TOWARD HIGH-PERFORMANCE POLYNOMIAL SYSTEM SOLVERS BASED ON TRIANGULAR DECOMPOSITIONS

(Spine title: Contributions to Polynomial System Solvers)

(Thesis format: Monograph)

by

Xin Li

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada
April, 2009

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

CERTIFICATE OF EXAMINATION

The thesis by

Xin Li

entitled:

Toward High-performance Polynomial System Solvers Based on Triangular Decompositions

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Date ______________________ Chair of the Thesis Examination Board

Abstract

This thesis is devoted to the design and implementation of polynomial system solvers based on symbolic computation. Solving systems of non-linear, algebraic or differential equations, is a fundamental problem in mathematical sciences. It has been studied for centuries and still stimulates many research developments, in particular on the front of high-performance computing.

Triangular decompositions are a highly promising technique with the potential to produce high-performance polynomial system solvers. This thesis makes several contributions to this effort.

We propose asymptotically fast algorithms for the core operations on which triangular decompositions rely. Complexity results and comparative implementation show that these new algorithms provide substantial performance improvements.

We present a fundamental software library for polynomial arithmetic in order to support the implementation of high-performance solvers based on triangular decompositions. We investigate strategies for the integration of this library in high-level programming environments where triangular decompositions are usually implemented.

We obtain a high performance library combining highly optimized C routines and solving procedures written in the MAPLE computer algebra system. The experimental result shows that our approaches are very effective, since our code often outperforms pre-existing solvers in a significant manner.

Acknowledgments

While my name is the only one that appears on the author list of this thesis, there are several other people deserving recognition. I would like to express my sincere appreciation to my supervisors, Dr. Marc Moreno Maza and Dr. Stephen M. Watt, for their guidance, support, encouragement and friendship through my entire Ph.D. study. I wish to extend my appreciation and gratitude to Dr. Marc Moreno Maza for introducing me to these interesting and challenging projects.

I would also like to express my sincere appreciation to my dear colleagues Éric, Yuzhen, Wei, Changbo, Filatei, Liyun, Raqeeb, and all the members from the ORCCA lab for their great help to my research.

Finally, I hope to share my happiness of the achievement from my Ph.D. study with my dear parents, sister and all my loved ones.

Without anyone of you, I couldn't reach the point where I am today. Thank you guys!

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis is devoted to the design and implementation of polynomial system solvers based on symbolic computation. Solving systems of non-linear, algebraic or differential equations is a fundamental problem in mathematical sciences. It has been studied for centuries and still continues to stimulate research.

Solving polynomial systems is also a driving subject for symbolic computation. In many computer algebra systems, the `solve` command involves nearly all libraries in the system, challenging the most advanced operations on matrices, polynomials, algebraic numbers, polynomial ideals, etc.

Symbolic solvers are powerful tools in scientific computing: they are well suited for problems where the desired output must be exact and they have been applied successfully in mathematics, physics, engineering, chemistry and education, with important outcomes. See Chapter 3 in [48] for an overview of these applications. While the existing computer algebra systems have met with some practical success, symbolic computation is still under-utilized in areas like mathematical modeling and computer simulation. Part of this is due to the fact that much larger and more complex computers are required - often beyond the scope of existing systems.

The implementation of symbolic solvers is, indeed, a highly difficult task. Symbolic solvers are extremely time-consuming when applied to large problems. Even worse, intermediate expressions can grow to enormous size and may halt the computations, even if the result is of moderate size [45]. Therefore, the implementation of symbolic solvers requires techniques that go far beyond the manipulation of algebraic or differential equations; these include efficient memory management, data compression, parallel and distributed computing, etc.

The development of polynomial system solvers, as computer programs based on symbolic computation, started four decades ago with the discovery of Gröbner bases in the Ph.D. thesis of B. Bucherger [20], whereas efficient implementation capable of tackling *real-world applications* is very recent [39].

Triangular decompositions are an alternative way for solving systems of algebraic equations symbolically. They focus on extracting geometrical information from the solution set $V(F)$ of the input polynomial system F rather than insisting on revealing algebraic properties as Gröbner bases do. A triangular decomposition of $V(F)$ is given by finitely many polynomial sets, each of them with a triangular shape and so-called a triangular set[1]; these sets describe the different components of $V(F)$, such as points, curves, surfaces, etc. Triangular decompositions were invented by J.F. Ritt in the 30's for systems of differential polynomials [81]. Their stride started in the late 80's with the method of W.T. Wu dedicated to algebraic systems [91]. Different concepts and algorithms extended the work of Wu. At the end of 90's the notion of a regular chain, introduced independently by M. Kalkbrener in [55] and by L. Yang and J. Zhang in [92], led to important algorithmic improvements, such as the Triade algorithm (for TRIAngular Decompositions) by M. Moreno Maza [75]. The era of polynomial system solvers based on triangular decompositions could commence.

Since 2000, exciting complexity results [29] and algorithms [27] have boosted the development of implementation techniques. From these works, triangular decompositions appear at the start of this Ph.D. thesis as highly promising techniques with the potential to produce high-performance solvers. The goals of the proposed research were then the following ones.

(1) Develop a high performance software library for polynomial arithmetic in order to support the implementation of high-performance solvers based on triangular decompositions.

(2) Integrate these routines in high-level programming environments where triangular decompositions are implemented.

(3) Design theoretically and/or practically efficient algorithms, based on the asymptotically fast algorithms and modular methods, for the key routines on which triangular decompositions rely.

(4) Evaluate the performances, including speed-up factors and bottlenecks, of this approach and compare it with the pre-existing polynomial system solvers.

[1]This notion extends to non-linear systems that of a triangular system, well-known in linear algebra.

With these goals in mind, we have developed the research directions described in the next section.

1.2 Research Directions

Polynomial arithmetic is at the foundation of our research subject. Since the early days of computer algebra systems, one of the main focuses has been on sparse polynomial arithmetic and classical algorithms (by opposition to asymptotically fast ones). A tentative explanation for this historical fact is given in the overview of Chapter 3. In the last decade, asymptotically fast algorithms for dense polynomial arithmetic have been proved to be practically efficient. Among them are FFT-based algorithms which are well adapted for supporting operations like the Euclidean division, Euclidean Algorithm and their variants which are at the core of methods for computing triangular decompositions. Indeed, these types of calculations tend to make intermediate data dense even if input and output polynomials are sparse; thus "dense methods" like FFT-based polynomial multiplication and division fit well in this context. It was, therefore, necessary to invest significant effort on these algorithms, which we actually started in the Masters thesis [64]. The theoretical and experimental results from this thesis are beyond our initial expectation.

Certainly, fast algorithms and high-performance are always popular topics. The SPIRAL [79] and FFTW [42] projects are well-known high-performance software packages based on FFT techniques for numerical computation and with application to areas like digital signal processing. A central feature of these packages is automatic code tuning for generating architecture-aware highly efficient code. In the case of our library for symbolic computation with polynomials, this feature remains future work. FFTs in computer algebra are primarily performed over finite fields leading to a range of difficulties which do not occur over the field of complex numbers. For instance, primitive roots of unity of large orders may not always exist in finite fields, making the use of the Cooley-Tukey Algorithm [25] not always possible. For this reason and others, such as performance considerations, each FFT-based polynomial operation (multiplication, division, evaluation, interpolation, etc.) has several implementations. Although we have not reached yet the level of automatic tuning, hand-tuning was used a lot. We have considered specific hardware features such as memory hierarchy, pipelining, vector instructions. We tried to write compiler-friendly code, relying on compiler optimization to generate highly efficient code. We have also considered the parallelization of polynomial arithmetic (mainly multiplication) and higher-level operations (normal form computations). Section 3.3 and Section 3.4 describe our implementation whereas Section 3.5 at Page 40 presents comparative experimental results. Chapter 9 is dedicated to the parallelism study.

Developing a fundamental high-performance software library for polynomial arithmetic appeared to be a necessary task. At the time of starting this work, there were no such packages that we could extend or build on. All the existing related software had either technical limitations (as was the case for the NTL [6] library, limited by its univariate arithmetic and the characteristic of its fields of coefficients) or availability issues (as was the case for the MAGMA [5] computer algebra system, that doesn't make it a research tool which is not open source). Developing this fundamental high-performance software library was also motivated by the desire of adapting it to our needs. For instance, when we started to develop higher-level algorithms, such as normal form computations, see Chapter 6, adjustments in our multivariate multiplication had to be done.

Implementing a polynomial system solver based on triangular decompositions from scratch was, however, out of question. First, because better subroutines for these decompositions had to be designed, such as those presented in Chapters 6 and 7, before engaging an implementation effort. Secondly, because the amount of coding would simply be too large for work of this scale. Polynomial system solvers such as `FGb` [40] or the command `Triangularize` in the `RegularChains` library [63] are the results of 20 and 16 years of continuous work respectively! Last, but not least, implementation techniques and programing environments are evolving quickly these days, stimulated by progress in hardware acceleration technologies. In order to avoid developing code that could quickly become obsolete, we were looking into strategies driven by code modularity and reusability. This led us to consider integrating our fundamental high-performance software library for polynomial arithmetic, written in the C programing language, into higher-level algorithms written in the computer algebra systems AXIOM and MAPLE. (These are presented in Section 3.2.2 and Section 8.2.3 respectively.) The overhead of data conversion between different polynomial representations from different language levels may be significant enough to slow down the whole computation. Thus, this technique of *mixing code* brings extra difficulties to achieve high performance for those applications involving frequent cross language-level data mapping (see Section 3.4.1 and Section 8.2 for details). Each of AXIOM and MAPLE has its specifies on this front.

AXIOM is a multiple-language-level system (see Section 2.2 for details). We took advantage of this feature for combining in the same application different polynomial data types realized at different language levels. Chapter 4 reports on this investigation and stresses the fact that selecting suitable polynomial data types is essential toward high performance implementation.

With MAPLE, we have focused on the integration of our C library with high-level algorithms. Our goal was to provide support and speed-up for the *RegularChains* library and its commands for computing triangular decompositions of polynomial systems. Since the technique of *mixing code* is much more challenging in the context of MAPLE than within AXIOM (See Chapter 8 for details) this objective is not guaranteed to be successful. In fact, we asked ourselves the following questions while designing this *mixing code* framework: to which extent can triangular decomposition algorithms (implemented in the MAPLE `RegularChains` library) take advantage of fast polynomial arithmetic (implemented in C)? What is a good design for such hybrid applications? Can an implementation based on this strategy outperform other highly efficient computer algebra packages? Does the performance of this hybrid C-MAPLE application comply to its complexity analysis? In Chapter 8, we will provide the answers to these questions.

Once our fundamental high-performance software library for polynomial arithmetic and its interface with AXIOM and MAPLE have been in place, we could start investigating the third objective of this PhD work: developing more efficient algorithms for the core operations involved in computing triangular decompositions, with an emphasis in dimension zero. (see Section 2.3 for the definition of triangular decompositions). We started with multiplication modulo a triangular set in Chapter 6, followed by regular GCD computations and regularity test with respect to a regular chain, see Chapter 7.

Triangular decompositions rely intensively on polynomial arithmetic operations (addition, subtraction, multiplication and division) modulo ideals given by triangular sets in dimension zero or regular chains in positive dimension. (see Chapter 2 for these terms.) Modular multiplication and division are expensive (often dominant) operations in terms of computational time when computing triangular decompositions. Under certain assumptions, the modular division can be achieved by two modular multiplications as reported in Section 2.1 in the *fast division* algorithm. Thus, *modular multiplication* is unarguably a “core” operation.

Triangular decompositions rely also on an univariate and recursive representation of polynomials. The motivation is to reduce solving systems of multivariate polynomials to univariate polynomials GCD computations. This reduction is achieved at the price of working over non-standard algebraic structures, more precisely modulo the so-called regular chains. We have designed and implemented the first algorithm for this kind of GCD computations which is based on asymptotically fast polynomial arithmetic and modular techniques, while not making any restrictive assumptions on the input. Chapter 7 presents this algorithm.

We designed these high-level operations in a way that our previous fast polynomial arithmetic implementation could efficiently be used. Certainly, these new algorithms are also better than existing ones in terms of complexity. All our reported new implementations and algorithms from this thesis have been finalized as a solid software library `modpn` with its MAPLE-level wrapper `FastArithmeticTools` (see Section 8.2): a C-MAPLE library dedicated to fast arithmetic for multivariate polynomials over prime field including fundamental operations modulo regular chains in dimension zero.

1.3 Contributions

As mentioned in Section 1.1 one of our motivations of this research is to design efficient algorithms, based on asymptotically fast algorithms and modular methods for the key routines. At the end of this research we have designed a set of asymptotically fast algorithms for core operations supporting polynomial system solvers based on triangular decompositions. These are fast multiplication modulo a triangular set (modular multiplication, see Chapter 6), fast regular GCD computation and regularity test, see Chapter 7. As a byproduct, we have obtained highly efficient algorithms for solving bivariate polynomial systems and multivariate systems of two equations, see Chapters 7 and 8. Our implementation effort for polynomial arithmetic over finite fields has led to an improved version of the so-called *Montgomery's trick* [74]. More precisely, we have obtained a fast integer reduction trick when the modulus is a Fourier prime number (see Section 6.3).

We have systematically investigated and documented a set of suitable implementation techniques adapted for asymptotically fast polynomial algorithms and operations supporting triangular decompositions (see Section 3.3, Section 3.4; Section 4.2-4.5; Section 5.2, Section 6.3, Section 8.2, and Chapter 9).

As mentioned in Section 1.1 another motivation for this research is to develop a foundational software library for polynomial arithmetic in order to support the implementation of high-performance solvers. At the end of this work, besides the theoretical contributions we have also provided a solid software result: the `modpn` library. The library `modpn` consists of a set of highly optimized C implementations including the base-level routines and operations modulo regular chains. Essentially, all the research results reported in this thesis have been implemented in the `modpn` library. The `modpn` library has been integrated into the computer algebra system MAPLE (version 13). Concretely, this provides MAPLE users with fast arithmetic for multivariate polynomials over the prime fields $\mathbb{F}_p$, where p is a machine word-size prime number. While being easy to use, it mainly focuses on high performance.

We present the experimental results in Chapter 7 and in Chapter 8 to compare our library with pre-existing MAPLE and MAGMA implementations. The experimental result show that our approaches are very effective, since they often significantly outperform pre-existing implementations. The experimentation effort meets our last motive mentioned in Section 1.1. Namely, we have systematically evaluated the performance, including speed-up factors and bottlenecks, of this approach and compared it with the pre-existing polynomial system solvers. For operations such as Regular GCD, Regularity Test, our new algorithm implementation has a factor of hundreds faster than pre-existing ones.

1.4 Outline

In Chapter 2, we provide an overview of the background knowledge related to this research, including implementation environment and existing asymptotically fast polynomial algorithms. Chapter 3 is the starting point of this research. In this chapter we investigate the existing fundamental fast polynomial algorithms; we demonstrate that by using suitable implementation techniques, these fast algorithms can outperform the classical ones in a significant manner; moreover, the new implementation can directly support existing popular computer algebra systems such as AXIOM (see Section 2.2), thus can speed up related higher-level packages. In Chapter 4 and Chapter 5 we focus on our new implementation strategies for asymptotically fast polynomial algorithms. More specifically, we investigate the implementation techniques suited to the multiple-level language environment in AXIOM. In Chapters 6, 7, and 8, we present the new fast algorithms we have developed and their implementation result which is integrated in MAPLE version 13. The new algorithms include *modular multiplication*, *regular GCD*, *bivariate solver*, *two-equation solver* and *regularity test.* In Chapter 9, we present our parallel implementation of efficiency-critical operations for fast polynomial arithmetic.

Chapter 2

Background

In this chapter we introduce asymptotically fast polynomial arithmetic, our implementation environment and the concept of a triangular decomposition.

2.1 Pre-existing Fast Algorithms

In this section, we describe, or give references to, a set of basic fast algorithms we have implemented. These algorithms are low-level operations in the sense that they will be used in almost all upper level algorithms reported in this thesis. We will describe our new asymptotically fast algorithms in Chapters 6, 7, and 8. In the following text, all rings are commutative with unity; we denote by M a multiplication time in Definition 1.

Definition 1. *A* multiplication time *is a map* $\mathsf{M} : \mathbb{N} \to \mathbb{R}$*, where* $\mathbb{R}$ *is the field of real numbers, such that:*

- *For any ring* R*, polynomials of degree less than* d *in* $R[X]$ *can be multiplied in at most* $\mathsf{M}(d)$ *operations* $(+, \times)$ *in* R*.*
- *For any* $d \leq d'$*, the inequality* $\frac{\mathsf{M}(d)}{d} \leq \frac{\mathsf{M}(d')}{d'}$ *holds.*

Examples of multiplication times are:

- Classical: $2d^2$;
- Karatsuba: $C\, d^{\log_2(3)}$ with some $C \geq 9$;
- FFT over an arbitrary ring: $C\, d \log(d) \log(\log(d))$ for some $C \geq 64$ [21].

Note that the FFT-based multiplication in degree d over a ring that supports the FFT (that is, possessing primitive n-th root of unity, where n is a power of 2 greater than $2d$) can run in $C\,d\log(d)$ operations in R, with some $C \geq 18$.

The Montgomery integer division trick. Montgomery integer division trick [74] is a fast way to compute integer division. Since our algorithms are mostly over $\mathbb{Z}/p\mathbb{Z}$, operations modulo prime number p are essential. We have designed various versions of this trick in order to improve performances as reported in Sections 5.2.2 and 6.3. Here we give the original Montgomery trick. The principle of this trick is that instead of computing an Euclidean division, it reduces the input integer w.r.t to a number which is power of 2. In machine arithmetic, an integer can be divided by a power of 2 can simply by bitwise operations which are very cheap.

Algorithm 1 The Montgomery Integer Division trick

```
INPUT: $Z,\ R,\ v,\ V \in \mathbb{Z}$, where $v$ is the modulus and $V \cdot v \equiv -1 \bmod R$, assume
       $Z < R \cdot v$, $R < v$, $R$ usually chosen to be some power of 2, and $GCD(R,\ v) = 1$.

OUTPUT: $T = Z \cdot R^{-1}$ rem $v$.

1   $A = V \cdot Z$
2   $B = A$ rem $R$
3   $C = B \cdot v$
4   $T = Z + C$ quo $R$
5   if $v < T$ then $T = T - v$
6   return $T$
```

Fast Fourier transform and truncated Fourier transform. The fast Fourier transform (FFT) is a fast algorithm for calculating the discrete Fourier transform (DFT) of a function, see [44] for details.

This algorithm was essentially known to Gauss and was rediscovered by Cooley and Turkey in 1965. In symbolic computations, the FFT algorithm has many applications [36]. The most famous one is the fast multiplication of polynomials. Even if the principles of these calculations are quite simple, their practical implementation is still an active area of investigation.

The principle of FFT-based univariate polynomial multiplication is the following. We consider two polynomials $f = \sum_{k=0}^{k=n-1} a_k x^k$ and $g = \sum_{k=0}^{k=n-1} b_k x^k$ over some field $\mathbb{K}$. We do not need to assume that they have the same degree; if they do not have the same degree, we add a "zero leading coefficient" to the one of smaller degree.

We want to compute the product $fg = \sum_{k=0}^{k=2n-2} c_k$. The classical algorithm would compute the coefficient c_k of fg by

$$c_k = \sum_{i=0}^{i=k} a_i b_{k-i} \tag{2.1}$$

for $k = 0, \ldots, 2n-2$, amounting to $O(n^2)$ operations in $\mathbb{K}$.

If the values of f and g are known at $2n-1$ different points of $\mathbb{K}$, say $x_0, \ldots, x_{2n-2}$, then we can obtain the product fg by computing $f(x_0)g(x_0), \ldots, f(x_{2n-2})g(x_{2n-2})$ amounting to $O(n)$ operations in $\mathbb{K}$. The second idea is to use points $x_0, \ldots, x_{2n-1}$ in $\mathbb{K}$ such that

(*i*) evaluating f and g at these points can be done in nearly linear time cost, such as $O(n\log(n))$,

(*ii*) interpolating the values $f(x_0)g(x_0), \ldots, f(x_{2n-2})g(x_{2n-2})$ can be done in nearly linear time, that is $O(n\log(n))$ again.

Such points $x_0, \ldots, x_{2n-2}$ do not always exist in $\mathbb{K}$. However, there are techniques to overcome this limitation (essentially by considering a field extension of $\mathbb{K}$ where the desired points can be found). In the end, this leads to an algorithm for FFT-based univariate polynomial multiplication which runs in $O(n\log(n)\log(\log(n)))$ operations in $\mathbb{K}$ [21]. This is the best known algorithm for arbitrary $\mathbb{K}$.

In this thesis, we restrict ourselves to the case where we can find points $x_0, \ldots, x_{2n-2}$ in $\mathbb{K}$ satisfying the above (*i*) and (*ii*). Most finite fields possess such points for n small enough. (Obviously n must be at most equal to the cardinality of the field.) More precisely, for $n,\ p > 1$, where p is a prime, the finite field $\mathbb{Z}/p\mathbb{Z}$ has a primitive m-th root of unity if and only if m divides $p-1$. (Recall that $\omega \in \mathbb{K}$ is a primitive m-th root of unity of the field $\mathbb{K}$ if and only if $\omega^m = 1$ and $\omega^k \neq 1$ for $0 < k < m$). If $\mathbb{Z}/p\mathbb{Z}$ has a primitive m-th root of unity ω, $m > 2n-2$

- then we use $x_k = \omega^k$ for $k = 0, \ldots, 2n-2$,
- Step (*i*) is the FFT of f and g at ω (to be detailed in the next section),
- Step (*ii*) is the FFT of $\frac{fg}{n}$ at ω^{-1}.

Again, we refer to [44] for more details.

In [51], J. van der Hoeven reported a truncated version of the classical fast Fourier transform. It is referred as the *truncated Fourier transform* (TFT) in the literature. When applied to polynomial multiplication, this algorithm has the nice property of eliminating the jumps in the complexity at powers of two. Essentially, this algorithm avoids computing the leading zeros during the DFT/evaluation and inverse-DFT/interpolation stages. We have implemented this algorithm which indeed removed the stair-case like timing curves from FFT based methods. However, this algorithm requires more complicated programming structures which may curb compilers to apply certain loop optimization techniques, whereas the standard iterative FFT implementation has a much simpler nested loop structure which is easy for compiler to optimize the code.

Power series inversion. Power series inversion using Newton iteration method provides a fast method of computing multiplicative inverses. Given a commutative ring R with a 1 and $\ell \in \mathbb{N}$, it computes the inverse of the polynomial $f \in R[x]$, such that, $f(0) = 1$ and $\deg f < \ell$, modulo x^{ℓ}. The Newton iteration is used in numerical analysis to compute successive approximations to solutions of $\phi(g) = 0$. From a suitable initial approximation g_0, subsequent approximations are computed using:

$$g_{i+1} = g_i - \frac{\phi(g_i)}{\phi'(g_i)} \tag{2.2}$$

where $\phi^{'}$ is the derivative of ϕ. This corresponds to intersecting the tangent with an axis or, in other words, replacing ϕ by its linearization at that point. If we apply this to the problem of finding a $g \in R[x]$, given $\ell \in \mathbb{N}$ with $f(0) = 1$, satisfying $f\,g \equiv 1 \bmod x^{\ell}$, we want to approximate a root of $1/g - f = 0$. The Newton iteration step becomes:

$$g_{i+1} = g_i - \frac{1/g_i - f}{-1/g_i^2} = 2\,g_i - f\,g_i^2 \tag{2.3}$$

.

Proposition 1 shows that this method converges quickly to a solution, also in this algebraic setting.

Proposition 1. *Let R be a ring (commutative with 1), $f, g_0, g_1, \ldots \in R[x]$, with $f(0) = 1$, $g_0 = 1$, and $g_{i+1} \equiv 2\,g_i - f\,g_i^2 \bmod x^{2^{i+1}}$, for all i. Then $f\,g_i \equiv 1 \bmod x^{2^i}$ for all $i \geq 0$.*

Proof. The proof is by induction on i. For $i = 0$ we have

$$f\,g_0 \equiv f(0)g_0 \equiv 1{\cdot}1 \equiv 1 \bmod x^{2^0} \tag{2.4}$$

For the induction step, we find

$$1 - f\,g_{i+1} \equiv 1 - f\,(2\,g_i - f\,g_i^2) \equiv 1 - 2\,f\,g_i + f^2\,g^2 \equiv (1 - f\,g_i)^2 \equiv 0 \bmod x^{2^{i+1}} \quad (2.5)$$

□

Based on the above, we obtain the following Algorithm 2 for computing the inverse of f mod x^ℓ.

Algorithm 2 Power Series Inversion of f to Precision ℓ

INPUT: $f \in R[x]$ such that $f(0) = 1$, $\ell \in \mathbb{N}$ such that $\deg(f) < \ell$ and $R[x]$ in variable x is a ring of power series.

OUTPUT: $g \in R[x]$ such that $f\,g \equiv 1 \bmod x^\ell$. Runs in $3\mathsf{M}(\ell) + 0(\ell)$ operations in R. Recall from Definition 1 that M is multiplication time whose value is dependent on the multiplication algorithm used.

```
Inv(f, ℓ) ==
  1   $g_0 := 1$
  2   $r := \lceil \log_2(\ell) \rceil$
  3   for $i = 1..r$ repeat $g_i := (2g_{i-1} - f{g_{i-1}}^2) \bmod x^{2^i}$
  4   return $g_r$
```

Proposition 2. *If ℓ is a power of 2, then Algorithm 2 uses at most $3\mathsf{M}(\ell) + O(\ell) \in O(\mathsf{M}(l))$ arithmetic operations in R [44, Ch. 9].*

PROOF. The correctness stems from Proposition 1 which concludes that

$$f\,g_i \equiv 1 \bmod x^{2^i} \quad (2.6)$$

for all $i \geq 0$. In line 3, all powers of x greater than 2^i can be dropped, and since,

$$g_i \equiv g_{i-1}\,(2 - f g_{i-1}) \equiv g_{i-1} \bmod x^{2^{i-1}} \quad (2.7)$$

the powers of x less than 2^{i-1} can also be dropped.

