{y}),\, \mathbb{Z}_{n,\psi},\, \mathbb{Z}_{n,\psi}^T\} \\ \text{with } \varphi, \psi \in \mathbb{K}, \text{ and } \mathsf{x}, \mathsf{y} \in \mathbb{K}^n \text{ two repetition-free vectors.}\end{gathered} \tag{3.4}$$

Assuming that the vector x encountered in diagonal displacement matrices is repetition-free will allow us to use the matrices $\mathbb{V}(\mathsf{x})^{-1}$ and $\mathbb{W}(\mathsf{x})$.

3.2 Polynomial expressions for structured matrix reconstruction

Let A be an $m \times n$ matrix such that $\nabla[\mathsf{M}, \mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ with (M, N) as in (1.5). Our goal in this section is, given an $n \times \beta$ matrix B, to express the product AB in terms of operations on polynomial matrices. Recall from Lemma 1.2 that one formula to recover A from G and H involves a sum of matrix products involving Krylov matrices and their transposes. Therefore, we start with several formulas involving the operator Pol_*, and the displacement matrices M, N, and their associated Krylov matrices. Then, we use them in order to prove a general expression for AB.

3.2.1 Polynomial expression for products with displacement matrices and their associated Krylov matrices

When applying operator Pol_* to the reconstruction formula of a structured matrix, the main issue becomes to transform expressions with the following shape: $\mathsf{Pol}_m(\mathsf{X}\,\mathsf{u})$, where X is either M, N, a Krylov matrix, or a slight variation of one of the previous matrices. We list here all the formulas that we will need in order to prove Theorem 3.1 in Section 3.2.2.

Case of $\mathbb{Z}_{m,\varphi}$, $\mathcal{K}_n(\mathbb{Z}_{m,\varphi}, \mathsf{u})$, and $\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})^T$

Lemma 3.1. *For any positive integer n, one has*

$$\mathsf{Pol}_m\big(\mathbb{Z}_{m,\varphi}^n \mathsf{u}\big) = x^n \mathsf{Pol}_m(\mathsf{u}) \bmod (x^m - \varphi).$$

This implies that

- *for any* $\psi \in \mathbb{K}$,

$$\mathsf{Pol}_m\big((\mathbb{Z}_{m,\varphi}^n - \psi\mathbb{I}_m)\mathsf{u}\big) = (x^n - \psi)\mathsf{Pol}_m(\mathsf{u}) \bmod (x^m - \varphi); \tag{3.5}$$

- *for* $\mathsf{v} \in \mathbb{K}^n$,

$$\mathsf{Pol}_m\big(\mathcal{K}_n(\mathbb{Z}_{m,\varphi}, \mathsf{u})\mathsf{v}\big) = \mathsf{Pol}_m(\mathsf{u})\mathsf{Pol}_n(\mathsf{v}) \bmod (x^m - \varphi). \tag{3.6}$$

Proof. The first equality, which is well-known [Pan01, §2.6], tells us that applying n times the shift matrix on vector u means multiplying the polynomial $u(x)$ by x^n modulo $(x^m-\varphi)$. It can be easily proved by induction over $n \in \mathbb{N}_{>0}$ by noticing that $\mathsf{Pol}_m(\mathbb{Z}_{m,\varphi}u) = x\mathsf{Pol}_m(\mathsf{u}) \bmod (x^m - \varphi)$. The second and third equations then come from the first point and the linearity of Pol_m. For the third one, we may consider the relation $\mathcal{K}_n(\mathbb{Z}_{m,\varphi}, \mathsf{u})\mathsf{v} = \sum_{i=0}^{n-1} v_i\, \mathbb{Z}_{m,\varphi}^i \mathsf{u}$. □

In the following, we will also encounter products of type $\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})^T\mathsf{v}$. It is easy to check that, for any $\mathsf{u} \in \mathbb{K}^n$, one has

$$\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})^T = \mathbb{J}_n\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})\mathbb{J}_n. \tag{3.7}$$

Hence, a polynomial interpretation can be deduced from the previous lemma and polynomial reversals.

Case of $\mathbb{D}(\mathsf{u})$ and $\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{u})$

Lemma 3.2. *Let* $\mathsf{x}, \mathsf{u}, \mathsf{v} \in \mathbb{K}^m$ *and assume* x *is repetition free. Then*

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathbb{D}(\mathsf{u})\mathsf{v}\big) = \mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{u}\big)\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{v}\big) \bmod P_{\mathsf{x}}. \tag{3.8}$$

Moreover, the three following identities hold:

- *For any* $n \in \mathbb{N}_{>0}$ *and* $\psi \in \mathbb{K}$,

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathbb{D}(\mathsf{x}^n - \psi\mathsf{e}_m)\mathsf{v}\big) = (x^n - \psi)\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{v}\big) \bmod P_{\mathsf{x}}; \tag{3.9}$$

- *For any* $n \in \mathbb{N}_{>0}$ *and* $\mathsf{w} \in \mathbb{K}^n$,

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathbb{V}(\mathsf{x}, n)\mathsf{w}\big) = \mathsf{Pol}_n(\mathsf{w}) \bmod P_{\mathsf{x}}; \tag{3.10}$$

- *For any* $n \in \mathbb{N}_{>0}$ *and* $\mathsf{w} \in \mathbb{K}^n$,

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathcal{K}_n(\mathbb{D}(\mathsf{x}), \mathsf{u})\mathsf{w}\big) = \mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{u}\big)\mathsf{Pol}_n(\mathsf{w}) \bmod P_{\mathsf{x}}. \tag{3.11}$$

Proof. The first identity tells us that scaling and then interpolating a vector v is the same as interpolating both v and the scaling vector u, and taking the product of the resulting polynomials modulo $P_{\mathsf{x}} = \prod_{i=1}^m (x - x_i)$. Indeed, in both cases, we end up with a polynomial of degree less than m whose value in x_i is u_iv_i.

The second identity comes from the fact that $\mathsf{Pol}_m(\mathbb{V}(\mathsf{x})^{-1}\mathsf{u}) = (x^n - \psi) \bmod P_{\mathsf{x}}$ when the vector u satisfies $u_i = x_i^n - \psi$.

The third identity says that evaluating a polynomial $w \in \mathbb{K}[x]_n$ at the x_i's for $1 \le i \le m$ and then interpolating at these same x_i's is the same as taking w modulo P_{x}. To prove it, remark that the polynomials on both sides of the equality are of degree less than m, and since they coincide at $x = x_i$ for $1 \le i \le m$, they are thus equal.

Finally, the fourth identity is obtained by combining the fact that, by definition of Krylov matrices, $\mathcal{K}(\mathbb{D}(\mathsf{x}), \mathsf{u}) = \mathbb{D}(\mathsf{u})\mathbb{V}(\mathsf{x}, n)$ with (3.8) and (3.10). □

Case of $\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{u})^T$

Finally, let us see a more technical formula that will be used when proving Theorem 3.1 for the Vandermonde-like structure.

Lemma 3.3. *Let $\mathsf{y} \in \mathbb{K}^n$ be repetition free, and let $m \in \mathbb{N}_{>0}$ and $\varphi \in \mathbb{K}$ be such that $\varphi \ne y_i^m$ for $i = 1, \dots, n$. Then for $\mathsf{u}, \mathsf{v} \in \mathbb{K}^n$, one has*

$$\mathsf{Pol}_m\big(\mathbb{J}_m\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{u})^T\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m)^{-1}\mathsf{v}\big) = P_{\mathsf{y}}^{-1}\,\mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathbb{D}(\mathsf{u})\mathsf{v}\big) \bmod (x^m - \varphi).$$

Proof. First, by definition of Krylov matrices, one has $\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{u}) = \mathbb{D}(\mathsf{u})\mathbb{V}(\mathsf{y}, m)$, so that

$$\mathbb{J}_m\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{u})^T\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m)^{-1}\mathsf{v} = \mathbb{J}_m\mathbb{V}(\mathsf{y}, m)^T\mathbb{D}(\mathsf{u})\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m)^{-1}\mathsf{v}.$$

Furthermore, for any $\mathsf{w} \in \mathbb{K}^n$, the following equation holds [BLS03, §6.2]:

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{y}, m)^T\mathsf{w}\big) = \sum_{j=1}^{n} w_j Z_j, \quad \text{where } Z_j = (1 - y_j x)^{-1} \bmod x^m.$$

Then, for $\mathsf{w} = \mathbb{D}(\mathsf{u})\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m)^{-1}\mathsf{v}$ and because of (3.3a), we obtain

$$\mathsf{Pol}_m\big(\mathbb{J}_m\mathbb{V}(\mathsf{y}, m)^T\mathsf{w}\big) = \mathsf{rev}_m\left(\sum_{j=1}^{n} w_j Z_j\right) = \sum_{j=1}^{n} \frac{u_j\, v_j}{\varphi - y_j^m}\mathsf{rev}_m(Z_j). \tag{3.12}$$

From the definition of Z_j, we deduce that $Z_j = \sum_{i=0}^{m-1}(y_j x)^i$ so that $\mathsf{rev}_m(Z_j) = \sum_{i=0}^{m-1} y_j^i x^{m-1-i}$. Thus, $(x - y_j)\mathsf{rev}_m(Z_j)$ equals $x^m - y_j^m$, and since $(x^m - y_j^m) \bmod (x^m - \varphi) = \varphi - y_j^m \ne 0$ and $x - y_j$ is invertible modulo $x^m - \varphi$ by assumption, we get

$$(\varphi - y_j^m)^{-1}\mathsf{rev}_m(Z_j) = (x - y_j)^{-1} \bmod (x^m - \varphi). \tag{3.13}$$

By combining (3.12) and (3.13), and since $P_{\mathsf{y}} = \prod_{j=1}^{n}(x - y_j)$ is invertible modulo $x^m - \varphi$, we obtain

$$\mathsf{Pol}_m\big(\mathbb{J}_m\mathbb{V}(\mathsf{y}, m)^T\mathsf{w}\big) = P_{\mathsf{y}}^{-1}\sum_{j=1}^{n} u_j v_j P_{\mathsf{y},j} \bmod (x^m - \varphi),$$

where $P_{\mathsf{y},j} = \dfrac{P_{\mathsf{y}}}{x - y_j} \in \mathbb{K}[x]_n$ for $j = 1, \dots, n$. We can then conclude by applying the following formula from [Pan01, p. 90]:

$$\sum_{j=1}^{n} x_j P_{\mathsf{y},j} = \mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathsf{x}\big)$$

with $\mathsf{x} = \mathbb{D}(\mathsf{u})\mathsf{v}$. □

Notation

For the sequel of this chapter, we will associate one polynomial P_{M} and three matrices $\mathcal{U}_{\mathsf{M}}$, $\mathcal{V}_{\mathsf{M}}$, and $\mathcal{W}_{\mathsf{M}}$ to the displacement matrix M. Their definitions are presented in Table 3.1. Similarly, we associate P_{N}, $\mathcal{U}_{\mathsf{N}}$, $\mathcal{V}_{\mathsf{N}}$, and $\mathcal{W}_{\mathsf{N}}$ to the displacement matrix N.

Table 3.1: Definition of P_{M}, $\mathcal{U}_{\mathsf{M}}$, $\mathcal{V}_{\mathsf{M}}$, and $\mathcal{W}_{\mathsf{M}}$ for a given displacement matrix M.

M	P_{M}	$\mathcal{U}_{\mathsf{M}}$	$\mathcal{V}_{\mathsf{M}}$	$\mathcal{W}_{\mathsf{M}}$
$\mathbb{Z}_{m,\varphi}$	$x^m - \varphi$	$\mathbb{I}_m$	$\mathbb{J}_m$	$\mathbb{I}_m$
$\mathbb{Z}_{m,\varphi}^T$	$x^m - \varphi$	$\mathbb{J}_m$	$\mathbb{I}_m$	$\mathbb{J}_m$
$\mathbb{D}(\mathsf{x})$	P_{x}	$\mathbb{V}(\mathsf{x})$	$\mathbb{V}(\mathsf{x})$	$\mathbb{W}(\mathsf{x})$

We can formulate a few remarks on these definitions:

- Since we assume x to be repetition-free when $\mathsf{M} = \mathbb{D}(\mathsf{x})$, matrices $\mathcal{U}_{\mathsf{M}}$, $\mathcal{V}_{\mathsf{M}}$, and $\mathcal{W}_{\mathsf{M}}$ are always invertible;
- P_{M} is the characteristic polynomial of M;
- $\mathcal{U}_{\mathsf{M}}$ was chosen so that a simple polynomial interpretation for the product $\mathcal{U}_{\mathsf{M}}^{-1}\mathcal{K}_n(\mathsf{M},\mathsf{u})\mathsf{v}$ holds (see (3.6) and (3.11) for instance).

Notice that, since we assume x to be repetition-free when $\mathsf{M} = \mathbb{D}(\mathsf{x})$, matrices $\mathcal{U}_{\mathsf{M}}$, $\mathcal{V}_{\mathsf{M}}$, and $\mathcal{W}_{\mathsf{M}}$ are always invertible.

3.2.2 Polynomial expression of AB for Sylvester's displacement

Theorem 3.1. *Let* $\mathsf{A} \in \mathbb{K}^{m\times n}$ *be such that* $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \mathsf{G}\mathsf{H}^T$ *with* M *and* N *as in* (3.4), *let* $\mathsf{B} \in \mathbb{K}^{n\times\beta}$, *and let* $\mathsf{U}^T = \mathsf{Pol}_m(\mathcal{U}_{\mathsf{M}}^{-1}\mathsf{G})$, $\mathsf{V}^T = \mathsf{Pol}_n(\mathcal{V}_{\mathsf{N}}^{-1}\mathsf{H})$, *and* $\mathsf{W}^T = \mathsf{Pol}_n(\mathcal{W}_{\mathsf{N}}^{-1}\mathsf{B})$. *If* $\gcd(P_{\mathsf{M}}, P_{\mathsf{N}}) = 1$ *then the matrix product* AB *is given by*

$$\mathsf{AB} = \mathcal{U}_{\mathsf{M}}\,\mathsf{Pol}_m^{-1}\Big(P_{\mathsf{N}}^{-1}\mathsf{R} \bmod P_{\mathsf{M}}\Big),$$

with

$$\mathsf{R} = \mathsf{U}^T\Big(\mathsf{V}\mathsf{W}^T \bmod P_{\mathsf{N}}\Big).$$

In order to prove Theorem 3.1, we have to cover nine cases. In fact, we will split them into three classes:

1. the Toeplitz-like class,[1] covering the four cases where M and N are both unit circulant matrices;
2. the Vandermonde-like class,[2] covering the four cases where either M or N is a diagonal matrix, the other being a unit circulant matrix;

[1] The Toeplitz-like class actually covers all the Toeplitz-like and Hankel-like structures from Table 1.2.

[2] The Vandermonde-like class actually covers all the Vandermonde-like and Vandermonde-transposed-like structures from Table 1.2.

3. the Cauchy-like class, covering the last case where M and N are both diagonal matrices.

Notice that, in all cases, the assumption $\gcd(P_\mathsf{M}, P_\mathsf{N}) = 1$ implies that $(\det \mathsf{M}, \det \mathsf{N}) \neq (0,0)$. Indeed, suppose both $\det \mathsf{M}$ and $\det \mathsf{N}$ were zeros, then x would divide both the characteristic polynomials P_M and P_N of M and N, which contradicts $\gcd(P_\mathsf{M}, P_\mathsf{N}) = 1$. Therefore, Lemma 1.2 can be applied for the first two classes.

Proof of Theorem 3.1 for the Toeplitz-like class

In this case $(\mathsf{M}, \mathsf{N}) \in \{\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{m,\varphi}^T\} \times \{\mathbb{Z}_{n,\psi}, \mathbb{Z}_{n,\psi}^T\}$, and thus $P_\mathsf{M} = x^m - \varphi$ and $P_\mathsf{N} = x^n - \psi$. Consider first the subcase where $(\mathsf{M}, \mathsf{N}) = (\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}^T)$. By applying Lemma 1.2 with $\ell = n$, and since $\mathbb{Z}_{n,\psi}^n = \psi \mathbb{I}_n$ and $\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})^T = \mathbb{J}_n \mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u}) \mathbb{J}_n$ for any $\mathsf{u} \in \mathbb{K}^n$, we obtain

$$\mathbb{Z}_{m,\varphi}^n \mathsf{A} - \psi \mathsf{A} = \sum_{j \le \alpha} \mathcal{K}_n(\mathbb{Z}_{m,\varphi}, \mathsf{g}_j) \mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j) \mathbb{J}_n.$$

By multiplying both sides of this equality by B, applying the linear operator Pol_m to every column, and using (3.5) and (3.6), we deduce that

$$P_\mathsf{N} \, \mathsf{Pol}_m(\mathsf{AB}) \equiv \sum_{j \le \alpha} \mathsf{Pol}_m(\mathsf{g}_j) \mathsf{Pol}_n\big(\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j) \mathbb{J}_n \mathsf{B}\big) \bmod P_\mathsf{M}.$$

Since P_N is invertible modulo P_M and since $\mathsf{Pol}_m(\mathsf{AB})$ has degree less than m, we deduce that

$$\mathsf{Pol}_m(\mathsf{AB}) = P_\mathsf{N}^{-1} \sum_{j \le \alpha} \mathsf{Pol}_m(\mathsf{g}_j) \mathsf{Pol}_n\big(\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j) \mathbb{J}_n \mathsf{B}\big) \bmod P_\mathsf{M}.$$

From (3.6) it follows that the right-hand side of the previous identity equals

$$P_\mathsf{N}^{-1} \sum_{j \le \alpha} \mathsf{Pol}_m(\mathsf{g}_j) \Big(\mathsf{Pol}_n(\mathsf{h}_j) \mathsf{Pol}_n(\mathbb{J}_n \mathsf{B}) \bmod P_\mathsf{N}\Big) \bmod P_\mathsf{M},$$

which can be rewritten in matrix form as

$$P_\mathsf{N}^{-1} \mathsf{Pol}_m(\mathsf{G}) \Big(\mathsf{Pol}_n(\mathsf{H})^T \mathsf{Pol}_n(\mathbb{J}_n \mathsf{B}) \bmod P_\mathsf{N}\Big) \bmod P_\mathsf{M}.$$

The desired expression for AB then follows from the fact that $(\mathcal{U}_\mathsf{M}, \mathcal{V}_\mathsf{N}, \mathcal{W}_\mathsf{N}) = (\mathbb{I}_m, \mathbb{I}_n, \mathbb{J}_n)$ when $(\mathsf{M}, \mathsf{N}) = (\mathbb{Z}_{m,\varphi}, \mathbb{Z}_{n,\psi}^T)$.

The other three subcases $(\mathsf{M}, \mathsf{N}^T)$, $(\mathsf{M}^T, \mathsf{N})$, $(\mathsf{M}^T, \mathsf{N}^T)$ can be reduced to the previous one by using (2.3a), (2.3b) and (2.3c), respectively.

Proof of Theorem 3.1 for the Vandermonde-like case

In this case the four possibilities for (M, N) are $(\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\psi}^T)$, $(\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\psi})$, $(\mathbb{Z}_{m,\varphi}, \mathbb{D}(\mathsf{y}))$, and $(\mathbb{Z}_{m,\varphi}^T, \mathbb{D}(\mathsf{y}))$. It suffices to prove the theorem for the first and third subcases, and then to deduce the second and fourth subcases from (2.3a) and (2.3b).

Let $(\mathsf{M}, \mathsf{N}) = (\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\psi}^T)$. Then $P_\mathsf{M} = P_\mathsf{x}$ and $P_\mathsf{N} = x^n - \psi$. We deduce from Lemma 1.2 with $\ell = n$, $\mathbb{Z}_{n,\psi}^n = \psi\mathbb{I}_n$, and $\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})^T = \mathbb{J}_n\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{u})\mathbb{J}_n$ that

$$\mathbb{D}(\mathsf{x})^n\mathsf{A} - \psi\mathsf{A} = \sum_{j\le\alpha} \mathcal{K}_n(\mathbb{D}(\mathsf{x}), \mathsf{g}_j)\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j)\mathbb{J}_n.$$

Recalling that x is repetition free, we can pre- and post-multiply both sides of the identity above by $\mathbb{V}(\mathsf{x})^{-1}$ and B:

$$\mathbb{V}(\mathsf{x})^{-1}\mathbb{D}(\mathsf{x}^n - \psi\mathsf{e}_m)\mathsf{AB} = \sum_{j\le\alpha} \mathbb{V}(\mathsf{x})^{-1}\mathcal{K}_n(\mathbb{D}(\mathsf{x}), \mathsf{g}_j)\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j)\mathbb{J}_n\mathsf{B}.$$

By applying Pol_m, using (3.9) and (3.11), and since P_N is invertible modulo P_M and the entries of $\mathsf{Pol}_m(\mathbb{V}(\mathsf{x})^{-1}\mathsf{AB})$ have a degree less than m, we obtain

$$\mathsf{Pol}_m(\mathbb{V}(\mathsf{x})^{-1}\mathsf{AB}) = P_\mathsf{N}^{-1}\sum_{j\le\alpha} \mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{g}_j\big)\mathsf{Pol}_n\big(\mathcal{K}_n(\mathbb{Z}_{n,\psi}, \mathsf{h}_j)\mathbb{J}_n\mathsf{B}\big) \bmod P_\mathsf{M}.$$

As for the Toeplitz-like case, we deduce from (3.6) that the right-hand side of the identity above can be written in matrix form as

$$P_\mathsf{N}^{-1}\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{G}\big)\big(\mathsf{Pol}_n(\mathsf{H})^T\mathsf{Pol}_n(\mathbb{J}_n\mathsf{B}) \bmod P_\mathsf{N}\big) \bmod P_\mathsf{M}.$$

The assertion follows from the fact that $(\mathcal{U}_\mathsf{M}, \mathcal{V}_\mathsf{N}, \mathcal{W}_\mathsf{N}) = (\mathbb{V}(\mathsf{x}), \mathbb{I}_n, \mathbb{J}_n)$ when $(\mathsf{M}, \mathsf{N}) = (\mathbb{D}(\mathsf{x}), \mathbb{Z}_{n,\psi}^T)$.

Let us now consider the case where $(\mathsf{M}, \mathsf{N}) = (\mathbb{Z}_{m,\varphi}, \mathbb{D}(\mathsf{y}))$. Using Lemma 1.2 with $\ell = m$ and $\mathbb{Z}_{m,\varphi}^m = \varphi\mathbb{I}_m$,

$$\mathsf{A}\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m) = \sum_{j\le\alpha} \mathcal{K}_m(\mathbb{Z}_{m,\varphi}, \mathsf{g}_j)\mathbb{J}_m\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{h}_j)^T.$$

On the other hand, $P_\mathsf{M} = x^m - \varphi$, $P_\mathsf{N} = P_\mathsf{y}$, and it follows that the diagonal matrix multiplying A in the above identity is invertible.[3] Consequently, the product AB is given by

$$\mathsf{AB} = \sum_{j\le\alpha} \mathcal{K}_m(\mathbb{Z}_{m,\varphi}, \mathsf{g}_j)\mathbb{J}_m\mathcal{K}_m(\mathbb{D}(\mathsf{y}), \mathsf{h}_j)^T\mathbb{D}(\varphi\mathsf{e}_n - \mathsf{y}^m)^{-1}\mathsf{B}.$$

Applying Pol_m, and then (3.6) and Lemma 3.3, we obtain

$$\mathsf{Pol}_m(\mathsf{AB}) = P_\mathsf{N}^{-1}\sum_{j\le\alpha} \mathsf{Pol}_m(\mathsf{g}_j)\mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathbb{D}(\mathsf{h}_j)\mathsf{B}\big) \bmod P_\mathsf{M}.$$

Since $\mathbb{W}(\mathsf{y})^{-1}\mathbb{D}(\mathsf{h}_j) = \mathbb{V}(\mathsf{y})^{-1}\mathbb{D}(\mathsf{h}_j)\mathbb{D}(P_\mathsf{y}'(\mathsf{y}))$, using (3.8) shows that the right-hand side of the above identity is

$$P_\mathsf{N}^{-1}\mathsf{Pol}_m(\mathsf{G})\Big(\mathsf{Pol}_n\big(\mathbb{V}(\mathsf{y})^{-1}\mathsf{H}\big)^T\mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathsf{B}\big) \bmod P_\mathsf{N}\Big) \bmod P_\mathsf{M}.$$

Observing that $(\mathcal{U}_\mathsf{M}, \mathcal{V}_\mathsf{N}, \mathcal{W}_\mathsf{N}) = (\mathbb{I}_m, \mathbb{V}(\mathsf{y}), \mathbb{W}(\mathsf{y}))$ when $(\mathsf{M}, \mathsf{N}) = (\mathbb{Z}_{m,\varphi}, \mathbb{D}(\mathsf{y}))$ concludes the proof in this case.

[3] Otherwise, we would have $y_j^m = \varphi$ for some index j. In this case, y_j would be a root of both $x^m - \varphi$ and $P_\mathsf{y} = \prod_{1\le i\le n}(x - y_i)$, which contradicts the coprimeness assumption.

Proof of Theorem 3.1 for the Cauchy-like case

In this case $(\mathsf{M}, \mathsf{N}) = (\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y}))$, so that $P_\mathsf{M} = P_\mathsf{x}$ and $P_\mathsf{N} = P_\mathsf{y}$. Furthermore, the assumption $\gcd(P_\mathsf{M}, P_\mathsf{N}) = 1$ implies that $x_i \neq y_j$ for all (i, j) and a recovery formula for A is given by (1.6b):

$$\mathsf{A} = \sum_{j \le \alpha} \mathbb{D}(\mathsf{g}_j)\mathbb{C}(\mathsf{x}, \mathsf{y})\mathbb{D}(\mathsf{h}_j).$$

Now pre-multiply by $\mathbb{V}(\mathsf{x})^{-1}$, post-multiply by B, and then apply Pol_m and (3.8). It follows that

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{A}\mathsf{B}\big) = \sum_{j \le \alpha} \mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{g}_j\big)\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathbb{C}(\mathsf{x}, \mathsf{y})\mathbb{D}(\mathsf{h}_j)\mathsf{B}\big) \bmod P_\mathsf{M}.$$

One has $\mathbb{C}(\mathsf{x}, \mathsf{y}) = \mathbb{D}(P_\mathsf{y}(\mathsf{x}))^{-1}\mathbb{V}(\mathsf{x}, n)\mathbb{W}(\mathsf{y})^{-1}$, which is simply the rectangular version of Equation (3.6.5) in [Pan01, p. 90] and can be shown in the same way. Using (3.8) and (3.10) together with the invertibility of $P_\mathsf{y} = P_\mathsf{N}$ modulo P_M, it follows that for any $\mathsf{v} \in \mathbb{K}^n$,

$$\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathbb{C}(\mathsf{x}, \mathsf{y})\mathsf{v}\big) = P_\mathsf{y}^{-1}\,\mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathsf{v}\big) \bmod P_\mathsf{x}.$$

Thus, proceeding as for the second Vandermonde-like case, we deduce that

$$\mathsf{Pol}_m(\mathbb{V}(\mathsf{x})^{-1}\mathsf{A}\mathsf{B}) = P_\mathsf{N}^{-1}\mathsf{Pol}_m\big(\mathbb{V}(\mathsf{x})^{-1}\mathsf{G}\big)\Big(\mathsf{Pol}_n\big(\mathbb{V}(\mathsf{y})^{-1}\mathsf{H}\big)^T\mathsf{Pol}_n\big(\mathbb{W}(\mathsf{y})^{-1}\mathsf{B}\big) \bmod P_\mathsf{N}\Big) \bmod P_\mathsf{M}.$$

The conclusion follows from the fact that $(\mathcal{U}_\mathsf{M}, \mathcal{V}_\mathsf{N}, \mathcal{W}_\mathsf{N}) = (\mathbb{V}(\mathsf{x}), \mathbb{V}(\mathsf{y}), \mathbb{W}(\mathsf{y}))$ when $(\mathsf{M}, \mathsf{N}) = (\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y}))$.

3.3 Computing the row vector $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P)$

The previous section has shown that computing the product $\mathsf{A}\mathsf{B}$ relies on the following problem:

> Given three polynomial vectors $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V} \in \mathbb{K}[x]_n^{\alpha\times 1}$, and $\mathsf{W} \in \mathbb{K}[x]_n^{\beta\times 1}$, together with a polynomial P in $\mathbb{K}[x]$ of degree exactly n, compute the polynomial row vector $\mathsf{R} \in \mathbb{K}[x]_{m+n-1}^{1\times\beta}$ such that $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P)$.

In this section, we shall study how to compute the polynomial row vector R in the case where $\beta = \alpha$. Note first that with no loss of generality n can be assumed to be an integer power of two: defining $\bar{n} = 2^{\lceil \log n \rceil}$ and $\delta = \bar{n} - n$, one may check that $a \bmod b = x^{-\delta}\big((x^\delta a) \bmod (x^\delta b)\big)$ for any a and nonzero b in $\mathbb{K}[x]$; applying this identity componentwise to the definition of R gives

$$\mathsf{R} = x^{-\delta}\mathsf{U}^T\big(\mathsf{V}(x^\delta\mathsf{W})^T \bmod (x^\delta P)\big), \tag{3.14}$$

where now $x^\delta\mathsf{W}$ has degree less than $\bar{n}$ and $x^\delta P$ has degree $\bar{n}$.

We have also seen in the previous section that P is either $x^n - \psi$ or $\prod_{1\le i\le n}(x - y_i)$ for some given $y_1, \dots, y_n, \psi$ in $\mathbb{K}$. Let us study these two cases separately.

3.3.1 Case where $P = x^n - \psi$

Algorithm for $\psi = 0$

A method for computing $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod x^n)$ when $\mathsf{U}, \mathsf{V}, \mathsf{W}$ are three column vectors in $\mathbb{K}[x]_n^{\alpha\times 1}$ is proposed in [Bos10, page 210]. The idea is to split the polynomials within U, V, and W into their low and high parts, while doubling the number of columns for these three matrices. Thus, starting from column vectors, we eventually obtain (nearly) square matrices, where we can apply the fast polynomial matrix multiplication mentioned in Property 3.1. We present in Algorithm 3.1 a slight generalization of this approach, where we allow U to be in $\mathbb{K}[x]_m^{\alpha\times 1}$ for some integer m independent of n.

Algorithm 3.1: mpx (modulo of power of x)

```
Input: $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_d^{\alpha\times\beta}$.
Assumption: $\beta$ and $d$ are powers of two, $\alpha \le \beta d = n$, $\beta \le \underline{\alpha} := 2^{\lfloor \log \alpha \rfloor}$.
Output: $\mathsf{R} \in \mathbb{K}[x]_{m+d-1}^{\alpha\times 1}$ such that $\mathsf{R} = \mathsf{U}^T\big(\mathsf{V}\mathsf{W}^T \bmod x^d\big)$.
if $\beta = \underline{\alpha}$ then $\mathsf{R} \leftarrow \mathsf{U}^T\big(\mathsf{V}\mathsf{W}^T \bmod x^n\big)$
else
    $\mathsf{V}_0 \leftarrow \mathsf{V} \bmod x^{d/2}$;   $\mathsf{V}_1 \leftarrow \mathsf{V} \operatorname{div} x^{d/2}$;   $\mathsf{V}' \leftarrow [\mathsf{V}_0\ \mathsf{V}_1]$
    $\mathsf{W}_0 \leftarrow \mathsf{W} \bmod x^{d/2}$;   $\mathsf{W}_1 \leftarrow \mathsf{W} \operatorname{div} x^{d/2}$;   $\mathsf{W}' \leftarrow [\mathsf{W}_1\ \mathsf{W}_0]$
    $\mathsf{R}' \leftarrow \mathtt{mpx}(\mathsf{U}, \mathsf{V}', \mathsf{W}')$
    $\mathsf{R} \leftarrow \mathsf{U}^T\mathsf{V}_0\mathsf{W}_0^T + x^{d/2}\,\mathsf{R}'$
return $\mathsf{R}$
```

Assume for now that Algorithm mpx is correct and uses $O(f_{\mathbb{K}}(\alpha, n+m))$ operations in $\mathbb{K}$, where $f_{\mathbb{K}}$ is the cost function defined in (3.2). We will present in Section 3.3.2 a generalization of mpx for which we shall prove the correctness and show that it also runs using $O(f_{\mathbb{K}}(\alpha, n+m))$ field operations. Before that, let us see how mpx allows us to cover the case of $x^n - \psi$ for all $n \in \mathbb{N}_{>0}$ and $\psi \in \mathbb{K}$.

Algorithm for $\psi \neq 0$

Let us now consider $P = x^n - \psi$ with $\psi \neq 0$. We can reduce to the case of x^n thanks to the following lemma.

Lemma 3.4. *Let* $\widetilde{\mathsf{U}} = \mathsf{rev}_m(\mathsf{U})$, $\widetilde{\mathsf{V}} = x\,\mathsf{rev}_n(\mathsf{V})$, *and* $\widetilde{\mathsf{W}} = \mathsf{rev}_n(\mathsf{W})$. *Then* R *can be written*

$$\mathsf{R} = \mathsf{R}_1 + \psi\,\mathsf{rev}_{m+n-1}(\mathsf{R}_2),$$

where $\mathsf{R}_1 = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod x^n)$ *and* $\mathsf{R}_2 = \widetilde{\mathsf{U}}^T\big(\widetilde{\mathsf{V}}\widetilde{\mathsf{W}}^T \bmod x^n\big)$.

Proof. Note first that $a \bmod (x^n - \psi) = a \bmod x^n + \psi(a \operatorname{div} x^n)$ for any $a \in \mathbb{K}[x]$ of degree less than $2n$. Applying this identity componentwise to the matrix $\mathsf{V}\mathsf{W}^T$ gives $\mathsf{R} = \mathsf{R}_1 + \psi\,\mathsf{R}_2'$, where $\mathsf{R}_1 = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod x^n)$ as wanted, and where $\mathsf{R}_2' = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \operatorname{div} x^n)$. The desired expression for $\widetilde{\mathsf{R}}_2$ then follows by (3.3c) and then (3.3b). □

Now, using the transformation described in (3.14) so as to deal with any value of n, and calling `mpx` twice for the computation of R_1 and R_2 as in Lemma 3.4, respectively, we obtain Algorithm 3.2.

```
Algorithm 3.2: ComputeRx
    Input: $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_n^{\alpha\times 1}$, $\psi \in \mathbb{K}$.
    Assumption: $\alpha \leq n$.
    Output: $\mathsf{R} \in \mathbb{K}[x]_{m+n-1}^{1\times\alpha}$ such that $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod (x^n - \psi))$.
    $\delta \leftarrow 2^{\lceil \log n \rceil} - n$
    $\mathsf{R}_1 \leftarrow x^{-\delta}\,\mathsf{mpx}(\mathsf{U}, \mathsf{V}, x^\delta\,\mathsf{W}, n+\delta)$
    $\widetilde{\mathsf{U}} \leftarrow \mathsf{rev}_m(\mathsf{U})$;    $\widetilde{\mathsf{V}} \leftarrow x\,\mathsf{rev}_n(\mathsf{V})$;    $\widetilde{\mathsf{W}} \leftarrow \mathsf{rev}_n(\mathsf{W})$
    $\mathsf{R}_2 \leftarrow x^{-\delta}\,\mathsf{mpx}(\widetilde{\mathsf{U}}, \widetilde{\mathsf{V}}, x^\delta\,\widetilde{\mathsf{W}}, n+\delta)$
    $\mathsf{R} \leftarrow \mathsf{R}_1 + \psi\,\mathsf{rev}_{m+n-1}(\mathsf{R}_2)$
    return R
```

Theorem 3.2. *Let $N = m+n$. Algorithm* `ComputeRx` *works correctly and uses $O(f_{\mathbb{K}}(\alpha, N))$ operations in $\mathbb{K}$.*

Proof. Correctness follows from Lemma 3.4. The two calls to the routine mpx cost $O(f_{\mathbb{K}}(\alpha, N))$ field operations, while the multiplications/additions involving the scalar ψ, the shifts by powers of x, and the four reversals imply an overhead in $O(\alpha N)$. The conclusion follows from the definition of the cost function $f_{\mathbb{K}}$. □

3.3.2 Case where $P = \prod_{1\leq i\leq n}(x - y_i)$

To deal with the case where $P = \prod_{1\leq i\leq n}(x-y_i)$, we propose to generalize Algorithm `mpx`. Notice that x^n actually corresponds to the special case where $y_i = 0$ for all i. While `mpx` relies on splittings of polynomials into their low and high parts, we use here splittings based on Euclidean division, as described in the following lemma:

Lemma 3.5. *For $\alpha, \beta, n \in \mathbb{N}_{>0}$, let $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_n^{\alpha\times\beta}$ and let $P \in \mathbb{K}[x]$ of degree n. Let $P_1, P_2 \in \mathbb{K}[x]$ be such that $P = P_1P_2$, and let $\mathsf{V}_0 = \mathsf{V} \bmod P_1$, $\mathsf{V}_1 = \mathsf{V} \operatorname{div} P_1$, $\mathsf{W}_0 = \mathsf{W} \bmod P_1$, $\mathsf{W}_1 = \mathsf{W} \operatorname{div} P_1$, and $\mathsf{W}_2 = \mathsf{W} \bmod P_2$. If the degrees n_1, n_2 of P_1, P_2 satisfy $n_1 \leq n_2$ then*

$$\mathsf{V}\mathsf{W}^T \bmod P = \mathsf{V}_0\mathsf{W}_0^T + P_1\big([\mathsf{V}_0\ \mathsf{V}_1][\mathsf{W}_1\ \mathsf{W}_2]^T \bmod P_2\big)$$

and the matrices $[\mathsf{V}_0\ \mathsf{V}_1]$ and $[\mathsf{W}_1\ \mathsf{W}_2]$ are in $\mathbb{K}[x]_{n_2}^{\alpha\times 2\beta}$.

Proof. By definition, $\mathsf{V} = \mathsf{V}_0 + P_1\mathsf{V}_1$, so that $\mathsf{V}\mathsf{W}^T = \mathsf{V}_0\mathsf{W}^T + P_1\mathsf{V}_1\mathsf{W}^T$. On the other hand, depending on whether we use P_1 or P_2, we can rewrite W in two different ways, leading to

$$\mathsf{V}\mathsf{W}^T = \mathsf{V}_0(\mathsf{W}_0 + P_1\mathsf{W}_1)^T + P_1\mathsf{V}_1(\mathsf{W}_2 + P_2\mathsf{X})^T$$

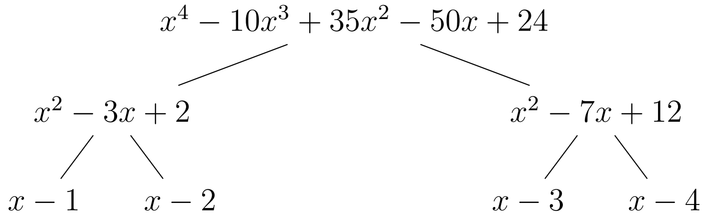
with X denoting $\mathsf{W} \operatorname{div} P_2$. Since $P = P_1P_2$, we deduce that

$$\mathsf{V}\mathsf{W}^T \bmod P = (\mathsf{V}_0\mathsf{W}_0^T \bmod P) + P_1((\mathsf{V}_0\mathsf{W}_1^T + \mathsf{V}_1\mathsf{W}_2^T) \bmod P_2).$$

The entries of V_0 and W_0 have degree less than n_1, those of W_2 have degree less than n_2, and those of V_1 and W_1 have degree less than $n - n_1 = n_2$. From $n_1 \leq n_2$ it then follows that the entries of $[\mathsf{V}_0\ \mathsf{V}_1]$ and $[\mathsf{W}_1\ \mathsf{W}_2]$ have degree less than n_2. This implies also that the entries of $\mathsf{V}_0\mathsf{W}_0^T$ have degree less than the degree n of P, so that $\mathsf{V}_0\mathsf{W}_0^T \bmod P$ equals $\mathsf{V}_0\mathsf{W}_0^T$. □

Note that, when n is a even, applying Lemma 3.5 with $P = x^n$ and $n_1 = n_2 = n/2$ yields the recursive formula used for R at line 6 in Algorithm `mpx`. Now, if we apply Lemma 3.5 with $P = P_{\mathsf{y}} = \prod_{i=1}^{n}(x - y_i)$ and $n_1 = n_2 = n/2$, we see that we need to know the coefficients of polynomials $P_1 = \prod_{i=1}^{n/2}(x - y_i)$ and $P_2 = \prod_{i=n/2+1}^{n}(x - y_i)$ in order to be able to carry out the computations. For now, we will assume that n is an integer power of two, and consider a subproduct tree $\mathcal{T}_{\mathsf{y}}$ for $\mathsf{y} \in \mathbb{K}^n$. This subproduct tree, like the one Figure 3.1, will contain all the polynomials needed recursively, and can be computed using $O(\mathsf{M}(n)\log n)$ operations in $\mathbb{K}$ [vzGG03, Algorithm 10.3]. Supposing that this tree $\mathcal{T}_{\mathsf{y}}$ has already been computed, we can retrieve P_{y} (the label at the root of $\mathcal{T}_{\mathsf{y}}$), and P_1 and P_2 (the label at the left and right children of the root, respectively) in constant time. Then, we can use P_1 and P_2 to perform the Euclidean divisions within the definitions of V_0, V_1, W_0, W_1, and W_2 in $O(\mathsf{M}(n))$ [vzGG03, Theorem 9.6]. Now, we can introduce Algorithm 3.3.

Figure 3.1: Example of subproduct tree $\mathcal{T}_{\mathsf{y}}$, where $\mathsf{y} \in \mathbb{R}^4$ is such that $y_i = i$.



Theorem 3.3. *Let $n = \beta d$ and $N = m + n$. Algorithm* `mpy` *works correctly and uses $O(f_{\mathbb{K}}(\alpha, N))$ operations in $\mathbb{K}$, where $f_{\mathbb{K}}$ is the cost function defined in* (3.2).

Proof. Let $k \in \mathbb{N}$ be such that $\beta = \underline{\alpha}/2^k$ and let us prove correctness by induction on k. (Here only β and d depend on $k \in \{0, 1, \ldots, \lfloor \log \alpha \rfloor\}$, while α, m, and the product $n = \beta d$ are fixed.) If $k = 0$ then $\beta = \underline{\alpha}$ and the algorithm returns the correct result. If $k \geq 1$ then the computed matrices V' and W' are in $\mathbb{K}[x]_{d/2}^{\alpha \times 2\beta}$ and $\mathcal{T}'$ is the subproduct tree associated to the vector $[y_{d/2+1}, \ldots, y_d]^T$ of length $d/2$. Since $2\beta = \underline{\alpha}/2^{k-1}$ and $d/2 = n/\underline{\alpha} \cdot 2^{k-1}$, the induction hypothesis gives $\mathsf{R}' = \mathsf{U}^T(\mathsf{V}'\mathsf{W}'^T \bmod P_2)$. Thus the matrix R returned by the algorithm when $k \geq 1$ is $\mathsf{U}^T(\mathsf{V}_0\mathsf{W}_0^T + (\mathsf{V}'\mathsf{W}'^T \bmod P_2)P_1)$ which, since $P_1P_2 = P_{\mathsf{y}}$ and $n_1 = n_2 = d/2$, is by Lemma 3.5 equal to the desired result $\mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P_{\mathsf{y}})$.

Let $C(k)$ denote the cost of the algorithm for a given value of k. If $k = 0$ then V and W are in $\mathbb{K}[x]_d^{\alpha \times \underline{\alpha}}$. Since $\underline{\alpha}$ is at most α, we may augment V and W with $\alpha - \underline{\alpha}$ zero columns, perform the multiplication, and then read $\mathsf{V}\mathsf{W}^T$, all this using $\mathsf{MM}(\alpha, d)$ field operations. Then, given $\mathsf{V}\mathsf{W}^T$ in $\mathbb{K}[x]_{2d}^{\alpha \times \alpha}$ together with P_{y} in $\mathbb{K}[x]$ of degree d, the cost of

Algorithm 3.3: mpy (modulo polynomial P_{y})

```
Input: $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_d^{\alpha\times\beta}$, subproduct tree $\mathcal{T}_{\mathsf{y}}$ for some $\mathsf{y} \in \mathbb{K}^d$.
Assumption: $\beta$ and $d$ are powers of two, $\alpha \le \beta d = n$, $\beta \le \underline{\alpha} := 2^{\lfloor \log \alpha \rceil}$.
Output: $\mathsf{R} \in \mathbb{K}[x]_{m+d-1}^{1\times\alpha}$ such that $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P_{\mathsf{y}})$.
if $\beta = \underline{\alpha}$ then
    $P_{\mathsf{y}} \leftarrow$ the polynomial at the root of $\mathcal{T}_{\mathsf{y}}$
    $\mathsf{R}' \leftarrow \mathsf{V}\mathsf{W}^T \bmod P_{\mathsf{y}}$
    $\mathsf{R} \leftarrow \mathsf{U}^T\mathsf{R}'$
else
    $P_1 \leftarrow$ the polynomial at the root of the left subtree of $\mathcal{T}_{\mathsf{y}}$
    $\mathcal{T}' \leftarrow$ the right subtree of $\mathcal{T}_{\mathsf{y}}$
    $P_2 \leftarrow$ the polynomial at the root of $\mathcal{T}'$
    $\mathsf{V}_0 \leftarrow \mathsf{V} \bmod P_1$;    $\mathsf{V}_1 \leftarrow \mathsf{V} \operatorname{div} P_1$
    $\mathsf{W}_0 \leftarrow \mathsf{W} \bmod P_1$;    $\mathsf{W}_1 \leftarrow \mathsf{W} \operatorname{div} P_1$;    $\mathsf{W}_2 \leftarrow \mathsf{W} \bmod P_2$
    $\mathsf{V}' \leftarrow [\mathsf{V}_0\ \mathsf{V}_1]$;    $\mathsf{W}' \leftarrow [\mathsf{W}_1\ \mathsf{W}_2]$
    $\mathsf{R}' \leftarrow \mathsf{mpy}(\mathsf{U}, \mathsf{V}', \mathsf{W}', \mathcal{T}')$
    $\mathsf{R} \leftarrow \mathsf{U}^T\mathsf{V}_0\mathsf{W}_0^T + P_1\mathsf{R}'$
return $\mathsf{R}$
```

computing R' is in $O(\alpha^2\mathsf{M}(d))$. Finally, given U in $\mathbb{K}[x]_m^{\alpha\times 1}$ and R' in $\mathbb{K}[x]_d^{\alpha\times\alpha}$, we set up R in two steps as follows: rewriting U^T in the form $\mathsf{U}^T = [1, x^c, x^{2c}, \dots, x^{(\alpha-1)c}]\mathsf{U}'^T$ with $c = \lceil m/\alpha \rceil$ and $\mathsf{U}' \in \mathbb{K}[x]_c^{\alpha\times\alpha}$, we compute first $\mathsf{Q} = \mathsf{U}'^T\mathsf{R}'$ using $\mathsf{MM}(\alpha, \max\{c, d\})$ field operations, and then deduce R from Q using $O(\alpha(m + d))$ field operations. In summary,

$$C(0) = 2\mathsf{MM}(\alpha, d) + \mathsf{MM}(\alpha, \max\{c, d\}) + O(\alpha^2\mathsf{M}(d) + \alpha m).$$

Now, since $k = 0$, we have $d = n/\underline{\alpha} < 2n/\alpha \le 2N/\alpha$. On the other hand, $c < m/\alpha + 1$ and $\alpha \le n$, so that $c < N/\alpha$. Using these bounds on c and d together with the properties of the cost functions M and MM allows us to conclude for the case $k = 0$.

Let us now bound $C(k)$ when $k \ge 1$. Given $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_d^{\alpha\times\beta}$ and P_1, P_2 in $\mathbb{K}[x]$ of degree $d/2$, we can compute the V_i's and the W_i's using $c_1\alpha\beta\mathsf{M}(d)$ field operations for some constant c_1 independent of k. Then R' is computed recursively using $C(k-1)$ field operations, and it remains to bound the cost of producing the result as $\mathsf{R} = \mathsf{Q} + \mathsf{Q}'$ with $\mathsf{Q} = (\mathsf{U}^T\mathsf{V}_0)\mathsf{W}_0^T$ and $\mathsf{Q}' = P_1\mathsf{R}'$.

For now let us write $D(k)$ for the cost of Q. Given R' in $\mathbb{K}[x]_{m+d/2-1}^{1\times\alpha}$ and P_1 in $\mathbb{K}[x]$ of degree $d/2$, we get Q' using $\alpha\mathsf{M}(m + d)$ field operations and, since both Q and Q' are in $\mathbb{K}[x]_{m+d-1}^{1\times\alpha}$, we can add them together using $\alpha(m+d)$ field operations. Since $d = n/\beta \le n$ and $N = m + n$, we deduce that $C(k) \le (c_1 + 2)\alpha\mathsf{M}(N) + C(k-1) + D(k)$ for $k \ge 1$. Since $k \le \log\alpha$, this implies

$$C(k) \le (c_1 + 2)\alpha\log\alpha\,\mathsf{M}(N) + C(0) + D(1) + \cdots + D(k).$$

In order to bound $D(k)$ let us rewrite U^T as $\mathsf{U}^T = [1, x^c, x^{2c}, \dots, x^{(\beta-1)c}]\mathsf{U}'^T$ with $c = \lceil m/\beta \rceil$ and $\mathsf{U}' \in \mathbb{K}[x]_c^{\alpha\times\beta}$ The product $\mathsf{U}'^T\mathsf{V}_0\mathsf{W}_0^T$ involves three polynomial matrices of respective dimensions $\beta \times \alpha$, $\alpha \times \beta$, $\beta \times \alpha$, and whose entries have degree less than $\max\{c, d/2\} \le \lceil N/\beta \rceil =: \delta$. Following [BJS08, Lemma 7] one may check that such a product has cost bounded by $c_2\alpha\beta^{\omega-1}\mathsf{M}(\delta)$ for some constant c_2 independent of k; if the field $\mathbb{K}$ has characteristic zero or is a finite field of cardinality at least 2δ, this bound becomes $c_2'\alpha\beta^{\omega-1}\delta + c_2''\alpha\beta\mathsf{M}(\delta)$ with c_2' and c_2'' two constants independent of k. Using the fact that $\delta < 2N/\beta$ and bounding by αN the number of additions used when multiplying by $[1, x^c, x^{2c}, \ldots, x^{(\beta-1)c}]$, we obtain

$$D(k) \le c_1'\alpha\beta^{\omega-2}\mathsf{M}(N),$$

or, if $\mathbb{K}$ has characteristic zero or is a finite field of cardinality at least $2N$,

$$D(k) \le c_1''\alpha\beta^{\omega-2}N + c_1'''\alpha\mathsf{M}(N);$$

here, c_1', c_1'', c_1''' denote some constants independent of k. Since $\sum_{1\le j\le k}(\underline{\alpha}/2^j)^{\omega-2}$ is in $O(\alpha^{\omega-2})$ for $\omega > 2$ and $\underline{\alpha} \le \alpha$, we deduce that $D(1)+\cdots+D(k)$ is either in $O(\alpha^{\omega-1}\mathsf{M}(N))$ or in $O(\alpha^{\omega-1}N + \alpha\log\alpha\mathsf{M}(N))$, depending on $\mathbb{K}$. (Note that $\omega = 2$ would imply the replacement of $\alpha^{\omega-1}$ by $\alpha\log\alpha$.) Hence $C(k)$ is bounded as wanted, and the conclusion follows. □

As in Section 3.3.1, let us conclude this section with an algorithm, Algorithm 3.4, that allows us to compute R given $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_n^{\alpha\times 1}$, and $\mathsf{y} \in \mathbb{K}^n$, and without assuming that n is an integer power of two.

