Western University

Scholarship@Western

Digitized Theses

Digitized Special Collections

2009

High-performance code generation for polynomials and power series

Ling Ding

Follow this and additional works at: https://ir.lib.uwo.ca/digitizedtheses

Recommended Citation

Ding, Ling, "High-performance code generation for polynomials and power series" (2009). *Digitized Theses*. 4105.
https://ir.lib.uwo.ca/digitizedtheses/4105

This Thesis is brought to you for free and open access by the Digitized Special Collections at Scholarship@Western. It has been accepted for inclusion in Digitized Theses by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

High-performance code generation for polynomials and power series

(Spine Title: Code generation for polynomials and power series)

(Thesis format: Monograph)

by

Ling Ding

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Ling Ding 2009

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Éric Schost	Dr. Mahmoud El-Sakka
	Dr. Marc Moreno Maza
	Dr. David Jeffrey

The thesis by

Ling Ding

entitled:

High-performance code generation for polynomials and power series

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date	Chair of the Thesis Examination Board Dr. Mark Daley

Abstract

Newton iteration is a versatile tool. In this thesis, we investigate its applications to the computation of power series solutions of first-order non-linear differential equations.

To speed-up such computations, we first focus on improving polynomial multiplication and its variants: plain multiplication, transposed multiplication and short multiplication, for Karatsuba's algorithm and its generalizations. Instead of rewriting code for different multiplication algorithms, a general approach is designed to output computer-generated code based on multiplication graph representations.

Next, we investigate the existing Newton iteration algorithms for differential equation solving problems. To improve their efficiency, we recall how one can reduce the amount of useless computations by using transposed multiplication and short multiplication. We provide an optimized code generator that applies these techniques automatically to a given differential equation.

Keywords: Differential Equation, Newton Iteration, Plain Multiplication, Transposed Multiplication, Short Multiplication, Automatic Code Generation.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Éric Schost for his guidance, support and encouragement. He impressed me with his deep knowledge in computer algebra and his confidence and ability to solve any kind of difficulties.

During my study, I took many excellent courses related to computer algebra, and also some fundamental courses in computer science. I would like to thank these professors, Dr. Marc Moreno Maza, Dr. Stephen Watt and many other professors. Moreover, I would like to thank all the ORCCA lab members. They provided me great advice on my thesis.

Last but not least, I wish to thank my parents, without whose endless support and encouragement, my studies leading to the Master degree of science would not have been possible.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Symbolic computation is a process using computer systems to manipulate mathematical equations and expressions in symbolic form. It is widely used for symbolic integration or differentiation, substitution of one expression into another, simplification of an expression, etc.

In this thesis, we focus on one classical application of symbolic computation: computing symbolic solutions to differential equations, by means of Newton iteration. Solving differential equations symbolically is not a new topic: some computer algebra systems, such as Maple, already have built-in functions to solve this problem. Similarly, using Newton iteration for this task is now a standard idea.

This thesis contributes to an aspect of this problem which has attracted little attention up to now: the design of high-performance implementations, in the particular case of first-order non-linear differential equations, with coefficients in rings such as $\mathbb{Z}/p\mathbb{Z}$. As an intermediate step, we study various algorithms for polynomial multiplication and related problems, under the high-performance viewpoint as well.

One of the main contributions of this thesis is the focus on *code generation techniques.* Rather than implementing C code from scratch by ourselves, we wrote Java programs that generate high-performance C implementations for our two main problems, polynomial multiplication and Newton iteration for differential equations. We expect that the experience acquired here can be applied to several other problems.

1.2 Background

One of the key ideas in the design of fast algorithms in computer algebra is to reduce complex problems to a few well-studied basic questions.

The algorithms described in this thesis illustrate this rule. The main operation we will consider is *polynomial multiplication*, with coefficients in a ring R. To fix notation, we will write the input polynomials as

$$A = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}, \quad B = b_0 + b_1x + \cdots + b_{m-1}x^{m-1};$$

note that the *number of terms* of A (resp. B) is at most n (resp. m). In many situations, we will actually have $n = m$. Our first objective is to compute the $m+n-1$ coefficients of the product

$$C = AB = c_0 + c_1x + \cdots + c_{m+n-2}x^{m+n-2},$$

with

$$c_i = \sum_{j+k=i,\ 0 \le j < n,\ 0 \le k < m} a_j b_k.$$

The naive algorithm uses $O(mn)$ ring operations to compute all coefficients of C, for an output of length $m+n-1$. This is far from optimal, especially when $n = m$. Several techniques are known to reduce this cost, using the idea of *divide-and-conquer*: split the problem into several parts, process them recursively, and recombine the partial results to obtain the desired output.

A well-known algorithm applying this idea is known as Karatsuba's multiplication [22]. It will be studied in detail in Chapter 3; we illustrate here the main idea of this trick by a simple example. Assume $n = m = 2^k$ for some $k \in \mathbb{N}$ and rewrite A, B in the form

$$A = A_0 + A_1x^{n/2}, \quad B = B_0 + B_1x^{n/2},$$

with A_0, A_1, B_0, $B_1 \in R[x]$ of degrees less than $n/2$. If we follow the naive method, we are led to write

$$AB = A_0B_0 + (A_0B_1 + A_1B_0)x^{n/2} + A_1B_1x^n.$$

Karatsuba's trick amounts to rewrite the product as

$$AB = A_0B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1)x^{n/2} + A_1B_1x^n.$$

The number of polynomial multiplications of degree less than $n/2$ has been reduced from four to *three*. Applying this idea recursively gives a complexity $O(n^{\log_2(3)})$ for multiplication.

In fact, several such algorithms exist, with different complexities. As a consequence, it is customary to use the notation $\mathsf{M}(n)$ to denote the complexity of one's favorite algorithm for polynomial multiplication: the naive algorithm has $\mathsf{M}(n) = O(n^2)$; Karatsuba's has $\mathsf{M}(n) = O(n^{\log_2(3)})$; a generalization of it by Toom [38] has $\mathsf{M}(n) = O(n^{\log_3(5)})$, and some other algorithms not described in this thesis have $\mathsf{M}(n) = O(n \log(n))$ or $\mathsf{M}(n) = O(n \log(n) \log\log(n))$, using Fast Fourier Transform (FFT) [33, 11]. As of now, no algorithm of linear complexity is known.

Many higher level algorithms are built on top of multiplication, using techniques such as Newton iteration or divide-and-conquer.

- The former idea, Newton iteration, is the key to algorithms for differential equations presented later, among many others. As a general rule, an algorithm based on Newton iteration usually has cost $O(\mathsf{M}(n))$ for an output of length n (this is indeed the case for differential equations).

- Divide-and-conquer techniques are used in other families of algorithms, such as fast Euclidean algorithm or evaluation and interpolation algorithms. Usually, such algorithms have costs $O(\mathsf{M}(n) \log(n))$ or $O(\mathsf{M}(n))$, depending on M.

As was said before, our focus will be on Newton iteration techniques; we will not discuss algorithms such as fast Euclidean algorithm of fast evaluation / interpolation.

1.3 Related work

In accordance with the general remarks of the last section, it turns out that in most fast algorithms for polynomials or power series, the main computation is multiplication. This is also the key to our research: since we pursue high performance, improving the multiplication becomes the first task.

Polynomial multiplication has been the subject of a large body of work; as of now, several libraries (NTL [35], FLINT [20], modpn [24, 14, 25]) or computer algebra systems (Magma [4]) provide implementations of algorithms such as Karatsuba's or the FFT.

Compared to previous work such as Li *et.al.* [24] and Filatei *et.al.* [14], this thesis focuses on the family of so-called "divide-and-conquer" algorithms, that generalize

Karatsuba's, as opposed to the FFT. Besides, while we reuse insights of NTL [35], Li *et.al.* [24], Filatei *et.al.* [14] for e.g. fast computations modulo a prime number p, we have not put our focus (yet) on a systematic exploration of optimization techniques such as cache-friendliness or parallelism.

Our goal was to build a platform with which one can generate automatically efficient implementation of several variants of polynomial multiplication, such as *transposed products* or *short products.* These two concepts are studied in detail later on. For the moment, we simply point out that are known to speed-up various forms of Newton iteration; the former idea (transposed multiplication) was studied in Hanrot, *et.al.* [17], and the latter (short multiplication) in Mulders [28] and Hanrot, *et.al.* [19].

After polynomial multiplication, fast algorithms to solve differential equations are the second significant focus in our research. Software packages such as [21] by Jorba, *et.al.* , already exist to treat such problems, but from the point of view of numerical integration. Though the context is different, the article [21] introduced a useful discussion about automatic differentiation and code generation issues, and partly motivated our study.

In the symbolic world, a main reference regarding the computation of power series solution of differential equations is Brent and Kung's paper [9]: the approach introduced in that paper is still used today, and is the basis of our work. To compute n terms of the solution, this algorithm uses $O(\mathsf{M}(n))$ operations: it is as fast as multiplication, up to a constant factor.

The paper Bostan, *et.al.* [7] describes an application of Brent and Kung's result to a problem from cryptology: this question boils down to finding a power series solution of an equation of the form

$$(x^6 + x^4 + 1)f'(x)^2 = 1 + 75f(x)^4 + 16f(x)^6, \quad f(0) = 0, \quad f'(0) = 1.$$

One remarkable feature is that the problem treated in [7] requires to solve this equation over a finite field. For instance, with coefficients in $\mathbb{Z}/101\mathbb{Z}$ (*i.e.*, taken modulo 101), the first few terms of the solution are

$$f = x + 68x^5 + 66x^7 + 60x^9 + 84x^{11} + \cdots .$$

Another noteworthy fact is that the application in the paper of Bostan, *et.al.* [7]

needs to compute a few hundreds coefficients of f, say about 500. Finally, high-performance is crucial for real-world applications.

These remarks have determined the context of our study: high-performance computations with coefficients modulo a prime number (even though floating point coefficients will be supported as well). Besides, they motivated our interest for divide-and-conquer univariant multiplication algorithms such as Karatsuba's, which can outperform FFT for moderate degrees, such as 100 to 500. Similarly to multiplication, our goal is to automatically generate efficient code.

Let us briefly mention other related results. Brent and Kung focused on first-order equations, and this was sufficient to deal with the problem of [7]. More recent results are in the papers of van der Hoeven [39] and Bostan,*et.al.* [5], which focus in particular on higher order equations. Though some of the techniques developed here will apply in that context, we do not discuss such extensions.

Recasting differential equations as fixed point problems makes it possible to obtain compact and efficient code. This idea was introduced by Watt [44], with algorithms of cost $O(n^2)$; van der Hoeven [39] used a similar idea in conjunction with fast "relaxed multiplication" to obtain a cost of $O(\mathsf{M}(n)\log(n))$ or $O(\mathsf{M}(n))$, depending on the assumptions we make on M. It would be interesting to compare experimental results obtained in that approach to ours.

Finally, one should mention the work of the Spiral group on efficient code generation for numerical FFT computations by Püschel, *et.al.* [29]. This work served as an inspiration for our own code generation techniques. We hope to adapt the code optimization techniques introduced in [29] to our context in the future.

1.4 Results

The main results of this thesis are the following.

Code generation for polynomial multiplication. We already mentioned that there exist families of divide-and-conquer multiplication algorithms that generalize Karatsuba's. Besides, several variants of each of these algorithms are used in Newton iteration algorithms, such as transposed product or short product.

We propose to free the programmer from the tedious work on writing code for all such variants. Indeed, it is well-known that these divide-and-conquer algorithms can be specified by graphical representations. We provide a Java platform that uses this representation as input and generates C code for operations such as plain, transposed or short products for any given divide-and-conquer algorithm. As of now, our code generator supports `double` and `unsigned long` data types.

The output code is aimed towards high-performance. While several directions for optimization remain to be explored, the results are already very encouraging, as we outperform most other libraries or systems for degrees up to 1000.

Removing redundancies in Newton iteration. Avoiding the computation or re-computation of useless coefficients is a key to improve the performance of Newton iteration; however, the Newton iteration for differential equations in Brent's paper [9] pays little attention to these issues. This question has been the subject of many later papers [32, 2, 18, 41], but they all focus on FFT multiplication, and on the applications of Newton iteration to other questions, such as inverses or exponentials.

We revisited Brent and Kung's algorithm and paid extra effort to eliminate redundant computations, by means of transposed or short products, or by eliminating useless nested loops. The gain is a constant, but significant, factor in running time. We focused on the technically simplest case of equations of the first order: this case already reveals many interesting problems, and is sufficient to cover many applications, such as the one from Bostan, *et.al.* [7] mentioned before. It is expected that many of our results extend to higher order.

Fast evaluation for Newton iteration. The work described in the previous paragraph addresses a large part of the operations done within Newton iteration algorithms; yet, some parts can still be improved.

When we solve an equation such as $G(x, f', f) = 0$, we need to evaluate G and its derivatives at (x, f', f), where f is the current approximation to the solution. Our last question is to improve this evaluation. Here as well, techniques such as transposed product and short product can be used to accelerate the operation. As for polynomial multiplication, our aim is to automatize this process.

Given a representation of the equation $G(x, f', f)$ as an expression tree (or rather, as a directed acyclic graph), we show how to automatically perform such optimizations. We focus on the multiplication nodes, which have most effect on the cost. Then, we determine what coefficients are needed for future computations, what coefficients are unnecessary, and what kind of multiplication should be used: beyond transposed and short product, we are led to introduce more variants, such as "short-long product" or "quarter product".

As before, we provide a Java platform that takes as input an expression that computes G, builds the corresponding directed acyclic graph, and outputs C code to use within Newton iteration.

1.5 Outline

In Chapter 2, we provide an overview of the graphical data structures related to this research: graph representations for linear expressions and for polynomial expressions.

Chapter 3 and 4 focus on polynomial multiplication. In Chapter 3, we review the background knowledge about fast algorithms for plain multiplication, and its variants: transposed multiplication and short multiplication; we write down general forms of transposed and short multiplication algorithms. In Chapter 4, we present our work on code generation for polynomial multiplication and give the results of experiments made using our multiplication code.

In Chapters 5, 6 and 7, we focus on improving the Newton iteration for solving the differential equations $G(x, f, f') = 0$. Chapter 5 introduces the current fast algorithms to solve differential equations and explains basic functions. In Chapter 6, we show how to remove redundancies in the main loop of Newton's iteration, using the variants of polynomial multiplication showed in Chapter 3. Chapter 7 shows how to apply this idea within the evaluation of the function G, to reduce the amount of unnecessary computations. We demonstrate another code generator that deals with this problem.

Chapter 2

Graphical data structures

In this chapter, we present two useful data structures: one to perform *linear* operations, to be used in Chapters 3 and 4, and the other to evaluate *polynomial expressions*, used in Chapter 7. The underlying objects are graphs in both cases, but the ways we use them are different. In all this chapter, we fix a coefficient ring R; all computations will involve coefficients in R.

Note that the material presented here is well-known: see [10] by Bürgisser *et.al.* for linear graphs and [1] by Aho *et.al.* for graphs computing polynomial expressions.

2.1 Graphs for linear expressions

This section describes a computational model for doing the following kind of operations: given $a_0, \dots, a_{k-1}$, we want to compute some linear combinations

$$\begin{aligned} f_0 &= L_{0,0}a_0 + \cdots + L_{0,k-1}a_{k-1} \\ &\vdots \\ f_{\ell-1} &= L_{\ell-1,0}a_0 + \cdots + L_{\ell-1,k-1}a_{k-1}. \end{aligned}$$

In other words, we want to compute the matrix-vector product

$$\begin{bmatrix} f_0 \\ \vdots \\ f_{\ell-1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & \cdots & L_{0,k-1} \\ \vdots & & \vdots \\ L_{\ell-1,0} & \cdots & L_{\ell-1,k-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{k-1} \end{bmatrix}$$

This will be done using *linear graphs*.

Definition. Linear graphs are graphic representation for the computation of linear operations as above. A linear graph is a quintuple $\mathcal{G} = (V, E, I, O, \lambda)$ consisting of

- a directed acyclic graph (V, E),
- a weight function λ which assigns a weight $\lambda(e) \in R$ to each edge e,
- an ordering $(A_0, \cdots, A_{k-1})$ of the input vertices, I,
- an ordering $(F_0, \cdots, F_{\ell-1})$ of the output vertices, O.

A linear graph can be used to compute "linear combinations of its inputs". Starting from the input nodes, each further vertex is assigned a value, obtained by following the flow from inputs to outputs. Going from a vertex v to a vertex v' along an edge e, the value at v is multiplied by the weight $\lambda(e)$; the value at the vertex v' is obtained by summing the contributions of all incoming edges.

One sees that the values obtained at each vertex are linear combinations of the values $a_0, \ldots, a_{k-1}$ given at the inputs $A_0, \cdots, A_{k-1}$. In particular, let $f_0, \ldots, f_{\ell-1}$ be the values computed by the output nodes $F_0, \cdots, F_{\ell-1}$. Each f_i can thus be written

$$f_i = L_{i,0}a_0 + \cdots + L_{i,k-1}a_{k-1},$$

for some constants $L_{i,j}$, so that we have

$$\begin{bmatrix} f_0 \\ \vdots \\ f_{\ell-1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & \cdots & L_{0,k-1} \\ \vdots & & \vdots \\ L_{\ell-1,0} & \cdots & L_{\ell-1,k-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{k-1} \end{bmatrix}$$

Thus, we say that the linear graph $\mathcal{G}$ *computes* the matrix

$$\begin{bmatrix} L_{0,0} & \cdots & L_{0,k-1} \\ \vdots & & \vdots \\ L_{\ell-1,0} & \cdots & L_{\ell-1,k-1} \end{bmatrix}.$$

Example. Figure 2.1 gives an example with input nodes A_0, A_1, A_2 and output nodes F_0, F_1, F_2. The values a_0, a_1, a_2 are input to the nodes A_0, A_1, A_2; the values f_0, f_1, f_2 are output by the nodes F_0, F_1, F_2. It is not hard to check that the linear

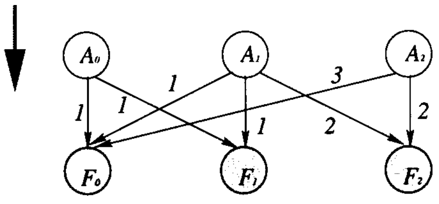

Figure 2.1: The linear graph $\mathcal{G}_0$

graph $\mathcal{G}_0$ of the figure computes the following matrix-vector product:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 1 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

Another issue we should point out: several linear graphs can compute the *same* families of outputs. If we modify Figure 2.1 by adding one "tmp" node, we obtain Figure 2.2.

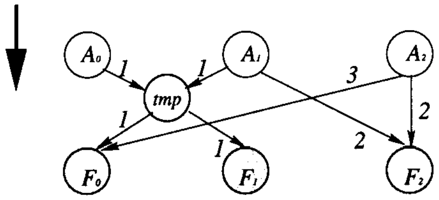

Figure 2.2: The linear graph $\mathcal{G}_1$

The graphs $\mathcal{G}_0$ and $\mathcal{G}_1$ compute the same matrix. However, in $\mathcal{G}_1$, we add an extra node for storing the result $a_0 + a_1$ as a temporary value, which will be reused to compute f_0 and f_1. With the "tmp" node, Figure 2.2 performs 3 additions, while Figure 2.1 has 4. Since Figure 2.2 has fewer operations, we think of it as a "better" graph. For this example, it is easy to find the "better" one; finding the "better" graph in a general situation is however a hard problem, that we will not discuss.

Cost. To measure the number of operations attached to a linear graph, we define a notion of *cost.* The number

$$c(\mathcal{G}) := |e \in E \mid \lambda(e) \neq \pm 1| + |E| - |V| + k$$

is called the cost of $\mathcal{G}$. Here is the idea behind this definition.

- Along the edges, each multiplication by a constant different from ± 1 costs one operation: this gives $|e \in E \mid \lambda(e) \neq \pm 1|$ operations in total.

- If the input of a vertex v consists of s edges $e_1, \ldots, e_s$, computing the value at v (after performing the multiplications on the edges) uses $s-1$ additions; this gives in total
$$\sum_{v \text{ vertex, } v \text{ not an input}} (i(v) - 1)$$
operations, where $i(v)$ is the number of edges entering v. This sum equals $|E| - (|V| - n) = |E| - |V| + n$.

Transposition. We will pay a special attention to the computation of *transposed* operations. To do so, we introduce here the *transposition principle.* This principle asserts that an algorithm performing a matrix-vector product can be transposed, producing an algorithm that computes the transposed matrix-vector product. Further, the transposed algorithm has almost the same complexity as the original one.

Using the linear graph model, the transposition principle is easy to prove. If $\mathcal{G} = (V, E, I, O, \lambda)$ is a linear graph that computes a matrix
$$\begin{bmatrix} L_{0,0} & \cdots & L_{0,k-1} \\ \vdots & & \vdots \\ L_{\ell-1,0} & \cdots & L_{\ell-1,k-1} \end{bmatrix},$$
we define the transposed graph
$$\mathcal{G}^t = (V, E^t, I^t, O^t, \lambda^t) \tag{2.1}$$
of $\mathcal{G}$ by $E^t := \{(w, v) \mid (v, w) \in E\}$, $I^t = O$, $O^t = I$, and $\lambda^t(w, v) := \lambda(v, w)$ for every $(v, w) \in E$. In other words, $\mathcal{G}^t$ results from $\mathcal{G}$ by reversing the arrows without changing the weights. Thus $\mathcal{G}^t$ computes n linear combinations of $f_0^t, \cdots, f_{\ell-1}^t$; it is proved in [10, Th. 13.10] that these combinations are given by
$$\begin{bmatrix} a_0 \\ \vdots \\ a_{k-1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & \cdots & L_{\ell-1,0} \\ \vdots & & \vdots \\ L_{0,k-1} & \cdots & L_{\ell-1,k-1} \end{bmatrix} \begin{bmatrix} f_0 \\ \vdots \\ f_{\ell-1} \end{bmatrix}.$$

In other words, $\mathcal{G}^t$ computes the transpose of the matrix of $\mathcal{G}$. To finish to establish the transposition principle, it remains to consider cost issues. One can prove that the cost $c(\mathcal{G}^t)$ is given by
$$c(\mathcal{G}^t) = c(\mathcal{G}) - k + \ell,$$

so the two costs are the same, up to a difference of $k - \ell$.

If we are in cases where the cost of multiplication weights much more than addition, we can only consider the number of multiplication in the complexity analysis. Then, the costs of computing $\mathcal{G}$ and $\mathcal{G}^t$ are the same.

Example. We are given the graph $\mathcal{G}$ of Figure 2.3, with $k = 3, \ell = 2$; the darkened nodes are output nodes. It computes the matrix

$$\begin{bmatrix} 6 & 2 \\ 3 & 4 \\ 0 & 5 \end{bmatrix}.$$

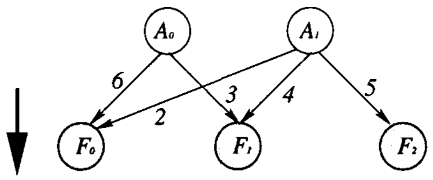

Figure 2.3: A graph $\mathcal{G}$ that computes the original linear combination

The transposed linear graph $\mathcal{G}^t$ is in Figure 2.4. If is obtained by reversing the arrows without changing the weights from $\mathcal{G}$; note that the position of output nodes (darkened) has changed.

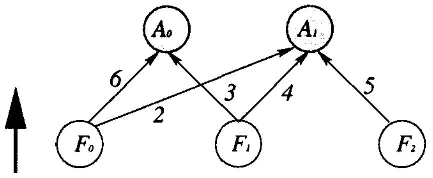

Figure 2.4: The graph $\mathcal{G}^t$ that computes the transposed linear combination

One verifies that $\mathcal{G}^t$ computes the matrix

$$\begin{bmatrix} 6 & 3 & 0 \\ 2 & 4 & 5 \end{bmatrix}.$$

which is the transpose of the original one. Let us then compare the costs of $\mathcal{G}$ and $\mathcal{G}^t$:

- $\mathcal{G}$ computes: $f_0 = 6a_0 + 2a_1$, $f_1 = 3a_0 + 4a_1$, $f_2 = 5a_1$ 5 mul+ 2 add

- $\mathcal{G}^t$ computes: $a_0 = 6f_0 + 3f_1$, $a_1 = 2f_0 + 4f_1 + 5f_2$ 5 mul+ 3 add

Counting the number of operations, the cost of $\mathcal{G}$ is 7, and that of $\mathcal{G}^t$ is 8. The difference between them is 1, which is $k - \ell$ as claimed above.