The cost for one iteration of line 3 is $\mathsf{M}(2^{i-1})$ for the computation of ${g_{i-1}}^2$, $\mathsf{M}(2^i)$ for the product $f\,{g_{i-1}}^2 \bmod x^{2^i}$, and then the negative of the upper half of $f{g_{i-1}}^2$ modulo x^{2^i} is the upper half g_i, taking 2^{i-1} operations. Thus we have $\mathsf{M}(2^i) + \mathsf{M}(2^{i-1}) + 2^{i-1} \leq \frac{3}{2}\,\mathsf{M}(2^i) + 2^{i-1}$, resulting in a total running time:

$$\sum_{1\le i\le r} \frac{3}{2}\,\mathsf{M}(2^i) + 2^{i-1} \le (\frac{3}{2}\,\mathsf{M}(2^r) + 2^{r-1}) \sum_{1\le i\le r} 2^{i-r} < 3\,\mathsf{M}(2^r) + 2^r = 3\,\mathsf{M}(\ell) + \ell \quad (2.8)$$

since $2\mathsf{M}(n) \le \mathsf{M}(2n)$ for all $n \in \mathbb{N}$ (see Definition 1 at Page 8) □

Fast division. Using fast multiplication enables us to write a fast Euclidean division for polynomials, using Cook-Sieveking-Kung's approach through power series inversion [43, Chapter 9]. Given two polynomials a and b, both $\in R[x]$ and b monic, where R is a ring (commutative with 1); assuming that a and b have respective degrees m and n, with $m \ge n$, we can compute the polynomials q and r in $R[x]$ satisfying $a = q\,b + r$ and $\deg(r) < \deg(b)$. Using standard techniques this takes $O(n^2)$ operations in R. Equipped with a fast power series inversion, it can be improved to $O(\mathsf{M}(n))$ operations in R [44].
We define A and B as the *reversals* of a and b:

$$A(x) = x^m\, a(1/x) \quad (2.9)$$

$$B(x) = x^n\, b(1/x) \quad (2.10)$$

With the inverse $C \equiv 1/B(x) \bmod x^{m-n+1}$, we obtain q as the reversal of Q from the subsequent multiplication:

$$Q(x) \equiv A(x)\,C(x) \bmod x^{m-n+1} \quad (2.11)$$

The full algorithm is shown in Algorithm 3.

Algorithm 3 Fast Division with Remainder Algorithm

Input: $a, b \in R[x]$, where R is a ring (commutative, with 1) and $b \neq 0$ is monic

Output: $q, r \in R[x]$ such that $a = qb + r$ and $deg\, r < \deg\, b$

```
$FDiv(a, b)$ ==
1:  $s := \mathrm{Rev}(b)^{-1} \mod x^{\deg(a)-\deg(b)+1}$
2:  $Q := \mathrm{Rev}(a)s \mod x^{\deg(A)-\deg(T_1)+1}$
3:  $q := \mathrm{Rev}(Q)$
4   $r := a - b\,q$
5:  **return** $(q, r)$
```

Kronecker's substitution. Let $\mathbb{A}$ be a commutative ring with units. Let $x_1 < x_2 < \cdots < x_n$ be n ordered variables and let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be n positive integers with $\alpha_1 = 1$. We consider the ideal $\mathcal{I}$ of $\mathbb{A}[x_1, x_2, \ldots, x_n]$ generated by $x_2 - x_1^{\alpha_2}, x_3 - x_1^{\alpha_3}, \ldots, x_n - x_1^{\alpha_n}$. Define $\alpha = (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n)$. Let Ψ_α be the canonical map from $\mathbb{A}[x_1, x_2, \ldots, x_n]$ to $\mathbb{A}[x_1, x_2, \ldots, x_n]/\mathcal{I}$, which substitutes the variables x_2, x_3, $\ldots$, x_n with $x_1^{\alpha_2}$, $x_1^{\alpha_3}$, $\ldots$, $x_1^{\alpha_n}$ respectively. We call it the Kronecker map of α. This map transforms a multivariate polynomial of $\mathbb{A}[x_1, x_2, \ldots, x_n]$ into an univariate polynomial of $\mathbb{A}[x_1]$. It has the following immediate property.

Proposition 3. *The map Ψ_α is a ring-homomorphism. In particular, for all $a, b \in \mathbb{A}[x_1, x_2, \ldots, x_n]$ we have*

$$\Psi_\alpha(ab) = \Psi_\alpha(a)\Psi_\alpha(b). \tag{2.12}$$

Therefore, if the product $\Psi_\alpha(a)\Psi_\alpha(b)$ has only one pre-image by Ψ_α, one can compute the product of the multivariate polynomials a and b via the product of the univariate polynomials $\Psi_\alpha(a)$ and $\Psi_\alpha(b)$. This is advantageous, when one has at hand a fast univariate multiplication. In order to study the pre-images of $\Psi_\alpha(a)\Psi_\alpha(b)$ we introduce additional material.

Let $d_1, d_2, \ldots, d_n$ be non-negative integers. We write $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ and we denote by Δ_n the set of the n-tuple $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ of non-negative integers such that we have $e_i \leq d_i$ for all $i = 1, \ldots, n$. We define

$$\delta_n = d_1 + \alpha_2 d_2 + \cdots + \alpha_n d_n \quad \text{and} \quad \delta_0 = 1, \tag{2.13}$$

and we consider the map $\psi_{\mathbf{d}}$ defined by

$$\psi_{\mathbf{d}} : \begin{array}{rcl} \Delta_n & \longrightarrow & [0, \delta_n] \\ (e_1, e_2, \ldots, e_n) & \longmapsto & e_1 + \alpha_2 e_2 + \cdots + \alpha_n e_n \end{array} \tag{2.14}$$

that we call the *packing exponent map.*

Proposition 4. *The* packing exponent map $\psi_{\mathbf{d}}$ *is one-to-one map if and only if the following relations holds:*

$$\begin{array}{rcl} \alpha_2 & = & 1 + d_1 \\ \alpha_3 & = & 1 + d_1 + \alpha_2 d_2 \\ \vdots & & \vdots \\ \alpha_n & = & 1 + d_1 + \sum_{i=2}^{i=n-1} \alpha_i d_i. \end{array}$$

that we call the packing relations.

Proof. We proceed by induction on $n \geq 1$. For $n = 1$ we have $\delta_1 = d_1$ and $\psi_{\mathbf{d}}(e_1) = e_1$ for all $0 \leq e_1 \leq d_1$. Thus the packing exponent map $\psi_{\mathbf{d}}$ is clearly one-to-one map in this case. Since the packing relations trivially hold for $n = 1$, the property is proved in this case. We consider now $n > 1$ and we assume that the property holds for $n - 1$. We look for necessary and sufficient conditions for $\psi_{\mathbf{d}}$ to be a one-to-one map. We observe that the partial function

$$\psi_{(d_1,\ldots,d_{n-1})} : \begin{array}{ccc} \Delta_{n-1} & \longrightarrow & [0, \delta_{n-1}] \\ (e_1, e_2, \ldots, e_{n-1}) & \longmapsto & \psi_{\mathbf{d}}(e_1, e_2, \ldots, e_{n-1}, 0) \end{array} \tag{2.15}$$

of ψ_d needs to be a one-to-one map for ψ_d to be a one-to-one map. Therefore, by induction hypothesis, we can assume that the following relations hold.

$$\begin{array}{rcl} \alpha_2 & = & 1 + d_1 \\ \alpha_3 & = & 1 + d_1 + \alpha_2 d_2 \\ \vdots & & \vdots \\ \alpha_{n-1} & = & 1 + d_1 + \sum_{i=2}^{i=n-2} \alpha_i d_i. \end{array}$$

Observe that the last relation writes

$$\alpha_{n-1} = 1 + \delta_{n-2}. \tag{2.16}$$

We consider now $f \in [0, \delta_n]$. Let q and r be the quotient and the remainder r of the Euclidean division of f by α_n. Hence, we have

$$f = q\alpha_n + r \quad \text{and} \quad 0 \leq r < \alpha_n. \tag{2.17}$$

Moreover, the couple (q, r) is unique with these properties. Assume that $\alpha_n = 1+\delta_{n-1}$ holds then f has a unique pre-image in Δ_n by ψ_d^{-1} which is

$$\psi_d^{-1}(f) = (\psi^{-1}_{(d_1,\ldots,d_{n-1})}(r), q). \tag{2.18}$$

If $\alpha_n > 1 + \delta_{n-1}$ holds, then $f = 1 + \delta_{n-1}$ has no pre-images in Δ_n by ψ_d^{-1}. If $\alpha_n < 1 + \delta_{n-1}$ holds, then $f = \alpha_n$ has two pre-images in Δ_n, namely

$$(0, \ldots, 0, 1) \quad \text{and} \quad \psi^{-1}_{(d_1,\ldots,d_{n-1})}(\alpha_n). \tag{2.19}$$

Finally, the map $\psi_{\mathbf{d}}$ is one-to-one if and only if the packing relations hold. □

Proposition 5. *Let* $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ *be in* Δ_n *and* $\mathbf{X} = x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n}$ *be a monomial of* $\mathbb{A}[x_1, x_2, \ldots, x_n]$. *We have*

$$\Psi_\alpha(\mathbf{X}) = x_1^{\psi_{\mathbf{d}}(\mathbf{e})}. \tag{2.20}$$

Moreover, for all $f = \sum_{\mathbf{X} \in S} c_{\mathbf{X}} \mathbf{X}$

$$\Psi_\alpha(f) = \sum_{\mathbf{X} \in S} c_{\mathbf{X}} \Psi_\alpha(\mathbf{X}). \tag{2.21}$$

where S *is the support of* f*, that is the set of the monomials occurring in* p.

Proof. Relation (2.20) follows easily from the definition of Ψ_α. Relation (2.21) follows from Proposition 3. □

We denote by $\mathbb{A}[\Delta_n]$ the set of the polynomials $p \in \mathbb{A}[x_1, x_2, \ldots, x_n]$ such that for every $\mathbf{X} = x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n}$ in the support of p we have $(e_1, e_2, \ldots, e_n) \in \Delta_n$. The set $\mathbb{A}[\Delta_n]$ is not closed under multiplication, obviously. Hence it is only a $\mathbb{A}$-module. The same remark holds for the set $\mathbb{A}[\delta_n]$ of univariate polynomials over $\mathbb{A}$ with degree equal or less than δ_n.

Kronecker's substitution based multivariate multiplication. Following the previous notations and definitions from *Kronecker's substitution*, we investigate Kronecker's substitution based multivariate multiplication as follows. Although the restriction of the map Ψ_α to $\mathbb{A}[\Delta_n]$ is not a ring isomorphism, it can be used for multiplying multivariate polynomial as follows. Let $f, g \in \mathbb{A}[x_1, x_2, \ldots, x_n]$ and let p be their product. For all $1 \leq i \leq n$ we choose

$$d_i = \deg(p, x_i), \tag{2.22}$$

that is the partial degree of p w.r.t. x_i. Observe that for all $1 \leq i \leq n$ we have

$$\deg(p, x_i) = \deg(f, x_i) + \deg(g, x_i). \tag{2.23}$$

It follows that the three polynomials f, g, p belong to $\mathbb{A}[\Delta_n]$. Moreover, from Proposition 3, we have

$$\Psi_\alpha(p) = \Psi_\alpha(f)\Psi_\alpha(g). \tag{2.24}$$

Therefore, we can compute p using the simple following algorithm . Let us assume that we have at hand a quasi-linear algorithm for multiplying in $\mathbb{A}[x_1]$, that is an

Algorithm 4 Kronecker Multiplication

Input: $f, g \in \mathbb{A}[\Delta_n]$ such that $fg \in \mathbb{A}[\Delta_n]$ holds.

Output: fg

```
1   $u_f := \Psi_\alpha(f)$
2   $u_g := \Psi_\alpha(g)$
3   $u_{fg} := u_f u_g$
4   $p := \Psi_\alpha^{-1}(u_{fg})$
5   **return** $p$
```

algorithm such that the product of two polynomials of degree less that k can be computed in $O(k^{1+\epsilon})$ operations in $\mathbb{A}$. Such algorithm exists over any ring $\mathbb{A}$ [21]. It follows that step **3** of the above algorithm can be performed in $O({\delta_n}^{1+\epsilon})$ for every $\epsilon > 0$. Therefore, we have:

Proposition 6. *For every $\epsilon > 0$, Algorithm 4 runs in $O(((d_1+1)\cdots(d_n+1))^{1+\epsilon})$ operations in $\mathbb{A}$.*

2.2 Implementation Environment

In this section, we introduce the computer algebra systems and their programming languages on which we rely to implement our algorithm and test the performance. We use two systems: AXIOM and Maple.

AXIOM [52] is a comprehensive Computer Algebra System which has been in development since 1971. It was originally developed by IBM under the direction of Richard Jenks. AXIOM has a very high level programming language called SPAD, the abbreviation of Scratchpad. It can be compiled into Common Lisp by its own built-in compiler. There is an external stand-alone compiler implemented in C which also accepts the SPAD language, called Aldor [1]. AXIOM has both an interactive mode for user interactions and a programming language for building library modules. The typical way of programming in AXIOM is as follows. The programmer creates an input file defining some functions for his or her application. Then, the programmer runs the file and tries the functions. Once everything works well, the programmer may want to add the functions to the local AXIOM library. To do so, the programmer needs to integrate his or her code in AXIOM type constructors and then invoke the compiler.

By definition, an AXIOM type constructor is a function that returns a type which can be either a *category*, a *domain*, or a *package*. Roughly speaking, a domain is a class of objects. For example, `Polynomial` domain denotes polynomials, `Matrix` domain denotes matrices. A category is a class of domains which has common properties. For example, the AXIOM category `Ring` designates the class of all rings with units, any AXIOM domain that has this property belongs to the category `Ring`. The source code for the category `Ring` is shown below.

```
Ring(): Category == Join(Rng,Monoid,LeftModule(%)) with
      --operations
      characteristic: () -> NonNegativeInteger
        ++ characteristic() returns the characteristic of the ring
        ++ this is the smallest positive integer n such that
        ++ \spad{n*x=0} for all x in the ring, or zero if no such n
        ++ exists.
        -- We can not make this a constant, since some domains are
        -- mutable
      coerce: Integer -> %
        ++ coerce(i) converts the integer i to a member of
        ++ the given domain.
      unitsKnown
        ++ recip truly yields
        ++ reciprocal or "failed" if not a unit.
        ++ Note: \spad{recip(0) = "failed"}.
   add
      n:Integer
      coerce(n) == n * 1$%
```

From the above AXIOM source code we can observe another important concept: categories form a hierarchy. We can see that `Ring` is extended from the categories `Rng`, `Monoid` and `LeftModule`. In addition, we can observe that `Ring` has

- 2 operations: `characteristic`, `coerce`,
- 1 attribute `unitsKnown`,
- and 1 default implementation for the operation `coerce: Integer -> %`.

The programmer can construct her/his own categories by extending existing categories. This requires knowledge of the existing hierarchies. Figure 2.1 shows a fragment of the hierarchy of the AXIOM algebraic categories.

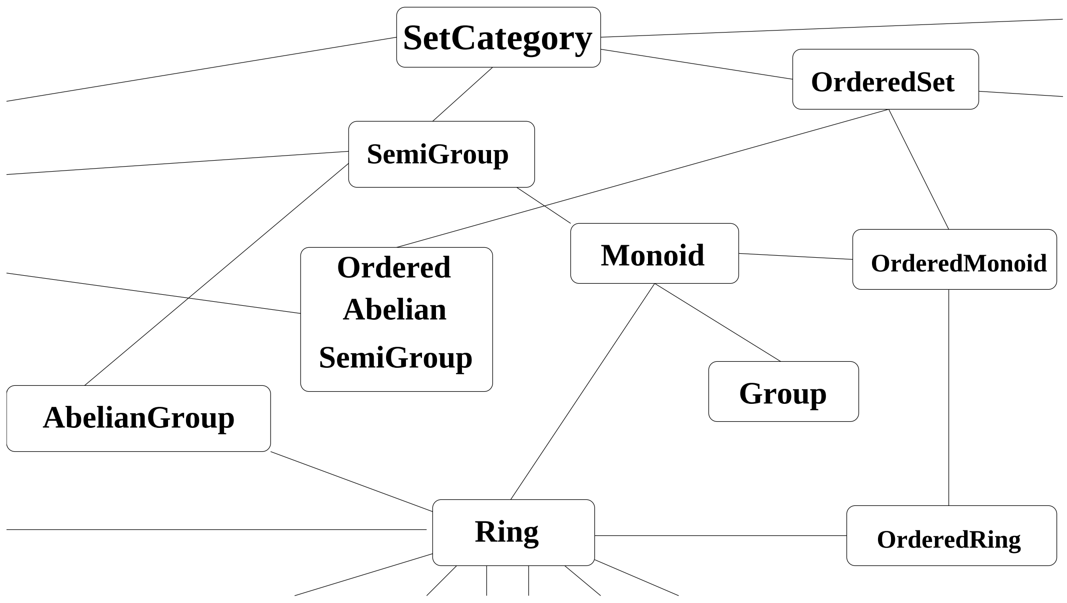


Figure 2.1: Algebraic categories' hierarchy in AXIOM (partial).

Next to the concept of category, domain is easier to understand. It actually corresponds to the notion of data type. When a domain is defined, it is asserted to belong to one or more categories and promises to implement the set of operations defined in these categories. After an newly defined domain is compiled, it becomes an AXIOM data type which can be used just like a system-provided data type. The programmer usually needs to design a lower level data structure to represent the objects of the domain. When a domain is instantiated, the AXIOM system will allocate memory for those data structures.

MAPLE is one of the most popular computer algebra systems. It was first developed by the Symbolic Computation Group at the University of Waterloo in 1980. Maple incorporates a dynamically typed imperative-style interpreted programming language. The language permits variables of lexical scope. There are also interfaces to other languages (C, Fortran, Java, Matlab, and Visual Basic). Maple is based around a small kernel, written in C, which provides the Maple language. Most functionality is provided by libraries. Most of the libraries are written in the Maple language. Symbolic expressions including polynomials are stored in memory as directed acyclic graphs. MAPLE has a set of powerful symbolic polynomial computation libraries. The related existing polynomial packages are `RegularChains`, `PolynomialIdeals`. MAPLE language is interpreted and easy to use. As reported in later chapters (Chapters 7, 8), the previous triangular decomposition technique based implementation in MAPLE relies on the MAPLE interpreted high level language and classical polynomial arithmetic. Our new MAPLE library `modpn` is developed based asymptotically fast polynomial arithmetic and the majority part written in C. Therefore, the new algorithms and implementation from this thesis practically have sped up the triangular decomposition packages in MAPLE. In Chapter 8 we will report the C/MAPLE code integration procedure in details.

2.3 Triangular Decompositions

2.3.1 Polynomial ideal and radical

Let $\mathbb{K}$ be a field and let $\mathbb{K}[\mathbf{x}] = \mathbb{K}[x_1, \ldots, x_n]$ be the ring of polynomials with coefficients in $\mathbb{K}$, with ordered variables $x_1 \prec \cdots \prec x_n$. Let $\overline{\mathbb{K}}$ be the algebraic closure of $\mathbb{K}$. If $\mathbf{u}$ is a subset of $\mathbf{x}$ then $\mathbb{K}(\mathbf{u})$ denotes the fraction field of $\mathbb{K}[\mathbf{u}]$.

Definition 1. *Let $F = \{f_1, \ldots, f_m\}$ be a finite subset of $\mathbb{K}[x_1, \ldots, x_n]$. The* ideal generated *by F in $\mathbb{K}[x_1, \ldots, x_n]$, denoted by $\langle F \rangle$ or $\langle f_1, \ldots, f_m \rangle$, is the set of all polynomials of the form*

$$h_1 f_1 + \cdots + h_m f_m$$

where $h_1, \ldots, h_m$ are in $\mathbb{K}[x_1, \ldots, x_n]$. If the ideal $\langle F \rangle$ is not equal to the entire polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$, then $\langle F \rangle$ is said to be a proper ideal.

Definition 2. *The* radical *of the ideal generated by F, denoted by $\sqrt{\langle F \rangle}$, is the set of polynomials $p \in \mathbb{K}[x_1, \ldots, x_n]$ such that there exists a positive integer e satisfying $p^e \in \langle F \rangle$. The ideal $\langle F \rangle$ is said to be radical if we have $\langle F \rangle = \sqrt{\langle F \rangle}$.*

Remark 1. *Let $f_1, \ldots, f_m \in \mathbb{K}[x_1]$ be univariate polynomials. The Euclidean Algorithm for computing greatest common divisors implies that the ideal $\langle f_1, \ldots, f_m \rangle$ is equal to $\langle g \rangle$, where $g = \gcd(f_1, \ldots, f_m)$. This means that there exists polynomials*

$a_1, \ldots, a_m, b_1, \ldots, b_m \in \mathbb{K}[x_1]$ such that we have

$$a_1 f_1 + \cdots + a_m f_m = g \quad \text{and} \quad f_i = b_i g \quad \text{for} \quad i = 1, \ldots, e.$$

Therefore, every ideal of $\mathbb{K}[x_1]$ is generated by a single element.

Definition 3. *A univariate polynomial $f \in \mathbb{K}[x_1]$ is said to be* squarefree *if for all non-constant polynomials $g \in \mathbb{K}[x_1]$ the polynomial g^2 does not divide f.*

Remark 2. *Let $f \in \mathbb{K}[x_1]$ be non-constant. It is not hard to see that the ideal $\langle f \rangle \subseteq \mathbb{K}[x_1]$ is radical if and only if f is squarefree.*

2.3.2 Zero-divisor, regular element, zero set

For a subset F of $\mathbb{K}[\mathbf{x}]$, let h be a polynomial in $\mathbb{K}[\mathbf{x}]$, the *saturated ideal* of $\langle F \rangle$ with respect to h, denoted by $\langle F \rangle : h^\infty$, is the ideal

$$\{q \in \mathbb{K}[\mathbf{x}] \mid \exists m \in \mathbb{N} \text{ such that } h^m q \in \langle F \rangle\}.$$

A polynomial $p \in \mathbb{K}[\mathbf{x}]$ is a *zero-divisor* modulo $\langle F \rangle$ if there exists a polynomial q such that $pq \in \langle F \rangle$, and neither p nor q belongs to $\langle F \rangle$. The polynomial p is *regular* modulo $\langle F \rangle$ if it is neither zero, nor a zero-divisor modulo $\langle F \rangle$. Geometrically, we denote by $V(F)$ the *zero set* (or solution set, or variety) of F in $\overline{\mathbb{K}}^n$. For a subset $W \subset \overline{\mathbb{K}}^n$, we denote by $\overline{W}$ its closure in the Zariski topology.

2.3.3 Triangular set and regular chains

Main variable and initial. If $p \in \mathbb{K}[\mathbf{x}]$ is a non-constant polynomial, the largest variable appearing in p is called the *main variable* of p and is denoted by $\mathrm{mvar}(p)$. The leading coefficient of p w.r.t. $\mathrm{mvar}(p)$ (p is viewed as an univariate polynomial in $\mathrm{mvar}(p)$) is its *initial*, written $\mathrm{init}(p)$ whereas $\mathrm{lc}(p, v)$ is the leading coefficient of p w.r.t. $v \in \mathbf{x}$. For example, let p be the polynomial $2y^3x^2 + 3yx + 1 \in \mathbb{K}[x, y]$, $x > y$, $\mathrm{init}(p) = 2y^3$ but $\mathrm{lc}(p, y) = 2x^2$.

Triangular Set. A subset T of non-constant polynomials of $\mathbb{K}[\mathbf{x}]$ is a *triangular set* if the polynomials in T have pairwise distinct main variables. Denote by $\mathrm{mvar}(T)$ the set of the main variables of the polynomials in T. A variable $v \in \mathbf{x}$ is *algebraic* with respect to T if $v \in \mathrm{mvar}(T)$; otherwise it is *free*. For a variable $v \in \mathbf{x}$ we denote by

$T_{<v}$ (resp. $T_{>v}$) the subsets of T consisting of the polynomials with main variable less than (resp. greater than) v. If $v \in \mathrm{mvar}(T)$, we denote by T_v the polynomial in T with main variable v. If T is not empty, we denote by $T_{\max}$ the polynomial of T with largest main variable.

Quasi-component and saturated ideal. Given a triangular set T in $\mathbb{K}[\mathbf{x}]$, denote by h_T the product of the $\mathrm{init}(p)$ for all $p \in T$. The *quasi-component* $W(T)$ of T is $V(T) \setminus V(h_T)$, that is, the set of the points of $V(T)$ which do not cancel any of the initials of T. We denote by $\mathrm{sat}(T)$ the *saturated ideal* of T, defined as follows: if T is empty then $\mathrm{sat}(T)$ is the trivial ideal $\langle 0 \rangle$; otherwise it is the ideal $\langle T \rangle : h_T^{\infty}$.