Algorithm 3.4: `ComputeRy`

```
Input: $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V}, \mathsf{W} \in \mathbb{K}[x]_n^{\alpha\times 1}$, $\mathsf{y} \in \mathbb{K}^n$.
Assumption: $\alpha \le n$.
Output: $\mathsf{R} \in \mathbb{K}[x]_{m+n-1}^{1\times\alpha}$ such that $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P_{\mathsf{y}})$.
$\delta \leftarrow 2^{\lceil \log n \rceil} - n$
$\overline{\mathsf{W}} \leftarrow x^\delta\,\mathsf{W}$;    $\bar{\mathsf{y}} \leftarrow \mathsf{y}$ augmented with $\delta$ zeros
$\mathcal{T}_{\bar{\mathsf{y}}} \leftarrow$ the subproduct tree associated to vector $\bar{\mathsf{y}}$
$\overline{\mathsf{R}} \leftarrow \mathsf{mpy}(\mathsf{U}, \mathsf{V}, \overline{\mathsf{W}}, \mathcal{T}_{\bar{\mathsf{y}}})$
$\mathsf{R} \leftarrow x^{-\delta}\,\overline{\mathsf{R}}$
return $\mathsf{R}$
```

Theorem 3.4. *Let $N = m+n$. Algorithm* `ComputeRy` *works correctly and uses $O(f_{\mathbb{K}}(\alpha, N) + \mathsf{M}(N)\log N)$ operations in $\mathbb{K}$.*

Proof. Correctness follows from the identity in (3.14) and the claim of correctness in Theorem 3.3. For the cost, it follows from $n + \delta < 2N$ that the subproduct tree can be deduced from $\bar{\mathsf{y}}$ using $O(\mathsf{M}(N)\log N)$ operations in $\mathbb{K}$. The conclusion follows from bounding the overhead due to scaling by powers of x by $O(\alpha N)$, from the cost bound given in Theorem 3.3, and from the definition of the cost function $f_{\mathbb{K}}$. □

3.4 Fast multiplication by a matrix and application to inversion

Let $\mathsf{A} \in \mathbb{K}^{m\times n}$ be a matrix structured according to operator $\nabla[\mathsf{M},\mathsf{N}]$ with M,N as in (3.4), and let $\mathsf{B} \in \mathbb{K}^{n\times\beta}$. Suppose $\gcd(P_{\mathsf{M}}, P_{\mathsf{N}}) = 1$. We have seen in Section 3.2 how to express the multiplication AB as a function of $\mathsf{R} = \mathsf{U}^T(\mathsf{VW}^T \bmod P_{\mathsf{N}})$ for some $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V} \in \mathbb{K}[x]_n^{\alpha\times 1}$, $\mathsf{W} \in \mathbb{K}[x]_n^{\beta\times 1}$, and P_{N} defined from a length-α generator for A. Next, we have seen in Section 3.3 efficient algorithms for the computation of R when the number of columns of B is equal to the displacement rank of A, that is, when $\beta = \alpha$. What remains is to deduce the overall cost for the computation of AB. Section 3.4.1 is devoted to this task, starting with the case $\beta = \alpha$ and then dealing with any value of β. Then, we review the costs for structured matrix inversion proposed in Section 2.3 and present some experimental results.

3.4.1 Fast multiplication by a matrix

Let us first consider the case where the number of columns of B is equal to the displacement rank of A. Using Theorem 3.1 and the algorithms `ComputeRx` and `ComputeRy` from Section 3.3, we obtain the algorithm to compute AB described in Figure 3.2.

Figure 3.2: General approach for the multiplication of an $m \times n$ structured matrix of displacement rank α by an $n \times \alpha$ matrix.

Given a length-α generator (G,H) for $\mathsf{A} \in \mathbb{K}^{m\times n}$, and $\mathsf{B} \in \mathbb{K}^{n\times\alpha}$,

1. Compute U, V, W, P_{M}, and P_{N} as defined in Theorem 3.1;
2. Using `ComputeRx` when $P_{\mathsf{N}} = x^n - \psi$ or `ComputeRy` when $P_{\mathsf{N}} = \prod_{i=1}^{n}(x - y_i)$, compute $\mathsf{R} = \mathsf{U}^T(\mathsf{VW}^T \bmod P_{\mathsf{N}})$;
3. Deduce $\mathsf{AB} = \mathcal{U}_{\mathsf{M}}\, \mathsf{Pol}_m^{-1}\Big(P_{\mathsf{N}}^{-1}\mathsf{R} \bmod P_{\mathsf{M}}\Big)$.

Theorem 3.5. *Let* M,N *be as in* (3.4) *and such that* $\gcd(P_{\mathsf{M}}, P_{\mathsf{N}}) = 1$, *and let* $\mathsf{A} \in \mathbb{K}^{m\times n}$ *be given by a* $\nabla[\mathsf{M},\mathsf{N}]$*-generator* (G,H) *of length* α. *Let also* $\mathsf{B} \in \mathbb{K}^{n\times\alpha}$. *One can compute the matrix product* AB *from* G, H, *and* B *using* $O\tilde{~}(\alpha^{\omega-1}N)$ *operations in* $\mathbb{K}$, *where* $N = m + n$.

More precisely, depending on the structure of matrix A *and on the field* $\mathbb{K}$, *one can achieve a cost as presented in the following table.*

	arbitrary field	*field of characteristic zero, or finite field of cardinality at least 2N*
Toeplitz-like	$O(\alpha^{\omega-1}\,\mathsf{M}(N))$	$O(\alpha^{\omega-1}\,N + \alpha\log\alpha\,\mathsf{M}(N))$
Vandermonde-like or Cauchy-like	$O(\alpha^{\omega-1}\,\mathsf{M}(N) + \alpha\,\mathsf{M}(N)\log N)$	$O(\alpha^{\omega-1}\,N + \alpha\,\mathsf{M}(N)\log N)$

Proof. It suffices to prove the four costs announced in the table. For this, we have to analyze the cost for each of the three steps in Figure 3.2. First, let us detail the cost of Step 1:

- For the Toeplitz-like class, $P_{\mathsf{M}} = x^m - \varphi$ and $P_{\mathsf{N}} = x^n - \psi$ and the matrices $\mathcal{U}_{\mathsf{M}}$, $\mathcal{V}_{\mathsf{N}}$, and $\mathcal{W}_{\mathsf{N}}$ are either the identity or the reflexion matrix. Hence, no operation in $\mathbb{K}$ is needed for this case.

- For the Vandermonde-like and Cauchy-like classes, at least one of $\mathcal{U}_{\mathsf{M}}$ and $\mathcal{V}_{\mathsf{N}}$ is a Vandermonde matrix, so that we can get U and V using $O(\alpha\,\mathsf{M}(N)\log N)$ operations in $\mathbb{K}$.

  It remains to check that this bound also covers the cost of computing W. This is clear if N is (the transpose of) $\mathbb{Z}_{n,\psi}$, since then $\mathcal{W}_{\mathsf{N}}$ is either $\mathbb{I}_n$ or $\mathbb{J}_n$. If $\mathsf{N} = \mathbb{D}(\mathsf{y})$ then $\mathcal{W}_{\mathsf{N}}^{-1}\mathsf{B} = \mathbb{V}(\mathsf{y})^{-1}\mathbb{D}(P'_{\mathsf{y}}(\mathsf{y}))\mathsf{B}$. Given y and B, one can compute the coefficients of P_{y} using $O(\mathsf{M}(n)\log n)$ field operations, then deduce those of P'_y using $O(n)$ field operations, and produce the vector of values $P'_{\mathsf{y}}(\mathsf{y})$ using $O(\mathsf{M}(n)\log n)$ field operations. Scaling $\mathsf{B} \in \mathbb{K}^{n\times\alpha}$ into $\mathsf{B}' = \mathbb{D}(P'_{\mathsf{y}}(\mathsf{y}))\mathsf{B}$ has cost $O(\alpha n)$ and the product $\mathbb{V}(\mathsf{y})^{-1}\mathsf{B}'$ can be done using $O(\alpha\,\mathsf{M}(n)\log n)$ field operations. Hence, if $\mathsf{N} = \mathbb{D}(\mathsf{y})$ then W can be obtained using $O(\alpha\,\mathsf{M}(N)\log N)$ operations in $\mathbb{K}$.

  Thus, in either case, the cost of Step 1 is in $O(\alpha\,\mathsf{M}(N)\log N)$.

The cost for Step 2 in Figure 3.2 are given by Theorems 3.2 and 3.4. For the Toeplitz-like class, the matrix N is always a unit circulant matrix, so we can get R using $O(f_{\mathbb{K}}(\alpha, n))$ operations in $\mathbb{K}$. As for the Vandermonde-like and Cauchy-like classes, we can always bound the cost to compute R by the cost of Algorithm `ComputeRy`, which is in $O(f_{\mathbb{K}}(\alpha, n) + N\log N)$.

Now, let us look at the cost of Step 3 in Figure 3.2:

- For the Toeplitz-like class, $P_{\mathsf{M}} = x^m - \varphi$ and $P_{\mathsf{N}} = x^n - \psi$. The cost of getting $b := P_{\mathsf{N}}^{-1} \bmod P_{\mathsf{M}} \in \mathbb{K}[x]_m$ is in $O(N)$ according to [BJMS11]. In addition, $c := \mathsf{R} \bmod P_{\mathsf{M}} \in \mathbb{K}[x]_m^{1\times\alpha}$ can be computed using $O(\alpha N)$ field operations. Then the product $bc \bmod P_{\mathsf{M}}$ is obtained using $O(\alpha\,\mathsf{M}(m))$ field operations and, since in this case $\mathcal{U}_{\mathsf{M}}$ is either the identity or the reflexion matrix, we conclude that the total cost for Step 3 is in $O(\alpha\,\mathsf{M}(N))$.

- For the Vandermonde-like and Cauchy-like classes, we get P_{M} and P_{N} in $O(\mathsf{M}(N)\log N)$ using subproduct trees. Then $a := P_{\mathsf{N}} \bmod P_{\mathsf{M}}$ and $c = \mathsf{R} \bmod P_{\mathsf{M}}$ can be deduced using $O(\mathsf{M}(N))$ and $O(\alpha\,\mathsf{M}(N))$ field operations, respectively (see [vzGG03, Exercise 9.16(i)]). On the other hand, the cost for the inverse $b = a^{-1} \bmod P_{\mathsf{M}}$

is in $O(\mathsf{M}(m)\log m)$ while the cost for the product $bc \bmod P_{\mathsf{M}}$ is in $O(\alpha\,\mathsf{M}(m))$ (see [vzGG03, Corollary 11.8]). Finally, applying $\mathcal{U}_{\mathsf{M}}$ costs in this case $O(\alpha\,\mathsf{M}(m)\log m)$, from which we conclude that, for Vandermonde-like and Cauchy-like classes, the total cost of Step 3 is in $O(\alpha\,\mathsf{M}(N)\log N)$.

We can summarize the costs of the three steps for the different classes as follow:

	Toeplitz-like class	Vandermonde-like and Cauchy-like classes
Step 1		$O(\alpha\,\mathsf{M}(N)\log N)$
Step 2	$O(f_{\mathbb{K}}(\alpha, N))$	$O(f_{\mathbb{K}}(\alpha\,N) + \mathsf{M}(N)\log N)$
Step 3	$O(\alpha\,\mathsf{M}(N))$	$O(\alpha\,\mathsf{M}(N)\log N)$

Using the definition of the cost function $f_{\mathbb{K}}(\alpha, N)$ in (3.2a) or (3.2b) depending on $\mathbb{K}$, adding the costs within each column of the table above, and recalling that $\alpha \leq n \leq N$ allows us to conclude. □

Remarks

Let us comment on the costs mentioned in Theorem 3.5. When α is a small constant, and for all field $\mathbb{K}$, the cost for the Toeplitz-like class when $m = n$ becomes $O(\mathsf{M}(n))$, which corresponds to the product "matrix Toeplitz × vector", as expected. The same remark holds for the Vandermonde-like and Cauchy-like classes where the cost becomes $O(\mathsf{M}(n)\log n)$.

Now, consider the unstructured case where $\alpha = n$, and assume that $\mathbb{K}$ is either a field of characteristic zero or a finite field of cardinality at least $2n$. When $\omega > 2$, we have $n\log n \in o(n^{\omega-1})$ and we obtain that, for all the structures, the product of an $n \times n$ (un)structured matrix by an $n \times n$ matrix is achieved in $O(n^{\omega})$. Thus, this extends the approach for Toeplitz-like matrices in [Bos10, page 210] to the case of the Vandermonde-like and Cauchy-like classes.

Extension to β columns

Let us denote by $\mathsf{SMM}(\alpha, N)$ an upper bound on the cost to multiply a structured matrix $\mathsf{A} \in \mathbb{K}^{m\times n}$, given by a generator of length α, by a matrix $\mathsf{B} \in \mathbb{K}^{n\times\alpha}$.

So far we have considered products AB for which the matrix B has exactly α columns. Let us now give a cost analysis in the general case where B has any number β of columns.

Theorem 3.6. *Let $\mathsf{A} \in \mathbb{K}^{m\times n}$ be a structured matrix according to $\nabla[\mathsf{M},\mathsf{N}]$ with M,N as in* (3.4) *and given by a length-α generator* (G,H)*. Suppose that* $\gcd(P_{\mathsf{M}}, P_{\mathsf{N}}) = 1$*, and let $\mathsf{B} \in \mathbb{K}^{n\times\beta}$. Let also $\underline{\alpha} = \min\{\alpha,\beta\}$ and $\bar{\alpha} = \max\{\alpha,\beta\}$. Then the product AB can be deduced from G, H, and B using $\bar{\alpha}N + 2\bar{\alpha}/\underline{\alpha}\cdot\mathsf{SMM}(\underline{\alpha}, N)$ field operations.*

Proof. Assume first that $\alpha < \beta$. Defining $k = \lceil\beta/\alpha\rceil$ and $\beta' = k\alpha$, let us augment B with zero columns into the $n \times \beta'$ matrix $\mathsf{B}' = [\mathsf{B}|0]$. This matrix can be partitioned into k blocks $\mathsf{B}'_1, \ldots, \mathsf{B}'_k$, each having exactly α columns, so that the product AB can be read off the k products $\mathsf{AB}'_1, \ldots, \mathsf{AB}'_k$. The cost of getting all these products is bounded by $k\,\mathsf{SMM}(\alpha, N)$, and the conclusion follows from the fact that $k < \beta/\alpha + 1 < 2\beta/\alpha$.

Assume now that $\alpha \geq \beta$. Defining $k = \lceil \alpha/\beta \rceil$ and $\alpha' = k\beta$, we augment the matrices G and H into, respectively, the $m \times \alpha'$ matrix $\mathsf{G}' = [\mathsf{G}|0]$ and the $n \times \alpha'$ matrix $\mathsf{H}' = [\mathsf{H}|0]$. Furthermore, we partition G' and H' as $\mathsf{G}' = [\mathsf{G}'_1|\cdots|\mathsf{G}'_k]$ and $\mathsf{H}' = [\mathsf{H}'_1|\cdots|\mathsf{H}'_k]$, each block G'_j or H'_j having β columns. Hence $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}) = \sum_{j\leq k} \mathsf{G}'_j(\mathsf{H}'_j)^T$. On the other hand, the invertibility of $\nabla[\mathsf{M},\mathsf{N}]$ implies that for each j, there is a unique matrix A_j in $\mathbb{K}^{m\times n}$ such that $\nabla[\mathsf{M},\mathsf{N}](\mathsf{A}_j) = \mathsf{G}'_j(\mathsf{H}'_j)^T$. Since $\nabla[\mathsf{M},\mathsf{N}]$ is linear and invertible, we deduce that $\mathsf{A} = \sum_{j\leq k} \mathsf{A}_j$. To obtain AB from $\mathsf{G},\mathsf{H},\mathsf{B}$ in this case it thus suffices to set up the blocks G'_j and H'_j, to compute the k products $\mathsf{A}_j\mathsf{B}$, and to add those products together. The multiplication step has cost $k\,\mathsf{SMM}(\beta, N)$ and the addition step has cost $(k-1)m\beta$. Noting that $k < \alpha/\beta + 1 \leq 2\alpha/\beta$ allows us to conclude. □

3.4.2 Application to structured matrix inversion

New cost for structured matrix inversion

Recall from Section 2.3.1 the cost functions MM_* for $* \in \{\mathsf{C},\mathsf{V},\mathsf{H}\}$ that we have introduced for the cost analyses of algorithms `GenInvLT` and `GenInvHT`. The quantity $\mathsf{MM}_*(\alpha, n)$ ($* \in \{\mathsf{C},\mathsf{V},\mathsf{H}\}$) denotes the cost of the multiplication of an $n \times n$ structured (Cauchy-, Vandermonde-, and Hankel-like, respectively) matrix, given by a length-α generator, by an $n \times \beta$ matrix. In Section 2.4.1, we have assumed a cost in $O\tilde{~}(\alpha^2\, n)$ for these three functions. Yet, Theorem 3.5 above gives us new achievable costs for $\mathsf{MM}_*(\alpha, n)$ that are in $O\tilde{~}(\alpha^{\omega-1}\, n)$. Using these new costs for $\mathsf{MM}_*(\alpha, n)$, and remarking that we can still assume the superlinearity of $\mathsf{MM}_*(\cdot, n)$ in this case, we can deduce new costs for the inversion of Cauchy-like, Vandermonde-like, and Hankel-like matrices from Theorems 2.4, 2.5, and 2.7, respectively.

Corollary 3.1. *Let n be a power of two and let A be strongly regular and structured as in* (1.6a) *(Cauchy-like),* (1.7a) *(Vandermonde-like), or* (1.8a) *(Hankel-like). Then we can compute the inverse of A using either Algorithm* `GenInvLT` *(for Cauchy- and Vandermonde-like structures) or Algorithm* `GenInvLT` *(for Hankel-like structure) using at most $O\tilde{~}(\alpha^{\omega-1} n)$ field operations.*

More precisely, depending on the structure of the matrix A and on the field $\mathbb{K}$, one can achieve a cost as presented in the following table.

	arbitrary field	*field of characteristic zero, or finite field of cardinality at least* $4n$
Hankel-like	$O(\alpha^{\omega-1}\,\mathsf{M}(n)\log n)$	$O(\alpha^{\omega-1}\, n\log n + \alpha\log\alpha\,\mathsf{M}(n)\log n)$
Vandermonde- or Cauchy-like	$O(\alpha^{\omega-1}\,\mathsf{M}(n)\log n + \alpha\,\mathsf{M}(n)\log^2 n)$	$O(\alpha^{\omega-1}\, n\log n + \alpha\,\mathsf{M}(n)\log^2 n)$

Inversion of structured matrices in $O\tilde{~}(\alpha^{\omega-1}\, n)$ was already obtained for the Toeplitz-like structure (operator $\Delta[\mathbb{Z}_{n,0}, \mathbb{Z}_{n,0}^T]$) in [BJS07, BJS08]. In addition, we can find in [BJS07, BJS08] reductions from the Vandermonde-like and Cauchy-like structures to the Toeplitz-like one. Using these reductions and the cost proposed in [Bos10, page

210] for the multiplication of Toeplitz-like matrices yields a cost of $O(\alpha^{\omega-1}\,\mathsf{M}(n)\log n)$ or $O(\alpha^{\omega-1}\,n\log n + \alpha\log\alpha\,\mathsf{M}(n)\log n)$ for inversion, depending on the field $\mathbb{K}$. Here, we propose a cost for the inversion of Vandermonde-like and Cauchy-like matrices that is slightly worse. Yet, our approach has the advantage that it provides direct (that is, without reduction to another structure) inversion algorithms, which can be interesting for practical reasons. First, our approach requires fewer efforts to be implemented when one only needs to deal with a structure other than Hankel- or Toeplitz-like. Indeed, support for this particular structure suffices with our approach, while support for two structures and one reduction among them is needed with the approach in [BJS07, BJS08]. Second, as soon as α is large enough, namely when $\alpha \geq (\log n)^{1/(\omega-2)}$, the cost of our approach for Vandermonde-like and Cauchy-like is dominated by $O(\alpha^{\omega-1}\mathsf{M}(n)\log n)$ which is the asymptotic cost for the Hankel-like structure. Finally, the reductions in [BJS07, BJS08] have the same asymptotic cost that Toeplitz-like matrix inversion, so that the overhead they imply may impact the overall cost in practice.

Experimental results

Let us conclude this section with some experiments. As we did for Chapter 2, we have implemented the algorithms introduced in this chapter within the library SLA, along with some extra support for polynomial matrix multiplication. This has yielded about 1000 additional lines of code in SLA.

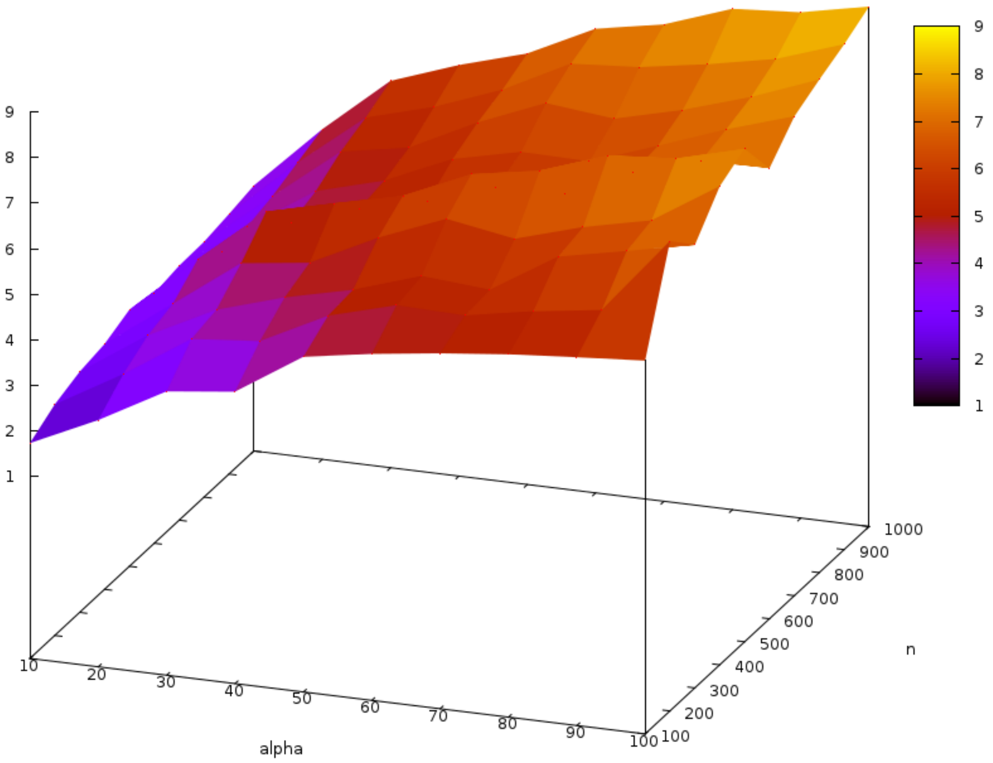
Our goal is to illustrate the impact of using the fast multiplication of a structured matrix by a matrix introduced in Figure 3.2 on our algorithms for structured matrix inversion. For this purpose, we compare, for the Cauchy-like structure, Algorithm `GenInvLT` with a naive multiplication “Cauchy-like matrix × vectors” in $O(\alpha^2\,\mathsf{M}(n)\log n)$, as in Section 2.4.1, to Algorithm `GenInvLT` with the fast multiplication “Cauchy-like matrix × vectors” in $O\tilde{~}(\alpha^{\omega-1}\,n)$, as described in this chapter. Note that, since we use a matrix multiplication based on Strassen's algorithm, and without any assumption on the field $\mathbb{K}$, we actually have here a cost in $O(\alpha^{1.81}\,\mathsf{M}(n) + \alpha\,\mathsf{M}(n)\log n)$ according to Theorem 3.5.

As in Section 2.4.1, computations are carried out on a desktop machine with an Intel® Core™ 2 Duo processor at 2.66 GHz. We work with the finite field $\mathbb{F}_p$ with $p = 999999937$, and the generator matrices G and H are picked randomly. In addition, we fix the values of x and y defining the displacement operator $\nabla[\mathbb{D}(\mathsf{x}), \mathbb{D}(\mathsf{y})]$ so that $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ has cardinality $2n$. Thus, we can use the fastest version of `GenInvLT`, that is, the one relying on “Cardinal's trick”.

Figure 3.3 illustrates the speed-up that we obtain when using the fast multiplication of a Cauchy-like matrix by a matrix, instead of the naive multiplication “Cauchy-like matrix × vectors”. The displacement rank α ranges from 10 to 100, while the dimension n ranges from 100 to 1000. As one can see on the figure, the version with fast multiplication is up to 8.5 times faster than the one using naive multiplication.

Moreover, the speed-up around 3 for $\alpha < 32$ is quite noticeable. Indeed, such values of α do not allow to use Strassen's algorithm for polynomial matrix multiplication. Yet, a significant speed-up is already achieved, so that using matrices of polynomials in order to perform a multiplication of the form “Cauchy-like matrix × vectors” seems intrinsically better than computing successively all the products “Cauchy-like matrix × one vector”.

Figure 3.3: Speed-up obtained by replacing the naive "Cauchy-like matrix × vectors" multiplication with the fast multiplication of a Cauchy-like matrix by a matrix from Figure 3.2.



Now, let us fix α and look at the behavior of the speed-up with respect to the dimension n. As mentioned earlier, both the naive and the fast routines for "Cauchy-like matrix × vectors" multiplication are in $O(\mathsf{M}(n)\log n)$ (α is a fixed constant). Therefore, they both lead to a cost in $O(\mathsf{M}(n)\log^2 n)$ for inversion. Yet, we can observe a slow increase of the speed-up with respect to n. For instance, when $\alpha = 100$, the speed-up goes from ≈ 5.2 for $n = 100$ to ≈ 8.5 for $n = 1000$. The fact that the speed-up actually decreases just after a power of two (see $n = 300$ and $n = 600$) lets us imagine that the overall increase might be due to threshold effects within the library NTL.

Finally, if we fix n and look at the behavior of the speed-up with respect to the dimension α, we observe a clean increase. For instance, when $n = 1000$, the speed-up varies from ≈ 2.8 when $\alpha = 10$ to ≈ 8.5 when $\alpha = 100$. This can be partially explained since the fast version uses Strassen's algorithm, which leads to a cost in $O(\alpha^{1.81})$ instead of $O(\alpha^2)$ (n being constant here). In addition, by looking at the behavior in the range $10 \leq \alpha \leq 32$, where Strassen's algorithm is not used, we also think that part of the speed-up's increase is due to another factor.

To conclude, note that this is a first experiment, and that the speed-up we obtain here is not completely understood. Nevertheless, using fast multiplication of a structured matrix by a matrix appears to be very promising in practice.

Conclusions and perspectives for Part I

Conclusions

Structured matrices of size $m \times n$ and displacement rank α are matrices that can be represented by a generator having only $\alpha(m+n)$ coefficients instead of mn. Many operations like transposition, multiplication by a vector, addition or multiplication of structured matrices, and inversion, can be performed using generators, which yields a sizeable gain when α is small. In particular, when α is a small constant, we obtain quasi-linear costs with respect to n (assuming $m \in O(n)$), whereas the corresponding costs for dense, unstructured matrices are in $O(n^2)$ or in $O(n^\omega)$. In this first part of the document, we have focused on the most common structures that are Cauchy-like, Vandermonde-, Toeplitz- and Hankel-like structures, and we have worked on two problems related to these structures: inversion and multiplication by a matrix.

Superfast algorithms for inverting a structured matrix are known since the apparition of the MBA algorithm in 1980 [Mor80, BA80]. A major issue with this approach is to control the size of the generators for intermediate quantities so as to ensure a final cost in $O\tilde{}(\alpha^2 n)$ for inversion. This control is usually achieved with so-called compression steps. A noticeable variant of MBA due to Cardinal [Car99, Car00] proposes to cope with this difficulty, in the case of Cauchy-like matrices, by computing specified generators for the intermediate quantities and the inverse, that have already an appropriate size. Following this idea, we have designed a general, compression-free algorithm relying on a new, recursive formula for the specified generator of the inverse. We have studied in detail the cost of this new approach for three structures: Cauchy-, Vandermonde- and Hankel-like matrices. Each cost was expressed as a function of $\mathsf{MM}_*(n, \alpha)$ with $* \in \{\mathsf{C}, \mathsf{V}, \mathsf{H}\}$, which is the cost for the multiplication of an $n \times n$ structured (Cauchy-, Vandermonde-, and Hankel-like, respectively) matrix A by α vectors, where α is the length of the generator provided for A. Comparing with several variants of MBA, it has appeared that removing the two compression steps within MBA, while keeping the asymptotic cost unchanged, leads to a significant decrease in size for the intermediate generators, and so to smaller products of the form "structured matrix × vectors". This yields theoretical speed-ups up to a factor of 7, which were indeed observed in practice in our experiments.

As a consequence of our work on structured matrix inversion, we have obtained that this operation heavily relies on a single basic block, which is the multiplication "structured matrix × vectors," where the displacement rank of the structured matrix and the number of vectors are of the same order. Recent works in [BJS07, BJS08] and [Bos10, page 210] show that, in the case of Toeplitz-like matrices (with Stein's displacement operator), such a multiplication can be achieved in $O\tilde{}(\alpha^{\omega-1} n)$ field operations using fast multiplication of polynomial matrices. This thesis extends this idea to all the aforementioned structures for Sylvester's displacement operator. First, we have introduced a polynomial expression of the matrix product AB, where A is an $m \times n$ structured matrix of displacement rank α and $\mathsf{B} \in \mathbb{K}[x]^{n\times\beta}$. This has pointed out the problem of computing the polynomial row vector $\mathsf{R} = \mathsf{U}^T(\mathsf{V}\mathsf{W}^T \bmod P)$ given $\mathsf{U} \in \mathbb{K}[x]_m^{\alpha\times 1}$, $\mathsf{V} \in \mathbb{K}[x]_n^{\alpha\times 1}$, and $\mathsf{W} \in \mathbb{K}[x]_n^{\beta\times 1}$, and where P is either $x^n - \psi$ or $\prod_{i=1}^n (x - y_i)$. Second, we have proposed algorithms to compute R in the case where $\alpha = \beta$, that are in fact generalizations of the approach in [Bos10, page 210] for the case where $P = x^n$ and $m = n$. Finally, combining the two previous points has led to a general algorithm for the multiplication "structured matrix × vectors" in $O\tilde{~}(\alpha^{\omega-1}(m+n))$ when the number of vectors is α. In addition, for each structure, the cost obtained matches with the one of the underlying special matrix (Cauchy for Cauchy-like structure, and so on) when α is a small constant. These costs also match with the cost $O(n^\omega)$ for dense, unstructured matrix multiplication when $m = n = \alpha$, assuming that the field $\mathbb{K}$ is either of characteristic zero or a finite field with $4n$. Finally, using the new costs for the multiplications "structured matrix × vectors," we have deduced new costs in $O\tilde{~}(\alpha^{\omega-1} n)$ for our inversion algorithms in Chapter 2. Compare to the asymptotic costs obtained with [BJS08] and [Bos10, page 210], our approach for inversion is slightly worse for Cauchy- and Vandermonde-like. Yet, it yields direct inversion algorithms for these two structures, contrary to the approach in [BJS08] which reduces them to the Toeplitz-like case. Furthermore, a first experiment with Cauchy-like matrices has shown that using our fast algorithm for the multiplications of the form "Cauchy-like matrix × vectors" within our inversion algorithm leads to significant speed-ups in practice.

Perspectives

As we have seen, our direct approach for the inversion of Cauchy- and Vandermonde-like matrices does not achieve the best known asymptotic costs to solve these two problems. One question is then to see whether it can be improved so as to compete with the best known algorithms that use a reduction to Toeplitz-like matrix inversion. If we look closely at the overall inversion algorithm, we can see that some intermediate quantities are computed several times within the routine for fast "structured matrix × vectors" multiplication. By identifying them and factorizing their computation, we may be able to solve this issue.

The implementation of the various algorithms presented in this first part of the thesis have also raised several questions. First, it would be interesting to estimate the quantity of extra-memory needed by our algorithms, and to see whether in-place versions can be designed for some of them. Second, adding some support for a given structure implies to code several routines like the multiplication by a vector. Such routines may be implemented in several ways, depending on the formula in use. To choose a formula involving a minimum number of memory operations like copying or reverting of vectors is a very difficult and tedious task. Therefore, it would be interesting to see if we can model the formulas using the language proposed in [FMMP09] and thus use the SPIRAL project [PMJ$^+$05, PFV11] in order to tune the code for these routines. Finally, we have mentioned that inversion of a given structured matrix can be achieved either by a direct approach, or by reducing the problem to the inversion of a matrix having another structure. We think that the various alternatives implied by the different reductions deserve investigation. However, this may take quite some time if done by hand. Again, it would be interesting to see how general frameworks for automatic code optimization such as SPIRAL can help us to speed up the exploration process.

Part II

Analyzing the implementations of arithmetic expressions

Chapter 4

On the evaluation of arithmetic expressions

This chapter serves as an introduction to our work on the evaluation of arithmetic expressions. First, we give an overview of some of the common issues arising when one is to implement a given arithmetic expression. Then, we focus on the initial context that has led us to this work, that is, the generation of polynomial evaluation codes for a library providing software support of floating-point numbers on integer processors. We also explain how this context has motivated us to design a general framework for analyzing the implementations of arithmetic expressions. Finally, we summarize our contributions to this domain.

4.1 Issues underlying the evaluation of arithmetic expressions

Arithmetic expressions are expressions made of sums and products of variables. While the order of the terms and the implicit parenthesization do not matter when we look at the mathematical object, they become an issue when one wants a good implementation of it on some architecture. Indeed, nowadays' architectures usually provide only binary operations, so that choices have to be made in order to go from the mathematical object to one concrete implementation. This raises several questions in various fields of computer science.

4.1.1 Issues in algebraic complexity

In this field, the main concern is to estimate the number of arithmetic operations needed to evaluate a given expression. This could mean finding the exact minimum number of operations, obtaining a relevant lower bound of this number and/or designing an algorithm with as few operations as possible.

Answers to these questions are quite sensitive to the computation model considered (see the discussion in [BCS97, §1]). The model of straight-line programs (SLP) [BCS97, §4.1] has been successfully used to derive some interesting results. In this model, an evaluation consists in a sequence of binary operations whose operands are taken among the input variables and the already computed quantities.

Among the main results of this field, one can cite the works on the minimum number of multiplications in order to evaluate a^n (also known as the problem of shortest addition chains for n) which are reviewed in [Knu98, §4.6.3]. Similarly, people have worked on the minimum number of multiplications in order to evaluate a degree-n, univariate polynomial. Pan showed in [Pan66] that, as thought since long, this minimum number is n, which is achieved by Horner's rule. Paterson and Stockmeyer went further and distinguished between scalar and non-scalar multiplications. They gave in [PS73] an algorithm to evaluate $p(x)$ (where the coefficients are considered as scalars, and x as non-scalar) with only $O(\sqrt{n})$ non-scalar multiplications. When non-scalar multiplications are more costly then scalar ones, like when we evaluate a polynomial $p(x)$ with scalar coefficients at a matrix point, this approach becomes very interesting. Moreover, [PS73] also showed that roughly $\sqrt{n}$ non-scalar multiplications are necessary to evaluate $p(x)$ upon several computation models based on SLPs.

Finally, we can cite the technique of preconditioning, that is, a "free" pre-computation that aims to turn the initial evaluation problem into a simpler one. This may be relevant when the underlying evaluation has to be performed with lots of different values for the inputs, so that the cost for the pre-computation does not matter much. Indeed, it has been intensively studied for that case of polynomial evaluation: [Knu62, Eve64] showed that this evaluation for polynomials in $\mathbb{C}[X]$ can be performed in n additions and only $\lceil \frac{n+3}{2} \rceil$ multiplications; [PS73] introduced another approach more suitable for polynomials in $\mathbb{R}[X]$, with $n + O(\log n)$ multiplications; and [Pan78] discusses optimality issues for polynomials over the reals and the complex numbers with respect to the degree.

4.1.2 Issues in combinatorics

In the SLP model without division, it has been proved that there exists a one-to-one correspondence between implementations for a^n and the set of binary rooted trees with exactly n leaves. The number of such trees, and so the number of implementations for a^n, is the nth number of the Wedderburn-Etherington sequence [Wed22, Eth37]. While we do not know any closed-form formula for this sequence, it is easy to compute the values for small values of n through a recurrence formula. Moreover, [Ott48] gives the asymptotic behavior of this sequence when n tends to infinity, showing that the number of implementations for a^n grows roughly exponentially with respect to n.

This kind of correspondence can be established for other simple arithmetic expressions, as soon as the set of implementations can be described within a theory called the *species theory* [FS09]. When it is the case, this theory provides useful tools to compute the number of objects of size n efficiently [Piv08, PSS08], and to describe the asymptotic behavior when n grows to infinity [FS09, §6]. In our context, this helps to get precise information about the number of implementations for simple arithmetic expressions.

4.1.3 Issues in compilation and code generation

We have already mentioned the question of minimizing the number of operations, which leads to efficient evaluations on a purely sequential architecture. However, architectures offering some kind of parallelism have been available since decades [Kuc77]. In our context, we may want to exploit instruction level parallelism (ILP). This type of parallelism can be either MIMD (multiple instruction multiple data) parallelism, where we assume to have p processors available that can perform any kind of instruction, or SIMD (single instruction multiple data) where we impose moreover that each processor performs the same instruction at each cycle.

In the MIMD model, the latency for evaluating an arithmetic expression (possibly with divisions) involving n variables (each appearing only once) has been well studied. Assuming that binary $+$ and $\times$ have a unit cost, it is remarked in [KM74] that this latency is at least $\lceil \log_2 n \rceil$, and that one can use associativity and commutativity in order to perform the evaluation in at most $\lceil \log_2 n \rceil + 2d + 1$ cycles, where d is the maximum level of nested parentheses. Moreover, the authors show that this evaluation can be carried out using at most $\lceil \frac{n-2d}{2} \rceil$ processors. This result is extended in [Bre74], where the author proves that, using also distributivity, and regardless of the depth of parenthesization, one can achieve a latency of $4 \log_2 n + \frac{10(n-1)}{p}$ cycles, where p is the number of processors in use. We can also cite [BKM73], which deals with expressions without divisions, and [KM75] were sharper bounds are provided in several cases.

While these results are effective (that is, we can deduce from the proofs algorithms to achieve the announced bound for a given expression), they have several limitations: first, even though the upper bounds are close to the lower bound $\lceil \log_2 n \rceil$, one can always hope to find a better solution for a special case; second, SIMD parallelism is not directly handled and, while one can simulate an MIMD architecture with a SIMD one and apply the techniques for the MIMD case, this multiplies the latency by a constant as noted in [Kuc77], and again, one may hope for a faster evaluation; third, the main criterion on pipelined architectures is throughput, and not latency; fourth, the usage of operators like the fused multiply-add (FMA), which maps (a, b, c) to $a \cdot b + c$ and is now a standard operator in floating-point arithmetic [IEE08], is not addressed.

Another approach in order to find a fast way to evaluate an expression on a given architecture is to write a program that will look at many possibilities and generate the code for the fastest evaluation encountered. This approach, which becomes more usable as computer power increases, is embraced for instance in [Gre02], and in [HKST99], where the authors describe a heuristic search for fast evaluation of univariate polynomials on the Itanium® processor using only the FMA operator.

4.1.4 Issues in numerical analysis

When one wants to perform a computation involving real or complex numbers, this computation is usually performed with some finite precision arithmetic instead. In this case, evaluations are subject to some rounding errors, and it becomes an issue to control these errors, or at least to have a good insight into their impact. Indeed, the way we perform some evaluation can greatly affect the numerical quality of the result. See for instance [Hig02, §4.2] which discusses the impact of ordering when performing a sum of n terms in floating-point arithmetic.

As soon as one uses fixed-point arithmetic (see [Yat09] for a good introduction on this topic) or floating-point arithmetic as defined in the IEEE-754 standard [IEE08], the behavior of operators + and × is well specified and models like the ones in [Hig02, §2.2] can be used to bound the errors occurring during the evaluation of an arithmetic expression. It thus becomes possible to deduce *a priori* error bounds, that is, bounds on the evaluation error that are functions of the precision and the actual values of the input variables. These *a priori* bounds may be pessimistic in practice for the evaluation of a univariate polynomial [Rev06], and one may prefer to compute a tighter error bound alongside with the actual computation (see the concept of running error in [Hig02, §3.3]). Another approach to obtain a tight error bound is embraced by the tool Gappa [Mel06]. This tools aims at certifying a given error bound[1] on the evaluation of an arithmetic expression by means of rewriting rules and bisection of the intervals where the variables range.

Another issue lies in improving the numerical quality. The software tool Fluctuat [PGM04] has been developed in order to analyze the sources of numerical errors and to indicate the places in `C` code that cause much of them. Moreover, Martel has designed some techniques described in [Mar07, Mar09a, Mar09b] in order to turn a code for an arithmetic expression into a mathematically equivalent but more accurate one. Other techniques can be used to improve numerical quality. For instance, the rounding error due to a sum of two floating-point numbers is also a floating-point number and it can be computed exactly [Knu98, Theorem B, page 236]. Thus, this computed error can be injected again later, in order to reduce the final evaluation error. This idea has led to the compensation technique [Kah65, Bab69], which was extensively used by Ogita, Rump, and Oishi in order to compute sums of n variables accurately [ORO05, Rum09].

Finally, we can cite the recent work in [LMT10], where the authors analyze the different possible implementations for a sum of 10 terms in order to find a trade-off between the latency and rounding errors.

4.2 Context and motivation

In this section, we present the initial context that have led us to study the evaluation of arithmetic expressions. It lies in the development of a software tool named CGPE[2] (Code Generation for Polynomial Evaluation). This tool is used to generate part of the code within the library FLIP[3] (Floating-point Library for Integer Processors), whose goal is to provide software single-precision support for integer processors. We first give an overview of FLIP. Then, we focus more especially on CGPE. Finally, we conclude this section by motivating our choice to design a general approach for analyzing the evaluation of a given arithmetic expression.

4.2.1 Floating-point arithmetic support for integer processors

Embedded systems are nowadays ubiquitous. Because embedded processors need to be small, some of them still do not have a floating-point unit, so that they can only perform computations on (finite-size) integers. In order to run codes relying on floating-point

[1] It can also find out one if none was provided.

[2] See `http://cgpe.gforge.inria.fr/`, [Rev09, MR11], as well as Section 8.1.1.

[3] `http://flip.gforge.inria.fr/`

arithmetic on these so-called *integer processors*, one needs a library that emulates floating-point numbers using the integer type available in hardware. FLIP is one of such libraries. It can provide single-precision floating-point support for any 32-bit integer processor, and has been especially optimized for VLIW (Very Long Instruction Word) architectures as implemented by the ST231 processor from STMicroelectronics's ST200 family [FFY05].

Implementation of floating-point operators in FLIP

Efficient implementations of a floating-point operator usually rely on the evaluation of a polynomial approximating this operator on a small interval. This is the approach embraced by FLIP to provide a software support for single precision on integer processors. Apart from the basic operators (+,−,×, and the FMA), the generic path of the other operators (elementary functions such as division, square root, trigonometric functions, exponential, logarithms, . . .) is or will be implemented according to the following classical process (see [Mul06, §3] and [MBdD+10, §11]):

1. **Range reduction**: Evaluation of operator f at point x is replaced with the evaluation of $\tilde{f}(\tilde{x})$ by use of mathematical properties on f (symmetries, algebraic identities). This is done so that all the possible values for $\tilde{x}$ lie in a small interval I.

2. **Polynomial approximation**: One computes some polynomial approximation of $\tilde{f}$ on I. In practice, the software tool Sollya [Che09] is used in order to get an approximation polynomial $p(x)$ of minimum degree, given some bound on the approximation error.

3. **Polynomial evaluation**: One chooses an implementation for the evaluation of $p(\tilde{x})$. This implementation should take advantage of the features of the target architecture as much as possible. However, as already mentioned in the previous section, this choice may impact the numerical quality of the result.

4. **Final routine**: One looks at the value resulting from the evaluation of $p(\tilde{x})$ in order to deduce the correct rounding for $f(x)$, or a result accurate for all but the last bit.

Note that, since the processor only provides integers, the evaluation at step 3 will be carried out using fixed-point arithmetic [Yat09]. This means that we use integer instructions, but that we read the result as if there were a radix point within the encoding of integers at a position determined in advance.

Furthermore, note that we only give here a general view of the whole process used within FLIP. Steps 1 and 4 are usually specific to each operator. Moreover, one needs to decode the input, handle special cases, and compute the sign and exponent in addition to $p(\tilde{x})$ in order to encode the result after step 4. All this work should be done as much as possible in parallel of the four aforementioned steps. We refer to [JMM+10] for more details on this topic.

Example: implementation of $\sqrt{x}$

The example presented here comes from [JKMR11]. Suppose we have an integer processor and we want to provide to it support for the square root operator in single precision and with correct rounding, as specified by the IEEE 754-2008 standard of floating-point arithmetic. Excluding special inputs (like negative numbers, infinities, or subnormals), we have a floating-point number $x = m \cdot 2^e$, where $m \in [1, 2)$ has binary expansion $(1.m_1 \dots m_{23})$ and where $-126 \leq e \leq 127$, and we need to compute $\mathrm{RN}(\sqrt{x})$, that is, the single precision floating-point number the nearest to the real number $\sqrt{x}$. Range reduction can be achieved by remarking that

$$\sqrt{x} = \underbrace{\sqrt{m\,2^\delta}}_{\ell} \cdot 2^{\lfloor e/2 \rfloor} \qquad \text{with } \delta = e - 2 \cdot \lfloor e/2 \rfloor \in \{0, 1\}.$$

Now, we have $m\,2^\delta \in [1, 4)$ and we may use a polynomial approximation of $\sqrt{\cdot}$ on this interval to deduce a sufficiently precise approximation v of ℓ and then deduce the correct mantissa for $\mathrm{RN}(\sqrt{x})$, the correct exponent being $\lfloor e/2 \rfloor$.

In fact, one of the contributions of [JKMR11] is to provide a clever way to obtain such a v. Their idea is to evaluate the special bivariate polynomial

$$q(x, y) = 2^{-25} + y \cdot p(x),$$

where p is a polynomial approximant of $\sqrt{1 + \cdot}$ on $[0, 1)$, at point

$$(x,\, y) = (m - 1,\, 2^\delta).$$

If $\|p - \sqrt{1 + \cdot}\|_{\infty,[0,1)}$ and the error entailed by the computation of $q(m - 1, 2^\delta)$ are small enough (see [JKMR11, Lemma 1] for the complete result) then the value v obtained will be accurate enough to deduce $\mathrm{RN}(\sqrt{x})$. The introduction of such a bivariate polynomial is motivated by the fact that it allows for more parallelism. Indeed, we can distribute y inside the evaluation of $p(x)$ and thus reduce the latency.

As mentioned above, the question of finding a good polynomial approximant can be solved by using the software tool Sollya. Therefore, what remains is to find an accurate-enough way to evaluate our special bivariate polynomial $q(x, y)$ on $[0, 1) \times \{1, \sqrt{2}\}$. Several tools to check the accuracy of a given code exist, like Fluctuat [PGM04] or Gappa [Mel06]. Yet, one has first to generate a code before calling one of these tools.

4.2.2 Generating fast and accurate-enough code for polynomial evaluation

The choice of the way to implement the polynomial evaluation at step 3 of the aforementioned process is a major issue. As we have said, the implementation should be accurate enough, in the sense that the error entailed by the usage of fixed-point arithmetic is no greater than a given bound. Moreover, the polynomial evaluation is typically the largest part of the code for an operator, and thus, we really need to do it as fast as possible.

Classical schemes for polynomial evaluation

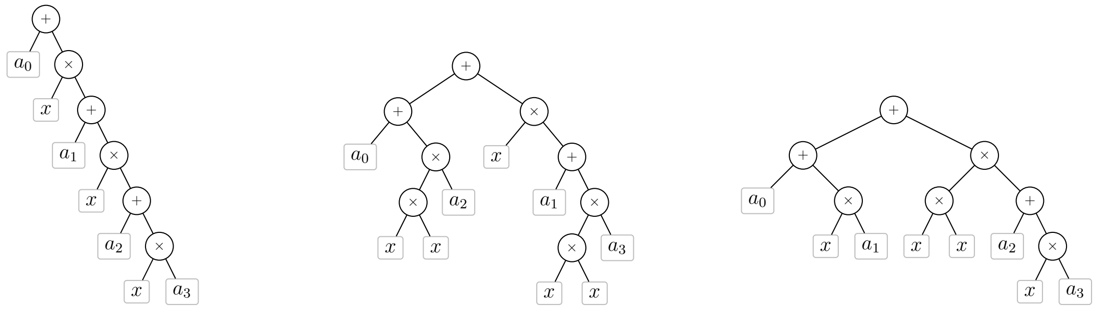
Several rules have been coined for the evaluation of a univariate polynomial [Knu98, §4.6.4]. Figure 4.1 presents three of them:

1. **Horner's rule.** It consists in n multiplications and n additions as shown in Figure 4.1(a). This is in fact one of the most commonly used schemes for evaluating polynomials in implementations of floating-point operators (see for instance [CLM+05, LV09] and [Lau08, §6.2]). Its interest lies in its good numerical stability, especially when x is not too close to a zero of $a(x)$ [Bol04, §9]. However, this sequential scheme does not expose any instruction-level parallelism (ILP) and thus gets inefficient as soon as parallelism is available.