2.2 Graphs for polynomial expressions

For the differential equation algorithm of Chapter 7, we will have to evaluate a polynomial $G(x, t, u)$, and its derivatives, at $x, t = f, u = f'$, for a given polynomial f in $R[x]$. The polynomial G will be given to us as directed acyclic graph, whose input vertices are t and u, with vertex labelled by arithmetic operations [1].

Definition. We give here a definition adapted to our needs. A *non-linear graph* $\mathcal{G} = (V, E, \mu)$ consists of a directed acyclic graph (V, E), with two input vertices, and of a labelling μ which assigns a label $\mu(v)$ to each vertex v in the graph, such that

- there is one input labelled t and the other one is labelled u;
- for a vertex v with two parents v_1, v_2, $\mu(v)$ is either $(+, v_1, v_2)$, $(\times, v_1, v_2)$, $(-, v_1, v_2)$, $(-, v_2, v_1)$;
- for a vertex v with one parent v_1, $\mu(v)$ is either $(+, v_1, a(x))$, $(-, v_1, a(x))$, $(-, a(x), v_1)$, $(\times, a(x), v_1)$, where $a(x)$ is a polynomial with coefficients in R, or $\wedge 2$ (the square operation).

One can assign inductively a polynomial P_v in $R[x, t, u]$ to each vertex v of $\mathcal{G}$: the input vertices are assigned respectively t and u; then, a product node is assigned the product of the polynomials at the parent nodes, etc. Then, we say that $\mathcal{G}$ *computes* the polynomials assigned to the output nodes.

As a convention, we let L be the number of multiplication nodes in the graph, either of the form $(\times, v_1, v_2)$ or $\wedge 2$ (we count squares as normal multiplications).

On the example of Figure 2.5, the very top node V_0 is the output of the graph; t and u are inputs; internal nodes are operators. $(\times, 5, u)$ means multiplying u by the constant 5. Here, the root computes $t^2 - 5u + u^2$. The number of multiplications is $L = 2$.

Differentiation. We will have to compute not only the value of G at $x, t = f, u = f'$, but also the values of the derivatives $\partial G/\partial u$ and $\partial G/\partial t$ at $x, t = f, u = f'$.

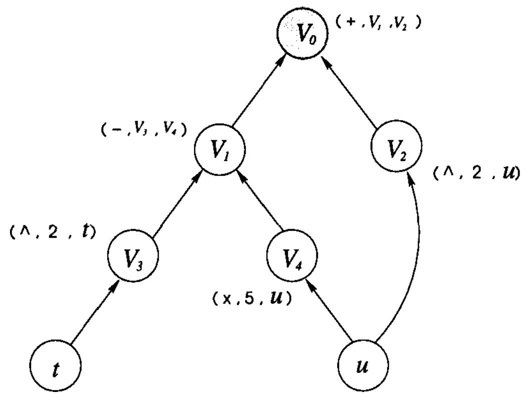

Figure 2.5: A standard graph that computes a polynomial expression

To compute the derivatives, we apply automatic differentiation techniques [30]: we construct an extended graph $\mathcal{G}'$ that computes not only G but also $\partial G/\partial u$ and $\partial G/\partial t$.

In $\mathcal{G}'$ each node is replaced by a triple $C, \partial C/\partial u, \partial C/\partial t$, plus maybe a few extra nodes. To perform this construction, we simply apply the rules for differentiating sums, products, etc. For instance, for a multiplication node, which computes a product $C(x,u,t) = A(x,u,t)B(x,u,t)$, there are two situations:

- A and B are the same polynomials: we rewrite it as $C(x,u,t) = A(x,u,t)^2$, so that
$$\frac{\partial C}{\partial u} = 2A\frac{\partial A}{\partial u}, \quad \frac{\partial C}{\partial t} = 2A\frac{\partial A}{\partial t}.$$
- A and B are different polynomials; then
$$\frac{\partial C}{\partial u} = A\frac{\partial B}{\partial u} + \frac{\partial A}{\partial u}B, \quad \frac{\partial C}{\partial t} = A\frac{\partial B}{\partial t} + \frac{\partial A}{\partial t}B;$$
remark that 4 multiplications are performed to compute both derivatives of C.

For an addition node, which computes $C(x,u,t) = A(x,u,t) + B(x,u,t)$, we get its derivatives directly from
$$\frac{\partial C}{\partial u} = \frac{\partial A}{\partial u} + \frac{\partial B}{\partial u}, \quad \frac{\partial C}{\partial t} = \frac{\partial A}{\partial t} + \frac{\partial B}{\partial t}.$$
The same process is extended easily for all kinds of operations introduced before.

As before, we let L be the number of multiplication nodes in the original graph $\mathcal{G}$. Since we do not generate extra multiplications from computing derivatives of addition or subtraction nodes, and since we generate at most 4 multiplications through the differentiation of a multiplication, the number of multiplication nodes in the new graph $\mathcal{G}'$ is at most $L + 4L = 5L$.

To give an example, we start by simplifying Figure 2.5 to obtain Figure 2.6, by keeping only the node indices and operator notations.

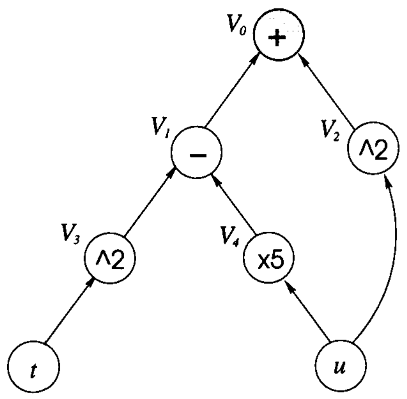

Figure 2.6: A simplified graph for a polynomial expression

Through traversing the graph from bottom to up and applying the automatic differentiation rules it, we compute V_0, $\partial V_0/\partial u$ and $\partial V_0/\partial t$, which is shown as Figure 2.7.

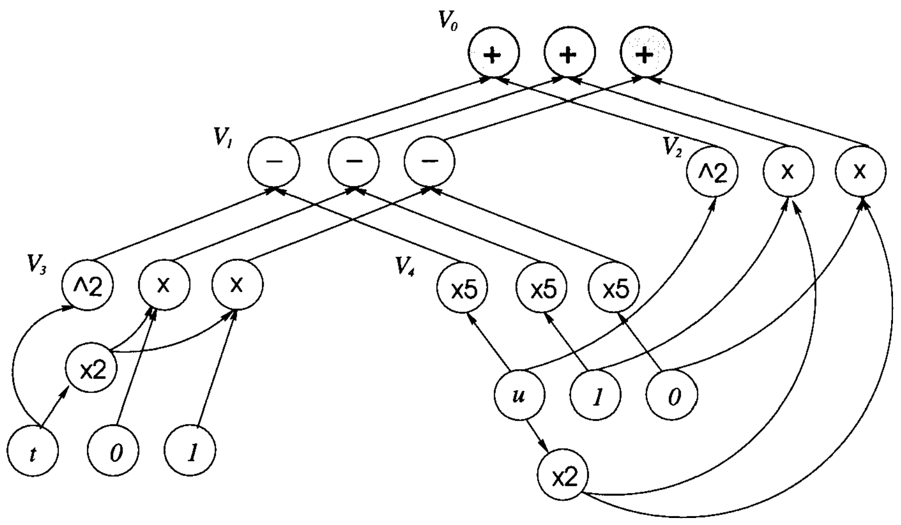

Figure 2.7: A graph for a polynomial expression and its derivatives

In Figure 2.7, each node from Figure 2.6 splits into three nodes. Take the leaf u for example: we get three nodes instead of one, u, $\partial u/\partial u$ and $\partial u/\partial t$. Their values are u, 1 and 0. We also list the result in Table 2.1.

node	operator	G	G_u	G_t
u(input)	N/A	u	1	0
t(input)	N/A	t	0	1
v_4	$\times 5$	$5u$	5	0
v_3	$\wedge 2$	t^2	0	$2t$
v_2	$\wedge 2$	u^2	$2u$	0
v_1	$-$	$t^2 - 5u$	-5	$2t$
v_0(output)	$+$	$t^2 + u^2 - 5u$	$2u - 5$	$2t$

Table 2.1: Triple value set for tree nodes

Chapter 3

Polynomial multiplication

In this chapter, we describe a key ingredient of this thesis, polynomial multiplication. We introduce three "forms" of multiplication and present several algorithms with their complexity, but we do not consider implementation issues for the moment.

Recall that on input

$$A = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}, \quad B = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1},$$

our first objective is to compute the $2n-1$ coefficients of the product

$$C = AB = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2},$$

with

$$c_i = \sum_{j+k=i,\ 0 \le j < n,\ 0 \le k < n} a_j b_k. \tag{3.1}$$

This operation will be called *plain* multiplication. Some useful variants of it will be considered as well:

- the *transposed* multiplication, or transposed product, defined in Section 3.2;
- the *short* multiplication, or short product, where only the coefficients $c_0, \ldots, c_{n-1}$ are computed.

All algorithms for the plain multiplication given here are well-known. The contribution of this chapter is to present general descriptions of the transposed and short multiplication. While it was known before that a transposed multiplication algorithm could in theory be designed from any plain multiplication algorithm [17, 6], no explicit description of the general process appears. Similarly, for the short multiplication, no description of a general algorithm appears, only a study of the simplest case [28, 19].

The following notation will be useful: the remainder of the Euclidean division of a polynomial A by a polynomial B will be denoted by $A \bmod B$, and the quotient will be denoted by $A \operatorname{div} B$; later on, a similar notation will be used for quotient and remainder for integer division.

A useful particular case is when B has the form x^n. Then $A \bmod x^n$ is obtained by discarding all monomials of A of degree greater than or equal to n; $A \operatorname{div} x^n$ is obtained by discarding all monomials of degree less than n, and by dividing all remaining ones by x^n.

As was said in the introduction, we use $\mathsf{M}(n)$ to denote the cost of the plain multiplication: we will see here algorithms having $\mathsf{M}(n) = O(n^{\log_2(3)})$, $\mathsf{M}(n) = O(n^{\log_3(5)})$, etc. The cost of the other operations mentioned above will be expressed similarly: we will see that the transposed multiplication has cost $\mathsf{M}(n) + O(n)$; the cost of the short multiplication will be denoted $\mathsf{m}(n)$. These notations are summarized in Table 3.1: the middle column gives the mathematical formula to denote a given operation, and the last column recalls corresponding complexity notation.

operation	notation	complexity notation
multiplication	$A, B \mapsto AB$	$\mathsf{M}(n)$ for $\deg(A) < n$, $\deg(B) < n$
transposed multiplication	$A, B \mapsto AB^t$	$\mathsf{M}(n) + O(n)$ for $\deg(A) < 2n-1$, $\deg(B) < n$
short multiplication	$A, B \mapsto AB \bmod x^n$	$\mathsf{m}(n)$ for $\deg(A) < n$, $\deg(B) < n$

Table 3.1: Notation for various kinds of multiplications

3.1 Plain multiplication

In this section, we review several algorithms for the plain multiplication of polynomials.

The naive algorithm. The schoolbook method, or *naive* method for computing a product $C = AB$ takes a quadratic number of operations. The naive algorithm simply computes the coefficients c_i of Equation (3.1) one after the other. The total cost is $O(n^2)$; in other words, for the naive method, we have $\mathsf{M}(n) = O(n^2)$.

Karatsuba's algorithm. The first algorithm with a better complexity is due to Karatsuba. It uses a divide-and-conquer approach:

- devise an algorithm performing 3 multiplications instead of 4 for the case of polynomials of degree 1;
- apply it recursively.

To deal with polynomials of degree 1 such as $A = a_0 + a_1x$ and $B = b_0 + b_1x$, we rewrite their product as

$$AB = a_0b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)x + a_1b_1x^2. \tag{3.2}$$

This leads to Algorithm 1.

Algorithm 1 Karatsuba's multiplication in degree less than 2

```
Input: $A = a_0 + a_1x$ and $B = b_0 + b_1x$
Output: $C = AB$.
1: compute $N_0 = a_0b_0$, $N_1 = (a_0 + a_1)(b_0 + b_1)$ and $N_2 = a_1b_1$
2: return $N_0 + (N_1 - N_0 - N_2)x + N_2x^2$.
```

To apply this idea recursively, given A and B of degrees less than n, we now write

$$h = \lfloor (n+1)/2 \rfloor, \quad h' = n - h, \quad A = A_0 + A_1x^h, \quad B = B_0 + B_1x^h. \tag{3.3}$$

The choice of h (which is approximately equal to $n/2$) implies

$$\deg(A_0) < h, \quad \deg(A_1) < h' \le h, \quad \deg(B_0) < h, \quad \deg(B_1) < h' \le h.$$

Formula (3.2) still holds, and gives

$$AB = A_0B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1)x^h + A_1B_1x^{2h}.$$

This gives the recursive Algorithm 2. The algorithm performs 3 recursive calls in size at most $(n+1)/2$, plus a number of extra operations (additions, subtractions) that is linear in n; we deduce using the Master theorem [12] that its complexity is $O(n^{\log_2(3)})$. In other words, for the Karatsuba method, we have $\mathsf{M}(n) = O(n^{\log_2(3)})$.

Algorithm 2 KaraMul(A, B, n), Karatsuba's multiplication

```
Input: $A, B, n$, with $\deg(A) < n$ and $\deg(B) < n$, where $h$ and $h'$ satisfy Eq.(3.3)
Output: $C = AB$
1: if $n = 1$ then
2:   return $AB$
3: end if
4: $A = A_0 + A_1x^h, B = B_0 + B_1x^h$
5: $N_0 = \mathsf{KaraMul}(A_0, B_0, h)$
6: $N_1 = \mathsf{KaraMul}(A_0 + A_1, B_0 + B_1, h)$
7: $N_2 = \mathsf{KaraMul}(A_1, B_1, h')$
8: return $N_0 + (N_1 - N_0 - N_2)x^h + N_2x^{2h}$.
```

Note the following easy points, which will help us generalize this algorithm.

- The polynomials A_0, B_0 and $(A_0 + A_1), (B_0 + B_1)$ have degree less than h; the polynomials A_1, B_1 have degree less than h'.
- The polynomials N_0 and N_1 have degree less than $2h - 1$, the polynomial N_2 has degree less than $2h' - 1$.
- The polynomials $C_0 = N_0$, $C_1 = N_1 - N_0 - N_2$ and $C_2 = N_2$ have degrees less than respectively $2h - 1$, $h + h' - 1$ and $2h' - 1$.
- For $h = 1$, the computation of $N_0 + (N_1 - N_0 - N_2)x + N_2x^2$ requires only the computation of $N_1 - N_0 - N_2$. In general, some extra additions are needed due to the overlaps between N_0, $(N_1 - N_0 - N_2)x^h$ and N_2x^{2h}.

Divide-and-conquer algorithms. It is possible to generalize Karatsuba's idea. We will call *divide-and-conquer* algorithm of parameters (k, ℓ) a multiplication algorithm that does the following:

- devise an algorithm performing ℓ multiplications for polynomials with degree less than k;
- apply it recursively.

We must explain more precisely the operations we allow for the case of polynomials with degree less than k, and how we perform the recursive calls.

First, we impose that on input polynomials A and B with degree less than k, the algorithm proceeds as follows.

Algorithm 3 Multiplication in degree less than k

```
Input: $A = a_0 + \cdots + a_{k-1}x^{k-1}$ and $B = b_0 + \cdots + b_{k-1}x^{k-1}$, with the parameters $(k, \ell)$.
Output: $C = AB$.
1: compute linear combinations $L_0, \dots, L_{\ell-1}$ of $a_0, \dots, a_{k-1}$
2: compute linear combinations $M_0, \dots, M_{\ell-1}$ of $b_0, \dots, b_{k-1}$
3: compute $N_0 = L_0M_0, \cdots, N_{\ell-1} = L_{\ell-1}M_{\ell-1}$
4: recover the coefficients of $C$ as linear combinations of $N_0, \dots, N_{\ell-1}$
```

To specify such an algorithm, one needs to give explicitly all required linear combinations. For instance, Karatsuba's algorithm has parameters $(2, 3)$, with

- $L_0 = a_0,\ L_1 = a_0 + a_1,\ L_2 = a_1$
- $M_0 = b_0,\ M_1 = b_0 + b_1,\ M_2 = b_1$
- $c_0 = N_0,\ c_1 = N_1 - N_0 - N_2,\ c_2 = N_2$.

For the recursive calls, given A and B with degree less than n, we let

$$h = \lfloor \frac{n+k-1}{k} \rfloor, \quad h' = n - (k-1)h, \tag{3.4}$$

so that h is approximately n/k. Precisely, if n has the form $n = qk$, we have $h = h' = q$; if n has the form $qk + r$, with $1 \le r < k$, we have $h = q + 1$ and $h' = q + r + 1 - k$. We want $h' > 0$; this will be the case as soon as $q > k - 2$, so that $n > (k-1)^2$ is sufficient. Then, we always have $0 < h' \le h$.

Now, we can write

$$A = A_0 + A_1x^h + \cdots + A_{k-1}x^{(k-1)h}, \quad B = B_0 + B_1x^h + \cdots + B_{k-1}x^{(k-1)h}$$

and

$$C = C_0 + C_1x^h + \cdots + C_{2k-2}x^{(2k-2)h}.$$

Thus, $A_0, \dots, A_{k-2}$ and $B_0, \dots, B_{k-2}$ have degrees less than h and A_{k-1} and B_{k-1} have degree less than h'. The polynomials $C_0, \dots, C_{2k-4}$ have degrees less than $2h - 1$, C_{2k-3} has degree less than $h + h' - 1$ and C_{2k-2} has degree less than $2h' - 1$.

In Algorithm 4 below, if $A = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ and p, q are integers, we use the notation slice(A, p, q) to denote the "slice" of A of length q starting at index

p, that is, the polynomial

$$a_p + a_{p+1}x + \cdots + a_{p+q-1}x^{q-1}.$$

This extra notation may seem slightly unnecessary here, but it will turn out useful for the transposed multiplication algorithm of the next section.

Next, we compute the linear combinations L_i of $A_0, \dots, A_{k-1}$ and M_i of $B_0, \dots, B_{k-1}$. To handle the recursive calls, we need bounds e_i and f_i such that

$$\deg(L_i) < e_i \quad \text{and} \quad \deg(M_i) < f_i$$

holds: we simply take $e_i = h$ if $L_i \neq B_{k-1}$, and $e_i = h'$ if $L_i = B_{k-1}$; the same construction holds for f_i (recall that for Karatsuba, this was mentioned in the remarks following Algorithm 2). For simplicity, we will assume here that $e_i = f_i$ holds for all i, though this is not necessarily the case.

Algorithm 4 Mul(A, B, n) Divide-and-conquer multiplication algorithm

```
Input: $A, B, n$, with $\deg(A) < n$ and $\deg(B) < n$, where $h$ and $h'$ satisfy Eq.(3.4),
    with the parameters $(k, \ell)$
Output: $C = AB$
1: if $n \le (k-1)^2$ then
2:     return $AB$                                naive multiplication
3: end if
4: for $i = 0$ to $k-2$ do
5:     $A_i = \text{slice}(A, ih, h)$
6: end for
7: $A_{k-1} = \text{slice}(A, (k-1)h, h')$
8: for $i = 0$ to $k-2$ do
9:     $B_i = \text{slice}(B, ih, h)$
10: end for
11: $B_{k-1} = \text{slice}(B, (k-1)h, h')$
12: compute the linear combinations $L_0, \dots, L_{\ell-1}$ of $A_0, \dots, A_{k-1}$
13: compute the linear combinations $M_0, \dots, M_{\ell-1}$ of $B_0, \dots, B_{k-1}$
14: for $i$=0 to $\ell-1$ do
15:     $N_i = \text{Mul}(L_i, M_i, e_i)$
16: end for
17: recover $C_0, \dots, C_{2k-2}$ as linear combinations of $N_0, \dots, N_{\ell-1}$
18: return $C = C_0 + C_1x^h + \cdots + C_{2k-2}x^{(2k-2)h}$.
```

Let $T(n)$ be the number of operations performed in size n. For $n \leq (k-1)^2$, we have $T(n) = O(n^2)$; for $n > (k-1)^2$, we have the recurrence

$$T(n) = (\ell - 1)T(h) + T(h') + \lambda n + \mu, \tag{3.5}$$

where λ and μ are constants that depend on the number of operations we do at steps steps 12, 13 and 18. Using the Master theorem, one deduces that a divide-and-conquer algorithm of parameters (k, ℓ) has complexity $O(n^{\log_k(\ell)})$. It is crucial to observe that the constant hidden in the $O()$ is proportional to the number of operations used to compute all linear combinations taking place at steps 12, 13 and 18: the easier these combinations are to compute, the faster the algorithm.

Example. We can recast the naive algorithm as a divide-and-conquer algorithm, in many possible ways. For instance, with parameters $(2, 4)$ we can compute the product of polynomials $A = a_0 + a_1 x$ and $B = b_0 + b_1 x$ as

- $L_0 = a_0,\ L_1 = a_0,\ L_2 = a_1,\ L_3 = a_1$
- $M_0 = b_0,\ M_1 = b_1,\ M_2 = b_0,\ L_3 = b_1$
- $c_0 = N_0,\ c_1 = N_1 + N_2,\ c_2 = N_3.$

Of course, we recover the cost $O(n^2)$, so this remark is not very interesting.

Some more useful divide-and-conquer algorithms are obtained as generalizations of Karatsuba's algorithm. To devise them, we start by observing that the linear combinations computed within Karatsuba's algorithm are

$$L_0 = a_0 = A(0), \quad L_1 = a_0 + a1 = A(1), \quad L_2 = a_1 = A(\infty)$$

and

$$M_0 = b_0 = B(0), \quad M_1 = b_0 + b_1 = B(1), \quad M_2 = b_1 = B(\infty),$$

where $A(\infty)$ and $B(\infty)$ are short-hand notations to denote the coefficients of A and B of maximal degree. Hence, Karatsuba's algorithm amounts to:

- evaluate A and B at $0, 1, \infty$,
- multiply the values pairwise,
- recover $C = AB$ from its values at $0, 1, \infty$.

Making this remark enables us to present Toom's family of divide-and-conquer algorithms, of parameters $(k, 2k-1)$ for any $k \geq 2$. The algorithm in size k chooses $2k-1$ pairwise distinct evaluation points $x_1, \ldots, x_{2k-1}$. Then, to multiply polynomials A and B with degree less than k, the algorithm proceeds as follows:

- evaluate A and B at $x_1, \ldots, x_{2k-1}$;
- multiply the values pairwise,
- recover $C = AB$ from its values at $x_1, \ldots, x_{2k-1}$ by interpolation.

Since C has degree at most $2k-2$, the last step is well-defined. By what was said above, this algorithm gives a complexity $\mathsf{M}(n) = O(n^{\log_k(2k-1)})$; this gets better as k gets larger.

However, one rarely sees implementations going beyond the simplest case $k = 2$. Indeed, the constant factors hidden in the $O()$ are a severe limiting factor. Concretely, as for Karatsuba's algorithm, one should first choose $x_1, \ldots, x_{2k-1}$ such that evaluation and interpolation are easy. For instance, with $k = 3$, a usual choice is $0, 1, -1, 2, \infty$. For this latter case, we can give the explicit values of all linear combinations that are performed. Since $k = 3$, we write

$$A = a_0 + a_1 x + a_2 x^2, \quad B = b_0 + b_1 x + b_2 x^2$$

and, since $2k - 1 = 5$, we have

$$C = AB = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4.$$

Then, the operations used to compute C are the following:

- $L_0 = a_0,\ L_1 = a_0 + a_1 + a_2,\ L_2 = a_0 - a_1 + a_2,\ L_3 = a_0 + 2a_1 + 4a_2$ and $L_4 = a_2$
- $M_0 = b_0,\ M_1 = b_0 + b_1 + b_2,\ M_2 = b_0 - b_1 + b_2,\ M_3 = b_0 + 2b_1 + 4b_2$ and $M_4 = b_2$
- $c_0 = N_0,\ c_1 = -\frac{1}{2}(N_0) + N_1 - \frac{1}{3}N_2 - N_3 + 2N_4,\ c_2 = \frac{1}{2}(N_2 - N_1) - N_4,\ c_3 = \frac{1}{2}(N_0 - N_1) - \frac{1}{6}(N_2 - N_3) - 2N_4,\ c_4 = N_4.$

Once $x_1, \ldots, x_{2k-1}$ are fixed, it remains to find a way to compute all needed linear combinations efficiently, by detecting common sub-expressions, etc. This problem was already mentioned in Section 2.1, with more details in [3]; it is not discussed in this thesis.