For the given regular chain $T = \{xy - z^2, y^4 - z^5\}$, the *quasi-component* $W(T) = V(xy - z^2, y^4 - z^5) \backslash V(y)$ is $W(T) = \{(x, y, z) | xy - z^2 = 0, y^4 - z^5 = 0, y \neq 0\}$. The saturate ideal of T is $\mathrm{sat}(T) =< x^3 - yz, xy - z^2, y^4 - z^5, xz^3 - y^3, zx^2 - y^2 >$.

Regular chain. A triangular set T is a *regular chain* if either T is empty, or $T - \{T_{max}\}$ is a regular chain and the initial of T_{max} is regular with respect to $\mathrm{sat}(T - \{T_{max}\})$. In this latter case, $\mathrm{sat}(T)$ is a proper ideal of $\mathbb{K}[\mathbf{x}]$. From now on $T \subset \mathbb{K}[\mathbf{x}]$ is a regular chain; moreover we write $m = |T|$, $\mathbf{s} = \mathrm{mvar}(T)$ and $\mathbf{u} = \mathbf{x} \setminus \mathbf{s}$. The ideal $\mathrm{sat}(T)$ enjoys several properties. First, its zero-set equals $\overline{W(T)}$. Second, the ideal $\mathrm{sat}(T)$ is unmixed with dimension $n - m$. Moreover, any prime ideal $\mathfrak{p}$ associate to $\mathrm{sat}(T)$ satisfies $\mathfrak{p} \cap \mathbb{K}[\mathbf{u}] = \langle 0 \rangle$. Third, if $n = m$, then $\mathrm{sat}(T)$ is simply $\langle T \rangle$. Given $p \in \mathbb{K}[\mathbf{x}]$ the *pseudo-remainder* (resp. *iterated resultant*) of p w.r.t. T, denoted by $\mathrm{prem}(p, T)$ (resp. $\mathsf{res}(p, T)$) is defined as follows. If $p \in \mathbb{K}$ or no variables of p is algebraic w.r.t. T, then $\mathrm{prem}(p, T) = p$ (resp. $\mathsf{res}(p, T) = p$). Otherwise, we set $\mathrm{prem}(p, T) = \mathrm{prem}(r, T_{<v})$ (resp. $\mathsf{res}(p, T) = \mathsf{res}(r, T_{<v})$) where v is the largest variable of p which is algebraic w.r.t. T and r is the pseudo-remainder (resp. resultant) of p and T_v w.r.t. v. The following holds: p is null (resp. regular) w.r.t. $\mathrm{sat}(T)$ if and only if $\mathrm{prem}(p, T) = 0$ (resp. $\mathsf{res}(p, T) \neq 0$).

2.3.4 Subresultants

We follow the presentation of [31]. Other references that we have used are [47, 93, 35].

Determinantal polynomial. Let $\mathbb{A}$ be a commutative ring with identity and let $m \leq n$ be positive integers. Let M be a $m \times n$ matrix with coefficients in $\mathbb{A}$. Let M_i be the square submatrix of M consisting of the first $m - 1$ columns of M and the i-th column of M, for $i = m \cdots n$; let $\det M_i$ be the determinant of M_i. We denote by

$\operatorname{dpol}(M)$ the element of $\mathbb{A}[X]$, called the *determinantal polynomial* of M, given by

$$\det M_m X^{n-m} + \det M_{m+1} X^{n-m-1} + \cdots + \det M_n.$$

Note that if $\operatorname{dpol}(M)$ is not zero then its degree is at most $n-m$. Let $P_1, \ldots, P_m$ be polynomials of $\mathbb{A}[X]$ of degree less than n. We denote by $\operatorname{mat}(P_1, \ldots, P_m)$ the $m \times n$ matrix whose i-th row contains the coefficients of P_i, sorting in order of decreasing degree, and such that P_i is treated as a polynomial of degree $n-1$. We denote by $\operatorname{dpol}(P_1, \ldots, P_m)$ the determinantal polynomial of $\operatorname{mat}(P_1, \ldots, P_m)$.

Subresultant. Let $P, Q \in \mathbb{A}[X]$ be non-constant polynomials of respective degrees p, q with $q \leq p$. Let d be an integer with $0 \leq d < q$. Then the d-th *subresultant* of P and Q, denoted by $S_d(P, Q)$, is

$$\operatorname{dpol}(X^{q-d-1}P, X^{q-d-2}P, \ldots, P, X^{p-d-1}Q, \ldots, Q).$$

This is a polynomial which belongs to the ideal generated by P and Q in $\mathbb{A}[X]$. In particular, $S_0(P, Q)$ is $\operatorname{res}(P, Q)$, the resultant of P and Q. Observe that if $S_d(P, Q)$ is not zero then its degree is at most d. When $S_d(P, Q)$ has degree d, it is said *non-defective* or *regular*; when $S_d(P, Q) \neq 0$ and $\deg(S_d(P, Q)) < d$, $S_d(P, Q)$ is said *defective*. We denote by s_d the coefficient of $S_d(P, Q)$ in X^d. For convenience, we extend the definition to the q-th subresultant as follows:

$$S_q(P, Q) = \begin{cases} \gamma(Q)Q, & \text{if } p > q \text{ or } \operatorname{lc}(Q) \in \mathbb{A} \text{ is regular} \\ \text{undefined}, & \textit{otherwise} \end{cases}$$

where $\gamma(Q) = \operatorname{lc}(Q)^{p-q-1}$. Note that when p equals q and $\operatorname{lc}(Q)$ is a regular element in $\mathbb{A}$, $S_q(P, Q) = \operatorname{lc}(Q)^{-1}Q$ is in fact a polynomial over the total fraction ring of $\mathbb{A}$.

We call *specialization property of subresultant sequence* the following statement. Let $\mathbb{B}$ be another commutative ring with identity and Ψ a ring homomorphism from $\mathbb{A}$ to $\mathbb{B}$ such that we have $\Psi(\operatorname{lc}(P)) \neq 0$ and $\Psi(\operatorname{lc}(Q)) \neq 0$. Then we have

$$S_d(\Psi(P), \Psi(Q)) = \Psi(S_d(P, Q)).$$

For example, the subresultant chain of $F_1 = x_2^4 + x_1x_2 + 1$ and $F_2 = 4x_2^3 + x_1$ is as follows:

$$\begin{aligned} S_4 &= x_2^4 + x_1x_2 + 1 \\ S_3 &= 4x_2^3 + x_1 \\ S_2 &= -4(3x_1x_2 + 4) \\ S_1 &= -12x_1(3x_1x_2 + 4) \\ S_0 &= -27x_1^4 + 256 \end{aligned}$$

Divisibility relations of subresultants. The subresultants $S_{q-1}(P,Q)$, $S_{q-2}(P,Q)$, $\ldots, S_0(P,Q)$ satisfy relations which induce an Euclidean-like algorithm for computing them. Following [31] we first assume that $\mathbb{A}$ is an integral domain. In the above, we simply write S_d instead of $S_d(P,Q)$, for $d = q-1,\ldots,0$. We write $A \sim B$ for $A, B \in \mathbb{A}[X]$ whenever they are associated. A is associated with B if and only if the following condition hold

$$aA = bB,\ a,b{\in}\mathbb{A}$$

For $d = q-1,\ldots,1$, we have:

(r_{q-1}) $S_{q-1} = \mathrm{prem}(P, -Q)$, the pseudo-remainder of P by $-Q$,

$(r_{<q-1})$ if $S_{q-1} \neq 0$, with $e = \deg(S_{q-1})$, then the following holds: $\mathrm{prem}(Q, -S_{q-1}) = \mathrm{lc}(Q)^{(p-q)(q-e)+1} S_{e-1}$,

(r_e) if $S_{d-1} \neq 0$, with $e = \deg(S_{d-1}) < d-1$, thus S_{d-1} is defective, and we have

- (i) $\deg(S_d) = d$, thus S_d is non-defective,
- (ii) $S_{d-1} \sim S_e$ and $\mathrm{lc}(S_{d-1})^{d-e-1} S_{d-1} = {s_d}^{d-e-1} S_e$, thus S_e is non-defective,
- (iii) $S_{d-2} = S_{d-3} = \cdots = S_{e+1} = 0$,

(r_{e-1}) if S_d, S_{d-1} are non zero, with respective degrees d and e then we have $\mathrm{prem}(S_d, -S_{d-1}) = \mathrm{lc}(S_d)^{d-e+1} S_{e-1}$,

We consider now the case where $\mathbb{A}$ is an arbitrary commutative ring, following Theorem 4.3 in [35]. If S_d, S_{d-1} are non zero, with respective degrees d and e and if s_d is regular in $\mathbb{A}$ then we have $\mathrm{lc}(S_{d-1})^{d-e-1} S_{d-1} = {s_d}^{d-e-1} S_e$; moreover, there exists $C_d \in \mathbb{A}[X]$ such that we have:

$$(-1)^{d-1}\mathrm{lc}(S_{d-1})\mathrm{coeff}(S_e, X^e) S_d + C_d S_{d-1} = \mathrm{lc}(S_d)^2 S_{e-1}.$$

In addition $S_{d-2} = S_{d-3} = \cdots = S_{e+1} = 0$ also holds.

2.3.5 Regular GCD

Regular GCD. Let I be the ideal generated by $\sqrt{\mathrm{sat}(T)}$ in $\mathbb{K}[x_1, \ldots, x_{n-1}][x_n]$. Then $\mathbb{L}(T) := \mathbb{K}(\mathbf{u})[\mathbf{s}]/I$ is a direct product of fields. It follows that every pair of univariate polynomials $p, t \in \mathbb{L}(T)[y]$ possesses a GCD in the sense of [76]. The following GCD notion [75] is convenient since it avoids considering radical ideals. Let $T \subset \mathbb{K}[x_1, \ldots, x_{n-1}]$ be a regular chain and let $p, t \in \mathbb{K}[\mathbf{x}]$ be two non-constant polynomials with the same main variable x_n. Assume that the initials of p and t are regular modulo $\mathrm{sat}(T)$. A non-zero polynomial $g \in \mathbb{K}[\mathbf{x}]$ is a *regular GCD* of p, t *w.r.t.* T if the following conditions hold:

(i) $\mathrm{lc}(g, x_n)$ is regular with respect to $\mathrm{sat}(T)$;

(ii) there exist $u, v \in \mathbb{K}[\mathbf{x}]$ such that $g - up - vt \in \mathrm{sat}(T)$;

(iii) if $g \notin \mathbb{K}$ and $\mathrm{mvar}(g) = x_n$ hold, then $\langle p, t\rangle \subseteq \mathrm{sat}(T \cup g)$.

In this case, the polynomial g has several properties. First, it is regular with respect to $\mathrm{sat}(T)$. Moreover, if $\mathrm{sat}(T)$ is radical and g has positive degree in x_n, then the ideals $\langle p, t\rangle$ and $\langle g\rangle$ of $\mathbb{L}(T)[x_n]$ are equal, so that g is a GCD of (p, t) w.r.t. T in the sense of [76]. The notion of regular GCD can be used to compute intersections of algebraic varieties. As an example we will make use of the following formula which follows from Theorem 32 in [75]. Assume that the regular chain T is simply $\{r\}$ where $r = \mathsf{res}(p, t, x_n)$, for $r \notin \mathbb{K}$, and let h is the product of the initials of p and t. Then, we have:

$$V(p, t) = \overline{W(r, g)} \ \cup \ V(h, p, t). \tag{2.25}$$

where $\overline{W(r, g)}$ is the algebraic closure of the quasi-component of r and g.

Splitting. Two polynomials p, t may not necessarily admit a regular GCD w.r.t. a regular chain T, unless $\mathrm{sat}(T)$ is prime, see our Example 1 of Section 7.3 at Page 95. However, if T is "split" into several regular chains, then p, t may admit a regular GCD w.r.t. each of them. To this end, we need a notation. For non-empty regular chains $T, T_1, \ldots, T_e \subset \mathbb{K}[\mathbf{x}]$ we write $T \longrightarrow (T_1, \ldots, T_e)$ whenever we have $\mathrm{mvar}(T) = \mathrm{mvar}(T_i)$ for all $1 \leq i \leq e$, $\mathrm{sat}(T) \subseteq \mathrm{sat}(T_i)$ and $\sqrt{\mathrm{sat}(T)} = \sqrt{\mathrm{sat}(T_1)} \cap \cdots \cap \sqrt{\mathrm{sat}(T_e)}$. If this holds, observe that any polynomial h regular w.r.t $\mathrm{sat}(T)$ is also regular w.r.t. $\mathrm{sat}(T_i)$ for all $1 \leq i \leq e$.

Chapter 3

Foundational Fast Polynomial Arithmetic and its Implementation

3.1 Overview

As mentioned in Section 1.2, one of the major contributions of this thesis is that we have developed a set of highly efficient implementation operations of asymptotically fast polynomial arithmetic and integrated it into several computer algebra systems. The existing fast polynomial arithmetic such as fast multiplication, division, fast GCD are the efficiency-critical ones. We report the implementation effort on these operations in this chapter and Chapters 4, 5, 9. Based on these implementations, we have developed new higher level polynomial operations for polynomial system solving. The new algorithms and new implementation result will be reported in Chapters 6, 7 and 8.

Asymptotically fast algorithms for polynomial arithmetic have been known for more than forty years. Among others, the work of Karatsuba [57], Cooley and Tukey [25], and Strassen [88] has initiated an intense activity in this area. Unfortunately, its impact on computer algebra systems has been reduced until recently. One reason was, probably, the belief that these algorithms were of very limited practical interest. In [45] p. 132, referring to [73], the authors state that the FFT-based univariate polynomial multiplication is "better than the classical method approximately when $n + m \geq 600$", where n and m are the degrees of the input polynomials. In [58] p. 501, quoting [18], Knuth writes "He (R. P. Brent) estimated that Strassen's scheme would not begin to excel over Winograd's until $n \approx 250$ and such enormous matrices rarely occur in practice unless they are very sparse, when other techniques apply."

Moreover, the implementation of asymptotically fast arithmetic was not the primary concern of the early computer algebra systems, which had many other challenges to face. For instance, one of the main motivations for the development of the AXIOM computer algebra system [52] was the design of a language where mathematical properties and algorithms could be expressed in a natural and efficient manner. Nevertheless, successful implementations of the FFT-based univariate polynomial multiplication [73] and Strassen's matrix multiplication [10] have been reported for several decades.

In the last decade, several software for performing symbolic computations have put a great deal of effort in providing outstanding performances, including successful implementation of asymptotically fast arithmetic. As a result, the general-purpose computer algebra system MAGMA [5] and the Number Theory Library NTL [6] have set world records for polynomial factorization and determining orders of elliptic curves. The book *Modern Computer Algebra* [44] has also contributed to increase the general interest of the computer algebra community for these algorithms. As to linear algebra, in addition to MAGMA, let us mention the C++ template library LinBox [7] for exact linear algebra computation with dense, sparse, and structured matrices over the integers and over finite fields. A cornerstone of this library is the use of BLAS libraries such as ATLAS to provide high-speed routines for matrices over small finite fields, through floating-point computations [33].

However little has been reported on the details of such effort. In this chapter, we mainly discuss how we achieve high performance for some well-studied fast polynomial algorithms in two high-level programming environments, ALDOR and AXIOM. Two approaches are investigated. With ALDOR we rely only on high-level generic code, whereas with AXIOM we endeavor to mix high-level, middle-level and low-level specialized code. We show that our implementations are satisfactory compared to other well-known computer algebra systems or libraries such as MAGMA v2.11-2 and NTL v5.4.

The outline of this chapter is as follows. Section 3.2 is an overview of the language features of AXIOM and ALDOR systems. In Sections 3.3 and 3.4, we discuss our implementation techniques in the ALDOR and AXIOM. In Section 3.5 we report our experimentation result. Our implementations in ALDOR generic code are only approximately twice slower than the highly optimized C++ implementation in of NTL. Our specialized implementation in AXIOM leads to comparable performance and sometimes outperforms those of MAGMA and NTL. All timings given in this chapter are obtained on a bi-Pentium 4, 2.80 GHz machine, with 1 Gb of RAM.

`NOTE: This chapter is written based on the published paper [65].`

3.2 High Level Programming Environment

AXIOM and ALDOR are the first two computer algebra systems on which we conduct our experimentation. We use the word "experimentation" since we have tried a few methods to speed up polynomial packages in these two systems by plugging in our new asymptotically fast implementation. The most appropriate methods and implementation are finally integrated in MAPLE as reported in Chapter 7 and 8.

Recall that in Section 2.2 at Page 17 we have provided a brief introduction of AXIOM and an example of its type system. Originally ALDOR is an extension language from AXIOM, thus it shares many language features. In the following text we describe the language features of these two systems. Primarily, AXIOM and ALDOR designers attempted to surmount the challenges of providing an environment for implementing the extremely rich relationships among mathematical structures. Hence, their design is of somewhat different direction than that of other contemporary programming languages. They have a two-level object model of *categories* (see the example: the AXIOM `Ring` category in Section 2.2) and *domains* that is similar to *Interfaces* and *Classes* in Java. They provide a type system that allows the programmer the flexibility to extend or build on existing types or create new type categories as is usually required in algebra.

In AXIOM and ALDOR, types and functions can be constructed and manipulated within programs dynamically like the way values are manipulated. This makes it easy to create generic programs in which independently developed components are combined in many useful ways. For instance, for a given AXIOM or ALDOR ring `R`, the domains `SUP(R)` and `DUP(R)`, for sparse and dense univariate polynomials respectively, provide exactly the same operations; that is they have the same user interface, which is defined by the category `UnivariatePolynomialCategory(R)`. But, of course, the implementation of the operations of `SUP(R)` and `DUP(R)` is quite different. While `SUP(R)` implements polynomials with linked lists of terms, `DUP(R)` implements them with arrays of coefficients indexed by their degrees. This allows us to specify a package, `FFTPolynomialMultiplication(R, U)`, parametrized by `R`, an `FFTRing`, that is, a ring supporting the FFT; and by `U`, a domain of `UnivariatePoly-`

`nomialCategory(R)`. After discussing the common part in AXIOM and ALDOR, we illustrate the unique features in each system environment. Based on the uniqueness we have developed suitable implementation techniques for each system respectively (see Section 3.3 and 3.4 for detail).

3.2.1 The ALDOR environment

ALDOR can be used both as a compiled and interpreted language. Code optimization is however only available in the compiled mode. An ALDOR program can be compiled into: stand-alone executable programs; object libraries in native operating system formats; portable byte code libraries; and C or LISP source [1]. Code improvement by techniques such as program specialization, cross-file procedural integration and data structure elimination, is performed at the optimization stage of the compilation [90].

3.2.2 The AXIOM environment

The general introduction of AXIOM has been given in Section 2.2. In this section, we provide more technical details. Based on these details, we can better understand how to make the lower level (GCL, C and ASSEMBLY) implementation packages available for AXIOM system. Recall that in Section 2.2, we have mentioned that AXIOM has both an interactive mode for user interactions and a high level programming language, called SPAD, for building library modules. Concretely, the compilation process in AXIOM is as follows:

- The SPAD code will be translated into COMMON LISP code by a built-in compiler.
- Then the COMMON LISP code will be translated into C code by the GCL compiler.
- Finally, GCL makes use of a native C compiler, such as GCC, to generate machine code.

Since these compilers can generate fairly efficient code, programmers can concentrate on their mathematical algorithms and write them in SPAD.

However, to achieve higher performance, our implementation also involves LISP, C, and assembly level code. By modifying the AXIOM makefiles, new LISP functions can be compiled and made available at SPAD level. Moreover, by using the GCL system provided *make-function*, one can add new C implementation in the format of functions into the GCL kernel. These new functionality will be available at the GCL and SPAD level. Finally Assembly code can either be inlined into C code or compiled into Lisp kernel images, and so available for Lisp and SPAD level.

3.3 Implementation Techniques: the Generic Case

Our goal in the generic case is to implement algorithms with quasi-linear time complexities in a high-level programming environment(Aldor), without resorting to low-level techniques. The primary focus is not to outperform other implementations of similar algorithms on other platforms, but rather to ensure that we achieve our best in terms of space and time complexities in our target environment. For instance, in the Aldor high level programming environment we write optimizer-friendly and garbage collector (GC)-friendly code without compromising the high-level nature of our implementations. The practically result shows that our efforts are effective. In Section 3.3.1 we describe the implementation techniques we developed for the efficiency-critical operations, and in Section 3.3.2 we show that the higher level algorithms in Aldor can be sped up in large scale consequently.

3.3.1 Efficiency-critical operations in Aldor

We first discuss the techniques and results of our Aldor implementation of two efficiency-critical algorithms: FFT and power series inversion as defined in Section 2.1 at Page 8.

FFT. We specify a FFT multiplication package that accepts a generic polynomial type, but performs all operations on arrays of coefficients, which are pre-allocated and released when necessary, without using the compiler's garbage collector. For coefficient fields $\mathbb{Z}/p\mathbb{Z}$, Aldor's optimizer produces code comparable to hand-optimized C code.

Power series inversion. We have implemented two versions of the power series inversion algorithm: a "naive" version without optimization and a space-efficient version. The latter implementation uses the following ideas:

- We pre-determine all array sizes and pre-allocate all needed buffers, so that there is no memory allocation in the loop.

- Even though we accept a generic polynomial type, we change the data representation to arrays of coefficients, work only with these arrays, and reuse DFT as much as possible.
- As in NTL, we use wrapped convolution to compute the n middle coefficients of a $(2n-1) \times n$ full product (this is the middle-product operation of [49]).

Figure 3.1 shows the running time of our two implementations, together with the time for a single multiplication, in a field of the form $\mathbb{Z}/p\mathbb{Z}$. We measured the maximum resident set size; Figure 3.2 shows that the naive version used a total of over 16000 Kb to invert a polynomial of degree 8000 while the space efficient version used less than 2500 Kb for the same polynomial. For examples with higher degrees, the factor of improvement is larger.

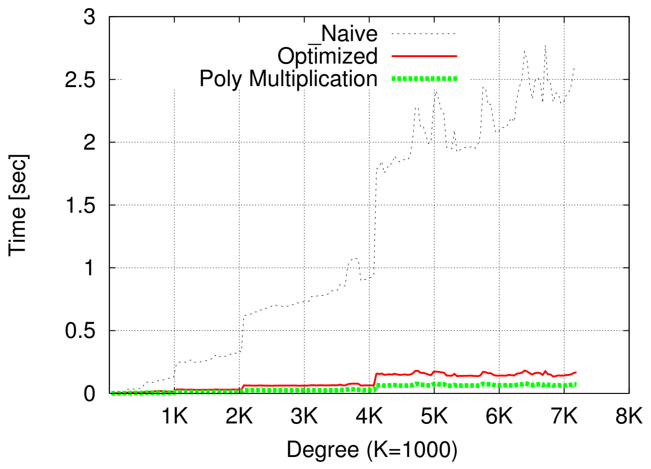


Figure 3.1: Power series inversion: naive vs. optimized implementation vs. multiplication, 27-bit prime.

We first give the source code of the naive version as follows:

```
modularInversion(f:U,n:Z):U == {
 assert(one?(trailingCoefficient(f)));
 local m,g0,g__old,g__new,mi:U;
 m: == monom;
 g0:U:=1; g__old:U:=1; g__new:U:=1;
 local r,mii:MI;
 if PowerOfTwo?(n) then r := length(n)-1;
```

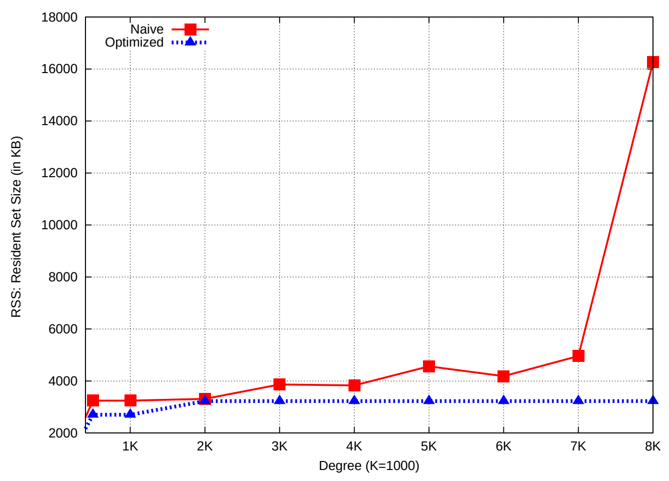


Figure 3.2: Power series inversion: space usage of naive vs. optimized implementations, 27-bit prime.

```
 else  r := length(n);

 for i in 1..r repeat {
  mi := m^(2^i);
  g__new := (2*(g__old)-(f*((g__old)*(g__old)))) mod mi;
  g__old := g__new;
 }
 return (g__new);
}
```

Then follows the source code of the efficient version:

```
macro {
    U == DenseUnivariatePolynomial(K:Field);
    Z == AldorInteger;
}
fastModInverse(f:U,n:Z):U == {
  import from Z,MI;
  local dftf,dftg,Y,G,workspace,dftw,op,coeff:AK;
  local di__1,di,r,mii:MI; local res:U; local wi:K;

  if PowerOfTwo?(n) then r := length(n)-1;
    else  r := length(n);
  nn:MI := shift(1,r); -- 2^r
```

– allocate storage

```
dftg := new(nn,0$K);
Y := new(nn,0$K);
G := new(nn,0$K);
workspace := new(nn,0$K);
op := new(nn,0$K);
```

– stores g_{i-1}

```
G.0 := 1$K;
dftg.0 := 1$K;
```

– stores truncated f

```
coeff := new(nn,0$K);
dftf := new(nn,0$K);
dftw := new(nn,0$K);
kk:MI := 0;
for k in coefficients(f) repeat {
 kk = nn => break;
 coeff.kk := k; kk := next(kk);
}
for i in 1..r repeat {
 mii := shift(1,i); -- 2^i
```

– degree of g_i

```
 di := mii - 1;
 w:Partial K := primitiveRootOfUnity(mii);
 wi := retract(w);
```

– op stores OmegaPowers up to mii

```
OmegaPowers!(op,wi,mii);
dftg := dft!(dftg,mii,i,op,workspace);
```

– $f \bmod X^{2^i}$**: truncates f**

```
for j in 0..di repeat dftf.j := coeff.j;
dftf := dft!(dftf,mii,i,op,workspace);
```

– dftf*dftg pointwise

```
    for j in 0..di repeat dftf.j := dftf.j*dftg.j;
    dftf := idft!(dftf,mii,i,op,workspace); -- invert dft
    di__1 := shift(1,i-1) - 1; --  degree of g_i_1
    ndi__1 := next di__1;
```

– takes the end part

```
    kk:=0;
    for j in ndi__1..di repeat {
      dftw.kk := dftf.j; kk:=next kk;
    }
    dftw := dft!(dftw,mii,i,op,workspace);
    for j in 0..di repeat dftg.j := dftg.j*dftw.j;
    dftg := idft!(dftg,mii,i,op,workspace);
```

– $X^{\texttt{ndi\_\_1}} * Y$: the middle product

```
    for j in 0..di__1 repeat Y.(j+(ndi__1)) := dftg.j;
    for j in ndi__1..di repeat G.j := G.j - Y.j;
```

– to allow dft! in-place of G, save G

```
      for j in 0..di repeat dftg.j := G.j;
  }
```

– convert to polynomial

```
  res := unvectorize(dftg,nn);
  free!(dftg); free!(dftf); free!(dftw); free!(workspace);
  free!(op); free!(coeff);
  return res;
}
```

3.3.2 Extended Euclidean algorithm

We implemented the Half-GCD algorithms of [93] and [19], adapted to yield monic remainders. The algorithms given in Section 2.1 at Page 8 contain the adaptation we made. Our implementation of Euclidean division uses power series inversion [43, Ch. 9], when the degree difference between two consecutive remainders is large enough. We use Strassen's algorithm [43, Ch. 13] for the 2×2 polynomial matrix multiplication; This implementation outperforms the standard Euclidean algorithm by a factor of 8 at degree 3000.