2. **Second-order Horner's rule.** This second rule extends Horner's rule in order to expose some ILP. It consists in splitting up the polynomial into its odd and even parts, evaluating both parts using Horner's rule, and finally combining both intermediate results using a last Horner's iteration [Knu98, §4.6.4]. It requires exactly $n + 1$ multiplications, as shown in Figure 4.1(b). Remark that it uses at most two ways of the architecture, and gets inefficient as soon as more parallelism is available, like for the ST231 processor which has 4 ways.

3. **Estrin's rule.** This last rule is based on the divide-and-conquer paradigm, and consists in splitting up the polynomial into its low and high parts. Then, both parts are evaluated in a recursive way, until getting degree-1 polynomials, as shown in Figure 4.1(c). Its implementation tends to expose more ILP than the previous rules, but to the detriment of an increase of the number of multiplications, since it requires about $n + \log(n + 1) - 1$ multiplications.

Figure 4.1: Classical rules for degree-3 univariate polynomial evaluation.



(a) Horner's rule. (b) Second-order Horner's rule. (c) Estrin's rule.

Generation of polynomial evaluation codes for FLIP with CGPE

The classical schemes mentioned above can be adapted for evaluating bivariate polynomials, as shown for example in [CK04, PS00] for Horner's rule. Yet, none of them appears to be appropriate for our context:

- As already mentioned, Horner's rule fails to expose ILP. On the contrary, Estrin's rule requires more multiplications, and thus may be harder to schedule on an architecture with limited parallelism;
- The numerical properties of these rules for a given polynomial are, to the best of our knowledge, not completely understood, especially for the special bivariate case;
- Finally, recall from the example of the square root operator that we need to deduce the values of x and y from the encoding of the floating-point number at input. In practice, the value of x is known before the one of y, and this has to be taken into account for the latency. In this situation, we will speak about the *delay* of y, that is, the number of cycles between the availability of the value of x and the availability of the value of y. Classical rules for bivariate polynomials were designed with the assumption that the values of x and y are known at the same time (thus, assuming a delay of 0 cycle). Therefore, they may not be adapted when the delay is larger, like in the case of the square root of FLIP, where it is of 3 cycles.

For all these reasons, we needed a tool to explore the possible implementations of polynomial evaluation, and this is why a first version of CGPE has been introduced in [Rev09]. Given:

- the coefficients of a polynomial $p(x)$ or $q(x, y) = \alpha + y \cdot p(x)$,
- a small interval for each variable x and y,
- a maximal evaluation error bound ε,
- information about the target architecture (cost of operators + and ×, available parallelism),
- possibly a delay on variable y,

CGPE returns a `C` code for the evaluation of $p(x)$ or $q(x, y)$, which is as fast as possible on the target architecture, and whose evaluation error due to the use of fixed-point arithmetic is guaranteed (through calls to Gappa) to be no greater than the bound ε. CGPE was mainly used in order to generate code for the ST231 processor, which is a 4-issue VLIW integer processor. This processor can perform four additions in parallel, but is limited to two multiplications in parallel. Moreover, while the addition is done in 1 cycle, one has to wait 3 cycles in order to get the result of a multiplication. Yet, multiplication is pipelined, so that we can launch two of them per cycle.

Other context requiring polynomial evaluation: matrix function evaluation

The context of libm design presented in Section 4.2.1 is not the only one that leads to the need of finding a good way to evaluate a polynomial p. For instance, efficient code for the evaluation of $f(\mathsf{A})$ where f is an elementary function and A is an $n \times n$ matrix may also rely on polynomial approximation. We refer to [Hig08, §10–13] for examples of algorithms based on Padé approximants to evaluate $f(\mathsf{A})$ where $f \in \{\exp, \log, \sin, \cos\}$.

In this context, the matter lies in the number of matrix-matrix multiplications, rather than the latency like in FLIP. Indeed, these operations are more costly than addition of matrices and scalar-matrix multiplications, so their cost dominates the total evaluation time. Again, the rules illustrated in Figure 4.1 fail to provide a good evaluation for $p(\mathsf{A})$, and better algorithms like the one in [PS73, Algorithm B] has to be preferred, as noted in [Hig08, §4.2].

4.2.3 Motivation

As we have seen in the previous section, polynomial evaluation raises several questions:

1. How to generate implementations for polynomial evaluation? What rules should we consider to deduce implementations from the mathematical expression of the polynomial?
2. How many ways are there to evaluation a polynomial?
3. Given some measure (the latency, the number of matrix-matrix multiplications,...) on the implementations, what is the minimum value that one can achieve?
4. How many of these implementations do achieve this minimum value?
5. Can we generate efficiently these optimal implementations?

Note that these questions may arise in a context different from polynomial evaluation. Indeed, several of them have already been mentioned (and sometimes answered) in Section 4.1 for the particular case of powers a^n.

What we aim at is a uniform approach to answer these questions. Inspired by the generation algorithm from [Rev09, §6.1], we propose to adopt an inductive view on the arithmetic expressions. Using the concepts of *evaluation schemes* (the model we will use for implementations, see Section 5.1.2) and *decompositions* (see Section 5.1.3), we will describe a general framework for the analysis of arithmetic expressions in Section 5.2. This will serve as a basis for the development of specific algorithms for purposes like counting the number of implementations or optimizing according to a given *measure* (our model for optimization criteria, introduced in Section 5.4).

4.3 Contributions of this thesis

4.3.1 Algorithms introduced in the following chapters

The generic approach mentioned previously, and explained in details in Chapter 5, will be applied to tackle three categories of issues:

Generation issues

Recall that one of our goals is to generate code for the evaluation of an arithmetic expression. We propose in Section 5.3 a first algorithm, called `Generate`, whose purpose is to compute all the evaluation schemes of a given arithmetic expression. This algorithm is the first example to illustrate our approach, and is mainly for theoretical purpose.

However, allowing the user to provide some constraints that will be used in the generation process leads to an effective algorithm named `GenerateWithHint` (see Section 5.4.3). It computes, given a measure on the evaluation schemes, all the schemes for an expression whose measure is less than a given threshold. This algorithm is used:

1. at the end of Chapter 5, in order to study the number of squarings lying in the evaluation schemes for a^n with a minimum number of multiplications;

2. in Section 8.2, in order to generate evaluation schemes for polynomials which achieve a minimum number of non-scalar multiplications.

Counting issues

We propose three counting algorithms. The first one, Algorithm `Count` (see Section 6.1), allows us to compute the number of evaluation schemes for a given arithmetic expression. This algorithm is used in Section 6.2.1 to retrieve several known sequences (like the number of schemes for a^n when $n \in \mathbb{N}$). Moreover, by applying it in Section 6.2.2 to univariate polynomials $p(x)$ and then to special bivariate polynomials $q(x, y) = \alpha + y \cdot p(x)$ with $\deg(p) = n$, we deduce two new sequences, $\texttt{A169608}(n)$ and $\texttt{A173157}(n)$.

In the previous algorithm, we have associated only one value to the given arithmetic expression: its number of schemes. Instead of considering the set of schemes as a whole, we can consider a partition of it with respect to some measure. Algorithm `CountPerMes` relies on this idea to compute recursively the distribution of the evaluation schemes for a given arithmetic expression according to a measure. We apply this algorithm in Sections 6.4.2 and 6.4.3 to deduce the distribution of the evaluation schemes for a univariate polynomial according to first the number of multiplications, and second the latency. In particular, we obtain a Gaussian-like shape for the latency, which confirms the intuition that there are only few optimal and nearly optimal schemes according to this measure.

In addition, as we did for generation, we design in Section 6.4.4 a counting algorithm, extending the previous one by adding a threshold at input that is used in order to reduce the number of schemes considered during the computation. This yields Algorithm `CountWithHint` that we use:

1. to compute faster than with `CountPerMes` the number of schemes for univariate polynomials having an optimal or nearly optimal latency;

2. to compute the number of schemes achieving the minimum number of matrix-matrix multiplications for $p(\mathsf{A})$ (see Section 8.2.2).

Optimization issues

In Section 7.1.2, we introduce a first optimization algorithm named `Optimizer`. While this algorithm is fast, its correctness requires the measure we want to optimize to be such that optimizing the subexpressions can lead to the optimal value for the initial expression. This is satisfied by the latency, but not by the number of multiplications, for instance. We illustrate this algorithm with three examples:

1. the computation of the minimum latency for polynomial evaluation schemes (see Section 8.1.2);

2. the heuristic computation of the minimum number of multiplications needed to compute a^n (see Section 7.1.3);

3. the heuristic optimization of first the latency, and second the accuracy, for a polynomial (see Section 7.3.1).

For measures that do not satisfy the requirement for the correctness of `Optimizer`, we propose three alternatives: `OptimizerSet` (see Section 7.2.1), which is a variation of `Optimizer` that optimizes a criteria for a set of expressions rather than a single expression; `GlobalOptimizer` (see Section 7.2.2) that, like `CountPerMes`, computes recursively all the achievable measures, so that the minimum can be extracted at the end of the process; and `GlobalOptimizerWithHint` (see Section 7.2.2) that is a variation of `GlobalOptimizer` with additional constraints. These three approaches are compared for the minimization of the number of multiplications for a^n, showing that `OptimizerSet` is faster for small values of n, while `GlobalOptimizerWithHint` is faster for large values of n. Moreover:

1. `GlobalOptimizerWithHint` is used in Section 7.3.1 in the context of polynomial approximation in order to optimize first the latency, and then the accuracy;

2. `GlobalOptimizerWithHint` is used in Section 8.2.2 in order to determine the minimum number of matrix-matrix multiplications for the evaluation of $p(\mathsf{A})$;

3. `OptimizerSet` is used in Section 8.2.2 to determine the minimum number of matrix-matrix multiplications for the simultaneous evaluation of the odd and even parts of $p(\mathsf{A})$.

Finally, we introduce Algorithm `BiOptimizer` whose purpose is to compute, given two criteria, the best trade-offs that can be achieved. We present two applications for this last algorithm:

1. the search of a trade-off between a minimum latency and a maximal delay for the evaluation of special bivariate polynomials on the ST231 processor;

2. a case study of the trade-off between latency and accuracy for a polynomial.

Summary

A summary of the different algorithms that will be introduced in the next chapters is provided by Table 4.1. These algorithms, along with support for the arithmetic expressions and the measures we used for the different applications mentioned above, have been implemented in a `C++` library. The total size of the code is around 3000 lines.

All the results and timings presented in the following chapters have been obtained using this library. Except for examples involving numerical accuracy issues (Sections 7.3.1, 7.3.3 and 8.1), computations were carried out on a server with two Quad-Core AMD Opteron$^{\text{TM}}$ running at 2.4GHz, and 80Go of RAM. When dealing with numerical accuracy, we used a desktop machine with an Intel$^{\circledR}$ Core$^{\text{TM}}$ 2 Duo processor at 2.66 GHz

Table 4.1: Main algorithms introduced in the next chapters.

Algorithms			Type of analysis
Generation	Counting	Optimization	

`Generate`	`Count`	`Optimizer`
		`BiOptimizer`
		`OptimizerSet`
			abstraction of the set of evaluation schemes by one value

	`CountPerMes`	`GlobalOptimizer`	abstraction of the set of evaluation schemes by its image according to a given measure
`GenerateWithHint`	`CountWithHint`	`GlobalOptimizerWithHint`	abstraction of the set of evaluation schemes by its image according to a given measure and limitation using a given threshold

and 2Go of RAM. In fact, this machine provided newer versions of the libraries MPFR[4] and MPFI[5] that we used in order to compute error bounds.

Finally, notice that the examples we present in the following chapters take mainly place in the context of CGPE and the generation of `C` code for the ST231 processor. In particular, for the computation of the latency, we will always consider a cost of 1 and 3 cycles for + and ×, respectively.

4.3.2 Other contributions

In addition to the algorithms listed above, this second part of the thesis offers three other contributions.

Asymptotic study of several sequences

Section 6.3 is dedicated to the study of several sequences that appear in the experiments using Algorithm `Count`. More precisely:

1. following [FS09, Proposition VII.5] which gives the asymptotics for the number of schemes for a^n, we obtain the asymptotics for the number of schemes for $a \cdot b^n$;
2. using the asymptotics for the case of $a \cdot b^n$, we prove that the logarithm of the sequence $\texttt{A169608}(n)$ is in $\Omega(n^2)$ and in $O(n^3)$;

[4] `http://www.mpfr.org/`

[5] `http://perso.ens-lyon.fr/nathalie.revol/software.html`

3. we deduce the same result for the sequence $\log\big(\texttt{A173157}(n)\big)$ by comparing it to $\log\big(\texttt{A169608}(n)\big)$.

Improvement of the software tool CGPE

Thanks to the library we have developed, we were able to partially rewrite the software tool CGPE. First, The resulting new design of CGPE differs from the previous one in two points: First, we added a precomputation with `Optimizer` to determine the minimum latency for the evaluation of the polynomial at input, and how to achieve it; Second, inspired by `GenerateWithHint`, we strengthen the constraints in the step producing schemes by adding systematically numerical and schedule checkings within this step. We refer to Section 8.1 for the details. This rewriting of CGPE has two main consequences:

1. on average, a significant speed-up in the overall generation process is achieved. Indeed, we observe on average a relative gain of about 50% in the generation time;

2. the usage of the pre-computation with `Optimizer` increases the relevance of schemes computed in the first step of CGPE, while demanding fewer parameters to be provided by the user.

Optimality of the evaluation of $p(\mathsf{A})$ and $\exp(\mathsf{A})$ for small polynomial degrees

The experiments involving polynomial matrices, mentioned previously and detailed in Section 8.2, lead us to three interesting conclusions:

1. Paterson and Stockmeyer's algorithm [PS73, Algorithm B] is optimal in terms of the number of matrix-matrix multiplications for an evaluation of $p(\mathsf{A})$ only based on additions and multiplications (without preconditioning), at least for $\deg(p) \leq 15$;

2. the evaluation schemes for $p(\mathsf{A})$ achieving the minimum number of non-scalar multiplications can be classified into three categories (one of them being the variants of Paterson and Stockmeyer's algorithm);

3. Higham's approach for the simultaneous evaluation of the odd and even parts of a given polynomial p (problem arising for the evaluation of $\exp(\mathsf{A})$) is shown to be optimal, at least for $\deg(p) \leq 7$.

Chapter 5

How to model and analyze implementations of arithmetic expressions

In this chapter, we present a way to model and analyze the different ways of evaluating some arithmetic expression thanks to the concept of evaluation scheme. The first section deals with the formal definition of evaluation schemes, while the second section shows how one can manipulate them algorithmically in order to perform some analysis. Then, a third section illustrates how to generate the evaluation schemes exhaustively. Finally, we will discuss how to model an optimization criterion in a fourth section and propose an algorithm for generating all the evaluation schemes that satisfy an optimization constraint.

5.1 Modelling implementations with the concept of evaluation scheme

5.1.1 Evaluation of arithmetic expressions

An arithmetic expression is basically a mathematical object made of constants in some commutative ring R and one or several variables, that are linked together by additions[1] and multiplications. Evaluating this expression means computing its value in R when one gives some values for all the variables. Without loss of generality, we can replace every constant in our expression with symbolic variables (whose value will always be the same for all evaluations) and only consider arithmetic expressions without constants.

The evaluation can then be performed as follows: we start with the actual values for each variable, then we take two already known values and we add or multiply them in order to create a new value, finally we go on until we get the value corresponding to our arithmetic expression. This corresponds to the concept of straight-line program [BCS97, §4.1]:

[1] Subtraction can be considered as a special case of addition, as noticed in [Bre74].

Definition 5.1. *A straight-line program (SLP) is a finite sequence* $(a_{1-k}, \dots, a_{-1}, a_0, a_1, \dots, a_r)$ *such that:*

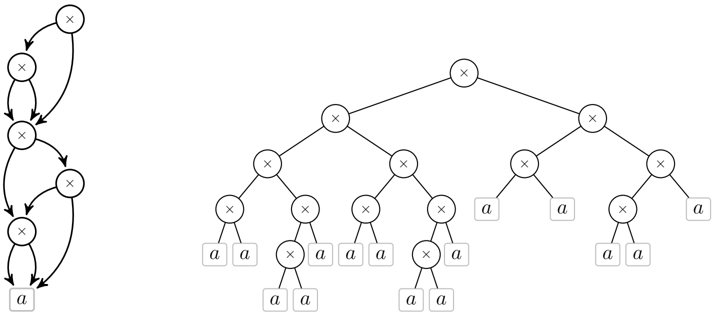
- *for all* $1 \leq j \leq k$, a_{1-j} *represents a value in* R *for the* j*th variable;*
- *for all* $1 \leq i \leq r$, $a_i = a_m \diamond a_n$ *with* $\diamond \in \{+, \times\}$ *and* $m, n < i$.

We will say that a given SLP evaluates an arithmetic expression if the final result a_r for this SLP is indeed the value corresponding to our expression, whatever the values we choose for the variables. For instance, Figure 5.1(a) shows an SLP for the evaluation of a^{15}. As noticed in [BCS97, page 105], such a sequence of binary operations can also be represented as a direct acyclic multigraph[2] (DAG) where all the nodes have an outer degree of 0 or 2. The DAG corresponding to the SLP in Figure 5.1(a) is given in Figure 5.1(b). Alternatively, one may consider ordered binary trees, like in Figure 5.1(c), instead of DAGs.

Figure 5.1: Example of SLP for evaluating a^{15} and its corresponding DAG and binary tree.

$$\begin{aligned}
a_0 &= a \\
a_1 &= a_0 \times a_0 \quad // \; a^2 \\
a_2 &= a_1 \times a_0 \quad // \; a^3 \\
a_3 &= a_1 \times a_2 \quad // \; a^5 \\
a_4 &= a_3 \times a_3 \quad // \; a^{10} \\
a_5 &= a_4 \times a_3 \quad // \; a^{15}
\end{aligned}$$

(a) SLP for evaluating a^{15}.



(b) Corresponding DAG.

(c) Corresponding ordered binary tree.

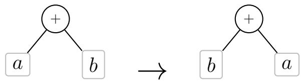
Given such a tree, we can get back to the corresponding DAG by merging the internal node corresponding to the same subexpression. It is also straightforward to retrieve the corresponding arithmetic expression. Going from an arithmetic expression to an ordered binary tree is not as direct since one may perform some choices like the order of the different terms or the way to perform a sum/product of more than two terms with binary operations only. These choices are usually dictated by conventions like left to right summation.

The different ways to evaluate a given arithmetic expression come from how we read this mathematical object. Moreover, one can replace the expression with a mathematically equivalent one by using some algebraic identity before performing the evaluation. While this transformation can be quite sophisticated, like when preconditioning a polynomial [Knu62, Eve64] in order to evaluate it with as few multiplications as possible, we will restrict ourselves, as in [Bre74] and [Kuc77], to the basic algebraic identities inferred

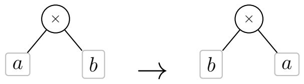
[2] When one binary operation uses the same operand twice, there will be two edges from the node labelled with this binary operation to the node corresponding to the operand. Hence, the appropriate structure is a multigraph. Nevertheless, this fact does not matter much in the sequel, so we will simply say graph instead of multigraph.

from the commutative ring structure, that is, commutativity, associativity, and distributivity. Let us list these rules exhaustively, and illustrate their effect on a given internal node within an ordered binary tree:

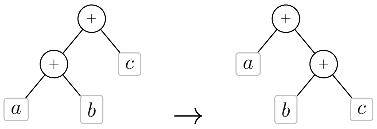
r_1 **[commutativity of +]**



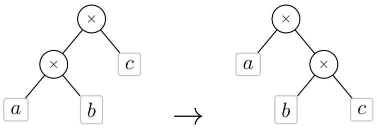
r_2 **[commutativity of ×]**



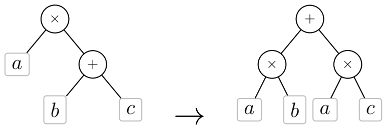
r_3 **[associativity of +]**



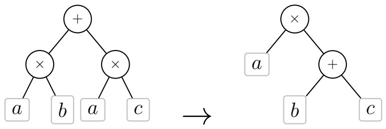
r_4 **[associativity of ×]**



r_5 **[distributivity of × over +]**



r_6 **[factorization]**



Notice that each rule is reversible, in the sense that we can go from the right hand side back to the left hand side using only the listed rules (for rules r_3 and r_4, use commutativity on both nodes, then associativity, and again commutativity on both nodes).

Previous works [KM74, Bre74, Kuc77] essentially used these rules in order to turn an ordered binary tree into another one with a smaller depth or with much more instruction level parallelism. This has yielded several bounds on the time needed to evaluate an arithmetic expression depending on the number of variables, the level of nested parentheses, and the number of processors available. Here, we intend to focus more on the set of all the trees that we could obtain by applying these rules successively. We define this set as follows:

Definition 5.2. *Let f be some arithmetic expression, and t be the ordered binary tree for evaluating f that is deduced from reading f using some implicit rules to cope with ambiguity. We will denote by $\mathcal{P}(f)$ the closure of $\{t\}$ under the rules r_1 to r_6, and we will call it the set of the* **parenthesizations** *of f hereafter.*

In order for this definition to be correct, we need to ensure that the set $\mathcal{P}(f)$ is the same regardless of the way we read f to deduce the first tree t. This holds since the variations in the reading of f can be handled using commutativity and associativity only. The motivation in adding distributivity (and so factorization) to the set of rules lies in the fact that this rule is the key to increase parallelism in some arithmetic expressions, like when moving from Horner's rule to Estrin's rule [Kuc77].

Now, one can think that this set of rules is quite restrictive. Indeed, adding new variables or changing the set of variables by some precomputation is not possible. Moreover, simple algebraic identities like $a^2 - b^2 = (a+b)\cdot(a-b)$ or $a - a = 0$ are not covered. Nevertheless, our choice is motivated by two reasons:

1. Given an arithmetic expression f, the diversity in the set of the possible parenthesizations $\mathcal{P}(f)$ already allows to derive interesting results.

2. We want the set $\mathcal{P}(f)$ to be finite, so that an exhaustive analysis of it makes sense. To handle an identity like $a - a = 0$, we need a rule in order to go from 0 to $a - a$. But this kind of rule can be used anywhere in order to make the size of an evaluation tree arbitrary large, and thus $\mathcal{P}(f)$ would be infinite. In our simple set of rules, only distributivity increases the size of a tree, but it also pushes down the multiplications so that we cannot use this rule endlessly.

Finally, we have not considered the case of other operators here. When working within a field $\mathbb{K}$, we may want to use division. We could have added a few rules for this operation, like one to go from $(a+b)/c$ to $a/c + b/c$ and *vice versa*. However, we have mainly worked on mathematical objects defined without divisions, like powers of a and polynomials. In this context, division may help (we can evaluate a^{15} as a^{16}/a for instance) but the relevant identity is $a/a = 1$, which would give us an infinite set for $\mathcal{P}(f)$. Another operator of interest is the fused multiply-add (FMA) which maps (a, b, c) to $a \cdot b + c$. Again, it is possible to add specific rules to handle it, but its ternary nature would make things more difficult, and we have not investigated this for now.

5.1.2 Going from evaluation trees to evaluation schemes

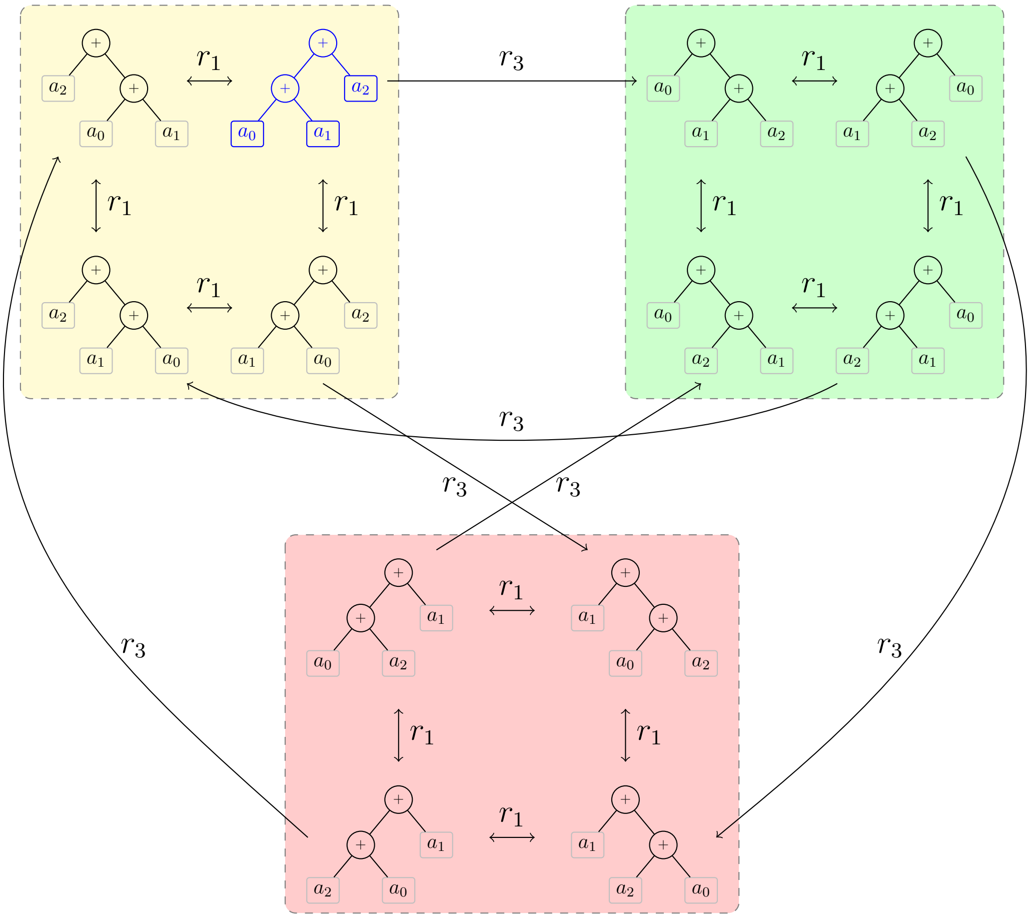
Typically, computations will be carried out using a standard fixed-point or floating-point arithmetic. In this context, the only rules that still hold among the rewriting rules mentioned above are the commutativity of $+$ and the commutativity of $\times$ [MBdD+10, §2.4]. Applying any other rule may lead to a different numerical result.

We are actually interested in the potentially numerically distinct evaluations of a given arithmetic expression f. For $u, v \in \mathcal{P}(f)$, let us note $u \equiv v$ when one can go from u to v only by applying commutativity of $+$ and $\times$. Then, what we want to consider is the set of equivalence classes for the relation $\equiv$ in $\mathcal{P}(f)$, which we will denote by $\mathcal{S}(f) = \mathcal{P}(f)/{\equiv}$.

The effect of commutativity on an ordered binary tree is to swap the two children of a given internal node. Thus, considering equality modulo $\equiv$ means that the order of the children in our trees does not matter. Therefore, one way to deduce $\mathcal{S}(f)$ from $\mathcal{P}(f)$ is to turn all the ordered binary trees in $\mathcal{P}(f)$ into unordered binary trees and to keep only one occurrence of each tree obtained. Notice that, in terms of SLPs, going from $\mathcal{P}(f)$ to $\mathcal{S}(f)$ means that we do not mind about the order of the operands for each operation in the sequence forming the SLP.

Figure 5.2 illustrates how to go from the arithmetic expression $a_0 + a_1 + a_2$ to all its evaluation schemes. If we read this expression with left-associativity for $+$ in mind, we will obtain a first ordered binary tree for the evaluation $a_0 + a_1 + a_2$, which is drawn in blue in the figure. Then, we apply commutativity of + (rule r_1) and associativity of + (rule r_3) as much as we can, so as to get a set of ordered binary trees closed by these rules[3], that is, the set $\mathcal{P}(a_0 + a_1 + a_2)$ of all parenthesizations for $a_0 + a_1 + a_2$. Then, we compute the equivalence classes in $\mathcal{P}(a_0 + a_1 + a_2)/\equiv \; = \; \mathcal{S}(a_0 + a_1 + a_2)$. We obtain the three classes represented in dashed boxes, that is, the three evaluation schemes for our expression $a_0 + a_1 + a_2$.

Figure 5.2: The set of all the evaluation schemes (represented as dashed boxes) for $a_0 + a_1 + a_2$.



Note that it was important to allow commutativity for + in a first step before reducing modulo $\equiv$. Indeed, we would never have gone from the initial ordered tree in blue to the dashed box in red at Figure 5.2 using only associativity. Therefore, we would have missed the evaluation scheme corresponding to this box.

5.1.3 Decompositions and subexpressions for an arithmetic expression

One advantage of representing evaluation schemes as binary trees lies in the fact that the inductive nature of trees helps to think in a divide-and-conquer way. Thus, a complex

[3] There is no multiplication, so that the other rules could never be applied.

scheme can be viewed as a binary operation on two simpler schemes, and so on until we arrive at a leaf corresponding to one of the variables within the initial arithmetic expression. This is summarized by the following characterization of evaluation schemes:

Definition 5.3. *Given an arithmetic expression f, an evaluation scheme s for f is:*

- *either a single leaf. In this case, f is a variable and s is therefore its unique evaluation scheme. In the following, we will call such a scheme a* **trivial scheme**, *and we will usually speak about the variable itself instead of its scheme;*
- *or an unordered binary tree with a root labelled with an operator $\diamond \in \{+, \times\}$ and two (potentially equal) sons s_1 and s_2. In this case, we will speak about a* **non-trivial scheme**.

Moreover, in the second case, we can completely define s by the pair $(\diamond,\ \{s_1,\ s_2\})$. In the sequel, such a pair will be referred to as the **decomposition** *of the scheme s.*

Notice that we used a set of schemes in this definition of decomposition. Indeed, evaluation schemes are unordered binary trees, and using a set helps to have no order between s_1 and s_2 since $\{s_1, s_2\} = \{s_2, s_1\}$.

Now, let us generalize this concept of decomposition to arithmetic expressions. A given evaluation scheme s being an unordered binary tree with operators at its internal nodes and variables at its leaves, we can associate a mathematical expression to it. More precisely, we can define inductively the semantic $\nu(s)$ of the scheme s by:

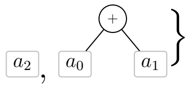
- if s is a leaf, then $\nu(s) = x$ where x is the variable labelling the leaf;
- otherwise, $s = (\diamond,\ \{s_1, s_2\})$ and $\nu(s) := \nu(s_1) \diamond \nu(s_2)$.

Thus, this function ν allows us to go back from an evaluation scheme to the underlying arithmetic expression. With it, we can extend the concept of decomposition to arithmetic expression in the following way:

Definition 5.4. *Let f be an arithmetic expression not reduced to a single variable. For every evaluation scheme $s = (\diamond,\ \{s_1, s_2\})$ for f, we can associate the pair $(\diamond,\ \{\nu(s_1), \nu(s_2)\})$. Such a pair will be called thereafter a* **decomposition** *of f, and we will denote the set of all the decompositions for f by*

$$\mathcal{D}(f) := \big\{(\diamond, \{\nu(s_1), \nu(s_2)\}),\ (\diamond,\ \{s_1, s_2\}) = s \in \mathcal{S}(f)\big\}.$$

Moreover, whenever f is a variable, we define $\mathcal{D}(f) := \emptyset$ and we say that f is **trivial**.

If we consider the example of Figure 5.2, the decomposition associated to the scheme corresponding to the yellow box for instance is $\Big(+,\ \Big\{$  $\Big\}\Big)$. Thus, we deduce the decomposition $(+,\ \{a_2,\ a_0 + a_1\})$ for $a_0 + a_1 + a_2$. By proceeding in the same way with the two other evaluation schemes for $a_0 + a_1 + a_2$, we obtain all its decompositions:

$$\mathcal{D}(a_0 + a_1 + a_2) = \Big\{(+,\ \{a_2,\ a_0 + a_1\}),\quad (+,\ \{a_1,\ a_0 + a_2\}),\quad (+,\ \{a_0,\ a_1 + a_2\})\Big\}.$$

Note that the number of decomposition for an arithmetic expression is always smaller than its number of evaluation schemes, because of the semantic abstraction performed with ν.

The set of decompositions for a given arithmetic formula f will play an important role in the sequel. Indeed, this is the key that will allow us to perform a divide-and-conquer analysis of the set of evaluation schemes $\mathcal{S}(f)$, since it turns a formula f into one operand with two smaller formulas as operands. These formulas can be analyzed recursively in order to deduce some relevant information of $\mathcal{S}(f)$. Because of the fact that f usually has several decompositions, and because of the successive recursive calls we may perform, many formulas are analyzed when we want some information on $\mathcal{S}(f)$. This leads us to introduce the following definition:

Definition 5.5. *Let f be an arithmetic expression. An arithmetic expression g is a* **subexpression** *of f if there exists an unordered binary tree s such that $\nu(s) = g$ and s is a strict subtree of at least one element of $\mathcal{S}(f)$.*

For instance, if we go back to the example of $a_0 + a_1 + a_2$ from Figure 5.2 and look at all the arithmetic expressions that we can form from the strict subtrees of the 3 evaluation schemes, we obtain the following list of subexpressions: a_0, a_1, a_2, a_0+a_1, a_0+a_2, a_1+a_2.

Roughly speaking, the subexpressions of an expression f are all the arithmetic expressions that are computed as intermediate quantities in the evaluation schemes in $\mathcal{S}(f)$. Thus, we may need to consider all of them to analyze $\mathcal{S}(f)$. Note that, because we only consider strict subtrees, f cannot be a subexpression of itself. Moreover, a variable has no subexpression since its only scheme is a leaf, which has no strict subtree.

5.2 Algorithmic analysis of the set of evaluation schemes

Now that we have delimited the set $\mathcal{S}(f)$ of all evaluation schemes for a given arithmetic expression f, we intend to address issues like counting the number of schemes in $\mathcal{S}(f)$, or finding an optimal scheme according to a given criteria. Our approach will consist in elaborating for each problem some generic algorithm, implementing it in `C++`, and testing it on one or several expressions.

As we have seen in the previous section, the concept of decomposition allows us to adopt a divide-and-conquer point of view, and thus to design divide-and-conquer analysis algorithms. The analysis of $\mathcal{S}(f)$ can then be summarized as going through all the decompositions of f, doing recursive calls, and deducing some result from what has been computed recursively. Since we have to deal with several arithmetic expressions recursively, our algorithm should be designed so as it can handle a family of arithmetic expressions $\mathcal{F}$. This section discusses the properties that must be satisfied by $\mathcal{F}$, presents several examples of families of arithmetic expressions, and ends up with some practical considerations.

5.2.1 Requirements for a family of arithmetic expressions

In order for a divide-and-conquer analysis of $\mathcal{S}(f)$ to work for any $f \in \mathcal{F}$, the family of arithmetic expressions $\mathcal{F}$ should however satisfy a few properties:

1. We have to guarantee that any subexpression g of $f \in \mathcal{F}$ is also in $\mathcal{F}$ in order to have a correct input while doing a recursive call.

2. We need a partial order $\prec_{\mathcal{F}}$ such that for each $f, g \in \mathcal{F}$, g is a subexpression of f implies $g \prec_{\mathcal{F}} f$. This is crucial to ensure the termination of the analysis: we need to perform recursive call on "smaller" objects. Notice that one cannot construct an infinite decreasing sequence for $\prec_{\mathcal{F}}$ because subexpressions come from strict subtrees and, as we start with finite trees, we will end up in finite time with a leaf (that is, a tree without subtrees).

3. In fact, the partial order of the previous point is not sufficient. Storing efficiently the results of all the recursive calls requires that we provide a total order $<_{\mathcal{F}}$ on $\mathcal{F}$. As before, we can see that $(\mathcal{F}, <_{\mathcal{F}})$ has a minimum. Notice also that one can define the equality $=_{\mathcal{F}}$ from $<_{\mathcal{F}}$ ($f =_{\mathcal{F}} g \Leftrightarrow \neg(f <_{\mathcal{F}} g)$ and $\neg(g <_{\mathcal{F}} f)$). This was not possible with the partial order $\prec_{\mathcal{F}}$ because of incomparable elements.

4. Finally, this approach can be applied only if we have a way to compute the set $\mathcal{D}(f)$ of all the decompositions for a given $f \in \mathcal{F}$. Therefore, we must suppose that we have a mapping, named `decompose` thereafter, that returns for every $f \in \mathcal{F}$ the list of all the decomposition for f.

Note that, if we have a total order $<_{\mathcal{F}}$ as in point 3, we can lighten the notation for the decompositions of an expression f. In the sequel, we will write such decompositions $(\diamond, f_1, f_2)$ instead of $(\diamond, \{f_1, f_2\})$, with the implicit hypothesis that $f_1 \leq_{\mathcal{F}} f_2$.

5.2.2 Examples of arithmetic expression families

Let us see a few families of arithmetic expressions which satisfy the requirements stated in the previous section.

Powers of a

One easy example is the family of the powers of some variable a, that is, $\mathcal{F} = \{a^n,\ n \in \mathbb{N}_{>0}\}$:

- Any element a^n of $\mathcal{F}$ can be represented efficiently since the whole information lies in its exponent n.
- We can naturally extend the natural order $<$ on $\mathbb{N}_{>0}$ to $\mathcal{F}$ by saying that $x^i <_{\mathcal{F}} x^j$ if and only if $i < j$,
- Given $n \in \mathbb{N}_{>1}$, the decompositions for x^n will all have the form $(\times, x^i, x^j)$, where $1 \leq i \leq j \leq n - 1$ and $i + j = n$. Hence, we can easily generate $\mathcal{D}(x^n)$.
- Finally, we can deduce from the decompositions of x^n that its subexpression set is $\{x^i,\ 1 \leq i \leq n - 1\}$, which is formed by all the expressions less than x^n.

Therefore, $\mathcal{F} = \{a^n,\ n \in \mathbb{N}_{>0}\}$ is a suitable family of arithmetic expressions. In a same way, if we are interested in a particular power a^N with $N > 1$, we can also consider the family $\{a^n,\ 1 \leq i \leq N\}$ with the encoding and total order as above.

Sums of $n+1$ variables

Suppose that we have a sequence $(a_i)_{i\in\mathbb{N}}$ of variables and that we want to evaluate $\sum_{i=0}^{n} a_i$ for some $n \in \mathbb{N}$. Subexpressions for this sum will be of the form $\sum_{i\in I} a_i$ with I some subset of $\{0,\dots,n\}$. Thus, let us consider the family $\mathcal{F} = \{\sum_{i\in I} a_i,\ I \subset \mathbb{N} \text{ finite}\}$.

- An element $f \in \mathcal{F}$ can be represented by its support (the set of indices for all the variables that appear in f). This support I can in turn be encoded as a non-negative integer $\sigma(I) = \sum_{i\in I} 2^i$.
- We can use the natural order in $\mathbb{N}$ to define the order $<_{\mathcal{F}}$. Note that the subexpressions of $f = \sum_{i\in I} a_i$ for a given finite $I \subset \mathbb{N}$ are the sums with a support $J \subsetneq I$, which are less than f with respect to $<_{\mathcal{F}}$ since $\sigma(J) < \sigma(I)$.
- The decompositions for $f = \sum_{i\in I} a_i$ will all be of the form $(+, \sum_{j\in J} a_j, \sum_{k\in K} a_k)$ with[4] $J \sqcup K = I$ and $\sigma(J) < \sigma(K)$ (we cannot have equality here because each variable in f will only appear on one side of the $+$ operator). If we denote by i_M the largest number in I and let $I' := I \setminus \{i_M\}$, we can generate all the pairs (J,K) corresponding to decompositions for f by enumerating all the bipartitions (J,K') of I' with $J \neq \emptyset$ and setting $K = K' \cup \{i_M\}$. Indeed, i_M being the largest number in I, it has to belong to K to ensure that $\sigma(J) < \sigma(K)$, and then J can be any non-empty subset of I'.

Therefore, $\mathcal{F} = \{\sum_{i\in I} a_i,\ I \subset \mathbb{N} \text{ finite}\}$ is a suitable family of arithmetic expressions.

Univariate polynomials

Finally, let us consider the case of univariate polynomials, which is a bit more sophisticated. The main mathematical object of interest is $p(x) = \sum_{i=0}^{n} a_i\, x^i$. Among the subexpressions of $p(x)$, we have, like for the sums of variables, expressions of the form $\sum_{i\in I} a_i\, x^i$ where $I \subsetneq \{0,\dots,n\}$. Moreover, we can find two other types of subexpressions for $p(x)$:

- all the powers x^i with $1 \le i \le n$,
- sums $\sum_{i\in I} a_{i+k}\cdot x^i$ with $I \subsetneq \{0,\dots,n\}$ and $1 \le k \le n - \max_{i\in I} i$.

These subexpressions appear when we apply factorization (rule r_6 page 95) to binary trees for the evaluation of $p(x)$.

Therefore, the family to consider in this case is $\mathcal{F} = \{x^n, n \in \mathbb{N}_{>0}\} \cup \{\sum_{i\in I} a_{i+k}\cdot x^i, I \subset \mathbb{N} \text{ finite}, k \in \mathbb{N}\}$.

- We can encode some $f \in \mathcal{F}$ with a pair of non-negative integers (m_f, n_f):
  1. m_f will represent the support for the sum. Having $m_f = 0$ will mean that f is actually a positive power of x. Otherwise, $f = \sum_{i\in I} a_{i+k}\cdot x^i$ for some finite set I and we define $m_f := \sigma(I) = \sum_{i\in I} 2^i$.

[4] Here and hereafter, the symbol $\sqcup$ is to indicate a union of disjoint sets.

2. n_f will be either the positive exponent n of $f = x^n$ when $m_f = 0$, or the non-negative integer k used to shift the indices in a sum (or, equivalently, the power of x which has already been factored out of f).

- Then, we can define $<_{\mathcal{F}}$ such that $g <_{\mathcal{F}} f$ if and only if $(m_g, n_g) <_{\text{lex}} (m_f, n_f)$, where $<_{\text{lex}}$ is the lexicographical order on $\mathbb{N}^2$. Let us check that this order is compatible with the subexpressions:

  * Subexpressions for x^n are x^i with $1 \leq i \leq n-1$. This case is handled correctly since $(0, i) <_{\text{lex}} (0, n)$.
  * A power x^i is always smaller than a monomial or a sum of monomials since $(0, \cdot) <_{\text{lex}} (\sigma, \cdot)$ when $\sigma > 0$. This covers the case of powers being subexpressions of sums.
  * Finally, $g = \sum_{i \in I} a_{i+k} \cdot x^i$ is a subexpression of $f = \sum_{j \in J} a_{j+k'} \cdot x^j$ when:

    1. $k \geq k'$ (we may have factored by some power of x),

    and

    2. $I + (k - k') := \{i + k - k', \, i \in I\} \subseteq J$ (we may have removed some monomials).

    Note that we cannot have both equalities here, since this would imply $g = f$ and f is not a subexpression of itself.

    This implies that $m_g = \sigma(I) < \sigma(J) = m_f$ since, either $k > k'$ and $\sigma(I) \leq 2^{k'-k}\sigma(J) < \sigma(J)$, or $k = k'$ and we cannot have the equality in the second part so that $I \subsetneq J$, hence $\sigma(I) < \sigma(J)$. Therefore, $g <_{\mathcal{F}} f$ since $(m_g, k) <_{\text{lex}} (m_f, k')$.

- It remains to see how one can compute the set of decompositions for all $f \in \mathcal{F}$. We have already dealt with the case where $f = x^n$ in the first example (even if we need to adjust the encoding for this example). As for $f = \sum_{i \in I} a_{i+k} \cdot x^i$, we get two types of decompositions:

  * First, we can split the sum into two parts as in the second example. Such decompositions are of the form $(+, q(x), r(x))$ where every monomial from $p(x)$ ends up either in $q(x)$, or in $r(x)$, except $a_{i_M} x^{i_M}$ which has to be in $r(x)$ to ensure that $q(x) <_{\mathcal{F}} r(x)$. Generating these decompositions can be done in the same way as we did for sums in the previous example.
  * Second, we can factor by some power x^r with $1 \leq r \leq \min_{i \in I} i$. The corresponding decompositions are $(\times, x^r, \sum_{i \in I-r} a_{i+(k+r)} \cdot x^i$ with $I - r := \{i - r, \, i \in I\}$, and their generation is not an issue.

5.2.3 Practical considerations

Let us first introduce a useful notation:

Notation 5.1. *Given a finite set X, we will denote the subsets of X of size 1 or 2 by $\mathcal{P}_2(X) := \{S \subset X, 1 \leq |S| \leq 2\}$, where $|S|$ denotes as usual the cardinality of the set S.*

This notation allows us to be more precise on the nature of the decompositions for a non-trivial arithmetic expression $f \in \mathcal{F}$. Indeed, since we suppose that any subexpression g of f has to be in $\mathcal{F}$, all the decompositions $(\diamond, \{f_1, f_2\})$ (or $(\diamond, f_1, f_2)$ with $f_1 \leq_{\mathcal{F}} f_2$) for f are mathematical objects belonging to the set $\{+, \times\} \times \mathcal{P}_2(\mathcal{F})$.

We can then summarized the requirements pointed out in Section 5.2.1 as follow:

Assumption 5.1. *In the sequel, each time we speak about a family of arithmetic expressions $\mathcal{F}$, we will assume that we have at our disposal:*

- *a reasonable way to encode the elements of $\mathcal{F}$,*
- *a total order $<_{\mathcal{F}}$ on $\mathcal{F}$,*
- *a map* `decompose` $: \mathcal{F} \to \mathcal{P}\big(\{+, \times\} \times \mathcal{P}_2(\mathcal{F})\big)$

such that:

1. *For all $f \in \mathcal{F}$ and subexpression g of f, g is also in $\mathcal{F}$ and $g <_{\mathcal{F}} f$. In other words, $\mathcal{F}$ is closed with respect to the concept of subexpression, and any expression is always greater than any of its subexpressions.*
2. *For all $f \in \mathcal{F}$,* `decompose`(f) *is the set of all the decompositions for f and can be computed algorithmically.*

In practice, the family $\mathcal{F}$ is given in the form of a class providing two methods $<$ and `decompose`. Elements of $\mathcal{F}$ will then be instances of this class. An example of `C++` interface for such a class is shown in Figure 5.3.

Figure 5.3: Example of `C++` interface for a class implementing a family of arithmetic expressions.

```
class F
{
    private:
        // some internal variable(s)

    public:
        // some relevant constructor(s)
        ~F();

        bool operator<(const F) const;
        std::list<decomp<F> > decompose() const;

        void print() const;
        // some other useful methods
};
```

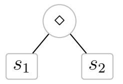
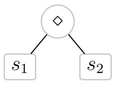
Here, `decomp<F>` *is a structure encoding objects in $\mathcal{D}(\mathcal{F}) = \{+, \times\} \times \mathcal{P}_2(\mathcal{F})$.*

In the sequel, we will introduce several algorithms to generate schemes in $\mathcal{S}(f)$, count them or find optimal schemes according to a given criteria. The input of one such algorithm will always contain some value $f \in \mathcal{F}$. In practice, our `C++` implementation of these algorithms are parameterized by a class `F` using the template mechanism, so that they may be used for any family of arithmetic expressions $\mathcal{F}$ as in Assumption 5.1, as soon as one provides a `C++` class `F` for it.

5.3 Exhaustive generation of the evaluation schemes

The first thing we may want to do with evaluation schemes is to generate them. In this section, we will present a generalized version of the algorithm introduced in [Rev09, §6.1] to compute the evaluation schemes for univariate and bivariate polynomials. Contrary to the original version which was based on dynamic programming [CLRS09, §15], we present here a divide-and-conquer algorithm (Algorithm 5.1) using the memoization technique [Mic68]. With memoization, we do not need to have some extra knowledge on the pattern of the successive recursive calls, as it is the case for dynamic programming. Therefore, it is more suited to deal with families of arithmetic expressions in all generality.

Algorithm 5.1: `Generate`

```
Input     : $f \in \mathcal{F}$
Output    : $\mathcal{S}$ = set of all the evaluation schemes for $f$
Parameter : a hash table $h : \mathcal{F} \to \mathcal{S}(\mathcal{F})$, initially empty, where the results
            obtained during recursive calls will be stored

if $h[f]$ is defined then return $h[f]$   // the result was already computed
$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then $\mathcal{S} \leftarrow \{f\}$   // $f$ is a variable
else $\mathcal{S} \leftarrow \emptyset$
 foreach $(\diamond, f_1, f_2) \in \ell$ do
    $\mathcal{S}_1 \leftarrow$ Generate($f_1$)
    if $f_1 = f_2$ then
        for $\{s_1, s_2\} \in \mathcal{P}_2(\mathcal{S}_1)$ do $\mathcal{S} \leftarrow \mathcal{S} \cup \Big\{$  ◇
s1
s2 $\Big\}$
        // for any set in $\mathcal{P}_2(\mathcal{S}_1)$ of size 1, we set $s_1 = s_2$.
    else
        $\mathcal{S}_2 \leftarrow$ Generate($f_2$)
        for $(s_1, s_2) \in \mathcal{S}_1 \times \mathcal{S}_2$ do $\mathcal{S} \leftarrow \mathcal{S} \cup \Big\{$  ◇
s1
s2 $\Big\}$
$h[f] \leftarrow \mathcal{S}$
return $\mathcal{S}$
```

Theorem 5.1. *Algorithm 5.1 computes the set of all the evaluation schemes for the arithmetic expression $f \in \mathcal{F}$ given as input. Moreover, each time the algorithm adds a new element in $\mathcal{S}$, this element is a new evaluation scheme for f.*

Proof. Let us prove the correctness of Algorithm 5.1 by complete induction on $f \in \mathcal{F}$. When f admits no decomposition (in particular, when f is the minimum for $(\mathcal{F}, <_{\mathcal{F}})$), f is a variable. In this case, the only evaluation scheme for f is the leaf labelled with f and the algorithm is correct.