Graphical representation. In the implementation described in Chapter 4, we will use graphical representations: since a multiplication algorithm involves 3 steps of linear combinations, the graphs are naturally split into 3 parts. In Figure 3.1, we give the full graph of Karatsuba's multiplication; the dash-line represents multiplying L_i by M_i to get N_i. In Figure 3.2, we give Toom's algorithm with $k = 3$; we do not display the combinations applied to B, since they are the same as those applied to A.

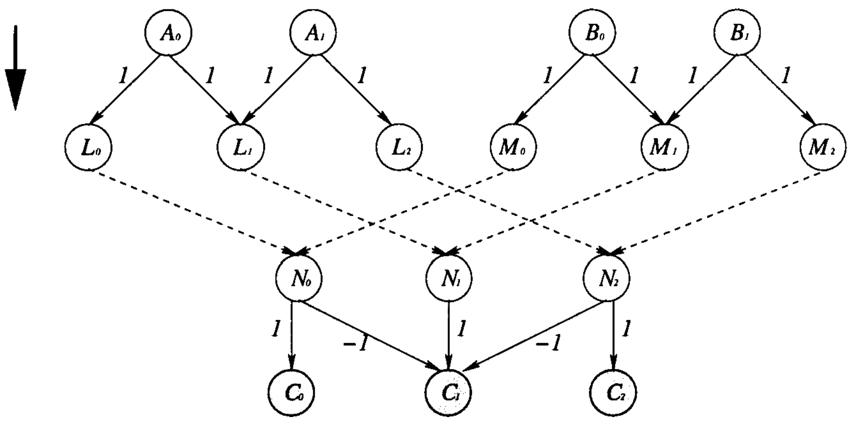

Figure 3.1: Karatsuba Multiplication

FFT multiplication. We conclude this subsection by a remark regarding multiplication using the Fast Fourier Transform [43] (FFT). This family of algorithms is known to yield algorithms with a better complexity of either $O(n\log(n))$ or $O(n\log(n)\log\log(n))$, depending on the properties of the coefficient ring. On the other hand, for moderate degrees (a few hundreds), it is expected that the divide-and-conquer methods can be competitive, due to the simpler form of the algorithms.

FFT algorithms do not belong to our family of divide-and-conquer algorithms. In degree n, polynomial multiplication by FFT involves computing with roots of unity of order 2^k, with $2^k \simeq n$, such as $\exp(2i\pi/2^k)$ if we work with complex coefficients. Such algorithms, which handle more and more complex linear combinations as the degree grows, will not enter our study.

Previous work. The content of this subsection is standard. Karatsuba's and Toom's algorithms appeared in [6] by Bostan *et.al.* and [17] by Hanrot *et.al.* . Our formalism of divide-and-conquer algorithms goes back to the 1970's as well; an extensive reference is from Winograd [45].

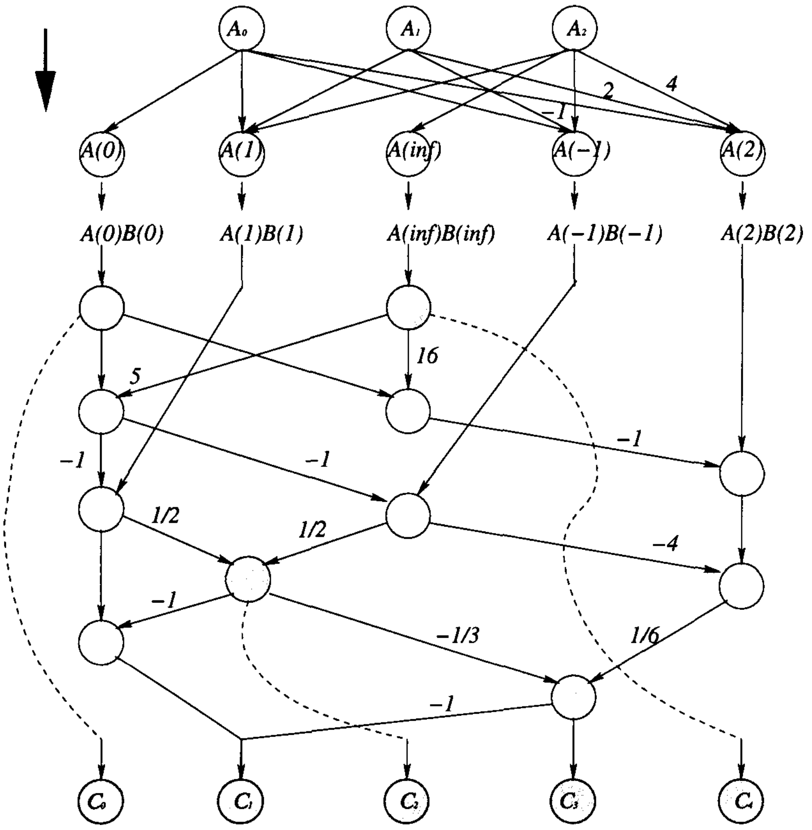

Figure 3.2: Toom Multiplication

3.2 Transposed multiplication

If we keep the polynomial A fixed, the multiplication operation

$$B \mapsto AB$$

is linear, and it makes sense to speak of its transposed map. We illustrate it by a simple example first.

Transposed product in degree less than 2. Given two polynomials $A = a_0 + a_1 x$ and $B = b_0 + b_1 x$, the result of the multiplication of A by B is

$$C = AB = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 a_1 x^2.$$

As we said before, we keep A fixed, and we assume that B varies. One sees that the coefficients of $C = AB$ are given by the matrix-vector product Mv:

$$M = \begin{bmatrix} a_0 & 0 \\ a_1 & a_0 \\ 0 & a_1 \end{bmatrix}, v = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \implies Mv = \begin{bmatrix} a_0 b_0 \\ a_1 b_0 + a_0 b_1 \\ a_1 b_1 \end{bmatrix}$$

Let now M^t be the transposed matrix of M; then, for a vector w, $M^t w$ is obtained as follows:

$$M^t = \begin{bmatrix} a_0 & a_1 & 0 \\ 0 & a_0 & a_1 \end{bmatrix}, w = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \implies M^t w = \begin{bmatrix} a_0 c_0 + a_1 c_1 \\ a_0 c_1 + a_1 c_2 \end{bmatrix}$$

To understand the interest of this computation, we extend M^t to a larger matrix N, defined by adjoining a top and a bottom row:

$$N = \begin{bmatrix} a_1 & 0 & 0 \\ a_0 & a_1 & 0 \\ 0 & a_0 & a_1 \\ 0 & 0 & a_0 \end{bmatrix}.$$

Multiplying N by the vector w now gives

$$Nw = \begin{bmatrix} a_1 c_0 \\ a_0 c_0 + a_1 c_1 \\ a_0 c_1 + a_1 c_2 \\ a_0 c_2 \end{bmatrix}.$$

The extended product Nw is the vector of coefficients of $C\widetilde{A}$, with

$$C = c_0 + c_1 x + c_2 x^2 \quad \text{and} \quad \widetilde{A} = a_1 + a_0 x.$$

The transposed product $M^t w$ computes only the middle of Nw, which represents the *middle* coefficients of $C\widetilde{A}$. We will show applications of this remark in Chapter 6.

Transposed product in general. Let us still assume that the polynomial A is fixed, now with $\deg(A) < n$. We consider the operation $B \mapsto C = AB$, with $\deg(B) < n$ as well; then, $\deg(C) < 2n - 1$.

The transposed multiplication takes a polynomial C of degree less than $2n-1$ and returns a polynomial B of degree less than n. Proceeding as in the previous example, by writing down the matrix of the operation, we see that the output B is obtained as follows: let

$$D = d_0 + d_1x + \cdots + d_{3n-2}x^{3n-2}$$

be the product $C\widetilde{A}$, where $\widetilde{A}$ is the reversed polynomial

$$\widetilde{A} = a_{n-1} + \cdots + a_0x^{n-1}.$$

Then B is the *middle part* of D, consisting of coefficients of degree $n-1$ to $2n-2$:

$$B = d_{n-1} + \cdots + d_{2n-2}x^{n-1}.$$

Applications of this remark will be made in Chapter 6. As announced in the introduction, we will use the short-hand notation $B = CA^t$.

Since A is fixed, the operation $B \mapsto AB$ is linear, so we can use the transposition principle to estimate the complexity of its transpose: the results of Section 2.1 show that one can compute the transposed product CA^t using $\mathsf{M}(n) + O(n)$ operations. If we use the naive algorithm for the direct product, the naive *transposed* algorithm amounts to compute each coefficient b_i of B as

$$b_i = \sum_{j=0}^{n-1} a_j c_{i+j}, \tag{3.6}$$

for $i = 0, \ldots, n-1$. It remains to give explicit algorithms for the divide-and-conquer approach.

Example: Transposed Karatsuba multiplication. We show here how to concretely devise a transposed multiplication algorithm, first in the case of Karatsuba's multiplication.

We start with polynomials of degree less than 2. As before, let $A = a_0 + a_1x$. The graphical representation of Karatsuba's multiplication was given in Figure 3.1. Following the rules given in Section 2.1, we deduce that we need to reverse all edges concerning B, but that all the edges concerning A remain the same; the output C becomes an input. The representation of the transposed graph is in Figure 3.3, and gives Algorithm 5,

In higher degree, we apply this approach recursively. Let h and h' be as in Equa-

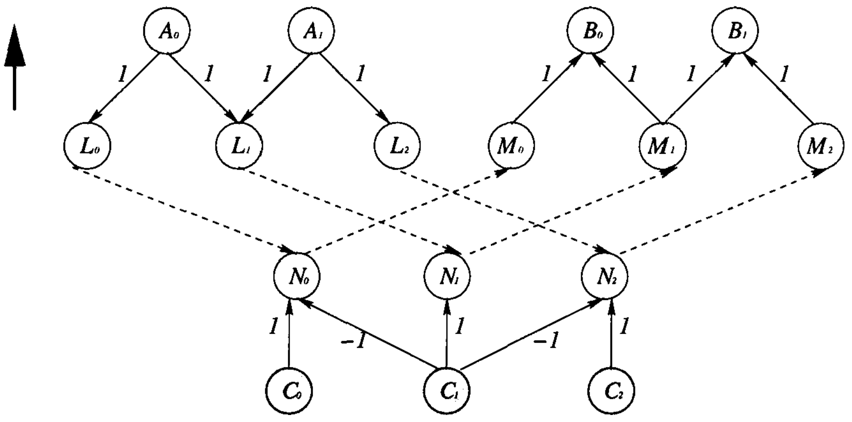

Figure 3.3: Transposed Karatsuba Multiplication

Algorithm 5 Transposed Karatsuba in degree less than 2

```
Input: $A = a_0 + a_1 x, C = c_0 + c_1 x + c_2 x^2$
Output: $B = CA^t$
1: $L_0 = a_0,\ L_1 = a_0 + a_1,\ L_2 = a_1$
2: $N_0 = c_0 - c_1,\ N_1 = c_1,\ N_2 = c_2 - c_1$
3: $M_0 = L_0 N_0,\ M_1 = L_1 N_1,\ M_2 = L_2 N_2$
4: return $(M_0 + M_1) + (M_2 + M_1)x$
```

tion (3.3). Writing the algorithm is straightforward, after we remember the following points.

- The degrees of C_0, C_1 and C_2 are bounded by $2h-1$, $h+h'-1$, $2h'-1$. We can thus define C_0, C_1 and C_2 formally using the "slice" notation introduced before: $C_0 = \text{slice}(C, 0, 2h-1)$, which means we split C and copy the terms from degree 0 to $2h-2$ to obtain C_0. Similarly, $C_1 = \text{slice}(C, h, h+h'-1)$ and $C_2 = \text{slice}(C, 2h, 2h'-1)$.

- In the direct algorithm N_0 and N_1 have degrees less than $2h-1$ and N_2 has degree less than $2h'-1$. We enforce the same degree constraints in the transposed algorithm, by applying truncation.

Transposed version of divide-and-conquer algorithms. Following the example of Karatsuba's algorithm, it is possible to give a general form for transposed divide-and-conquer algorithms: the operations for A remain the same, whereas those regarding B and C are reversed. We directly give the recursive version in Algorithm 7.

Previous work. Most of the content of this subsection is already known, though not widely so. The example of the transposed product is discussed in detail in [45];

Algorithm 6 TranKaraMul(A, C, n) Transposed Karatsuba multiplication

```
Input: $A, C, n$, with $\deg(A) < n$ and $\deg(C) < 2n-1$
Output: $B = CA^t$
1: if $n = 1$ then
2:     return $AC$
3: end if
4: $h = \lfloor (n+1)/2 \rfloor$, $h' = n - h$
5: $A_0 = \text{slice}(A, 0, h)$, $A_1 = \text{slice}(A, h, h')$
6: $C_0 = \text{slice}(C, 0, 2h-1)$, $C_1 = \text{slice}(C, h, h+h'-1)$, $C_2 = \text{slice}(C, 2h, 2h'-1)$
7: $L_0 = A_0$, $L_1 = A_0 + A_1$, $L_2 = A_1$
8: $N_0 = C_0 - C_1 \bmod x^{2h-1}$, $N_1 = C_1 \bmod x^{2h-1}$, $N_2 = C_2 - C_1 \bmod x^{2h'-1}$
9: $M_0 = \text{TranKaraMul}(L_0, N_0, h)$
10: $M_1 = \text{TranKaraMul}(L_1, N_1, h)$
11: $M_2 = \text{TranKaraMul}(L_2, N_2, h')$
12: return $(M_0 + M_1) + (M_2 + M_1)x^h$
```

it was rediscovered independently in [17], which also studied in detail its applications to Newton iteration. Transposed multiplication was also studied in [6]. As far as divide-and-conquer algorithms are concerned, the above previous works contained descriptions of a recursive transposed Karatsuba algorithm; a general algorithm such as Algorithm 7 has not appeared before.

3.3 Short multiplication

The last variant of polynomial multiplication we consider here is the *short multiplication* or *short product*. As before, we consider as input polynomials

$$A = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}, \quad B = b_0 + b_1x + \cdots + b_{n-1}x^{n-1};$$

we are interested here in computing the truncated, or short, product

$$C = AB \bmod x^n = c_0 + \cdots + c_{n-1}x^{n-1}.$$

Of course, it would be sufficient to compute the full product AB and discard all terms of degree at least n. The question we consider here is thus how to compute C directly without computing any useless terms.

If we use the *naive* product, it is clear that such savings are possible: one simply does not compute the unneeded coefficients, which saves a factor of 2. However, for divide-and-conquer algorithms, it is less obvious to see what savings are possible,

Algorithm 7 TranMul(A, C, n)Transposed multiplication

```
Input: $A, C, n$, with $\deg(A) < n$ and $\deg(C) < 2n-1$, where the parameters $(k, \ell)$.
Output: $B = CA^t$
 1: if $n \leq (k-1)$ then
 2:     return $CA^t$                        naive transposed multiplication using (3.6)
 3: end if
 4: $h = \lfloor (n+k-1)/k \rfloor$, $h' = n - (k-1)h$
 5: for $i$=0 to $k-2$ do
 6:     $A_i = \text{slice}(A, ih, h)$
 7: end for
 8: $A_{k-1} = \text{slice}(A, (k-1)h, h')$
 9: for $i$=0 to $2k-4$ do
10:     $C_i = \text{slice}(C, ih, 2h-1)$
11: end for
12: $C_{2k-3} = \text{slice}(C, (2k-3)h, h+h'-1)$
13: $C_{2k-2} = \text{slice}(C, (2k-2)h, 2h'-1)$
14: compute the linear combinations $L_0, \ldots, L_{\ell-1}$ of $A_0, \ldots, A_{k-1}$
15: compute the transposed linear combinations $N_0, \ldots, N_{\ell-1}$ of $C_0, \ldots, C_{2k-2}$, with
    $N_i$ truncated modulo $x^{2e_i-1}$
16: for $i$=0 to $\ell-1$ do
17:     $M_i = \text{TranMul}(L_i, N_i, e_i)$
18: end for
19: compute the transposed linear combinations $B_0, \ldots, B_{k-1}$ of $M_0, \ldots, M_{\ell-1}$
20: return $B_0 + \cdots + B_{k-1}x^{(k-1)h}$
```

since the coefficients in the result are obtained as linear combinations of a whole set of intermediate products. It turns out that a divide-and-conquer approach is possible here too, but using a different division pattern, using the idea of *decimation* [42], introduced now.

Given $k \geq 2$, $n > 0$ and i in $\{0, 1, \cdots, n+k-1\}$, we define

$$h_i = \lfloor (n+k-1-i)/k \rfloor. \qquad (3.7)$$

Then, if we consider a polynomial

$$A = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

as above, we will denote by $A_i = \text{decimation}(A, i, h_i)$ the polynomial

$$A_i = a_i + a_{i+k} x + \cdots + a_{i+(h_i-1)k} x^{h_i - 1},$$

which keeps one of every k coefficients of A, starting from a_i. The definition of h_i is such that $i+(h_i-1)k \leq n-1$, whereas the next index $i+h_ik$ is $\geq n$, so it is useless in A_i. Thus, we have

$$A = A_0(x^k) + \cdots + x^{k-1}A_{k-1}(x^k).$$

Example: Karatsuba multiplication. Let us show how to use this idea on the easiest meaningful example, Karatsuba multiplication for polynomials of degree less than 4. Consider

$$A = a_0 + a_1x + a_2x^2 + a_3x^3 \quad \text{and} \quad B = b_0 + b_1x + b_2x^2 + b_3x^3.$$

The short product of A and B is

$$C = AB \bmod x^4 = c_0 + c_1x + c_2x^2 + c_3x^3.$$

Define now

$$A_0 = a_0 + a_2x, \quad A_1 = a_1 + a_3x, \quad B_0 = b_0 + b_2x, \quad B_1 = b_1 + b_3x,$$

so that

$$A = A_0(x^2) + xA_1(x^2), \quad B = B_0(x^2) + xB_1(x^2);$$

A_0, A_1, B_0, B_1 are the decimations introduced above. Let also

$$C_0 = A_0B_0, \quad C_1 = A_0B_1 + A_1B_0, \quad C_2 = A_1B_1.$$

This implies

$$C = AB \bmod x^4 = C_0(x^2) + xC_1(x^2) + x^2C_2(x^2) \bmod x^4.$$

To compute a product of the form $x^iC_i(x^k) \bmod x^n$, it is enough to compute $C_i \bmod x^{h_i}$, with $h_i = \lfloor (n+k-1-i)/k \rfloor$ as in Equation (3.7), since this gives us all the coefficients that we need. Here with $n=4$ and $k=2$, this means that we only need to compute recursively

$$C_0 \bmod x^2, \quad C_1 \bmod x^2, \quad C_2 \bmod x.$$

To compute them, we use Karatsuba's formula. If we did not compute modulo powers of x, we would compute

$$N_0 = A_0 B_0, \quad N_1 = (A_0 + A_1)(B_0 + B_1), \quad N_2 = A_1 B_1$$

and

$$C_0 = N_0, \quad C_1 = N_1 - N_0 - N_2, \quad C_2 = N_2.$$

Because we do compute modulo these powers of x, we see that we need $N_0 \bmod x^2$, $N_1 \bmod x^2$, but also $N_2 \bmod x^2$, since it is used in C_1. This gives us the truncation degrees we need to use in the recursive call.

The general case. Following the previous idea, we can use any divide-and-conquer algorithm to perform short multiplications. Using an algorithm of parameters (k, ℓ), and given A and B with degree $< n$, we write

$$A = A_0(x^k) + \cdots + x^{k-1} A_{k-1}(x^k), \quad B = B_0(x^k) + \cdots + x^{k-1} B_{k-1}(x^k).$$

If we define $C_i = \sum_{r+s=i} A_r B_s$, we see that

$$C = AB \bmod x^n = C_0(x^k) + \cdots + x^{2k-2} C_{2k-2}(x^k) \bmod x^n.$$

As said before, it is enough to compute each $C_i \bmod x^{h_i}$, with $h_i = \lfloor (n + k - 1 - i)/k \rfloor$. To obtain them, recall that our divide-and-conquer algorithm computes some products $N_0, \ldots, N_{\ell-1}$. By inspecting the linear combinations we perform to obtain $C_0, \ldots, C_{2k-2}$ from $N_0, \ldots, N_{\ell-1}$, we can deduce integers g_i such that N_i should be computed modulo x^{g_i}: $g_i = h_{i_0}$, where i_0 is the largest index such that N_i contributes to C_{i_0}. Using this remark, we obtain Algorithm 8.

Algorithm 8 ShortMul(A, B, n) Short multiplication algorithm

```
Input: $A, B, n$, with $\deg(A) < n$, $\deg(B) < n$, with $h$ and $h'$ satisfy Eq.(3.7), with
    the parameters $(k, \ell)$.
Output: $C = AB \bmod x^n$
 1: if $n = 1$ then
 2:     return $AB$
 3: end if
 4: for $i$=0 to $k-2$ do
 5:     $A_i = \text{decimation}(A, i, h_i)$
 6: end for
 7: for $i$=0 to $k-2$ do
 8:     $B_i = \text{decimation}(B, i, h_i)$
 9: end for
10: compute the linear combinations $L_0, \dots, L_{\ell-1}$ of $A_0, \dots, A_{k-1}$
11: compute the linear combinations $M_0, \dots, M_{\ell-1}$ of $B_0, \dots, B_{k-1}$
12: for $i$=0 to $\ell-1$ do
13:     $N_i = \mathsf{ShortMul}(L_i \bmod x^{g_i}, M_i \bmod x^{g_i}, g_i)$
14: end for
15: compute the linear combinations $C_0, \dots, C_{2k-2}$ of $N_0, \dots, N_{\ell-1}$, truncating $C_i$
    modulo $x^{h_i}$
16: return $C = C_0(x^k) + xC_1(x^k) + \cdots + x^{2k-2}C_{2k-2}(x^k)$
```

Analyzing the complexity of short product algorithms is difficult. The Master theorem shows that starting from a divide-and-conquer algorithm of parameters (k, ℓ), the complexity is $O(n^{\log_k(\ell)})$. This cost is thus the same as that of a plain product, up to the constant factor in the $O()$. Besides, the $O()$ notation only gives an upper bound, whereas we would need sharper inequalities to compare short and plain products.

For Karatsuba's multiplication, Hanrot and Zimmermann [19] proved that the ratio between the number of multiplications in a short product and in a plain product varies between 3/5 and 1; however, this does not count additions. Moreover, if we introduce a threshold such as 16 under which we use the naive short product, this conclusion does not hold anymore: the number of multiplications is reduced by 10% to 40%.

To summarize, we will not quantify the gain brought by using short products, since this kind of analysis is very complex. The cost of short product in degree less than n will be written $\mathsf{m}(n)$, and we will remember that $\mathsf{m}(n) \leq \mathsf{M}(n)$.

Previous work. The idea of short product goes back to Mulders [28], and was studied in detail by Hanrot and Zimmermann in [19]. Mulders' algorithm proceeds differently, by cutting the polynomial into non-equal slices; the length of the slices determines the complexity. That algorithm was limited to Karatsuba multiplication.

Hanrot and Zimmermann introduced a variant using the idea of decimation, and gave details for Karatsuba multiplication. They mention briefly an extension to Toom multiplication, but do not indicate that this approach can be applied in all generality, and how to do it, as we do.

Chapter 4

Code generation

In this chapter, we describe our work on code generation for polynomial multiplication. We only consider divide-and-conquer algorithms, with the convention of Section 3.1; recall that this includes algorithms such as Karatsuba's or Toom's, but not Fast Fourier Transform algorithms (that case is developed by Li *et.al.* and Filatei *et.al.* in detail in [24, 14]). Some effort is put on producing optimized code, but no attention was paid to write cache-friendly code or exploit parallelism.