3.4 Implementation Techniques: the Non-generic Case

For AXIOM the non-generic case, we put additional efforts on investigating the efficiency of the compiled code. The reasons are as following. First, we are curious that, to what extend, a compiler optimizes our polynomial applications. Second, our work is largely motivated by the implementation of modular methods. High performance for these methods relies on appropriately utilizing machine arithmetic as well as carefully constructing underlying data representation. This leads us to look into machine-level questions, such as machine integer arithmetic, memory hierarchy, and processor architecture. At this level, C is preferred and assembly is used if necessary. Third, we are interested in parallel programming, which is not available at SPAD level, but can be achieved in LISP and C (see Chapter 9 at Page 139 for our parallel implementation result). In the following text, we focus on the major efforts: suitable data representation, SIMD instructions, loop unrolling and thread-level parallelism.

3.4.1 Data representation

We use *dense* polynomials as the data representation. We have in mind to implement algorithms for solving polynomial systems by modular methods over $\mathbb{Z}/p\mathbb{Z}$. Polynomials appearing in such applications tend to become "densified" due to intensive use of Euclidean algorithm, Hensel lifting techniques, etc.

In concrete terms, elements of the prime field $\mathbb{Z}/p\mathbb{Z}$ are encoded by integers in the range $0, \ldots, p-1$. This allows us to use C-like arrays such as `fixnum-array` in LISP to encode polynomials in $\mathbb{Z}/p\mathbb{Z}[X]$. If p is small enough, we tell the compiler to use machine integer arithmetic; for large p, we use the Gnu Multiple Precision library (GMP).

To test the best performance, we write C and assembly code for the operations such as univariate polynomial addition, multiplication : we pass the array of references to our C and assembly code, then return the result back to AXIOM. In the final implementation as reported in later chapters, we avoid using assembly code for maintaining the good code portability.

We compare the performance of two univariate polynomial constructors `SUP` and `UMA`. `SUP` is a pure SPAD level implementation, and `UMA` is written in LISP, C and assembly with a SPAD level wrapper. `UMA` means Univariate Modular Arithmetic, since it is designed for polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. Over a 64-bit prime field, `UMA` addition of polynomials is up to 20 times faster than `SUP` addition, in degree 30000; the quadratic `UMA` implementation of polynomial multiplication is up to 10 times faster than `SUP` multiplication, in degree 5000. The FFT multiplication will be discussed in later text. The `UMA` implementation is integrated into AXIOM library and used in an user-transparent way, thanks to the concept of *conditional implementation* in AXIOM. Namely, on the condition where polynomial computation is over $\mathbb{Z}/p\mathbb{Z}$, `UMA` will be automatically used.

Similarly, we have implemented a specialized multivariate polynomial domain over $\mathbb{Z}/p\mathbb{Z}$. The operations in this domain are mostly implemented at the LISP level which offers us more flexibility (less type checking, better support from the machine arithmetic) than at the SPAD level. We follow the *vector-based approach* proposed by Fateman [38] where a polynomial is either a number or a vector: If a coefficient is a polynomial, then the corresponding slot of the "parent" vector keeps a reference to that polynomial or, say, another vector; otherwise, if the coefficient is a number, the slot keeps that number.

3.4.2 The implementation of FFT

Our implementation of FFT-based univariate polynomial multiplication in $\mathbb{Z}/p\mathbb{Z}[X]$ distinguishes the cases of small (single-precision) primes and big (multiple-precision) primes. For the big prime case, one can either directly use the big integer arithmetic, Or use the Chinese Remainder Theorem (CRT) based approach. The general principle of CRT is to reduce the big integer problem into 2 or more smaller integer problems [86, 43].

For both small and big prime cases, we used the algorithm of [26] and techniques discussed in Subsection 3.4.3 below. Figure 3.3 shows a comparison between these two approaches. We put special effort on the big prime case. We rewrite some GMP low-level functions for double word size prime arithmetic which is the most useful case in our polynomial computation. Figure 3.3 shows that the specialized double precision big prime functions and CRT approaches are faster than the generic GMP functions. The CRT recombination part spends a negligible 0.06% to 0.07% percent of the time in the whole FFT algorithm.

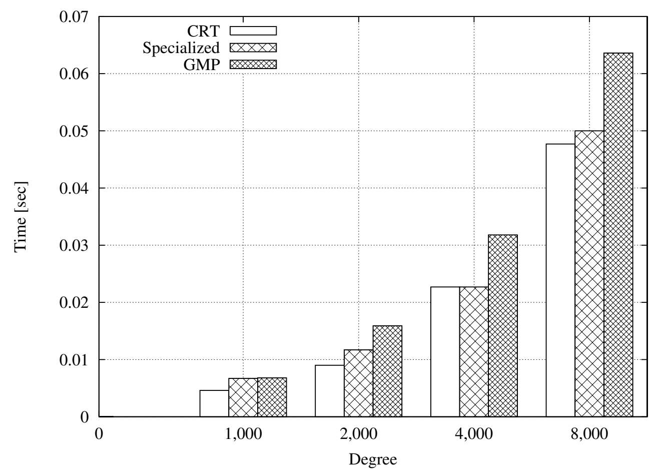


Figure 3.3: FFT multiplication: GMP functions vs. double precision integer functions vs. CRT, 64 bit prime.

3.4.3 SSE2, loop unrolling, parallelism

Modern compilers can generate highly efficient code, however for some cases the hand-tuned code still outperforms the compiler optimization. We show three examples of hand-tuned improvement from our FFT implementation.

Single precision integer division with SSE2. The single precision modular reduction uses floating point arithmetic, based on the formula $a \equiv a - \lfloor a * 1/p \rfloor * p$ [86]. We have implemented this idea in assembly for the Pentium IA-32 architecture with SSE2 support. This set of instructions is Single Instruction Multiple Data (SIMD); they make use of XMM registers which pack 2 double floats or 4 single floats/integers in one single register. The following sample code computes $(a * b) \mod p$ with SSE2 instructions.

1	movl RPTR, %edx	11	movups (%eax), %xmm0
2	movl WD1, %eax	12	cvttpd2pi %xmm2, %mm2
3	movl WPD1, %ecx	13	cvtpi2pd %mm2, %xmm2
4	movq (%edx), %mm0	14	mulpd %xmm2, %xmm0
5	movups (%eax), %xmm1	15	subpd %xmm0, %xmm1
6	cvtpi2pd %mm0, %xmm0	16	cvttpd2pi %xmm1, %mm1
7	movups (%ecx), %xmm2	17	movq %mm1, (%edx)
8	movl PD, %eax	18	emms
9	mulpd %xmm0, %xmm1	19	ret
10	mulpd %xmm0, %xmm2		

Figure 3.4 shows that our SSE2-based `FFT` implementation is significantly faster than our generic assembly version.

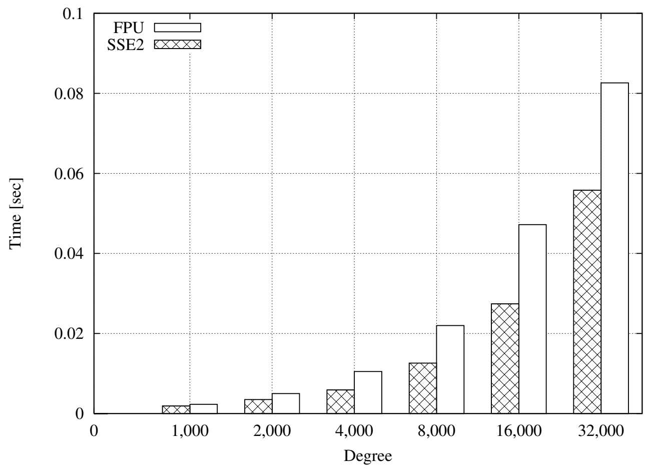


Figure 3.4: FFT multiplication: generic assembly vs. SSE2 assembly, 27-bit prime.

Reducing loops overhead. Many algorithms operating on dense polynomials have an iterative structure. One major overhead for such algorithms is loop indexing and loop condition testing. We can reduce this overhead by unrolling loops. This technique is provided by some compilers. For example GCC has a compiler option `funroll-loops` which may unroll the loops when certain conditions are satisfied.

However, there is a trade-off: although the overhead mentioned above can be reduced after loop unrolling, the transformed code may suffer from code size growth which will aggravate the burden of instruction caching. If the loop body contains branching statements, increased number of branches in each iteration will have a negative impact on branch prediction. Hence, compilers and interpreters usually do static or run-time analysis to decide how much to unroll a loop. However, the analysis may not be precise when loops become complex and nested. Moreover, compilers usually do not check if there is a possibility to combine the unrolled straight line statements for better performance. Therefore, we have unrolled some loop structures by hand to better control the trade-off mentioned above. We have also recombined the "flat" code (after the unrolling) into small assembly functions. This allows us to keep some values in registers or evict those unwanted ones at the most suitable time. Our purpose is to investigate how much the hand-tuned code outperforms the compiler optimized code. The assembly implementation is not a part of our final library implementation due to the portability and maintainability issue.

The following is a fragment of our implementation of the FFT-based univariate polynomial multiplication.

```
#include "fftdfttab_4.h"
typedef void (* F) (long int *, long int,  long int,
                         long int *, long int, int);
typedef void (* G) (long int *, long int *,
                         long int *, long int, int);
inline void
fftdftTAB_4( long int * a, long int * b, long int * w,
                         long int p, F f, G g1, G g2 ){
long int w0=1, w4=w[4], * w8=w+8;
f(a, w0, w4, a+2, p, 8); g2(a+4, w8, a+8, p, 4);
g2(a+12, w8, a+16, p, 4); g1(a+8, w8, a+16, p, 8);
f(b, w0, w4, b+2, p, 8); g2(b+4, w8, b+8, p, 4);
g2(b+12, w8, b+16, p, 4); g1(b+8, w8, b+16, p, 8); return;}
```

This function is dedicated to compute the case where $n = 4$ (see Section 2.1 at Page 8 in the FFT algorithm. The functions `f, g1, g2` are small assembly functions which recombine the “flat” (straight-line) statements for higher efficiency. We also developed similar functions for the cases from $n = 5$ to 8. However, starting for $n \geq 6$, these straight-line functions are less efficient than the ones using original loop structure, for the reason of code growth. Figure 3.5 shows that for the small degree examples, the loop-unrolling version may gain about 10% of the running time of the complete FFT computation. Actually, this is a significant improvement, since there are at least 50% time spending on integer division which is irrelevant to loop-unrolling.

Parallelism. Parallelism is a fundamental technique used to achieve high performance. In the FFT-based polynomial multiplication, the DFT of the input polynomials are independent, hence, they can be computed simultaneously. Another example is the (standard) Chinese remaindering algorithm, where the computations w.r.t. each modulo can be performed in parallel. This can be achieved by thread-level parallelism. However, AXIOM compiler doesn’t generate parallel code. Therefore, we directly use the native Posix Thread Library to achieve explicit thread-level parallelism. In Chapter 9 we report our parallel implementation for more complex algorithms.

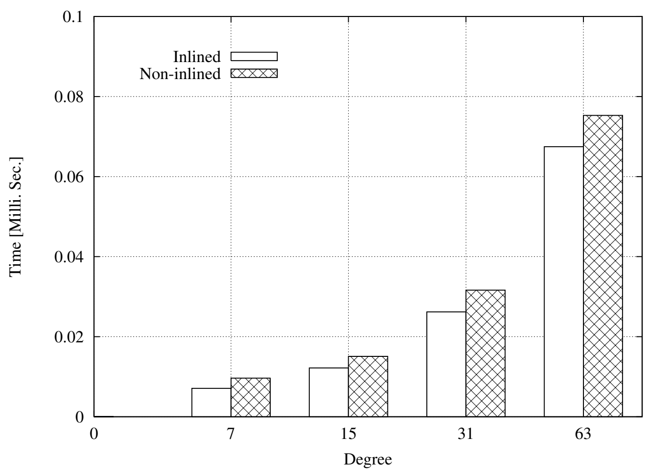


Figure 3.5: FFT multiplication: inlined vs. non-inlined, 27-bit prime.

3.5 Performance

In this section, we provide a set of benchmark results. These benchmark programs are implemented either in ALDOR high level code, or in AXIOM mixing code or in both. The performance of our code demonstrates that by using suitable implementation techniques asymptotically fast polynomial arithmetic can outperform the classical one with relatively low cut-off.

3.5.1 FFT multiplication

We compared our implementations with their counterparts in NTL and MAGMA. For NTL-v5.4, we used the functions `FFTMul` in the classes `zz_p` and `ZZ_p`, respectively for small and big primes. For MAGMA-v2.11-2, we used the general multiplication function "*" over GF(p), the prime field with the prime number p. The input polynomials are randomly generated, with no zero term. In the non-generic case, as shown in Figures 3.6 and 3.7, our AXIOM implementation is faster than NTL's over small primes, but slower than NTL over big primes; but we are faster than MAGMA and other known computer algebra systems in both cases. One possible reason is that NTL re-arranges the computations in a more "cache-friendly" way. In the generic case, the ALDOR implementation is comparable to (generally slightly slower than) MAGMA's counterpart. ALDOR's implementation is at pure high level with high level abstraction of coding, thus, the performance is still satisfactory.

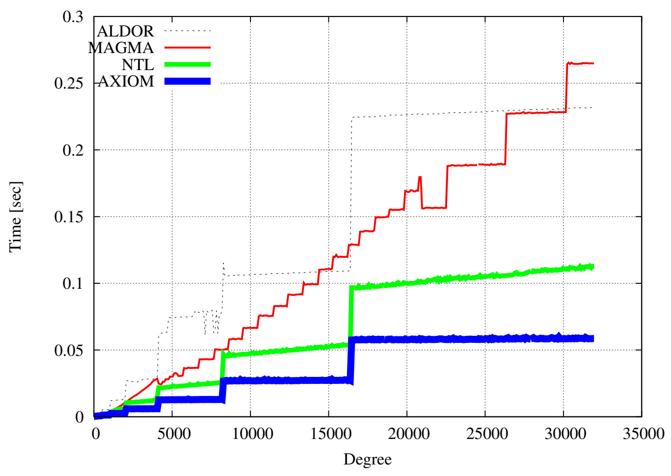


Figure 3.6: Multiplication modulo a 27-bit prime.

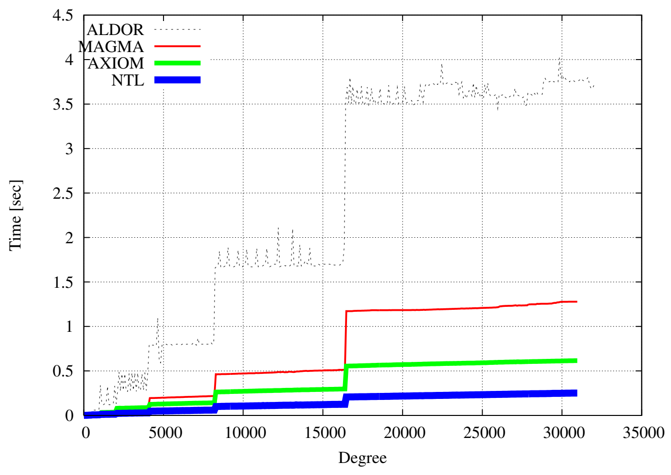


Figure 3.7: Multiplication modulo a 64-bit prime.

3.5.2 Multivariate multiplication

We compute the product of multivariate polynomials via the Kronecker substitution (see the appendix). Recall that we use vector-based recursive representation for multivariate polynomials, and one-dimensional arrays for univariate ones. So, the forward substitution simply copies coefficients from the coefficient tree of a multivariate polynomial to the coefficient array of an univariate polynomial. We use a recursive depth first tree walk to compute all the univariate polynomial exponents from the corresponding multivariate monomials' exponents; at the same time, according to this correspondence we conduct the forward substitution. We use the same idea for the backward substitution. The comparisons between MAGMA and our AXIOM code are given in Figures 3.8 to 3.10, where "degree" denotes the degree of the univariate polynomials obtained through Kronecker's substitution. We used random inputs, with no zero terms.

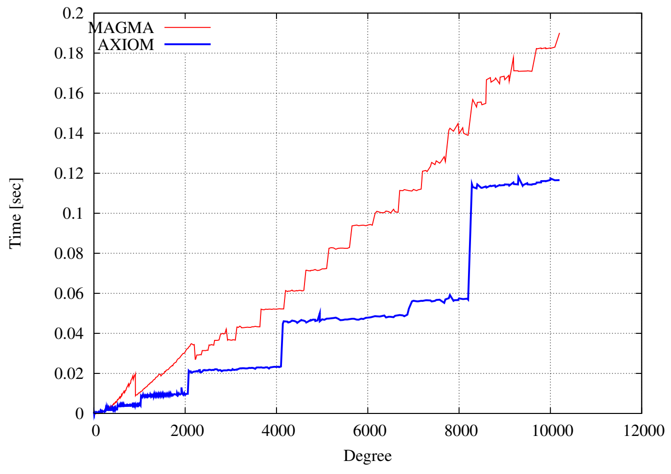


Figure 3.8: Bivariate multiplication, 27-bit prime.

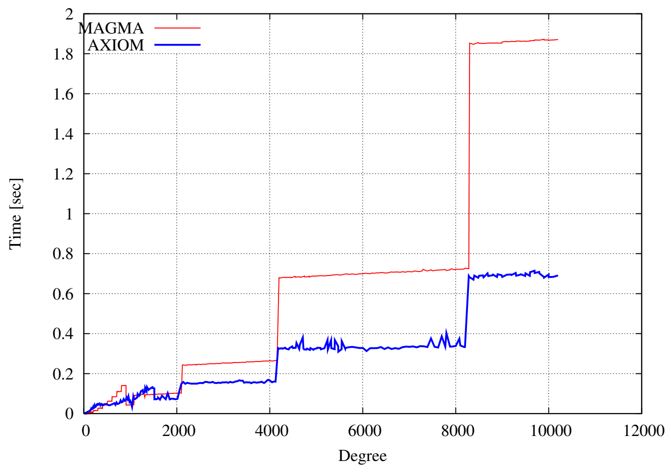


Figure 3.9: Bivariate multiplication, 64-bit prime.

Our FFT-based multivariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ outperforms Magma's in these cases. Figure 3.8 may infer that Magma is in the "classical multiplication" stage; our FFT-based implementation is already faster. From Figures 3.9, 3.10 we observe that both our and Magma's FFT's show the FFT staircase-like curves.

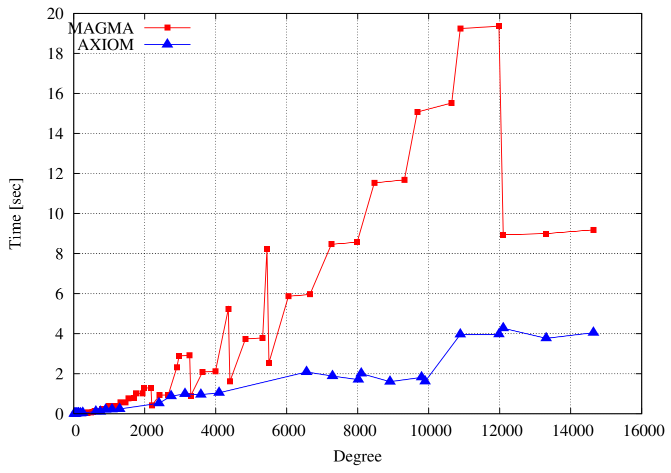


Figure 3.10: Four-variable multiplication, 64-bit prime.

3.5.3 Power series inversion

We compare here the power series inversion, in the optimized Aldor version, with NTL's and Magma's implementations. Magma offers a built-in `InverseMod` function (called "builtin" in the figure), but the behavior of this generic function is that of an extended GCD computation. We have also compared the Magma `PowerSeriesRing` domain inversion (called "powerseries" in the figure) with our own implementation of the Newton iteration. Figure 3.11 shows the relative performances: NTL is the fastest one in this case, and Aldor is the second, within a factor of 2 slower.

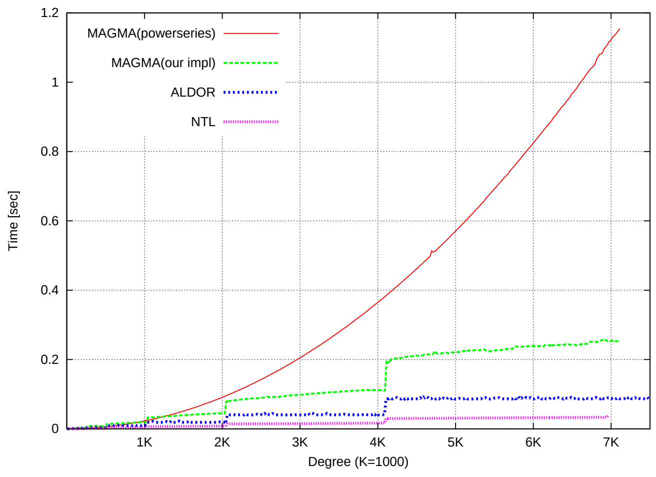


Figure 3.11: Power series inversion: Aldor vs. NTL vs. MAGMA, 27-bit prime.

3.5.4 Fast extended Euclidean algorithm

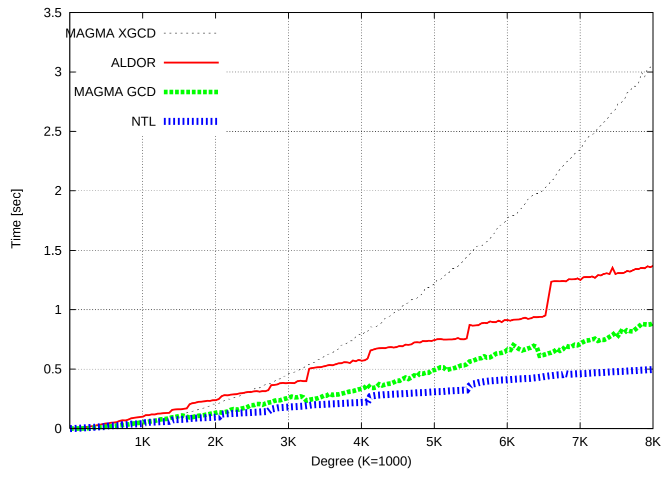


Figure 3.12: EEA: ALDOR vs. NTL vs. MAGMA, 27-bit prime.

In Section 3.3.2 we have reported the relative performance between the existing standard (non-fast) Euclidean algorithm in Aldor and the implementation of the fast algorithm. We have also compared our Aldor generic fast algorithm with the existing implementations in NTL and Magma. In the following benchmark, we compare our fast extended Euclidean algorithm implementation in Aldor with NTL and Magma again. Unlike ours, the NTL implementation is not over a generic field but over a finite field, and uses improvement like FFT-based polynomial matrix multiplication. Magma's performance differs, according to whether the `GCD` or `XGCD` commands are used. Figure 3.12 shows the relative performances; our input is degree d polynomials, with a GCD of degree $d/2$. Again, NTL is the fastest and Aldor's performance is in between two flavors of Magma's implementation (using `GCD` or `XGCD`).

3.6 Summary

The work reported in this chapter is the beginning of a large scale effort; The result from this chapter demonstrates that asymptotically fast polynomial arithmetic can outperform the classical one with relatively low cut-off. The implementation technique is highly important for reducing the overhead in these fast methods. After replacing the classical polynomial arithmetic by the fast ones, the higher level algorithms can be sped up in a significant manner.