Let $f \in \mathcal{F}$ be a non-trivial arithmetic formula such that `Generate`(g) is correct for all $g <_{\mathcal{F}} f$. Since f is non-trivial, all its evaluation schemes have the shape $(\diamond, \{s_1, s_2\})$ where each s_i, $i \in \{1, 2\}$, is a scheme for some subexpression $f_i = \nu(s_i)$ of f. We may assume without loss of generality that $f_1 \leq_{\mathcal{F}} f_2$, so that the corresponding decomposition for f of the scheme $(\diamond, \{s_1, s_2\})$ can be written $(\diamond, f_1, f_2) \in \mathcal{D}(f)$. Therefore, the set of evaluation schemes we are aiming at is:

$$\begin{aligned}
\mathcal{S}(f) &= \bigsqcup_{(\diamond, f_1, f_2) \in \mathcal{D}(f)} \left\{ \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array} \in \mathcal{S}(f),\ \nu(s_1) = f_1 \text{ and } \nu(s_2) = f_2 \right\} \\
&= \bigsqcup_{(\diamond, f_1, f_2) \in \mathcal{D}(f)} \left\{ \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array} \in \mathcal{S}(f),\ s_1 \in \mathcal{S}(f_1) \text{ and } s_2 \in \mathcal{S}(f_2) \right\} \\
&= \bigsqcup_{\substack{(\diamond, f_1, f_2) \in \mathcal{D}(f) \\ f_1 =_{\mathcal{F}} f_2}} \left\{ \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array} \in \mathcal{S}(f),\ s_1 \in \mathcal{S}(f_1) \text{ and } s_2 \in \mathcal{S}(f_1) \right\} \\
&\sqcup \bigsqcup_{\substack{(\diamond, f_1, f_2) \in \mathcal{D}(f) \\ f_1 <_{\mathcal{F}} f_2}} \left\{ \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array} \in \mathcal{S}(f),\ s_1 \in \mathcal{S}(f_1) \text{ and } s_2 \in \mathcal{S}(f_2) \right\}.
\end{aligned}$$

Algorithm 5.1 considers all the decompositions for f with its `foreach` loop, and then adds evaluations schemes to the variable S depending on the equality of f_1 and f_2. This approach matches the last part of the aforementioned equation. What remains is to show that, in both cases, the algorithm computes accurately and without redundancy the set of evaluation schemes corresponding to the current decomposition for f.

Let us fix a decomposition $(\diamond, f_1, f_2)$ for f with $f_1 <_{\mathcal{F}} f_2$. Because $f_i <_{\mathcal{F}} f$ for $i \in \{1, 2\}$, the inductive hypothesis implies that S_i is actually $\mathcal{S}(f_i)$. What we have to prove in this case is that

$$\begin{array}{rccl}
\gamma : & \mathcal{S}(f_1) \times \mathcal{S}(f_2) & \to & \left\{ \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array} \in \mathcal{S}(f),\ s_1 \in \mathcal{S}(f_1) \text{ and } s_2 \in \mathcal{S}(f_2) \right\} \\
& (s_1, s_2) & \mapsto & \begin{array}{c} \diamond \\ \diagup\ \diagdown \\ \boxed{s_1}\ \ \boxed{s_2} \end{array}
\end{array}$$

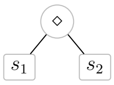
is bijective. This function is obviously surjective. Suppose now that[5] $\gamma(x, y) \equiv \gamma(x', y')$. Because $f_1 \neq_{\mathcal{F}} f_2$, we cannot have $x \equiv y'$. Hence, we deduce that $x \equiv x'$ and $y \equiv y'$ so that (x, y) and (x', y') correspond to the same element in $\mathcal{S}(f_1) \times \mathcal{S}(f_2)$. Therefore, γ

[5] Remember from Section 5.1.2 that evaluation schemes are equivalence classes for the relation $\equiv$, that is, equality modulo commutativity.

is also injective, so it is bijective and we can conclude that Algorithm 5.1 is correct and without redundancy in this case.

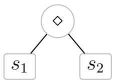
Suppose now that we have a decomposition $(\diamond, f_1, f_2)$ for f where $f_1 =_{\mathcal{F}} f_2$. This time, we know by induction that S_1, which is the result of the unique recursive call, is equal to $\mathcal{S}(f_1)$, and we have to show that

$$\gamma' : \mathcal{P}_2(\mathcal{S}(f_1)) \rightarrow \Bigg\{$$



$$\in \mathcal{S}(f),\ s_1 \in \mathcal{S}(f_1) \text{ and } s_2 \in \mathcal{S}(f_1)\Bigg\}$$

$$\{s_1, s_2\} \mapsto$$



is bijective. As we consider all singletons and all pairs of schemes for f_1 within $\mathcal{P}_2(\mathcal{S}(f_1))$, γ' is surjective. Next, let us assume that $\gamma'(\{x, y\}) \equiv \gamma'(\{x', y'\})$ for some x, y, x', y'. We have either $x \equiv x'$ and $y \equiv y'$, or $x \equiv y'$ and $y \equiv x'$, so we deduce that $x, y \in \{x', y'\}$ (that is, $\{x, y\} \subset \{x', y'\}$) and that $x', y' \in \{x, y\}$ (that is, $\{x', y'\} \subset \{x, y\}$). Therefore, $\{x, y\} = \{x', y'\}$, so that γ' is also injective, and Algorithm 5.1 is correct and without redundancy in this case too. □

The cost of this algorithm highly depends on the final number of evaluation schemes, and we will see in the next chapter that this number can be quite large even for relatively small expressions like a degree-6 polynomial (see Section 6.2.2). Therefore, `Generate` is usually not a practical algorithm. However, it plays a significant theoretical role since algorithms that we will introduce in the sequel are heavily based on this one and share its skeleton. Moreover, algorithm correctness proofs in Chapters 6 and 7 will be much simpler because we will be able to reuse the work we have done here to show that `Generate(f)` actually returns all the evaluation schemes for f and that each scheme was only considered once.

Remark also that we presented here a complete version of the algorithm, with explicit memoization through the use of the hash table h as a parameter. For the sake of simplicity, the following algorithms will be written without the extra lines due to memoization. However, this technique will still be used implicitly and when we will discuss the costs of our algorithms.

5.4 How to model an optimization criterion

The main issue with the evaluation of some arithmetic expression is to find out one or several good evaluation schemes to perform it. Of course, the preferred schemes depend heavily on the context. We may want the scheme to be fast on some architecture with parallelism, or very accurate, or even with as few operations as possible. This section introduces a way to tackle these goals through the concept of measure.

5.4.1 Modelling an optimization criterion with a measure

The basic idea in order to compare schemes and to choose the best ones is to associate to every scheme some value, which will reflect its quality. This leads us to the following definition:

Definition 5.6. *Given a family of arithmetic expressions $\mathcal{F}$, we will call* **measure** *any function $\varphi : \mathcal{S}(\mathcal{F}) \rightarrow T$, where $(T, <_T)$ is a totally ordered set.*

Here, we choose to allow any return type T for a measure as soon as we have a total order on T, which is mandatory in order to be able to compare the quality of schemes. While $(\mathbb{N}, <)$ is a usual choice for T, it can be more relevant to consider more sophisticated sets for two reasons: first, to fit better with what we want to model (see Example 2 below); second, for efficiency (see Example 3 below).

While a measure as in the above definition is sufficient in order to compare schemes, it does not fit well with our idea of analyzing schemes in a divide-and-conquer way. For this purpose, we need to express the measure of a non-trivial scheme $s = (\diamond, \{s_1, s_2\})$ as a function of $\diamond$ and the measures for s_1 and s_2. Thus, we introduce the following definition:

Definition 5.7. *We say that a measure $\varphi : \mathcal{S}(\mathcal{F}) \rightarrow T$ is* **recursively computable** *when there exists a function $\rho : \{+, \times\} \times T \times T \rightarrow T$ such that for every non-trivial scheme $s = (\diamond, \{s_1, s_2\})$, we have:*

$$\varphi(s) = \rho(\diamond, \varphi(s_1), \varphi(s_2)).$$

Notice that $\rho(a, b, c)$ must be symmetric with respect to b and c since we can swap s_1 and s_2 in the decomposition of s.

Not all measures are recursively computable, as we will see in the next section. However, whenever it is the case, the measure φ is completely determined as soon as we have its values on variables and the corresponding function ρ. Figure 5.4 shows what a `C++` interface for a recursively computable measure may look like. Notice that, in addition to ρ (method `rho`) and the values on variables (available through the method `phi`), we also export the return type T along with a total order for it (available through `M::Compare()`). These elements are indeed part of the definition of our measure.

5.4.2 Examples of measures

Let us see three examples of measures that we will use to illustrate some of the algorithms in the next chapters.

Latency on unbounded parallelism

Given an arithmetic expression f, one of the most simple criteria we may want to optimize is the depth of an evaluation scheme for f. This depth is defined as the length of the longest past from the root to any leaf in the DAG corresponding to the evaluation scheme. Actually, operators $+$ and $\times$ may have different costs (like in the ST231 processor). Hence, instead of the length, it becomes more relevant to consider the sum of the costs related to the internal nodes from the root to a leaf. The maximum sum then corresponds to the latency[6] of the evaluation scheme, and our goal is to find the minimal feasible latency

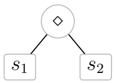
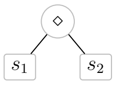
[6] We place ourselves in a situation with unbounded parallelism here, so that an operation can be executed as soon as the values of its operands are known.

Figure 5.4: Example of a C++ interface for a class implementing a measure.

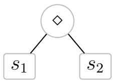
```
class M
{
    public:
        // some useful constructor(s)
        ~Measure();

        // definition of T through typedef

        struct Compare {
            bool operator() (const T& lhs, const T& rhs) const;
        };

        T phi(VARIABLE);
        T rho(OPERATOR, const T&, const T&);

    private:
        // some useful internal variables
};
```

among all the evaluation schemes for f. Notice that depth is handled as the special case where the costs for $+$ and $\times$ are both equal to 1.

Let us denote by $\mathcal{L}(s)$ the latency for the scheme s and by $\mathcal{C}_\diamond$ the cost for operator $\diamond \in \{+, \times\}$, and show that latency is a recursively computable measure. This is a consequence of the following facts:

- $\mathcal{L}($  $) \leq \max\{\mathcal{C}_\diamond + \mathcal{L}(s_1),\ \mathcal{C}_\diamond + \mathcal{L}(s_2)\}$.
  Indeed, a path from $\diamond$ to a leaf is either $\diamond$ followed by a path in s_1, whose cost is at most $\mathcal{C}_\diamond + \mathcal{L}(s_1)$, or $\diamond$ followed by a path in s_2, whose cost is at most $\mathcal{C}_\diamond + \mathcal{L}(s_2)$.
- $\mathcal{L}($  $) \geq \mathcal{C}_\diamond + \mathcal{L}(s_i)$ for $i \in \{1, 2\}$.
  There exists for $i \in \{1, 2\}$ a path p_i in s_i whose cost is $\mathcal{L}(s_i)$, and putting $\diamond$ in front of p_i makes a path in s of cost $\mathcal{C}_\diamond + \mathcal{L}(s_i)$.

Therefore, we can conclude that:

$\mathcal{L}($  $) = \mathcal{C}_\diamond + \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\}.$ (5.1)

This last equation tells us that $\rho(\diamond, a, b) := \mathcal{C}_\diamond + \max\{a,\ b\}$ is an appropriate function to compute $\mathcal{L}(\cdot)$ recursively.

Accuracy

Another interesting property for an evaluation scheme is its accuracy. The true accuracy, defined as the maximum absolute or relative error entailed by the evaluation among all the possible inputs, is usually out of reach. A classical way to tackle accuracy issues is then to use abstractions in order to get a (possibly pessimistic) bound on the error.

We can cite a recent work of Martel [Mar09a, Mar09b], where interval arithmetic [Hig02, §26.4] is used to achieve this abstraction. Any arithmetic expression, evaluated using floating-point arithmetic with rounding to nearest (RN), is associated to two intervals (one to bound the possible values at execution, and one to bound the corresponding error). These pairs of intervals (denoted in bold font hereafter) are computed inductively using the following rules [Mar09a, Figure 2]:

- For each variable, the interval of values has to be provided by the user, and the corresponding error is set to $\mathbf{0} = [0, 0]$.
- We associate
$$\big(\mathbf{v_1} + \mathbf{v_2},\ \boldsymbol{\varepsilon_1} \oplus \boldsymbol{\varepsilon_2} \oplus \frac{1}{2}\mathrm{ulp}(\mathbf{v_1} + \mathbf{v_2})\big)$$
to $f = f_1 + f_2$, where:
  * $(\mathbf{v_i},\ \boldsymbol{\varepsilon_i})$ is the pair of intervals associated to f_i for $i \in \{1, 2\}$;
  * operator $+$ indicates that computations are carried out with rounding to nearest, while operator $\oplus$ is used for computations with rounding down and up so as to ensure that the inclusion property holds:
$$\forall a \in \mathbf{a} := [\underline{a}, \overline{a}], \quad b \in \mathbf{b} := [\underline{b}, \quad \overline{b}], a + b \in \mathbf{a} \oplus \mathbf{b};$$
  * $\mathrm{ulp}([\underline{x}, \overline{x}])$ bounds the actual error due to rounding and is defined by $[-y, y]$ with $y = \max\{\mathrm{ulp}(|\underline{x}|), \mathrm{ulp}(|\overline{x}|)\}$ and $\mathrm{ulp}(x)$ standing for the *unit in the last place*, that is, the weight of the least significant bit of x.

  Notice that it is mandatory to use operator $\oplus$ (and not $+$) for the error part in order to get a correct bound.
- Similarly, we associate to $f = f_1 \times f_2$ the pair
$$\big(\mathbf{v_1} \times \mathbf{v_2},\ \boldsymbol{\varepsilon_1} \otimes \mathbf{v_2}\ \oplus\ \mathbf{v_1} \otimes \boldsymbol{\varepsilon_2}\ \oplus\ \boldsymbol{\varepsilon_1} \otimes \boldsymbol{\varepsilon_2}\ \oplus\ \frac{1}{2}\mathrm{ulp}(\mathbf{v_1} \times \mathbf{v_2})\big),$$
where $(\mathbf{v_i},\ \boldsymbol{\varepsilon_i})$ is again the pair of intervals associated to f_i for $i \in \{1, 2\}$, and $\otimes$ is defined so that the corresponding inclusion property holds.

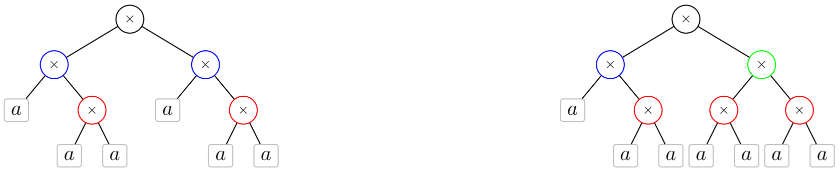
Note that this way of measuring accuracy actually corresponds to a recursively computable measure in our model. Indeed, each expression f is associated to an object $\varphi(s) \in T$, where T is the set of pairs of intervals. Moreover, the inductive rules for $+$ and $\times$ mentioned above allow us to compute $\varphi(f)$ recursively. Finally, a relevant total order for $<_T$ can be defined by looking at the magnitudes and widths of the error and value parts. We will present with more details in Section 8.1.2 a model to analyze accuracy, which is an adaptation of the one mentioned here to fit with the fixed-point arithmetic available on the ST231 processor.

Number of multiplications

In this last example, we consider the number of multiplications as our optimization criterion. We can easily define $\varphi : \mathcal{S}(f) \to \mathbb{N}$ such that $\varphi(s)$ returns the number of internal nodes labelled with operator × in the DAG[7] associated to s. However, this definition is

[7] This DAG is obtained by getting rid of the common subexpressions in the unordered binary tree s.

a typical example of a measure *not* having the property of recursive computability. A counterexample is illustrated in Figure 5.5. Looking at the scheme in Figure 5.4(a), we want to set $\rho(\times, 2, 2) := 3$, since the left and right subschemes have both two multiplications, and the whole scheme has three multiplications (the red, blue, and black ones). However, Figure 5.4(b) tells us that we should have $\rho(\times, 2, 2) = 4$ instead, which conflicts with what we have for Figure 5.4(a).



(a) Example where 2 multiplications on the left and 2 multiplications on the right give 3 multiplications overall.

(b) Example where 2 multiplications on the left and 2 multiplications on the right give 4 multiplications overall.

Figure 5.5: The number of multiplications in evaluation schemes is not recursively computable.

In this case, a solution is to ask for a more precise measure. Indeed, if we define $T := \mathcal{P}\big(\mathcal{S}(\mathcal{F})\big)$ and $\varphi : \mathcal{S}(\mathcal{F}) \to T$ such that $\varphi(s)$ returns the set of all the multiplications (given as the schemes below each internal node labelled with $\times$), we can easily cope with the issue of common subexpressions.

Then, if we define ρ as:

$$\begin{array}{rccc} \rho: & \mathcal{S}(\mathcal{F}) \times \{+, \times\} \times T \times T & \to & T \\ & (s, +, a, b) & \mapsto & a \cup b \\ & (s, \times, a, b) & \mapsto & \{s\} \cup a \cup b \end{array}$$

we can compute φ recursively, from which the number of multiplications can be deduced. Let us make a few remarks about this solution:

1. First, function ρ needs to have access to the current scheme $s = (\diamond, \{s_1, s_2\})$ in addition to $\diamond$, $\varphi(s_1)$, and $\varphi(s_2)$. While this is slightly more complicated than in the definition of recursive computability, it is not a real issue in practice.

2. Second, manipulating sets of schemes is more costly than working with integers. Hence, using this new measure will yield slower algorithms than if we used the measure for latency or accuracy mentioned in the previous two examples.

3. Third, we ask for a total order for $T = \mathcal{P}\big(\mathcal{S}(\mathcal{F})\big)$, but the only natural order on T, which is the inclusion, is a partial order. To get a total order compatible with the number of multiplications, we have to compare the cardinalities, but we still need to add some arbitrary order on sets which share the same cardinality (this can always be done since these are finite sets).

While the situation is not as nice as for latency or accuracy, we will be able to derive interesting results with this measure in Chapter 7. Furthermore, a variation of this measure will be used successfully in Sections 6.4.2 and 8.2.

5.4.3 Generation under constraints

Now that we have a way to evaluate the quality of evaluation schemes, let us see a variant of the generation algorithm introduced in Section 5.3, where we generate only schemes whose measure achieves a certain bound provided by the user.

Here, we fix some family of arithmetic expressions $\mathcal{F}$ and a measure $\varphi : \mathcal{S}(\mathcal{F}) \to T$. We assume that the total order $<_T$ on T is chosen so that a lower value according to $<_T$ means a better quality. The problem is therefore to generate all the schemes s such that $\varphi(s)$ is less than or equal to some bound $B \in T$. This bound B is therefore a new input to our algorithm, and we will need to update it appropriately before each recursive call. Such an update requires that we have a good insight into the evolution of the measure when we decompose a formula, and it will be represented by a function `update`: $\mathcal{D}(\mathcal{F}) \times T \to T$. For instance, supposing we are interested in the number of multiplications, we can define:

$$\begin{array}{rrcl} \texttt{update}: & \mathcal{D}(\mathcal{F}) \times T & \to & T \\ & (+, f_1, f_2),\, b & \mapsto & b \\ & (\times, f_1, f_2),\, b & \mapsto & b-1 \end{array}$$

meaning that, whenever the operator within the decomposition is ×, we consume one multiplication. This function `update`, which will be a parameter to our algorithm, has also to be provided by the user.

The result for this approach is Algorithm 5.2. The name `GenerateWithHint` refers to the fact that we use the measure φ (along with adaptive bounds) as a hint about which scheme is worth generating.

Theorem 5.2. *Provided that for all evaluation scheme $s = (\diamond, s_1, s_2)$ where s_1 and s_2 are schemes for f_1 and f_2, respectively, and for all $b \in T$, we have*

$$\varphi(s) \leq_T b \;\Rightarrow \varphi(s_1) \leq_T b' \;\wedge\; \varphi(s_2) \leq_T b',$$

where $b' := update((\diamond, f_1, f_2),\; b)$, Algorithm 5.2 generates all the evaluation schemes s for f such that $\varphi(s) \leq_T B$. Moreover, each time it tries to add an evaluation scheme to the set r, this scheme is actually a new one for f (that is, unions at lines 10 and 14 are disjoint).

Proof. Let us prove by complete induction on $f \in \mathcal{F}$ that for all $B \in T$, a call to `GenerateWithHint`$(f,\; B)$ returns the set of all the evaluation schemes s for f such that $\varphi(s) \leq_T B$.

When f admits no decomposition (in particular, when f is the minimum for $(\mathcal{F}, <_{\mathcal{F}})$), f is a variable. In this case, the only evaluation scheme for f is f itself, and the algorithm correctly returns, for any B, the singleton $\{f\}$ if $\varphi(f) \leq_T B$ and an empty set otherwise.

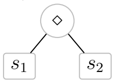
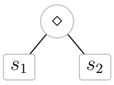
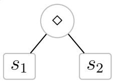
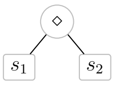
Let $f \in \mathcal{F}$ be a non-trivial arithmetic expression, $B \in T$ be some bound, and suppose that the induction hypothesis holds for all $g <_{\mathcal{F}} f$. As f is non-trivial, any scheme s for f must have the following shape: $(\diamond, s_1, s_2)$ where $\diamond$ is some operator, and s_1 (resp. s_2) is a scheme for $f_1 <_{\mathcal{F}} f$ (resp. $f_2 <_{\mathcal{F}} f$).

Algorithm 5.2: `GenerateWithHint`

Input : $f \in \mathcal{F}$, and a bound $B \in T$.

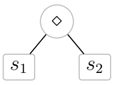
Parameter: A recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to (T, <_T)$, and a function `update`: $\mathcal{D}(\mathcal{F}) \times T \to T$ to adjust the bound before the recursive calls.

Output : The set r of all the evaluation schemes s for f such that $\varphi(s) \leq_T B$.

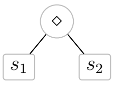
```
ℓ ← decompose(f)
r ← ∅
if ℓ = ∅ and φ(f) ≤_T B then  r ← {f}
else
    foreach (◇, f_1, f_2) ∈ ℓ do
        B′ ← update((◇, f_1, f_2), B)
        r_1 ← GenerateWithHint(f_1, B′)
        if f_1 = f_2 then
            for {s_1, s_2} ∈ P_2(r_1) do
                if φ(  ) ≤_T B then  r ← r ∪ {  }
        else
            r_2 ← GenerateWithHint(f_2, B′)
            for (t_1, t_2) ∈ r_1 × r_2 do
                if φ(  ) ≤_T B then  r ← r ∪ {  }
return r
```

The set we want to generate is:

$\{s \in \mathcal{S}(f),\ \varphi(s) \leq_T B\} =$

$\displaystyle\bigsqcup_{\substack{(\diamond, f_1, f_2)\\ f_1 =_{\mathcal{F}} f_2}}$ $\Big\{ s \in \mathcal{S}(f),\ s =$  with $\{s_1,\ s_2\} \in \mathcal{P}_2(\mathcal{S}(f_1))$ such that $\varphi(s) \leq_T B \Big\}$

$\bigsqcup$

$\displaystyle\bigsqcup_{\substack{(\diamond, f_1, f_2)\\ f_1 <_{\mathcal{F}} f_2}}$ $\Big\{ s \in \mathcal{S}(f),\ s =$  with $(s_1,\ s_2) \in \mathcal{S}(f_1) \times \mathcal{S}(f_2)$ such that $\varphi(s) \leq_T B \Big\}$.

The hypothesis in our theorem tells us that any $s = (\diamond, \{s_1,\ s_2\})$ such that $\varphi(s) \leq_T B$, s_1 and s_2 must satisfy $\varphi(s_1) \leq_T B'$ and $\varphi(s_2) \leq_T B'$. That is to say, if we seek for a scheme s whose value according to φ is less than B, we only need to look at subschemes s_1 and s_2 whose values by φ are less than or equal to B'. But the induction hypothesis applied on f_i ($i \in \{1, 2\}$) and for B' tells us that r_i is the set of schemes for f_i whose value by φ is less than or equal to B', so that we can restrict ourselves to r_i instead of $\mathcal{S}(f_i)$ in the unions above. What we obtain is then exactly what Algorithm 5.2 computes in r, which ends the proof for correctness.

The second part of the theorem is easy to prove. Indeed, the output of the algorithm GenerateWithHint for a given $f \in \mathcal{F}$ and $B \in T$ is included (because of the additional constraint) in the output of Generate for the same f. Otherwise, both algorithms perform the same way by going through the decomposition for f and using the results of recursive calls to fill the current set of schemes. Since the unions in the inner loops were already disjoint unions in Algorithm 5.1, so are they for Algorithm 5.2 which manipulates a smaller data set. □

We can express a few remarks on this new algorithm:

- Its cost is usually hard to determine. If the bound B is too small, then the algorithm will quickly return an empty set. On the opposite, if B is so large that any scheme for f meets this bound, then the computation time will be approximately the same as for Generate. In this case, GenerateWithHint will not be useful in practice. Therefore, B should be chosen with great care.

- If we get a non-empty set at the end of the algorithm, then we know that the minimum of $\varphi(s)$ when $s \in \mathcal{S}(f)$ is smaller than B. Moreover, as we obtain all the schemes whose measure is smaller than B, we can compare their respective measures and deduce the minimum achievable value for $\varphi(s)$. Namely, GenerateWithHint allows us to answer optimization questions.

- Actually, if we want to find out the optimal value for $\varphi(s)$, one solution is to set B to the minimum value of $(T, <_T)$, and then run GenerateWithHint several times with increasing values of B until we finally get a non-empty set of schemes.

- In the actual implementation, we also propose to replace the comparison with the bound B with a predicate parameter that the user can provide. This slightly more general approach makes things easier at least for the case of the number of multiplications, where the bound B would be a set whereas we only want to perform cardinality checks.

Let us conclude this chapter with one example illustrating the usage of Algorithm GenerateWithHint. We have used this algorithm in order to generate all the evaluation schemes for a^n that have a minimum number of multiplications, for n ranging from 1 to 120. This was achieved by successive incrementation of the bound until we get some schemes, as noted above. Then, we have looked over at the set of schemes obtained in order to analyze the number of squarings in these schemes. This is motivated by the fact that squaring may be performed more efficiently than a true multiplication. It happens in several contexts where we can use the symmetry underlying the squaring operation to improve its implementation, like in hardware (see [Mat09] and the references therein), in the software support of floating-point arithmetic [JJLMR11], and in exact linear algebra [Bod10].

Table 5.1 provides a summary of this study. The first terms of four sequences are presented. While the minimal number of multiplications in order to evaluate a^n was already known (see [Knu98, §4.6.3] and the references therein) and appears in the On-Line Encyclopedia of Integer Sequences[8] as sequence `A003313`, the three other sequences are new ones that we have added to the encyclopedia:

- `A186435`(n) gives the number of schemes achieving the minimal number of multiplications.
- `A186437`(n) is the maximal number of squarings that we can extract from an evaluation scheme for a^n with a minimal number of multiplications. This sequence coincides with $\lfloor \log_2(n) \rfloor$ for many values of n. Nevertheless, there are values were `A186437`(n) is one less than $\lfloor \log_2(n) \rfloor$, the first ones being $n = 23, 39, 43, 46, 75, \dots$.
- Finally, `A186520`(n) gives the number of evaluation schemes for a^n that achieve both the minimal number of multiplications and the maximal number of squarings therein. It can be noted that the proportion of such schemes among the schemes with a minimal number of multiplications (that is, the ratio between `A186520`(n) and `A186435`(n)) varies a lot. Obviously, it is equal to 1 when n is a power of 2, but it may become quite small, as we can see with $n = 79$, where we get a ratio of $1/330 \approx 0.003$. In this case, choosing an evaluation scheme for a^{79} that only minimizes the number of multiplications will likely be non-optimal in a context where squaring is faster than a general multiplication.

[8] This encyclopedia, initiated by Sloane, and maintained by the OEIS Foundation Inc., is available at `http://oeis.org/`.

Table 5.1: Values of sequences A003313(n), A186435(n), A186437(n) and A186520(n) for $n \in \{1, \ldots, 120\}$.

n	A003313	A186435	A186437	A186520	timings
1	0	1	0	1	0s
2	1	1	1	1	0s
3	2	1	1	1	0s
4	2	1	2	1	0s
5	3	2	2	1	0s
6	3	2	2	2	0s
7	4	6	2	4	0s
8	3	1	3	1	0s
9	4	3	3	1	0s
10	4	4	3	2	0s
11	5	19	3	4	0s
12	4	3	3	3	0s
13	5	10	3	5	0s
14	5	16	3	10	0s
15	5	4	3	2	0s
16	4	1	4	1	0s
17	5	2	4	1	0s
18	5	7	4	2	0s
19	6	37	4	4	0s
20	5	6	4	3	0s
21	6	31	4	5	0s
22	6	48	4	10	0s
23	6	4	3	2	0s
24	5	4	4	4	0s
25	6	14	4	7	0s
26	6	24	4	12	0s
27	6	5	4	2	0s
28	6	26	4	16	0s
29	7	152	4	47	1s
30	6	12	4	6	1s
31	7	80	4	22	1s
32	5	1	5	1	1s
33	6	2	5	1	1s
34	6	4	5	2	1s
35	7	51	5	4	1s
36	6	12	5	3	1s
37	7	39	5	5	1s
38	7	100	5	10	2s
39	7	20	4	10	2s
40	6	8	5	4	2s
41	7	23	5	6	2s
42	7	90	5	12	2s
43	7	4	4	2	2s
44	7	81	5	18	2s
45	7	14	5	2	2s
46	7	8	4	4	3s
47	8	242	5	10	21s
48	6	5	5	5	21s
49	7	12	5	7	21s
50	7	36	5	17	21s
51	7	4	5	2	21s
52	7	38	5	19	21s
53	8	215	5	55	33s
54	7	16	5	6	33s
55	8	172	5	28	43s
56	7	36	5	22	43s
57	8	190	5	49	52s
58	8	395	5	120	62s
59	8	40	5	8	70s
60	7	24	5	12	70s
61	8	91	5	28	76s
62	8	239	5	68	84s
63	8	94	5	18	90s
64	6	1	6	1	90s
65	7	2	6	1	90s
66	7	4	6	2	90s
67	8	27	6	4	94s
68	7	6	6	3	94s
69	8	29	6	5	98s
70	8	154	6	10	102s
71	9	1485	6	28	1518s
72	7	18	6	4	1518s
73	8	49	6	6	1521s
74	8	104	6	12	1524s
75	8	32	5	15	1527s
76	8	173	6	18	1529s
77	8	4	5	2	1531s
78	8	68	5	34	1534s
79	9	660	6	2	2279s
80	7	10	6	5	2279s
81	8	36	6	8	2280s
82	8	54	6	14	2282s
83	8	4	5	2	2283s
84	8	173	6	21	2285s
85	8	8	6	2	2286s
86	8	8	5	4	2287s
87	9	571	6	8	2786s
88	8	114	6	26	2787s
89	9	523	6	65	3234s
90	8	48	6	6	3235s
91	9	388	6	16	3637s
92	8	12	5	6	3638s
93	9	319	6	8	4000s
94	9	694	6	34	4422s
95	9	199	6	8	4744s
96	7	6	6	6	4744s
97	8	14	6	8	4745s
98	8	28	6	16	4745s
99	8	4	6	2	4746s
100	8	62	6	28	4746s
101	9	280	6	72	4983s
102	8	12	6	6	4983s
103	9	99	6	20	5194s
104	8	52	6	26	5195s
105	9	352	6	64	5385s
106	9	553	6	140	5627s
107	9	30	6	6	5793s
108	8	34	6	12	5793s
109	9	125	6	26	5943s
110	9	579	6	92	6139s
111	9	88	6	10	6269s
112	8	46	6	28	6270s
113	9	146	6	48	6387s
114	9	557	6	126	6549s
115	9	44	6	8	6654s
116	9	688	6	212	6803s
117	9	86	6	14	6900s
118	9	102	6	24	7036s
119	9	24	6	8	7120s
120	8	40	6	20	7120s

Chapter 6

On the combinatorics of evaluation schemes

In this chapter, we will focus on combinatorics issues. First, we explain how to adapt the generation algorithm, introduced in the previous chapter, in order to count the number of evaluation schemes for a given arithmetic expression. This gives us a counting algorithm that we apply to various classes of arithmetic expressions. Thus, we discover two new sequences that we have added to the On-Line Encyclopedia of Integer Sequences. Then, we present some asymptotic results, and more precisely we give asymptotic bounds on the number of schemes for polynomials. Finally, we discuss how, given an additional measure φ, we can perform a finer counting. For that purpose, we introduce two new algorithms: the first one allows us to count the number of schemes with respect to φ; and the second one focuses on counting the number of schemes whose value by φ is below a given threshold.

6.1 Counting evaluation schemes

Suppose that we have a class $\mathcal{F}$ satisfying the requirements exposed in Section 5.2.3. The problem is then to design an algorithm that returns for every $f \in \mathcal{F}$ the corresponding number of evaluation schemes. The idea is to proceed as in Algorithm 5.1, but we settle here for counting how many schemes we encounter instead of actually creating and adding them in a set. What results is Algorithm 6.1.

Before we prove the correctness of Algorithm 6.1, let us state a simple property on $\mathcal{P}_2(X)$ that will be useful for the sequel:

Property 6.1. *If X is finite then* $|\mathcal{P}_2(X)| = \dfrac{|X| \cdot (|X|+1)}{2}$.

Proof. If X is finite, it admits $|X|$ subsets of size 1, and $\dbinom{|X|}{2} = \dfrac{|X| \cdot (|X|-1)}{2}$ subsets of size 2. Adding these two terms gives the result. □

We can now tackle the issue of correctness:

Theorem 6.1. *Algorithm 6.1 is correct.*

Algorithm 6.1: `Count`

```
Input  : $f \in \mathcal{F}$
Output: $n$ = number of evaluation schemes for $f$
$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then  $n \leftarrow 1$
else
    $n \leftarrow 0$
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        if $f_1 = f_2$ then $n \leftarrow n + \frac{\texttt{Count}(f_1) \cdot (\texttt{Count}(f_1) + 1)}{2}$
        else $n \leftarrow n + \texttt{Count}(f_1) \cdot \texttt{Count}(f_2)$
return $n$
```

Proof. Let us proceed by complete induction on $f \in \mathcal{F}$. If f admits no decomposition (in particular when f is the minimum for $(\mathcal{F}, <_{\mathcal{F}})$), it is a variable. Hence, there is one way to evaluate f, and Algorithm 6.1 is correct.

Suppose now that f is not a variable, and that for all $g < f$, $\texttt{Count}(g)$ is the number of evaluation schemes for g. We have seen in Theorem 5.1 that we encountered each scheme for f exactly once during Algorithm 5.1.

Here, instead of combining evaluation schemes with some loop like in Algorithm 5.1, Algorithm 6.1 just adds the number of schemes that would have been produced. This number for a given decomposition $\delta = (\diamond, f_1, f_2)$ depends on the equality of the two subexpressions f_1 and f_2:

- When $f_1 \neq f_2$, there is a bijection between $\mathcal{S}(f_1) \times \mathcal{S}(f_2)$ and the schemes of $\mathcal{S}(f)$ corresponding to δ. Therefore, we have to add $|\mathcal{S}(f_1)| \cdot |\mathcal{S}(f_2)| = \texttt{Count}(f_1) \cdot \texttt{Count}(f_2)$ by induction (each f_i is smaller than f).

- When $f_1 = f_2$, there is a bijection between $\mathcal{P}_2(\mathcal{S}(f_1))$ and the schemes of $\mathcal{S}(f)$ corresponding to δ. By induction, $|\mathcal{S}(f_1)| = \texttt{Count}(f_1)$ and thus, we must add $\frac{\texttt{Count}(f_1) \cdot (\texttt{Count}(f_1) + 1)}{2}$ by Property 6.1.

□

Let us conclude this section with a remark about the cost of Algorithm 6.1. While not explicitly written, we also use memoization in the actual implementation in order to avoid redundancy in the successive recursive calls. Thus, as the arithmetic cost in the main loop is constant,[1] the overall cost is bounded by the product of the number of distinct recursive calls by the maximum size of a decomposition ℓ. Concrete costs for different families of arithmetic expressions will be presented in the next section. Notice that we consider here the number of arithmetic operations and comparisons in $\mathcal{F}$ as the

[1] We assume that elements in $\mathcal{F}$ can be stored efficiently, so that the comparisons $<_{\mathcal{F}}$, and as a consequence $=_{\mathcal{F}}$, can be done in constant time.

cost of `Count`. Yet, core operations consist in basic operations on big integers. Therefore, the binary cost may be higher depending on how big these integers get.

6.2 Application examples

We have implemented the algorithm mentioned in the previous section within CGPE. By fixing one class $\mathcal{F}$ satisfying the constraints mentioned in Section 5.2.3, we obtain an instance of our algorithm for this class. We can then compute the number of evaluation schemes for each $f \in \mathcal{F}$. Usually, we are more interested in a sequence $(f_i)_{i\in\mathbb{N}_{>0}}$ of arithmetic expressions in $\mathcal{F}$. In this case, we obtain an integer sequence made of the number of schemes for the f_is. In this section, we will first see how our approach allows us to retrieve some known OEIS sequences, before dealing with new sequences.

6.2.1 Retrieving three already known sequences

For testing purpose, we have computed the first terms of the following sequences of the OEIS:

- `A001147`(n) = number of ways to sum $n+1$ variables,
- `A001190`(n) = number of ways to evaluate a^n with a commutative non-associative multiplication,
- `A085748`(n) = number of ways to evaluate $a \cdot b^n$ with a commutative non-associative multiplication.

For the sum of variables $\sum_{i=0}^{n} a_i$, there actually exists a closed formula for the number of evaluation schemes (see Property 6.2 in Section 6.3). Therefore, it was easy to validate our implementation using this case. Yet, the cost for our approach is not so good, since it is in $O(4^n)$ instead of $O(n)$ for the closed formula. Indeed, the number of recursive calls is in $O(2^n)$ since we have to consider all the sums $\sum_{i\in I} a_i$ for $I \subset \{0, \ldots, n\}$. Moreover, the number of decompositions for a given sum is also exponential in the number of variables in this sum, which is bounded by $n+1$. So it is in $O(2^n)$. One should note that our algorithm fails to make use of the fact that the number of evaluation schemes for a sum only depends on the number of variables (and not on the name of the variables). If we can forget all the indexes, and group together all the decompositions with the same number of variables at each side of the operator $+$, we obtain a more reasonable cost of $O(n^2)$ ($O(n)$ recursive calls, with $O(n)$ groups of decompositions, and a cost of $O(1)$ per group). We will reuse this remark for the case of polynomials in the next section.

For sequence `A001190`(n) (case of a^n), it turns out that our algorithm reduces to computing the successive terms using the classical recursive formula [Eth37, page 37] along with the technique of memoization. We get $O(n)$ recursive calls, each of them with a cost in $O(n)$, giving us an overall cost in $O(n^2)$. In practice, we are able to compute the 6000 first terms of these sequences in about 1 minute on a desktop machine. However, one can be even more efficient. Indeed, evaluation schemes for a^n are objects that fit into the theory of combinatorial species.[2] In this theory, the sequence corresponding to the number u_n of objects of size n is associated with a so-called generating function, which is basically a series whose coefficients are the u_n's. Pivoteau, Salvy, and Soria [PSS08, Piv08] have shown how to use Newton iterations in order to get the first coefficients of the series in $O(n \log n)$ arithmetic operations.

The results for `A085748`(n) are similar to the ones for `A001190`(n). We still have $O(n)$ recursive calls ($a \cdot b^i$ for $0 \leq i \leq n$, and b^i for $1 \leq i \leq n$), each involving $O(n)$ decompositions. So our algorithm runs in $O(n^2)$. In practice, we obtain the first 4000 terms of `A085748`(n) in about one minute. Yet, the technique based on Newton iteration described above can also be applied in this case so as to get an asymptotic cost of $O(n \log n)$ arithmetic operations.

6.2.2 On the number of schemes for evaluating polynomials

Our initial motivation for counting evaluation schemes lies in the polynomial case [JMM+10]. We want to generate efficient code for evaluating polynomials, and thus we are interested in the growth of the number of evaluation schemes with respect to the degree of the polynomial. This information actually gives us a clue about how scalable the other algorithms applied on polynomials may be, and about the necessity to restrict ourselves by using strong heuristics. In fact, we will study here two types of polynomials: first, univariate polynomials; and second, special bivariate polynomials $q(x, y) = \alpha + y \cdot p(x)$, where α is a constant and p is a univariate polynomial.

Univariate case

Table 6.1 shows the number of evaluation schemes for a univariate polynomial with respect to the degree. This sequence, added in the OEIS as `A169608`, can be obtained by applying Algorithm 6.1 parameterized with the class of univariate polynomials as described at Section 5.2.2. We can easily provide an asymptotic bound on the asymptotic arithmetic cost by noting that:

- the number of recursive calls is equal to

$$\underset{\substack{\nearrow\\ \text{number of}\\ \text{powers of } x}}{n} \quad + \sum_{i=0}^{n} \underset{\substack{\uparrow\\ \text{number of possible}\\ \text{factorizations}}}{(i+1)} \quad \cdot \quad \underset{\substack{\uparrow\\ \text{number of poly-}\\ \text{nomials with a}\\ \text{valuation equal to } i}}{2^{n-i}} \quad = \; 2^{n+2} - 3,$$

- the cost for each recursive call is bounded by $O(2^n)$, since we can do at most n factorizations by a power of x, and at most 2^n support splittings.

[2] See [FS09] for a general introduction to this theory, as well as the study in Section 6.3.2.

Hence, the arithmetic cost is in[3] $O(4^n)$. Notice that the indices do not matter here in the sense that, for instance, the number of schemes for $a_1 + a_2 \cdot x$ is the same as the one for $a_0 + a_1 \cdot x$. Therefore, it is better to use a slightly modified version of the class of univariate polynomials $\mathcal{F}$ introduced in Section 5.2.2, where this behavior is taken into account. We achieve this by redefining the order $<_{\mathcal{F}}$ in the following way:

- $(0, b) <_{\mathcal{F}} (0, d) \Leftrightarrow b < d$,
- if $a, c \in \mathbb{N}, (a, c) \neq (0, 0)$ then $(a, b) <_{\mathcal{F}} (c, d) \Leftrightarrow a < c$.

With this new definition, two polynomials will only be compared through their supports, so that $a_1 + a_2 \cdot x$ and $a_0 + a_1 \cdot x$ will be seen as equal. The new number of recursive calls is then:

$$n \; + \; \sum_{i=0}^{n} 2^{n-i} = 2^{n+1} + n - 1,$$

which is asymptotically half of the previous number.

Table 6.1: Number A169608(n) of evaluation schemes for $p(x)$ with $\deg p = n$.

0	1
1	1
2	7
3	163
4	11602
5	2334244
6	1304066578
7	1972869433837
8	8012682343669366
9	86298937651093314877
10	2449381767217281163362301
11	181946042281864335296699104207
12	35214642830352768473736504891079096
13	176799500809931343108912035970703333111130
14	22949757304967067003110681455541422272014085754
15	76785653012153687095082012207894031948677316414249517
16	660540179458536479946190871277257414784783199736666825008687
17	14576296180923225062225536944551003951936206481743538633380388885044
18	823527733662852414003779291170387870424274811249262438593723335345572925211
19	118912764383373778483239032470394556964632754977476858805545876597866138902807545506
20	43815217447980619517384340909119437904825169188466343660040369661961437018207159467054414458
21	41139424563719124303308971600900790503169908493013011218714101190911828512176192086142942490918929666
22	98306274678792418958407003311891661330085568650718791880371716010534604164335259978285452114558560737182707190
23	597167132160350916374463047269632852142604453612573458534675223786369128352593603859027880740427478752916941197389662649
24	9211905096084970437001158092396260745826915516783761256182607855008023462016541985217669532198776877098993859462172197958883078113
25	360517987734096702705591354595085593601157571179725269984759541321703981558852376110617902415559288049329744309234423108734250571338828849529

In practice, the terms A169608(n) for $n \leq 20$ were obtained in about 105 minutes with our C++ implementation using the new order mentioned above. As we wanted a few more terms, we have also implemented an optimized C version, specialized for univariate polynomials, where:

- powers of x are handled separately,

[3] This bound is slightly pessimistic, and we may actually obtain the sharper bound of $O(3^n)$ by noting that expressions with a support of size i have only $O(2^i)$ decompositions (instead of $O(2^n)$). Yet, the main conclusion still remains that the cost is exponential with respect to n.

- univariate polynomials can therefore be represented by a unique integer for its support (instead of a class encapsulating two integers),
- we use dynamic programming and store the results in a pre-allocated array.

This version gives us the 20th first terms in about 15 minutes, and we were able to compute the values of A169608(n) for $n \leq 25$ in about 4.5 days.

Special bivariate case: $q(x, y) = \alpha + y \cdot p(x)$

Another useful class of arithmetic expressions for the implementation of floating-point operators through polynomial approximation is the class of bivariate polynomials of the form $q(x, y) = \alpha + y \cdot p(x)$ where α is a constant and $p(x)$ is a univariate polynomial [JKMR08].

We can define this class of arithmetic expressions in our framework in the following way:

- We consider the family $\mathcal{F} = \{x^i,\ i \in \mathbb{N}\} \cup \{y \cdot x^i,\ i \in \mathbb{N}_0\} \cup \{\sum_{i \in I} a_{i+k}\, x^i,\ k \in \mathbb{N} \text{ and } I \text{ finite}\} \cup \{\sum_{i \in I} a_{i+k}\, y\, x^i,\ k \in \mathbb{N} \text{ and } I \text{ finite}\} \cup \{\alpha + \sum_{i \in I} a_i\, y\, x^i,\ I \text{ finite}\}$.
- We encode an element $f \in \mathcal{F}$ with:
  - * one boolean b which indicates whether the expression is bivariate (that is, whether y or α is present) or not,
  - * one integer s representing the support of the sum (the lowest bit of s corresponding to α, and its $(i + 1)$st bit corresponding to the presence of i in I). The special case $s = 0$ means that the expression is either x^n or $y\, x^n$,
  - * one integer k for the power of x when $s = 0$ or the shift in the indices when $s \neq 0$.

  Table 6.2 illustrates the encoding for some expressions. Notice that some encodings are forbidden: $(f, 0, 0)$ (x^0 is not in $\mathcal{F}$), $(f, s, \cdot)$ with s an odd number (the presence of α implies that the expression is bivariate), and (t, s, k) with s an odd number and $k \in \mathbb{N}_{>0}$ (the presence of α implies that the shift must be 0).
- We use the lexicographical order for $<_{\mathcal{F}}$ (`true` being larger than `false`).
- Finally, we can compute the decomposition for $f \in \mathcal{F}$ in a way similar to what we did for univariate polynomials. Indeed, we only need to add factorizations by y for bivariate expressions.

Table 6.2: Example of encodings for some bivariate polynomials.
Here, booleans are represented by t (true) and f (false).

Expression	x^3	y	$y\, x^2$	$a_1\, x + a_2\, x^2$	$a_1 + a_2\, x$	$\alpha + a_2\, y\, x^2$
Encoding	$(f, 0, 3)$	$(t, 0, 0)$	$(t, 0, 2)$	$(f, 12, 0)$	$(f, 6, 1)$	$(t, 9, 0)$

We have applied Algorithm 6.1 to this new family of arithmetic expressions, and deduced a new sequence added into the OEIS as `A173157`. Table 6.3 shows the first 21 terms, that were obtained in 12.5 hours. While the encoding here is a bit more complicated than for univariate polynomials, the orders for the number of recursive calls and for the cost per call are the same. So, this algorithm also runs in $O(4^n)$.

Table 6.3: Number `A173157`(n) of evaluation schemes for $q(x,y) = \alpha + y \cdot p(x)$ with $\deg p = n$.

0	1
1	10
2	481
3	88384
4	57363910
5	122657263474
6	829129658616013
7	17125741272619781635
8	1055157310305502607244946
9	190070917121184028045719056344
10	98543690848554380947490522591191672
11	145116280500705029382538760693673579842113
12	600389262260332475581344592426808202086238189757
13	6913679873559110751999558552753066871243850857297843450
14	219818569723083678610243316929195180335704885852763713503976272
15	19162926142068679602438699174852128697491376005582403369119017353063265
16	4552448454789529967485885941331752172203695897453674329533365668776502212569642
17	2931311231540228910575194831264088989654650737483864389059778916867287798067599170920710
18	5091204464543963879691003133701407685542313277599147121851105778554620100305242128889458969346627
19	23749006382789593005844142225233063744306003198014730722303614726659027968689731888694435043822418450563784
20	296379085578725515136571356176087132773942459824021302148422623559448996578176684852299047816041306257212599991533184

Concluding remarks

The results for these two experiments suggest that exhaustive search among the evaluation schemes for polynomials becomes out of reach as soon as the total degree is greater than 5. Typical degrees for polynomial approximants for binary32 (single precision) function being around 10 (see for instance Table 5.3 in [Rev09]), we will have to use heuristics in order to reduce the number of evaluation schemes considered when we seek for a fast and accurate enough way to evaluate such polynomials. Moreover, the fast growth within the two sequences tells us that these heuristics must be drastic if we want our approach to be even slightly scalable.

6.2.3 Summary

All the sequences mentioned in the previous two sections are listed in Table 6.4. This table also gives, for each sequence, the corresponding complexity when using `Count`, and the best known complexity. Recall that we consider here arithmetic complexity, so that we do not take the size of integers involved into account. Even if arithmetic complexity does not reflect the real cost, it is still a relevant indicator to compare the performance between our algorithm and other works. While our general approach fails to achieve a reasonable complexity in the case of sums of variables, where a closed formula exists, we obtain a satisfactory complexity for the two other classical sequences. Better complexity may be achieved through techniques from the theory of combinatorial species. However, this is advanced material, quite different from our approach, and it is not clear to us whether it can cover the polynomial cases that we handle here.

Table 6.4: Summary of the sequences computed with algorithm `Count` along with the corresponding complexities.

expression	complexity		entry in the OEIS	
	with `Count`	best known		
$\sum_{i=0}^{n} a_i$	$O(4^n)$	$O(n)$	`A001147`	classical
a^n	$O(n^2)$	$O(n \log n)$	`A001190`	
$a \cdot b^n$	$O(n^2)$	$O(n \log n)$	`A085748`	
$\sum_{i=0}^{n} a_i \cdot x^i$	$O(4^n)$	–	`A169608`	**new**
$\alpha + y \cdot \sum_{i=0}^{n} a_i \cdot x^i$	$O(4^n)$	–	`A173157`	

6.3 Asymptotics of counting sequences

Several sequences have been mentioned in Section 6.2. Table 6.5 shows the first terms of each of these sequences side by side. It gives us a first idea about how fast these various sequences grow. Our goal in this section is to discuss the asymptotic behavior of `A169608`(n). To achieve this, let us first review some already known results.