4.1 Coefficient arithmetic

Our focus is on coefficient types that can be represented using machine data types; namely, we consider the following:

- polynomials with `double` coefficients;
- polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where p is an integer (typically a prime) that fits in a machine word; these will be called *modular* coefficients.

In the latter case, since our implementations are all done on 64 bit platforms (Intel Core2 or AMD 64), long machine words can hold up to 64 bits. We will actually slightly reduce this bound, for reasons explained in the latter sections.

While the `double` case is widely used in numerical analysis, the modular case is a basic ingredient for symbolic computation, and we showed in the introduction an application that required it. The 64 bit limit is not a very strong restriction: multiple precision integers arithmetic can be implemented on top of 64 bit arithmetic, using the Chinese remainder theorem. This aspect is developed in [24]; we will not develop it here, and limit ourselves to the 64 bit case.

4.1.1 Double coefficients

To support this coefficient type, very little implementation effort is needed: we plainly use the `double` type in our code, with operations $+$, $-$, $*$, $/$.

Due to cancellations, operations on `double`'s do not satisfy the ring axioms of associativity or distributivity. Hence, formally, `double` coefficients do not enter in our framework. Nevertheless, we decided to support this type for our multiplication implementations, for the following reasons:

- We want to compare the running times between `double` and modular coefficients. Such comparisons have already been performed for linear algebra algorithms [13]; while the underlying techniques were different, they showed that modular operations can be almost as efficient as the operations with `double`. We want to see if this conclusion carries over to polynomial operations.
- We want to measure experimentally to what extent divide-and-conquer algorithms suffer from precision loss. Theoretical analyses are pessimistic, and we wish to test whether this is indeed the case in practice.

Contrary to the examples given in [24, 14], we did not dedicate efforts to write SSE2 code by ourselves. We left this task to the compiler.

4.1.2 Modular arithmetic

Given a positive integer p, *modular arithmetic* modulo p means that we maintain all results reduced modulo p:

- the set of values we handle is $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\}$;
- addition, subtraction and multiplication are done as integer operations, but the result is reduced to the range $\{0, \ldots, p-1\}$ by means of Euclidean division;
- inversion of m modulo p is possible only if m and p have no common factor; if so, the inverse of m modulo p is found by means of the extended Euclidean algorithm.

The remainder of an integer a through Euclidean division by the integer p will be written $a \bmod p$.

To represent the elements $\{0, \ldots, p-1\}$, we use the `unsigned long` integer data type. Then, our main focus is on the operations $+$, $-$ and $*$ (inversions and divisions play a less important role in our code). The straightforward approach to implementing these operations modulo p consists in applying first the corresponding operation as integers, then performing the reduction modulo p.

There is one drawback, however: in C, reduction of an `unsigned long` a modulo p can be implemented using the expression $a\%p$. As it turns out, this instruction is practically very slow (about one order of magnitude slower than multiplication), so alternative solutions are needed, since performance is an issue. We present the solutions we used for the various operations we need to perform.

Modular addition. For additions, simple tests would seem to be sufficient. Indeed, to compute the modular addition $a + b \bmod p$, the algorithm could be implemented as in Algorithm 9.

Algorithm 9 Modular addition

```
Input: $p$ and $a, b$ in $\{0, \ldots, p-1\}$
Output: $c = a + b \bmod p$
1: $c = a + b$
2: if $c > p$ then
3:     $c = c - p$
4: end if
5: return $c$
```

This avoids the call to the reduction operator, but introduces a conditional statement. Again, since performance is an issue, we wish to avoid branchings, since they can affect the CPU instruction prediction. On our platforms, the construction of Algorithm 10 below is a more efficient workaround; it is taken from the library [35], and adapted to 64 bit integer representation. Here, we need to switch from pseudo-code to actual C language.

Algorithm 10 Efficient modular addition

```
Input: unsigned long p, unsigned long a, unsigned long b,
       with 0 ≤ a, b < p
Output: unsigned long c, with c = a + b mod p
1: unsigned long c=a+b
2: long d=c-p
3: long e=d+(d>>63)&p
4: return (unsigned long) e
```

Let us justify this procedure. We know that $c = a + b$ is always in $\{0, \ldots, 2p-2\}$, so that $d = c - p$ is in $\{-p, \ldots, p-2\}$. Using the sign bit of d as a mask, we see that the signed integer $d \gg 63$ is either -1 if $d < 0$, or 0 if $d \geq 0$. Thus, the logical and $(d \gg 63)\&n$ is either p if $d < 0$, or 0 if $d \geq 0$. As a consequence, the output is always in $\{0, \ldots, p-1\}$, as needed.

Modular subtraction. Subtractions follow the same pattern as additions. To compute $a - b \bmod n$, the algorithm could be implemented as follows:

Algorithm 11 Modular subtraction

```
Input: $p$ and $a, b$ in $\{0, \ldots, p-1\}$
Output: $a - b \bmod p$
1: $c = a - b$
2: if $c < 0$ then
3:     $c = c + p$
4: end if
5: return $c$
```

Here, $c = a - b$ is always in $\{-(p-1), \ldots, p-1\}$, so that at most one addition of p is needed; this proves correctness. As before, however, we want to avoid the test; we use a method similar to the one for addition, given in Algorithm 12. Again, we use C syntax.

Algorithm 12 Efficient modular subtraction

```
Input: unsigned long p, unsigned long a, unsigned long b,
       with 0 ≤ a,b < p
Output: unsigned long c, with c = a − b mod p
1: unsigned long c=a-b
2: long d=c+(c>>63)&p
3: return (unsigned long) d
```

Here, $c = a - b$ is always in $\{-(p-1), \ldots, p-1\}$, and $c \gg 63 = -1$ if $c < 0$ and 0 otherwise. Thus, as before, $(c \gg 63)\&p$ is either p if $c < 0$, or 0 if $c \geq 0$. As a consequence, the output is always in $\{0, \ldots, p-1\}$, as needed.

Modular multiplication by small constants. We will see in the next paragraphs that modular multiplication is a complex operation. Hence, when multiplications by small constants are needed, some adapted code can be used with profit. For instance, the computation of $2a \bmod p$ is just the addition of a with itself, so the addition code can be used; besides, in this case, in Algorithm 10, the addition $c = a + a$ can be replaced by $c = a \ll 1$.

As another example, the computation of $4a \bmod p$ can be done as follows: first, a is multiplied by 4 as an integer through the instruction $c = a \ll 2$. Now, c is in $\{0, \dots, 4p - 4\}$; the reduction of c modulo p can be done using a generalization of the previous examples. We write the code using tests in Algorithm 13; as we did previously, one can deduce test-free code using bit signs as masks.

Algorithm 13 Modular multiplication by 4

```
Input: $p$ and $a$ in $\{0, \dots, p-1\}$
Output: $4a \bmod p$
 1: $c = 4a$
 2: $c' = c - 2p$                            $c'$ is in $\{-2p, \dots, 2p-4\}$
 3: if $c' < 0$ then
 4:     $c' = c' + 2p$                       $c'$ is in $\{0, \dots, 2p-1\}$
 5: end if
 6: $c'' = c' - p$                           $c''$ is in $\{-p, \dots, p-1\}$
 7: if $c'' < 0$ then
 8:     $c'' = c'' + p$                      $c''$ is in $\{0, \dots, p-1\}$
 9: end if
10: return $c''$.
```

In our multiplication code, we use such adapted multiplication operations when we perform the 3 linear combinations, for multiplication by powers of 2 and by 3.

Modular multiplication in general. The general case of modular multiplication $a, b \mapsto ab \bmod p$ is a delicate operation. First, a and b are multiplied as long integers; then, the result $c = ab$ must be reduced modulo p. Remark that if p is 64 bits long, c can be 128 bit long: we will discuss how to handle integers of such size. Then, as before, we wish to avoid the modulo operator %: following [26], we will use *Montgomery multiplication* to solve this issue.

Previous implementations [35, 24, 14, 13] rely on computations using `double` coefficients as intermediate results. Indeed, the Euclidean division of c by n can be written as

$$c = qp + r,$$

where $r \in \{0, \dots, p-1\}$ is the remainder we want. Using `double` coefficients, one can compute c/p in floating point representation. Truncating to the nearest integer gives an integer q' close to q; knowing it, one can recover r, up to $\pm p$. However, on a 64 bit machine, as explained by Giorgi [15], this approach restricts p to be at most 2^{52}.

As in [26], our implementation uses integer operations. However, our modulus p is now allowed to be 64 bit long. To handle 128-bit long products, we use a data type specific to the `gcc` compiler, `__uint128_t`, which encodes 128 bit unsigned integers. As the machine instruction level, the multiplication of two `unsigned long` arguments into a `__uint128_t` result is handled by means of a single multiplication; the result is stored in two 64 bit registers. Thus, there is no need for us to write assembly code; the assembly code obtained this way shows satisfactory performance.

To describe modular reduction, we will assume that p is an *even* number. Our solution uses *Montgomery multiplication* [27], which allows for operations using integers only. Our explanations below are dedicated to 64 bit integers, though the algorithm carries over to arbitrary length.

Montgomery's algorithm does not compute the remainder $c \bmod p$, but $c/2^{64} \bmod p$; we will explain in the next section our workaround. Montgomery's algorithm precomputes the number $p' = -1/p \bmod 2^{64}$.

Algorithm 14 Montgomery multiplication

```
Input: $p, p', a, b$ with $a, b$ in $\{0, \dots, p-1\}$ and $p' = -1/p \bmod 2^{64}$
Output: $ab/2^{64} \bmod p$
1: $c = ab$
2: $q = (c \bmod 2^{64})p' \bmod 2^{64}$
3: $t = (c + pq)/2^{64}$
4: if $t \geq p$ then
5:     $t = t - p$
6: end if
7: return $t$
```

We refer to the original article for the correctness proof. Here, we rather insist on the key advantages of this algorithm: it replaces computations modulo p, which are difficult, by computations modulo 2^{64}, which are easy to implement in C. The implementation in [26] uses special tricks for special forms of p. We do not use such tricks; we follow the above pseudo-code, so that each modular multiplication performs 3 long integers multiplications.

Division by constants. Some algorithms, such as Toom's, require multiplication by constants such as $1/2$ or $1/3$ that are not integers. When the data type is `double`, we simply use a `double` approximation. When the data type is `unsigned long`, a constant such as $1/3$ will be replaced by its image modulo p, which is obtained by applying the extended Euclidean algorithm with 3 and p.

As for multiplication, however, faster solutions are available for special constants, such as $1/2$. Suppose indeed that we want to compute $a/2 \bmod p$, where p is odd. Let us write $p = 2k + 1$; then, $k = -1/2$ modulo p, so that $-k = 1/2 \bmod p$. Let us then write $a_0 = a \bmod 2$ and $a_1 = a \operatorname{div} 2$, so that $a = 2a_1 + a_0$. This implies that $a/2 = a_1 + a_0/2$. Thus, if $a_0 = 0$, $a/2 = a_1 \bmod p$ and if $a_0 = 1$, $a/2 = a_1 - k \bmod p$. Thus, we have replaced the division by 2 modulo p by a shift and an addition.

4.2 Polynomial arithmetic

In this section, we describe our code generation techniques for polynomial arithmetic. We saw in Chapter 3 that a divide-and-conquer algorithm of parameters (k, ℓ) for polynomial multiplication is specified by the operations it performs for the case of polynomials with k terms. This itself can be described by three linear graphs, that describe the three series of linear combinations the algorithm performs.

We developed a Java program that generates C code for such algorithms. Adjacency matrices are stored in directories with names such as `kara`, `toom`, etc. On input such a triple of adjacency matrices, the code generator produces C code for the following operations:

- plain multiplication
- transposed multiplication
- short multiplication
- square (given A of degree $< n$, compute A^2)
- short square (given A of degree $< n$, compute $A^2 \bmod x^n$);

the two latter are specialized versions of previous ones, expected to be slightly faster (e.g., we avoid computing twice the same linear combinations, etc). The data type used for all these operations can be chosen to be either `double` or `unsigned long`.

Suppose for instance that we consider the Karatsuba algorithm; the code generator reads the corresponding three adjacency matrices in a directory called `kara`. The three input files matrices are

```
5            5            6
```

```
2           2           3
3           3           3
0 0 0 0 0   0 0 0 0 0   0  0  0 0 0 0 0
0 0 0 0 0   0 0 0 0 0   0  0  0 0 0 0 0
1 0 0 0 0   1 0 0 0 0   0  0  0 0 0 0 0
1 1 0 0 0   1 1 0 0 0   1  0  0 0 0 0 0
0 1 0 0 0   0 1 0 0 0   -1 1 -1 0 0 0 0
                        0  0  1 0 0 0 0
```

In each case, the first numbers are the number of vertices in the graph, the number of inputs and the number of outputs of the graph.

Given these matrices, for a chosen multiplication type, and for a chosen data type, our code generator creates several files. We give an overview of this process for the case of plain multiplication, with `unsigned long` data type (all other cases are processed similarly). Concretely, the following output will be produced:

- a file `kara_unsigned_long.c` containing the top-level function: this function allocates temporary memory to be used as a workspace in the recursive calls, precomputes some constants if needed (more explanations are given below), performs a scaling on one of the arguments (more explanations are given below), and calls the recursive function.
- a file `kara_unsigned_long_rec.c` containing the main recursive function; this function performs the first two linear combinations, the necessary recursive calls and the final linear combinations.
- files `kara_unsigned_long1.c`, `kara_unsigned_long2.c` and `kara_unsigned_long3.c`; each of them contains a function that performs the corresponding linear combination (respectively first, second and third one).

We continue with describing these files in more detail.

Top level function. The following shows the source code produced for the top-level function (the layout was rearranged to fit this page, and `#include`'s have been removed). For this function, little optimization effort is needed, since it is rather simple.

```
void kara_unsigned_long(unsigned long * __restrict__ C,
                        const unsigned long * __restrict__ A,
                    const unsigned long * __restrict__ B,
          const unsigned long ip, const unsigned long jp,
                            const unsigned long p, int N){
    int i;
    unsigned long *tmp_A=(unsigned long *) malloc(N*sizeof(unsigned long));
    for(i=0; i<N; i++)
      tmp_A[i] =  mul_montgomery(p, ip, A[i], jp);
    int l = (long) (4*N*2/((double)1));
    unsigned long *wk = (unsigned long *) malloc(l*sizeof(unsigned long));
    kara_unsigned_long_rec(C, tmp_A, B, wk, ip, p, N);
    free(wk);
    free(tmp_A);
}
```

The integer `N` is such that $\deg(A) < N$ and $\deg(B) < N$. The pointer `wk` points to a temporary workspace that will be used by the recursive function; the amount of space needed is determined by the code generator during the code generation.

The case of `unsigned long` coefficients requires extra operations compared to `double` coefficients:

- Remember that after all linear combinations are done, the entries of A should be multiplied by those of B. However, as said in Section 4.1.2, we use Montgomery's algorithm for the multiplication, so that the result we obtain will be $AB/2^{64} \bmod p$. To solve this, we create a temporary array *tmp_A* which stores a copy of A, where all elements are premultiplied by $2^{64} \bmod p$. There is one last difficult point: since multiplication by $2^{64} \bmod p$ will be done with Montgomery's algorithm, we actually precompute the value $j' = 2^{128} \bmod p$. Thus, the Montgomery product of j' by $A[i]$ computes

$$A[i]j'/2^{64} \bmod p = A[i]2^{128}/2^{64} \bmod p = A[i]2^{64} \bmod p.$$

- Another task, not needed in the case of Karatsuba multiplication, is the precomputation of constants. Suppose indeed that the linear combinations involve multiplication by numbers such as $1/3$. Since p is not known at compile-time, the top-level function will have to compute the inverse of $1/3$ modulo p; then, it will pass it as extra arguments to the recursive calls.

Recursive function. The following shows the source code produced for the recursive function. One sees three recursive calls, the three linear combinations, as well as the final addition mentioned in Section 3. When the length N is less than a certain threshold (17), we switch to the classical multiplication.

```
void kara_unsigned_long_rec(unsigned long* __restrict__ C,
                     const unsigned long * __restrict__ A,
                     const unsigned long * __restrict__ B,
                          unsigned long * __restrict__ wk,
     const unsigned long ip, const unsigned long p, int N){
  int i, j;
  if (N<17){
    mul_unsigned_long(C, A, B, ip, p, N);
    return;
  }
  const int h = (N+(2-1))/2;
  const int g = 2*h-1;
  const int r = N-h*2;
  const int lengthG0=h + r*0;
  ...
  const int lengthM2=g + r*2;
  kara_unsigned_long1_all(wk + h*0, A, A + h, h+r*1, ip, p, r);
  kara_unsigned_long2_all(wk + h*1, B, B + h, h+r*1, ip, p, r);
  unsigned long * new_wk = wk+h*4;
  kara_unsigned_long_rec(C, A, B, new_wk, ip, p, lengthG0);
  kara_unsigned_long_rec(wk + h*2, wk + h*0, wk + h*1, new_wk, ip, p,
                        lengthG1);
  kara_unsigned_long_rec(C + h*2, A + h, B + h, new_wk, ip, p,
                        lengthG2);
  kara_unsigned_long3_all(wk + h*0, C, wk + h*2, C + h*2, g+r*2, ip,
                         p, r);
  for (i=2*h*0+lengthM0; i<h*2 && i<2*N-1; i++)
    C[i]=0;
  for (i=2*h*1+lengthM2; i<h*4 && i<2*N-1; i++)
    C[i]=0;
  add_in_place_unsigned_long(C + h*1, wk + h*0, ip, p, lengthM1);
}
```

Some variables are defined (such as `lengthM2`, which stores a length), but not used; this is not a problem. One another hand, some optimization efforts are needed, in particular to avoid using too much temporary memory. We use the following allocation algorithm.

- When an output of the linear combinations is actually a copy of an input, we do not perform any operation: we simply reuse the input instead, in all other operations. Here for instance, the first linear combination

  $$L_0 = a_0, \quad L_1 = a_0 + a_1, \quad L_2 = a_1$$

  only computes one "real" output L_1. The corresponding function will only perform the addition.

- The intermediate results are stored in temporary memory, in successive slots of length either $h = \lfloor (N+1)/2 \rfloor$ or $2h - 1$, starting from the address given by the pointer `wk`.

  When one can determine that an memory area can be reused, we reuse it: here, after the recursive calls to the multiplication function, the first $2h$ entries of the workspace have become useless; we reuse them for the last linear combination.

- We can thus determine at code generation how much workspace will be needed in a single call to the function `kara_unsigned_long_rec` in length N; here, it is at most $4h \leq 2N$. The *total* amount for length N takes into account all levels of the recursion: here, it is at most

  $$2N + 2\frac{N}{2} + 2\frac{N}{4} + \cdots \leq 4N.$$

  In general, if the call in length N uses rN memory space, the overall amount will be

  $$rN + r\frac{N}{k} + r\frac{N}{k^2} + \cdots \leq rN\frac{k}{k-1}.$$

  This is the quantity which is used by the top-level function.

Linear combinations. We generate three linear combination functions; each of them has to perform one linear combination of polynomials. This is rather straightforward, as long as we are careful with the fact that not all polynomials may have the same length. Thus, there are to steps:

- the main loop, which goes up to the minimal length of the polynomials involved: for the first addition in Karatsuba, one polynomial has length h and the other $h' \leq h$, so the main loop will go up to h';
- a fix-up step, where we finish by taking care of the length differences: this step concerns at most k coefficients (which is 2 for Karatsuba).

The key part to optimize is the main loop. Here is the code generated for the Karatsuba case, for the first linear combination. This operation is simply an addition, so our code generator uses Algorithm 10 for modular addition.

```
void kara_unsigned_long1(unsigned long * __restrict__ b1,
                         const unsigned long * __restrict__ a0,
                         const unsigned long * __restrict__ a1,
                 const unsigned long ip, const unsigned long p, int N){
  int i;
  long __s_tmp;
  for(i=0; i<N; i++){
    __s_tmp = (long)(a0[i]) + (long)(a1[i]) -p;
    b1[i] = __s_tmp + ((__s_tmp>>63)&p);
  }
}
```

To optimize for speed, we implemented the option of unrolling these loops manually:

- we write specialized functions kara_unsigned_long1_1 up to kara_unsigned_long1_16 (assuming we unroll up to length 16), each of which dealing with fixed-length arguments;
- then, the function in length N follows the following pattern:
  - while $N > 16$, we call kara_unsigned_long1_16 and let $N = N - 16$
  - when we reach $N \leq 16$, call the corresponding function kara_unsigned_long1_N

Naive algorithm. Finally, we implemented the naive algorithm for the case of degrees up to 16. Our code for this case is generated automatically as well, so as to unroll loops: we found that the compiler we used was not doing a very good job by itself (due probably to the structure of the two nested loops of the naive algorithm).

For the naive algorithm, we do not perform modular reduction after each step: we first compute the whole result without any reduction, and apply the reduction in the end. In degree $< n$, this reduces the number of reductions from n^2 to n. However, this slightly reduces the possible size of the modulus: only 60-bit modulus can now be used.

4.3 Experimental results

This section gives the results of experiments made using our code. The experimentation platform is a Intel Core2 Duo CPU T7300 with 4Gb RAM; since the CPU clock speed can fluctuate, it was set to 800Mhz. Two compilers were used, `gcc` (the default choice) and `icc`; all parts of the code handling 128 bit integers were compiled with `gcc`, since `icc` does not support this type. The timings are in seconds, for 500 repetitions of the same computation.

Our experiments use Karatsuba's and Toom's multiplication algorithms; for Toom, we took $k = 3$, with evaluation points $0, 1, -1, 2, \infty$. We also use another less well-known algorithm of parameters $(3, 6)$ due to Winograd:

- $L_0 = a_0,\ L_1 = a_0 + a_1,\ L_2 = a_0 + a_2,\ L_3 = a_1 + a_2,\ L_4 = a_1,\ L_5 = a_2$
- $M_0 = b_0,\ M_1 = b_0 + b_1,\ M_2 = b_0 + b_2,\ M_3 = b_1 + b_2,\ M_4 = b_1,\ M_5 = b_2$
- $N_0 = L_0 M_0,\ \dots,\ N_5 = L_5 M_5$
- $c_0 = N_0,\ c_1 = N_1 - N_0 - N_4,\ c_2 = N_2 - N_0 + N_4 - N_5,\ c_3 = N_3 - N_4 - N_5,\ c_4 = N_5$

The complexity of Karatsuba's algorithm is $O(n^{\log_2(3)}) \simeq O(n^{1.59})$; that of Toom's algorithm is $O(n^{\log_3(5)}) \simeq O(n^{1.47})$. The complexity of Winograd's algorithm is $O(n^{\log_3(6)})$, so it should be slower than Karatsuba's and Toom's. However, the simple structure of the linear combinations suggested that it would be worthwhile to experiment with it.

For Karatsuba's and Winograd's algorithms, the linear combinations are easy to perform. More work is needed to find a short instruction sequence for Toom's algorithm. We mainly used an optimized form due to [3]; we will also give timings obtained with a more naive version.