Chapter 4

Efficient Implementation of Polynomial Arithmetic in a Multiple-level Programming Environment

4.1 Overview

In Chapter 3 we have discussed the asymptotically fast polynomial arithmetic and our preliminary implementation effort towards high performance. In this chapter we proceed to more intensive investigation on the implementation technique itself. More specifically, we investigate the implementation techniques suited to the multiple-level language environment in AXIOM. We target on the implementation for polynomial arithmetic in this chapter. Indeed, some polynomial data types and algorithms can further take advantage of the unique features in lower level languages, such as the specialized data structures or the direct accessing to machine level arithmetic. On the other hand, some data types or algorithms maybe more abstract and suited to be implemented in a very expressive high level languages. Therefore, we are interested in the integration of polynomial data type and implementation realized at different language levels. In particular, we consider the situation for which code from different language levels can be combined together within the same application.

However, linkage to specialized code is a substantial bonus when low-level implementation can take advantage of special software or hardware features. The purpose of this study is to investigate implementation techniques for polynomial arithmetic in a multiple-level programming environment. We are interested in the integration of polynomial data type implementations realized at the different code levels. In particular, we consider situations for which code from different levels can be combined together within the same application in order to achieve high-performance. As a driving example, we use the modular algorithm of van Hoeij and Monagan [50]. We recall its specifications. Let $\mathbb{K} = \mathbb{Q}(a_1, a_2, \ldots, a_e)$ be an algebraic number field over the field $\mathbb{Q}$ of the rational numbers. Let $f_1, f_2 \in \mathbb{K}[y]$ be univariate polynomials over $\mathbb{K}$. The algorithm of van Hoeij and Monagan computes $\gcd(f_1, f_2)$. To do so, for several prime numbers p, a tower of simple algebraic extensions $\mathbb{K}_p$ of the prime field $\mathbb{Z}/p\mathbb{Z}$ is used. Arithmetic operations in $\mathbb{K}_p$ are performed by means of operations on multivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, whereas the operations on the images of f_1, f_2 modulo p are performed in the univariate polynomial ring $\mathbb{K}_p[y]$. Therefore, several types of polynomials are used simultaneously in this algorithm. This is why it is a good candidate for our study. We chose AXIOM as our implementation environment based on the following observations. AXIOM has a high-level programming language, called SPAD, which possesses all the essential features of object-oriented languages. Libraries written in SPAD implement a hierarchy of algebraic structures (groups, rings, fields, ...) and a hierarchy of algebraic domains ($\mathbb{Q}$, $\mathbb{A}$[x] for a given ring $\mathbb{A}$, ...).

As mentioned in Section 2.2 at Page 2.2, the SPAD compiler translates SPAD code into COMMON LISP, then invokes the underlying LISP compiler to generate machine code. Today, GCL [3] (GNU COMMON LISP) is the underlying LISP of AXIOM [2]. The design of GCL makes use of the native C compiler for compiling to native machine code. In addition, GCL employs the GNU Multi-Precision library (GMP) [4] for its arbitrary precision number arithmetic. Therefore, AXIOM is an efficient multiple language level system. Moreover, the complete AXIOM system is open-source. Hence, we can implement our packages at any language level and even modify the AXIOM kernel. This allows us to take advantage of each language level's strength and access machine arithmetic directly when necessary. Therefore, we believe that AXIOM, with its different implementation levels, all in open source, provides an exceptional development environment among all computer algebra systems, for the purpose of our study.

The outline of this chapter is as following. In Sections 4.2, 4.3 and 4.4 we discuss the unique features (in view of our objectives) of the SPAD, LISP, C and ASSEMBLY level from AXIOM. Implementation techniques at each level are also discussed respectively. In Section 4.6, we report our experimentation result. Our result suggests that choosing adapted data structures and writing code at suitable language level are essential for high-performance for our polynomial applications.

`NOTE: This chapter is written based on the published paper [65].`

4.2 The SPAD Level

From Section 3.2 at Page 28, we know that the SPAD language of AXIOM has a two-level object model of *categories* and *domains*. In fact, the user can define an new category or domain and add it into the library modules. The new definition is called an AXIOM *type constructor*. An AXIOM *type constructor* is simply a function which returns an AXIOM type, that is a category or a domain. For instance, `SparseUnivariatePolynomial`, abbreviated to `SUP`, is a type constructor, which takes an argument `R` of type `Ring` and returns an instance of the type: univariate polynomials over `R`, with an underlying sparse polynomial data representation. The interface (in sense of Java) of `SUP(R)` is `UnivariatePolynomialCategory(R)` where `UnivariatePolynomialCategory` is a category constructor.

The SPAD language supports *conditional exports*. This permits to implement the following statement: if `R` has type `Field` then `SUP(R)` implements `EuclideanDomain`. SPAD also supports *conditional implementation*. This is similar to the concept of generics in Java. For instance, if `R` has type `PrimeFieldCategory`, The specialized "modular integer arithmetic" package can be automatically chosen. These features of the SPAD language are important for combining different data types and achieving high-performance.

To implement an new domain constructor, the programmer may have to choose a data representation for this domain type. For example `SUP` uses sparse polynomial data representation and `DUP` uses dense polynomial data representation. After an newly defined domain or category is compiled, it becomes an AXIOM data type which can be used just like any system provided data type.

In the light of these properties of the SPAD language, we describe briefly the polynomial type constructors that we use in this study. Please, see [52] and [64] for more details. Let `R` be an AXIOM `Ring` and `V` be an AXIOM `OrderedSet`.

`SUP or UP.` As mentioned above, the domain `SUP(R)` implements the ring of univariate polynomials with coefficients in `R`. More precisely, it satisfies the AXIOM category `UnivariatePolynomialCategory(R)`. The representation of these polynomials is *sparse*, that is, only non-zero terms are encoded.

`DUP`. The domain `DUP(R)` implements `UnivariatePolynomialCategory(R)` as well. The representation is dense: all terms, null or not, are encoded.

`NSMP`. The domain `NSMP(R,V)` implements the ring of multivariate polynomials with coefficients in `R` and variables in `V`. (To be precise, it implements the AXIOM category `RecursivePolynomialCategory(R, V)`.) A non-constant polynomial f of `NSMP(R)`, with greatest variable v, is regarded as an univariate polynomial in v implemented as an element of `SUP(NSMP(R))`. Therefore, the representation is recursive and sparse.

`DRMP`. The domain `DRMP(R,V)` implements the same category as `NSMP(R,V)`. The representation is also recursive. However, it is based on `DUP` rather than `SUP`.

The constructors `SUP` and `NSMP` are provided by the AXIOM standard distribution, whereas `DUP` and `DRMP` are our implementation. As mentioned in Section 1.2 modular methods tend to "densify" the polynomial computation. Therefore, dense polynomial representation is the most suitable one for this kind of methods. Our algorithms in this thesis are mostly related to modular arithmetic, thus dense polynomials is our canonical data representation in our implementation. One example of modular algorithms we implemented as a benchmark program in this Chapter is van Hoeij and Monagan's modular GCD algorithm [50]. We will use this implementation as the principle benchmark program to test the performance of all the polynomial data types and their combination.

4.3 The Lisp Level

The domain constructors `SUP`, `DUP`, `NSMP` and `DRMP` allow the user to construct polynomials over any AXIOM `Ring`. So we say that their code is generic. Ideally, one would like to use also *conditional data representations*. For instance, one could think of a domain `U(R)` implementing univariate polynomials over `R` such that sparse polynomials have a sparse representation and dense polynomials have a dense representation. In addition, if `R` implements a prime field $\mathbb{Z}/p\mathbb{Z}$ for a machine word size prime p, one could encode each dense polynomial of `U(R)` by an array of machine words (such that the slot of index i contains the coefficient in the term of degree i). But this ideal type constructor `U` would be very difficult to be analyzed by the run-time system. Indeed, many tests would be needed for selecting the appropriate representation for the right computation, at the right moment. Therefore, we use specialized domain constructors (say, dense univariate polynomials over a prime field) canonically for a specific algorithm (for instance, the modular GCD algorithm by van Hoeij and Monagan). By experimentation, we observe that this approach is more effective than switching data representation at run-time for our application due to the overhead mentioned above.

For these reasons, we have defined at the SPAD level a specialized polynomial type constructor `MultivariateModularArithmetic`, abbreviated to `MMA`. It takes a prime integer `p` and `V` an `OrderedSet` as its arguments. `MMA(p,V)` implements the same interface as `DRMP(PF(p),V)` does where `PF(p)` is a prime field of characteristic `p`. In fact, all the concrete operations of `MMA(p,V)` have been implemented at Lisp level. The SPAD level `MMA(p,V)` domain is just a wrapper. The reason we write `MMA` at Lisp level instead at SPAD level is as following:

In `MMA`, we have used the *vector-based recursive dense* representation proposed by Richard J. Fateman [38]: a multivariate polynomial f is encoded by a number (to be precise, an integer modulo `p`) if f is constant and, otherwise, by a Lisp vector storing the coefficients of f w.r.t its leading variable. At the SPAD level, such disjunction has to be implemented by an union type bringing an extra indirectness. However, this can be avoided at the Lisp level. Not like SPAD doing strict compile time type-checking, Lisp only does run-time type-checking. Moreover, for the Lisp implementation such as GCL, the run-time type-checking can be switched off manually. Thus, f can be assigned by a number or a vector in Lisp without any compilation error and run-time overhead optionally.

In addition, in Lisp for the machine integer size prime case we can decorate the code to force the Lisp compiler to use machine integer array (fixnum-array). However this is a non-easy task in SPAD language. Even we decorate the code at SPAD level, the array type used is an array of reference to the machine integers. This array type is far less efficient than the C-like array "fixnum-array" while the dense polynomials are over $\mathbb{Z}/p\mathbb{Z}$ with p a machine integer prime. Therefore, for certain applications such as our example the Lisp level code may yield more efficient implementation comparing to the SPAD level.

We have defined at the SPAD level an univariate type constructor `UnivariateModularArithmetic`, abbreviated to `UMA`, taking a prime integer `p` as argument and implementing the same operations as `DUP(PF(p))`. It is also a SPAD level wrapper for two Lisp level implementations: one for the machine prime case and one for the big prime case. In both cases, univariate polynomials are again encoded by `fixnum-array`. It is possible directly using C arrays to encode univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, but we prefer, at this experimentation stage, the Lisp level garbage collection system which is more convenient. For the machine prime case, each entry in `fixnum-array` is a coefficient. For the big prime case, two or more entries encode one coefficient up to the size the prime number. All these specialized implementations at the Lisp level yield significant speed up comparing to the original SPAD level packages. The benchmark result is reported in Section 4.6 at Page 52.

4.4 The C Level

GCL is implemented in C language and uses the native optimizing C compiler to generate native machine code. This allows us to extend the functionality of the Lisp level in AXIOM by writing new C code. For example, we can integrate an new C function into the GCL kernel image, or add it into a GCL library.

This interoperability between Lisp and C has at least two benefits. First, our Assembly code (written for some efficiency critical operation previously, see Section 4.5 below) can be inlined in the C code, thus available for Lisp function. Second, we can directly use existing C libraries such as GMP library [4] or NTL [6] to speed up Lisp level implementation. We illustrate these two benefits by an important example: the implementation of dense univariate polynomial domain over the prime field $\mathbb{Z}/p\mathbb{Z}$, i.e the `UMA` domain constructor (see Section 4.3 at Page 48).

Recall that we have two implementation cases for `UMA`: one for small primes p (that fits in single precious machine word) and one for big primes p (that is bigger than the biggest single precious machine word). For both the small and big prime cases, we have the following code which has been integrated into the GCL kernel:

- classical multiplication, addition and Chinese remaindering algorithm written in Assembly,
- FFT-based multiplication written in C with Assembly sub-routines.

Moreover, in the big prime case, we have developed a highly efficient double precision big integer arithmetic package by modifying GMP multiprecision subroutines. This is motivated by the importance of the double precision integer computation in our application. For instance, most prime numbers used in the modular method of [27] are of that size.

4.5 The Assembly Code Level

Primarily, our Assembly level implementation is for univariate polynomial addition and multiplication. As we know that big integer arithmetic is basically the same as univariate polynomial arithmetic modulo a prime number. The only difference is the "carry" issue. So we can directly modify the existing big integer libraries such as GMP to perform univarivate polynomial arithmetic over $\mathbb{Z}/p\mathbb{Z}$. Since the related GMP operations are implemented in assembly, we directly modify their Assembly level operations and link the modified operations into AXIOM. In this way, we have avoided extra encoding effort and obtained highly efficient Assembly level polynomial operations. Besides this, there are two other reasons to use Assembly code in our AXIOM environment: "controlling register allocation" and "using architecture specific features".

4.5.1 Controlling register allocation

In a modern computer architecture, CPU registers sit at the top level of the memory hierarchy. Although optimizing compilers devote special efforts to make good use of the target machine's register set, this effort can be constrained by numerous factors, such as:

- difficulty to estimate the execution frequencies of each part of the program,
- difficulty to allocate or evict ambiguous values,
- difficulty to take advantage of some new hardware features on specific platforms.

Therefore, some high-performance oriented applications require programmers to better exploit the power of registers on a target machine. In fact, we have spent a great effort in this direction in our implementation. First, we directly program the efficiency-critical parts in Assembly language in order to explicitly manipulate data in registers. For example, for dense univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, we write the classical multiplication algorithm in both C and Assembly language. The Assembly version is faster than the C version since we always try to keep frequently used variables in registers instead of a memory location. Although in C we can declare a variable to be of "register" type as in GCC, this does not guarantee that the register is reserved for this variable. According to our benchmark results, our explicit register allocation method is always faster than the C compiler's optimization.

Besides the general purpose registers, we also can use MMX, XMM registers if they are available. Keeping the working set in registers will yield significant performance improvement comparing to keeping them in the main memory.

4.5.2 Using architecture specific features

Polynomial arithmetic in $\mathbb{Z}/p\mathbb{Z}[x]$ makes an intensive use of integer division. This integer operation has a dominant cost in crucial polynomial operations like the FFT-based multiplication over $\mathbb{Z}/p\mathbb{Z}$. Therefore, improving the performance of integer division is one of the key issues in our implementation.

In Section 3.4.3 at Page 37, we have introduced the fast integer division trick by using assembly code with SIMD instructions. Our implementation of the FFT-based polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ uses this technique. It is faster than using FPU unit, as reported in Section 4.6 at Page 52. In Section 5.2.2 at Page 59 and Section 6.3 at Page 79. we present other integer division tricks we have developed whereas implemented at C level.

4.6 Experimentation

4.6.1 Benchmarks for the Lisp level implementation

The goal of these benchmarks is to measure the performance improvements obtained by our specialized dense multivariate polynomial domain constructor `MMA` implemented at the Lisp level and described in Section 4.3 of this chapter. We are also curious about measuring the practical benefit of dense recursive polynomial domains in a situation (polynomial GCD computations over algebraic number fields) where AXIOM libraries traditionally use sparse recursive polynomials.

As mentioned in the introduction, our test algorithm is that of van Hoeij and Monagan [50]. Recall that, given an algebraic number field $\mathbb{K} = \mathbb{Q}(a_1, a_2, \ldots, a_e)$, this algorithm computes GCDs in $\mathbb{K}[y]$ by means of a small prime modular algorithm, leading to computations over a tower of simple algebraic extensions $\mathbb{K}_p$ of $\mathbb{Z}/p\mathbb{Z}$. Recall also that the algorithm involves two polynomial data types:

- a multivariate one for the elements of $\mathbb{K}$ and $\mathbb{K}_p$,
- a univariate one for the polynomials in $\mathbb{K}[y]$ and $\mathbb{K}_p[y]$.

Figure 4.1 shows the different combinations that we have used.

$\mathbb{Q}(a_1, a_2, \ldots, a_e)$	$\mathbb{K}[y]$
`NSMP` in SPAD	`SUP` in SPAD
`DMPR` in SPAD	`DUP` in SPAD
`MMA` in LISP	`SUP` in SPAD
`MMA` in LISP	`DUP` in SPAD

Note that:

- the first two combinations, that is `NSMP + SUP` (sparse polynomial domains) and `DMPR + DUP` (dense polynomial domains), involve only SPAD code,
- the other two combinations use MMA - our dense multivariate polynomials implemented at the LISP level and SUP/DUP - univariate polynomials written at the SPAD level.

We would like to stress the following facts:

- the algorithms for addition, multiplication, division of `DRMP` and `MMA` are identical,
- none of the above polynomial types uses fast arithmetic, such as FFT-based or Karatsuba multiplication.

Remember also that:

- the SPAD constructors `NSMP`, `DMPR`, `UP`, and `DUP` are generic constructors, i.e. they work over any AXIOM ring,
- however, our dense multivariate polynomials implemented at the LISP level (provided by the `MMA` constructor) only work over a prime field.

Therefore, we are comparing here is the performances of

- specialized code at the LISP level versus generic code at the SPAD level,
- sparse representation versus dense representation.

We have set the benchmark of van Hoeij and Monagan's algorithm for

- different degrees of the extension $\mathbb{Q} \to \mathbb{K}$,
- different degrees of the input polynomials
- and different sizes for their coefficients.

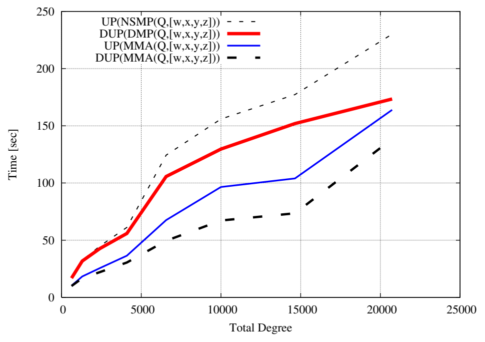
Figure 4.1 p. 55 shows our benchmark results. First, we fix the coefficient size bound to 5 and increase the total degree (degree of the extension plus maximum degree of an input polynomial). The charts (a), (b) and (c) correspond to towers of 3, 4 and 5 simple extensions respectively. Second, we fix the total degree to 2000 and increase the coefficient bound. The charts (d), (e) and (f) correspond to towers of 3, 4 and 5 simple extensions respectively. In (a), (b) and (c) fixing the coefficient size bound, and increase the total degree of input polynomials. Conversely in (d), (e), and (f) fixing the total degree and increase the coefficient size bound. We observe the following facts.

`Charts (a), (b), (c).` For univariate polynomial data types, `DUP` outperforms `SUP` and, for the multivariate polynomial data types, `MMA` outperforms `DRMP`, which outperforms `NSMP`. For the largest degrees, the timing ratio between the best combination, `DUP + MMA`, and the worst one, `SUP + NSMP` is in the range $2 \cdots 3$.

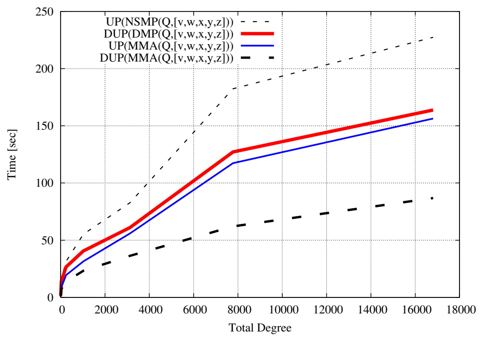
`Charts (d), (e), (f).` The best and the worst combinations are the same as above, however the timing ratio is in the range $3 \cdots 4$. Interestingly, the second best combination is `SUP + MMA` for small coefficients and `DUP + DRMP` for larger ones. This fact maybe explained by following reasons: First, the `SUP` constructor relies on some fast routines which allows it to compete with the `DUP` constructor for small input data. Second, memory allocation and garbage collection of polynomials built with `DUP + DRMP` appears to be more efficient than for `SUP + MMA` polynomials, for large size data.

4.6.2 Benchmarks for the multi-level implementation

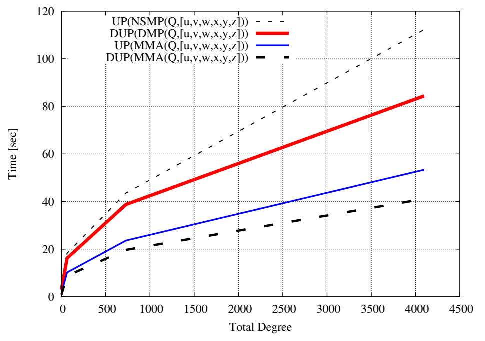
In the previous chapter, we have already demonstrated that our AXIOM fast multivariate multiplication based Kronecker substitution is competitive and often outperforms its counterpart - a similar computer algebra system, namely Magma. This implementation involves code from SPAD, Lisp, C and Assembly level. For the Kronecker substitution part, we write code at SPAD and Lisp levels. However, for the FFT-based univariate multiplication, we write code at C and Assembly level as reported in Section 3.4.2 at Page 36, since the optimized grouping of machine level operations has a huge impact on the performance. As shown in Figures 3.7, this mix-code approach yields high-performance result.



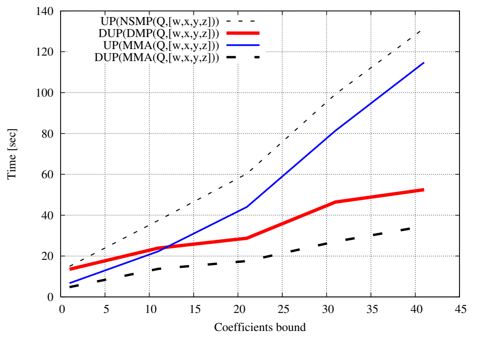
(a)



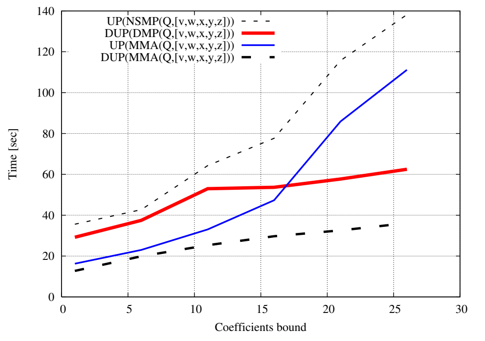
(b)



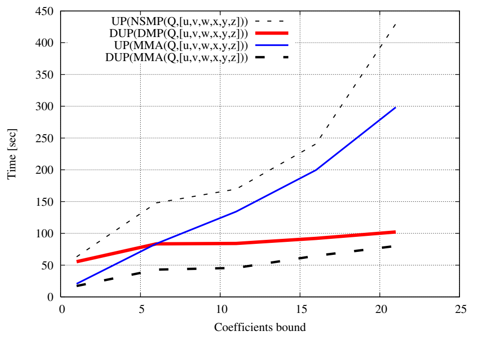
(c)



(d)



(e)



(f)

Figure 4.1: Benchmark of van Hoeij and Monagan's algorithm

4.7 Summary

We have investigated implementation techniques for polynomial arithmetic in the multiple-level programming environment of the AXIOM computer algebra system. Our benchmark results show that careful integration of data structures and code from different levels can improve the performances in a significant manner (a ratio of 2-4 speed up reported in Section 4.6). The integration process requires deep understanding of polynomial arithmetic, machine arithmetic and compiler optimization techniques. However, we believe that it should be implemented in a transparent way for the end-user.

Chapter 5

How Much Can We Speed-up the Existing Library Code in AXIOM with the C Level Implementation?

5.1 Overview

In Chapter 4 we use *van Hoeij and Monagan's modular GCD algorithm* as a benchmark example. By choosing different polynomial data type and implementation technique at different language level, the performance on the same algorithm are obviously different. Based on this experimentation, we believe that the appropriate combination of lower level language implementations are the most efficient way to realize high performance packages for our dense classical and asymptotically fast polynomial applications. However, one problem of this strategy is that the multiple language level implementation is difficult to maintain and has limited portability. Therefore, we try to compress the multiple level code into a single level - the C level. From Lisp level to C level, the code becomes more complex. However, for our application, this step is relatively simple since our Lisp level code doesn't use too much functional language features. We also carefully study the C compiler optimization technique. We try to write highly efficient C code which is close to the performance of our previous Assembly level code. Therefore, similarly to Chapter 4 we need to systematically measure the performance improvement for SPAD level algorithms after supplied with C level support. More precisely, in this chapter our experimentation examples can be formulated as following: given a high-level AXIOM package `P` parametrized by a univariate polynomial domain `U`, we compare the performances of `P` when applied to different U's. One of the Us is our C asymptotically fast polynomial arithmetic implementation wrapped in a SPAD level domain constructor. The rest Us are existing AXIOM domain constructors implemented at SPAD level.

Our experiments show that when P relies our C level fast arithmetic implementation a significant speed-up observed comparing to those relying on other Us. We also compare with other systems. For instance, the square-free factorization in AXIOM with the new support is 7 times faster than the one in MAPLE and very close to the one in MAGMA. Therefore, we believe our asymptotically fast algorithm implementation in C can speed up high level language interfaces generally.

The outline of this chapter is as following. In Section 5.2.1, we review the AXIOM polynomial domain constructors used in our experimentation. In Section 5.2.2 and 5.2.3, we discuss finite field arithmetic and polynomial arithmetic. In Section 5.3, we compare the benchmark results.