Table 6.5: Numbers of evaluation schemes for several arithmetic expressions.

	a^n	$a \cdot b^n$	$\sum_{i=0}^{n} a_i$	$p(x)$ with $\deg p = n$	$\alpha + y \cdot p(x)$ with $\deg p = n$
n	A001190(n)	A085748(n)	`A001147`(n)	`A169608`(n)	`A173157`(n)
0	1	1	1	1	1
1	1	1	1	1	10
2	1	2	3	7	481
3	1	4	15	163	88384
4	2	9	105	11602	57363910
5	3	20	945	2334244	122657263474
6	6	46	10395	1304066578	829129658616013
7	11	106	135135	1972869433837	17125741272619781635
8	23	248	2027025	8012682343669366	10551573103055026072449 46
9	46	582	34459425	86298937651093314877	1900709171211840280457190563 44
10	98	1376	654729075	2449381767217281163362301	985436908485543809474905225911916 72

6.3.1 Preliminary remarks

We have already mentioned the existence of a closed formula for the number of evaluation schemes for $\sum_{i=0}^{n} a_i$. Here is the precise result along with a proof.

Property 6.2. *Let* $\texttt{A001147}(n)$ *be the number of evaluation schemes for* $\sum_{i=0}^{n} a_i$*. We have*

$$\texttt{A001147}(n) = \prod_{i=1}^{n-1} (2i+1) = \frac{(2n)!}{2^n \cdot n!} \sim_{n\to+\infty} \sqrt{2}\left(\frac{2n}{e}\right)^n.$$

Proof. The main idea is first to count the different parenthesizations for $\sum_{i=0}^{n} a_i$, and then to notice that each class modulo commutativity has the same cardinality. Hence, we can easily deduce the number of evaluation schemes from the number of parenthesizations.

A parenthesization for the arithmetic expression $\sum_{i=0}^{n} a_i$ can be viewed as a binary tree with + at its internal nodes and the a_i's in some order at its leaves. One has $(n+1)!$ possible permutations for the a_i's, and the number of binary trees with exactly $n+1$ leaves is the nth Catalan number[4] $C_n = \dfrac{\binom{2n}{n}}{n+1} = \dfrac{(2n)!}{n! \cdot (n+1)!}$. Therefore, we deduce that the number of parenthesizations for $\sum_{i=0}^{n} a_i$ is actually $\dfrac{(2n)!}{n!}$.

Now, let us take a look at what commutativity is in this case. Changing the order of the two operands for a given + corresponds to swapping the two sons of the corresponding internal node in the binary tree. As we have exactly n internal nodes, we can apply commutativity to any subset of these n nodes to obtain an equivalent binary tree modulo commutativity. Moreover, this process allows us to cover all the equivalence classes of a given tree, which has thus a cardinality of 2^n.

Therefore, all the equivalence classes modulo commutativity have the same cardinality 2^n, and the number of evaluation schemes for $\sum_{i=0}^{n} a_i$ is the number of parenthesizations divided by 2^n, that is,

$$\texttt{A001147}(n) = \frac{(2n)!}{2^n \cdot n!}.$$

The first equality in Property 6.2 is easily deduced by unfolding the definition of the factorial at the numerator and denominator, and simplifying all the even numbers. Finally, the asymptotic equivalent is a consequence of Stirling's formula[5] about the equivalence for $n!$. □

The asymptotic study of sequence A001147(n) is fairly simple because all the equivalence classes modulo commutativity share the same cardinality, so that we can go easily from the number of parenthesizations to the number of schemes. Unfortunately, this does not hold for the other sequences we are interested in, which may explain why the results in the next two sections (6.3.2 and 6.3.3) will be less strong than the previous one.

6.3.2 Asymptotic equivalence for sequences A001190 and A085748

The number of evaluation schemes for x^n corresponds to the number of weakly binary trees. This sequence, called Wedderburn-Etherington sequence, has been well studied, and despite the lack of closed formula, we have the following result [Ott48] about the asymptotic behavior of the sequence:

[4] E. Catalan introduced these numbers, which have many interpretations, in 1838. See for instance Example 5.3.12 in [Sta99] for the interpretation mentioned here.

[5] J. Stirling proved in 1730 that $n! \sim_{n\to+\infty} \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$.

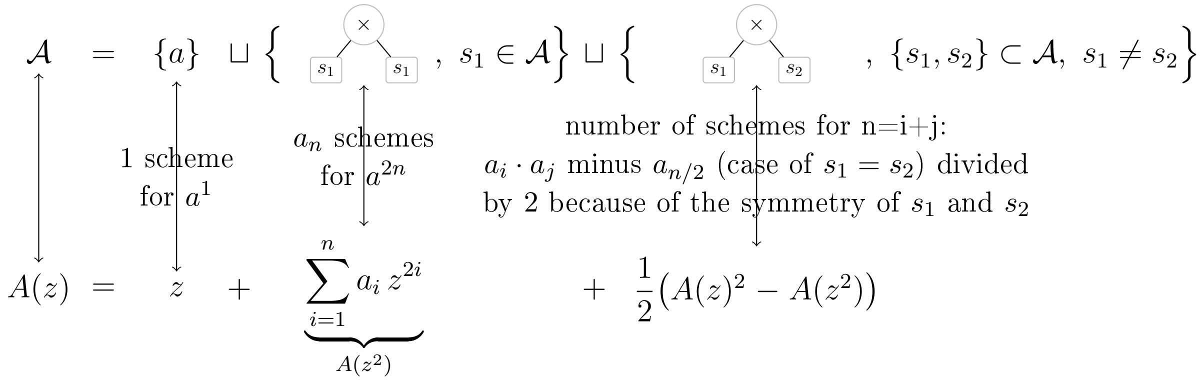
Property 6.3. *Let a_n be the nth term of the Wedderburn-Etherington sequence, that is, the number of evaluation schemes for a^n. We have*

$$a_n \sim_{n\to\infty} \frac{\eta\, \xi^n}{n^{3/2}} \text{ with } \xi \approx 2.48325 \text{ and } \eta \approx 0.31877.$$

Proof. Here, we will only summarize with our notations the main steps of the proof of Proposition VII.5 in [FS09] (see also [Fin94, §5.6] for a similar proof).

First, we consider the set $\mathcal{A}$ of all the evaluation schemes for a^n when $n \in \mathbb{N}_{>0}$. For all $n \in \mathbb{N}_{>0}$, $\mathcal{A}$ contains exactly a_n evaluation schemes for a^n, and we associate to it the series $A(z) := \sum_{i=1}^{+\infty} a_i\, z^i$. Notice that we could prove that the convergence radius for $A(z)$ is at least $1/4$ by comparing a_n with the nth Catalan number.

Now, if we look at the inductive nature of $\mathcal{A}$, we can see that each of its elements is either a or $s_1 \times s_2$, where $\{s_1,\, s_2\} \in \mathcal{P}_2(\mathcal{A})$. We can use this information in order to derive a functional equation satisfied by $A(z)$:



$\mathcal{A} = \{a\} \sqcup \{$ ×, s_1, s_1 , $s_1 \in \mathcal{A}\} \sqcup \{$ ×, s_1, s_2 , $\{s_1, s_2\} \subset \mathcal{A},\ s_1 \neq s_2\}$
1 scheme for a^1
a_n schemes for a^{2n}
number of schemes for n=i+j: $a_i \cdot a_j$ minus $a_{n/2}$ (case of $s_1 = s_2$) divided by 2 because of the symmetry of s_1 and s_2
$A(z) = z + \underbrace{\sum_{i=1}^{n} a_i\, z^{2i}}_{A(z^2)} + \frac{1}{2}\big(A(z)^2 - A(z^2)\big)$

So, we have

$$A(z) = z + \frac{1}{2}\big(A(z)^2 + A(z^2)\big).$$

We can then study this function equation in order to show that

$$A(z) \sim_{z\to 1/\xi} 1 - 2\sqrt{\pi}\,\eta\,\sqrt{1-\xi\, z},$$

where η and ξ are constants that can be efficiently approximated by adapting the approach in [Wed22]. This equivalence allows us to conclude. Indeed, it is known (see [FS09, Corollary VI.1]) that the coefficient a_n of $A(z)$ is equivalent to its counterpart in the series on the right-hand side, for which we can easily see that it is itself equivalent to $\frac{\eta \xi^n}{n^{3/2}}$. □

Sequences A001190 and A085748 are closely related due to the similarity between a^n and $a \cdot b^n$. In fact, one evaluation scheme for $a \cdot b^n$ is made of a multiplied by some scheme for b^{i_1}, the resulting product $a \cdot b^{i_1}$ being in turn multiplied by a scheme for b^{i_2}, and so on until b^{i_r}, for some sequence $(i_1, \ldots, i_r)$ such that $i_1 + i_2 + \cdots + i_r = n$. Let us denote by $\mathcal{B}$ the set of all the evaluation schemes for $a \cdot b^n$ when $n \in \mathbb{N}$. What the previous remark tells us is that there is a natural bijection between $\mathcal{B}$ and the finite sequences of elements in $\mathcal{A}$. This is summarized by the following equation

$$\mathcal{B} = \mathrm{SEQ}(\mathcal{A}), \tag{6.1}$$

using the notation from [FS09, §I.2]. This equation is the key to transpose Property 6.3 to A085748:

Theorem 6.2. *Let b_n be the number of evaluation schemes for $a \cdot b^n$. We have*

$$b_n \sim_{n\to\infty} \frac{\xi^n}{2\pi\,\eta\,\sqrt{n}}$$

with ξ, η as in Property 6.3.

Proof. Like for $\mathcal{A}$, we can associate to $\mathcal{B}$ the series $B(z) := \sum_{i=0}^{+\infty} b_i z^i$. The link between $\mathcal{A}$ and $\mathcal{B}$ expressed in Equation (6.1) implies the following relation for the corresponding series:

$$B(z) = \frac{1}{1 - A(z)}. \tag{6.2}$$

One can think of the similarity between $\frac{1}{1-x} = \sum_{i\leq 0} x^i$ and a sequence of elements of X which is either the empty sequence, or an element of X^i with $i \geq 1$. A formal explanation of this fact can be found in [FS09, Theorem I.1].

Now, we can use Equation (6.2) and the asymptotic equivalent for $A(z)$ mentioned before to deduce the following asymptotic equivalent for $B(z)$:

$$B(z) \sim_{z\to 1/\xi} \frac{1}{2\sqrt{\pi}\,\eta}(1 - \xi\,z)^{-1/2}. \tag{6.3}$$

All that remains is to express the right-hand side as an infinite series, and deduce some asymptotic equivalence for its nth coefficient, which will also hold for b_n. Since

$$\begin{aligned}\frac{1}{(1-z)^{1/2}} &= \sum_{n=0}^{+\infty} \binom{n-1/2}{n} z^n \quad \text{with} \quad \binom{r}{n} = \frac{r(r-1)\dots(r-n+1)}{n!}\\ &= \sum_{n=0}^{+\infty} \frac{(2n)!}{4^n\,(n!)^2} z^n,\end{aligned}$$

we deduce that

$$\frac{1}{2\sqrt{\pi}\,\eta}(1 - \xi\,z)^{-1/2} = \sum_{n=0}^{+\infty} \frac{\xi^n}{2\sqrt{\pi}\,\eta}\frac{(2n)!}{4^n\,(n!)^2} z^n,$$

so that the nth coefficient of the series is equivalent to $\dfrac{\xi^n}{2\pi\eta\,\sqrt{n}}$, which ends the proof. □

6.3.3 Lower and upper bounds on the number of evaluation schemes for polynomials

In this section, we aim at estimating how fast the sequences `A169608`(n) and `A173157`(n) grow when n tends to infinity. Let us focus on univariate polynomials to begin with. The approach mentioned previously for a^n and $a \cdot b^n$ does not seem to fit for this case. In fact, the family $\mathcal{F}$ of arithmetic expressions from which we have deduce `A169608`(n) contains sparse polynomials, which are best represented by sets of integers rather than integers.

This makes it difficult to define properly a series corresponding to the set $\mathcal{S}(\mathcal{F})$. Moreover, we cannot adopt a labelled point of view where we fix an unordered tree and then put the a_i's in any order at the different leaves. This covers the case of $\sum_{i=0}^{n} a_i$, where the variables play the same role, but we do not have such a symmetry in a polynomial.

Therefore, given $n \in \mathbb{N}$, we propose to find two sets of schemes S_n and S'_n satisfying

$$S_n \subset \mathcal{S}(\sum_{i=0}^{n} a_i\, x^i) \subset S'_n$$

and such that we could easily express their cardinalities as a function of the sequences from the previous section. Then, asymptotics for $\log |S_n|$ and $\log |S'_n|$ will give us a lower and an upper bound, respectively, for the asymptotics of $\log \texttt{A169608}(n)$, that is, a precise information about the growth of $\texttt{A169608}(n)$.

Lower bound for $\texttt{A169608}(n)$

In order to get a good lower bound, we need to consider a significant part of the schemes in $\mathcal{S}(\sum_{i=0}^{n} a_i\, x^i)$. We choose here to consider the set S_n of all schemes corresponding to a parallel evaluation of all the monomials $a_i \cdot x^i$ for $0 \leq i \leq n$ followed by a sum of the $n+1$ resulting quantities. Actually, S_n is what we would obtain if we remove factorization (rule r_6) from our set of rules in Section 5.1.1. As we only remove one rule, we hope that S_n will be close to $\mathcal{S}(\sum_{i=0}^{n} a_i\, x^i)$. The number of schemes in S_n satisfies:

$$|S_n| = \underset{\substack{\uparrow \\ \text{number of ways to add} \\ \text{the } n+1 \text{ monomials}}}{\texttt{A001147}(n)} \cdot \prod_{i=0}^{n} \underset{\substack{\nwarrow \\ \text{number of ways} \\ \text{to evaluate } a_i \cdot x^i}}{b_i}$$

so that

$$\log |S_n| = \underbrace{\log \texttt{A001147}(n)}_{\underset{n\to+\infty}{\sim} \; n\log n} + \sum_{i=0}^{n} \underbrace{\log b_i}_{\underset{i\to+\infty}{\sim} \; i\log\xi} . \tag{6.4}$$

Indeed, the asymptotics for $\log \texttt{A001147}(n)$ is straightforward given the closed formula from Property 6.2 and Stirling's formula, and the asymptotics for $\log b_i$ can be verified as follows:

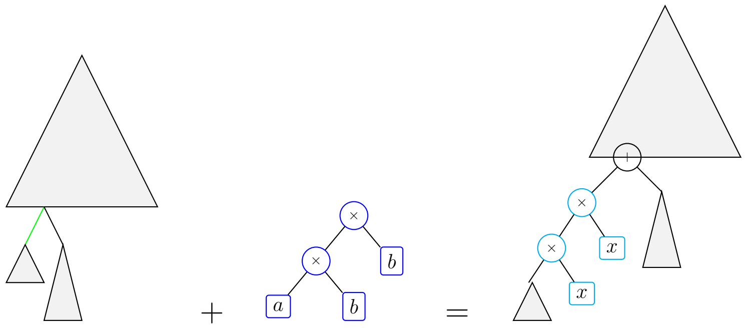
$$\frac{\log b_i}{i\log\xi} = \frac{\log b_i}{i\log\xi - \frac{1}{2}\log i} \cdot \frac{i\log\xi - \frac{1}{2}\log i}{i\log\xi} = \Big(1 + \underbrace{\frac{\log(b_i \cdot \sqrt{i}/\xi^i)}{\log(\xi^i/\sqrt{i})}}_{\underset{i\to+\infty}{\longrightarrow} \; -\log(2\pi\eta)/+\infty=0}\Big) \cdot (1 + o(1)) \underset{i\to+\infty}{\longrightarrow} 1.$$

Now, since $i\log\xi$ is non-negative for all $i \in \mathbb{N}$ and $\sum i\log\xi$ diverges, we deduce that

$$\sum_{i=0}^{n} \log b_i \sim_{n\to+\infty} \frac{n(n+1)}{2}\log\xi \sim_{n\to+\infty} n^2 \log\sqrt{\xi}.$$

Figure 6.1: How to insert multiplications in order to turn a scheme for $\sum_{i=0}^{n} a_i$ into a scheme for $\sum_{i=0}^{n} a_i\, x^i$.

For each edge in the scheme for $\sum_{i=0}^{n} a_i$ (the green one here), we choose a scheme for $a \cdot b^i$ with $0 \le i \le n$ (here, we choose the blue scheme corresponding to $i = 2$), and we insert this scheme alongside the edge.



Therefore, the sum in Equation (6.4) is predominant over $\log \texttt{A001147}(n)$, and we can conclude that $\log |S_n| \sim_{n \to +\infty} n^2 \log \sqrt{\xi}$.

Since $\log \texttt{A169608}(n) \ge \log |S_n|$, this tells us that the asymptotic growth of the logarithm of the number of schemes for univariate polynomials is at least quadratic. If we look at the overall shape of the numbers of schemes in Table 6.1, we can indeed see that the curve made of the leading digits looks like a parabola.

Upper bound for $\texttt{A169608}(n)$

In order to get an upper bound, we will consider all the unordered binary trees of the following form:

- First, we start from an evaluation scheme for $\sum_{i=0}^{n} a_i$. We have thus an unordered binary tree with $2n$ edges.
- Second, we choose for each edge one evaluation scheme for $a \cdot b^i$ with $0 \le i \le n$ and we insert it at the level of the edge as illustrated in Figure 6.1.

This gives us a new set S'_n. We can first notice that, if we only allow the insertions at edges connecting an internal node to a leaf then we would have obtained the set S_n introduced for the lower bound. Now, by allowing insertions everywhere, we are able to restore all the chains of multiplications within a given evaluation scheme for $\sum_{i=0}^{n} a_i\, x^i$ given its skeleton (that is, the unordered binary tree where we have removed all the internal nodes labelled with a multiplication). Of course, this construction creates many unordered binary trees whose underlying mathematical expression is not $\sum_{i=0}^{n} a_i\, x^i$. In fact, we obtain $\sum_{i=0}^{n} a_i\, x^{e_i}$ for some non-negative integers $(e_0, \ldots, e_n)$ depending on how we have inserted schemes.

Remark that we limit ourselves to insert schemes for $a \cdot b^i$ with $0 \le i \le n$ since the maximum power of x in $\sum_{i=0}^{n} a_i\, x^i$ is x^n, and that we allow the trivial scheme for $a \cdot b^0 = a$ so that we can leave some edges untouched.

The number of unordered binary trees in S'_n is:

$$|S'_n| \;=\; \underset{\substack{\uparrow \\ \text{number of} \\ \text{possible skeletons}}}{\texttt{A001147}(n)} \;\cdot\; \underset{\substack{\nwarrow \\ \text{number of schemes} \\ \text{for evaluating } a \cdot b^i \\ \text{with } 0 \le i \le n}}{\left(\prod_{i=0}^{n} b_i\right)^{\overset{\text{number of edges in the skeleton}}{\overset{\swarrow}{2n}}}}$$

Now, we can proceed as for the lower bound. First we apply the logarithm to both sides of the previous identity:

$$\log|S'_n| \;=\; \underbrace{\log \texttt{A001147}(n)}_{\sim_{n\to+\infty}\; n\log n} \;+\; 2n \cdot \sum_{i=0}^{n} \underbrace{\log b_i}_{\sim_{i\to+\infty}\; i\log\xi} \;.$$

The same argument holds for the asymptotics of $\log b_i$, and the one of $\sum_{i=0}^{n} \log b_i$. So we conclude that $\log|S'_n| \sim_{n\to+\infty} n^3 \log\xi$. While this does not match the lower bound mentioned earlier, it is not too far from it. Moreover, this upper bound proves that the number of evaluation schemes for $p(x)$ is not doubly exponential.

Bounds for $\texttt{A173157}(n)$

We can formulate of few remarks concerning $\texttt{A173157}(n)$:

- Obviously, any scheme for $p(x)$ gives us a scheme for $q(x,y) = \alpha + y \cdot p(x)$, so that $\texttt{A173157}(n) \geq \texttt{A169608}(n)$.
- Actually, we even have $\texttt{A173157}(n) \geq \texttt{A169608}(n+1)$. Indeed, replacing y with x in a scheme for $q(x,y)$ gives us a scheme for evaluating a degree-$(n+1)$ polynomial, and this mapping is surjective. We can move y before evaluating $y \cdot x^i$ using associativity and commutativity, which becomes impossible when y is turned into x.
- Finally, we can take a scheme for a degree-$(n+1)$ univariate polynomial, which will have at most $\frac{(n+1)(n+2)}{2}$ leaves labelled with x, and decide for each of those leaves to turn the x into a y. This is kind of the opposite of the process mentioned in the previous point. It allows us to get back, among others, all the schemes for $q(x,y) = \alpha + y \cdot p(x)$ (with $\deg p = n$). Therefore, using this idea, we can deduce the identity
$$\texttt{A173157}(n) \leq (n+2)^{2n+3} \cdot \texttt{A169608}(n+1),$$
which is actually a consequence of Property 6.4 below.

Property 6.4. *Let $(m,k) \in \mathbb{N}^2$, $p(x)$ be the univariate polynomial whose encoding is $(m, k+1)$, and $q(x,y)$ be the special bivariate polynomial whose encoding is $(true, m, k)$. Then, for all $s \in \mathcal{S}(q(x,y))$, there exists $s' \in \mathcal{S}(p(x))$ such that changing at most* $\max\{|m|, 1\}$ *instances of x by y in s' gives (modulo a renaming of the coefficients) s, where $|m|$ is the number of bits with a value of* 1 *in m.*

Proof. We proceed by strong induction on $(m,k) \in (\mathbb{N}^2, <_{\text{lex}})$:

- For $(0,0)$, we have $q(x,y) = y$, $p(x) = x$, $|m| = 0$, and the result holds since we need to replace $1 = \max\{|m|,\, 1\}$ instance of x in the sole scheme for x in order to get the sole scheme for y.

- For $(m,k) = (1,0)$, we have $q(x,y) = \alpha$, and we do not need to change anything in the sole scheme $s' \in \mathcal{S}(a_1)$ to obtain the sole scheme s of $q(x,y)$.

- Otherwise, (m,k) is such that the bivariate polynomial $q(x,y)$, whose encoding is (true, m, k), is non-trivial. Indeed, either $|m| > 1$ and we can (at least) split the monomials in $q(x,y)$ to form a sum of two new polynomials, or m is even[6] and we can (at least) factor by some power of x.

  So, let $s := (\diamond,\, \{s_1, s_2\}) \in \mathcal{S}(q(x,y))$. We need to find $s' \in \mathcal{S}(p(x))$ satisfying the statement in our property. Let $q_1(x,y)$ and $q_2(x,y)$ be such that $s_i \in \mathcal{S}(q_i(x,y))$ and the encoding for $q_i(x,y)$ is $(m_i,\, k_i)$, for $i \in \{1,2\}$. Let us distinguish between four cases:

  * If $\diamond = +$, we have $|m| > 1$ and $|m| > |m_i| \geq 1$ for $i \in \{1,2\}$. We can apply the induction hypothesis on s_1 and s_2 in order to get s'_1 and s'_2, and define $s' = (\diamond, \{s'_1, s'_2\})$. Since we can go from s'_i to s_i by turning at most $|m_i|$ instances of x into y, for $i \in \{1,2\}$, we can go from s' to s in at most $|m_1| + |m_2| = |m| = \max\{|m|,\, 1\}$ changes;

  * If $\diamond = \times$, $m \neq 0$, and, say, $q_1(x,y) = x^j$ for some positive integer j, we have $(m_2, k_2) = (m/2^j, k+j) <_{\text{lex}} (m,k)$ and we conclude by applying the induction hypothesis on s_2. Indeed, we then obtain a scheme s'_2, so that we can define $s' = (\diamond, \{s_1, s'_2\})$ which is a scheme of $p(x)$, and from which we can to s in at most $|m_2| = |m|$ changes (in s'_2);

  * Finally, if $\diamond = \times$ and, say, $q_1(x,y) = y \cdot x^j$ for some non-negative integer j, we have $(m_1, k_1) = (0, j) <_{\text{lex}} (m,k)$ (either $m > 0$, or $m = 0$ and $q(x,y) = y \cdot x^k$ with $k > j$). In the case, we conclude by applying the induction hypothesis on s_1. We then obtain s'_1, so that we can define $s' = (\diamond, \{s'_1, s_2\})$ which is a scheme of $p(x)$ and from which we can go to s in at most $\max\{|m_1|,\, 1\} = 1 \leq \max\{|m|,\, 1\}$ changes (in s'_1).

□

[6] Recall from page 122 that $k > 0$ implies that m must be even, and notice that, for $k = 0$, $|m| = 1$ with m an odd integer only happens for $(m,k) = (1,0)$, which we have already considered.

With this property, we can conclude that

$$\begin{aligned}
\texttt{A173157}(n) \leq & \overbrace{\sum_{k=0}^{n+1} \binom{(n+1)(n+2)/2}{k}}^{\substack{\text{bound on the number of ways to turn} \\ \text{at most } n+1 \text{ instances of } x \text{ into } y \text{ in a} \\ \text{scheme for } p(x) \text{ with } \deg(p) = n+1}} \cdot \texttt{A169608}(n+1) \\
& \leq \sum_{k=0}^{n+1} \binom{(n+2)^2}{k} \cdot \texttt{A169608}(n+1) \leq \sum_{k=0}^{n+1} (n+2)^{2k} \cdot \texttt{A169608}(n+1) \\
& \leq \sum_{k=0}^{n+1} (n+2)^{2n+2} \cdot \texttt{A169608}(n+1) \leq (n+2)^{2n+3} \cdot \texttt{A169608}(n+1).
\end{aligned}$$

Therefore, we have

$$\log\big(\texttt{A169608}(n+1)\big) \leq \log\big(\texttt{A173157}(n)\big) \leq \log\big(\texttt{A169608}(n+1)\big) + (2n+3)\log(n+2),$$

and it is straightforward to conclude that:

- $\log\big(\texttt{A173157}(n)\big) \in \Theta\Big(\log\big(\texttt{A169608}(n+1)\big)\Big)$,
- $\log\big(\texttt{A173157}(n)\big) \in \Omega(n^2)$,
- $\log\big(\texttt{A173157}(n)\big) \in O(n^3)$.

Summary of the different bounds

Let us summarize the different bounds that we have obtained in this section:

Theorem 6.3. *We have* $\log\big(\texttt{A169608}(n)\big) \in \Omega(n^2)$ *and* $\log\big(\texttt{A169608}(n)\big) \in O(n^3)$. *Furthermore, we have the following inequalities:*

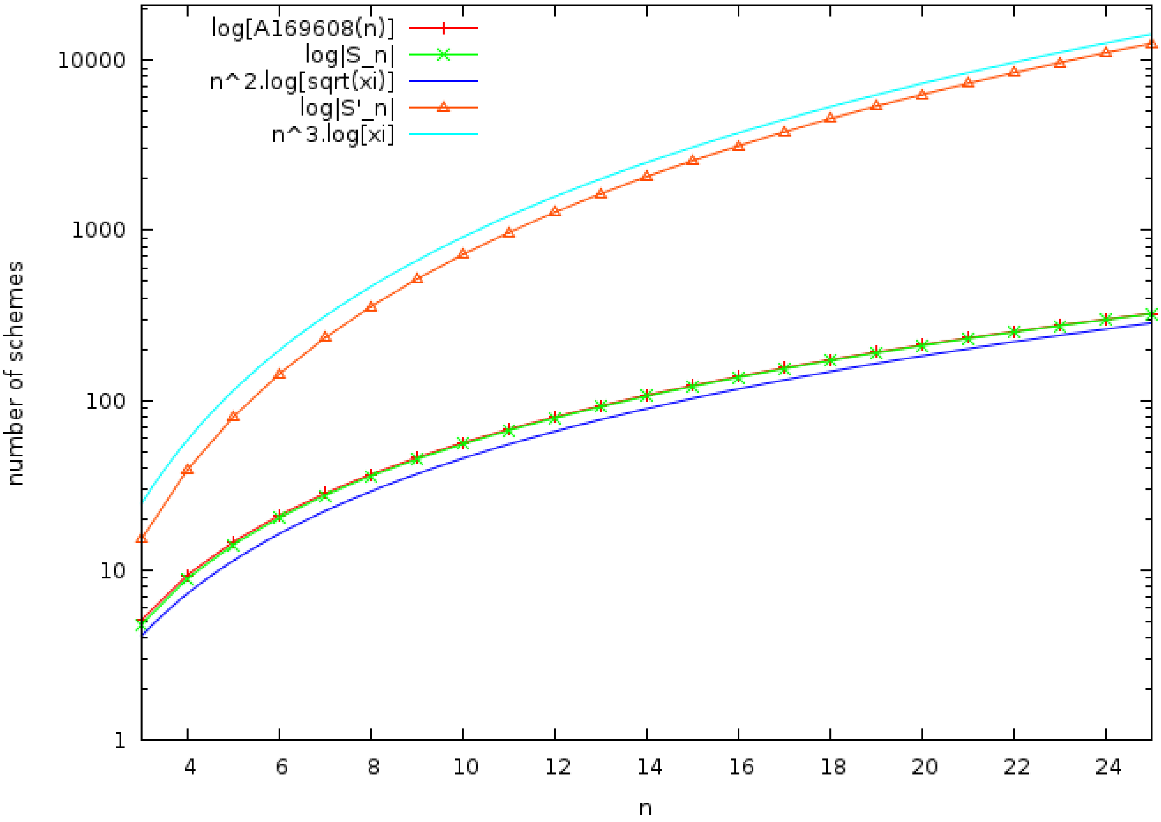
$$\texttt{A169608}(n+1) \leq \texttt{A173157}(n) \leq (n+2)^{2n+3}\texttt{A169608}(n+1)$$

so that $\log\big(\texttt{A173157}(n)\big) \in \Theta(\log\big(\texttt{A169608}(n+1)\big))$.
As a consequence, we also have $\log\big(\texttt{A173157}(n)\big) \in \Omega(n^2)$ *and* $\log\big(\texttt{A173157}(n)\big) \in O(n^3)$.

To conclude this asymptotic study, let us see graphically how the different sequences and bounds mentioned in this section compare. For the univariate case, we can see in Figure 6.2 that the lower bound $\log|S_n|$ for $\log\big(\texttt{A169608}(n)\big)$ is quite sharp compared to the upper bound $\log|S'_n|$. Indeed the curves for $\log|S_n|$ and $\log\big(\texttt{A169608}(n)\big)$ are overlaid in the figure. Therefore, we speculate that $\log\big(\texttt{A169608}(n)\big)$ is actually in $\Theta(n^2)$.

Finally, Figure 6.3 illustrates how the sequence $\log\big(\texttt{A173157}(n)\big)$ is located in relation with its lower and upper bounds. As we have said, the asymptotic behavior of $\log\big(\texttt{A173157}(n)\big)$ coincides with the one of $\log\big(\texttt{A169608}(n+1)\big)$. Again, by comparing with the curves of the asymptotics found for the lower and upper bounds in the univariate case, it is more likely that the order of magnitude for $\log\big(\texttt{A169608}(n)\big)$ is $\Theta(n^2)$.

Figure 6.2: Log-lin graph of the different functions mentioned during the asymptotic study of `A169608`(n).



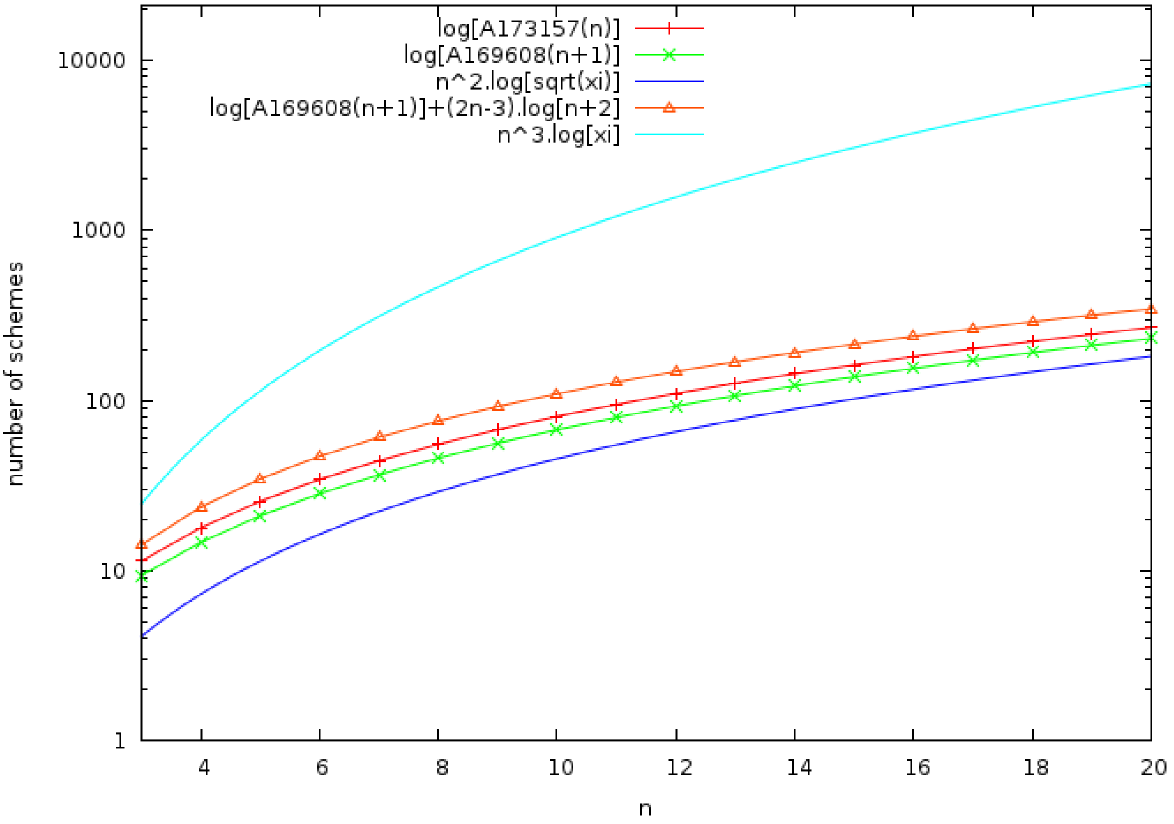
6.4 Counting evaluation schemes with respect to a given measure

One question that may arise when trying to optimize some criterion is to find the number of optimal schemes. More generally, when we have some measure defined on the evaluation schemes, we may want to know how the different evaluation schemes split up according to this measure. After introducing a new algorithm for this purpose in Section 6.4.1, we will discuss the number of multiplications and the latency of evaluation schemes for univariate polynomials in Sections 6.4.2 and 6.4.3, respectively. Finally, we will discuss how to restrict ourselves to nearly optimal schemes, and what we gain by doing so in Section 6.4.4.

6.4.1 A finer-grained adaptation of the generation algorithm

Let us assume that we have a recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to T$. Hence, we have the values associated to all the trivial schemes, plus a function $\rho : \{+, \times\} \times T \times T \to T$ allowing us to compute the measure of a non-trivial scheme. While we have focused on the cardinality of $\mathcal{S}(f)$ for $f \in \mathcal{F}$ in Algorithm 6.1, we propose here to consider the partition of $\mathcal{S}(f)$ according to φ and thus to propagate recursively the distribution of the number

Figure 6.3: Log-lin graph of the different functions mentioned during the asymptotic study of `A173157`(n).



of schemes for f according to φ, represented as mapping $r : T \to \mathbb{N}$. This approach is presented in Algorithm 6.2. Note that, in this algorithm, we assume that each time we try to access a value $r(t)$ which was not previously defined, the corresponding value is 0.

Theorem 6.4. *Algorithm 6.2 is correct.*

Proof. Let us proceed by complete induction on $f \in \mathcal{F}$. If f admits no decomposition (in particular when f is the minimum for $(\mathcal{F}, <_{\mathcal{F}})$), it is a variable. Hence, f has only one evaluation scheme which is f itself, and Algorithm 6.2 returns $r : \varphi(f) \to 1$, which is correct.

Suppose now that f is not a variable, and that for all $g < f$, `CountPerMeasure`(g) is correct and gives us the distribution of the evaluation schemes for g with respect to the measure φ.

As we have seen in Theorem 5.1, we encountered each scheme for f exactly once during Algorithm 5.1.

Let $s := (\diamond, \{s_1, s_2\})$ be such a scheme. Each s_i is thus a scheme for some arithmetic expression $f_i <_{\mathcal{F}} f$. Moreover, $f_1 \leq_{\mathcal{F}} f_2$. We then have two cases:

- Either $f_1 <_{\mathcal{F}} f_2$. In this case, the schemes $s := (\diamond, \{s_1, s_2\})$ for f corresponding to

Algorithm 6.2: `CountPerMeasure`

```
Input  : $f \in \mathcal{F}$, and a measure function $\varphi$ : schemes $\mapsto T$
Output: a partial map $r : T \to \mathbb{N}$ which gives for every $t \in Dom(r)$ the number of
        evaluation schemes $s$ for $f$ such that $\varphi(s) = t$
$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then define $r$ with $r(\varphi(f)) \leftarrow 1$
else
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $r_1 \leftarrow$ CountPerMeasure($f_1$)
        if $f_1 = f_2$ then
            for $\{t_1, t_2\} \in \mathcal{P}_2(Dom(r_1))$ do
                $t \leftarrow \rho(\diamond, t_1, t_2)$
                if $t_1 = t_2$ then $r(t) \leftarrow r(t) + \frac{r_1(t_1) \cdot (r_1(t_1) + 1)}{2}$
                else $r(t) \leftarrow r(t) + r_1(t_1) \cdot r_1(t_2)$
        else
            $r_2 \leftarrow$ CountPerMeasure($f_2$)
            for $(t_1, t_2) \in Dom(r_1) \times Dom(r_2)$ do
                $t \leftarrow \rho(\diamond, t_1, t_2)$
                $r(t) \leftarrow r(t) + r_1(t_1) \cdot r_2(t_2)$
return $r$
```

the decomposition $(\diamond, f_1, f_2)$ are in bijection with

$$\mathcal{S}(f_1) \times \mathcal{S}(f_2) = \bigsqcup_{t_1 \in Dom(r_1)} \bigsqcup_{t_2 \in Dom(r_2)} \mathcal{S}(f_1)|_{t_1} \times \mathcal{S}(f_2)|_{t_2},$$

where $\mathcal{S}(f_i)|_{t_i} := \{s \in \mathcal{S}(f_i), \varphi(s) = t_i\}$. By assumption, $\varphi(s)$ only depends on $\diamond$, $\varphi(s_1)$ and $\varphi(s_2)$, so that all $(s_1, s_2) \in \mathcal{S}(f_1)|_{t_1} \times \mathcal{S}(f_2)|_{t_2}$ lead to schemes s with the same value $t := \varphi(s) = \rho(\diamond, t_1, t_2)$. Algorithm 6.2 thus considers each pair (t_1, t_2) and adds to $r(t)$ the corresponding number of schemes, that is, $r_1(t_1) \cdot r_2(t_2)$.

- Or $f_1 =_{\mathcal{F}} f_2$. In this case, the schemes $s := (\diamond, \{s_1, s_2\})$ for f corresponding to the decomposition $(\diamond, f_1, f_1)$ are in bijection with

$$\mathcal{P}_2(\mathcal{S}(f_1)) = \bigsqcup_{t_1 \in Dom(r_1)} \mathcal{P}_2(\mathcal{S}(f_1)|_{t_1}) \quad \sqcup \quad \bigsqcup_{\substack{t_1, t_2 \in Dom(r_1) \\ t_1 <_T t_2}} \mathcal{S}(f_1)|_{t_1} \times \mathcal{S}(f_1)|_{t_2}.$$

Indeed, if $s_1, s_2 \in \mathcal{S}(f_1)$, we have to consider $\{s_1, s_2\}$ only once, and:

* either $\varphi(s_1) \neq_T \varphi(s_2)$, and the scheme is taken into account once in the disjoint union on the right side,

  * or $\varphi(s_1) = \varphi(s_2)$ and the scheme is taken into account once in the disjoint union on the left side because of $\mathcal{P}_2(\cdot)$.

Then, again, each set of the union is constant with respect to φ, and Algorithm 6.2 adds the corresponding cardinalities correctly.

□

Remark that Algorithm `CountPerMeasure` may be quite significantly costly than Algorithm `Count`, especially when there are a lot of possible values for $\varphi(s)$ when $s \in \mathcal{S}(f)$. After illustrating this problem on two examples, we will see how to get good performance if we are only interested in nearly optimal schemes.

6.4.2 Number of evaluation schemes for polynomials with respect to the number of multiplications

We can apply Algorithm 6.2 on univariate polynomials in order to deduce the distribution of schemes according to the number of multiplications. For this purpose, we will use for φ a variant of the measure we introduced at the end of Section 5.4.2. In fact, we do not need to keep trace of all the multiplications in schemes as before, since only multiplications of type $x^i \times x^j$ may occur more than once.[7] Therefore, we propose to do the following:

- First, we precompute all the evaluation schemes for x^i where $1 \leq i \leq n$ and we number them.
- Then, we define $T := \mathbb{N} \times \mathcal{P}(\mathbb{N}^2)$ and $\varphi : \mathcal{S}(\mathcal{F}) \to T$. The integer will correspond to the number of multiplications involving at least one coefficient a_i, and the set will list all the multiplications of the type $x^i \times x^j$, represented as pairs $(i + j,\, \sigma)$ where $i + j$ is the corresponding power of x and where σ indicates the actual scheme in use for x^{i+j}.
- Trivial schemes have no multiplication, so we set their measure to $(0, \{\})$. We also define the following function ρ in order to recursively compute φ:

$$\begin{aligned}
\rho(+,\ (a_1,b_1),\ (a_2,b_2)) &= (a_1 + a_2,\ b_1 \cup b_2),\\
\rho(\times,\ (a_1,b_1),\ (a_2,b_2)) &= (a_1 + a_2 + 1,\ b_1 \cup b_2) \text{ when } f_1 \text{ or } f_2 \text{ is not a power of } x,\\
\rho(\times,\ (0,b_1),\ (0,b_2)) &= \big(0,\ b_1 \cup b_2 \cup \{(i+j,\sigma)\}\big) \text{ when } f_1 = x^i,\ f_2 = x^j,\\
&\quad\ \text{and the scheme used for } x^{i+j} \text{ is the } \sigma\text{th one.}
\end{aligned}$$

  Notice that, in the last case, we only have multiplications between powers of x, so the first component is always 0.
- Finally, we need to provide a total order on $T = \mathbb{N} \times \mathcal{P}(\mathbb{N}^2)$, which has to be consistent with the total number of multiplications. We will say that $(a_1, b_1) <_T (a_2, b_2)$ when:

[7] Each coefficient a_i can only appear once in an evaluation scheme for $\sum_{i\in I} a_{i+k}\, x^i$.

Table 6.6: Distribution of the evaluation schemes for a degree-7 polynomial and a degree-8 polynomial according to the number of multiplications.

multiplications	number of schemes
7	1
8	585
9	28545
10	724188
11	12066156
12	142484691
13	1210062345
14	7184777823
15	30029531607
16	89724191454
17	195479643387
18	318178508025
19	395160692850
20	379752728565
21	284251875780
22	165437390460
23	74023293645
24	24939084240
25	6146417970
26	1066285080
27	121943745
28	7702695

multiplications	number of schemes
8	1
9	1572
10	116280
11	4245072
12	100901301
13	1737792846
14	22606907520
15	224473674759
16	1679317135200
17	9419838853410
18	39810790839105
19	128418624743772
20	321614460141702
21	636563506187631
22	1012043941159125
23	1310128238636700
24	1395753710386005
25	1233103860578010
26	907475440807200
27	556919177973840
28	284254828903080
29	119811812319675
30	41193779294115
31	11346619224150
32	2444441312880
33	399502303200
34	47528694675
35	3828239415
36	172297125

* $a_1 + |b_1| < a_2 + |b_2|$ (there are fewer multiplications on the left-hand side),
* $a_1 + |b_1| = a_2 + |b_2|$ and $a_1 < a_2$ (the number of multiplications is the same on both sides, but the left-hand side has fewer multiplications that are not[8] between two powers of x),
* or $b_1 < b_2$, that is, assuming $b_i = \{(n_i^{(k)}, \sigma_i^{(k)})\}$ sorted according to the lexicographical order $<_{\text{lex}}$, there exists k_0 such that $(n_1^{(k_0)}, \sigma_1^{(k_0)}) <_{\text{lex}} (n_2^{(k_0)}, \sigma_2^{(k_0)})$ and for all $k < k_0$, $(n_1^{(k)}, \sigma_1^{(k)}) =_{\text{lex}} (n_2^{(k)}, \sigma_2^{(k)})$.

Because of the complexity in the choice for type T, the analysis will be more costly than the one in the previous section. Moreover, we get a more refined result than what we were aiming for, so that we need to perform a post-treatment to actually deduce the number of multiplications from the different $t \in T$. Table 6.6 shows the distribution of the evaluation schemes for a degree-7 and a degree-8 polynomial according to the number

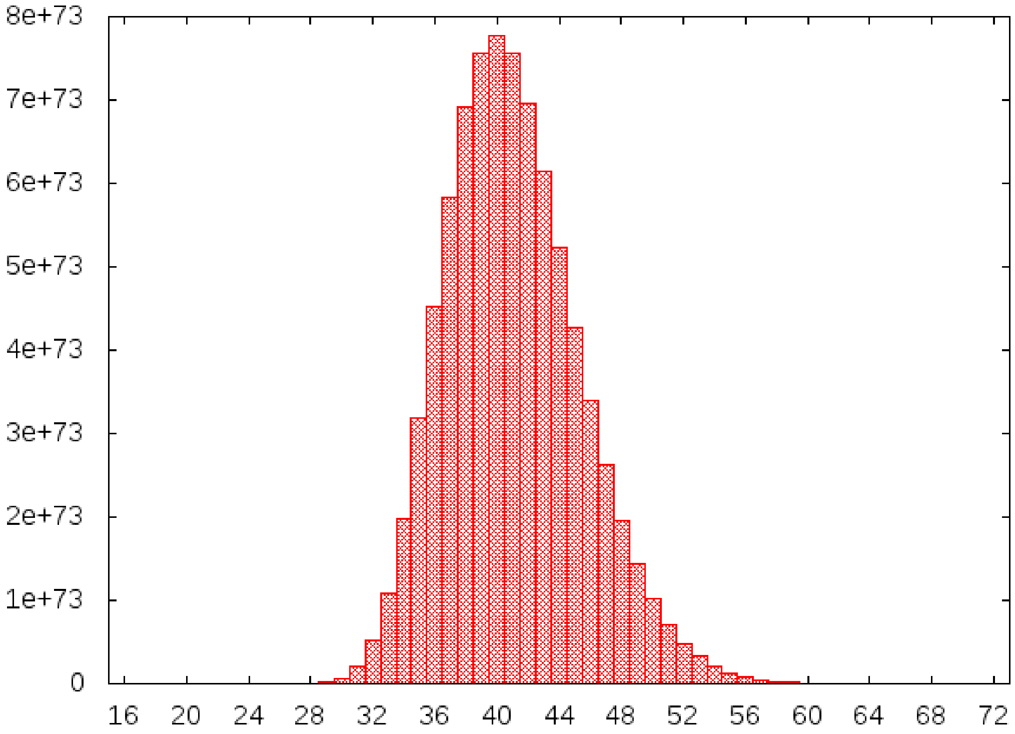
[8] We choose to give priority to schemes with more multiplications between powers of x since they are more likely to bring more common subexpressions in the sequel.

of multiplications. This number can range from n, which is known to be optimal [Pan66] and uniquely achieved [Bor71] by Horner's rule, to $\frac{n(n+1)}{2}$, which corresponds to naive schemes where we compute $a_i \cdot x, (a_i \cdot x) \cdot x, (a_i \cdot x^2) \cdot x, \ldots, (a_i \cdot x^{i-1}) \cdot x$ for all $1 \leq i \leq n$. The computation takes about 86 seconds for $n = 7$ and 1.5 hours for $n = 8$.

6.4.3 Number of evaluation schemes for polynomials with respect to the latency

We have also tested Algorithm `CountPerMeasure` on univariate polynomials using latency as the measure. Figure 6.4 illustrates how the evaluation schemes for a degree-18 polynomial split up with respect to the latency on unbounded parallelism, assuming that addition and multiplication cost 1 and 3 cycles, respectively. This figure was generated from the results obtained after about 69 hours of computation. The latency ranges from 16 cycles, which is 1 cycle less than the classical Estrin scheme, to 72 which is the latency for Horner's rule. The distribution looks like a Gaussian, while slightly more concentrated on the left, and it is actually centered around 40 cycles which is a bit less than $44 = \frac{72+16}{2}$.

Figure 6.4: Distribution of the evaluation schemes for $p(x)$ with $\deg p = 18$ according to the latency on unbounded parallelism when $\mathcal{C}_+ = 1$ and $\mathcal{C}_\times = 3$.



What interests us here more particularly in Figure 6.4 is the part near the optimal latency, shown in Table 6.7. As one can see, the number of nearly optimal schemes is quite small in comparison to the total number of schemes. Nevertheless, even the number of schemes with the optimal latency of 16 cycles is too large ($> 10^{24}$) to think about exhaustive generation. On the other hand, having so many optimal schemes lets us hope that, when we aim at a fast and accurate-enough scheme, restricting the search to optimal (or nearly optimal) schemes should be enough. This is actually the point of view we embrace in the software CGPE [MR11].

Table 6.7: Number of evaluation schemes for $p(x)$ with $\deg p = 18$ with a latency at most 20 on unbounded parallelism when $\mathcal{C}_+ = 1$ and $\mathcal{C}_\times = 3$.

latency ℓ	number of schemes n_ℓ	$\log_{10} n_\ell$
16	8358152077260267744786057	24.92
17	1472070576061528900246619602065309	33.17
18	22240147119728481951968575944076920440734990	43.35
19	768600347405960200312345740633767165234208387412527	50.89
20	5782427313764344437576581286875140118439932677073357566	54.76

6.4.4 Counting only nearly optimal schemes

Deducing the results shown in Table 6.7 via Algorithm `CountPerMeasure` is quite costly since we actually compute all the distribution of schemes according to the latency, as presented in Figure 6.4. Algorithm 6.3 allows us to focus only on the part where nearly optimal schemes lies, so that we can deduce Table 6.7 directly. One can see that this algorithm is actually a mix between `CountPerMeasure` and `GenerateWithHint` introduced in the previous chapter.

Theorem 6.5. *Provided that for all evaluation scheme $s = (\diamond, s_1, s_2)$ where s_1 and s_2 are schemes for f_1 and f_2, respectively, and for all $b \in T$, we have*

$$\varphi(s) \leq_T b \;\Rightarrow \varphi(s_1) \leq_T b' \;\wedge\; \varphi(s_2) \leq_T b'$$

where $b' := update((\diamond, f_1, f_2),\ b)$, Algorithm 6.3 is correct.