Comparison between divide-and-conquer algorithms. We show in Figure 4.1 a comparison between the algorithms of Karatsuba, Toom and Winograd, for plain multiplication, using `unsigned long` data types (other types of multiplication behave similarly). As predicted, Winograd's algorithm does not perform very well. More

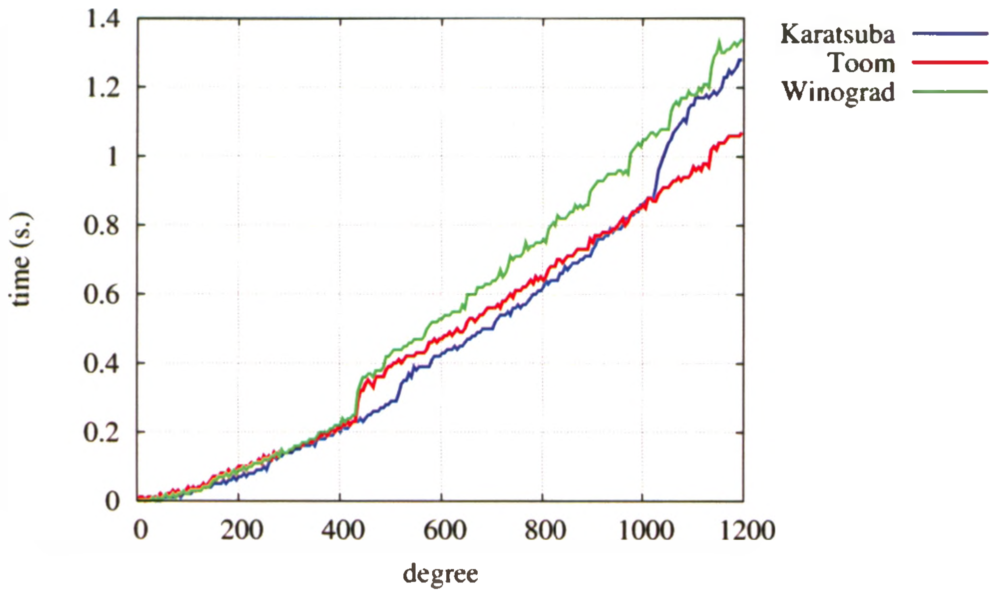

Figure 4.1: Comparison between divide-and-conquer algorithms

surprisingly, Toom's algorithm is competitive with Karatsuba's for small degrees; it takes the lead for degrees around 1000.

Comparison between data types. We show in Figure 4.2 a comparison between computations using `unsigned long` and `double` data types; we use Karatsuba's algorithm, for plain multiplication. As expected, operations with `double` coefficients are faster, but only by a factor of about 2, so that the modular arithmetic is close to being as fast as floating-point one. Again, we stress that the main reason we implemented a version for `double` coefficients of our code was specifically to perform this kind of comparison.

Comparison between multiplication types. Figure 4.3 gives a comparison between plain, transposed, square, short and short square multiplications. We use `unsigned long` data types and Karatsuba's algorithm; the results obtained with the other multiplication algorithms are similar. These graphs show that the transposed algorithms are slightly faster than their plain counterpart; the reasons are unclear to us as of now. The time for a short product is about 60% to 70% that of a plain product, similarly to what was observed in [19]. Finally, the square product and the short square are faster than their non-square counterparts, but not spectacularly so.

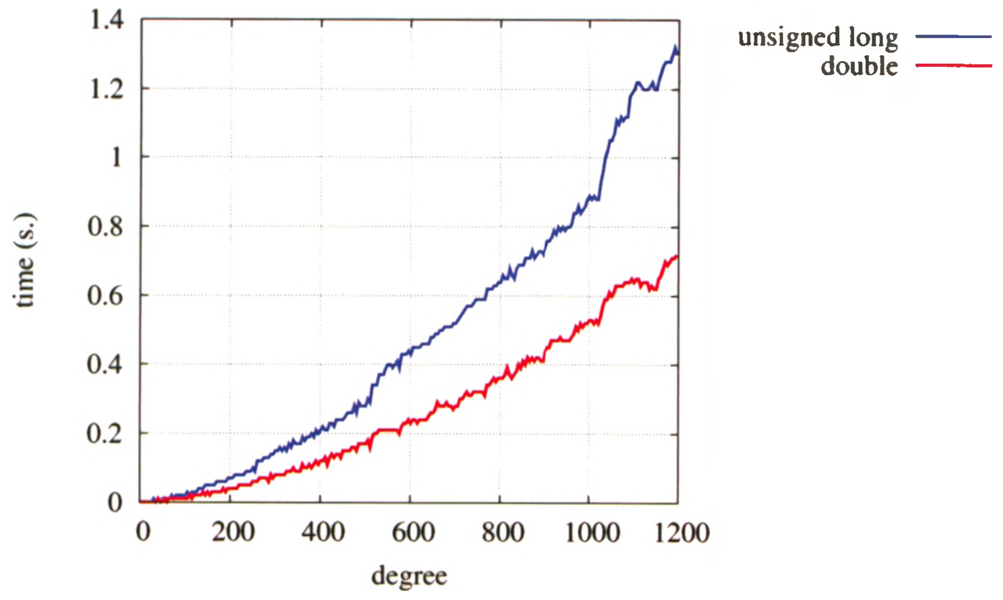

Figure 4.2: Comparison between data types

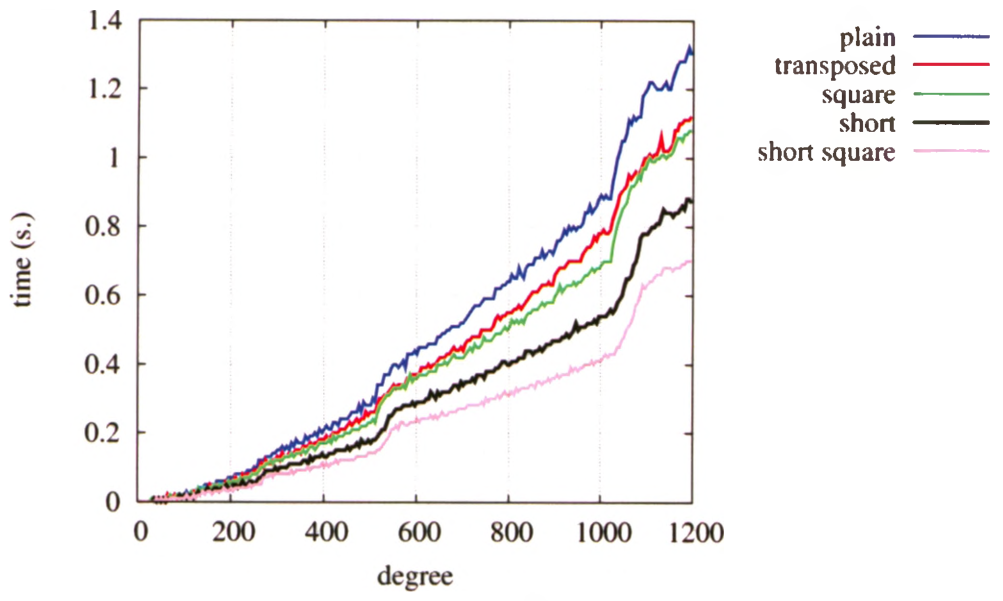

Figure 4.3: Comparison between multiplication types

Comparison between compilers. We show in Figure 4.4 a comparison between the timings obtained by `gcc` and `icc` for the plain multiplication, using Karatsuba's algorithm and `double` data types (recall that for the `unsigned long` data type, `icc` does not support 128 bit integers). As it turns out, there is a slight advantage for `icc`, by up to 7% at best.

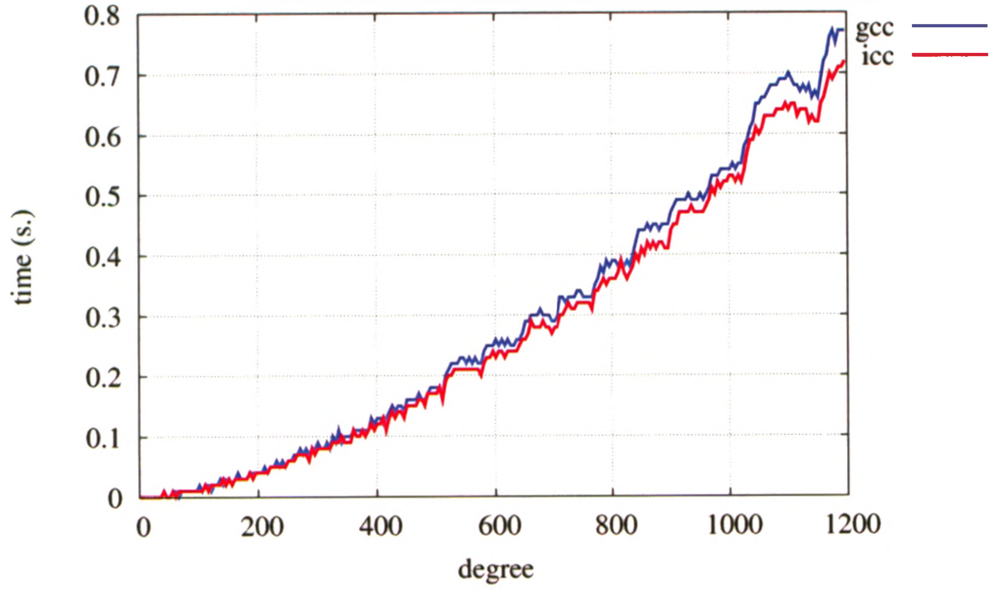

Figure 4.4: Comparison between compilers

Comparison between loop unrolling strategies. Next, we give in Figure 4.5 a comparison between computations using manual loop unrolling or letting the compiler do it for us. We use Karatsuba's algorithms, for plain multiplication, with `unsigned long` data type. As was to be expected, the compiler does a good job at unrolling the simple loops used in divide-and-conquer algorithms; we observe no significant difference in the timings (the curves completely overlap).

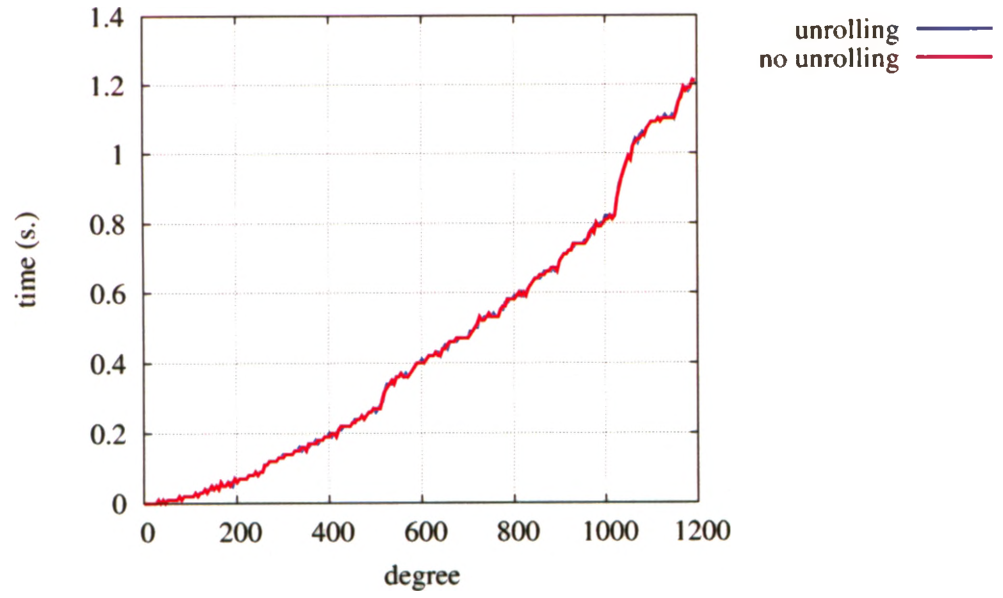

Figure 4.5: Comparison between loop unrolling strategies

Comparison between linear graphs. To understand the cost of the linear combinations, we show in Figure 4.6 a comparison between two ways of writing the third linear graph used in Toom's algorithm: the "optimized" one is the one we used by default, due to Bodrato *et.al.* [3]; the "naive" one is given in Figure 3.2 of Chapter 3. The graphs differ in the number of operations performed (the optimized one performs only one division by 3, whereas the naive one has one division by 3 and one division by 6). We use plain multiplication, with `unsigned long` data type. The advantage of using the optimized graph clearly appears.

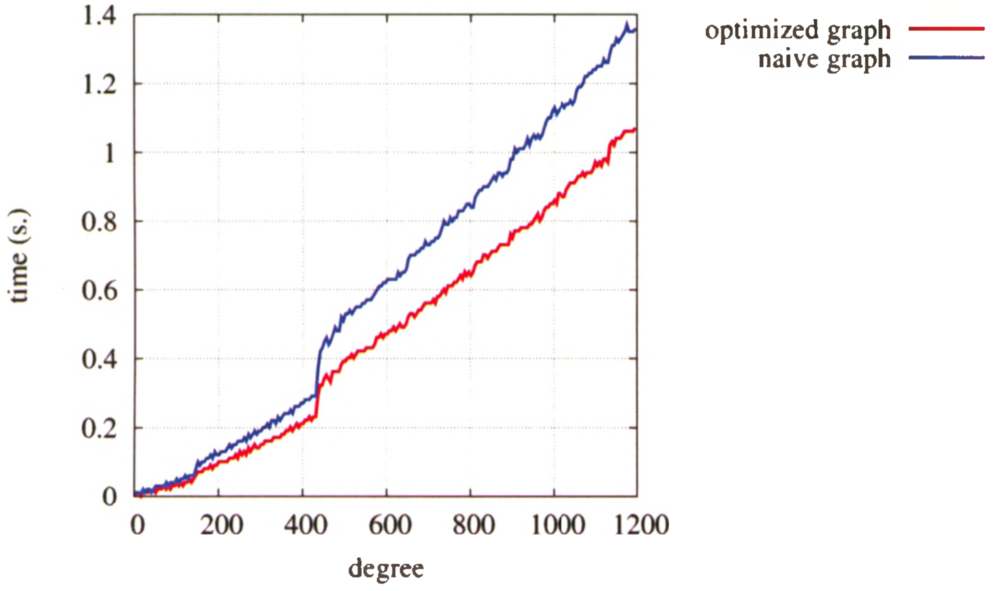

Figure 4.6: Comparison between linear graphs

Comparison with other systems. We compared our implementation with the fastest software known to us. For primes of size 64 bit, the library NTL [35] and the computer algebra system Magma [4] are available. NTL uses two representations for modular computations, `l_zz` for small primes and `ZZ` for larger ones. However, the primes we use in our benchmarks are too large for the `l_zz` type, which is limited to 52 bit. Thus, Figure 4.7 gives running times for Magma and for the `ZZ` data type in NTL. On the degree range we consider, our implementation is by far the fastest.

If one wants to consider smaller primes, other libraries come into play, in Figure 4.8. In addition to the `l_zz` data type in NTL, we can now make comparisons with the libraries modpn [25] and FLINT [20], which are limited to 32 bit primes as of now. Remark that modpn is oriented towards FFT multiplication, so that it supports

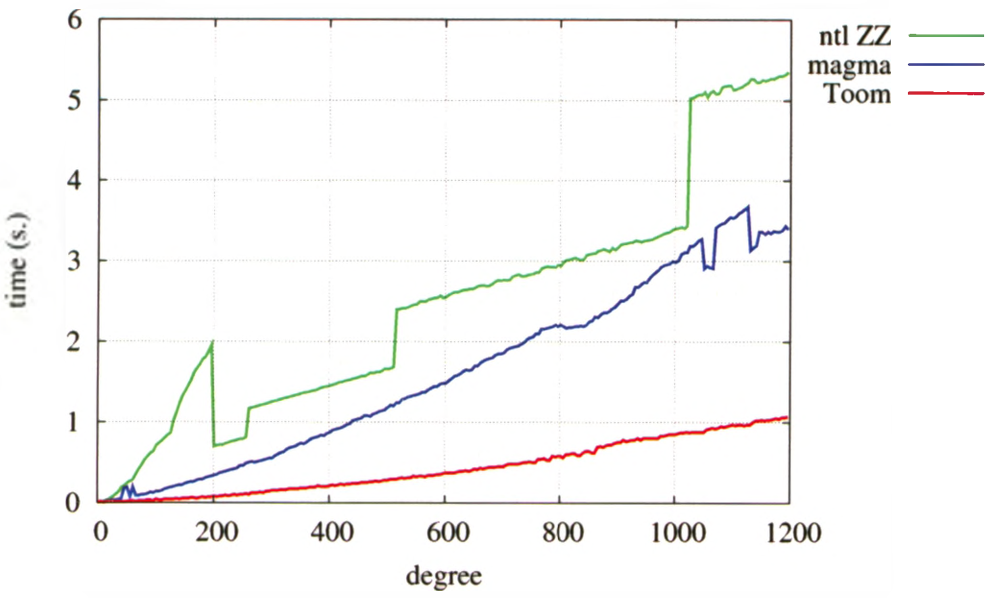

Figure 4.7: Comparison with other systems

only special values of p called "FFT primes". As expected, it performs better than our code (FFT is faster when applicable), but our code is quite close.

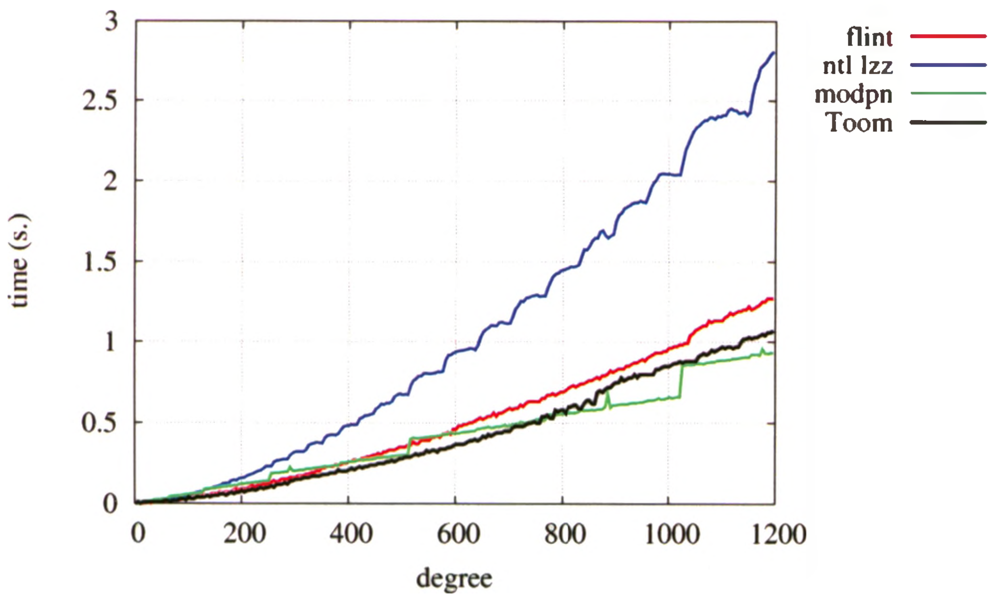

Figure 4.8: Comparison with other systems, 32 bit primes

Accuracy of `double` computations. The last question we consider is whether it makes sense to use such fast algorithms with `double` coefficients. We show in Figure 4.9 the relative error for a fairly typical example of the applications we have in mind, the product e^2, where $e = \sum_{i=0}^{100} x^i/i!$ is a truncation of the exponential power series. We show here the relative error on the coefficients up to degree 150, for the algorithms described up to now. Toom's algorithm performs very badly; Winograd's comes second, and Karatsuba's relative errors are smaller, though too large to make this practical. By comparison, we also show the relative error obtained using the naive algorithm: it is almost zero. Our results confirm previous ones: `double` coefficients are not well-suited to most divide-and-conquer algorithms.

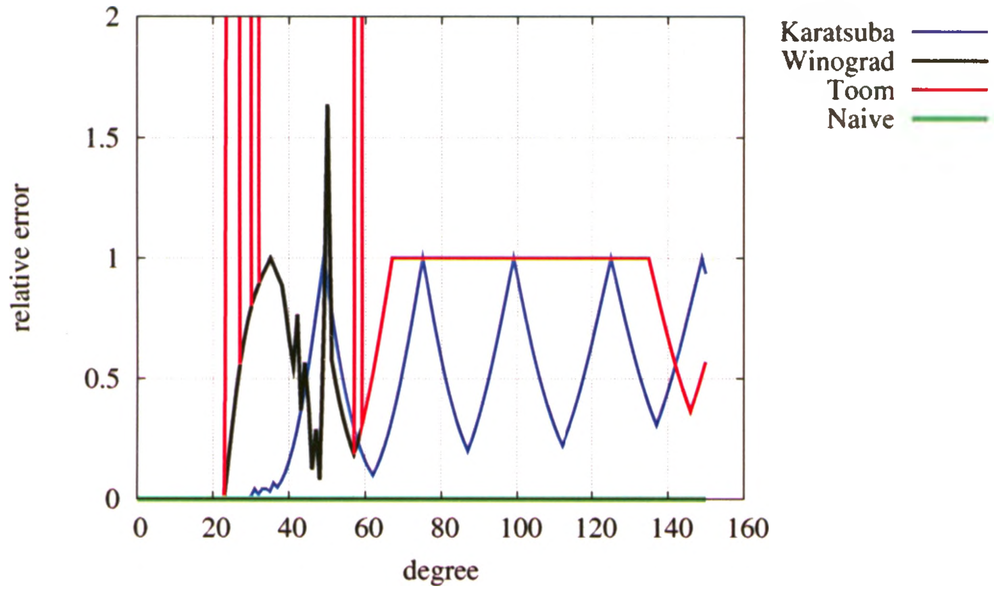

Figure 4.9: Relative error of `double` computations

Chapter 5

Newton iteration for differential equations

5.1 Introduction

This chapter reviews several uses of Newton iteration, with the aim of computing solutions of differential equations as *power series.* As intermediate steps, we present a sequence of applications of Newton iteration:

- computing inverses of power series,
- computing exponentials of power series,
- computing solution of first order linear differential equations,
- and finally computing solutions of first order non-linear differential equations.

We focus on the simplest case of equations of the first order: higher order equations are dealt with using more involved algorithms [39, 5]. The contents of this chapter are well-known; references are given within the text.

Power series. A *power series* is an infinite "formal" sum of the type

$$f = \sum_{i \geq 0} f_i x^i,$$

with coefficients f_i in a ring R. The set of all such power series is written $R[[x]]$. Since f is an infinite object, we can only handle *truncations* of it on a computer, as the finite sums

$$f \bmod x^n = \sum_{0 \le i < n} f_i x^i;$$

remark that the notation f mod x^n extends the one used for polynomials up to now.

Power series can be added, multiplied, etc. Since the truncations f mod x^n are actually polynomials, we can use the fast algorithms seen before to multiply them. In particular, truncated multiplication

$$f = gh \bmod x^n = (g \bmod x^n)(h \bmod x^n) \bmod x^n$$

can be done using the *short multiplication* of Chapter 3.

Operations on power series. A major difference between power series and polynomials is that power series can be inverted. If the constant coefficient f_0 is not zero, there exists a power series g such that $fg = 1$: for instance, with $f = 1 - x$, g is given by

$$g = 1 + x + x^2 + x^3 + x^4 + \cdots$$

More generally, one can compute other functions of f such as the *exponential* of f

$$\exp(f) = 1 + f + \frac{f^2}{2!} + \frac{f^3}{3!} + \cdots,$$

assuming that $f_0 = 0$ and that $1, 2, 3, 4, \ldots$ can be inverted in R. If R is a finite field $\mathbb{Z}/p\mathbb{Z}$, with p a prime number, we cannot compute infinitely many terms of $\exp(f)$, since $p = 0$ in R. However, it still makes sense to compute the first p terms of $\exp(f)$ in this case.

Even more generally, one can compute power series solutions of differential equations: for instance, the exponential of f is a solution of

$$y' = f'y.$$

More complex differential equations can be considered. In this chapter, we treat differential equations of the form

$$G(x, f', f) = 0,$$

for which we compute power series solutions f mod x^n, given suitable initial conditions. For instance, given the differential equation over $\mathbb{Z}/101\mathbb{Z}$

$$(x^6+x^4+1)f'(x)^2 = 1+75f(x)^4+16f(x)^6, \quad f(0)=0, \quad f'(0)=1,$$

we compute the solution f

$$f = x+68x^5+66x^7+60x^9+84x^{11} \bmod x^{12}.$$

This section present some previously known algorithms to perform the above computations, using a symbolic form of Newton iteration. The following paragraph provides a simple, yet fundamental, argument to show that such algorithms feature running times of the form $O(\mathsf{m}(n))$, with the notation of Chapter 3.

A useful property of the functions M and m. We defined in Chapter 3 some functions M and m such that polynomials of degree less than n can be multiplied in $\mathsf{M}(n)$ operations (resp. multiplied modulo x^n). To understand the running-time properties of Newton's iteration, we will use the following simple lemma on these functions.