`NOTE: This chapter is written based on the published paper [70].`

5.2 Software Overview

5.2.1 AXIOM polynomial domain constructors

In Section 4.2 at Page 47, we have introduced AXIOM univariate and multivariate polynomial domain constructors. In this section, we introduce an new univariate polynomial domain constructor `DUP2(R)` (DenseUnivariatePolynomialDomain version two of the base ring R) `DUP2(R)` is designed for the one whose base ring R is a prime field. In Section 5.2.1 at Page 58 we will review all the related AXIOM constructors, then illustrate `DUP2(R)`. In Section 5.3 at Page 61 we will provide the benchmark between `DUP2(R)` and other constructors. Let `R` be an AXIOM `Ring`. The domain `SUP(R)` implements the ring of univariate polynomials with coefficients in `R`. The data representation of `SUP(R)` is *sparse*: only non-zero terms are encoded. The domain constructor `SUP` is written in the SPAD language.

The domain `DUP(R)` implements exactly the same operations in `SUP(R)`. More precisely, these two domains satisfy the same category `UnivariatePolynomial-Category(R)` (or interface in the sense of Java). However, the representation of the latter domain is *dense*: all terms, null or not, are encoded. The domain constructor `DUP` is also implemented in the SPAD language, see [64] for details.

Another important domain constructor in our study is `PF`: for a prime number `p`, the domain `PF(p)` implements the prime field $\mathbb{Z}/p\mathbb{Z}$. Our C code is dedicated to polynomial computation over $\mathbb{Z}/p\mathbb{Z}$ with dense polynomial representation. To make this code available at the AXIOM level, we have implemented a wrapper domain constructor `DUP2` to wrap up our C code. For a prime number `p`, the domain `DUP2(p)` implements the same category as `DUP(FP(p))` does , i.e. `UnivariatePolynomialCategory(PF(p))`.

5.2.2 Finite field arithmetic

As mentioned in previous chapters, finite field arithmetic is especially important for our modular fast algorithms. Thus, we have put great effort on it. On the one hand, we design more efficient tricks for finite field arithmetic as reported in this section (and later an new trick in Section 6.3 at Page 79), on the other hand we implement these tricks in C code. The C implementation of the new tricks has even better performance comparing to the previous ASSEMBLY level implementation (the one reported in Section 3.4.3 at Page 37). There are two reasons: better arithmetic and highly optimized C code.

In this section, we focus on some *special small* finite fields. By a *small* finite field, we mean a field of the form $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$, for p a prime that fits in a 26-bit word (so that the product of two elements reduced modulo p fits into a `double floating-point` register). Furthermore, the primes p we consider have the form $k2^\ell + 1$, with k a *special small* odd integer (typically $k \leq 7$), which enables us to write specific code for integer Euclidean division. Although this is a trick for *special* prime numbers, it is good enough for the most of our polynomial applications where we have the freedom to choose prime numbers.

The elements of $\mathbb{Z}/p\mathbb{Z}$ are represented by integers from 0 to $p-1$. Additions and subtractions in $\mathbb{Z}/p\mathbb{Z}$ are performed in a straightforward way: we perform integer operations, and the result is then reduced modulo p. Since the result of additions and subtractions is always in $-(p-1), \dots, 2(p-1)$, modular reduction requires at most a single addition or subtraction of p; for the reduction, we use routines from Shoup's NTL library [6, 86]. Multiplication in $\mathbb{Z}/p\mathbb{Z}$ requires more work. A popular solution implemented in NTL conducts a multiplication in `double` precision floating-point registers, computes numerically the quotient appearing in the Euclidean division by p, and finally deduces the remainder.

Using the special form of the prime p, we have designed the following faster "approximate" Euclidean division, that shares similarities with Montgomery's REDC algorithm [74]; for another use of arithmetic modulo special primes, see [37]. Let thus Z be in $0, \ldots, (p-1)^2$; in actual computations, Z is obtained as the product of two integers less than p. The following algorithm computes an approximation of the remainder of kZ by p, where we recall that p has the form $k2^\ell + 1$:

1. Compute $q = \lfloor \frac{Z}{2^\ell} \rfloor$.
2. Compute $r = k(Z - q2^\ell) - q$.

Proposition 5.2.1. *Let r be as above and let $r_0 < p$ be the remainder of kZ by p. Then $r \equiv r_0 \bmod p$ and $r = r_0 - \delta p$, with $0 \le \delta < k+1$.*

PROOF. Let us write the Euclidean division of kZ by p as $kZ = q_0 p + r_0$. This implies that

$$q = q_0 + \left\lfloor \frac{q_0 + r_0}{k2^\ell} \right\rfloor$$

holds. From the equality $qp + r = q_0 p + r_0$, we deduce that we have

$$r = r_0 - \delta p \quad \text{with} \quad \delta = \left\lfloor \frac{q_0 + r_0}{k2^\ell} \right\rfloor p.$$

The assumption $Z \le (p-1)^2$ enables us to conclude that $\delta < k+1$ holds. $\square$

In terms of operations, this reduction is faster than the usual algorithms which rely on either Montgomery's REDC or Shoup's floating-point techniques. The computation of q is done by a logical `shift`; that of r requires a logical `and` (to obtain $Z - 2^\ell q$), and a single multiplication by the constant c. Classical reduction algorithms involve 2 multiplications, and other operations (additions and logical operations). Accordingly, in practical terms, our approach turns out to be the most efficient one.

There are however drawbacks to this approach. First, the algorithm above does not compute $Z \bmod p$, but a number congruent to kZ modulo p (this multiplication by a constant is also present in Montgomery's approach). This is however easy to circumvent in several cases, for instance when doing multiplications by precomputed constants (this is the case in FFT polynomial multiplication, see below), since a correcting factor $k^{-1} \bmod p$ can be incorporated in these constants. The second drawback is that the output of our reduction routine is not reduced modulo p. When results are reused in several computations, errors accumulate, so it is necessary to perform some error reduction regularly which is an overhead.

In Section 6.3 at Page 79, we extend this special prime reduction trick into a more generic method. The trick presented in this section has approximation steps in the middle stage, and obtain the exact result in the end after removing the "errors". The more generic trick will maintain all intermediate results exact and it works for all Fourier prime numbers.

5.2.3 Polynomial arithmetic

In Section 2.1 at Page 8 we have introduced a set of FFT-based algorithms. In this section, we briefly review the fast division and Half-GCD algorithms. We use these two as benchmark programs in Section 5.3 of at Page 61.

Our implementation of fast Euclidean division is based on Cook-Sieveking-Kung's approach [43, Chapter 9]. One of the major steps in this approach is to compute Newton's iteration. We have implemented Newton's iteration with the support of *middle product technique* [49]. This technique can reduce the cost of a direct implementation by a constant factor. For the GCD computation, we have implemented both the classical Euclidean algorithm and the faster Half-GCD techniques [43, Chapter 11]. The classical one has complexity in $O(d^2)$, whereas the latter one is in $O(d \log(d)^2)$ with a large multiplicative constant factor.

5.2.4 Code connection

Recall that in Section 3.2.2 at Page 29, we have described the way to integrate multiple level code in AXIOM. Actually, the crucial step is converting different polynomial data representations between AXIOM and the ones in our C library via GCL level. The overhead of these conversions may significantly reduce the effectiveness of our C implementation. Thus, good understanding of data structures in AXIOM and GCL is a necessity to establish an efficient code connection.

5.3 Experimentation

In this section, we compare our specialized domain constructor `DUP2` with our generic domain constructor `DUP` and the existing (default) AXIOM domain constructor `SUP`. Our experimental computations are in the polynomial rings:

- $A_p = \mathbb{Z}/p\mathbb{Z}[x]$,
- $B_p = (\mathbb{Z}/p\mathbb{Z}[x]/\langle m \rangle)[y]$,

for a machine word prime number p and an irreducible polynomial $m \in \mathbb{Z}/p\mathbb{Z}[x]$. The ring A_p can be implemented by any of the three domain constructors `DUP2`, `DUP` and `SUP` applied to `PF(p)`, whereas B_p is implemented by either `DUP` and `SUP` applied to A_p. In both A_p and B_p, we compare the performances of factorization and resultant computation. We have two goals for this experimentation:

(G_1) When a large portion of the running time spends on computing products, remainders, quotients, GCDs in A_p, we believe that there are opportunities for significant speed-up when using DUP2 and we want to measure this speed-up w.r.t. SUP and DUP.

(G_2) Otherwise, when there is a little portion spends on computing products, remainders, quotients, GCDs in A_p, we want to check whether using DUP2 is still better than using SUP and DUP.

For computing univariate polynomial resultants over a field, AXIOM calls the package PseudoRemainderSequence (using the algorithms of Ducos [32]). This package takes R: IntegralDomain and polR: UnivariatePolynomialCategory(R) as parameters. However, this code has its private divide operation and does not rely on the one provided by the domain polR. In fact, the only non-trivial operation will be used from polR is polynomial addition. Therefore, the package PseudoRemainderSequence does not take advantage of our fast division even it's available. Hence, for this example, there is very little speedup when using DUP2 instead SUP and DUP.

For square-free factorization over a finite field, AXIOM calls the package UnivariatePolynomialSquareFree. It takes RC: IntegralDomain and P: UnivariatePolynomialCategory(RC) as parameters. In this case, the code relies on the operations gcd and exquo provided by P. Hence, if P provides fast GCD computations and fast divisions, UnivariatePolynomialSquareFree can use them. In this case, DUP2 does help.

We start the description of our experimental results with resultant computations in $A_p = \mathbb{Z}/p\mathbb{Z}[x]$. As mentioned above, this is not a good example for significant performance improvement. Figure 5.1 shows that computations with DUP2 are just slightly faster than those with SUP. In fact, it is satisfactory to verify that using DUP2, which implies data-type conversions between the AXIOM and C data-structures, does not slow down computations.

We continue with square-free factorization and irreducible factorization in A_p. Figure 5.2 (resp. Figure 5.3) shows that DUP2 provides a speed-up ratio of 8 (resp. 7) for polynomial with degrees about 9000 (resp. 400). This shows that the combination of the fast arithmetic (FFT-based multiplication, Fast division, Half-GCD) and highly optimized code from DUP2 does help.

In the case of irreducible factorization, we could have obtained a better ratio if the code was *more generic*. Indeed, the irreducible factorization over finite fields in AXIOM involves a package which has its private univariate polynomial arithmetic, leading to a problem similar to that observed with resultant computations. The package in question is `ModMonic`, parametrized by `R: Ring` and `Rep: UnivariatePolynomialCategory(R)`, which implements the Frobenius map.

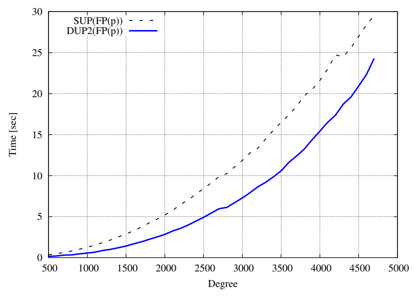


Figure 5.1: Resultant computation in $\mathbb{Z}/p\mathbb{Z}[x]$

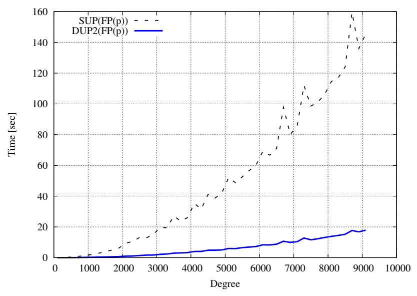


Figure 5.2: Square-free factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

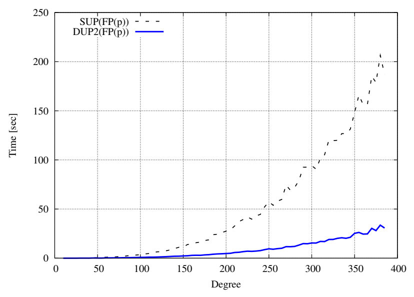


Figure 5.3: Irreducible factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

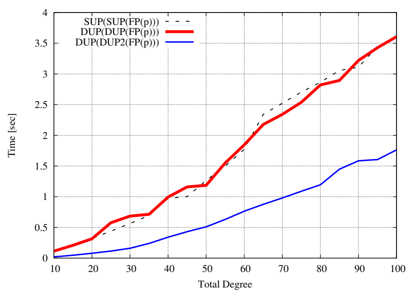


Figure 5.4: Resultant computation in $(\mathbb{Z}/p\mathbb{Z}[x]/\langle m\rangle)[y]$

We conclude this section with our benchmarks in $B_p = (\mathbb{Z}/p\mathbb{Z}[x]/\langle m\rangle)[y]$. For resultant computations in B_p the speed-up ratio obtained with `DUP2` is better than in the case of A_p. This is because the arithmetic operations of `DUP2` (addition, multiplication, inversion) perform better than those of `SUP` or `DUP`. Finally, for irreducible factorization in B_p, the results are quite surprising. Indeed, AXIOM uses Trager's algorithm (which reduces computations to resultants in B_p, irreducible factorization in A_p and GCDs in B_p) and, based on our previous results, we could have anticipated a good speed-up ratio. Unfortunately, the package `AlgFactor`, which is used for algebraic factorization, has its private arithmetic. More precisely, it "re-defines" B_p with `SUP` and factorizes the input polynomial over this new B_p. Therefore, there is no impressive speed-up at all.

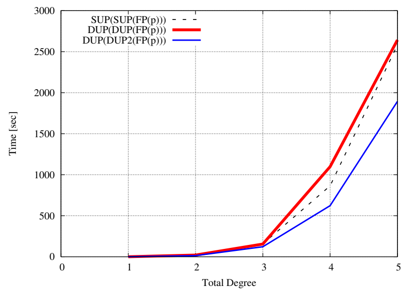


Figure 5.5: Irreducible factorization in $(\mathbb{Z}/p\mathbb{Z}[x]/\langle m \rangle)[y]$

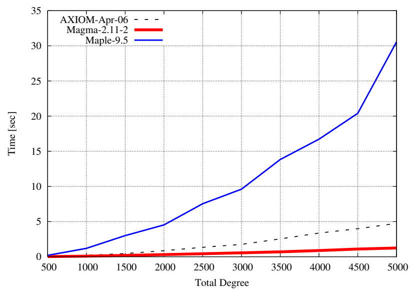


Figure 5.6: Square-free factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

5.4 Summary

The purpose of this Chapter is to measure the impact of our C level specialized implementation for fast polynomial arithmetic on the performances of AXIOM high-level algorithms. Generic programming is well designed in the AXIOM system. The experimental results demonstrate that by replacing a few important operations in `DUP(PF(p))` with our C level implementation, the original AXIOM univariate polynomial arithmetic over $\mathbb{Z}/p\mathbb{Z}$ has been sped up by a large factor in general. For algorithm such as univariate polynomial square free factorization over $\mathbb{Z}/p\mathbb{Z}$, the improved AXIOM code is 7 times faster than the one in MAPLE and very close to the one in MAGMA (see Figure 5.6 at Page 64).

Chapter 6

Fast Arithmetic for Triangular Sets: from Theory to Practice

6.1 Overview

In Chapters 3, 4 and 5 we have presented implementation techniques for both asymptotically fast and classical polynomial arithmetic. From this chapter to Chapter 8 we focus on developing faster algorithms with comparing to the best known algorithms in terms of complexity for triangular decompositions technique (see Section 2.3 at Page 20).

For each new algorithm, we have realized a high performance implementation based on our previous techniques. The benchmark result for each implementation will be reported at the end of each chapter. As a starting point of our new algorithms development, we study those "core" operations. In this chapter, we have identified, improved and implemented one of these operations: *modular multiplication*. Indeed, all triangular decomposition technique based methods involve polynomial arithmetic operations (addition, subtraction, multiplication and division) modulo a triangular set. We call them modular operations. As we explained in this chapter, modular multiplication and division are expensive (often dominant) operations in terms of computational time in triangular decompositions based polynomial solving. Under certain assumptions, the modular division can be achieved by two modular multiplications as described in this chapter. Thus, *modular multiplication* is unarguably the "core" operation and at the base level to support all triangular decompositions based algorithms such as *Regular GCD*, *Bivariate Solver*, *regularity test* reported in

Chapters 7 and 8. In the following text, we will give an overview of the concepts of triangular set, modular multiplication, and our contributions in this chapter.

Triangular sets. Triangular set is an useful data structure for dealing with a variety of problems, from computations with algebraic numbers to the symbolic solution of polynomial or differential systems. At the core of the algorithms for these objects, one finds a few basic operations, such as multiplication and division in dimension zero. Higher-level algorithms can be built on these subroutines, using for instance modular algorithms and lifting techniques [27]. The zero-dimensional case is discussed in detail (with worked-out examples) in [62]; a general introduction (including positive dimensional situations) is given in Section 2.3 at Page 20 (also see [9]). In this chapter, we adopt the following convention: a *monic triangular set* is a family of polynomials $T = (T_1, \ldots, T_n)$ in $R[X_1, \ldots, X_n]$, where R is a commutative ring with 1. For all i, we impose that T_i is in $R[X_1, \ldots, X_i]$, is *monic* in X_i and *reduced* with respect to $T_1, \ldots, T_{i-1}$. Since all our triangular sets will be monic, we simply call them triangular sets.

The natural approach to arithmetic modulo triangular sets is recursive: to work in the residue class ring $\mathbb{L}_T = R[X_1, \ldots, X_n]/\langle T_1, \ldots, T_n\rangle$, we regard it as $\mathbb{L}_{T_-}[X_n]/\langle T_n\rangle$, where $\mathbb{L}_{T_-}$ is the ring $R[X_1, \ldots, X_{n-1}]/\langle T_1, \ldots, T_{n-1}\rangle$. This point of view allows one to design elegant recursive algorithms, whose complexity is often easy to analyze, and which can be implemented in a straightforward manner in high-level languages such as AXIOM or MAPLE [63]. However, as shown below, this approach is not necessarily optimal, regarding both complexity and practical performance.

Complexity issues. The core of our problematic is *modular multiplication*: given A and B in the residue class ring $\mathbb{L}_T$, compute their product; here, one assumes that the input and output are reduced with respect to the polynomials T. Besides, one can safely suppose that all degrees are at least 2 (see the discussion in the next section).

In one variable, the usual approach consists in multiplying A and B and reducing them by Euclidean division. Using classical arithmetic, the cost is about $2d_1^2$ multiplications and $2d_1^2$ additions in R, with $d_1 = \deg(T_1, X_1)$. Using fast arithmetic, polynomial multiplication becomes essentially linear, the best known result ([21], after [83, 82]) being of the form $\mathsf{k}\, d_1 \lg(d_1) \lg\lg(d_1)$, with k a constant and $\lg(x) = \log_2 \max(2, x)$. A Euclidean division can then be reduced to two polynomial multiplications, using Cook-Sieveking-Kung's algorithm [24, 87, 59]. In n variables, the measure of complexity is $\delta_T = \deg(T_1, X_1) \cdots \deg(T_n, X_n)$, since representing a polynomial modulo T requires storing δ_T elements. Then, applying the previous results recursively leads to bounds of order $2^n \delta_T^2$ for the standard approach, and $(3\mathsf{k}+1)^n \delta_T$ for the fast one, neglecting logarithmic factors and lower-order terms. An important difference with the univariate case is the presence of the overheads 2^n and $(3\mathsf{k}+1)^n$, which cannot be absorbed in a big-Oh estimate anymore (unless n is bounded).

Improved algorithms and the advantages of fast arithmetic. Our first contribution is the design and implementation of a faster algorithm: while still relying on the techniques of fast Euclidean division, we show in Theorem 6.2.1 that a mixed dense / recursive approach yields a cost of order $4^n \delta_T$, neglecting again all lower order terms and logarithmic factors; this is better than the previous bound for $\delta_T \geq 2^n$. Building upon previous work [41], the implementation is done in C, and is dedicated to small finite field arithmetic.

The algorithm uses fast polynomial multiplication and Euclidean division. For univariate polynomials over $\mathbb{F}_p$, such fast algorithms become advantageous for degrees of approximately 100. In a worst-case scenario, this may suggest that for multivariate polynomials, fast algorithms become useful when the partial degree in each variable is at least 100, which would be a severe restriction. Our second contribution is to contradict this expectation, by showing that the cut-off values for which the fast algorithm becomes advantageous *decrease* with the number of variables.

A quasi-linear algorithm for a special case. We next discuss a particular case, where all polynomials in the triangular set are actually univariate, that is, with T_i in $\mathbb{K}[X_i]$ for all i. Despite its apparent simplicity, this problem already contains non-trivial questions, such as power series multiplication modulo $\langle X_1^{d_1}, \ldots, X_n^{d_n} \rangle$, taking $T_i = X_i^{d_i}$. For the question of power series multiplication, no quasi-linear algorithm was known until [85]. We extend this result to the case of arbitrary $T_i \in \mathbb{K}[X_i]$, the crucial question being how to *avoid* expanding the (polynomial) product AB before reducing it. Precisely, we prove that for $\mathbb{K}$ of cardinality greater than, or equal to, $\max_{i \leq n} d_i$, and for $\varepsilon > 0$, there exists a constant K_ε such that for all n, products modulo $\langle T_1(X_1), \ldots, T_n(X_n) \rangle$ can be done in at most $\mathsf{K}_\varepsilon \delta_T^{1+\varepsilon}$ operations, with δ_T as before.

Following [13, 14, 12, 85], the algorithm uses deformation techniques, and is unfortunately not expected to be very practical, except for example. when all degrees equal 2. However, this shows that for a substantial family of examples, and in suitable (large enough) fields, one can suppress the exponential overhead seen above. Generalizing this result to an arbitrary T is a major open problem.

Applications to higher-level algorithms. Fast arithmetic for basic operations modulo a triangular set is fundamental for a variety of higher-level operations. By embedding fast arithmetic in high-level environments like AXIOM (see [41, 65]) or MAPLE, one can obtain a substantial speed-up for questions ranging from computations with algebraic numbers (GCD, factorization) to polynomial system solving via triangular decomposition, such as in the algorithm of [75], which is implemented in AXIOM and MAPLE [63].

Our last contribution is to demonstrate such a speed-up on the example of van Hoeij and Monagan's algorithm for GCD computation over number fields. This algorithm is modular, most of the effort consisting in GCD computations over small finite fields. We compare a direct AXIOM implementation to one relying on our low-level C implementation, and obtain improvement of orders of magnitude.

Outline of this chapter. Section 6.2 presents our multiplication algorithms, for general triangular sets and triangular sets consisting of univariate polynomials. We next describe our implementation in Section 6.3; experiments and comparisons with other systems are given in Section 6.4.

`NOTE: This chapter is written based on the published paper [69].`

6.2 Algorithms

We describe here our main algorithm. It relies on the Cook-Sieveking-Kung idea but differs from a direct recursive implementation: recalling that we handle multivariate polynomials makes it possible to base our algorithm on fast multivariate multiplication.

6.2.1 Notation and preliminaries

Notation. Triangular sets will be written as $T = (T_1, \ldots, T_n)$. The multi-degree of a triangular set T is the n-tuple $d_i = \deg(T_i, X_i)_{1 \le i \le n}$. We will write $\delta_T = d_1 \cdots d_n$; in Subsection 6.2.3 at Page 73, we will use the notation $r_T = \sum_{i=1}^{n}(d_i - 1) + 1$. Writing $\mathbf{X} = X_1, \ldots, X_n$, we let $\mathbb{L}_T$ be the residue class ring $R[\mathbf{X}]/\langle T \rangle$, where R is our base ring. Let M_T be the set of monomials $M_T = \left\{ X_1^{e_1} \cdots X_n^{e_n} \mid 0 \le e_i < d_i \text{ for all } i \right\}$; then, because of our monicity assumption, the free R-submodule generated by M_T in $R[\mathbf{X}]$, written

$$\mathsf{Span}(M_T) = \Big\{ \sum_{m \in M_T} a_m m \mid a_m \in R \Big\},$$

is isomorphic to $\mathbb{L}_T$. Hence, in our algorithms, elements of $\mathbb{L}_T$ are represented on the monomial basis M_T. Without loss of generality, we always assume that *all degrees d_i are at least 2*. Indeed, if T_i has degree 1 in X_i, the variable X_i appears neither in the monomial basis M_T nor in the other polynomials T_j, so one can express it as a function of the other variables, and T_i can be discarded.

Standard and fast modular multiplication. As said before, standard algorithms have a cost of roughly $2^n \delta_T^2$ operations in R for multiplication in $\mathbb{L}_T$. This bound seems not even polynomial in δ_T, due to the exponential overhead in n. However, since all degrees d_i are at least 2, δ_T is at least 2^n; hence, any bound of the form $\mathsf{K}^n \delta_T^\ell$ is actually polynomial in δ_T, since it is upper-bounded by $\delta_T^{\log_2(\mathsf{K})+\ell}$.

Our goal is to obtain bounds of the form $\mathsf{K}^n \delta_T$ (up to logarithmic factors), that are thus softly linear in δ_T for fixed n; of course, we want the constant K as small as possible. We will use fast polynomial multiplication, denoting by $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ a function such that over any ring, polynomials of degree less than d can be multiplied in $\mathsf{M}(d)$ operations, and which satisfies the super-linearity conditions of [43, Chapter 8]. Using the algorithm of Cantor-Kaltofen [21], one can take $\mathsf{M}(d) \in O(d \log(d) \log\log(d))$. Precisely, we will denote by k a constant such that $\mathsf{M}(d) \le \mathsf{k}\, d \lg(d) \lg\lg(d)$ holds for all d, with $\lg(d) = \log_2 \max(d, 2)$

In one variable, fast modular multiplication is done using the Cook-Sieveking-Kung algorithm [24, 87, 59]. Given T_1 monic of degree d_1 in $R[X_1]$ and A, B of degrees less than d_1, one computes first the product AB. To perform the Euclidean division $AB = QT_1 + C$, one first computes the inverse $S_1 = U_1^{-1} \bmod X_1^{d_1-1}$, where $U_1 = X_1^{d_1} T_1(1/X_1)$ is the reciprocal polynomial of T_1. This is done using Newton iteration, and can be performed as a precomputation, for a cost of $3\mathsf{M}(d_1) + O(d_1)$. One recovers first the reciprocal of Q, then the remainder C, using two polynomial products. Taking into account the cost of computing AB, but leaving out precomputations, these operations have cost $3\mathsf{M}(d_1) + d_1$. Applying this result recursively leads to a rough upper bound of $\prod_{i \le n}(3\mathsf{M}(d_i) + d_i)$ for a product in $\mathbb{L}_T$, without taking into account the similar cost of precomputation (see [60] for similar considerations); this gives a total estimate of roughly $(3\mathsf{k}+1)^n \delta_T$, neglecting logarithmic factors.