Proof. The proof can easily be derived from the proofs of correctness for `GenerateWithHint`, since what `CountWithHint` does is to keep the cardinalities instead of the sets manipulated within `GenerateWithHint`.

We still proceed by complete induction on $f \in \mathcal{F}$, and the case where f admits no decomposition is correct since the algorithm returns $r(\varphi(f)) = 1$ if the measure for the sole scheme of f is less than or equal to B, and nothing (that is, 0 scheme) otherwise.

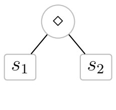
Suppose now that f is not a variable and that for all $g < f$, `CountWithHint`(g, B') is correct and gives us the distribution with respect to the measure φ of the evaluation schemes $s \in \mathcal{S}(g)$ such that $\varphi(s) \leq_T B'$.

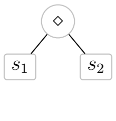
As we have seen in Theorem 5.2, each scheme $s \in \mathcal{S}(f)$ such that $\varphi(s) \leq_T B$ is encountered exactly once during Algorithm 5.2. What we need here is therefore to count the exact number of schemes involved in each case:

Algorithm 6.3: `CountWithHint`

```
Input      : $f \in \mathcal{F}$ and a bound $B \in T$.
Parameter: A recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to (T, <_T)$, and a
           function update: $\mathcal{D}(\mathcal{F}) \times T \to T$ to adjust the bound before the
           recursive calls.
Output     : The distribution according to $\varphi$ of all the evaluation schemes $s$ for $f$
           such that $\varphi(s) \leq_T B$.

$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ and $\varphi(f) \leq_T B$ then $r(\varphi(f)) \leftarrow 1$
else
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $B' \leftarrow$ update($(\diamond, f_1, f_2)$, $B$)
        $r_1 \leftarrow$ CountWithHint($f_1$, $B'$)
        if $f_1 = f_2$ then
            for $\{t_1, t_2\} \in \mathcal{P}_2(Dom(r_1))$ do
                $t \leftarrow \rho(\diamond, t_1, t_2)$
                if $t \leq_T B$ then
                    if $t_1 = t_2$ then $r(t) \leftarrow r(t) + \frac{r_1(t_1) \cdot (r_1(t_1) + 1)}{2}$
                    else $r(t) \leftarrow r(t) + r_1(t_1) \cdot r_1(t_2)$
        else
            $r_2 \leftarrow$ CountWithHint($f_2$, $B'$)
            for $(t_1, t_2) \in Dom(r_1) \times Dom(r_2)$ do
                $t \leftarrow \rho(\diamond, t_1, t_2)$
                if $t \leq_T B$ then $r(t) \leftarrow r(t) + r_1(t_1) \cdot r_2(t_2)$
return $r$
```

- When $f_1 <_{\mathcal{F}} f_2$, the schemes involved are

$\Big\{ s \in \mathcal{S}(f),\ s =$  with $(s_1,\ s_2) \in \mathcal{S}(f_1) \times \mathcal{S}(f_2)$ and $\varphi(s) \leq_T B \Big\}$

$= \displaystyle\bigsqcup_{\substack{t_1 \in Dom(r_1)\\ t_2 \in Dom(r_2)}} \Big\{ s \in \mathcal{S}(f),\ s =$  with $(s_1,\ s_2) \in \mathcal{S}(f_1)|_{t_1} \times \mathcal{S}(f_2)|_{t_2}$ and $\varphi(s) \leq_T B \Big\}$

and each inner set of schemes contains either $r_1(t_1) \cdot r_2(t_2)$ schemes (case when $\rho(\diamond, t_1, t_2) =: t \leq_T B$), or is empty.

- When $f_1 =_{\mathcal{F}} f_2$, we have to take care of avoiding redundancy. This time, the set of schemes involved is, like in the proof of Theorem 6.4,

$$\left\{s \in \mathcal{S}(f),\ s = \begin{array}{ccc} & \diamond & \\ \diagup & & \diagdown \\ \boxed{s_1} & & \boxed{s_2} \end{array} \text{ with } \{s_1,\ s_2\} \in \mathcal{P}_2(\mathcal{S}(f_1)) \text{ and } \varphi(s) \leq_T B\right\}$$

$$= \bigsqcup_{t_1 \in Dom(r_1)} \left\{s \in \mathcal{S}(f),\ s = \begin{array}{ccc} & \diamond & \\ \diagup & & \diagdown \\ \boxed{s_1} & & \boxed{s_2} \end{array} \text{ with } \{s_1,\ s_2\} \in \mathcal{P}_2(\mathcal{S}(f_1)|_{t_1}) \text{ and } \varphi(s) \leq_T B\right\}$$

$$\bigsqcup$$

$$\bigsqcup_{\substack{\{t_1, t_2\} \in \mathcal{P}_2(Dom(r_1)) \\ t_1 <_T t_2}} \left\{s \in \mathcal{S}(f),\ s = \begin{array}{ccc} & \diamond & \\ \diagup & & \diagdown \\ \boxed{s_1} & & \boxed{s_2} \end{array} \text{ with } (s_1,\ s_2) \in \mathcal{S}(f_1)|_{t_1} \times \mathcal{S}(f_1)|_{t_2} \text{ and } \varphi(s) \leq_T B\right\}$$

So, if $t_1 =_T t_2$ and $\rho(\diamond, t_1, t_2) \leq_T B$, we get $\frac{r_1(t_1)\cdot(r_1(t_1)+1)}{2}$ schemes by Property 6.1. Otherwise, we have $t_1 <_T t_2$ and the corresponding number of schemes is $r_1(t_1) \cdot r_1(t_2)$.

In all cases, Algorithm 6.3 actually adds in $r(t)$ the correct number of schemes. Hence it is correct.

□

Implementation results

We have implemented `CountWithHint` and tested it on univariate polynomials with the latency as a measure in order to generate Table 6.7 again. The function in use to update the bound before recursive calls was defined by:

$$\begin{array}{rccc} \texttt{update}: & \mathcal{D}(\mathcal{F}) \times T & \to & T \\ & (\diamond, f_1, f_2),\ b & \mapsto & \max\{b - \mathcal{C}_\diamond,\ 0\}. \end{array}$$

The computation time was around 8 hours, which means that the speed-up achieved with this approach is approximately 8.6.

Moreover, suppose that we only want the number of optimal schemes with respect to the latency. What we can do is to initialize B with 0, run `CountWithHint`, see if we get a positive number of evaluation schemes, and start again after increasing B by 1 if it is not the case. At the end of this process, we will obtain the minimal latency, along with the number of schemes achieving it. Doing this for a degree-18 polynomial allows us to find the first row of Table 6.7 in about 3 hours, which is 23 times faster than computing the complete distribution of schemes according to the latency.

Chapter 7

Optimization

We have seen so far how to generate, for a given arithmetic expression f, all its evaluation schemes and how to count them. For practical purposes, we are more concerned with finding, among all the evaluation schemes, some relevant subset which contains only optimal or nearly optimal schemes according to a given measure. The first section deals with a direct adaptation of the first generation algorithm mentioned in Section 5.3 in order to tackle optimization issues. Unfortunately, this simple approach fails to find the optimal value for some of the measures we are interested in, like the minimization of the number of multiplications. Therefore, we will present in a second section several other algorithms that handle all the recursively computable measures, and compare them. Finally, a third section will discuss multicriteria optimization. Furthermore, we will address the issue of finding a good trade-off between two measures and see a couple of situations where we can quickly find relevant information about the possible trade-offs.

7.1 Adapting the generation algorithm for optimization

7.1.1 Optimizing the latency on unbounded parallelism

We can naively optimize the latency on unbounded parallelism[1] by first generating all the schemes and then checking their latencies to deduce its minimum feasible value, but this is absolutely not efficient. In fact, minimum latency is a property attached to an arithmetic expression, and the idea is therefore to abstract the set of evaluation schemes with the best latency achieved among them. Hence, we can manipulate only latencies and forget about the underlying schemes, as shown in Algorithm 7.1.

Theorem 7.1. *Algorithm 7.1 is correct.*

Proof. If the arithmetic expression f in input is a variable, then no computation is needed and Algorithm 7.1 correctly outputs a latency of 0 cycle.

Otherwise, the minimal latency for f is by definition the minimal latency $\mathcal{L}(s)$ among all the evaluation schemes s for f:

[1] See Section 5.4.2.

Algorithm 7.1: `MinLat`

```
Input     : $f \in \mathcal{F}$.
Parameter: The costs $\mathcal{C}_+$ and $\mathcal{C}_\times$ for operators + and × respectively.
Output    : the minimum feasible latency $r$ for $f$.
1  $\ell \leftarrow$ decompose($f$)
2  if $\ell = \emptyset$ then $r \leftarrow 0$
3  else
4      $r \leftarrow \infty$
5      foreach $(\diamond, f_1, f_2) \in \ell$ do
6          $r_1 \leftarrow$ MinLat($f_1$)
7          if $f_1 = f_2$ then $r \leftarrow \min\{r,\ \mathcal{C}_\diamond + r_1\}$
8          else
9              $r_2 \leftarrow$ MinLat($f_2$)
10             $r \leftarrow \min\{r,\ \mathcal{C}_\diamond + \max\{r_1, r_2\}\}$
11 return $r$
```

$$\min_{s \in \mathcal{S}(f)} \mathcal{L}(s) = \min_{(\diamond, f_1, f_2) \in \mathcal{D}(f)} \ \min_{\substack{s_1 \in \mathcal{S}(f_1) \\ s_2 \in \mathcal{S}(f_2)}} \mathcal{L}\left(\begin{array}{ccc} & \diamond & \\ \boxed{s_1} & & \boxed{s_2} \end{array}\right). \tag{7.1}$$

Note that we do not bother to avoid redundancy in the inner minimum on the right hand side, since it does not change the value for this minimum. Hence we can handle both cases from the test at line 7 in one shot.

Moreover, recall that latency is recursively computable as we have seen in Section 5.4.2, and that we have:

$$\mathcal{L}\left(\begin{array}{ccc} & \diamond & \\ \boxed{s_1} & & \boxed{s_2} \end{array}\right) = \mathcal{C}_\diamond + \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\}. \tag{7.2}$$

In addition:

- $\max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\} \geq \mathcal{L}(s_i) \geq \min_{s' \in \mathcal{S}(f_i)} \mathcal{L}(s')$ for $i \in \{1, 2\}$,
  so for all $s_1 \in \mathcal{S}(f_1),\ s_2 \in \mathcal{S}(f_2)$ we have

  $$\max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\} \geq \max\{\min_{s_1 \in \mathcal{S}(f_1)} \mathcal{L}(s_1),\ \min_{s_2 \in \mathcal{S}(f_2)} \mathcal{L}(s_2)\}$$

  and we can take the minimum over $(s_1, s_2) \in \mathcal{S}(f_1) \times \mathcal{S}(f_2)$ to deduce that

  $$\min_{\substack{s_1 \in \mathcal{S}(f_1) \\ s_2 \in \mathcal{S}(f_2)}} \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\} \geq \max\{\min_{s_1 \in \mathcal{S}(f_1)} \mathcal{L}(s_1),\ \min_{s_2 \in \mathcal{S}(f_2)} \mathcal{L}(s_2)\}.$$

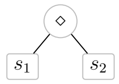
- The left-hand side of this last inequality is also not larger than the right-hand side, as we can see by taking for s_i some scheme minimizing $\mathcal{L}(s_i)$ for $i \in \{1, 2\}$.

Therefore, the following equality holds:

$$\min_{\substack{s_1 \in \mathcal{S}(f_1)\\ s_2 \in \mathcal{S}(f_2)}} \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\} = \max\{\min_{s_1\in\mathcal{S}(f_1)} \mathcal{L}(s_1),\ \min_{s_2\in\mathcal{S}(f_2)} \mathcal{L}(s_2)\}. \tag{7.3}$$

Putting Equations (7.1), (7.2), and (7.3) together, we deduce that:

$$\min_{s\in\mathcal{S}(f)} \mathcal{L}(s) = \min_{(\diamond,f_1,f_2)\in\mathcal{D}(f)} \min_{\substack{s_1 \in \mathcal{S}(f_1)\\ s_2 \in \mathcal{S}(f_2)}} \mathcal{L}\Bigl($$



$$\Bigr)$$

$$\begin{aligned}
&= \min_{(\diamond,f_1,f_2)\in\mathcal{D}(f)} \min_{\substack{s_1 \in \mathcal{S}(f_1)\\ s_2 \in \mathcal{S}(f_2)}} \bigl\{\mathcal{C}_\diamond + \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\}\bigr\}\\
&= \min_{(\diamond,f_1,f_2)\in\mathcal{D}(f)} \bigl\{\mathcal{C}_\diamond + \min_{\substack{s_1 \in \mathcal{S}(f_1)\\ s_2 \in \mathcal{S}(f_2)}} \max\{\mathcal{L}(s_1),\ \mathcal{L}(s_2)\}\bigr\}\\
&= \min_{(\diamond,f_1,f_2)\in\mathcal{D}(f)} \bigl\{\mathcal{C}_\diamond + \max\{\min_{s_1\in\mathcal{S}(f_1)} \mathcal{L}(s_1),\ \min_{s_2\in\mathcal{S}(f_2)} \mathcal{L}(s_2)\bigr\}.
\end{aligned}$$

By induction, we know that $r_1 = \min_{s_1\in\mathcal{S}(f_1)} \mathcal{L}(s_1)$ and $r_2 = \min_{s_2\in\mathcal{S}(f_2)} \mathcal{L}(s_2)$, and Algorithm 7.1 correctly computes $r = \min_{(\diamond,f_1,f_2)\in\mathcal{D}(f)} \bigl\{\mathcal{C}_\diamond + \max\{r_1, r_2\}\bigr\} = \min_{s\in\mathcal{S}(f)} \mathcal{L}(s)$. □

7.1.2 Generalization to recursively computable measures

Algorithm 7.1 can actually be generalized so that we obtain an optimization algorithm that works for different measures. As in Section 6.4, we ask the user to provide as input a measure φ satisfying the recursive computability hypothesis. Thus, φ is given by its values on atoms and a function ρ for recursive computation. With this input, we can still go through all the decompositions for f, compute the corresponding values, and take the minimum, as we did in Algorithm 7.1. What results from this generalization is Algorithm 7.2.

Theorem 7.2. *If $\rho(\diamond, x, y)$ is non-decreasing with respect to x and with respect to y for all $\diamond$ then Algorithm 7.2 is correct.*

Proof. The case of f being a variable is obviously handled correctly. What remains is to generalize the reasoning in the proof of Theorem 7.1 when f is non-trivial.

The key points were Equations (7.1) to (7.3). The first one (along with the remark about the redundancy we did not take care of) still holds when we replace $\mathcal{L}$ with φ. Equation (7.2) actually illustrates the fact that the function $(\diamond, x, y) \mapsto \mathcal{C}_\diamond + \max\{x, y\}$ is the appropriate function ρ in order to compute the latency recursively. In the present case, it is automatically deduced from the definition of ρ (which is a consequence of the recursive computability for φ). As for the third equation, we need to generalize it, and this is where the non-decreasing hypothesis will play an important role.

Algorithm 7.2: `Optimizer`

```
Input     : $f \in \mathcal{F}$.
Parameter : a recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to (T, <_T)$ defined by its
            values on variables and by $\rho : \{+, \times\} \times T \times T \to T$.
Output    : the optimal value for $f$ according to $\varphi$.

$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then $r \leftarrow \varphi(f)$
else
    $r \leftarrow \infty_{<_T}$
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $r_1 \leftarrow$ optim($f_1$)
        if $f_1 = f_2$ then $r \leftarrow \min_{<_T} \big\{r,\ \rho\big(\diamond, r_1, r_1\big)\big\}$
        else
            $r_2 \leftarrow$ optim($f_2$)
            $r \leftarrow \min_{<_T} \big\{r,\ \rho(\diamond, r_1, r_2)\big\}$
return $r$
```

What we want to prove now is the following equation:

$$\min_{\substack{s_1 \in \mathcal{S}(f_1) \\ s_2 \in \mathcal{S}(f_2)}} \rho\big(\diamond,\ \varphi(s_1),\ \varphi(s_2)\big) = \rho\big(\diamond,\ \min_{s_1 \in \mathcal{S}(f_1)} \varphi(s_1),\ \min_{s_2 \in \mathcal{S}(f_2)} \varphi(s_2)\big). \tag{7.4}$$

Since ρ is non-decreasing with respect to its second and third variables, we have for all $(s_1, s_2) \in \mathcal{S}(f_1) \times \mathcal{S}(f_2)$ that

$$\rho\big(\diamond,\ \varphi(s_1),\ \varphi(s_2)\big) \geq \rho\big(\diamond,\ \min_{s_1 \in \mathcal{S}(f_1)} \varphi(s_1),\ \min_{s_2 \in \mathcal{S}(f_2)} \varphi(s_2)\big)$$

and, taking the minimum over $(s_1, s_2) \in \mathcal{S}(f_1) \times \mathcal{S}(f_2)$, we deduce that the left-hand side in Equation (7.4) is greater than or equal to the right-hand side. The other inequality can be obtained by considering schemes s_i such that $\varphi(s_i) = \min_{s_i \in \mathcal{S}(f_i)} \varphi(s_i)$, for $i \in \{1, 2\}$, as we did before.

We can then conclude as we did in the proof of Theorem 7.1 by saying that $r_i = \min_{s_i \in \mathcal{S}(f_i)} \varphi(s_i)$ for $i \in \{1, 2\}$ by induction hypothesis, so that the computed value $r = \min_{(\diamond, f_1, f_2) \in \mathcal{D}(f)} \rho(\diamond,\ r_1,\ r_2)$ is actually equal to the minimal value of φ over all the schemes for f by Equations 7.2 and 7.4. □

7.1.3 Some remarks about this approach and its limitation

Let us comment on some aspects of the algorithm introduced in the previous section:

1. Similarly to Algorithm `Count` from Section 6.1, the cost for `Optimizer` is bounded by the product of the number of recursive calls, the maximum number of iterations

in the `foreach` loop, and the cost for each step in this loop. Moreover, elements of $\mathcal{F}$ and T are often simple enough so that the last cost is constant (think of the example with latency in Section 7.1.1).

2. Instead of computing the minimum according to $<_T$ in the `foreach` loop, one can also keep one or several decompositions that achieve this minimum. By doing so, it becomes straightforward to rebuild one optimal scheme afterwards.

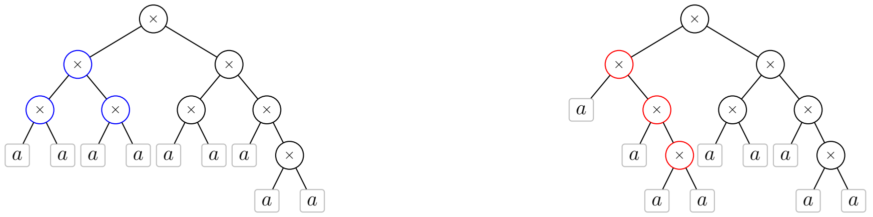
Comparison with `GenerateWithHint` and `CountWithHint`

As already mentioned, we can deduce information about the optimality according to a given measure by using either `GenerateWithHint` or `CountWithHint`. However, whenever one wants to get only the optimal value, along with a few schemes achieving it, `Optimizer` is far more efficient. For instance, if we come back to the example of the minimum latency for a univariate polynomial with $\mathcal{C}_+ = 1$ and $\mathcal{C}_\times = 3$, we have seen in Table 6.7 that the optimal latency is 16 and is achieved by $\approx 8.36 \times 10^{24}$ schemes. Getting this results took about 3 hours with `CountWithHint`, and generating all these schemes with `GenerateWithHint` is totally out of reach. Yet, using `Optimizer`, we can get one optimal scheme in only 22 minutes. This efficiency is explained by the two following reasons:

- First, `Optimizer` manipulates only decompositions and elements in T, while algorithms `CountWithHint` and `GenerateWithHint` have also to deal with huge integers and sets of schemes, respectively.

- Second, `Optimizer` focuses on *recursively optimal* schemes, that is, optimal schemes that are only made of optimal subschemes. The existence of such recursively optimal schemes is implied by the fact that ρ is non-decreasing, which tells us that improving a subscheme will never give a worse final result. However, not all the optimal schemes are *recursively optimal*, as illustrated by Figure 7.1. Therefore, contrary to `CountWithHint` and `GenerateWithHint` that take all the optimal schemes into account, `Optimizer` is not suitable for an exhaustive study of the optimal schemes.

Figure 7.1: Evaluation schemes with a minimal depth for a^9 can be formed with a non-optimal evaluation scheme for a^4.



(a) The evaluation scheme for a^4 (part in blue) has depth 2.

(b) The evaluation scheme for a^4 (part in red) has depth 3, which is not optimal.

Limitation of this approach: the case of the number of multiplications

Minimizing the number of multiplications needed in order to evaluate an arithmetic expression like a^n is a difficult problem, see the discussion about conjecture failures in [Knu98, page 477]. In our context, this case provides an example of optimal scheme $s = (\diamond, \{s_1, s_2\})$ where neither s_1 nor s_2 is optimal. Indeed, consider the following optimal[2] scheme for a^{29}:

$$\begin{array}{llll} a^2 = a \times a & a^3 = a \times a^2 & a^5 = a^2 \times a^3 & a^6 = a \times a^5 \\ a^{12} = a^6 \times a^6 & a^{17} = a^5 \times a^{12} & a^{29} = a^{12} \times a^{17} & \end{array} \tag{7.5}$$

At the end of this sequence, we multiply a^{12} obtained after 5 operations with a^{17} obtained after 6 operations. But optimal evaluation schemes for a^{12} and a^{17} have only 4 and 5 operations, respectively (refer to [Knu98, Figure 15 in §4.6.3] or Section 7.2.1).

Recall that we can measure the number of multiplications for a given scheme using $\varphi : \mathcal{S}(\mathcal{F}) \to \mathcal{P}(\mathcal{S}(\mathcal{F}))$ defined in Section 5.4.2. This is a recursively computable measure, but the example mentioned above points out that the corresponding function ρ is not non-decreasing. Indeed, we may have used the following scheme in order to evaluate a^{12}:

$$a^2 = a \times a \qquad a^4 = a^2 \times a^2 \qquad a^8 = a^4 \times a^4 \qquad a^{12} = a^4 \times a^8.$$

This scheme s_1' is better than the scheme s_1 that can be extracted from Equation (7.5) since there is one less multiplication. However, if we try to evaluate a^{29} using s_1' in addition to the scheme s_2 for a^{17} that we deduce from Equation (7.5), we will have fewer common subexpressions (a^{12} is computed in two different ways), so that $\rho(\times, \varphi(s_1'), \varphi(s_2))$ will contain more schemes than $\rho(\times, \varphi(s_1), \varphi(s_2))$.

Nevertheless, it is still possible to use `Optimizer` so as to minimize *heuristically* the number of multiplications required to evaluate a^n. Namely, we can run the algorithm while forgetting about the hypothesis on ρ, and hope that the final result will not be too far from the optimal one. We have tested it and compared the result with the table given in the On-Line Encyclopedia of Integer Sequences[3]. For $1 \leq n \leq 10001$, our algorithm answers in 2 minutes and 55 seconds, and gives the optimal number of multiplications, along with an optimal scheme, for $\approx 87.3\%$ of the inputs n. The first values of n where our algorithm fails to find an optimal scheme are $n = 77, 154, 229, 233, 294, \ldots$. In this case, we obtain each time a scheme with only 1 extra multiplication compared to the optimal. Therefore, despite the lack of correctness, `Optimizer` still produces a reasonable result in this case. Furthermore, it should be noted that, among the values of n for which `Optimizer` is not correct, we find several cases that are known to be difficult ones. For instance, Knuth's power tree method fails for $n = 77, 154, 233, \ldots$ [Knu98, page 464].

7.2 Global optimization

In order to tackle the limitation of the algorithm `Optimizer` introduced in the previous section, we propose three other algorithms:

[2] $n = 29$ is the first number such that evaluating a^n requires at least 7 multiplications, as noticed in [Knu98, Figure 15 in §4.6.3].

[3] `http://oeis.org/A003313/b003313.txt`

1. `OptimizerSet`, which is a generalization of `Optimizer` in order to optimize a criterion for a set of arithmetic expressions rather than for a single expression;

2. `GlobalOptimizer`, which is the counterpart of `CountPerMeasure` for optimization;

3. `GlobalOptimizerWithHint`, which is the counterpart of `GenerateWithHint` and `CountWithHint`.

In practice, the third algorithm performs better than the other two. Yet the first algorithm may be useful when one wants to optimize the evaluation of two quantities in the same time. This may happen for instance when using a rational function instead of a polynomial to approximate a mathematical function. As for the second algorithm, it is presented for the sake of completeness.

7.2.1 Detour via the optimization of sets of expressions

So far, each time one of our algorithms has to deal with a decomposition $(\diamond, \{f_1, f_2\})$, it performs two recursive calls on f_1 and f_2. By doing so, it becomes harder to handle properly the common subexpressions in f_1 and f_2. One idea, suggested in [Knu69, §4.6.3, Problem 32] for the question of minimizing the number of multiplications for evaluating a^n, consists in generalizing the optimization problem for $f \in \mathcal{F}$ to sets of elements in $\mathcal{F}$. Thus, instead of the two recursive calls mentioned above, one can make a recursive call on the set $\{f_1, f_2\}$.

Actually, [DLS81] answers [Knu69, §4.6.3, Problem 32] by showing that the generalized problem is NP-complete. Nevertheless, we think that this approach deserves investigation. Indeed, we do not know the complexity of the initial problem, and, as already mentioned, there are situations where we really want to deal with several expressions at the same time.

In order to embrace this approach, let us generalize for this section only some of the concepts introduced in Chapter 5:

- We still ask for a family of arithmetic expressions as in Section 5.2.3;

- A measure will be a function $\varphi : \mathcal{P}(\mathcal{S}(\mathcal{F})) \to T$, where T has a total order $<_T$;

- The problem of optimization we consider here is to find, given a set of arithmetic expressions $F = \{f_1, \ldots, f_r\}$, the quantity
$$\min_{(s_i \in \mathcal{S}(f_i))_{1 \leq i \leq r}} \varphi(\{s_1, \ldots, s_r\}) \ ;$$

- We will assume that all the measures φ will be recursively computable in the following sense: there exists $\rho : \{+, \times\} \times T \to T$ such that for all set of schemes $S = \{s_1, \ldots, s_r\}$ and for all i such that $s_i = (\diamond, s_\ell, s_r)$, we have
$$\varphi(S) = \rho\Big(\diamond,\ \varphi\big(\{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_r\} \cup \{s_\ell, s_r\}\big)\Big).$$

In particular, for a non-trivial scheme $s = (\diamond, s_\ell, s_r)$, the following holds:

$$\varphi(\{s\}) = \rho\big(\diamond, \varphi(\{s_\ell, s_r\})\big).$$

Note the difference between this and Definition 5.7, where s_ℓ and s_r were separated;

- Finally, we define the set of decompositions for $F = \{f_1, \ldots, f_r\}$ by

$$\mathcal{D}(F) = \bigcup_{1 \le i \le r} \Big\{ \big(\diamond,\ \{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_r\} \cup \{f_i^\ell, f_i^r\}\big),\ (\diamond, f_i^\ell, f_i^r) \in \mathcal{D}(f_i) \Big\}.$$

With these new definitions, we can derive from `Optimizer` a new algorithm, Algorithm 7.3, which optimizes a criterion for a given set of arithmetic expressions.

Algorithm 7.3: `OptimizerSet`

```
Input     : $F \in \mathcal{P}(\mathcal{F}),\ F \neq \emptyset$.
Parameter: $\varphi : \mathcal{P}(\mathcal{S}(\mathcal{F})) \mapsto (T, <_T)$ defined on sets of variables and by
           $\rho : \{+, \times\} \times T \to T$.
Output    : the optimal value for $F$ according to $\varphi$.
$\ell \leftarrow$ decompose($F$)
if $\ell = \emptyset$ then $r \leftarrow \varphi(F)$
else
    $r \leftarrow \infty_{<_T}$
    foreach $(\diamond, F') \in \ell$ do
        $r \leftarrow \min\{r,\ \rho(\diamond,$ OptimizerSet($F'$)$)\}$
return $r$
```

Theorem 7.3. *If $\rho(\diamond, \cdot)$ is non-decreasing for all $\diamond$ then Algorithm 7.3 is correct.*

Proof. First, notice that we can compare two elements F_1 and F_2 of $\mathcal{P}(\mathcal{F})$ by sorting each F_i ($i \in \{1, 2\}$) according to $>_\mathcal{F}$ and using the lexicographical order on the sorted sequences. This gives us a total order $\prec$ on $\mathcal{P}(\mathcal{F}) \setminus \emptyset$.

Now, we can show the theorem by complete induction. If F is a set made of trivial expressions only (and in particular when it is the singleton containing the minimum of $(\mathcal{F}, <_\mathcal{F})$, which is the minimum for order $\prec$), F is its own and only scheme, and Algorithm 7.3 is correct.

Suppose now that F contains one non-trivial expression and that the algorithm's correctness holds for all set $G \prec F$. In this case $\mathcal{D}(F)$ is not empty, so that:

$$\begin{aligned} \min_{s \in \mathcal{S}(F)} \varphi(\{s_1, \ldots, s_r\}) &= \min_{(\diamond, F') \in \mathcal{D}(F)} \min_{s' \in \mathcal{S}(F')} \rho\big(\diamond,\ \varphi(s')\big) \\ &= \min_{(\diamond, F') \in \mathcal{D}(F)} \Big\{ \rho\big(\diamond,\ \min_{s' \in \mathcal{S}(F')} \varphi(s')\big) \Big\}, \qquad (7.6) \end{aligned}$$

where $F = \{f_1, \ldots, f_r\}$, $F' = \{f'_1, \ldots, f'_k\}$, and $\mathcal{S}(F) = \mathcal{S}(f_1) \times \cdots \times \mathcal{S}(f_r)$ is the set of of evaluation schemes for F. The first equality comes from the definition of ρ, and the second equality holds since we have supposed that $\rho(\diamond, \cdot)$ is non-decreasing.

Now, it is easy to see that for all $(\diamond, F') \in \mathcal{D}(F)$, we have $F' \prec F$ (F' is obtained by replacing one element of F with 0, 1, or 2 smaller elements). Hence the induction hypothesis tells us that $\texttt{OptimizerSet}(F') = \min_{s' \in \mathcal{S}(F')} \varphi(s')$. Therefore, our algorithm actually computes the right part of Equation (7.6) and thus it is correct. □

We have tested `OptimizerSet` in order to compute the first terms of the sequence `A003313` in the On-Line Encyclopedia of Integer Sequences, whose nth term is the minimal number of multiplications for evaluating a^n. The main advantage of `OptimizerSet` is that we can use directly the number of multiplications for the measure, instead of computing sets of schemes. The corresponding function ρ is then defined by:

$$\begin{array}{rccc} \rho: & \{+, \times\} \times \mathbb{N} & \to & \mathbb{N} \\ & (+, i) & \mapsto & i \\ & (\times, i) & \mapsto & 1 + i, \end{array}$$

and, as $\rho(+, \cdot)$ and $\rho(\times, \cdot)$ are non-decreasing, the hypothesis of Theorem 7.3 is satisfied. Furthermore, the cost of `OptimizerSet` for a^n can be bounded by $2^{n+O(\log n)}$ as follows:

- As a^i is represented by i, the sets of arithmetic expressions encountered recursively will be subsets of $\{1, \ldots, n\}$. Hence, we have at most $O(2^n)$ recursive calls;
- These sets have a size in $O(n)$, and are composed of arithmetic expressions having $O(n)$ decompositions each, so they have $O(n^2)$ decompositions. Moreover, computing each corresponding F' may cost $O(n)$ (or $O(\log n)$ if we use an appropriate data structure to represent sets like red-black trees [CLRS09, §13]). So, the call to `decompose` costs $O(n^3)$;
- Finally, the cost at each step of the `foreach` loop is constant.

This bound on the cost of our approach is fair considering that the problem solved here is NP-complete. For $1 \leq n \leq 100$, we obtain a computation time of about 35 minutes, the important point being that, contrary to `Optimizer`, we get the correct value for $n = 77$.

If one really wants the optimal number of multiplications in order to evaluate a^n, we suggest to look at [Knu98, §4.6.3] and the references therein for a more practical approach, especially if n is a fixed constant. Yet, our approach instantiated for this issue performs reasonably well, and its generality enables to tackle other problems. See for instance the last paragraph of Section 8.2.2, where we use `OptimizerSet` to minimize the number of non-scalar multiplications for the simultaneous evaluation of two polynomials.

7.2.2 Algorithms `GlobalOptimizer` and `GlobalOptimizerWithHint`

Instead of considering sets of arithmetic expressions like in the previous section, we can, as in `CountPerMeasure`, return for each subexpression the set of all achievable values according to the given measure. Then, it is easy to deduce the optimal value since we

Algorithm 7.4: `GlobalOptimizer`

```
Input     : $f \in \mathcal{F}$.
Parameter: A recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to (T, <_T)$ defined by its
           values on variables and by $\rho : \{+, \times\} \times T \times T \to T$.
Output    : $r = \varphi(\mathcal{S}(f)) \in \mathcal{P}(T)$
$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then $r \leftarrow \{\varphi(f)\}$
else
    $r \leftarrow \emptyset$
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $r_1 \leftarrow$ GlobalOptimizer($f_1$)
        if $f_1 = f_2$ then
            for $t_1 \in r_1$ do $r \leftarrow r \cup \{\rho(\diamond, t_1, t_1)\}$
        else
            $r_2 \leftarrow$ GlobalOptimizer($f_2$)
            for $(t_1, t_2) \in r_1 \times r_2$ do $r \leftarrow r \cup \{\rho(\diamond, t_1, t_2)\}$
return $r$
```

just have to take the minimum among these achievable values. This approach is illustrated in Algorithm 7.4.

The correction for this algorithm can be shown similarly to the proof of Theorem 6.4. Notice that we do not need to have a non-decreasing function ρ here, since we consider its whole image at each step and only extract the optimal value after the last step. As for the cost, Algorithm `GlobalOptimizer` suffers, like `CountWithHint`, from the slowness implied by the large sets that have to be manipulated all along the algorithm. To cope with this difficulty, we can ask for an additional bound in order to limit the search space, as in `GenerateWithHint` and `CountWithHint`. This yields Algorithm 7.5.

Theorem 7.4. *Provided that for all $s = (\diamond, s_1, s_2)$ with $s_1 \in \mathcal{S}(f_1)$ and $s_2 \in \mathcal{S}(f_2)$, and for all $B \in T$, we have $\varphi(s) \leq_T B \;\Rightarrow \varphi(s_1) \leq_T B' \;\wedge\; \varphi(s_2) \leq_T B'$ with $B' :=$ update$((\diamond, f_1, f_2),\ B)$, Algorithm 7.5 is correct.*

The only difference with `CountWithHint` is that, here, we only keep trace of the achievable values which are better than the bound B instead of also counting the corresponding number of schemes. Adapting accordingly the proof of Theorem 6.5 hence yields a proof for Theorem 7.4.

In order to compare the three approaches, we have computed the first terms of the sequence `A003313`. The resulting timings are presented in Table 7.1. As already mentioned, `GlobalOptimizer` is quite slow compared to the other algorithms. Moreover, its memory consumption is so large that we are not able to exceed $n = 28$. For small values of n, `OptimizerSet` is the fastest. This can be explained by the simplicity of the measure in use. However, the number of sets of expressions to consider recursively grows quickly with respect to n, so that `GlobalOptimizerWithHint` ends up being more efficient. Note

Algorithm 7.5: `GlobalOptimizerWithHint`

```
Input     : $f \in \mathcal{F}$, and a bound $B \in T$.
Parameter: A recursively computable measure $\varphi : \mathcal{S}(\mathcal{F}) \to (T, <_T)$ defined by its
           values on variables and by $\rho : \{+, \times\} \times T \times T \to T$, and a function
           update: $\mathcal{D}(\mathcal{F}) \times T \to T$ to adjust the bound before the recursive
           calls.
Output    : $r = \{\varphi(s),\ s \in \mathcal{S}(f)$ and $\varphi(s) \leq_T B\} \in \mathcal{P}(T)$.

$\ell \leftarrow$ decompose($f$)
$r \leftarrow \emptyset$
if $\ell = \emptyset$ and $\varphi(f) \leq_T B$ then $r \leftarrow \{\varphi(f)\}$
else
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $B' \leftarrow$ update($(\diamond, f_1, f_2)$, $B$)
        $r_1 \leftarrow$ GlobalOptimizerWithHint($f_1, B'$)
        if $f_1 = f_2$ then
            for $t_1 \in r_1$ do
                $t \leftarrow \rho(\diamond, t_1, t_1)$
                if $t \leq_T B$ then $r \leftarrow r \cup \{t\}$
        else
            $r_2 \leftarrow$ GlobalOptimizerWithHint($f_2, B'$)
            for $(t_1, t_2) \in r_1 \times r_2$ do
                $t \leftarrow \rho(\diamond, t_1, t_2)$
                if $t \leq_T B$ then $r \leftarrow r \cup \{t\}$
return $r$
```

that the cost of `GlobalOptimizerWithHint` is not very smooth with respect to n. In fact, computing recursively `A003313`(n) implies that all the values `A003313`(i) for $i \leq n$ are computed, and since we use increasing bounds within `GlobalOptimizerWithHint` until we get a result, the cost for `A003313`(i) when it is a local maximum is significantly larger. Noting that $i = 79$ achieves such a local maximum thus explains the difference between the costs for $n = 60$ and $n = 80$.

Table 7.1: Timings for the computation of `A003313(n)` with our three approaches for global optimization.

n	20	28	40	60	80	100	120	140
`OptimizerSet`	**0.01s**	**0.04s**	0.54s	13.51s	**3.32m**	34m	4.58h	31.5h
`GlobalOptimizer`	1.96s	26m	–	–	–	–	–	–
`GlobalOptimizerWithHint`	0.01s	0.08s	**0.37s**	**9.45s**	3.58m	**7.31m**	**11.1m**	**2.64h**

Finally, recall that the first values for `A003313`(n) were already computed with Algorithm GenerateWithHint and presented in Table 5.1. However, the approach consisting in generating and then deducing the optimal value is more costly than the optimization algorithms mentioned in this section, as we can see by comparing[4] the timings from Tables 5.1 and 7.1. Therefore, when one wants one optimal scheme rather than to perform an exhaustive study, OptimizerSet and GlobalOptimizerWithHint are more efficient.

7.3 Multicriteria optimization

Up to now, we have discussed the optimization of a single criterion. Yet one may want to optimize more than one aspect of the evaluation. Typically, one may aim at some evaluation scheme that is both as fast as possible and very accurate. However, this is usually not possible to optimize several criteria at the same time, so that trade-offs have to be made. This section will first describe several solutions, based on the optimization algorithms already mentioned, in order to find relevant evaluation schemes in a context of multicriteria optimization. Then, we will introduce a new algorithm to tackle the issue of trade-offs.

7.3.1 How monocriterion optimization may help in a multicriteria context

It often happens that, among a set of criteria to be considered, one outclasses the others. If we come back to the example of speed and accuracy, some applications like image processing for video games ask for a very fast evaluation with a few bits of accuracy, whereas the implementation of an efficient floating-point operator with correct rounding as recommended by the IEEE 754-2008 standard requires first guarantees on the accuracy. Whenever the main motivation lies in only one criteria, we can use the algorithms already mentioned in this chapter in order to try to find satisfactory evaluation schemes. We propose three ways to achieve this:

- First, we can call GenerateWithHint in order to get optimal or nearly optimal evaluation schemes according to the main criterion. Then, an exhaustive search among the set of schemes generated allows us to find the one that fits the most with the other criteria. This was the approach embraced by CGPE, where evaluation schemes of optimal latency for polynomials are generated and then kept or discarded depending on their numerical properties [MR11]. Because of the possibly large number of optimal schemes, it may take a very long time to generate and analyze all the schemes. Setting a maximum number (typically, 50 in CGPE) of generated schemes for each subexpression helps to cope with this.

- Second, suppose we have a set of r criteria sorted in decreasing order of importance, and whose corresponding measures $\varphi_i : \mathcal{S}(\mathcal{F}) \to T_i$ for $1 \leq i \leq r$ are recursively

[4] It should be noted that the cost due to the computation of the other sequences in Table 5.1 is small compared to the overall computation time.

computable. Then we can define the new measure

$$\begin{array}{rccc} \varphi : & \mathcal{S}(\mathcal{F}) & \to & T \\ & s & \mapsto & \big(\varphi_1(s),\, \ldots,\, \varphi_r(s)\big) \end{array},$$

where $T := T_1 \times \cdots \times T_r$. It is in turn a recursively computable measure, since we can define:

$$\begin{array}{rccc} \rho : & \{+, \times\} \times T \times T & \to & T \\ & \big(\diamond,\, (a_1, \ldots, a_r),\, (b_1, \ldots, b_r)\big) & \mapsto & \big(\varphi_1(\diamond, a_1, b_1),\, \ldots,\, \varphi_r(\diamond, a_r, b_r)\big) \end{array}.$$

Then, using the lexicographical order on T, we can call `Optimizer` and eventually get a satisfactory scheme. If the first measure φ_1 satisfies the hypothesis of Theorem 7.2, that is, if ρ_1 is non-decreasing with respect to its second and third variables respectively, we will at least get an optimal scheme according to this measure. Indeed, the lexicographical order on T implies that the order on the first component prevails over the other.

Unfortunately, even though all the ρ_i are non-decreasing with respect to their second and third variables, we have no guarantee to get an optimal result in the sense of $<_{\text{lex}}$ (that is, a result where the ith component is best possible after we have optimized the $(i-1)$st first components). Think of an optimization of latency and then accuracy for $f = (\diamond, f_1, f_2)$. We may have to choose between a fast but highly inaccurate scheme for f_1, and a slightly slower but very accurate one. Then, if the fastest scheme for f_2 is slow enough, it is best to choose the accurate scheme for f_1 since we get the same speed but a greater accuracy. However, `Optimizer` will choose the faster scheme for f_1.

- Third, it is possible to cope with the non-optimality in Algorithm `Optimizer` by using `GlobalOptimizerWithHint` instead. Indeed, as soon as we have a good insight on how the behavior of φ_1 when we decompose an expression, we can derive a function `update` that we can use within `GlobalOptimizerWithHint`. Notice that information on φ_1 is enough since we can restrict ourselves to bounds B in $T = T_1 \times \cdots \times T_r$ of type $(x, \infty_2, \ldots, \infty_r)$. This technique allows us to greatly limit the search space, and thus to deduce the optimal achievable value according to $<_{\text{lex}}$, along with one or a few optimal schemes.

We have tried the last two approaches on a concrete example. Suppose we want to evaluate the polynomial

$$\begin{aligned} p(x) = {} & 2.718281852081418037414550781 25 & & + 0.0000014398247003555297851 5625 & & x \\ & + 1.3590787407010793685913085937 5 & x^2 & - 0.9050387283787131309509277343 75 & & x^3 \\ & + 1.0100360410287976264953613281 25 & x^4 & - 0.947496345266699790954589843 75 & & x^5 \\ & + 0.8318204507231712341308593 75 & x^6 & - 0.6024215556681156158447265625 & & x^7 \\ & + 0.3216943116858601570129394531 25 & x^8 & - 0.1083150384947657585144042968 75 & & x^9 \\ & + 0.0168869150802493095397949218 75 & x^{10}, & & & \end{aligned}$$

Figure 7.2: Evaluation schemes obtained by optimizing first the latency and then the accuracy.

$$((((x).((x).(x))).(((x).(x)).((x).(x)))).((((x).(x)).(((x).(a_{10})) + (a_9))) + (((x).(a_8)) + (a_7)))) + (((((x).(x)).((x).(x))).((x).(((x).(a_6)) + (a_5)))) + ((((x).((x).(x))).(((x).(a_4)) + (a_3))) + (((x).((x).(a_2))) + (((x).(a_1)) + (a_0)))))$$

(a) Scheme returned by `Optimizer`.

$$((((x).((x).(x))).(((x).(x)).((x).(x)))).(((((x).(x)).((x).(a_{10}))) + ((x).(((x).(a_9)) + (a_8)))) + (a_7))) + (((((x).((x).(x))).(((x).(x)).(((x).(a_6)) + (a_5)))) + ((((x).((x).(x))).(((x).(a_4)) + (a_3))) + (((x).((x).(a_2))) + ((x).(a_1))))) + (a_0))$$

(b) Scheme returned by `GlobalOptimizerWithHint`.

which is a good approximant of $\frac{\exp(1+x)}{1+x}$ on the interval $[0, 0.99999988079071044921875]$, using double-precision floating-point arithmetic. We use the model described in Section 5.4.2 to measure the accuracy, except that we choose to consider a 1-ulp error for + and × (instead of $\frac{1}{2}$-ulp errors) so that it covers every rounding mode of the IEEE-754 standard. The resulting schemes are shown in Figure 7.2: On the one hand, `Optimizer` gives us in 1 second the scheme of Figure 7.2(a) which has a latency of 13 cycles and an error bound of $4.45862214951341271429 \times 10^{-15}$ according to our model; On the other hand, `GlobalOptimizerWithHint` returns after 72 seconds the scheme of Figure 7.2(b) which has also a minimal latency of 13 cycles, but a slightly smaller error bound of $3.64884386153025096137 \times 10^{-15}$. Thus, as we can see on this example, `Optimizer` is able to quickly optimize the latency and give a scheme with a reasonable error bound, while `GlobalOptimizerWithHint` takes more time to optimize first the latency and then the error bound.

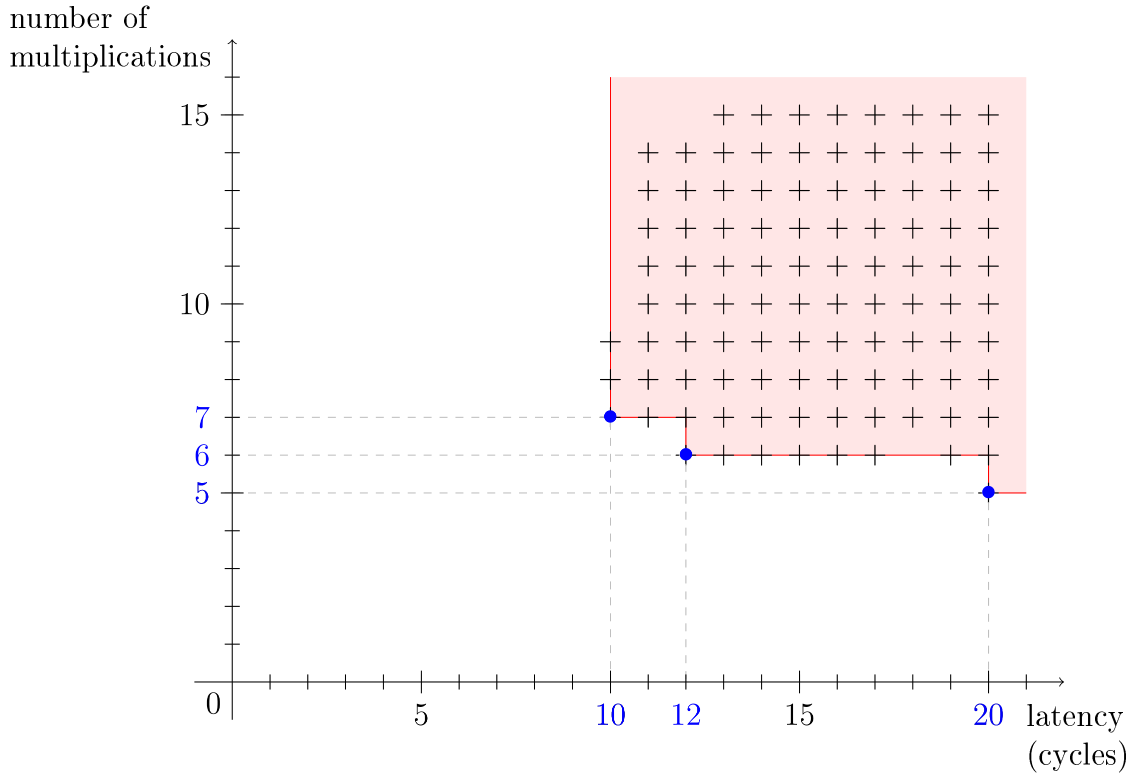
Even if we have not tried the first approach based on `GenerateWithHint`, we can claim that it would have taken much more time than the two others. Indeed, generating all the 48151536 evaluation schemes with the optimal latency of 13 already takes around 11 minutes. In addition, while limiting the number of evaluation schemes generated at each step would decrease the computation time, there is no reason that the best error bound among the remaining schemes would be as good as the one provided by `GlobalOptimizerWithHint`. In this situation, `GlobalOptimizerWithHint` should therefore be preferred over `GenerateWithHint`.

7.3.2 Search for a trade-off

Let us see now a last algorithm for this chapter, designed in order to address the issue of trade-off for a set of criteria. We will actually restrict ourselves to the case of two criteria for the sake of simplicity, and in order to avoid prohibitive computation time. However, the ideas presented here can be extended to more criteria.

Figure 7.3 illustrates how the evaluation schemes for a degree-5 polynomial split up according to their latency (assuming a cost of 1 cycle for addition and of 3 cycles for multiplication) and their number of multiplications. All the evaluation schemes, represented with crosses, lie in the red array. If we want to minimize both the latency and the number of multiplications, the more at bottom-left we are, the better it is. Therefore, the possible best choices in this case are the corners of the red area represented with blue dots, that is, schemes with a latency of 10 and 7 multiplications, or with a latency of 12 and 6 multiplications, or with a latency of 20 but only 5 multiplications.[5] All the other schemes will have either a larger latency, or a larger number of multiplications (or both) than the schemes at the blue dots.

Figure 7.3: Distribution of the evaluation schemes for a degree-5 polynomial according to their latency and number of multiplications.



Our purpose here is to design an algorithm that will compute the set of blue dots, so that one can easily decide a good choice when facing a trade-off issue. In fact, what we propose is a mix between algorithms `Optimizer` and `GlobalOptimizer`, that we will call `BiOptimizer`. Like in `GlobalOptimizer`, we will compute a set of achievable trade-offs at each step (the crosses in Figure 7.3), but then we will select only the optimal ones (the blue dots) like when we only keep the optimal value in `Optimizer`. What we compute recursively is therefore the list of optimal trade-offs for each subexpression.