Lemma 1. *Suppose that $\alpha > 1$ is such that*

$$\mathsf{M}(n) \le \frac{1}{\alpha}\mathsf{M}(2n) \quad \textit{and} \quad \mathsf{m}(n) \le \frac{1}{\alpha}\mathsf{m}(2n)$$

and let $\beta = \alpha/(\alpha-1)$. Then, for $k \ge 0$,

$$\mathsf{M}(1)+\mathsf{M}(2)+\cdots+\mathsf{M}(2^k) \le \beta\mathsf{M}(2^k)$$

and

$$\mathsf{m}(1)+\mathsf{m}(2)+\cdots+\mathsf{m}(2^k) \le \beta\mathsf{m}(2^k).$$

Proof. For M, one successively obtains

$$\mathsf{M}(2^{k-1}) \le \frac{1}{\alpha}\mathsf{M}(2^k), \quad \mathsf{M}(2^{k-2}) \le \frac{1}{\alpha^2}\mathsf{M}(2^k), \quad \ldots \quad \mathsf{M}(2^{k-h}) \le \frac{1}{\alpha^h}\mathsf{M}(2^k).$$

Summing over all h shows

$$\mathsf{M}(1)+\mathsf{M}(2)+\cdots+\mathsf{M}(2^k) \le (1+\frac{1}{\alpha}+\frac{1}{\alpha^2}+\cdots)\mathsf{M}(2^k),$$

and the sum on the right is bounded by $\beta\mathsf{M}(2^k)$. The proof for m is similar. $\square$

Remarks:

- This lemma shows that the sum of the costs $\mathsf{M}(\ell_i)$, where $\ell_i = 2^i$ for $i = 0, \dots, k$, is bounded by β times the cost of the last step $\mathsf{M}(\ell_k) = \mathsf{M}(2^k)$. We detailed the proof to show the basic idea used to obtain this result.

  In the further sections, we apply a slightly stronger result, to the case where ℓ_i are not necessarily powers of 2. The idea remains similar, but the proof becomes more technical: we will rely on Exercise 9.6 in [43], which justifies the same result in the general case.

- For $\mathsf{M}(n)$ of the form $\mathsf{M}(n) = cn^e$, with $e > 1$, we can take $\alpha = 2^e$; this gives for instance $\alpha = 3$ and $\beta = 3/2$ for Karatsuba multiplication. The same holds for m.

In all this chapter, the functions M and m and α, β are fixed, once and for all. For the moment, the operation count is naive: the next chapters will show what savings are possible.

5.2 Newton iteration in numerical analysis

In numerical analysis, Newton's method is an approximation of zeros of a differentiable function $P : \mathbb{R} \to \mathbb{R}$ [8]. The idea of this method is to *linearize* the equation, as illustrated in Figure 5.1:

- We start with an initial value x_n, which is reasonably close to the true root x.
- The function is approximated by its tangent line. One computes the x-intercept of this tangent line, which is the point x_{n+1}.
- x_{n+1} will typically be a better approximation to the function's root x than the initial value x_n, and the method can be iterated.

From the current approximation x_n, we can derive the formula for the next approximation x_{n+1}. We know from the definition of the derivative at a given point that it is the slope of a tangent at that point:

$$P'(x_n) = \frac{\Delta y}{\Delta x} = \frac{P(x_n) - 0}{x_n - x_{n+1}} \tag{5.1}$$

Hence,

$$x_{n+1} = x_n - \frac{P(x_n)}{P'(x_n)}. \tag{5.2}$$

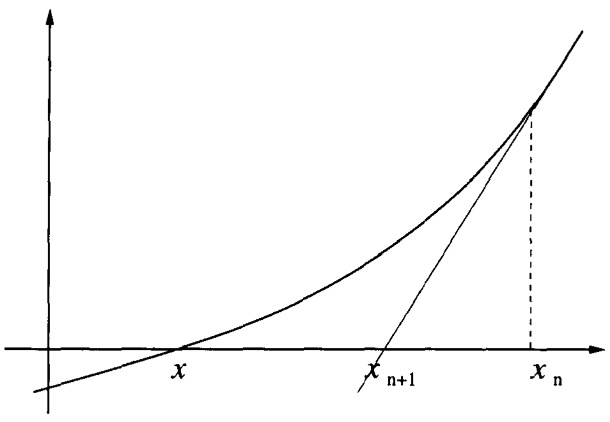

Figure 5.1: Newton's Iteration

As an example, consider the function $P : x \mapsto x^2 - 2$, and let $x_0 = 2$. The iteration is

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n};$$

the first terms are

$$\begin{aligned}
x_0 &= 2\\
x_1 &= \mathbf{1}.5\\
x_2 &= \mathbf{1.41}6666666666666666\\
x_3 &= \mathbf{1.41421}56862745098039\\
x_4 &= \mathbf{1.41421356237}46899106\\
x_5 &= \mathbf{1.41421356237309504 88}.
\end{aligned}$$

The limit is $\sqrt{2}$; we have indicated in bold face the correct digits of each approximate root. One sees that the number of correct digits approximately doubles at each iteration.

In the following sections, we will show how to use Newton iteration to perform symbolic computations. Instead of computing with floating-point numbers, we use *formal power series*; in this context, we will observe the same phenomenon of doubling the number of correct coefficients.

5.3 Power series inverse

Let R be our ring of coefficients, and let

$$f = \sum_{i \geq 0} f_i x^i \quad \text{and} \quad g = \sum_{i \geq 0} g_i x^i$$

be in $R[[x]]$, such that $f_0 = 1$ and $\widetilde{g} = 1/f$. Given the coefficients of f, the coefficients $g_0, g_1, \dots$ of $\widetilde{g}$ can be computed by the formula:

$$g_0 = \frac{1}{f_0}, \quad g_i = -\frac{1}{f_0}\sum_{j=1}^{i} f_i g_{i-j} \quad \text{for} \quad i \geq 1.$$

The cost to compute $g_0 = 1/f_0, \dots, g_n$ using this formula is $O(n^2)$. To do better, we use Newton iteration. Indeed, $\widetilde{g}$ is a solution of

$$P(\widetilde{g}) = 0, \quad \text{with} \quad P(\widetilde{g}) = \frac{1}{\widetilde{g}} - f.$$

Suppose that we only know $g = \widetilde{g} \bmod x^{\ell}$. Applying Formula (5.2), and after simplifications, one deduces that

$$G = g(2 - fg) \mod x^{2\ell}$$

satisfies $G = \widetilde{g} \bmod x^{2\ell}$, see [43]. For $\ell = 1$, one lets $g = 1/f_0$. Then, applying the previous formula for $\ell = 1, 2, \dots$, we obtain inverses of f modulo $x^2, x^4, x^8, \dots$. This result is classical; the idea appears in work of Sieveking [37] and Kung [23].

If we want $1/f$ modulo x^n, where n is a power of 2, we obtain it directly in this manner. If we want $1/f$ modulo x^n, where n is not a power of 2, we can compute it modulo $x^{n'}$, where n' is the smallest power of greater than or equal to n, and reduce the final result modulo x^n. However, this approach computes too many terms. Rather than computing unnecessary terms, the standard workaround [43, Ex. 9.6] is to compute the sequence $1/f \bmod x^{\ell_i}$, where the sequence $(\ell_i)_{i \geq 0}$ is defined backward as follows. Starting from $r = \lceil \log_2(n) \rceil$, we let $\ell_r = n$; assuming ℓ_i is defined, we let

- $\ell_{i-1} = \ell_i/2$ if ℓ_i is even
- $\ell_{i-1} = (\ell_i + 1)/2$ if ℓ_i is odd,

so that in both cases $\ell_i \leq 2\ell_{i-1}$, and we choose the smallest ℓ_{i-1} that satisfies this property. One can prove that in this case, ℓ_i is given by the formula $\ell_i = \lceil n/2^{r-i} \rceil$.

Algorithm 15 Inverse(f, n)

```
Input:  $f$ in $R[[x]]$, $n$, $f$ such that coeff$(f, 0) \neq 0$.
Output:  $g = 1/f \bmod x^n$
 1: $\ell = 1$, $i = 1$, $r = \lceil \log_2(n) \rceil$
 2: while $\ell < n$ do
 3:   if $\ell = 1$ then
 4:     $g = 1/\text{coeff}(f, 0)$
 5:   else
 6:     $t = fg \bmod x^{2\ell}$                          short product, $\mathsf{m}(2\ell)$
 7:     $u = 2 - t$                                                    $O(\ell)$
 8:     $g = gt \bmod x^{2\ell}$                          short product, $\mathsf{m}(2\ell)$
 9:   end if
10:   $\ell = \lceil n/2^{r-i} \rceil$, $i = i + 1$
11: end while
12: return $g \bmod x^n$
```

Lemma 2. *The cost of Algorithm 15 is* $2\beta\mathsf{m}(n) + O(n)$.

Proof. From Lemma 1, we easily obtain the result if n is a power of 2. The more general statement of the lemma is proved in [43, Ex. 9.6]. □

5.4 Exponential

The next task is to compute exponentials of power series. More precisely, given f in $R[[x]]$, we want to compute the "exponential-integral" of f, defined by

$$\widetilde{g} = \exp(F) = 1 + F + \frac{F^2}{2!} + \frac{F^3}{3!} + \cdots,$$

where $F = \int f$ is the integral of f with constant coefficient equal to zero. We mention a naive algorithm, then develop the fast version. Both algorithms use the fact that $\widetilde{g}'/\widetilde{g} = f$.

Rewriting the last equality as $\widetilde{g}' = f\widetilde{g}$, is it possible to compute the coefficients of $\widetilde{g}$ one after another, for a total cost of $O(n^2)$ to compute the first n of them: this is the naive algorithm.

The fast algorithm uses Newton iteration, in an indirect way. Assuming that we know only $g = \widetilde{g} \bmod x^\ell$, we look for the next approximation $G = \widetilde{g} \bmod x^{2\ell}$ in the form $G = g(1+h) \bmod x^{2\ell}$; h is to be determined, with $h = 0 \bmod x^{2\ell}$. The condition $G'/G = f \bmod x^{2\ell-1}$ gives

$$\frac{g'}{g} + \frac{h'}{1+h} = f \bmod x^{2\ell-1}.$$

Taking this equality modulo $x^{2\ell-1}$, we obtain

$$\frac{g'}{g} + h' = f \quad \bmod x^{2\ell-1},$$

since

$$\frac{h'}{1+h} = h'(1 - h + h^2 - h^3 + \cdots) = h' \bmod x^{2\ell-1}.$$

This gives h' modulo $x^{2\ell-1}$, then h modulo $x^{2\ell}$ by integration, and finally G. As in the previous section, when the target precision n is not a power of 2, one should avoid computing modulo $x, x^2, x^4, x^8, \ldots$, but use the successive precisions $(\ell_i)_{i\geq 0}$, with by $r = \lceil \log_2(n) \rceil$ and $\ell_i = \lceil n/2^{r-i} \rceil$.

Algorithm 16 Exponential(f, n)

```
Input: $f$ in $R[[x]]$, $n$
Output: $g = \exp(\int f) \bmod x^n$
1: $\ell = 1$, $i = 1$, $r = \lceil \log_2(n) \rceil$
2: while $\ell < n$ do
3:     if $i = 1$ then
4:         $g = 1$
5:     else
6:         let $g = g(1 + \int(f - \frac{g'}{g})) \bmod x^{2\ell}$        $2(\beta+1)\mathsf{m}(2\ell) + O(\ell)$
7:     end if
8:     $\ell = \lceil n/2^{r-i} \rceil$, $i = i + 1$
9: end while
10: return $g \bmod x^n$
```

Lemma 3. *The cost of Algorithm 16 is* $2\beta(\beta+1)\mathsf{m}(n) + O(n)$.

Proof. Lemma 2 shows that the inverse computation $1/g \bmod x^{2\ell}$ at step 6 takes time $2\beta\mathsf{m}(2\ell) + O(\ell)$; one does two extra multiplications modulo $x^{2\ell}$, for a total of $2(\beta+1)\mathsf{m}(2\ell) + O(\ell)$. We finish the proof using Lemma 1 in the case where n is a power of 2, and [43, Ex. 9.6] in general. □

Brent [8] first proved that power series exponentials could be computed in time $O(\mathsf{m}(n))$ using Newton iteration, and the computation of logarithms as an intermediate step. The presentation we give here is from Schönhage [31] and avoids the use of logarithms.

5.5 First order linear differential equations

Let now a, b, c be in $R[x]$, with $a(0) \neq 0$, and let γ be in R. We want to find the first n terms of the power series f such that

$$af' + bf = c \quad \text{and} \quad f(0) = \gamma.$$

As in the previous examples, it is possible to write a naive algorithm that computes the coefficients of f one after the other; the cost to obtain n coefficients is $O(n^2)$. Is is however possible to do better, using the following idea, due to Brent and Kung [9]. Let

$$d = \frac{b}{a} \bmod x^{n-1}, \quad e = \frac{c}{a} \quad \bmod x^{n-1}, \quad j = \exp(\textstyle\int d) \quad \bmod x^n.$$

Then f satisfies the relation

$$f = \frac{\gamma + \int ej}{j} \quad \bmod x^n.$$

Using the fast algorithms in the previous sections, $f \bmod x^n$ can thus be computed in time $O(\mathsf{m}(n))$. Precisely, we have the following bound.

Lemma 4. *The cost of computing* $f \bmod x^n$ *is* $2(\beta+1)(\beta+2)\mathsf{m}(n) + O(n)$.

Proof. Computing d and e takes time $2(\beta+1)\mathsf{m}(n) + O(n)$; computing j takes time $2\beta(\beta+1)\mathsf{m}(n) + O(n)$. Computing ej takes time $\mathsf{m}(n)$; multiplying $\gamma + \int ej$ by $1/j$ takes time $(2\beta+1)\mathsf{m}(n) + O(n)$. Simplifying the sum of costs gives the bound in the lemma. $\square$

5.6 First order non-linear differential equations

Let finally $G(x,t,u)$ be a polynomial in variables x, t, u, and let γ, η in R. We want to compute $\widetilde{f} \in R[[x]]$ such that

$$G(x, \widetilde{f}', \widetilde{f}) = 0 \quad \widetilde{f}(0) = \gamma, \quad \widetilde{f}'(0) = \eta; \tag{5.3}$$

we will assume that γ and η are such that a solution exists. The following algorithm works by linearization; one can see it as a form of Newton iteration adapted to this problem. This algorithm is due to Brent and Kung [9].

Let $\widetilde{f}$ be the solution of the previous equation. For $\ell \geq 0$, suppose that we know $f = \widetilde{f} \bmod x^{\ell}$; we look for a solution F with higher precision under the form $F = f + h$, where h is to be determined, such that $h = 0 \bmod x^{\ell}$. The equation

$$G(x, \widetilde{f}', \widetilde{f}) = G(x, f' + h', f + h) = 0$$

is converted by Taylor expansion to the linearized equation

$$\frac{\partial G(x, f', f)}{\partial u} h' + \frac{\partial G(x, f', f)}{\partial t} h = -G(x, f', f) \mod x^{2\ell-2}, \tag{5.4}$$

since all terms such as $h'^2, hh', h^2, \ldots$ are zero modulo $x^{2\ell-2}$. Therefore, we can apply the algorithm of the previous paragraph to compute h. Writing first

$$a = \frac{\partial G(x, f', f)}{\partial u} \mod x^{2\ell-2}, \quad b = \frac{\partial G(x, f', f)}{\partial t} \mod x^{2\ell-2},$$

$$c = -G(x, f', f) \mod x^{2\ell-2}$$

and next

$$d = \frac{b}{a} \mod x^{2\ell-2}, \quad e = \frac{c}{a} \mod x^{2\ell-2}, \quad j = \exp(\textstyle\int d) \mod x^{2\ell-1},$$

we obtain

$$h = \frac{\int ej}{j} \mod x^{2\ell-1},$$

and thus $F = f + h$ is known modulo $x^{2\ell-1}$. This time, starting from $f \bmod x^2$, we obtain solutions modulo $x^2, x^3, x^5, \ldots$, that is, x^{ℓ} for ℓ of the form $2^k + 1$.

For n arbitrary, we will compute the sequence $f \bmod x^{\ell_i}$, where now, $(\ell_i)_{i \geq 0}$ is defined backward as follows. Starting from $r = \lceil \log_2(n) \rceil$, we let $\ell_r = n$; assuming ℓ_i is defined, we let

- $\ell_{i-1} = (\ell_i + 1)/2$ if ℓ_i is odd,
- $\ell_{i-1} = \ell_i/2 + 1$ if ℓ_i is even.

The last value is $\ell_0 = 2$; in all cases, we have either $\ell_i = 2\ell_{i-1} - 1$ or $\ell_i = 2\ell_{i-1} - 2$. Since we do not know a closed-form formula for these ℓ_i, we assume to simplify the pseudo-code that they are given to us as input. Then, Algorithm 17 gives all the steps necessary to obtain $f \bmod x^n$; we use a function Eval to evaluate G and its derivatives at x, f', f.

```
Algorithm 17 DEnaive(G(x,t,u), $(\ell_i)_{i\geq 0}$, n, $\gamma$, $\eta$)
Input: $G(x,t,u), (\ell_i)_{i\geq 0}, n, \gamma, \eta$
Output: the solution $f$ of Eq.(5.3) modulo $x^n$
 1: $i = 0,\ \ell = \ell_0,\ f = \gamma + \eta x$
 2: while $\ell_{i+1} < n$ do
 3:     $a = \mathsf{Eval}(\partial G/\partial u, x, f', f) \bmod x^{2\ell-2}$
 4:     $b = \mathsf{Eval}(\partial G/\partial t, x, f', f) \bmod x^{2\ell-2}$
 5:     $c = -\ \mathsf{Eval}(G, x, f', f) \bmod x^{2\ell-2}$          $5L\mathsf{m}(2\ell) + O(M\ell)$
 6:     $u = \mathsf{Inverse}(a, 2\ell-2)$                              $2\beta\mathsf{m}(2\ell) + O(\ell)$
 7:     $d = ub \bmod x^{2\ell-2}$                                       $\mathsf{m}(2\ell)$
 8:     $e = uc \bmod x^{2\ell-2}$                                       $\mathsf{m}(2\ell)$
 9:     $j = \mathsf{Exponential}(d, 2\ell-1)$                          $2\beta(\beta+1)\mathsf{m}(2\ell) + O(\ell)$
10:     $k = \mathsf{Inverse}(j, 2\ell-1)$                              $2\beta\mathsf{m}(2\ell) + O(\ell)$
11:     $v = \int ej \bmod x^{2\ell-1}$                                  $\mathsf{m}(2\ell)$
12:     $h = kv \bmod x^{2\ell-1}$                                       $\mathsf{m}(2\ell)$
13:     $f = f + h,\ \ell = \ell_{i+1},\ i = i + 1$
14: end while
15: return $f \bmod x^n$
```

To estimate the complexity of this algorithm, we suppose that we are given G through a graph as in Section 2.2, and that this graph performs L multiplications and M other operations. Then, we have the following bound.

Lemma 5. *The cost of computing* $f \bmod x^n$ *is* $\big(2\beta(\beta+1)(\beta+2)+5\beta L\big)\mathsf{m}(n)+O(Mn)$.

Proof. For a fixed i, the cost of all steps is given within the algorithm; for simplicity, all costs are expressed using $\mathsf{m}(2\ell)$, though slightly better bounds such as $\mathsf{m}(2\ell-1)$ could be used. The proof for the total cost is similar to the ones seen before. □

Thus, the algorithm indeed has a cost of $O(\mathsf{m}(n))$, but the constant $2\beta(\beta+1)(\beta+2)+5\beta L$ in front of the $\mathsf{m}(n)$ is quite large. The next chapters show how to lower this cost.

Chapter 6

Improving Newton iteration

In this chapter, we introduce a first series of improvements to Newton iteration. We focus here on the main loop of Algorithm 17, and we postpone the question of evaluating G and its derivatives at steps 3, 4 and 5 to the next chapter. As intermediate steps, we re-examine all algorithms introduced in the previous chapter.

The algorithms of that chapter are already almost optimal, in the sense that they have complexity $O(\mathsf{m}(n))$: the improvements we give are in the constant factors. Some of these ideas were known before (using the middle product for power series inversion) and some are new; more precise comments are given within the following sections.

We start this section by introducing several new variants of polynomial multiplication: all of them can be achieved using a plain multiplication algorithm, but the algorithms we describe here have better complexity, by a constant factor. Turning to Newton iteration itself, our key idea will be to avoid computing or recomputing useless quantities, using these new variants of polynomial multiplication.

In particular, we will focus on how to *update* computations: if a and A are two polynomials, with $\deg(a) < n$ and $\deg(A) < 2n$, and if $a = A \bmod x^n$, we say that A is an *update* of a in degree $2n$. This means that we have

$$a = a_0 + \cdots + a_{n-1}x^{n-1}, \quad A = a_0 + \cdots + a_{n-1}x^{n-1} + a_nx^n + \cdots + a_{2n-1}x^{2n-1}.$$

In this kind of situation, the polynomial before update is in lower case, and the polynomial after update is in upper case.

6.1 Some variants of polynomial multiplication

Chapter 3 introduced plain, transposed and short polynomial multiplications; we showed how to obtain fast implementations of all of them using divide-and-conquer algorithms. We show here how to build further variants on top of the previous ones.

Middle product(s). Given A and C with $\deg(A) < n$ and $\deg(C) < 2n$, consider the product

$$D = CA = d_0 + d_1x + \cdots + d_{3n-2}x^{3n-2}.$$

Two quite similar notions of middle product can be defined. The first one considers the computation of the coefficients

$$d_{n-1} + \cdots + d_{2n-2}x^{n-1}. \tag{6.1}$$

This operation is similar to the transposed product, up to the reversing of one polynomial. Indeed, in Section 3.2, we pointed out that the former polynomial is just the transposed product of $\widetilde{C} = C \bmod x^{2n-1}$ and of

$$\widetilde{A} = a_{n-1} + \cdots + a_0x^{n-1};$$

the truncation of C is necessary, since the transposed product specifications require an argument of degree less than $2n-1$. The cost is thus that of the transposed product, that is, $\mathsf{M}(n) + O(n)$.

Algorithm 18 MiddleProduct(A, C, n)

```
Input: $A, C, n$, with $\deg(A) < n$ and $\deg(C) < 2n$.
Output: The polynomial $d_{n-1} + \cdots + d_{2n-2}x^{n-1}$ of Equation (6.1)
1: $\widetilde{A} = a_{n-1} + \cdots + a_0x^{n-1}$
2: $\widetilde{C} = C \bmod x^{2n-1}$
3: return $\widetilde{C}\widetilde{A}^t$
```

The second, similar form of middle product is concerned with the computation of the coefficients (note the shift by 1 unit)

$$d_n + \cdots + d_{2n-1}x^{n-1}. \tag{6.2}$$

It is solved quite similarly, up to modifying the definition of $\widetilde{C}$, writing now $\widetilde{C} = C \operatorname{div} x$.

Algorithm 19 MiddleProduct_2(A, C, n)

```
Input: $A, C, n$, with $\deg(A) < n$ and $\deg(C) < 2n$.
Output: The polynomial $d_n + \cdots + d_{2n-1}x^{n-1}$ of Equation (6.2)
1: $\widetilde{A} = a_{n-1} + \cdots + a_0 x^{n-1}$
2: $\widetilde{C} = C \text{ div } x$
3: return $\widetilde{C}\widetilde{A}^t$
```

The following remark justifies the interest of this discussion, say for the second form we just introduced. The naive way to compute the middle part of the product $D = AC$ consists in first computing D, then discarding the unwanted coefficients. Because C has twice as many coefficients as A, the good way of computing $D = AC$ is to write $C = C_0 + C_1 x^n$, and compute $AC_0 + x^n(AC_1 \bmod x^n)$. This costs $\mathsf{M}(n) + \mathsf{m}(n) + O(n)$ operations. Our previous discussion shows how to reduce the cost to $\mathsf{M}(n) + O(n)$, saving a factor of 2.

The middle product was introduced under this name by Hanrot, Quercia and Zimmermann [17], with the objective of improving several forms of Newton iteration (see the next section). However, the use of transposed algorithms for computing middle coefficients of products was already mentioned in e.g. [45].