One can reduce the $(3\mathsf{k}+1)^n$ overhead: since additions and constant multiplications in $\mathbb{L}_T$ can be done in linear time, it is the *bilinear* cost of univariate multiplication which governs the overall cost. Over a field of large enough cardinality, using evaluation / interpolation techniques, univariate multiplication in degree less than d can be done using $2d - 1$ bilinear multiplications; this yields estimates of rough order $(3 \times 2)^n \delta_T = 6^n \delta_T$. Studying more precisely the multiplication process, we prove in Theorem 6.2.1 that one can compute products in $\mathbb{L}_T$ using at most $\mathsf{K}\,4^n \delta_T \lg(\delta_T) \lg\lg(\delta_T)$ operations, for an universal constant K. This is a synthetic but rough upper bound; we give more precise estimates within the proof. Obtaining results linear in δ_T, without an exponential factor in n, is a major open problem. When the base ring is a field of large enough cardinality, we obtain first results in this direction in Theorem 6.2.2: in the case of families of *univariate* polynomials, we present an algorithm of quasi-linear complexity $\mathsf{K}_\varepsilon \delta_T^{1+\varepsilon}$ for all ε.

Basic complexity considerations. Since we are estimating costs that depend on an *a priori* unbounded number of parameters, big-Oh notation is delicate to handle. We rather use explicit inequalities when possible, all the more as an explicit control is required in the proof of Theorem 6.2.2. For similar reasons, we do not use $O\tilde{}$ notation.

We denote by $\mathsf{C}_{\mathsf{Eval}}$ (resp. $\mathsf{C}_{\mathsf{Interp}}$) functions such that over any ring R, a polynomial of degree less than d can be evaluated (resp. interpolated) at d points $a_0, \dots, a_{d-1}$ in $\mathsf{C}_{\mathsf{Eval}}(d)$ (resp. $\mathsf{C}_{\mathsf{Interp}}(d)$) operations, assuming $a_i - a_j$ is a unit for $i \neq j$ for interpolation. From [43, Chapter 10], we can take both quantities in $O(\mathsf{M}(d)\lg(d))$, where the constant in the big-Oh is universal. In Subsection 6.2.3, we will assume without loss of generality that $\mathsf{M}(d) \le \mathsf{C}_{\mathsf{Eval}}(d)$ for all d.

Recall that k is such that $\mathsf{M}(d)$ is bounded by $\mathsf{k}\,d\lg(d)\lg\lg(d)$ for all d. Up to maybe increasing k, we will thus assume that both $\mathsf{C}_{\mathsf{Eval}}(d)$ and $\mathsf{C}_{\mathsf{Interp}}(d)$ are bounded by $\mathsf{k}\,d\lg^2(d)\lg\lg(d)$. Finally, we let $\mathsf{MM}(d_1, \dots, d_n)$ be such that over any ring R, polynomials in $R[X_1, \dots, X_n]$ of degree in X_i less than d_i for all i can be multiplied in $\mathsf{MM}(d_1, \dots, d_n)$ operations. One can take

$$\mathsf{MM}(d_1, \dots, d_n) \le \mathsf{M}((2d_1 - 1) \cdots (2d_n - 1))$$

using Kronecker's substitution. Let $\delta = d_1 \cdots d_n$. Assuming $d_i \ge 2$ for all i, we deduce the inequalities

$$(2d_1 - 1) \cdots (2d_n - 1) \le 2^n \delta \le \delta^2,$$

which imply that $\mathsf{MM}(d_1, \dots, d_n)$ admits the upper bound

$$\mathsf{k}2^n \delta \lg(2^n\delta) \lg\lg(2^n\delta) \;\le\; 4\mathsf{k}2^n \delta \lg(\delta) \lg\lg(\delta).$$

Up to replacing k by $4\mathsf{k}$, we thus have

$$\delta \leq \mathsf{MM}(d_1, \ldots, d_n) \leq \mathsf{k}\, 2^n \delta \lg(\delta) \lg\lg(\delta). \tag{6.1}$$

Pan [78] proposed an alternative algorithm, that requires the existence of interpolation points in the base ring. This algorithm is more efficient when for example d_i are fixed and $n \to \infty$. However, using it below would not bring any improvement, due to our simplifications.

6.2.2 The main algorithm

Theorem 6.2.1. *There exists a constant* K *such that the following holds. Let* R *be a ring and let* T *be a triangular set in* $R[\mathbf{X}]$*. Given* A, B *in* $\mathbb{L}_T$*, one can compute* $AB \in \mathbb{L}_T$ *in at most* $\mathsf{K}\, 4^n \delta_T \lg(\delta_T) \lg\lg(\delta_T)$ *operations* $(+, \times)$ *in* R*.*

Proof. Let $T = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$ in $R[\mathbf{X}] = R[X_1, \ldots, X_n]$. We then introduce the following objects:

- We write $T_- = (T_1, \ldots, T_{n-1})$, so that $\mathbb{L}_{T_-} = R[X_1, \ldots, X_{n-1}]/\langle T_- \rangle$.
- For $i \leq n$, the polynomial $U_i = X_i^{d_i} T_i(X_1, \ldots, X_{i-1}, 1/X_i)$ is the reciprocal polynomial of T_i; S_i is the inverse of U_i modulo $\langle T_1, \ldots, T_{i-1}, X_i^{d_i-1} \rangle$. We write $\mathbf{S} = (\mathbf{S_1}, \ldots, \mathbf{S_n})$ and $\mathbf{S}_- = (\mathbf{S_1}, \ldots, \mathbf{S_{n-1}})$.

Two subroutines are used, which we describe in Figure 5. In these subroutines, we use the following notation:

- For D in $R[X_1, \ldots, X_i]$ such that $\deg(D, X_i) \leq e$, $\mathsf{Rev}(D, X_i, e)$ is the reciprocal polynomial $X_i^e D(X_1, \ldots, X_{i-1}, 1/X_i)$.
- For D in $R[\mathbf{X}]$, $\mathsf{Coeff}(D, X_i, e)$ is the coefficient of X_i^e.

We can now give the specification of these auxiliary algorithms. These algorithms make some assumptions, that will be satisfied when we call them from our main routine.

- The first one is $\mathsf{Rem}(A, T, \mathbf{S})$, with A in $R[\mathbf{X}]$. This algorithm computes the normal form of A modulo T, assuming that $\deg(A, X_i) \leq 2d_i - 2$ holds for all i. When $n = 0$, A is in R, T is empty and $\mathsf{Rem}(A, T, \mathbf{S}) = \mathbf{A}$.

- The next subroutine is $\mathsf{MulTrunc}(A, B, T, \mathbf{S}, \mathbf{d_{n+1}})$, with A, B in $R[\mathbf{X}, X_{n+1}]$; it computes the product AB modulo $\langle T, X_{n+1}^{d_{n+1}}\rangle$, assuming that $\deg(A, X_i)$ and $\deg(B, X_i)$ are bounded by $d_i - 1$ for $i \le n+1$. If $n = 0$, T is empty, so this function return $AB \bmod X_1^{d_1}$.

To compute $\mathsf{Rem}(A, T, \mathbf{S})$, we use the Cook-Sieveking-Kung idea in $\mathbb{L}_{T_-}[X_n]$: we reduce all coefficients of A modulo T_- and perform two truncated products in $\mathbb{L}_{T_-}[X_n]$ using $\mathsf{MulTrunc}$. The operation $\mathsf{MulTrunc}$ is performed by multiplying A and B as polynomials, truncating in X_{n+1} and reducing all coefficients modulo T, using Rem.

Algorithm 5 Modular Reduction

```
Rem($A, T, \mathbf{S}$)
1 if $n = 0$ return $A$
2 $A' \leftarrow \sum_{i=0}^{2d_n-2} \mathsf{Rem}(\mathsf{Coeff}(A, X_n, i), T_-, \mathbf{S}_-)\mathbf{X_n^i}$
3 $B \leftarrow \mathsf{Rev}(A', X_n, 2d_n - 2) \bmod X_n^{d_n-1}$
4 $P \leftarrow \mathsf{MulTrunc}(B, S_n, T_-, \mathbf{S}_-, \mathbf{d_n - 1})$
5 $Q \leftarrow \mathsf{Rev}(P, X_n, d_n - 2)$
6 return $A' \bmod X_n^{d_n} - \mathsf{MulTrunc}(Q, T_n, T_-, \mathbf{S}_-, \mathbf{d_n})$

MulTrunc($A, B, T, \mathbf{S}, \mathbf{d_{n+1}}$)
1 $C \leftarrow AB$
2 if $n = 0$ return $C \bmod X_1^{d_1}$
3 return $\sum_{i=0}^{d_{n+1}-1} \mathsf{Rem}(\mathsf{Coeff}(C, X_{n+1}, i), T, \mathbf{S})\mathbf{X_{n+1}^i}$
```

For the complexity analysis, assuming for a start that all inverses $\mathbf{S}$ have been precomputed, we write $\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n)$ for an upper bound on the cost of $\mathsf{Rem}(A, T, \mathbf{S})$ and $\mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_{n+1})$ for a bound on the cost of $\mathsf{MulTrunc}(A, B, T, \mathbf{S}, \mathbf{d_{n+1}})$. Setting $\mathsf{C}_{\mathsf{Rem}}(\,) = 0$, the previous algorithms imply the estimates

$$\begin{aligned}\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) &\le (2d_n - 1)\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1}) + \mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n - 1)\\ &\quad + \mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n) + d_1 \cdots d_n;\\ \mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n) &\le \mathsf{MM}(d_1, \ldots, d_n) + \mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1})d_n.\end{aligned}$$

Assuming that $\mathsf{C}_{\mathsf{MulTrunc}}$ is non-decreasing in each d_i, we deduce the upper bound

$$\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) \le 4\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1})d_n + 2\mathsf{MM}(d_1, \ldots, d_n) + d_1 \cdots d_n,$$

for $n \geq 1$. Write $\mathsf{MM}'(d_1, \dots, d_n) = 2\mathsf{MM}(d_1, \dots, d_n) + d_1 \cdots d_n$. This yields

$$\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_n) \;\leq\; \sum_{i=1}^{n} 4^{n-i}\, \mathsf{MM}'(d_1, \dots, d_i) d_{i+1} \cdots d_n,$$

since $\mathsf{C}_{\mathsf{Rem}}(\,) = 0$. In view of the bound on MM given in Equation (6.1), we obtain

$$\mathsf{MM}'(d_1, \dots, d_i) d_{i+1} \cdots d_n \;\leq\; 3\mathsf{k}2^i \delta_T \lg(\delta_T) \lg\lg(\delta_T).$$

Taking e.g. $\mathsf{K} = 3\mathsf{k}$ gives the bound $\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_n) \leq \mathsf{K}\, 4^n \delta_T \lg(\delta_T) \lg\lg(\delta_T)$. The product $A, B \mapsto AB$ in $\mathbb{L}_T$ is performed by multiplying A and B as polynomials and returning $\mathsf{Rem}(AB, T, \mathbf{S})$. Hence, the cost of this operation admits a similar bound, up to replacing K by $\mathsf{K} + \mathsf{k}$. This concludes our cost analysis, excluding the cost of the precomputations. We now estimate the cost of precomputing the inverses $\mathbf{S}$: supposing that $S_1, \dots, S_{n-1}$ are known, we detail the cost of computing S_n. Our upper bound on $\mathsf{C}_{\mathsf{MulTrunc}}$ shows that, assuming $S_1, \dots, S_{n-1}$ are known, one multiplication modulo $X_n^{d'_n}$ in $\mathbb{L}_{T_-}[X_n]$ can be performed in

$$\mathsf{k}2^n \delta' \lg(\delta') \lg\lg(\delta') + \mathsf{K}\, 4^n \delta' \lg(\delta_{T_-}) \lg\lg(\delta_{T_-})$$

operations, with $\delta_{T_-} = d_1 \cdots d_{n-1}$ and $\delta' = \delta_{T_-} d'_n$. Up to replacing K by $\mathsf{K} + \mathsf{k}$, and assuming $d'_n \leq d_n$, this yields the upper bound $\mathsf{K}\, 4^n \delta' \lg(\delta_T) \lg\lg(\delta_T)$. Let now $\ell = \lceil \log_2(d_n - 1) \rceil$. Using Newton iteration in $\mathbb{L}_{T_-}[X_n]$, we obtain S_n by performing 2 multiplications in $\mathbb{L}_{T_-}[X_n]$ in degrees less than m and $m/2$ negations, for $m = 2, 4, \dots, 2^{\ell-1}$, see [43, Chapter 9]. By the remark above, the cost is at most

$$\mathsf{t}(n) \;=\; \sum_{m=2,\dots,2^{\ell-1}} 3\mathsf{K}\, 4^n d_1 \cdots d_{n-1} m \lg(\delta_T) \lg\lg(\delta_T) \;\leq\; 3\mathsf{K}\, 4^n \delta_T \lg(\delta_T) \lg\lg(\delta_T).$$

The sum $\mathsf{t}(1) + \cdots + \mathsf{t}(n)$ bounds the total precomputation time; one sees that it admits a similar form of upper bound. Up to increasing K, this gives the desired result. $\square$

6.2.3 The case of univariate polynomials

To suppress the exponential overhead, it is necessary to avoid expanding the product AB. We discuss here the case of triangular sets consisting of univariate polynomials, where this is possible. We provide a quasi-linear algorithm, that works under mild assumptions. However, the techniques used (deformation ideas, coming from fast matrix multiplication algorithms [13, 14, 12]) induce large sub-linear factors.

Theorem 6.2.2. *For any $\varepsilon > 0$, there exists a constant K_ε such that the following holds. Let $\mathbb{K}$ be a field and $T = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$ in $\mathbb{K}[X_1] \times \cdots \times \mathbb{K}[X_n]$, with $2 \le d_i \le |\mathbb{K}|$ for all i. Given A, B in $\mathbb{L}_T$, one can compute $AB \in \mathbb{L}_T$ using at most $\mathsf{K}_\varepsilon\, \delta_T^{1+\varepsilon}$ operations $(+, \times, \div)$ in $\mathbb{K}$.*

Step 1. We start by a special case. Let $T = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$; for later applications, we suppose that it has coefficients in a ring R. Our main assumption is that for all i, T_i is in $R[X_i]$ and factors as

$$T_i = (X_i - \alpha_{i,0}) \cdots (X_i - \alpha_{i,d_i-1}),$$

with $\alpha_{i,j} - \alpha_{i,j'}$ a unit in R for $j \ne j$'. Let $V \subset R^n$ be the grid

$$V = [\ (\alpha_{1,\ell_1}, \ldots, \alpha_{n,\ell_n}) \mid 0 \le \ell_i < d_i\ ],$$

which is the zero-set of $(T_1, \ldots, T_n)$ (when the base ring is a domain). Remark that T_i and T_j can have non-trivial common factors: all that matters is that for a given i, evaluation and interpolation at the roots of T_i is possible.

Proposition 6.2.3. *Given A, B in $\mathbb{L}_T$, as well as the set of points V, one can compute $AB \in \mathbb{L}_T$ using at most*

$$\delta_T \left(1 + \sum_{i \le n} \frac{2\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}\right)$$

operations $(+, \times, \div)$ in R.

In view of our remarks on the costs of evaluation and interpolation, this latter cost is at most $\mathsf{K}'\, \delta_T \lg^2(\delta_T) \lg\lg(\delta_T)$, for an universal constant K', which can be taken as $\mathsf{K}' = 3\mathsf{k} + 1$.

Proof. The proof uses an evaluation / interpolation process. Define the evaluation map

$$\begin{array}{rccc} \mathsf{Eval} : & \mathsf{Span}(M_T) & \to & R^{\delta_T} \\ & F & \mapsto & [F(\alpha) \mid \alpha \in V]. \end{array}$$

Since all $\alpha_{i,j} - \alpha_{i,j'}$ are units, the map Eval is invertible. To perform evaluation and interpolation, we use the algorithm in [78, Section 2], which generalizes the multidimensional Fourier transform: to evaluate F, we see it as a polynomial in $\mathbb{K}[X_1, \ldots, X_{n-1}][X_n]$, and evaluate recursively its coefficients at $V' = [(\alpha_{1,\ell_1}, \ldots, \alpha_{n-1,\ell_{n-1}}) \mid 0 \le \ell_i < d_i]$. We conclude by performing $d_1 \cdots d_{n-1}$ univariate evaluations in X_n in degree d_n.

Extending our previous notation, we immediately deduce the recursion for the cost $\mathsf{C}_{\mathsf{Eval}}$ of multivariate evaluation

$$\mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_n) \le \mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_{n-1})\, d_n + d_1 \cdots d_{n-1} \mathsf{C}_{\mathsf{Eval}}(d_n),$$

$$\text{so that} \quad \mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_n) \;\le\; \delta_T \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i)}{d_i}.$$

The inverse map of Eval is the interpolation map Interp. Again, we use Pan's algorithm; the recursion and the bounds for the cost are the same, yielding

$$\mathsf{C}_{\mathsf{Interp}}(d_1, \ldots, d_n) \;\le\; \delta_T \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}.$$

To compute $AB \bmod T$, it suffices to evaluate A and B on V, multiply the δ_T pairs of values thus obtained, and interpolate the result. The cost estimate follows. □

This algorithm is summarized in Figure 6, under the name MulSplit (since it refers to triangular sets which completely split into linear factors).

Algorithm 6 MulSplit

```
MulSplit($A, B, V$)
1 $\mathsf{Val}_A \leftarrow \mathsf{Eval}(A)$
2 $\mathsf{Val}_B \leftarrow \mathsf{Eval}(B)$
3 $\mathsf{Val}_C \leftarrow [\ \mathsf{Val}_A(\alpha)\mathsf{Val}_B(\alpha) \mid \alpha \in V\ ]$
4 return $\mathsf{Interp}(\mathsf{Val}_C)$
```

Step 2. We continue with the case where the polynomials T_i do not split anymore. Recall our definition of the integer $r_T = \sum_{i=1}^{n}(d_i - 1) + 1$; since the polynomials T form a Gröbner basis for any order, r_T is the regularity of the ideal $\langle T \rangle$. In the following, the previous exponential overhead disappears, but we introduce a quasi-linear dependency in r_T: these bounds are good for triangular sets made of many polynomials of low degree.

Proposition 6.2.4. *Under the assumptions of Theorem 6.2.2, given A, B in $\mathbb{L}_T$, one can compute the product $AB \in \mathbb{L}_T$ using at most*

$$\mathsf{k}' \ \delta_T \ \mathsf{M}(r_T) \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i},$$

operations $(+, \times, \div)$ in $\mathbb{K}$, for an universal constant k'.

As before, there exists an universal constant K'' such that this estimate simplifies as

$$\mathsf{K}'' \, \delta_T \, r_T \left(\lg(\delta_T) \lg(r_T) \right)^3. \tag{6.2}$$

Proof. Let $T = (T_1, \ldots, T_n)$ be a triangular set with T_i in $\mathbb{K}[X_i]$ of degree d_i for all i. Let $\mathsf{U} = (U_1, \ldots, U_n)$ be the polynomials

$$U_i = (X_i - a_{i,0}) \cdots (X_i - a_{i,d_i-1}),$$

where for fixed i, the values $a_{i,j}$ are pairwise distinct (these values exist due to our assumption on the cardinality of $\mathbb{K}$). Let finally η be a new variable, and define $V^0 = (V_1, \ldots, V_n) \subset \mathbb{K}[\eta][\mathbf{X}]$ by $V_i = \eta T_i + (1 - \eta) U_i$, so that V_i is monic of degree d_i in $\mathbb{K}[\eta][X_i]$. Remark that the monomial bases M_T, M_{U} and M_V^0 are all the same, that specializing η at 1 in V^0 yields T and that specializing η at 0 in V^0 yields U.

Lemma 6.2.5. *Let A, B be in $\mathsf{Span}(M_T)$ in $\mathbb{K}[\mathbf{X}]$ and let $C = AB \bmod \langle V^0 \rangle$ in $\mathbb{K}[\eta][\mathbf{X}]$. Then C has degree in η at most $r_T - 1$, and $C(1, \mathbf{X})$ equals AB modulo $\langle T \rangle$.*

Proof. Fix an arbitrary order on the elements of M_T, and let $\mathsf{Mat}(X_i, V^0)$ and $\mathsf{Mat}(X_i, T)$ be the multiplication matrices of X_i modulo respectively $\langle V^0 \rangle$ and $\langle T \rangle$ in this basis. Hence, $\mathsf{Mat}(X_i, V^0)$ has entries in $\mathbb{K}[\eta]$ of degree at most 1, and $\mathsf{Mat}(X_i, T)$ has entries in $\mathbb{K}$. Besides, specializing η at 1 in $\mathsf{Mat}(X_i, V^0)$ yields $\mathsf{Mat}(X_i, T)$. The coordinates of $C = AB \bmod \langle V^0 \rangle$ on the basis M_T are obtained by multiplying the coordinates of B by the matrix $\mathsf{Mat}(A, V^0)$ of multiplication by A modulo $\langle V^0 \rangle$. This matrix equals $A(\mathsf{Mat}(X_1, V^0), \ldots, \mathsf{Mat}(X_n, V^0))$; hence, specializing its entries at 1 gives the matrix $\mathsf{Mat}(A, T)$, proving our last assertion. To conclude, observe that since A has total degree at most $r_T - 1$, the entries of $\mathsf{Mat}(A, V^0)$ have degree at most $r_T - 1$ as well. □

Let R be the ring $\mathbb{K}[\eta]/\langle \eta^{r_T} \rangle$ and let A, B be in $\mathsf{Span}(M_T)$ in $\mathbb{K}[\mathbf{X}]$. Define $C_\eta = AB \bmod \langle V^0 \rangle$ in $R[\mathbf{X}]$ and let C be its canonical preimage in $\mathbb{K}[\eta][\mathbf{X}]$. By the previous lemma, $C(1, \mathbf{X})$ equals $AB \bmod \langle T \rangle$. To compute C_η, we will use the evaluation /

interpolation techniques of Step 1, as the following lemma shows that the polynomials V^0 split in $R[\mathbf{X}]$. The corresponding algorithm is in Figure 7; it uses a Newton-Hensel lifting algorithm, called Lift, whose last argument indicates the target precision.

Lemma 6.2.6. *Let i be in $\{1, \dots, n\}$. Given $a_{i,0}, \dots, a_{i,d_i-1}$ and T_i, one can compute $\alpha_{i,0}, \dots, \alpha_{i,d_i-1}$ in R^{d_i}, with $\alpha_{i,j} - \alpha_{i,j'}$ invertible for $j \neq j'$, and such that*

$$V_i = (X_i - \alpha_{i,0}) \cdots (X_i - \alpha_{i,d_i-1})$$

holds in $R[X_i]$, using $O(\mathsf{M}(r_T)\mathsf{C}_{\mathsf{Eval}}(d_i))$ operations in $\mathbb{K}$. The constant in the big-Oh estimate is universal.

Proof. As shown in [17, Section 5], the cost of computing U_i from its roots is $\mathsf{C}_{\mathsf{Eval}}(d_i) + O(\mathsf{M}(d_i))$, which is in $O(\mathsf{C}_{\mathsf{Eval}}(d_i))$ by our assumption on $\mathsf{C}_{\mathsf{Eval}}$; from this, one deduces V_i with $O(d_i)$ operations. The polynomial $U_i = V_i(0, X_i)$ splits into a product of linear terms in $\mathbb{K}[X_i]$, with no repeated root, so V_i splits into $R[X_i]$, by Hensel's lemma. The power series roots $\alpha_{i,j}$ are computed by applying Newton-Hensel lifting to the constants $a_{i,j}$, for $j = 0, \dots, d_i - 1$. Each lifting step then boils down to evaluate the polynomial V_i and its derivative on the current d_i-tuple of approximate solutions and deducing the required correction. Hence, as in [43, Chapter 15], the total cost is $O(\mathsf{M}(r_T)\mathsf{C}_{\mathsf{Eval}}(d_i))$ operations; one easily checks that the constant hidden in this big-Oh is universal. $\square$

Algorithm 7 Lift Roots

```
LiftRoots($a_{i,0}, \dots, a_{i,d_i-1}, T_i$)
1 $U_i \leftarrow (X_i - a_{i,0}) \cdots (X_i - a_{i,d_i-1})$
2 $V_i \leftarrow \eta T_i + (1 - \eta) U_i$
3 return Lift($a_{i,0}, \dots, a_{i,d_i-1}, V_i, \eta^{r_T}$)
```

We can finally prove Proposition 6.2.4. To compute $AB \bmod \langle T \rangle$, we compute $C_\eta = AB \bmod \langle V^0 \rangle$ in $R[\mathbf{X}]$, deduce $C \in \mathbb{K}[\eta][\mathbf{X}]$ and evaluate it at 1. By the previous lemma, we can use Proposition 6.2.3 over the coefficient ring R to compute C_η. An operation $(+, \times, \div)$ in R has cost $O(\mathsf{M}(r_T))$. Taking into account the costs of Step 1 and Lemma 6.2.6, one sees that there exists a constant k' such that the cost is bounded by

$$\mathsf{k}' \; \delta_T \; \mathsf{M}(r_T) \sum_{i \leq n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}.$$

□

The algorithm is given in Figure 8, under the name MulUnivariate; we use a function called Choose($\mathbb{K}, d$), which returns d pairwise distinct elements from $\mathbb{K}$.