Let $\preccurlyeq_{T_1 \times T_2}$ be the partial order on $T_1 \times T_2$ defined by:

$$\forall (a_1, a_2) \in T_1,\ (b_1, b_2) \in T_2,\ (a_1, a_2) \preccurlyeq_{T_1 \times T_2} (b_1, b_2) \Leftrightarrow a_1 \leq_{T_1} b_1 \text{ and } a_2 \leq_{T_2} b_2.$$

[5] In fact, Horner's rule is the only scheme that achieves this last trade-off, as we have already seen in Section 6.4.2.

The optimal trade-offs are the minimal pairs for this order, and we can store all of them in a list sorted such that the first components of pairs are decreasing with respect to $<_{T_1}$ and the second components of pairs are increasing with respect to $<_{T_2}$. In order to maintain this structure when adding a new trade-off, we will use the routine `insert` described in Algorithm 7.6. The idea is to determine the position i where the new pair x should be inserted according to its first component. Then, we have two cases illustrated in Figure 7.4: Either $\ell^{(i-1)} \preccurlyeq_{T_1 \times T_2} x$ and we do not need to keep x (see Figure 7.4(a)); Or $x_2 <_{T_2} \ell_2^{(i-1)}$ so we keep it and eventually remove elements $\ell^{(n)}$ of ℓ such that $x \preccurlyeq_{T_1 \times T_2} \ell^{(n)}$ (see Figure 7.4(b)).

Algorithm 7.6: `insert` (subroutine for `BiOptimizer`)

Input : A pair $x := (x_1, x_2) \in T_1 \times T_2$ and a list of pairs in $T_1 \times T_2$ $\ell = [\ (\ell_1^{(1)}, \ell_2^{(1)}),\ \dots,\ (\ell_1^{(k)}, \ell_2^{(k)})\ ]$ such that for all $i < j$, $\ell_1^{(i)} <_{T_1} \ell_1^{(j)}$ and $\ell_2^{(i)} >_{T_2} \ell_2^{(j)}$.

Output: A new list $\tilde{\ell} = [\ (\tilde{\ell}_1^{(1)}, \tilde{\ell}_2^{(1)}),\ \dots,\ (\tilde{\ell}_1^{(r)}, \tilde{\ell}_2^{(r)})\ ]$ such that:

- for all $i < j$, $\tilde{\ell}_1^{(i)} <_{T_1} \tilde{\ell}_1^{(j)}$ and $\tilde{\ell}_2^{(i)} >_{T_2} \tilde{\ell}_2^{(j)}$;
- for all $y \in \ell \cup \{x\}$, there exists $i \in \{1, \dots, r\}$ such that $\tilde{\ell}^{(i)} \preccurlyeq_{T_1 \times T_2} y$.

```
$i \leftarrow 1$
while $i \le k$ and $x_1 >_{T_1} \ell_1^{(i)}$ do $i \leftarrow i + 1$
if $x_2 \ge_{T_2} \ell_2^{(i-1)}$ then $\tilde{\ell} \leftarrow \ell$
else
    $j \leftarrow i$
    while $j \le k$ and $x_2 \le_{T_2} \ell_2^{(j)}$ do $j \leftarrow j + 1$
    $\tilde{\ell} \leftarrow \left[\ell^{(1)}, \dots, \ell^{(i-1)}, x, \ell^{(j)}, \dots, \ell^{(k)}\right]$
return $\tilde{\ell}$
```

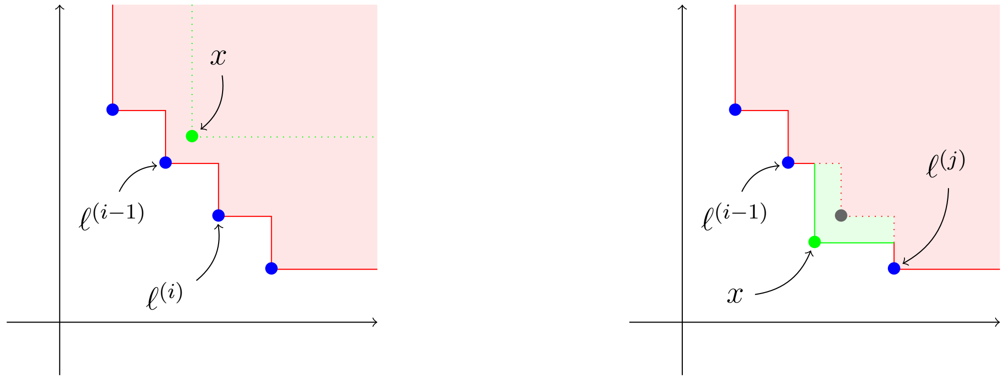
Thanks to `insert`, it is possible to compute the list of all the optimal trade-offs among a given list of achievable values. Indeed, we just need to start with an empty list and then "add" the achieved trade-offs one by one using `insert`. We can now write down the algorithm that computes the optimal trade-offs when given an arithmetic expression and two recursively computable measures. This yields Algorithm 7.7.

Theorem 7.5. *If ρ_1 and ρ_2 are non-decreasing with respect to their second and third variables then Algorithm 7.7 computes the list of the optimal trade-offs for a given arithmetic expression $f \in \mathcal{F}$ according to the measures φ_1 and φ_2.*

Proof. Let us proceed by complete induction on $f \in \mathcal{F}$. The case of f being a variable is straightforward, so let us fix a non-trivial $f \in \mathcal{F}$ and suppose that the theorem holds for any $g <_{\mathcal{F}} f$. If `BiOptimizer`(f) returns the list r, we have to prove two facts:

1. All the points in r are achieved by some scheme of $\mathcal{S}(f)$;

Figure 7.4: How the routine `insert` works.



(a) First case: there exists an i such that $\ell^{(i-1)} \preccurlyeq_{T_1\times T_2} x$ so we do not need to keep x.

(b) Second case: x is an optimal trade-off, so we insert it and remove all the $\ell^{(n)}$ such that $i-1 < n < j$ since $x \preccurlyeq_{T_1\times T_2} \ell^{(n)}$.

2. For every scheme $s \in \mathcal{S}(f)$, there exists an index n such that $r^{(n)} \preccurlyeq_{T_1\times T_2} \big(\varphi_1(s),\, \varphi_2(s)\big)$. In other words, there should exist at least one trade-off in r that is not worse than what we obtain with s for both measures.

The first fact is a simple consequence of the fact that each point added in r is achievable by construction. Consider now a scheme $s = (\diamond, \{s_1, s_2\})$ for f with $s_1 \in \mathcal{S}(f_1)$ and $s_2 \in \mathcal{S}(f_2)$. Since $f_i <_{\mathcal{F}} f$ for $i \in \{1,2\}$, the induction hypothesis tells us that there exists $r_i^{(n_i)} \in r_i$ such that $r_i^{(n_i)} \preccurlyeq_{T_1\times T_2} (\varphi_1(s_i),\, \varphi_2(s_i))$. Hence, with $(a_1, a_2) := r_1^{(n_1)}$, $(b_1, b_2) := r_2^{(n_2)}$, and $x := \big(\rho_1(\diamond, a_1, b_1),\, \rho_2(\diamond, a_2, b_2)\big)$, we deduce that

$$x_i = \rho_i(\diamond, a_i, b_i) \leq_{T_i} \rho_i(\diamond, \varphi_i(s_1), \varphi_i(s_2)) = \varphi_i(s)$$

for $i \in \{1,2\}$ since ρ_i is non-decreasing by assumption, and $a_i \leq_{T_i} \varphi_i(s_1)$ and $b_i \leq_{T_i} \varphi_i(s_2)$ by definition of $\preccurlyeq_{T_1\times T_2}$.

Therefore, $x \preccurlyeq_{T_1\times T_2} \big(\varphi_1(s),\, \varphi_2(s)\big)$. Moreover, since this x is eventually taken into account in one of the inner loops of the algorithm, and thanks to the behavior of the routine `insert`, we will always have some element $y \in r$ such that $y \preccurlyeq_{T_1\times T_2} x$ after the call to $\texttt{insert}(x, r)$. Let $r^{(n)}$ be such an element of r at the end of the *foreach* loop. By transitivity, we have $r^{(n)} \preccurlyeq_{T_1\times T_2} x \preccurlyeq_{T_1\times T_2} \big(\varphi_1(s),\, \varphi_2(s)\big)$, which proves the second fact. □

We can make a few remarks on this new algorithm:

- Let us comment on the expected cost of `BiOptimizer`. On one hand, it will be higher than when we use `Optimizer`, since we propagate lists of achievable values instead of only one optimal value. On the other hand, we can hope that the list of optimal trade-offs will be significantly smaller than the set of all the values achieved by the different evaluation schemes. In this case, `BiOptimizer` will run much faster than `GlobalOptimizer` or even than `GlobalOptimizerWithHint`.

- Like for all the previous optimization algorithms, we can always keep trace of the decompositions achieving a given value, so that we can get back one or several schemes corresponding to it.

Algorithm 7.7: `BiOptimizer`

```
Input     : $f \in \mathcal{F}$.
Parameter : Two recursively computable measures $\varphi_i : \mathcal{S}(\mathcal{F}) \to (T_i, <_{T_i})$ for
            $i \in \{1, 2\}$.
Output    : The list of all the optimal achievable trade-offs (represented by pairs
            in $T_1 \times T_2$) sorted by lexicographical order.
$\ell \leftarrow$ decompose($f$)
if $\ell = \emptyset$ then $r \leftarrow \Big[\big(\varphi_1(f), \varphi_2(f)\big)\Big]$
else
    $r \leftarrow [\,]$
    foreach $(\diamond, f_1, f_2) \in \ell$ do
        $r_1 \leftarrow$ bioptim($f_1$)
        if $f_1 = f_2$ then
            for $\{(a_1, a_2), (b_1, b_2)\} \in \mathcal{P}_2(r_1)$ do
                $x \leftarrow \big(\rho_1(\diamond, a_1, b_1), \rho_2(\diamond, a_2, b_2)\big)$
                insert($x$, $r$)
        else
            $r_2 \leftarrow$ bioptim($f_2$)
            for $(a_1, a_2), (b_1, b_2) \in r_1 \times r_2$ do
                $x \leftarrow \big(\rho_1(\diamond, a_1, b_1), \rho_2(\diamond, a_2, b_2)\big)$
                insert($x$, $r$)
return $r$
```

- The choice underlying the trade-off may be linked to variables. In practice, we handle this case by allowing the user to set her/his own optimal lists of trade-offs for the variables. See the first example of the next section for an illustration of this aspect.

- We may still have relevant results when the ρ_i's are not non-decreasing, as we will see in the second example of the next section. However, we lose the guarantee of optimality.

7.3.3 Application to polynomial evaluation

Let us see two applications of the algorithm introduced in the previous section.

Trade-off between latency and delay for special bivariate polynomials

The first example directly comes from the implementation of floating-point operators for VLIW integer processors, like the ST231 processor, in the framework of the FLIP software library (see Section 4.2.1). Recall that for some operators, the polynomial to evaluate is of the form $q(x, y) = \alpha + y \cdot p(x)$ where $p(x) = \sum_{i=0}^{n} a_i \cdot x^i$ is some univariate polynomial of degree-n [JKMR08]. In this case, the actual values for x and y have to be deduced from the encoding of the floating-point number in input. This is done with a few instructions designed by hand, and minimizing the latency for this part is tedious. However, the value of x is usually known before the value of y. This was modelled in the software CGPE by a parameter called the delay D, that represents the number of cycles we have to wait for before we can access the value of y when evaluating $q(x, y)$. A natural question is then to see the impact of this delay on the minimum latency for the evaluation. Indeed, it may be hard and time-consuming to gain even 1 cycle in the sequence of instructions for the computation of the value of y, and doing so gives no *a priori* guarantee that the overall latency will also decrease.

We propose here to use `BiOptimizer` in order to get more information about the impact of the delay D on the minimum latency. The two measures we will use are first the latency (with $\mathcal{C}_+ = 1$ and $\mathcal{C}_\times = 3$ as it stands for the ST231 processor), and a new measure for the delay defined as follows:

- the delay for x, α, and any a_i is ∞;
- the delay for y is some parameter D;
- the delay for a non-trivial expression $f = (\diamond, f_1, f_2)$ is the minimum of the delays for f_1 and f_2, meaning that we use $(\diamond, x, y) \mapsto \min\{x, y\}$ for the recursive computation of the delay.

In fact, the measure of scheme s can be thought of as being the maximum allowed delay for an evaluation of s, which will be linked to the minimum latency we try to achieve. The order we use is therefore $>$ since a greater allowed delay means less effort on the design of the sequence of instructions to get the value of y. Now, in order to compute the trade-off between the minimum latency and the maximum delay, what we do is to fix for y the following list of optimal trade-offs: $[(0, 0),\ (1, 1),\ \ldots,\ (M, M)]$, where M is some maximal bound for the delay. This tells us that y can be evaluated in i cycles when the delay is at most i, for $0 \leq i \leq M$. Now, we can use `BiOptimizer` to propagate this information and obtain the list of optimal trade-offs for $q(x, y)$, which is what we were looking for.

We have fixed $M = 12$ and run `BiOptimizer` for $q(x, y)$ with a degree for $p(x)$ ranging from 0 to 12. The corresponding result, obtained in about 7 seconds, is illustrated at Table 7.2, where each row corresponds to one degree for $p(x)$, each column to one value for the delay D, and each entry to the corresponding latency. With this table, we can conclude for instance that, for $\deg(p) = 8$, one cannot hope to decrease the latency by decreasing the delay D if D is already at most 3. On the contrary, it is worth trying to get a delay of 3 cycles instead of 4 in this case, since it allows us to save 1 cycle for the evaluation scheme.

Trade-off between latency and accuracy

As a second illustration of `BiOptimizer`, let us come back to the example mentioned at the end of Section 7.3.1, where we wanted to evaluate a degree-10 polynomial $p(x)$

Table 7.2: Minimum latency on unbounded parallelism for the evaluation of $q(x, y) = \alpha + y \cdot p(x)$ with respect to $\deg(p)$ and the delay D for y.

<table>
<tr><th>D
deg(p)</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th></tr>
<tr><td>5</td><td colspan="4">11</td><td colspan="2">12</td><td colspan="3">13</td><td colspan="2">14</td><td>15</td><td>16</td></tr>
<tr><td>6</td><td colspan="2">11</td><td>12</td><td colspan="4">13</td><td colspan="3">14</td><td colspan="2">15</td><td>16</td></tr>
<tr><td>7</td><td colspan="7">13</td><td colspan="2">14</td><td colspan="3">15</td><td>16</td></tr>
<tr><td>8</td><td colspan="4">13</td><td colspan="4">14</td><td>15</td><td colspan="3">16</td><td>17</td></tr>
<tr><td>9</td><td colspan="4">13</td><td colspan="4">14</td><td>15</td><td colspan="3">16</td><td>17</td></tr>
<tr><td>10</td><td>13</td><td colspan="6">14</td><td colspan="2">15</td><td colspan="3">16</td><td>17</td></tr>
<tr><td>11</td><td colspan="5">14</td><td colspan="4">15</td><td colspan="3">16</td><td>17</td></tr>
<tr><td>12</td><td colspan="4">14</td><td colspan="2">15</td><td colspan="4">16</td><td colspan="3">17</td></tr>
</table>

approximating $\frac{\exp(1+x)}{1+x}$ on the interval [0, 0.99999988079071044921875]. The list of trade-offs obtained using the latency and accuracy as first and second measures respectively is shown in Figure 7.5.

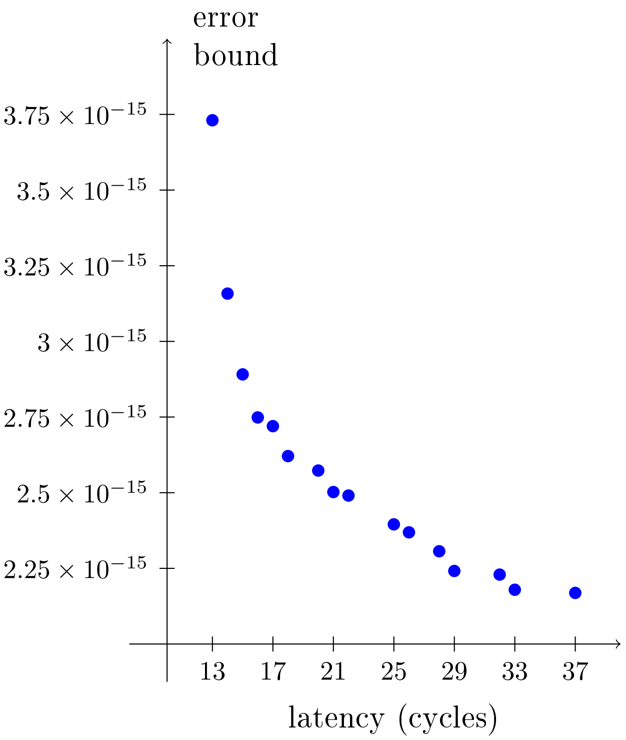
The conclusion is similar to the one from Section 7.1.3. We used here as the second measure the accuracy, whose corresponding function ρ is not non-decreasing in all generality. Therefore, one hypothesis of Theorem 7.5 is not satisfied, and we have thus no guarantee to get optimal trade-offs with `BiOptimizer`. Yet, using this algorithm yields relevant results. Indeed, for the optimal latency of 13, we find the maximum error bound $3.72775790717401075969 \times 10^{-15}$, which is only slightly worse than the best achievable bound $3.64884386153025096137 \times 10^{-15}$ pointed out by `GlobalOptimizerWithHint`. However, the computation time for this new approach is 12 times smaller (6 seconds instead of 72). Moreover, we get the behavior of the error bound (according to our model) with respect to the latency achieved. Notice also that, compared to `Optimizer` whose computation time was around 1 second, we manage to get a better accuracy for the optimal latency along with more information on the trade-off between latency and accuracy at a reasonable extra cost.

Finally, one interesting issue about the latency-accuracy trade-off lies in the behavior of the minimum error bound when one increases the acceptable latency. For sums of variables, this issue was addressed in [LMT10], where the authors conclude that relaxing slightly the time constraints for evaluating the sum helps to strongly improve accuracy. Here, we are able with our general approach to show experimentally that the same conclusion holds for some univariate polynomials. Indeed, if we accept to use a scheme of 16 cycles instead of 13 in the example of Figure 7.5, we obtain an error bound three times closer to the best one. Note that the same outcome was observed on several other univariate polynomials.

Figure 7.5: List of trade-offs between latency and accuracy for a degree-10 approximant polynomial for $\frac{\exp(1+x)}{1+x}$ on [0, 0.999999880790710449218 75].

latency	corresponding error bound
13	$3.72775790717401075969 \times 10^{-15}$
14	$3.15490087180916402536 \times 10^{-15}$
15	$2.88800877566542230613 \times 10^{-15}$
16	$2.74574380811116173023 \times 10^{-15}$
17	$2.71702017475887173616 \times 10^{-15}$
18	$2.61807325236003336294 \times 10^{-15}$
20	$2.57029342444246858722 \times 10^{-15}$
21	$2.49926275603996578948 \times 10^{-15}$
22	$2.48760573017377085851 \times 10^{-15}$
25	$2.39243078716182357609 \times 10^{-15}$
26	$2.36607232433974726910 \times 10^{-15}$
28	$2.30360836551992626475 \times 10^{-15}$
29	$2.23846786224949313284 \times 10^{-15}$
32	$2.22644280989158904673 \times 10^{-15}$
33	$2.17642785364488461850 \times 10^{-15}$
37	$2.16595520235060421263 \times 10^{-15}$

(a) Table summarizing the list of trade-offs.



(b) Graphical representation.

Chapter 8

Application examples

In this last chapter, we present two more advanced examples, where we have successfully used the material introduced in the second part of this thesis in order to derive interesting results. First, we have partially rewritten the software tool CGPE. The new design, which heavily relies on the addition of constraints within the generation of the set of schemes to be analyzed, allows us to find out fast and accurate-enough schemes for polynomial evaluation significantly faster. Second, we study the case of evaluating a polynomial at a matrix point. Such an evaluation, which appears for instance when one has to compute some function of matrices, raises the issue of minimizing the number of matrix-matrix multiplications. We will see how to address this issue with our methods, and how we were able to show that Paterson and Stockmeyer's algorithm [PS73, Algorithm B] is optimal for a degree $d \leq 15$. Further investigation allows us to categorize optimal schemes for small degrees in up to three classes, depending on the degree. Finally, we discuss the evaluation of a rational approximation for $\exp(\mathsf{A})$, and explain how CGPE can find automatically schemes similar to the one crafted by hand by Higham [Hig08, page 244].

8.1 New design for CGPE

8.1.1 The initial design for the tool and its limitations

The software tool CGPE was initially designed by Revy in order to address the issue of generating efficiently C codes for polynomial evaluation [Rev09, §6]. The main goals were to:

1. optimize the latency for a given VLIW architecture,

2. add systematically numerical accuracy tests in order to prove that the error entailed by the computation in fixed-point arithmetic will be less than a given error bound,

3. have a tool with a low computation time, that could eventually be used at compile time.

Design of CGPE's initial version

Details about the approach embraced by this tool are provided in [MR11], and illustrated in Figure 8.1(a). The user has to provide a univariate or bivariate polynomial (the values of the coefficients and some intervals for the variables), information about her/his target architecture (degree of parallelism, number of multipliers, cost of $+$ and $\times$), and a set of criteria (delay for the second variable y, evaluation error bound, bound on the latency). Then, the tool proceeds in two steps. First, it creates a set of schemes whose latency, on unbounded parallelism and using the provided costs for $+$ and $\times$, is no more than the given bound. Actually, if the user omits to provide a bound then the tool optimizes the latency on unbounded parallelism. Second, each scheme from the computed set goes through a sequence of filters whose purpose is to verify several properties, like the latency on the real architecture and the evaluation error bound. Finally, a `C` code and an accuracy certificate are generated for each scheme that passes all the filters.

To cope with the fast growth for the number of schemes mentioned in Section 6.2.2, several heuristics have been introduced within the "scheme set computation" step:

- The main goal of this step lies in the generation of schemes with low latency. This is achieved by using a target latency (either a bound provided by the user or some estimation of the minimum latency on unbounded parallelism) in order to decide which schemes have a possibly low latency on the target architecture, and drop the others.

- If the support of the input polynomial p is larger than a fixed parameter, the tool performs a non-exhaustive generation. Namely, instead of considering all the possible decompositions for p, it only takes into account decompositions corresponding to a split into its low and high parts.

- Finally, one can decide to keep only a fixed number of schemes for each subexpression.

As we can see, the "scheme set computation" step relies on several parameters that may be set by the user through the command line interface of CGPE.

As for the "scheme selection" step, it is made of three successive filters:

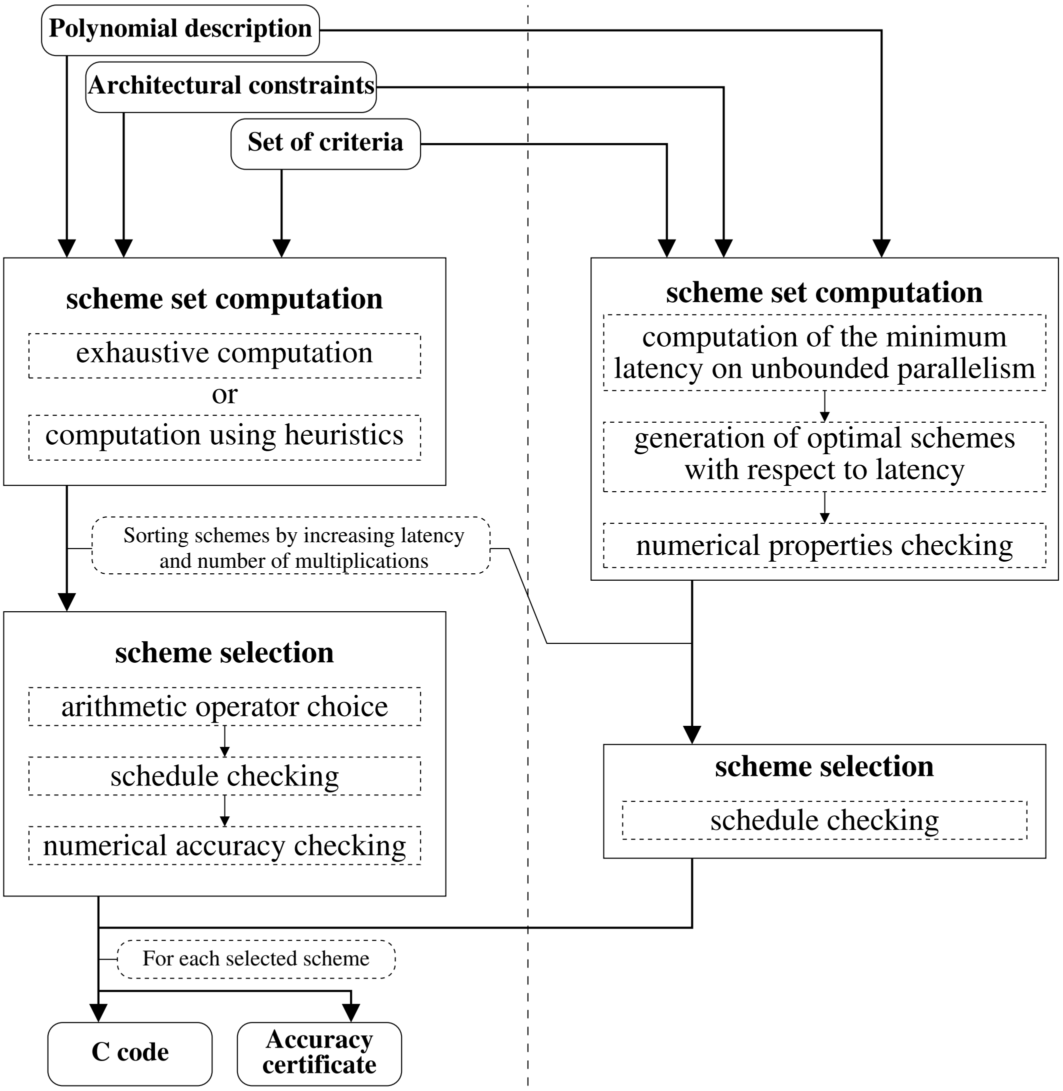
- The first filter, called "arithmetic operator choice", was designed in order to ensure that a given scheme can be evaluated in unsigned fixed-point arithmetic, and without the need to add shifters for mantissa alignment.

- Then, a scheduler tries to check if the latency on the target architecture will meet with the one on unbounded parallelism,

- Finally, the third filter checks that additions never cause an overflow, and that the error entailed by the evaluation in unsigned fixed-point arithmetic is less than the error bound at input. This is done by calling the software tool Gappa[1] on a script generated on purpose. In fact, this script will serve as the accuracy certificate returned as output.

[1] See http://gappa.gforge.inria.fr/ and [Mel06].

Figure 8.1: Evolution in the architecture of the tool CGPE.


Polynomial description
Architectural constraints
Set of criteria
scheme set computation
exhaustive computation
or
computation using heuristics
Sorting schemes by increasing latency and number of multiplications
scheme selection
arithmetic operator choice
schedule checking
numerical accuracy checking
For each selected scheme
C code
Accuracy certificate
scheme set computation
computation of the minimum latency on unbounded parallelism
generation of optimal schemes with respect to latency
numerical properties checking
scheme selection
schedule checking

(a) Architecture of the initial version of the tool.

(b) New design proposed.

Illustration and limitations of this approach

In order to get a good insight into the performance of the approach described previously, we have considered the implementation of various functions for the ST231 processor. For each function, we have computed a polynomial approximant and a certified evaluation error bound using the software tool Sollya[2] and the framework presented in [Rev09, §6.4]. Then, we have used CGPE to handle the generation of the code for polynomial evaluation. Table 8.1 shows the computation time for the different steps, along with the number of schemes at the end of each step represented in square brackets.

[2] http://sollya.gforge.inria.fr/

Table 8.1: Timings for the initial design of CGPE.

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$	$\frac{\sin(1+x)}{1+x}+1$	$\exp(\cos(1+x))$
Degree (d_x,d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)	(5,0)	(8,0)
Delay on y	2	3	9	9	–	–	–	–	–
Approximation interval	$\{1, 2^{1/2}\}\times[0,1]$		$\{1, 2^{1/3}, 2^{2/3}\}\times[0,1]$		$[0.5, 1]$	$[0, 0.5]$	$[0, 1]$	$[0, 1]$	$[0, 1]$
Initial target latency	13	13	16	16	10	10	13	10	13
Final target latency	13	13	16	16	11	11	13	10	13
Achieved latency	13	14	16	16	11	11	13	10	13
Scheme computation	183ms [50]	81ms [50]	25s [50]	26s [50]	8ms [50]	5ms [50]	43ms [50]	1ms [12]	123ms [50]
Arithmetic operator choice	3ms [36]	3ms [28]	5ms [30]	5ms [26]	1ms [2]	2ms [12]	3ms [28]	1ms [8]	2ms [16]
Scheduling checking	22s [9]	1m57s [1]	35ms [30]	423ms [24]	2ms [1]	63ms [5]	1m7s [5]	1ms [8]	88ms [15]
Certification (Gappa)	8s [9]	871ms [1]	23s [30]	22s [24]	204ms [1]	1.3s [5]	6.5s [4]	1.3s [8]	8.4s [13]
Total time ($\approx$)	30s	1m59s	49s	49s	0.2s	1.4s	1m14s	1.3s	8.6s

While the timings are quite good, we can still point out a few issues:

- A bound on the latency is never provided, so the target latency in use came from a heuristic estimation of the minimum latency on unbounded parallelism. Unfortunately, this estimation was 1 cycle below the actual value for two cases.

- Numerical aspects are only taken into account after the "scheme set computation" step. Therefore, one numerically bad subscheme could end up in several of the final sets of schemes, which will have to be invalidated individually despite the common source of error. This is what happens for instance with function $\log_2(1 + x)$ in Table 8.1.

- What dominates the generation cost is the last two filters. In particular, the scheduler is very slow when given an invalid scheme (see the timings for functions $x^{-1/2}$ and $\frac{\exp(1+x)}{1+x}$ in Table 8.1, for instance), and Gappa may lose some time in analyzing several times the same subscheme for the reason mentioned in the previous point.

In addition, setting properly all the parameters for the scheme computation step required us to acquire some good knowledge on the combinatorics of evaluation schemes. In particular, the decision to limit exhaustive generation to polynomials with a support of size no more than 5 was motivated by the values in Tables 6.1 and 6.3. Moreover, the study on the distribution of schemes according to latency of Section 6.4, which tells us for instance that there are $69,384,330$ optimal schemes for a degree-8 univariate polynomial, confirms the need for other heuristics. Thus, a user unfamiliar with these issues may have difficulties to use CGPE efficiently. Finally, the choice to consider only decompositions into low part plus high part for polynomials with large supports can be motivated by noting that this should lead, in the manner of Estrin's rule, to schemes with low latency. Nevertheless, this still remains a somewhat arbitrary choice.

8.1.2 Adding more constraints within the "scheme set computation" step

In the design of CGPE introduced in the previous section, the approach of the "scheme set computation" step is very similar to our algorithm `GenerateWithHint` used with the measure corresponding to latency. The lesson learnt from the previous chapters is that we can really improve some recursive analysis of the set of evaluation schemes, both in terms of speed and result's relevance, by adding more constraints. Thus, we propose here three ideas, detailed below, to improve CGPE (the first two being already mentioned in [MR11]). This yields the new design shown in Figure 8.1(b).

Optimizing the latency in a precomputation

As we have seen in Section 7.1.1, minimizing the latency can be achieved efficiently using `Optimizer`. Therefore, we propose to call this algorithm before we start the "scheme set computation" step. The motivation for this precomputation is twofold:

- Recall that the initial design of CGPE relies on a heuristic estimation of the minimum latency on unbounded parallelism. Instead of starting with a possibly underestimated value, it is better to actually compute the exact value.
- We can keep trace of all the decompositions leading to optimal schemes. Then, instead of restricting ourselves to decompositions of the type "low part plus high part" as explained before, we would rather use the set of decompositions pointed out by `Optimizer`, at least as long as we generate subschemes on a critical path.

It should also be noted that using `Optimizer` beforehand results in a simplification of the user interface. Indeed, several of the technical parameters for the "scheme set computation" step were introduced in order to lead the recursive generation of schemes. With our precomputation, a more relevant information is automatically deduced, so that the user now only has to state the number of schemes to be kept for each subexpression.

Inserting numerical constraints during the computation of schemes

The second improvement lies in the introduction of a systematic verification of the numerical properties as soon as a scheme is generated. Then, we can discard on the fly schemes that may not be evaluated with unsigned fixed-point arithmetic or the one with overflowing additions. This way, we can avoid the situation where one good subscheme according to latency, but not suitable numerically speaking, ends up in many of the final schemes, invalidating them all.

To perform this verification, we propose to adapt the model of accuracy of [Mar09a] and presented in Section 5.4.2. Thus, we will attach two intervals to each generated scheme s: a first interval **value**(s) enclosing all the possible values at execution time; and a second interval **error**(s) enclosing the difference between the real mathematical value and the result actually obtained on the architecture. Checking the constant-sign constraint reduces therefore to see whether 0 belongs to the interior of **value**(s) or not, and the absence of overflow for an addition can be proved whenever all the values in **value**(s) fit in the chosen fixed-point format.

For any trivial scheme s, thus corresponding to a coefficient or a variable, $\mathbf{value}(s)$ is an input of our problem, and we set $\mathbf{error}(s) = [0, 0]$ since coefficients and variables are supposed to be exactly representable. Otherwise, we have $s = (\diamond, \{s_1, s_2\})$. The quantity $\mathbf{value}(s)$ is obtained directly by using multiple precision floating-point interval arithmetic [Moo66, AH83] as provided by the library MPFI,[3] and $\mathbf{error}(s)$ can be deduced as follows:

- If $\diamond = +$ then s_1 and s_2 have to be in the same fixed-point format. Moreover, $\mathbf{value}(s)$ must have a constant sign and ensure that no overflow occurs. When it is the case, the addition entails no error and thus we define

  $$\mathbf{error}(s) = \mathbf{error}(s_1) + \mathbf{error}(s_2).$$

  Otherwise, the current scheme is not suitable for our purpose, and so we discard it;

- If $\diamond = \times$, we first check whether the fixed-point numbers in $\mathbf{value}(s)$ can be stored with no more than f bits for the integer part, where f is the integer part size chosen for the current subexpression. If not, s is not suitable and thus discarded. Otherwise, its error bound is computed as follows:

  $$\begin{aligned} \mathbf{error}(s) &= \mathbf{error}_{\text{mul}} + \mathbf{error}(s_1) \cdot \mathbf{error}(s_2) \\ &+ \mathbf{error}(s_1) \cdot \mathbf{value}(s_2) \\ &+ \mathbf{value}(s_1) \cdot \mathbf{error}(s_2), \end{aligned}$$

  where $\mathbf{error}_{\text{mul}}$ is the error entailed by the unsigned multiplication itself. On the ST231 processor, we have a truncated multiplication so that $\mathbf{error}_{\text{mul}} = [0, 2^{f-32}]$.

Notice that, with this verification, both the first and third filters mentioned in the previous section become useless. Indeed, generated schemes will pass the first filter by construction, and it is straightforward to check, for each scheme s in the final set, whether $\mathbf{error}(s)$ has a magnitude less than the maximum error bound provided by the user. Therefore, we can avoid the calls to Gappa.

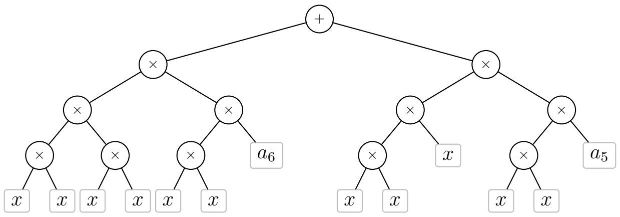
Finally, it should be noted that interval arithmetic can also be used to bound values and errors for a polynomial evaluation in floating-point arithmetic, thanks to models like the one in [Mar09a]. Therefore, while we have focused here on fixed-point arithmetic, one can design a similar numerical constraint in order to check, for instance, the absence of overflow and underflow in an evaluation using floating-point arithmetic.

Early schedulability checking

As a third improvement, we propose to add a simple test of schedulability on the target architecture for each generated scheme. The main motivation for this is to avoid dealing with schemes that will obviously not have the same latency on the target architecture as on unbounded parallelism. As a consequence, we hope to increase the proportion of schemes that will pass the schedule checking in the selection step.

[3] See `http://gforge.inria.fr/projects/mpfi/` and [RR05].

Figure 8.2: Example of scheme that passes the early schedulability test, but fails to achieve the same latency on unbounded parallelism and on the ST231 processor.



The test we have implemented is based on the following remark. If we encounter a scheme with several critical paths (that is, with several paths having a cost equal to the latency on unbounded parallelism), we will need to launch many operations during the first cycles. However, we usually have only a few issues available, like for the ST231 processor which has 4 issues and only 2 multipliers. Hence, we may not have enough resources to launch all the operations in the critical paths.

Therefore, each time we create a new scheme, we compute for each cycle the number of operations and the number of multiplications that should be launched before this cycle. Then, we check that, at any time, the number of operations (resp. multiplications) to be launched is no more than the maximal number of operations (resp. multiplications) that can be launched on our architecture. Consider for instance the scheme of $p(x) = a_6 \cdot x^6 + a_5 \cdot x^5$ from Figure 8.2, whose latency on unbounded parallelism (using ST321 processor's costs of $+$ and $\times$) is 10 cycles. This scheme will pass our test since there are $1 \leq 2$ multiplication $(x \times x)$ to be launched at cycle 0 (before cycle 1), $5 \leq 2 \times 4$ multiplications to be launched before cycle 4, $7 \leq 2 \times 7$ multiplications to be launched before cycle 7, and finally $7 \leq 2 \times 10$ multiplications and $8 \leq 4 \times 10$ operations to be launched before cycle 10.

Notice that we completely forget about dependency between the operations here, so a scheme that passes our test may still have no good scheduling on the target architecture. Actually, the scheme presented in Figure 8.2 is not schedulable in 10 cycles on the ST231 processor. Indeed, operations $x \times x^2$, $x^2 \times x^2$, $x^2 \times a_6$ and $x^2 \times a_5$ cannot be launched before cycle 3, since the value of x^2 is needed. And, as they all are on critical paths, we have four multiplications to schedule on cycle 3, which exceeds the number of available multipliers on the ST231 processor. Nevertheless, we observe in practice that this simple test already removes thousands of unschedulable schemes for several of the functions considered in Table 8.2.

8.1.3 Experimental results

In order to compare this new design of CGPE to the previous one, we have redone the experiments from Table 8.1. The new timings are shown in Table 8.2. Before commenting the new results in detail, it should be noted that the naive usage of interval arithmetic presented in Section 8.1.2 is a fast but rough way to bound the actual values and errors. Therefore, using it within the "scheme set computation" step may lead us to drop many valid schemes. Actually, there were two cases, $\log_2(1+x)$ and $\frac{\exp(1+x)}{1+x}$, where we ended up with no scheme left at the end of this step. When this situation happens, we run again the scheme computation step without the numerical checking that is postponed to an additional filter, based on a call to Gappa, within the "scheme selection" step. In fact, Gappa makes a more clever use of interval arithmetic, so that it usually provides better bounds than naive interval arithmetic. Yet, it is also slower, and we prefer to avoid it when naive interval arithmetic is sufficient.

As we can see in Figure 8.2, the precomputation of the minimal latency on unbounded parallelism is merely free. Moreover, the restriction in the decompositions resulting from this precomputation allows us to quickly compute a set of optimal schemes with respect to the latency. This is particularly true for $x^{1/3}$ and $x^{-1/3}$ where we do not suffer from a slow computation step due to the large target latency anymore.

The "scheme selection" step is also greatly improved by the new design. As remarked earlier, the numerical checking applied during the "scheme set computation" step renders two of the three filters obsolete. Thus, for 7 of the 9 functions, we can avoid costly calls to Gappa. Moreover, we are able (except for one function) to exhibit more schemes that can be evaluated in unsigned fixed-point arithmetic, as we can see by comparing the number of schemes left after the "scheme set computation" step in Table 8.2 to the number of schemes left after the arithmetic operator choice filter in Table 8.1. As for the two remaining cases (that is, $\log_2(1+x)$ and $\frac{\exp(1+x)}{1+x}$), the first scheme computation with numerical checking is actually quite fast,[4] so it does not entail a large overhead. In addition, we can still benefit from the other additional constraints during the second scheme computation. For instance, in the case of $\frac{\exp(1+x)}{1+x}$, we obtain thanks to our schedulability test 23 schedulable schemes instead of 5. Since our scheduler usually runs faster on schedulable schemes, this saves a lot of work later in the scheduling checking. In practice, the cost for this filter indeed drops from 67 to 3 seconds.

Table 8.2: Timings for the new design of CGPE.

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$	$\frac{\sin(1+x)}{1+x}+1$	$\exp(\cos(1+x))$
Degree (d_x,d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)	(5,0)	(8,0)
Delay on y	2	3	9	9	–	–	–	–	–
Approximation interval	$\{1, 2^{1/2}\} \times [0,1]$		$\{1, 2^{1/3}, 2^{2/3}\} \times [0,1]$		$[0.5,1]$	$[0,0.5]$	$[0,1]$	$[0,1]$	$[0,1]$
Minimum latency	13	13	16	16	11	11	13	10	13
Achieved latency	13	14	16	16	11	11	13	10	13
Latency optimization	49ms	141ms	60ms	193ms	2ms	5ms	127ms	1ms	15ms
Scheme computation	29ms	51ms	4.5s	2.4s	3ms	2ms	123ms	1ms	47ms
	[50]	[50]	[50]	[50]	[13]	[50]	[50]	[21]	[50]
Scheduling checking	23s	1m38s	74ms	349ms	49ms	766ms	3s	3ms	812ms
	[11]	[1]	[50]	[47]	[1]	[5]	[23]	[21]	[4]
Additional numerical checking	–	–	–	–	236ms	–	29s	–	–
					[1]		[18]		
Total time ($\approx$)	23s	1m38s	5.3s	3.3s	0.3s	0.8s	32s	60ms	1s

[4] In fact, we arrive soon at a point where many subexpressions have no valid scheme, so that the tool has no work to do after the corresponding recursive calls.

We draw up a summary of the results and total computation times for both approaches in Table 8.3. As we can see, our new design usually produces at least as many schemes as the previous approach, and with a smaller amount of time. It only fails once to achieve as many schemes, but in exchange for a sizeable speed-up. Moreover, it is slower than the previous design only once. Finally, while it is delicate to draw quantitative conclusions since the computation time for both approaches is very sensitive to several aspects (like the inherent difficulty of the polynomial evaluation and the number of schemes left after each step), we observe for our small set of examples an average gain of $\approx$ 50% in the total generation time. This, in addition to the decrease in the number of parameters to be fixed by the user, allows us to conclude that the new design performs better and that its underlying approach is promising.

Table 8.3: Comparison between the initial version of CGPE and our new design.

	Function	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$	$\frac{\sin(1+x)}{1+x}+1$	$\exp(\cos(1+x))$
final number of schemes	version 1	9	1	30	24	1	5	4	8	**13**
	new design	**11**	1	**50**	**47**	1	5	**18**	**21**	4
computation time	version 1	30s	1m59s	49s	49s	**0.2s**	1.4s	1m14s	1.3s	8.6s
	new design	**23s**	**1m38s**	**5.3s**	**3.3s**	0.3s	**0.8s**	**32s**	**60ms**	**1s**
gain in the computation time	$t_{\text{old}} - t_{\text{new}}$	7s	19s	43.7s	45s	-0.1s	0.6s	52s	1.2s	7.6s
	$\frac{t_{\text{old}} - t_{\text{new}}}{t_{\text{old}}}$	23%	17.6%	89%	93%	-50%	43%	62%	95%	88%

8.2 Evaluation of a polynomial at a matrix point

Up to now, we have only addressed the issue of evaluating polynomials at a scalar point. Yet the framework introduced in the previous chapters is suitable for the study of evaluation schemes for $p(\mathsf{A})$ where p is a degree-d polynomial with coefficients in $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$, and A is a matrix in $\mathbb{K}^{n\times n}$.

8.2.1 Motivation and underlying issues

Context and motivation

One can extend the definition of functions from $\mathbb{K}$ to $\mathbb{K}$ like sines, logarithms or exponentials in order to get a new function taking a matrix as input and returning another matrix. Evaluating such matrix functions is of great interest. In particular, we can find exponentials of matrices in the solutions of linear ordinary differential equations, which in turn appear in control theory and in simulations in physics. We refer to [Hig08, §1.1 – §1.4] for more details.

Higham proposes in [Hig08] algorithms for evaluating $f(\mathsf{A})$ with $f \in \{\exp, \log, \sin, \cos\}$ based on rational approximants for these functions. To be more precise, these algorithms use a (d, d')-Padé approximant of the corresponding power series $f(x) = \sum_{i\geq 0} f_i\, x^i$, that is, a pair of polynomials $(p,\, q)$ such that:

- $\deg p \leq d$ and $\deg q \leq d'$,
- $q(x) \cdot f(x) - p(x) = O(x^{d+d'+1})$.

The problem of evaluating $f(\mathsf{A})$ reduces then to the evaluation of two polynomials p and q at a matrix point, followed by the computation of $\mathsf{D}^{-1}\mathsf{N}$ or $\mathsf{N}\mathsf{D}^{-1}$ where $\mathsf{N} = p(\mathsf{A})$ is the matrix numerator and $\mathsf{D} = q(\mathsf{A})$ is the matrix denominator.

How to evaluate $p(\mathsf{A})$

The evaluation of $p(\mathsf{A})$ can be achieved in three ways:

1. In a context where transforming the polynomial or the matrix at input is not allowed, the evaluation will correspond to a straight-line program starting with A and the coefficients of p and using only matrix additions, scalar-matrix multiplications, and matrix-matrix multiplications. The efficiency will depend essentially on the number of matrix-matrix multiplications in the scheme, since this operation costs $O(n^3)$ or $O(n^\omega)$, while the others are in $O(n^2)$. The question of minimizing the number of non-scalar multiplications in a polynomial evaluation scheme is addressed by [PS73], where the authors give an algorithm (Algorithm B in their article) for evaluating $p(\mathsf{A})$ with about $2\sqrt{d}$ non-scalar multiplications, which leads to a cost of $2\mathsf{MM}(n)\sqrt{d} + o(\mathsf{MM}(n)\sqrt{d})$ arithmetic operations on $\mathbb{K}$;
2. The authors of [PS73] also proposed one algorithm (Algorithm C in their article) mixing their preconditioning techniques [PS73, Algorithm A] for polynomial evaluations with their previous approach to minimize further the number of non-scalar multiplications. Thus, they manage to decrease the number of non-scalar multiplications to $\sqrt{2d}+O(\log d)$, which gives an overall cost of $\mathsf{MM}(n)\sqrt{2d}+o(\mathsf{MM}(n)\sqrt{d})$;
3. Finally, one can compute a suitable decomposition for the matrix A, so that the problem of computing $p(\mathsf{A})$ becomes easier. In [Gie95] for instance, the author uses the decomposition underlying the Frobenius normal form in order to achieve a cost of $O\tilde{}(\mathsf{MM}(n) + d)$ for the evaluation of $p(\mathsf{A})$.

In the current context, the more relevant way to evaluate $p(\mathsf{A})$ seems to be the first one, for the two following reasons:

- First, the degree of the polynomial coming from the Padé approximants (typically, $d \leq 13$) mentioned before are too small for the polynomial preconditioning techniques to become better than the approach without preconditioning. Recall indeed the extra $O(\log d)$ in the number of non-scalar multiplications achieved, which is not compensated for by the gain of $\sqrt{2d}$ non-scalar multiplications for small values of d;
- Second, we want to consider only simple algorithms, using + and × only and without any preconditioning, so that the error bound on the polynomial evaluation described in [Hig08, Theorem 4.5] remains valid. This way, we can change the polynomial evaluation without compromising the numerical properties of the whole approach.

Therefore, the reference for the evaluation of $p(\mathsf{A})$ will be here the first algorithm of [PS73] mentioned above. It should be noted that the authors give a lower bound[5] of $\sqrt{d}$ non-scalar multiplications for this evaluation. On the other hand, and that a careful analysis of their algorithm shows that it uses $\left\lfloor \dfrac{d-1}{k} \right\rfloor + k - 1$ non-scalar multiplications, where k is a parameter typically fixed to $\lfloor \sqrt{d} \rfloor$ of $\lceil \sqrt{d} \rceil$ in order to get a final value of roughly $2\sqrt{d}$.

The special case of $\exp(\mathsf{A})$

Algorithm 8.1 illustrates Higham's approach for the case of $\exp(\mathsf{A})$. To ensure a good numerical result, the matrix A is first pre-scaled so that $\exp(\mathsf{A})$ is computed through $\big(\exp(\tilde{\mathsf{A}})\big)^{(2^s)}$, where $\tilde{\mathsf{A}} := \mathsf{A}/2^s$. Then, as $\tilde{\mathsf{A}}$ has a small norm, the exponential is replaced with a (d, d)-Padé approximant.

Algorithm 8.1: Sketch of the approach in [Hig08, §10.3] for the evaluation of $\exp(\mathsf{A})$.

```
Input     : $\mathsf{A} \in \mathbb{C}^{n \times n}$.
Parameter: a number $s$ used for scaling purpose, and a degree $d$ for the Padé
            approximant.
Output    : $\exp(\mathsf{A})$.
$\tilde{\mathsf{A}} \leftarrow \mathsf{A}/2^s$
compute a $(d, d)$-Padé approximant $\dfrac{p(x)}{q(x)}$ for $\exp(x)$
// in the case of $\exp(x)$, we have $q(x) = p(-x) = p_{\text{even}}(x) - p_{\text{odd}}(x)$,
// where $p_{\text{even}}$ and $p_{\text{odd}}$ are the even and odd parts of $p$,
// respectively
$\mathsf{U} \leftarrow p_{\text{even}}(\tilde{\mathsf{A}})$ ;      $\mathsf{V} \leftarrow p_{\text{odd}}(\tilde{\mathsf{A}})$
$\tilde{\mathsf{R}} \leftarrow (\mathsf{U} - \mathsf{V})^{-1}(\mathsf{U} + \mathsf{V})$
compute $\mathsf{R} = \tilde{\mathsf{R}}^{(2^s)}$ by $s$ successive squarings
return $\mathsf{R}$
```

What is really interesting in this example is the link between the two polynomials involved in this Padé approximation, which actually reflects the fact that $e^{-x} = 1/e^x$. This rises the question of evaluating simultaneously both the even and odd parts of a given polynomial at a matrix point in an efficient way.