Quarter multiplication. Given A and B, with $\deg(A) < 2n$ and $\deg(B) < 2n$, we want to compute $C = (AB \bmod x^{2n}) \text{ div } x^n$, that is, the coefficients of degrees $\{n, \ldots, 2n-1\}$ of AB. Since AB has degree up to $4n-2$, our output represents about a quarter of the coefficients of AB, hence the name. Algorithm 20 shows how to perform this task. We write $B = B_0 + B_1 x^n$ (so $B_0 = b$). We only need to compute the useful parts:

- the middle part of AB_0, by a middle product of the second type;
- the lower part of AB_1, by a short product.

Algorithm 20 QuarterProduct(A, B, n)

```
Input: $A, B, n$, with $\deg(A) < 2n$ and $\deg(B) < 2n$.
Output: $C = (AB \bmod x^{2n}) \text{ div } x^n$
1: write $B = B_0 + B_1 x^n$
2: $r = \text{MiddleProduct\_2}(B_0, A, n)$                $\mathsf{M}(n) + O(n)$
3: $v = AB_1 \bmod x^{n-1}$                                 $\mathsf{m}(n)$
4: return $r + v$                                           $O(n)$
```

The algorithm is illustrated in Figure 6.1. Its cost of is $\mathsf{M}(n)+\mathsf{m}(n)+O(n)$. This result should be compared to the cost of applying a short product, which is in this case $\mathsf{m}(2n)$. As far as we know, this trick did not appear before; a related idea for so-called "relaxed multiplication" is in [40].

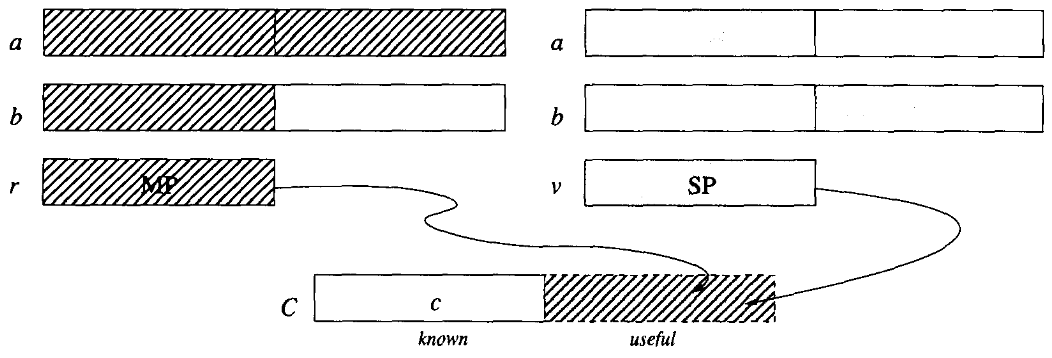

Figure 6.1: Quarter product

We point out the following application. Suppose that we have computed a short product $c = ab \bmod x^n$, where $\deg(a) < n$ and $\deg(b) < n$. Given updates A and B of a and b in degree $2n$, the previous algorithm enables us to compute $C = AB \bmod x^{2n}$ without recomputing c. We will denote this operation by $\mathsf{UpdateShort}(c, A, B, n)$; its cost is $\mathsf{M}(n) + \mathsf{m}(n) + O(n)$, as before.

Long-short multiplication. Given A and B, with $\deg(A) < n$ and $\deg(B) < 2n$, we want to compute $C = AB \bmod x^{2n}$. Remark the difference with the middle product: here, we are interested in all coefficients up to $2n$. This is done by writing $B = B_0 + B_1x^n$, with $\deg(B_0) < n$ and $\deg(B_1) < n$; then

$$AB \bmod x^{2n} = AB_0 + x^n(AB_1 \bmod x^n).$$

This gives Algorithm 21.

Algorithm 21 $\mathsf{LongShortProduct}(A, B, n)$

```
Input: $A, B, n$ with $\deg(A) < n$ and $\deg(B) < 2n$
Output: $C = AB \bmod x^{2n}$
1: write $B = B_0 + B_1x^n$
2: let $C_0 = AB_0$                                    $\mathsf{M}(n)$
3: let $C_1 = AB_1 \bmod x^n$                          $\mathsf{m}(n)$
4: return $C_0 + x^nC_1$                               $O(n)$
```

The cost is $\mathsf{M}(n)+\mathsf{m}(n)+O(n)$, whereas the naive approach consists in pretending that A has degree $2n$, for a cost of $\mathsf{M}(2n)$. The algorithm is illustrated in Figure 21.

Reverse short multiplication. Given A and B, with $\deg(A) < n$ and $\deg(B) < n$, we want to compute $C = AB \text{ div } x^n$, that is, the coefficients of degrees $\{n, \dots, 2n-2\}$ of AB. This is done by applying a short product to the reverse of the polynomials A and B, defined by

$$\widetilde{A} = a_{n-1} + \cdots + a_0 x^{n-1}$$

and

$$\widetilde{B} = b_{n-1} + \cdots + b_0 x^{n-1}.$$

Let further $\widetilde{C} = \widetilde{A}\widetilde{B} \bmod x^{n-1}$. One easily proves that if we write $\widetilde{C} = c_0 + \cdots + c_{n-2}x^{n-2}$, we have

$$C = c_{n-2} + \cdots + c_0 x^{n-2}. \tag{6.3}$$

This gives Algorithm 22.

Algorithm 22 ReverseShortProduct(A, B, n)

```
Input: $A, B, n$, with $\deg(A) < n$ and $\deg(B) < n$
Output: $C = AB$ div $x^n$
1: $\widetilde{A} = a_{n_1} + \cdots + a_0 x^{n-1}$
2: $\widetilde{B} = b_{n_1} + \cdots + b_0 x^{n-1}$
3: $\widetilde{C} = \widetilde{A}\widetilde{B} \bmod x^{n-1}$                    $\mathsf{m}(n)$
4: return $c_{n-2} + \cdots + c_0 x^{n-2}$ as in (6.3).
```

The cost is $\mathsf{m}(n-1)$, but we rather use $\mathsf{m}(n)$ to match the other notation. The naive approach multiplies A and B and extracts C from the result, for a cost of $\mathsf{M}(n)$.

6.2 Updating a power series inverse

Recall Newton's iteration for power series inverse in Section 5.3: knowing the inverse $g = 1/f \bmod x^n$, we compute $t = fg \bmod x^{2n}$, followed by $u = 2 - t$ and finally update g to $G = gu \bmod x^{2n}$.

By construction, the product $fg \bmod x^{2n}$ has the form $1 + x^n r$, with $\deg(r) < n$, so that $u = 2 - t = 1 - x^n r$. Thus, it is enough to compute the higher part r of t using a middle product. Then, G will be given by

$$G = gu \bmod x^{2n} = g - x^n(rg \bmod x^n).$$

Since the first n terms of G are known from the last step, it is thus sufficient to compute $v = rg \bmod x^n$ by a short product. This is illustrated on Figure 6.2.

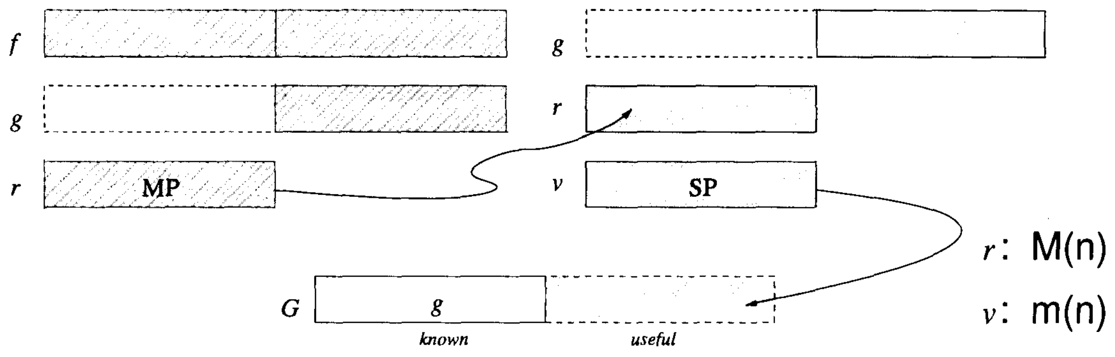

Figure 6.2: Updating inverse

```
Algorithm 23 UpdateInv(g, f, n)
Input: $f, g, n$, with $\deg(f) < 2n$, $\deg(g) < n$ and $g = 1/f \bmod x^n$.
Output: $G = 1/f \bmod x^{2n}$
1: $r = \mathsf{MiddleProduct\_2}(g, f, n)$        $\mathsf{M}(n) + O(n)$
2: $v = rg \bmod x^n$                              $\mathsf{m}(n)$
3: $G = g - x^n v$                                 $O(n)$
4: return $G$
```

Algorithm 23 uses $\mathsf{M}(n) + \mathsf{m}(n) + O(n)$ operations. This trick is well-known; it is described in detail in [17] for a general multiplication algorithm, but appeared before in [36] for the case of FFT multiplication.

6.3 Updating an exponential

In the Newton iteration for exponential of the previous chapter, we recomputed the inverse of g at each step. From one step to the next one, g is updated; so, its inverse should be updated as well, instead of completely recomputed. The updating function for exponential should now take f, $g = \exp(\int F) \bmod x^n$ and also the inverse $h = 1/g \bmod x^n$ as input; it will output an updated version $G = \exp(\int f) \bmod x^{2n}$ and its inverse $H = 1/G \bmod x^{2n}$. Recall that the original algorithm computes

$$G = g(1 + \int(f - \frac{g'}{g})) \bmod x^{2n}.$$

We detail the computations as

$$u = f - \frac{g'}{g} \bmod x^{2n-1}, \quad v = \textstyle\int u, \quad G = g + vG \bmod x^{2n}.$$

We would like to use the inverse $h = 1/g \bmod x^n$ to compute u, but this would require updating it to $1/g \bmod x^{2n-1}$, because the operations are done modulo x^{2n-1}. To solve this issue, we rewrite

$$u = f - \frac{g'}{g} \bmod x^{2n-1} = \frac{s}{g} \bmod x^{2n-1}, \quad \text{with} \quad s = gf - g' \bmod x^{2n-1}.$$

By assumption, s is zero modulo x^{n-1}, so we can write it as $s = x^{n-1}t$. Then,

$$\frac{s}{g} \bmod x^{2n-1} = \frac{x^{n-1}t}{g} \bmod x^{2n-1} = x^{n-1} \times \left(\frac{t}{g} \bmod x^n\right) = x^{n-1} \times \left(th \bmod x^n\right).$$

By construction, t consists of the coefficients of $fg - g'$ of degrees $n-1, \ldots, 2n$. Since $\deg(g') < n-1$, only the coefficients of fg are used, so we can compute t by the middle product between f and g.

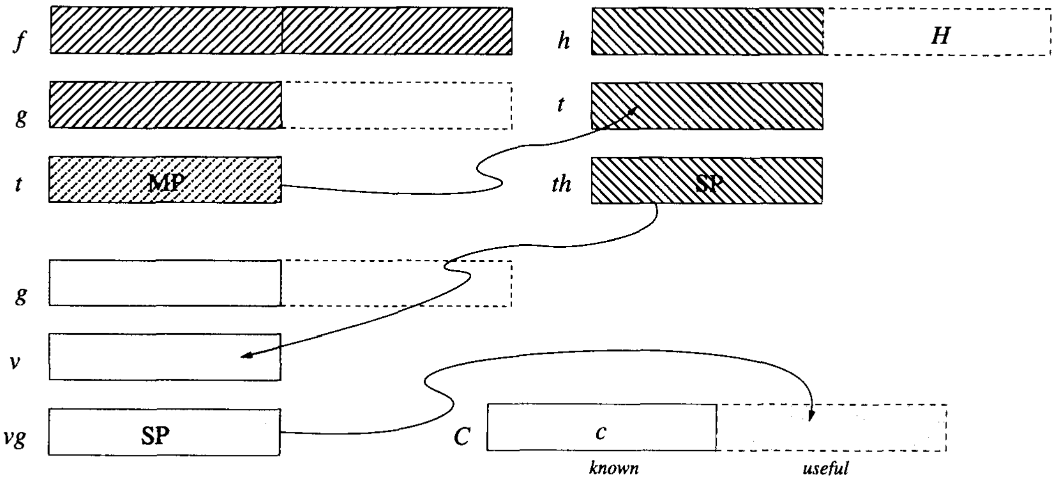

Figure 6.3: Updating exponential

Algorithm 24 UpdateExp(g, h, f, n)

```
Input:  $g, h, f, n$, with $\deg(f) < 2n$, $\deg(g) < n$ and $\deg(h) < n$
        $g = \exp(\int f) \bmod x^n$ and $h = 1/g \bmod x^n$.
Output: $G = \exp(\int f) \bmod x^{2n}$ and $H = 1/G \bmod x^{2n}$
1: $t = \mathsf{MiddleProduct}(g, f, n)$                $\mathsf{M}(n) + O(n)$
2: $u = x^{n-1}(th \bmod x^n)$                          $\mathsf{m}(n) + O(n)$
3: $v = \int u$                                         $O(n)$
4: $G = g + (vg \bmod x^{2n})$                          $\mathsf{m}(n)$
5: $H = \mathsf{UpdateInv}(h, G, n)$                    $\mathsf{M}(n) + \mathsf{m}(n) + O(n)$
6: return $G, H$
```

Algorithm 24 uses $2\mathsf{M}(n) + 3\mathsf{m}(n) + O(n)$ operations. The only non-obvious point is step 4, where we claim that we can compute $vg \bmod x^{2n}$ in time $\mathsf{m}(n)$. This is because $v = 0 \bmod x^n$, so we can write it in the form $v = x^n w$, whence $vg \bmod x^{2n} = x^n(wg \bmod x^n)$.

The idea using middle products for improving exponential computations is not new. The approach we present here appears in [18], with a complexity analysis specific to FFT multiplication.

6.4 Updating solutions to differential equations

To follow the ideas of the previous sections in the case of differential equations, we need to look closer where useless computations are done. In this section, we reuse all the notation of Algorithm 17.

The first main remark is that most computations can be done at precision $\ell - 1$ instead of $2\ell - 2$.

- $c = 0$ modulo $x^{\ell-1}$, so we can write it $c = x^{\ell-1}\tilde{c}$.
- Thus at step 8, $e = x^{\ell-1}\tilde{e}$, with $\tilde{e} = \tilde{c}u \bmod x^{\ell-1}$.
- Thus at step 11, $ej \bmod x^{2\ell-2} = x^{\ell-1}(\tilde{e}j \bmod x^{\ell-1})$.
- Thus, $v = \int ej \bmod x^{2\ell-1}$ has the form $v = x^{\ell}\tilde{v}$.
- Thus at step 12, $h = x^{\ell}(k\tilde{v} \bmod x^{\ell-1})$.
- This shows that it is actually enough to compute a, b, u, d, j, k modulo $x^{\ell-1}$; however, c is still needed modulo $x^{2\ell-2}$, since we need $\tilde{c}$.

The second improvement consists in updating all products, inverses and exponentials, not recompute them: in particular, this will avoid nested loops in the inverse and exponential computations.

Thus, we present hereafter a function that updates the solution f, as well as a, b, u, d, j, k. In view of Algorithm 17, we have either ℓ of the form $2\ell' - 1$, or of the form $2\ell' - 2$, for some integer ℓ'. Thus, we take as input f modulo x^ℓ and a, b, u, d, j, k modulo $x^{\ell'-1}$. We update f modulo $x^{2\ell-1}$ and a, b, u, d, j, k modulo $x^{\ell-1}$. To update a and b, the algorithm uses a function UpdateEval that computes at once

$$A = \frac{\partial G}{\partial t}(x, f', f) \bmod x^{\ell-1}, \quad B = \frac{\partial G}{\partial u}(x, f', f) \bmod x^{\ell-1}, \quad C = G(x, f', f) \bmod x^{2\ell-2}. \tag{6.4}$$

The optimizations we can bring for this task depend on the function G; this is the topic of the next chapter.

Algorithm 25 DEupdate(G(x, t, u), f, a, b, u, d, j, k, ℓ, ℓ')

```
Input:  the solution $f$ of Eq.(5.3) modulo $x^\ell$
        $a, b, u, d, j, k$ as above, of degree less than $\ell' - 1$
Output: the solution $F$ of Eq.(5.3) modulo $x^{2\ell-1}$
        $A, B, U, D, J, K$ as above, of degree less than $\ell - 1$
1:  $A, B, C = \mathsf{UpdateEval}(G, f, f', a, b, \ell - 1)$      $L\mathsf{m}(2\ell) + 4L\mathsf{m}(\ell) + O(M\ell)$
2:  $U = \mathsf{UpdateInv}(u, A, \ell' - 1)$                       $\mathsf{M}(\ell') + \mathsf{m}(\ell') + O(\ell')$
3:  $D = \mathsf{UpdateShort}(d, U, B, \ell' - 1)$                  $\mathsf{M}(\ell') + \mathsf{m}(\ell') + O(\ell')$
4:  $\widetilde{C} = C \operatorname{div} x^{\ell-1}$
5:  $E = U\widetilde{C} \bmod x^{\ell-1}$                           $\mathsf{m}(\ell) + O(\ell')$
6:  $J, K = \mathsf{UpdateExp}(j, k, D, \ell' - 1)$                 $2\mathsf{M}(\ell') + 3\mathsf{m}(\ell') + O(\ell')$
7:  $V = \int x^{\ell-1}EJ \bmod x^{\ell-1}$                        $\mathsf{m}(\ell) + O(\ell)$
8:  $H = KV \bmod x^{2\ell-1}$                                      $\mathsf{m}(\ell) + O(\ell)$
9:  $F = F + H$
10: return $F, A, B, U, D, J, K$
```

The cost of all steps in this algorithm follows from the previous sections, except for the evaluation at step 1. For that step, as before, we assume that we are given G through a DAG that performs L multiplications and M other operations; then, the cost reported in the first step is explained in the next chapter.

To give an overall estimate on the costs, we let α and β be as in the introduction of Chapter 5. Summing all terms and using the inequalities

$$\mathsf{M}(n) \leq \frac{1}{\alpha}\mathsf{M}(2n) \quad \text{and} \quad \mathsf{m}(n) \leq \frac{1}{\alpha}\mathsf{m}(2n),$$

we obtain that the total cost of the previous algorithm is

$$\frac{4}{\alpha^2}\mathsf{M}(2\ell) + (L + \frac{4L+3}{\alpha} + \frac{5}{\alpha^2})\mathsf{m}(2\ell) + O(M\ell).$$

As in Lemma 5, we deduce the cost of computing $f \bmod x^n$ starting from the initial conditions $f = \gamma + \eta x \bmod x^2$.

Lemma 6. *The cost of computing* $f \bmod x^n$ *is*

$$\frac{4\beta}{\alpha^2}\mathsf{M}(2n) + \beta(L + \frac{4L+3}{\alpha} + \frac{5}{\alpha^2})\mathsf{m}(2n) + O(n).$$

For example, for Karatsuba, we have $\alpha = 3$ and $\beta = 3/2$. Then, the bound of Lemma 5 becomes $(7.5L + 26.25)\mathsf{m}(n) + O(Mn)$, whereas we obtained here the much better estimate

$$0.666\ldots\mathsf{M}(n) + (3.5L + 2.333\ldots)\mathsf{m}(n) + O(Mn).$$

6.5 Experimental results

We conclude this chapter with two series of experimental results: computations of inverses and exponentials; the case of differential equations is left to the next chapter. In both cases, we compare the "naive" version of Newton iteration presented in the previous chapter to the "fast" one given here.

The following experiments use the polynomial multiplication code presented in Chapter 4; the experimentation machine is the same. For these experiments, we use Karatsuba's multiplication; the results obtained with Toom's algorithm are similar.

We start with inverses. Figure 6.4 compares timings for the direct implementation of Newton's iteration of Section 5.3 to the one using middle and short products given in Section 6.2 of this chapter: the latter behaves much better, as predicted. Besides, we give as a reference the cost of one multiplication, which turns out to be higher than that of inversion. Indeed, analysing the algorithm of Section 6.2 shows that the overall cost to compute an inverse using that algorithm is $\mathsf{M}(n)/2 + \mathsf{m}(n)/2 + O(n)$. This matches the behavior we observe.

We continue with exponentials. Figure 6.5 compares timings for the direct implementation of Newton's iteration of Section 5.4 to the one using middle and short products given in Section 6.3 of this chapter. As before, we give as a reference the

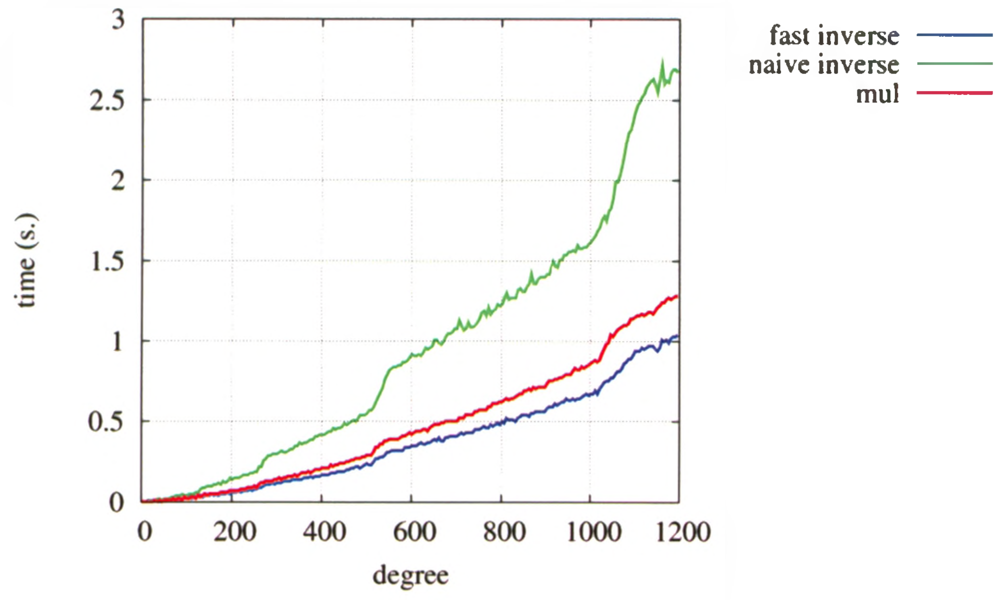

Figure 6.4: Power series inverse

cost of one multiplication; now, the algorithm of Section 6.3 has an overall cost of $\mathsf{M}(n) + 3\mathsf{m}(n)/2 + O(n)$. Again, this matches the behavior we observe.

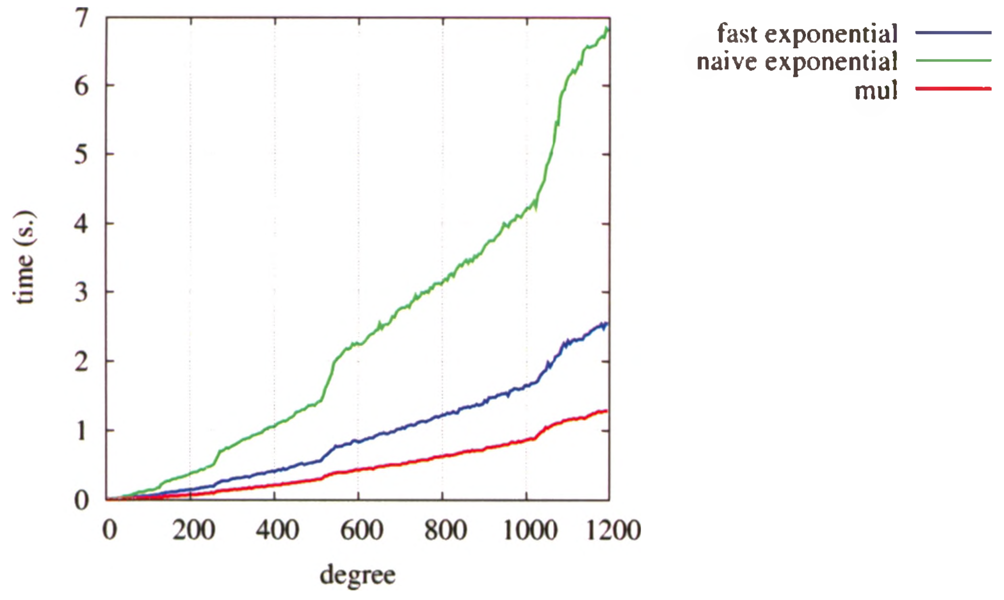

Figure 6.5: Power series exponential

Chapter 7

Evaluation techniques for Newton iteration

The final part of this thesis describes our code generation techniques for the evaluation of the function G and its derivatives. In the first section, we describe the input and output of our code generator; the second section describes the optimization techniques we use to reduce the amount of unnecessary computations. We conclude with experimental results.

To the best of our knowledge, the ideas presented here are new.

7.1 Overview of the code generator

In this section, we consider the following example: $G(x, f', f) = 0$, with

$$G(x,t,u) = (1+x+x^2)u^2 - (2+x)ut^2 - t^2 + 5u + 3.$$

Recall that to solve the above differential equation, Algorithm 25 is given a solution f at precision ℓ and computes

$$A = \frac{\partial G}{\partial t}(x, f', f) \bmod x^{\ell-1}, \;\; B = \frac{\partial G}{\partial u}(x, f', f) \bmod x^{\ell-1}, \;\; C = G(x, f', f) \bmod x^{2\ell-2}.$$

The polynomial C is computed by the graph $\mathcal{G}$ given in Figure 7.1 (the extra tags such as $N, 2N$ are explained in the next section). The series A and B are obtained by evaluating derivatives of G; these derivatives are computed by a graph $\mathcal{G}'$ which is obtained by automatic differentiation (Section 2.2). Since $\mathcal{G}'$ is more complicated than $\mathcal{G}$, we only show $\mathcal{G}$ here.

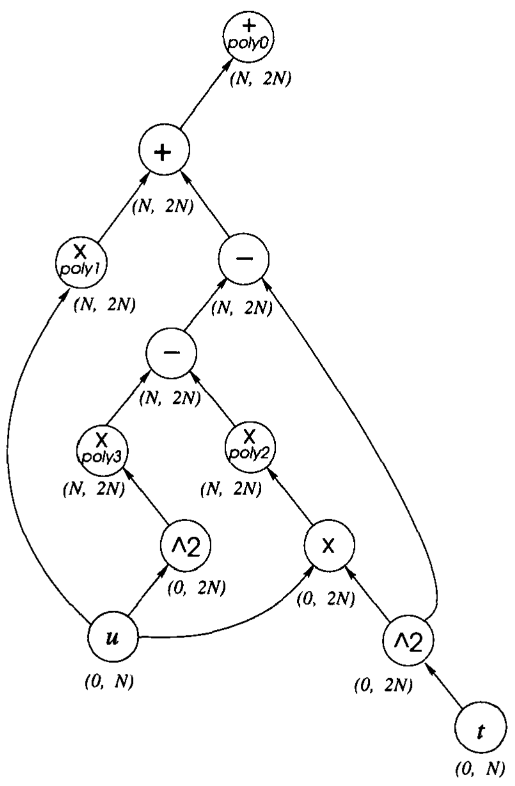

Figure 7.1: The graph $\mathcal{G}$ that computes G

We wrote in Java a code generator, which takes as input the graph $\mathcal{G}$, represented by its adjacency matrix, and the labels of the nodes. It performs automatic differentiation to get the graph $\mathcal{G}'$ for A and B, and outputs C code that evaluates A, B, C. As for multiplication, we support either `unsigned long` (*i.e.*, modular) or `double` coefficients; we explain the modular case (remark the suffix `unsigned_long` for all functions typed for `unsigned long` arguments).

Some important optimizations are performed at this level: they will be explained in the next section. Here, we show a few sample lines and explain the main steps of our C code.

- **Workspace allocation.**

  We allocate the memory which will be used as the workspace (for temporary results) at the beginning of the evaluation function. The size of the workspace is determined by the Java code; for the moment, little efforts were spent to reduce this size.

- **Initialization.**

  We initialize constants and polynomials given in the graph $\mathcal{G}$: `poly0` $=$ 3, poly1 = 5, poly2 = $2 + x$, poly3 = $1 + x + x^2$. They will be passed as arguments to the functions, when we reach the node containing these value. Some pre-multiplications are performed at this level; they are explained below.

- **Evaluation.**

  We evaluate C, A, B by following the graph $\mathcal{G}'$.

- **Cleaning.**

  We free all the memory for the workspace and given constants and polynomials.

```
void G_unsigned_long(unsigned long * __restrict__ C,
                     unsigned long * __restrict__ A,
                     unsigned long * __restrict__ B,
                     const unsigned long * __restrict__ t,
                     const unsigned long * __restrict__ u,
                     const unsigned long p, const unsigned long ip,
                     const unsigned long jp, int N){

/*------------ Workspace allocation ------------*/
unsigned long *wk=(unsigned long *) malloc(30*N*sizeof(unsigned long));

/*--------------- Initialization ---------------*/
unsigned long *poly0=(unsigned long *)malloc(1*sizeof(unsigned long));
unsigned long *poly0_pre=(unsigned long *)malloc(1*sizeof(unsigned long));

poly0[0]=3;
poly0_pre[0]=mulredcred(p, ip, jp, 3);
...
/*---------------- Evaluation C ---------------*/
mul_plain_unsigned_long(wk+N*0, u, u, p, ip,  N);
constant_mul_unsigned_long(wk+N*2, wk+N*0, poly3_pre, 3,
1*N, 2*N, 0*N, 2*N, p, ip);
...
constant_add_unsigned_long(C, wk+N*16, poly0, 1, 1*N, 2,
1*N, 2*N, p, ip);

/*---------------- Evaluation A ---------------*/
zero_unsigned_long(wk+N*18, p, ip, N);
add_unsigned_long(wk+N*19, t, t, p, ip, 0*N, 1*N);
...
sub_unsigned_long(A, wk+N*22, wk+N*23, p, ip, 0*N, 1*N);

/*--------------- Evaluation B --------------*/
add_unsigned_long(wk+N*24, u, u, p, ip, 0*N, 1*N);
constant_mul_unsigned_long(wk+N*25, wk+N*24, poly3_pre, 3,
0*N, 1*N, 0*N, 1*N, p, ip);
...
add_unsigned_long(B, wk+N*27, wk+N*29, p, ip, 0*N, 1*N);

/*---------- workspace, polys free -----------*/
  free(wk);
  free(poly0);
  free(poly0_pre);
...
}
```

This code can be compiled, using the multiplication code of Chapter 4 (or any other multiplication functions that support the same functionalities: product, middle product, short product), and used directly in our implementation of Algorithm 25. We use wrapper functions such as `mul_plain_unsigned_long` or `mul_short_long_unsigned_long`, which implement the variants of polynomial multiplication introduced in Section 6.1.

We also use use functions such as `constant_mul_unsigned_long` for multiplications by constant polynomials, which implement the naive algorithm (since our optimized multiplication functions assume that both arguments have the same length). The constant polynomials `poly0`, ... are defined in the beginning of the function, in two possible versions:

- `poly0` holds all coefficients as `unsigned long` (or `double` if this is the data type we use), for use in additions;

- if the data type is `unsigned long`, `poly0_pre` precomputes the product of this polynomial by 2^{64} modulo p, as in Chapter 4, for use in multiplications; otherwise it just copies `poly0`.

7.2 Precision issues in the evaluation

It remains to determine what precise variant of polynomial multiplication one should use for each multiplication we have to perform. We saw in the previous chapter that middle and short product were quite useful to avoid computing useless quantities. We will discuss here how to extend these ideas to the problem of evaluating G and its derivatives.

Recall that for a given integer ℓ, Algorithm 25 requires to perform the following evaluations:

- $A = \partial G/\partial t(x, f', f) \bmod x^{\ell-1}$,
- $B = \partial G/\partial u(x, f', f) \bmod x^{\ell-1}$,
- $C = G(x, f, f) \bmod x^{2\ell-2}$.

More precisely, for C, we only need the $\ell - 1$ coefficients of degrees $\{\ell - 1, \dots, 2\ell - 3\}$. Besides, remember that we know $a = A \bmod x^{\ell'-1}$ and $b = B \bmod x^{\ell'-1}$, with either $\ell = 2\ell' - 1$ or $\ell = 2\ell' - 2$.

The polynomials $\partial G/\partial t$ and $\partial G/\partial u$ are given by a graph $\mathcal{G}'$ obtained from the graph $\mathcal{G}$ of G through the differentiation process of Section 2.2; in particular, we compute the required coefficients of A, B, C at once, to share computations. Following the notation of the previous chapter, we assume that the graph $\mathcal{G}$ for G performs L multiplications; then, the graph $\mathcal{G}'$ performs $4L$ extra multiplications, for a total of $5L$.

Turning the graph into co; they are explained below.de. To compute A, B, C, we substitute $t = f'$ and $u = f$ in $\mathcal{G}'$ and follow the path from the inputs to the outputs, performing the operations labelling the vertices along the way; one safe way to compute A, B, C is to do all operations modulo $x^{2\ell-2}$. Formally, we assign to each vertex v of the graph a polynomial p_v, such that:

- if v is the leaf with label t, $p_v = f'$;
- if v is the leaf with label u, $p_v = f$;
- if v has two parents v_1 and v_2 and label $(+, v_1, v_2)$, $p_v = p_{v_1} + p_{v_2}$;
- if v has two parents v_1 and v_2 and label $(\times, v_1, v_2)$, $p_v = p_{v_1} p_{v_2} \bmod x^{2\ell-2}$;
- etc.

However, this process gives more than what is needed:

- the coefficients of A and B of degrees $\ell - 1, \ldots, 2\ell - 3$ are computed but not needed;
- the coefficients of C of degrees $0, \ldots, \ell - 2$ are computed but not needed either.; they are explained below.

We are going to show how to reduce the amount of coefficients we compute for all intermediate results.

To do so, we assign two quantities to each vertex v of $\mathcal{G}'$, a *low degree* ℓ_v and a *high degree* h_v, such that computing only the coefficients of p_v in degrees $\{\ell_v, \ldots, h_v - 1\}$ will be sufficient to obtain a correct output. As of now, our code supports ℓ_v and h_v of the form 0, $\ell - 1$ or $2\ell - 2$.

Assigning low- and high-degrees. The high-degrees are assigned by traversing the graph from inputs to outputs:

- both leaves t and u have high-degree $\ell - 1$ (since they have degree less than $\ell - 1$);
- the high-degree of any vertex used only to compute A or B is $\ell - 1$, since we need only $\ell - 1$ terms for A and B;
- the high-degree of a product used to compute C is $2\ell - 2$ (except when it is the product by a constant, where we keep the high-degree of the argument);
- the high-degree of a sum or difference used to compute C is the maximum of the high-degrees of the arguments.

The low-degrees are assigned by traversing the graph from outputs to inputs:

- the output C has low-degree $\ell - 1$; the outputs A and B have low-degrees 0;
- the two arguments of a product have low-degree 0 (except when it is the product by a constant, where we keep the low-degree of the result);
- the two arguments of a sum or a difference have the same low-degree as the result.

There is a possible issue in the assignment of low-degrees, since a given vertex can be used as argument for several operations: in this case, we keep the lowest of the low-degrees prescribed by the above rules. The extra tags such as $N, 2N$ in Figure 7.1 are the low- and high-degrees of all vertices, for the original graph $\mathcal{G}$ (remember, we display only $\mathcal{G}$ here, not $\mathcal{G}'$).

Applying the appropriate functions. Once the low- and high-degrees of all vertices are known, we can determine how to perform the operations at the vertices using this information. Additions and subtractions do not create any difficulties, and have a cumulative cost of $O(\ell)$; the delicate question is to determine how to perform multiplications. We will use here all variants of multiplication introduced in Section 6.1.

Parts of the answer are easy. If the result vertex v has high-degree $\ell - 1$, we can either perform an update of a short product, or simply recompute all of it, in output length $\ell - 1$ The cost is $\mathsf{M}(\ell') + \mathsf{m}(\ell') + O(\ell')$ with the first solution, and $\mathsf{m}(\ell)$ with the second solution. Remark that at least $4L$ of the multiplications in the graph $\mathcal{G}$ fall into this case.

If the result vertex v has high-degree $2\ell - 2$, more cases arise. Remark that our rules for computing low-degrees imply that in any case, the input vertices have low-degree 0.

Suppose first that v has low-degree 0; that is, we want all coefficients of p_v from degree 0 to $2\ell - 3$. We do this as follows.

1. Both input vertices have high-degree $\ell - 1$: plain multiplication, with inputs of degree $\ell - 1$.
2. Both input vertices have high-degree $2\ell - 2$: short multiplication modulo $x^{2\ell-2}$.
3. One input has high-degree $\ell - 1$ and the other high-degree $2\ell - 2$: short-long multiplication.

Suppose now that v has low-degree $\ell - 1$; that is, we want all coefficients of p_v from degree $\ell - 1$ to $2\ell - 3$. We do this as follows.

1. Both input vertices have high-degree $\ell - 1$: reverse short product in degree $\ell - 1$
2. Both input vertices have high-degree $2\ell - 2$: quarter product in degree $2\ell - 2$.
3. One input has high-degree $\ell - 1$ and the other high-degree $2\ell - 3$: short-long product.

The costs of these operations vary, but none of them goes above $\mathsf{m}(2\ell)$, so the total is at most $L\mathsf{m}(2\ell)$.

Thus, giving a precise estimate on the cost of the evaluation is impossible *a priori*, but in any case, the cost is no more than $L\mathsf{m}(2\ell) + 4L\mathsf{m}(\ell) + O(M\ell)$; this remark completes the proof of the cost estimate of the last chapter.

7.3 Experimental results

We conclude this chapter with experimental results. Our contributions in the last two chapters are improvements of Newton iteration for differential equations. Thus, for our experiments, we found that the most useful test was to compare two versions of Newton iteration: the "naive" implementation of Newton's iteration, and our "improved" one obtained by applying the optimizations described in this chapter and the previous one.

The experiments use the polynomial multiplication code presented in Chapter 4; the experimentation machine is the same. We used Karatsuba's multiplication; the results obtained with Toom's algorithm are similar. The timings are in seconds, for 500 repetitions of the same computation.

As in the previous chapter, the results show that all the optimizations presented here save a large constant factor in running time. For purposes of comparison, we give as well the time necessary for a single polynomial multiplication.

Example. We solve $G(x, f', f) = 0$ given at the beginning of this chapter:

$$G(x,t,u) = (1 + x + x^2)u^2 - (2 + x)ut^2 - t^2 + 5u + 3,$$

with initial conditions $f(0) = 1$ and $f'(0) = -2$. The comparison of running time is shown in Figure 7.2.

Example. We revisit the differential equation in Section 1.3,

$$(x^6 + x^4 + 1)f(x)^2 = 1 + 75f(x)^4 + 16f(x)^6,$$

with initial conditions $f(0) = 0$ and $f'(0) = 1$. Recall that this equation was introduced by Bostan *et.al.* [7]; our desire to prove a high-performance implementation for this equation was one of the starting points of this work. The experimental result

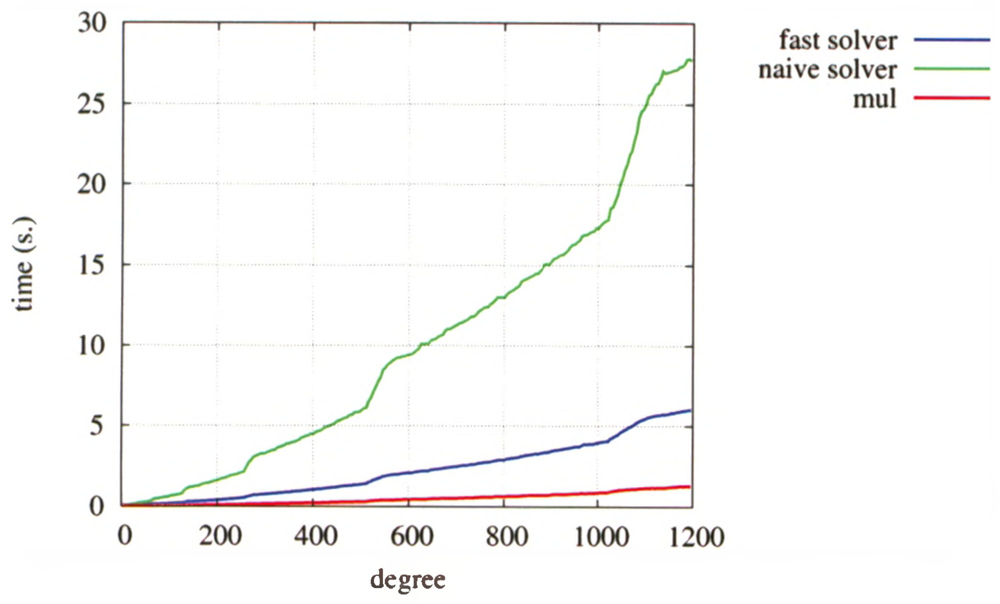

Figure 7.2: Solving a non-linear differential equation (Example 1)

in Figure 7.3 shows that we were successful. Here, the time for solving the equation is about 5 times that of multiplication.

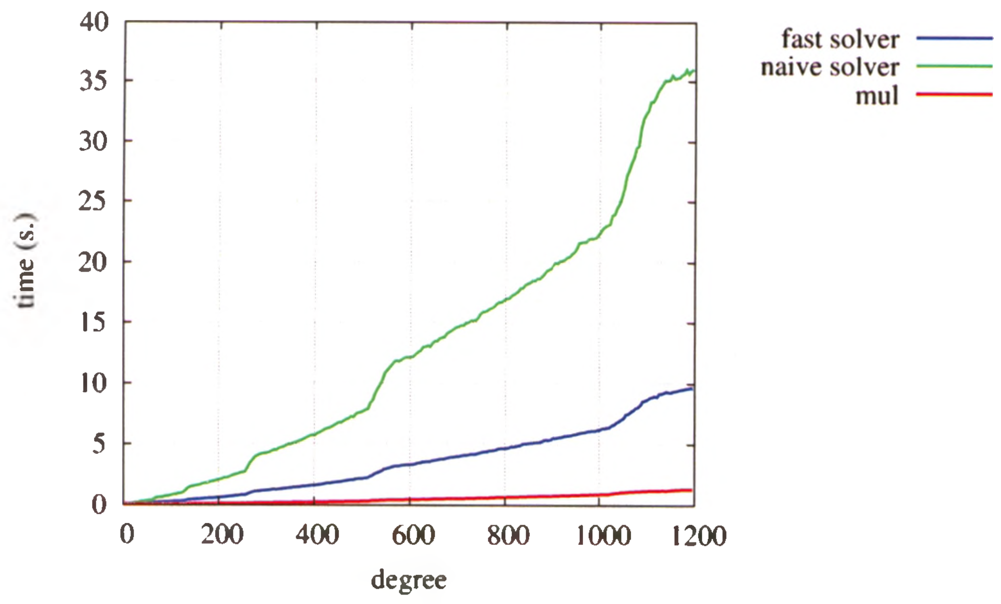

Figure 7.3: Solving a non-linear differential equation (Example 2)

Conclusion

This thesis demonstrated that code generation techniques can lead to high-performance implementations of low-level algorithms, such as polynomial multiplications, and higher-level ones, such as Newton iteration.

As to polynomial multiplication, more work is needed to assess whether e.g. our implementations are sufficiently cache-friendly, and if not, improve on such aspects. Besides, further families graphs should be considered, such as those involving $\sqrt{-1}$ in [45], over finite fields which contain a square root of -1.

As to Newton iteration, the main direction we want to explore is to determine whether the evaluation techniques of Chapter 7 can be reused in other contexts. A natural application would be Newton iteration for *polynomial equations.* The simplest versions of it amount to compute power series solutions of one or several polynomial equations; more advanced ones involve Newton iteration for e.g. *geometric resolutions* [16] or *triangular representations* [34]. The paper [25] presents a high-performance C implementation of the latter, which could possibly take advantage from our insights.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[2] D. J. Bernstein. Removing redundancy in high-precision Newton iteration. `http://cr.yp.to/papers.html#fastnewton`.

[3] M. Bodrato and A. Zanoni. Integer and polynomial multiplication: towards optimal Toom-Cook matrices. In *ISSAC'07*, pages 17–24, 2007.

[4] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.

[5] A. Bostan, F. Chyzak, F. Ollivier, B. Salvy, É. Schost, and A. Sedoglavic. Fast computation of power series solutions of systems of differential equations. In *SODA '07*, pages 1012–1021, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[6] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *ISSAC '03*, pages 37–44, New York, NY, USA, 2003. ACM.

[7] A. Bostan, F. Morain, B. Salvy, and É. Schost. Fast algorithms for computing isogenies between elliptic curves. *Mathematics of Computation*, 77(263):1755–1778, July 2008.

[8] R. P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic computational complexity*, pages 151–176. Academic Press, 1976. Proceedings of a Symposium held at Carnegie-Mellon University, Pittsburgh, Pa., 1975.

[9] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. ACM*, 25(4):581–595, 1978.

[10] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 1997. With the collaboration of Thomas Lickteig.

[11] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[13] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35(3), 2008.

[14] A. Filatei, X. Li, M. Moreno Maza, and É. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *ISSAC'06*, pages 93–100. ACM, 2006.

[15] P. Giorgi. *Arithmetic and algorithmic in exact linear algebra for the LinBox library*. PhD thesis, Ecole normale superieure de Lyon, 2004.

[16] M. Giusti, G. Lecerf, and B. Salvy. A Gröbner free alternative for polynomial system solving. *J. Complexity*, 17(2):154–211, 2001.

[17] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm. I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.

[18] G. Hanrot and P. Zimmermann. Newton iteration revisited. `http://www.loria.fr/~zimmerma/papers`.

[19] G. Hanrot and P. Zimmermann. A long note on Mulders' short product. *J. Symb. Comput.*, 37(3):391–401, 2004.

[20] W. Hart and D. Harvey. Flint: Fast library for number theory. `http://www.flintlib.org/`.

[21] À. Jorba and M. Zou. A software package for the numerical integration of ODEs by means of high-order Taylor methods. *Experiment. Math.*, 14(1):99–117, 2005.

[22] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Math. Dokl.*, 7:595–596, 1963.

[23] H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22:341–348, 1974.

[24] X. Li. Efficient management of symbolic computation with polynomials. Master's thesis, The University of Western Ontario, 2005.

[25] X. Li, M. Moreno Maza, R. Rasheed, and É Schost. The `Modpn` library: Bringing fast polynomial arithmetic into Maple. In *electronic proceedings, MICA '08*, 2008.

[26] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: from theory to practice. In *ISSAC'07*, pages 269–276. ACM, 2007.

[27] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[28] T. Mulders. On short multiplications and divisions. *Appl. Algebra Engrg. Comm. Comput.*, 11(1):69–88, 2000.

[29] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. 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, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[30] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, 1981.

[31] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity. Technical report, Mathematisches Institut der Universität Tübingen, 1982. Preliminary report.

[32] A. Schönhage. Variations on computing reciprocals of power series. *Inform. Process. Lett.*, 74:41–46, 2000.

[33] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[34] É. Schost. Computing parametric geometric resolutions. *Appl. Algebra Engrg. Comm. Comput.*, 13(5):349–393, 2003.

[35] V. Shoup. A library for doing number theory. `http://www.shoup.net/ntl/`.

[36] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[37] M. Sieveking. An algorithm for division of powerseries. *Computing*, 10:153–156, 1972.

[38] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akad. Nauk USSR*, 150(3):496–498, 1963.

[39] J. van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Comput.*, 34(6):479–542, 2002.

[40] J. van der Hoeven. Relaxed mltiplication using the middle product. In *ISSAC'03*, pages 143–147. ACM, 2003.

[41] J. van der Hoeven. Newton's method and FFT trading. Technical Report 2006-17, Université Paris-Sud, 2006. `http://www.math.u-psud.fr/~vdhoeven/`.

[42] F. von Haeseler and W. Jürgensen. Irreducible polynomials generated by decimations. In *Finite fields and applications (Augsburg, 1999)*, pages 224–231. Springer, Berlin, 2001.

[43] J. von zur Gathen and J. Gerhard. *Modern computer algebra.* Cambridge University Press, Cambridge, second edition, 2003.

[44] S. M. Watt. A fixed point method for power series computation. In *ISSAC'88*, number 358 in LNCS, pages 206–217. Springer Verlag, 1988.

[45] S. Winograd. *Arithmetic complexity of computations*, volume 33 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pa., 1980.