We do not insist on the choices for parameters s and d, see the discussion in [Hig08, §10.3] for the details. The conclusion is that a trade-off between the degree d for the approximation and the scaling factor 2^s (and so the number s of final squarings) has to be found, that this trade-off depends on the number of matrix-matrix multiplications involved in the polynomial evaluations, and that the interesting range for d is $1 \leq d \leq 13$. What we aim at here is to study the polynomial evaluation step itself, and to automatically retrieve or improve the evaluation scheme proposed in [Hig08, Equation 10.34].

[5] Division is not allowed in their model.

Summary of the issues we want to tackle using CGPE

In the next section, we will focus on the following problems:

1. Determine the minimum number of matrix-matrix multiplications in an evaluation scheme for $p(\mathsf{A})$, and see how it compares to the lower and upper bounds from [PS73];

2. Count the number of optimal schemes for small values of d and generate them, in order to look at the possible improvements (like minimizing the memory usage) for the evaluation;

3. Investigate the case of the simultaneous evaluation of p_{even} and p_{odd} appearing for $\exp(\mathsf{A})$, and more generally the simultaneous evaluation of two polynomials, which is the situation brought by the Padé-approximation approach for evaluating $f(\mathsf{A})$.

8.2.2 Modelling with CGPE and experimental results

Relevance of evaluation schemes for the evaluation of $p(\mathsf{A})$

First of all, let us see why the concept of evaluation scheme remains suitable to study the evaluation of a polynomial at a matrix point. In Sections 5.1.1 and 5.1.2, we have successively introduced for a given arithmetic expression f the set of parenthesizations $\mathcal{P}(f)$ (that were obtained by considering the closure of the natural parenthesizations over the basic set of rules listed on page 95), and the set of evaluation schemes $\mathcal{S}(f) := \mathcal{P}(f)/\equiv$ where $\equiv$ stands for the syntactic equality modulo commutativity.

Here, we have to deal with matrices and scalars (the coefficients a_i of polynomial p). For consistency, we will regard a scalar a_i as the matrix $a_i\mathbb{I}_n$ when applying one of the rules used to define parenthesizations. Yet, multiplying by a_i will remain a scalar-matrix multiplication. Now, we have to look at the validity of these rules: the set of $n \times n$ matrices over a field $\mathbb{K}$ has a non-commutative ring structure, and as we will only encounter matrices that are polynomials in the given matrix A, commutativity of $\times$ will also be satisfied. Therefore, the set of parenthesizations $\mathcal{P}(p(\mathsf{A}))$ still corresponds to a set of ways to evaluate $p(\mathsf{A})$.

When introducing the concept of evaluation scheme in Section 5.1.2, we motivated the choice of considering the quotient set $\mathcal{P}(f)/\equiv$ rather than $\mathcal{P}(f)$ itself by noting that the commutativity of $+$ and $\times$ is still satisfied when computing in finite-precision arithmetic. Unfortunately, this argument falls down here, since we may have $\mathsf{A}\cdot\mathsf{B} \neq \mathsf{B}\cdot\mathsf{A}$ in finite-precision arithmetic even though the mathematical equality holds. Nevertheless, we have seen in the previous section that the main issue will be to minimize the number of non-scalar multiplications. These multiplications are all of the form $\mathsf{X}\cdot\mathsf{Y}$, where X and Y are either two powers of A, or one power and one polynomial of A, and thus their number does remain unchanged when applying the commutativity of $\times$. Moreover, as soon as the numerical analysis is carried out using matrix norms, the error bound on the evaluation of $\mathsf{A}\cdot\mathsf{B}$ becomes symmetric in A and B, like in [Hig02, page 71]:

$$\|\mathsf{AB} - \widehat{\mathsf{AB}}\|_p \leq \gamma_n \|\mathsf{A}\|_p \|\mathsf{B}\|_p, \qquad p = 1, \infty, F.$$

So we will not be able to distinguish between $\mathsf{A}\cdot\mathsf{B}$ and $\mathsf{B}\cdot\mathsf{A}$ anyway. Therefore, the set $\mathcal{S}(p(\mathsf{A}))$ remains relevant as a model for the different ways to evaluate $p(\mathsf{A})$ on a machine with finite-precision arithmetic.

Minimizing the number of non-scalar multiplications

In order to minimize the number of non-scalar multiplications in an evaluation scheme for $p(\mathsf{A})$, we will use a new measure mainly based on the one described in Section 6.4.2. Actually, the only difference with the previous measure lies in the fact that we have to avoid counting multiplications of type $a_i \cdot \mathsf{A}^j$ since they are scalar-matrix ones. Thus, we will use here the measure $\varphi : \mathcal{S}(\mathcal{F}) \to T := \mathbb{N} \times \mathcal{P}(\mathbb{N}^2)$ such that:

- For a given scheme s, $\varphi(s)$ returns both the number of matrix-matrix multiplications involving at least one coefficient a_i and the set of powers of A computed within the scheme;
- For any trivial scheme s (that is, a single leaf), we set $\varphi(s) := (0, \{\})$;
- For any non-trivial scheme, φ can be computed recursively using the function ρ defined by:

$$\begin{aligned}
\rho(+,\ (a_1,b_1),\ (a_2,b_2)) &= (a_1+a_2,\ b_1\cup b_2),\\
\rho(\times,\ (a_1,b_1),\ (a_2,b_2)) &= (a_1+a_2,\ b_1\cup b_2) \text{ when } f_1 \text{ or } f_2 \text{ is reduced to } a_i,\\
\rho(\times,\ (a_1,b_1),\ (a_2,b_2)) &= (a_1+a_2+1,\ b_1\cup b_2) \text{ when } f_1 \text{ or } f_2 \text{ is not a power of } \mathsf{A},\\
\rho(\times,\ (0,b_1),\ (0,b_2)) &= \big(0,\ b_1\cup b_2\cup\{(i+j,\sigma)\}\big) \text{ when } f_1=\mathsf{A}^i,\ f_2=\mathsf{A}^j,\\
&\ \text{and the scheme in use for } \mathsf{A}^{i+j} \text{ is the } \sigma\text{th one.}
\end{aligned}$$

The total order for T will be the same as in Section 6.4.2, that is: Fewer multiplications is better; Then, the more A^i's computed, the better (since this increases the potential of common subexpressions); Finally, the sets of schemes for the A^is are compared.

Now that we have an appropriate measure, we can use some of the algorithms introduced in the previous chapters in order to study the set of evaluations schemes for $p(\mathsf{A})$ with a minimum number of matrix-matrix multiplications. The result of our study is presented in Table 8.4:

- Using `GlobalOptimizerWithHint`, we can first determine this minimum number of matrix-matrix multiplications, and produce one scheme that is optimal with this respect. The main conclusion we can draw here is that, for $d \leq 15$, this minimum number coincides with the value achieved through Paterson and Stockmeyer's algorithm [PS73]. Hence, even though the best known lower bound is smaller, their approach is optimal.
- The number of optimal schemes can be determined with `CountWithHint`. Comparing with the total number of evaluation schemes shown in Table 6.1, we can see that only a few schemes are optimal in this context. In fact, their number is even small enough to think of an exhaustive search. Notice also that, as `CountWithHint` has only to deal with small-size numbers, its computing time is only slightly larger that the one for `GlobalOptimizerWithHint`.

- Finally, we have used `GenerateWithHint` first in order to confirm the results of the previous two algorithms, and second to look at the shape of the optimal evaluation schemes. As one can see, the generation time is quite low for $d \leq 9$, so that a fine tuning through exhaustive analysis is possible even at compile time. Also, one can go through the exhaustive study for $d \leq 12$ within a reasonable amount of time.

Table 8.4: Analysis of the evaluations schemes for $p(\mathsf{A})$.

	`GlobalOptimizerWithHint`		`CountWithHint`		`GenerateWithHint`
$\deg(p)$	minimum number of non-scalar multiplications	optimization time	number of optimal schemes	counting time	generation time
0	0	0s	1	0s	0s
1	0	0s	1	0s	0s
2	1	0s	7	0s	0s
3	2	0s	67	0s	0s
4	2	0s	24	0s	0s
5	3	0s	1056	0s	0s
6	3	0s	702	1s	1s
7	4	1s	74412	1s	1s
8	4	2s	71766	2s	5s
9	4	3s	17550	3s	14s
10	5	159s	15094908	161s	1606s
11	5	360s	9593100	369s	$\approx$ 2.4h
12	5	793s	11820600	819s	$\approx$ 7.8h
13	6	$\approx$ 22h	6558513300	$\approx$ 23.5h	–
14	6	$\approx$ 50h	11623313700	$\approx$ 54h	–
15	6	$\approx$ 98h	18470713500	$\approx$ 100h	–

While Paterson and Stockmeyer's algorithm appears to be optimal in our computation model for $d \leq 15$, there is still room for further investigation. First, let us discuss a classification of optimal schemes for small degrees. By looking at the output of `GenerateWithHint` for $0 \leq d \leq 12$, we were able to distinguish between three classes of optimal schemes:

1. First, we have several variations of Paterson and Stockmeyer's algorithm. Namely, a sequence $\mathsf{A}, \mathsf{A}^2, \dots, \mathsf{A}^k$ is computed for some k, then chunks of size k are formed, and the final result is deduced by using Horner's rule with A^k and these chunks. The differences between these variations come from the associativity of $+$ within the chunks, and the way these chunks are formed ($a_{\ell k} \cdot \mathsf{A}^{\ell k}$ may be placed either as $a_{\ell k} \cdot \mathsf{A}^k$ in the $(\ell - 1)$st chunk, or as $a_{\ell k} \cdot \mathbb{I}_n$ in the ℓth chunk).

2. Second, we find schemes based on the sequence $\mathsf{A}, \mathsf{A}^2, \mathsf{A}^4, \ldots$, performing the evaluation in an even/odd way. These schemes are actually variations of the modular splitting algorithm presented in [BZ09, §4.4], which has usually a few more non-scalar multiplications than Paterson and Stockmeyer's algorithm. However, modular splitting appears to be competitive for $d \in \{5, 7, 10\}$;

3. Third, we were able to extract, for $d = 7$, some exotic schemes (in the sense they do not seem to look like known schemes) such as

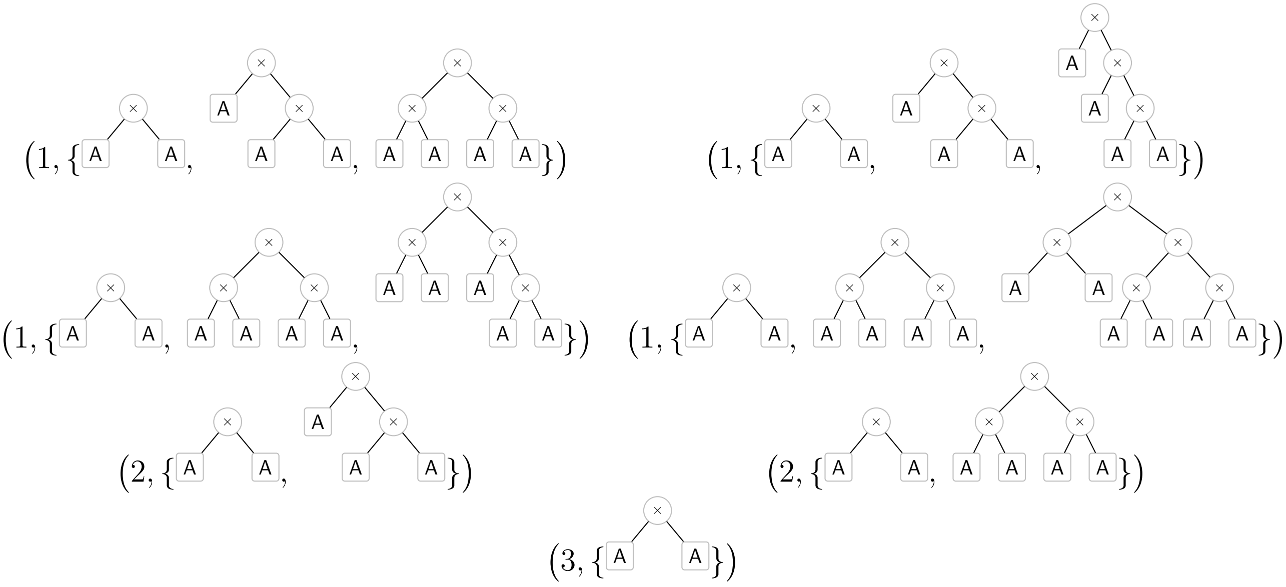
$$\Big((a_0 \cdot \mathbb{I}_n + a_1 \cdot \mathsf{A}) + (a_4 \cdot \mathsf{A}^4 + a_5 \cdot \mathsf{A}^5)\Big) + \Big(\mathsf{A}^2 \times \big(a_2 \cdot \mathbb{I}_n + (a_3 \cdot \mathsf{A} + a_6 \cdot \mathsf{A}^4)\big)\Big),$$

   where

   - $\mathsf{A}^2 = \mathsf{A} \times \mathsf{A}$,
   - $\mathsf{A}^4 = \mathsf{A}^2 \times \mathsf{A}^2 = (\mathsf{A} \times \mathsf{A}) \times (\mathsf{A} \times \mathsf{A})$,
   - $\mathsf{A}^5 = \mathsf{A} \times \mathsf{A}^4 = \mathsf{A} \times \big((\mathsf{A} \times \mathsf{A}) \times (\mathsf{A} \times \mathsf{A})\big)$.

As implied by the optimality of Paterson and Stockmeyer's algorithm, we are able to find, for every degree $d \leq 15$, many schemes that belong to the first class. On the contrary, optimal schemes from the second and third classes only exist for some degrees. By looking at the complete output of `GlobalOptimizerWithHint` for $13 \leq d \leq 15$, we have determined that modular splitting can be optimal only for $d \in \{5, 7, 10, 13, 14\}$, and we have exposed exotic schemes only for $n \in \{7, 13, 14\}$.

Figure 8.3: Set of values for $\varphi(s)$, where s is a scheme for a degree-7 polynomial with the minimum number of non-scalar multiplications.



We have represented here the schemes for each A^i, but they are actually stored as a pair of integers in the implementation.

Such a diversity in the optimal schemes lets us think that improvements of the parallelism, the memory usage, or the numerical quality may be achieved by exhaustive or heuristics search. Yet, what is the most interesting here is the variable number of powers of A computed as intermediate quantities. Indeed, recall that our problem was to evaluate two polynomials at the same time. If we maximize the number of powers of A computed in the first polynomial evaluation, then we will potentially have more operations for free when performing the second evaluation. Think for instance of the case of a $(7, 7)$-Padé approximation (p, q). Figure 8.3 illustrates for a degree-7 polynomial all the values for $\varphi(s)$ where s has the minimum number of matrix-matrix multiplications, that is, four. Suppose we use schemes for p and q whose value by φ is the top-left one in Figure 8.3. Then, evaluating $p(\mathsf{A})$ will cost $1 + 3 = 4$ matrix-matrix multiplications, but since we have also computed A^2, $\mathsf{A}^3 = \mathsf{A} \times \mathsf{A}^2$, and $\mathsf{A}^4 = \mathsf{A}^2 \times \mathsf{A}^2$, evaluating $q(\mathsf{A})$ will only cost 1 matrix multiplication. Therefore, the total number of matrix-matrix multiplications is 5. Supposing now that we choose for p or q a scheme s where $\varphi(s)$ is the value at bottom of Figure 8.3, the only intermediate quantity that we can reuse is A, and so we get a number of matrix-matrix multiplications of $3 + 1 + 3 = 7$, which is two more compared to the previous choice. This illustrates that, in our context, all schemes minimizing the number of non-scalar multiplications are not even, and that it is important to look deeply into the structure of the scheme in order to make an appropriate choice.

Case of $\exp(\mathsf{A})$

Finally, let us discuss the case of $\exp(\mathsf{A})$. Recall that the problem in this case is to evaluate simultaneously the even and odd parts of a polynomial p with a minimum number of matrix-matrix multiplications. Assuming that

$$p(\mathsf{A}) = q(\mathsf{A}^2) + \mathsf{A} \cdot r(\mathsf{A}^2) \quad \text{so that} \quad p_{\text{even}}(\mathsf{A}) = q(\mathsf{A}^2) \text{ and } p_{\text{odd}}(\mathsf{A}) = \mathsf{A} \cdot r(\mathsf{A}^2), \qquad (8.1)$$

we can evaluate the odd and even parts of p by first computing A^2, then evaluating simultaneously r and q at the matrix point A^2, and finally performing one extra addition to retrieve the $p_{\text{odd}}(\mathsf{A})$ from $r(\mathsf{A}^2)$. For the simultaneous evaluation of r and q, we can apply the remark formulated at the end of the previous paragraph. For instance, when $\deg(\mathsf{A}) = 12$, one can evaluate the degree-5 polynomial r with 3 non-scalar multiplications ($\mathsf{A}^2 = \mathsf{A} \times \mathsf{A}$, $\mathsf{A}^3 = \mathsf{A} \times \mathsf{A}^2$, and 1 additional non-scalar multiplication), and the degree-6 polynomial q with 3 non-scalar multiplications (again, $\mathsf{A}^2 = \mathsf{A} \times \mathsf{A}$, $\mathsf{A}^3 = \mathsf{A} \times \mathsf{A}^2$, and 1 additional non-scalar multiplication). Therefore, the simultaneous evaluation will cost $6 - 2 = 4$ non-scalar multiplications, since we can reuse the values of A^2 and A^3, and the final cost for $\{p_{\text{even}}(\mathsf{A}), p_{\text{odd}}(\mathsf{A})\}$ is $1 + 4 + 1 = 6$ matrix-matrix multiplications, which matches the value proposed by Higham [Hig08, Table 10.3].

In fact, the previous reasoning allows us to quickly retrieve fully parenthesized variants of the scheme proposed on [Hig08, page 244] for $d = 12$, along with several other schemes achieving the same number (6) of matrix-matrix multiplications. Indeed, all we need is to generate schemes minimizing the number of non-scalar multiplications for a degree-5 polynomial and then for a degree-6 polynomial, and to consider pairs of schemes which maximize the number of common powers of A. According to Table 8.4, this takes around 1 second.

Yet, one may wonder if there exist better schemes. To answer this question, we use algorithm `OptimizerSet`, along with a variation of the measure described in Section 7.2.1

where non-scalar multiplications are not taken into account. As a first experiment, we ask for the minimal number of non-scalar multiplications for the evaluation of the set

$$\Bigg\{ \sum_{\substack{0 \le i \le d,\\ i \text{ even}}} a_i x^i, \sum_{\substack{0 \le i \le d,\\ i \text{ odd}}} a_i x^i \Bigg\},$$

which is the exact form of our current problem. Thus, we were able to obtain the minimum number of non-scalar multiplications needed to evaluate simultaneously the even and odd parts of a degree-d polynomial when $d \le 7$. These numbers, along with the computation time, are presented in Table 8.5. Since they match the ones from [Hig08, Table 10.3], we are able to conclude that, for $d \le 7$, one cannot hope to perform the simultaneous evaluation of the odd and even parts of p faster than what Higham proposes.

Table 8.5: Minimal number μ_d of non-scalar multiplications for evaluating $p_{\text{even}}(\mathsf{A})$ and $p_{\text{odd}}(\mathsf{A})$ when $\deg(p) = d$ (heuristic for $d \ge 8$).

	exhaustive search								heuristic search					
d	0	1	2	3	4	5	6	7	8	9	10	11	12	13
μ_d	0	0	1	2	3	3	4	4	5	5	6	6	6	6
computation time	0s	0s	0s	0s	0s	1s	22s	694s	0s	1s	8s	132s	824s	$\approx$ 1h

For $d \ge 8$, an exhaustive search with `OptimizerSet` is impossible because of a too large memory consumption. To cope with this problem, we propose a second experiment where we compute with `OptimizerSet` the minimum number of non-scalar multiplications in order to evaluate $\{q, r\}$ as in Equation (8.1), and add 2 to the result in order to take into account the multiplications $\mathsf{A} \times \mathsf{A}$ and $\mathsf{A} \times r(\mathsf{A}^2)$. Since we put some constraints on the way of evaluating the odd and even parts of p, this second experiment is actually a heuristic optimization, and thus it only gives an upper bound on the minimal number of non-scalar multiplications. Results are presented in Table 8.5, and again, they match with the one from Table [Hig08, Table 10.3]. This proves that, if we impose the evaluation of the odd and even parts of p to be performed as in Equation (8.1) then Higham's approach is optimal, at least for $d \le 13$.

Conclusions and perspectives for Part II

The implementation of an given arithmetic expression f rises several issues. People from algebraic complexity try to determine the minimum number of operations in an evaluation scheme for f. The set $\mathcal{S}(f)$ of all the evaluation schemes for f is an interesting combinatorial object by itself. Moreover, an efficient implementation of f on a given architecture requires that we use as much as possible the available parallelism, or that we control the evaluation error due to the computations carried out in finite-precision artihmetic. Sometimes, a trade-off between speed and accuracy must be found, like in the context of CGPE.

The purpose of the second part of this document was to provide a general framework that helps us to tackle several of the aforementioned issues. The main idea lies in the recursive expression of the set of evaluation schemes $\mathcal{S}(f)$ that is achieved thanks to the concept of decomposition. Analyzing the set $\mathcal{S}(f)$ can then be summarized as considering successively all the decompositions for f, analyzing recursively all the corresponding subexpressions, and deducing the final result. This approach was mainly used for three purposes: generation, counting, and optimization.

Generation of evaluation schemes is usually too costly when done exhaustively. However, we have seen that adding sufficiently strong constraints can make it relevant. Thus, by generating all the schemes for a^n with a minimum number of multiplications, we were able to study the number of squarings that can be extracted from the multiplications.

Counting the number of evaluation schemes gives a good insight into the cost of other analyses and the strength of the heuristics we may need in order to render these analyses efficient. Thus, we have studied in detail the numbers of schemes for the polynomials appearing as input of CGPE, that is, univariate polynomials $p(x)$ and bivariate polynomials of the form $q(x, y) = \alpha + y \cdot p(x)$. This gave us two new sequences that we have added to the On-Line Encyclopedia of Integer Sequences. We have then proved that the logarithm of the number of schemes for both $p(x)$ and $q(x, y)$ grows in $\Omega(n^2)$ and in $O(n^3)$ with respect to $n = \deg p$, which explains the fast growth observed in practice. Moreover, we have looked at the distribution of the evaluation schemes according to some measure. In particular, we have seen that only a small proportion of schemes for polynomials achieves a minimum latency, and concluded that it is interesting as a first heuristic to focus on those schemes inside CGPE. Yet, schemes with a minimum latency are too many, and we still need additional heuristics to reduce the search space further.

Optimization according to some criterion like the latency can be easily achieved through the aforementioned divide-and-conquer approach. Yet, some measures like the number of multiplications are more difficult to optimize. For them, we have proposed and compared several solutions: a fast heuristic optimization (`Optimizer`), a generalization of the optimization problem to sets of expressions (`OptimizerSet`), and a global optimization relying on additionnal constraints (`GlobalOptimizerWithHint`). We have also seen how using prioritization within a set of criteria allows us to reuse monocriterion optimization for the purpose of multicriteria optimization. Finally, we have proposed an algorithm aiming at finding good trade-offs given two criteria, that we have applied first to solve a question about latency and delay raised by the design of the FLIP library, and second to study the trade-off between latency and accuracy for polynomial approximation. This last experiment leads us to the conclusion that relaxing slightly the constraint on optimal latency may lead to far more accurate schemes.

Moreover, the material introduced in this part has been used in two more advanced contexts. First, our general framework allowed us to partially rewrite CGPE. By adding more constraints within the step that computes schemes, we were able to significantly decrease the overall generation time for several functions. Second, we have reviewed the evaluation of a univariate polynomial at a matrix point, and shown that, in the absence of preconditionning, Paterson and Stockmeyer's algorithm achieves the minimal number of matrix-matrix multiplications, at least up to degree 15. A similar study for the computation of $\exp(\mathsf{A})$ with Padé approximants also proved the optimality of Higham's approach [Hig08, §10.3] in terms of the number of matrix-matrix multiplications.

Last but not least, we have developed a `C++` library that contains all the algorithms mentioned in Chapters 5 to 7 ($\approx$ 1200 lines of code), several families of arithmetic expressions ($\approx$ 1500 lines of code), and several measures ($\approx$ 400 lines of code). This library was used to develop the example codes (33 of them, for a total of $\approx$ 3200 lines of code) that lead to the results presented all along this part of the document, and we are eager to test it for other types of arithmetic expressions, or with other optimization criteria, like the memory usage for software implementation, and the area usage for hardware. For the case of polynomial matrices, we still need to design a model for the numerical error analysis so as to obtain a complete code generator that really exploits the values of the coefficients of p in order to produce a sharp error bound. Finally, a generalization of our approach in order to handle 3-ary operators would allow us to analyze implementations relying on the FMA operator, which is nowadays heavily used for accurate floating-point implementations.

Final words

The design of efficient code is a very difficult task. At the algorithmic level, significant improvements can be achieved, that either lead to a better asymptotic cost, or to a cost with a better constant hidden within the “big-O” expression. Yet, considering the algorithm with the best known asymptotic complexity does not mean having an efficient code. First, one has to actually implement this algorithm. This usually implies to make implementation choices, which may impact the actual performance of the algorithm. Second, asymptotically fast algorithms are usually slower than more naive ones for small inputs. Thus, one may need to implement several algorithms and switch from one to another depending on the size of the input. Again, choices for the set of algorithms and the ranges associated to each of them have to be made. Third, the produced code will be compiled and run on a given architecture, usually offering parallelism and finite-precision arithmetic for computation involving real or complex numbers. Using at most the features of the architecture within the compiler is then yet another issue.

The main lesson learnt from this work is that, given the increasing complexity of both the algorithms and the architectures on which they are intended to run, designing tools to automatically analyze and optimize code becomes more and more an issue. For structured matrices, we managed to design asymptotically fast algorithms. Nevertheless, despite the time we spent on their implementation, we are convinced that better performance would be achieved if only we could ask a computer to generate the code. As for the evaluation of arithmetic expressions, our framework for automatic analysis of the set of evaluation schemes really helped us to produce efficient code both for the FLIP library and for the evaluation of a polynomial at a matrix point. Thus, in both cases, the solution in order to produce efficient code seems to lie in the automatic generation of optimized code.

List of notation

$\mathbb{N}$	set of natural integers
$\mathbb{N}_{>0}$	set of positive integers
$\mathbb{R}$	set of real numbers
$\mathbb{R}_{>0}$	set of positive real numbers
$\mathbb{C}$	set of complex numbers
$\mathbb{K}$	arbitrary field
$\mathbb{F}_q$	finite field of cardinality q
$\mathbb{K}^{m\times n}$	set of $m \times n$ matrices with coefficients in $\mathbb{K}$
$\mathbb{K}[x]$	set of polynomials with coefficients in $\mathbb{K}$
$\mathbb{K}[x]_d$	set of polynomials with coefficients in $\mathbb{K}$ and degree less than d
$\mathbb{K}[x]_d^{m\times n}$	set of $m \times n$ matrices with coefficients in $\mathbb{K}[x]_d$
$a \operatorname{div} b, a \bmod b$	quotient and remainder in the division of $a \in \mathbb{K}[x]$ by $b \in \mathbb{K}[x]$, $b \neq 0$

$\mathbb{I}_n$	identity matrix of order n	9
$\mathbb{J}_n$	reflexion matrix of order n	9
$\mathsf{e}_{n,i}$	ith unit vector of $\mathbb{K}^n$	9
e_n	vector of $\mathbb{K}^n$ whose all entries are equal to 1	9
$\mathbb{T}(\mathsf{x})$	square Toeplitz matrix defined by the vector $\mathsf{x} \in \mathbb{K}^{2n-1}$	12
$\mathbb{T}(\mathsf{x}, m, n)$	$m \times n$ Toeplitz matrix defined by the vector $\mathsf{x} \in \mathbb{K}^{m+n-1}$	12
$\mathbb{L}(\mathsf{x})$	lower-triangular Toeplitz matrix defined by the vector $\mathsf{x} \in \mathbb{K}^n$	12
$\mathbb{U}(\mathsf{x})$	upper-triangular Toeplitz matrix defined by the vector $\mathsf{x} \in \mathbb{K}^n$	12
$\mathbb{V}(\mathsf{x})$	square Vandermonde matrix defined by the vector $\mathsf{x} \in \mathbb{K}^n$	13
$\mathbb{V}(\mathsf{x}, n)$	$m \times n$ Vandermonde matrix defined by the vector $\mathsf{x} \in \mathbb{K}^m$	13
$\mathbb{H}(\mathsf{x}, m, n)$	$m \times n$ Hankel matrix defined by the vector $\mathsf{x} \in \mathbb{K}^{m+n-1}$	14
$\mathbb{C}(\mathsf{x}, \mathsf{y})$	$m \times n$ Cauchy matrix defined by vectors $\mathsf{x} \in \mathbb{K}^m$ and $\mathsf{y} \in \mathbb{K}^n$	14
$\mathbb{Z}_{n,\varphi}$	unit φ-circulant matrix of order n	16
$\mathbb{D}(\mathsf{x})$	diagonal matrix with x_i as diagonal elements	17
$\mathcal{K}_\ell(\mathsf{M}, \mathsf{u})$	$m \times \ell$ Krylov matrix defined by $\mathsf{M} \in \mathbb{K}^{m\times n}$ and $\mathsf{u} \in \mathbb{K}^n$	19
$\mathsf{X}^{\mapsto\alpha}$	matrix made of the first α columns of X	35
P_{x}	polynomial $\prod_{i=1}^{n}(x - x_i)$	55
$\mathbb{W}(\mathsf{x})$	matrix $\mathbb{D}(P'_{\mathsf{x}}(\mathsf{x}))^{-1}\mathbb{V}(\mathsf{x})$	55

ω	exponent in the cost of matrix multiplication	10
$\mathsf{M}(n)$	cost function for the multiplication of two degree-n polynomials	12
$O\tilde{~}(f(n))$	Landau's “Big-O” notation, where logarithmic factors have been dropped	15

$\mathsf{MM}(n,d)$	cost function for the multiplication of two matrices in $\mathbb{K}[X]_d^{n\times n}$	54
$f_{\mathbb{K}}(n,d)$	cost function introduced for the cost analyses in Chapter 3	54
$\mathsf{SMM}(\alpha, m+n)$	cost function for the multiplication "structured matrix × matrix"	69

$\mathcal{P}(f)$	set of all the parenthesizations for expression f	95
$\mathcal{S}(f)$	set of all the evaluation schemes for expression f	96
$\nu(t)$	arithmetic expression associated to the tree t	98
$\mathcal{D}(f)$	set of all the decompostitions for expression f	98
$\sqcup$	disjoint union	101
$\mathcal{P}_2(X)$	set of all singletons and all pairs included in set X	102

Bibliography

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Analysis*. Academic Press, 1983. [170]

[BA80] R. R. Bitmead and B. D. O. Anderson. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra Appl.*, 34:103–116, 1980. [xi, 24, 25, 73]

[Bab69] I. Babuska. Numerical stability in mathematical analysis. In *Proceedings of the 1968 IFIP Congress*, volume 1, pages 11–23, 1969. [82]

[BCS97] B. Bürgisser, C. Clausen, and M.A. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1997. [79, 93, 94]

[BH74] James R. Bunch and John E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28:231–236, 1974. [11]

[BJMS11] Alin Bostan, Claude-Pierre Jeannerod, Christophe Mouilleron, and Éric Schost. Fast simultaneous multiplication of a structured matrix by vectors, 2011. Draft, available upon request. [26, 53, 68]

[BJS07] Alin Bostan, Claude-Pierre Jeannerod, and Éric Schost. Solving Toeplitz- and Vandermonde-like linear systems with large displacement rank. In *ISSAC'07*, pages 33–40. ACM, 2007. [2, 25, 52, 53, 70, 71, 73]

[BJS08] Alin Bostan, Claude-Pierre Jeannerod, and Éric Schost. Solving structured linear systems with large displacement rank. *Theoretical Computer Science*, 407(1:3):155–181, 2008. [2, 25, 29, 52, 53, 66, 70, 71, 73, 74]

[BKM73] R. Brent, D. Kuck, and K. Maruyama. The parallel evaluation of arithmetic expressions without division. *Computers, IEEE Transactions on*, C-22(5):532 – 534, may 1973. [81]

[BLS03] Alin Bostan, Gregoire Lecerf, and Éric Schost. Tellegen's principle into practice. In *ISSAC'03 proceedings*, pages 37–44, New York, NY, USA, 2003. ACM. [57]

[Bod10] Marco Bodrato. A Strassen-like matrix multiplication suited for squaring and higher power computation. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 273–280, New York, NY, USA, 2010. ACM. [113]

[Bol04] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante.* PhD thesis, ÉNS Lyon, November 2004. [85]

[Bor71] Allan Borodin. Horner's rule is uniquely optimal. *Theory of machines and computations*, pages 45–58, 1971. [138]

[Bos10] Alin Bostan. Algorithmes rapides pour les polynômes, séries formelles et matrices. In *Les cours du C.I.R.M.*, volume 1, pages 75–262, 2010. [2, 3, 26, 28, 29, 52, 53, 62, 69, 70, 73, 74]

[BP94] Dario Bini and Victor Y. Pan. *Polynomial and Matrix Computations, volume 1: Fundamental Algorithms.* Birkhäuser, 1994. [9]

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, April 1974. [81, 93, 94, 95]

[BS05] Alin Bostan and Éric Schost. Polynomial evaluation and interpolation on special sets of points. *Journal of Complexity*, 21:420–446, 2005. [54]

[BZ09] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic.* March 2009. Version 0.2.1. Available at `http://www.loria.fr/~zimmerma/mca/mca-0.2.1.pdf`. [179]

[Car99] Jean-Paul Cardinal. On a property of Cauchy-like matrices. *C. R. Acad. Sci. Paris - Série I - Analyse numérique/Numerical Analysis*, 328:1089–1093, 1999. [2, 25, 26, 27, 31, 33, 41, 42, 73]

[Car00] Jean-Paul Cardinal. A divide and conquer method to solve Cauchy-like systems. Technical report, The FRISCO Consortium, 2000. [xi, 2, 3, 25, 26, 27, 31, 33, 34, 73]

[Che09] Sylvain Chevillard. *Évaluation efficace de fonctions numériques - Outils et exemples.* PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2009. [83]

[CK91] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991. [54]

[CK04] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bulletin*, 38(1):8–15, 2004. [85]

[CLM$^+$05] Ray C. C. Cheung, Dong-U Lee, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Automating custom-precision function evaluation for embedded processors. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 22–31, New York, NY, USA, 2005. ACM. [85]

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009. [104, 151]

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM. [10]

[DLS81] Peter J. Downey, Benton L. Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, pages 638–646, 1981. [149]

[Eth37] Ivor M. H. Etherington. Non-associate powers and a functional equation. *The Mathematical Gazette*, 21(242):36–39, 1937. [80, 119]

[Eve64] James Eve. The evaluation of polynomials. *Numerische Mathematik*, (6):17–21, 1964. [80, 94]

[FFY05] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, 2005. [83]

[Fin94] Steven R. Finch. *Mathematical Constants.* Cambridge University press, 1994. [126]

[FMMP09] Franz Franchetti, Frédéric Mesmay, Daniel Mcfarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 385–409, Berlin, Heidelberg, 2009. Springer-Verlag. [74]

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics.* Princeton University, 2009. [80, 90, 120, 126, 127]

[Gie95] Mark Giesbrecht. Nearly optimal algorithms for canonical matrix forms, 1995. [174]

[GJV03] Pascal Giorgi, Claude-Pierre Jeannerod, and Gilles Villard. On the complexity of polynomial matrix computations. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 135–142. ACM Press, 2003. [26]

[GKO95] I. Gohberg, T. Kailath, and V. Olshevsky. Fast Gaussian elimination with partial pivoting for matrices with displacement structure. *Math. of Comp.*, 64(212):1557–1576, 1995. [26]

[GO94a] I. Gohberg and V. Olshevsky. Complexity of multiplication with vectors for structured matrices. *Linear Algebra Appl.*, 202:163–192, 1994. [19]

[GO94b] I. Gohberg and V. Olshevsky. Fast state space algorithms for matrix Nehari and Nehari-Takagi interpolation problems. *Integral Equations and Operator Theory*, 20:44–83, 1994. [25, 26, 27, 32]

[Gre02] Robin Green. Faster Math Functions. *Tutorial at Game Developers Conference*, 2002. [81]

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. [81, 82, 109, 176]

[Hig08] Nicholas J. Higham. *Functions of Matrices: Theory and Computation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. [xi, 5, 86, 87, 165, 173, 174, 175, 180, 181, 184]

[HKST99] John Harrison, Ted Kubaska, Shane Story, and Peter Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4:1–7, 1999. [81]

[HR84] Georg Heinig and Karla Rost. *Algebraic methods for Toeplitz-like matrices and operators.* Akademie-Verlag, 1984. [34]

[IEE08] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic.* August 2008. [81]

[IMH82] O.H. Ibarra, S. Moran, and R. Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3:45–56, 1982. [10]

[JJLMR11] Claude-Pierre Jeannerod, Jingyan Jourdan-Lu, Christophe Monat, and Guillaume Revy. How to square floats accurately and efficiently on the ST231 integer processor. In Elisardo Antelo, David Hough, and Paulo Ienne, editors, *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, pages 77–81, Tübingen, Germany, July 2011. IEEE Computer Society. [113]

[JKMR08] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, 2008. [122, 161]

[JKMR11] Claude-Pierre Jeannerod, Herve Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. *IEEE Transactions on Computers*, 60:214–227, 2011. [84]

[JM10a] Claude-Pierre Jeannerod and Christophe Mouilleron. Computing specified generators of structured matrix inverses, 2010. LIP research report RR2010-04, available at `http://hal-ens-lyon.archives-ouvertes.fr/ensl-00450272/en/`. [20]

[JM10b] Claude-Pierre Jeannerod and Christophe Mouilleron. Computing specified generators of structured matrix inverses. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 281–288, New York, NY, USA, 2010. ACM. [31]

[JMM$^+$10] Claude-Pierre Jeannerod, Christophe Mouilleron, Jean-Michel Muller, Guillaume Revy, Christian Bertin, Jingyan Jourdan-Lu, Hervé Knochel, and Christophe Monat. Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors. In *Proc. of the 4th International Workshop on Parallel and Symbolic Comp. (PASCO '10)*, pages 1–9, New York, NY, USA, 2010. ACM. [83, 120]

[Kah65] William Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965. [82]

[Kal94] Erich Kaltofen. Asymptotically fast solution of Toeplitz-like singular linear systems. In *ISSAC'94*, pages 297–304. ACM, 1994. [22, 25, 48]

[Kal95] Erich Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, 1995. [25]

[Kha08] Houssam Khalil. *Matrices structurées et matrices de Toeplitz par blocs de Toeplitz en calcul numérique et formel.* PhD thesis, Université Claude-Bernard Lyon 1, July 2008. [26]

[KKM79] T. Kailath, S. Y. Kung, and M. Morf. Displacement ranks of matrices and linear equations. *J. Math. Anal. Appl.*, 68(2):395–407, 1979. [15, 16, 24]

[KM74] David J. Kuck and Yoichi Muraoka. Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity. *Acta Inf.*, pages 203–216, 1974. [81, 95]

[KM75] David J. Kuck and Kiyoshi Maruyama. Time bounds on the parallel evaluation of arithmetic expressions, 1975. [81]

[Knu62] Donald E. Knuth. Evaluation of polynomials by computers. *Communications of the ACM*, 5(12):595–599, 1962. [80, 94]

[Knu69] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 1969. [149]

[Knu98] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Third edition, 1998. [80, 82, 85, 114, 148, 151]

[KS99] T. Kailath and A. H. Sayed, editors. *Fast reliable algorithms for matrices with structure.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. [26]

[Kuc77] David J. Kuck. A survey of parallel machine organization and programming. *ACM Comput. Surv.*, 9:29–59, March 1977. [81, 94, 95, 96]

[Lau08] Christoph Lauter. *Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation.* PhD thesis, Univ. de Lyon - ÉNS Lyon, October 2008. [85]

[LBC95] G. Labahn, B. Bechermann, and S. Cabay. Inversion of Mosaic Hankel Matrices via Matrix Polynomial Systems. *Linear Algebra and its Applications*, (221): 253–280, 1995. [26]

[LCC90] G. Labahn, D. K. Choi, and S. Cabay. The Inverses of Block Hankel and Block Toeplitz Matrices. *SIAM J. of Computing*, (19): 98–123, 1990. [26]

[LMT10] Philippe Langlois, Matthieu Martel, and Laurent Thévenoux. Accuracy versus time: a case study with summation algorithms. In *Proc. of the 4th International Workshop on Parallel and Symbolic Computation (PASCO '10)*, pages 121–130, New York, NY, USA, 2010. ACM. [82, 162]

[LV09] Dong-U Lee and John D. Villasenor. Optimized Custom Precision Function Evaluation for Embedded Processors. *IEEE Transactions on Computers*, 58(1):46–59, 2009. [85]

[Mar07] Matthieu Martel. Semantics-based transformation of arithmetic expressions. In *SAS'07*, volume 4634 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. [82]

[Mar09a] Matthieu Martel. Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics. In *Journal of Formal Methods in System Design*, volume 35, pages 265–278. Springer, 2009. [82, 109, 169, 170]

[Mar09b] Matthieu Martel. Program transformation for numerical precision. In *PEPM'09*. ACM Press, 2009. [82, 109]

[Mat09] David Matula. Higher radix squaring operations employing left-to-right dual recoding. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 39 –47, june 2009. [113]

[MBdD$^+$10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser Boston, 2010. [83, 96]

[Mel06] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes.* PhD thesis, ÉNS Lyon, November 2006. [82, 84, 166]

[Mic68] Donald Michie. “memo” functions and machine learning. *Nature*, 218:19–22, 1968. [104]

[Moo66] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966. [170]

[Mor80] M. Morf. Doubling algorithms for Toeplitz and related equations. *IEEE Conference on Acoustics, Speech, and Signal Processing*, pages 954–959, 1980. [xi, 24, 25, 73]

[MR11] Christophe Mouilleron and Guillaume Revy. Automatic generation of fast and certified code for polynomial evaluation. In Elisardo Antelo, David Hough, and Paulo Ienne, editors, *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, pages 233–242, Tübingen, Germany, July 2011. IEEE Computer Society. [82, 139, 154, 166, 169]

[Mul00] Thom Mulders. On short multiplications and divisions. *Applicable Algebra in Engineering, Communication and Computing*, 11:69–88, 2000. 10.1007/s002000000037. [12]

[Mul06] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation.* Birkhäuser Boston, MA, 2nd edition, 2006. [83]

[OP98] Vadim Olshevsky and Victor Y. Pan. A unified superfast algorithm for boundary rational tangential interpolation problems and for inversion and factorization of dense structured matrices. In *Proc. 39th IEEE FOCS*, pages 192–201, 1998. [32]

[ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26:1955–1988, June 2005. [82]

[OS03] V. Olshevsky and M. Amin Shokrollahi. A displacement approach to decoding algebraic codes. In *Contemporary mathematics: theory and applications*, pages 265–292, Boston, MA, USA, 2003. AMS. [25]

[Ott48] Richard Otter. The number of trees. *The Annals of Mathematics*, 49(3):pp. 583–599, 1948. [80, 125]

[Pan66] Victor Y. Pan. Methods of Computing Values of Polynomials. *Russian Mathematical Surveys*, 21(1):105–136, 1966. [80, 138]

[Pan78] Victor Y. Pan. Strassen's algorithm is not optimal: trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 166 –176, oct. 1978. [80]

[Pan89] Victor Y. Pan. On some computations with dense structured matrices. In *Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation*, ISSAC '89, pages 34–42, New York, NY, USA, 1989. ACM. [15, 34]

[Pan92a] Victor Y. Pan. Parallel solution of Toeplitz-like linear systems. *Journal of Complexity*, 8(1):1–21, 1992. [25]

[Pan92b] Victor Y. Pan. Parametrization of Newton's iteration for computations with structured matrices and applications. *Computers Math. Applic.*, 24(3):61–75, 1992. [25]

[Pan93] Victor Y. Pan. Decreasing the displacement rank of a matrix. *SIAM J. Matrix Anal. Appl.*, 14(1):118–121, 1993. [25]

[Pan00] Victor Y. Pan. Nearly optimal computations with structured matrices. In *SODA'00*, pages 953–962. ACM, 2000. [23, 32, 52]

[Pan01] Victor Y. Pan. *Structured Matrices and Polynomials.* Birkhäuser Boston Inc., 2001. [9, 11, 13, 15, 18, 19, 20, 21, 22, 24, 25, 28, 32, 38, 39, 40, 45, 56, 57, 61]

[PFV11] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011. [74]

[PGM04] Sylvie Putot, Eric Goubault, and Matthieu Martel. Static analysis-based validation of floating-point computations. In *Novel Approaches to Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 295–312, 2004. [82, 84]

[Piv08] Carine Pivoteau. *Génération aléatoire de structures combinatoires : méthode de Boltzmann effective.* PhD thesis, LIP6, December 2008. [80, 120]

[PMJ$^+$05] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005. [74]

[PS73] Mike S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evalutate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. [5, 80, 87, 91, 165, 174, 175, 176, 177]

[PS00] Juan Manuel Peña and Thomas Sauer. On the multivariate Horner scheme. *SIAM Journal on Num. Analysis*, 37(4):1186–1197, 2000. [85]

[PSS08] Carine Pivoteau, Bruno Salvy, and Michèle Soria. Boltzmann oracle for combinatorial systems. In *Algorithms, Trees, Combinatorics and Probabilities*, page 475–488. Discrete Mathematics and Theoretical Computer Science, 2008. Proceedings of the Fifth Colloquium on Mathematics and Computer Science. Blaubeuren, Germany. September 22-26, 2008. [80, 120]

[PW02] Victor Y. Pan and Xinmao Wang. Inversion of displacement operators. *SIAM J. Matrix Anal. Appl.*, 24:660–677, March 2002. [19, 20]

[PZ00] Victor Y. Pan and Ailong Zheng. Superfast algorithms for Cauchy-like matrix computations and extensions. *Linear Algebra Appl.*, 310:83–108, 2000. [25, 48]

[Rev06] Guillaume Revy. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Master's thesis, École normale supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2006. [82]

[Rev09] Guillaume Revy. *Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation.* PhD thesis, Univ. de Lyon - ÉNS Lyon, December 2009. [3, 82, 86, 87, 104, 123, 165, 167]

[RR05] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005. [170]

[Rum09] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502, 2009. [82]

[Sta99] Richard P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University press, 1999. [125]

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969. [10]

[vzGG03] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, second edition, 2003. [9, 10, 11, 12, 13, 54, 64, 68, 69]

[vzGS92] J. von zur Gathen and V. Shoup. Computing Frobenius maps and factoring polynomials. *Comput. Complexity*, 2(3):187–224, 1992. [15]

[Wed22] Joseph Wedderburn. The functional equation $g(x^2) = 2\alpha x + [g(x)]^2$. *The Annals of Mathematics*, 24(2):121–140, 1922. [80, 126]

[Yat09] Randy Yates. *Fixed-Point Arithmetic: An Introduction.* Digital Signal Labs, 2009. [81, 83]

Abstract

Designing efficient code in practice for a given computation is a hard task. In this thesis, we tackle this issue in two different situations.

The first part of the thesis introduces some algorithmic improvements in structured linear algebra. We first show how to extend an algorithm by Cardinal for inverting Cauchy-like matrices to the other common structures. This approach, which mainly relies on products of the type "structured matrix × matrix", leads to a theoretical speed-up of a factor up to 7 that we also observe in practice. Then, we extend some works on Toeplitz-like matrices and prove that, for any of the common structures, the product of an $n \times n$ structured matrix of displacement rank α by an $n \times \alpha$ matrix can be computed in $\tilde{O}\big(\alpha^{\omega-1}\big)$. This leads to direct inversion algorithms in $\tilde{O}\big(\alpha^{\omega-1}\big)$, that do not rely on a reduction to the Toeplitz-like case.

The second part of the thesis deals with the implementation of arithmetic expressions. This topic raises sereval issues like finding the minimum number of operations, and maximizing the speed or the accuracy when using some finite-precision arithmetic. Making use of the inductive nature of arithmetic expressions enables the design of algorithms that help to answer such questions. We thus present a set of algorithms for generating evaluation schemes, counting them, and optimizing them according to one or several criteria. These algorithms are part of a library that we have developed and used, among other things, in order to decrease the running time of a code generator for a mathematical library, and to study optimality issues about the evaluation of a small degree polynomial with scalar coefficients at a matrix point.

Keywords: structured linear algebra, matrix product, matrix inversion, arithmetic expressions, code generation, combinatorics and optimization of evaluation schemes.

Résumé

Le développement de code efficace en pratique pour effectuer un calcul donné est un problème difficile. Cette thèse présente deux situations où nous avons été confronté à ce problème.

La première partie de la thèse propose des améliorations au niveau algorithmique dans le cadre de l'algèbre linéaire structurée. Nous montrons d'abord comment étendre un algorithme de Cardinal pour l'inversion de matrices de type Cauchy afin de traiter les autres structures classiques. Cette approche, qui repose essentiellement sur des produits de type « matrice structurée × matrice », conduit à une accélération d'un facteur allant jusqu'à 7 en théorie et constaté en pratique. Ensuite, nous généralisons des travaux sur les matrices de type Toeplitz afin de montrer comment, pour les structures classiques, calculer le produit d'une matrice structurée $n \times n$ et de rang de déplacement α par une matrice $n \times \alpha$ en $\tilde{O}\big(\alpha^{\omega-1}n\big)$. Cela conduit à des algorithmes en $\tilde{O}\big(\alpha^{\omega-1}n\big)$ pour l'inversion de matrices structurées, sans avoir à passer par des matrices de type Toeplitz.

La deuxième partie de la thèse traite de l'implantation d'expressions arithmétiques. Ce sujet soulève de nombreuses questions comme le nombre d'opérations minimum, la vitesse, ou encore la précision des calculs en arithmétique approchée. En exploitant la nature inductive des expressions arithmétiques, il est possible de développer des algorithmes aidant à répondre à ces questions. Nous présentons ainsi plusieurs algorithmes de génération de schémas d'évaluation, de comptage et d'optimisation selon un ou plusieurs critères. Ces algorithmes ont été implanté dans une librairie qui a en autre été utilisée pour accélérer un logiciel de génération de code pour une librairie mathématique, et pour étudier des questions d'optimalité pour le problème de l'évaluation d'un polynôme à coefficients scalaires de petit degré en une matrice.

Mots-clés : algèbre linéaire structurée, produit de matrices, inversion de matrices, expressions arithmétiques, génération de code, combinatoire et optimisation des schémas d'évaluation.