Algorithm 8 MulUnivariate

```
MulUnivariate($A, B, T$)
1 for $i = 1, \ldots, n$ do
1.1   $a_{i,0}, \ldots, a_{i,d_i-1} \leftarrow$ Choose($\mathbb{K}, d_i$)
1.2   $\alpha_{i,0}, \ldots, \alpha_{i,d_i-1} \leftarrow$ LiftRoots($a_{i,0}, \ldots, a_{i,d_i-1}, T_i$)
2 $V \leftarrow [\ (\alpha_{1,\ell_1}, \ldots, \alpha_{n,\ell_n}) \mid 0 \le \ell_i < d_i\ ]$
3 $C_\eta \leftarrow$ MulSplit($A, B, V$)    (computations done mod $\eta^{r_T}$)
4 return $C_\eta(1, \mathbf{X})$               ($C_\eta$ is seen in $\mathbb{K}[\eta][\mathbf{X}]$)
```

Step 3: conclusion. To prove Theorem 6.2.2, we combine the previous two approaches (the general case and the deformation approach), using the former for large degrees and the latter for smaller ones. Let ε be a positive real, and define $\omega = 2/\varepsilon$. We can assume that the degrees in T are ordered as $2 \le d_1 \cdots \le d_n$, with in particular $\delta_T \ge 2^n$. Define an index ℓ by the condition that $d_\ell \le 4^\omega \le d_{\ell+1}$, taking $d_0 = 0$ and $d_{n+1} = \infty$ for definiteness, and let

$$T' = (T_1, \ldots, T_\ell) \quad \text{and} \quad T'' = (T_{\ell+1}, \ldots, T_n).$$

Then the quotient $\mathbb{L}_T$ equals $R[X_{\ell+1}, \ldots, X_n]/\langle T'' \rangle$, with $R = \mathbb{K}[X_1, \ldots, X_\ell]/\langle T' \rangle$. By Equation (6.2), a product in R can be done in $\mathsf{K}''\, \delta_{T'}\, r_{T'}\, \big(\lg(\delta_{T'})\lg(r_{T'})\big)^3$ operations in $\mathbb{K}$; additions are cheaper, since they can be done in time $\delta_{T'}$. By Theorem 6.2.1, one multiplication in $\mathbb{L}_T$ can be done in $\mathsf{K}\, 4^{n-\ell}\delta_{T''}\lg(\delta_{T''})\lg\lg(\delta_{T''})$ operations in R. Hence, taking into account that $\delta_T = \delta_{T'}\delta_{T''}$, the total cost for one operation in $\mathbb{L}_T$ can be roughly upper-bounded by

$$\mathsf{K}\,\mathsf{K}''\, 4^{n-\ell}\, \delta_T\, r_{T'}\, \big(\lg(\delta_{T'})\lg(r_{T'})\lg(\delta_{T''})\big)^3.$$

Now, observe that $r_{T'}$ is upper-bounded by $d_\ell n \le 4^\omega \lg(\delta_T)$. This implies that the factor

$$r_{T'}\big(\lg(\delta_{T'})\lg(r_{T'})\lg(\delta_{T''})\big)^3$$

is bounded by $\mathsf{H}\lg^{10}(\delta_T)$, for a constant H depending on ε. Next, $(4^{n-\ell})^\omega = (4^\omega)^{n-\ell}$ is bounded by $d_{\ell+1}\cdots d_n \le \delta_T$. Raising to the power $\varepsilon/2$ yields $4^{n-\ell} \le \delta_T^{\varepsilon/2}$; thus, the previous estimate admits the upper bounds

$$\mathsf{K}\,\mathsf{K}''\,\mathsf{H}\,\delta_T^{1+\varepsilon/2}\lg^{10}(\delta_T) \le \mathsf{K}\,\mathsf{K}''\,\mathsf{H}\,\mathsf{H}'\delta_T^{1+\varepsilon},$$

where H' depends on ε.

6.3 Implementation Techniques

The previous algorithms were implemented in C; most efforts were devoted to the generic algorithm of Section 6.2.2. As in Chapter 5 (or see Papers [41, 65]), the C code was interfaced with AXIOM. In this section, we describe this implementation.

Arithmetic in $\mathbb{F}_p$. Our implementation is devoted to small finite fields $\mathbb{F}_p$, with p a machine word prime of the form $c2^n + 1$, for $c < 2^n$. Multiplications in $\mathbb{F}_p$ are done using Montgomery's REDC routine [74]. A straightforward implementation does not bring better performance than the floating point techniques of Shoup [86]. We use an improved scheme, adapted to our special primes, presented below. Compared to a direct implementation of Montgomery's algorithm, it lowers the operation count by 2 double word shifts and 2 single word shifts. This approach performs better on our experimentation platform (Pentium 4) than Shoup's implementation, the gain being of 32%. It is also more efficient and more portable than the one in [41], which explicitly relied on special machine features like SSE registers of late IA-32 architectures. We formally describe this scheme as following:
Let p be a prime of the form $p = c2^n + 1$, for $c < 2^n$ (in our code, n ranges from 20 to 23 and c is less than 1000). Let $\ell = \lceil \log_2(p) \rceil$ and let $R = 2^\ell$. Given a and ω, both reduced modulo p, Montgomery's REDC algorithm computes $a\omega/R \bmod p$. We present our tailor-made version here. Precomputations will be authorized for the argument ω (this is not a limitation for our main application, FFT polynomial multiplication). We compute

1. $M_1 = a\omega$
2. $(q_1, r_1) = (M_1 \text{ div } R, M_1 \text{ mod } R)$
3. $M_2 = r_1 c 2^n$
4. $(q_2, r_2) = (M_2 \text{ div } R, M_2 \text{ mod } R)$
5. $M_3 = r_2 c 2^n$

6. $q_3 = M_3 \text{ div } R$

7. $A = q_1 - q_2 + q_3$.

Proposition 6.3.1. *Suppose that $c < 2^n$. Then A satisfies $A \equiv a\omega/R \bmod p$ and $-(p-1) < A < 2(p-1)$.*

Proof. By construction, we have the equalities $Rq_1 = M_1 - r_1$ and $Rq_2 = M_2 - r_2$. Remark next that 2^n divides M_2, and thus r_2 (since R is a power of two larger than 2^n). It follows that 2^{2n} divides M_3. Since we have $c < 2^n$, p is at most 2^{2n}, so R is at most 2^{2n} as well. Hence, R divides M_3, so that $Rq_3 = M_3$. Putting this together yields

$$RA = M_1 - r_1 - M_2 + r_2 + M_3.$$

Recall that $M_2 = r_1 c 2^n$, so that $M_2 = -r_1 \bmod p$. Similarly, $M_3 = r_2 c 2^n$, so $M_3 = -r_2 \bmod p$. Hence, $RA = M_1 \bmod p$, which means that $A = a\omega/R \bmod p$, as claimed. As to the bounds on A, we start by remarking that $M_1 < (p-1)^2$, so that $q_1 < p - 1$. Next, since $r_1 < R$, we deduce that $M_2 < c2^n R$ which implies that $q_2 < c2^n = p - 1$. Similarly, we obtain that $q_3 < p - 1$, which implies the requested inequalities. □

Let us now describe our implementation on 32-bit x86 processors. We use an assembly macro `MulHiLo`(a, b) from the GMP library; this macro computes the product d of two word-length integers a and b and puts the high part of the result ($d \text{ div } 2^{32}$) in the register `AX` and the lower part ($d \bmod 2^{32}$) in the register `DX`, avoiding shifts. In our case, R does not equal 2^{32}. However, since we allow precomputations on ω, we will actually store and use $\omega' = 2^{32-\ell}\omega$ instead of ω; hence, `MulHiLo`(a, ω') directly gives us q_1 and $r'_1 = 2^{32-\ell} r_1$. Similarly, we do not compute the product $r_1 c 2^n$; instead, we use `MulHiLo`(r'_1, c'), where c' is the precomputed constant $c2^n$, to get q_2 and $r'_2 = 2^{32-\ell} r_2$.

To compute q_3, it turned out to be better to do as follows. We write q_3 as $r_2 c / 2^{\ell - n}$. Now, recall from the proof of the previous proposition that 2^n divides r_2. Under the assumption that $c < 2^n$, we saw in the proof that $\ell \leq 2n$, so that $2^{\ell-n}$ divides r_2. Hence, we obtain q_3 by right-shifting r_2 by $\ell - n$ places, or, equivalently, r'_2 by $32 - n$ places, and multiplying the result by c. Eventually, we need to bring the result A between 0 and $p - 1$. As in NTL [86], we avoid `if` statements: using the sign bit of A as a mask, one can add p to A in the case $A < 0$; by subtracting p and correcting once more, we obtain the correct remainder.

In the following benchmark we compare our specialized trick versus the standard Montgomery trick when applying them into a FFT computation over a 32 bit FFT prime number. The specialized trick outperforms the standard one. Note that our specialized trick utilizes an assembly subroutine for multiplying machine integers whereas the standard one is implemented in pure C language.

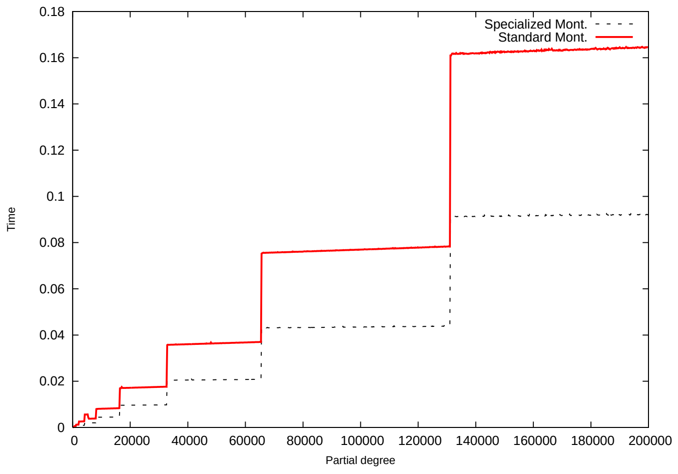


Figure 6.1: TFT vs. FFT.

Arithmetic in $\mathbb{F}_p[X]$. Univariate polynomial arithmetic is crucial: multiplication modulo a triangular set boils down to multivariate polynomial multiplications, which can then be reduced to univariate multiplications through Kronecker's substitution. We use classical and FFT multiplication for univariate polynomials over $\mathbb{F}_p$. We use two FFT multiplication routines: the first one is that from [26]; its implementation is essentially the one described in [41], up to a few modifications to improve cache-friendliness. The second one is van der Hoeven's TFT (Truncated Fourier Transform) [51], which is less straightforward but can perform better for transform sizes that are not powers of 2. We tried several data accessing patterns; the most suitable solution is platform-dependent, since cache size, associativity properties and register sets have huge impact. Going further in that direction will require automatic code tuning techniques, as in [54, 53, 79].

Multivariate arithmetic over $\mathbb{F}_p$. We use a dense representation for multivariate polynomials: important applications of modular multiplication (GCD computations, Hensel lifting for triangular sets) tend to produce dense polynomials. We use multi-dimensional arrays (encoded as a contiguous memory block of machine integers) to represent our polynomials, where the size in each dimension is bounded by the corresponding degree $\deg(T_i, X_i)$, or twice that much for intermediate products. Multivariate arithmetic is done using either Kronecker's substitution as in [41] or standard multidimensional FFT. While the two approaches share similarities, they do not access data in the same manner. In our experiments, multidimensional FFT performed better by 10-15% for bivariate cases, but was slower for larger number of variables with small FFT size in each dimension.

Triangular sets over $\mathbb{F}_p$. Triangular sets are represented in C by an array of multivariate polynomials. For the algorithm of Subsection 6.2.3, we only implemented the case where all degrees are 2; this mostly boils down to evaluation and interpolation on n-dimensional grids of size 2^n, over a power series coefficient ring.

More work was devoted to the algorithm of Subsection 6.2.2. Two strategies for modular multiplication were implemented, a plain one and that of Subsection 6.2.2. Both first perform a multivariate multiplication then do a multivariate reduction; the plain reduction method performs a recursive Euclidean division, while the faster one implements both algorithms Rem and MulTrunc of Subsection 6.2.2. Remark in particular that even the plain approach is not the entirely naive, as it uses fast multivariate multiplication for the initial multiplication. Both approaches are recursive, which makes it possible to interleave them. At each level $i = n, \ldots, 1$, a cut-off point decides whether to use the plain or fast algorithm for multiplication modulo $\langle T_1, \ldots, T_i \rangle$. These cut-offs are experimentally determined: as showed in Section 6.4, they are surprisingly low for $i > 1$.

The fast algorithm uses precomputations (of the power series inverses of the reciprocals of the polynomials T_i). In practice, it is of course better to store and reuse these elements: in situations such as GCD computation or Hensel lifting, we expect to do several multiplications modulo the same triangular set. We could push further these precomputations, by storing Fourier transforms; this is not done yet.

GCD's. One of the first applications of fast modular multiplication is GCD computation modulo a triangular set, which itself is central to higher-level algorithms for solving systems of equations. Hence, we implemented a preliminary version of such GCD computations using a plain recursive version of Euclid's algorithm. This implementation has not been thoroughly optimized. In particular, we have not incorporated any half-GCD technique, except for *univariate* GCD's; this univariate half-GCD is far from optimal.

The AXIOM level. Integrating our fast arithmetic into AXIOM is straightforward, after dealing with the following two problems. First, AXIOM is a Lisp-based system, whereas our package is implemented in C. Second, in AXIOM, dense multivariate polynomials are represented by recursive trees, but in our C package, they are encoded as multidimensional arrays. Both problems are solved by modifying the GCL kernel. For the first issue, we integrate our C package into the GCL kernel, so that our C-level functions from can be used by AXIOM at run-time. For the second problem, we realized a tree / array polynomial data converter. This converter is also linked to GCL kernel; the conversations, happening at run-time, have negligible cost.

6.4 Experimental Results

The main part of this section describes experimental results attached to our main algorithm of Subsection 6.2.2; we discuss the algorithm of Subsection 6.2.3 in the last paragraphs. For the entire set of benchmarks, we use random dense polynomials. Our experiments were done on a 2.80 GHz Pentium 4 PC, with 1GB memory and 1024 KB cache.

6.4.1 Comparing different strategies

We start by experiments comparing different strategies for computing products modulo triangular sets in $n = 1, 2, 3$ variables, using our general algorithm.

Strategies. Let $\mathbb{L}_0 = \mathbb{F}_p$ be a small prime field and let $\mathbb{L}_n$ be $\mathbb{L}_0[X_1, \ldots, X_n]/\langle T \rangle$, with T a n-variate triangular set of multi-degree $(d_1, \ldots, d_n)$. To compute a product $C = AB \in \mathbb{L}_n$, we first expand $P = AB \in \mathbb{L}_0[\mathbf{X}]$, then reduce it modulo T. The product P is always computed by the same method; we use three strategies for computing C.

- Plain. We use univariate Euclidean division; computations are done recursively in $\mathbb{L}_{i-1}[X_i]$ for $i = n, \ldots, 1$.
- Fast, using precomputations. We apply the algorithm $\mathsf{Rem}(C, T, \mathbf{S})$ of Algorithm 5, assuming that the inverses $\mathbf{S}$ have been precomputed.
- Fast, without precomputations. We apply the algorithm $\mathsf{Rem}(C, T, \mathbf{S})$ of Algorithm 5, but recompute the required inverses on the fly.

Our ultimate goal is to obtain a highly efficient implementation of the multiplication in $\mathbb{L}_n$. To do so, we want to compare our strategies in $\mathbb{L}_1$, $\mathbb{L}_2$, ..., $\mathbb{L}_n$. In this report we give details for $n \leq 3$ and leave for future work the case of $n > 3$, as the driving idea is to tune our implementation in $\mathbb{L}_i$ before investigating that of $\mathbb{L}_{i+1}$. This approach leads to determine cut-offs between our different strategies. The alternative is between PLAIN and FAST strategies, depending on the assumption regarding precomputations. For applications discussed before (quasi-inverses, polynomial GCDs modulo a triangular set), using precomputations is realistic.

Univariate multiplication. Figure 6.2 compares our implementation of the Truncated Fourier Transform (TFT) multiplication to the classical Fast Fourier Transform (FFT). Because the algorithm is more complex, especially the interpolation phase, the TFT approach does not outperform the classical FFT multiplication in all cases.

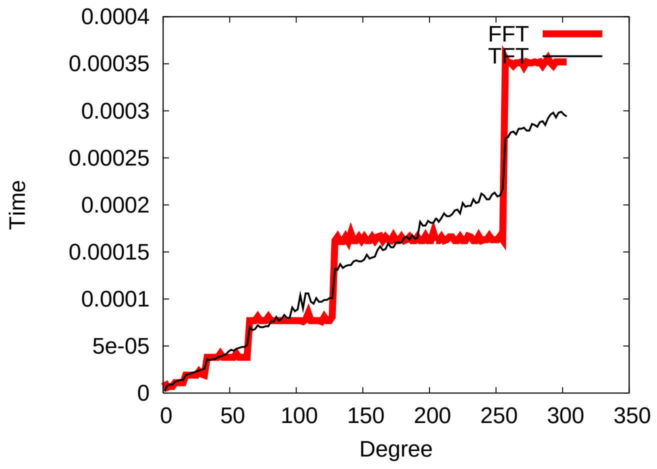


Figure 6.2: TFT vs. FFT.

Univariate triangular sets. Finding the cut-offs between our strategies is straightforward. Figure 6.3 shows the result using classical FFT multiplication; the cut-off point is about 150. If precomputations are not assumed, then this cut-off doubles. Using Truncated Fourier Transform, one obtains roughly similar results.

Bivariate triangular sets. For $n = 2$, we let in Figure 6.4 d_1 and d_2 vary in the ranges $4, \ldots, 304$ and $2, \ldots, 102$. This allows us to determine a cut-off for d_2 as a function of d_1. Surprisingly, this cut-off is essentially independent of d_1 and can be chosen equal to 5. We discuss this point below. To continue our benchmarks in $\mathbb{L}_3$, we would like the product $d_1 d_2$ to play the role in $\mathbb{L}_3$ that d_1 did in $\mathbb{L}_2$, so as to determine the cut-off for d_3 as a function of $d_1 d_2$. This leads to the question: for a *fixed* product $d_1 d_2$, does the running time of the multiplication in $\mathbb{L}_2$ stay constant

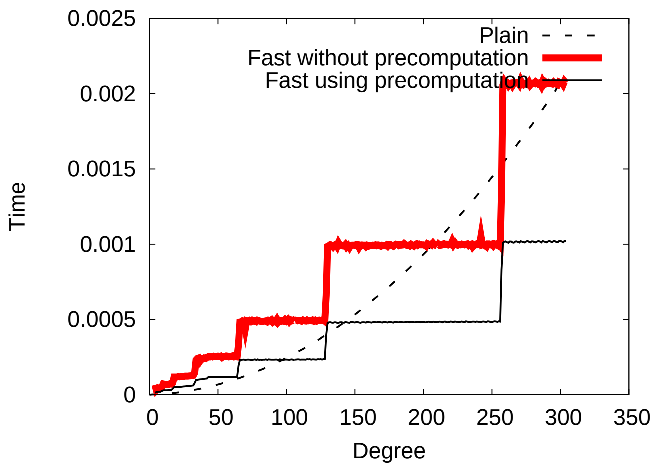


Figure 6.3: Multiplication in $\mathbb{L}_1$, all strategies, using FFT multiplication.

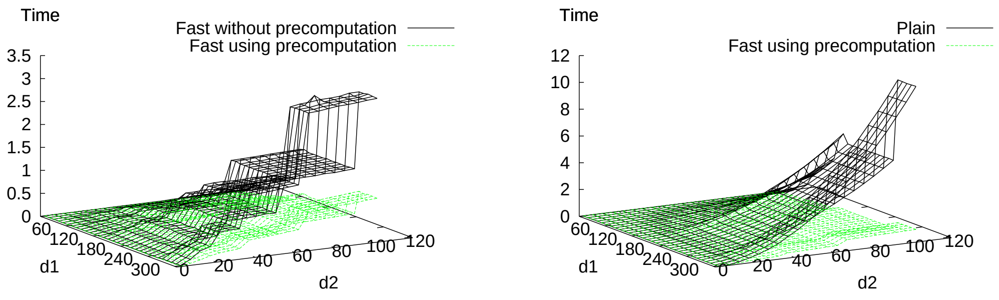


Figure 6.4: Multiplication in $\mathbb{L}_2$, FAST without precomputations vs. FAST using precomputations (top) and PLAIN vs. FAST using precomputations.

when (d_1, d_2) varies in the region $4 \leq d_1 \leq 304$ and $2 \leq d_2 \leq 102$? Figure 6.5 gives timings obtained for this sample set; it shows that the time varies mostly for the PLAIN strategy (the levels in the FAST case are due to our FFT multiplication). These results guided our experiments in $\mathbb{L}_3$.

Trivariate triangular sets. For our experiments with $\mathbb{L}_3$, we consider three patterns for (d_1, d_2). Pattern 1 has $d_1 = 2$, Pattern 2 has $d_1 = d_2$ and Pattern 3 has $d_2 = 2$. Then, we let $d_1 d_2$ vary from 4 to 304 and d_3 from 2 to 102. For simplicity, we also report only the comparison between the strategies PLAIN and FAST USING PRECOMPUTATIONS. The timings are in Figure 6.6; they show an impressive speed-up for the FAST strategy. We also observe that the cut-off between the two strategies can be set to 3 for each of the patterns. Experiments as in Figure 6.5 gives similar conclusion: the timing depends not only on $d_1 d_2$ and d_3 but also on the ratios between these degrees.

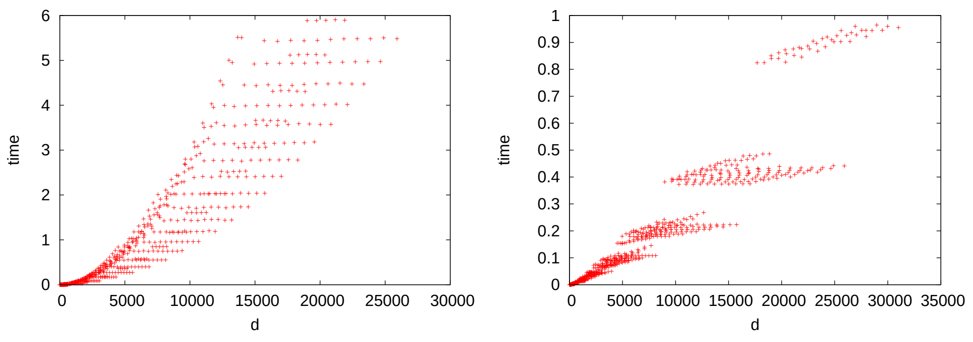


Figure 6.5: Multiplication in $\mathbb{L}_2$, time vs. $d = d_1 d_2$, Plain (left) and Fast using precomputations (right).

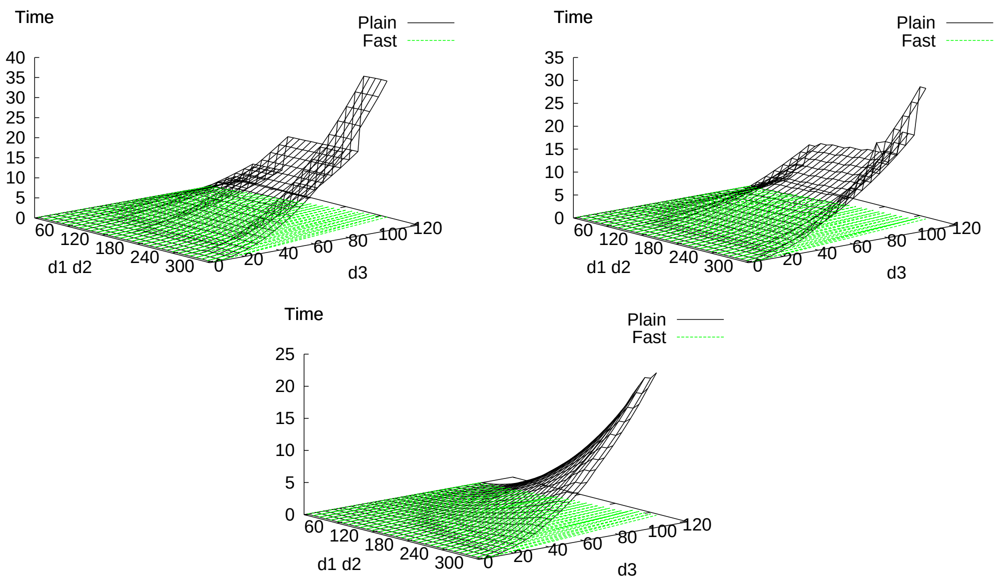


Figure 6.6: Multiplication in $\mathbb{L}_3$, PLAIN vs. FAST, patterns 1–3 from top left to bottom.

Discussion of the cut-offs. To understand the low cut-off points we observe, we have a closer look at the costs of several strategies for multiplication in $\mathbb{L}_2$. For a ring R, classical polynomial multiplication in $R[X]$ in degree less than d uses about (d^2, d^2) operations $(\times, +)$ respectively (we omit linear terms in d). Euclidean division of a polynomial of degree $2d-2$ by a monic polynomial T of degree d has essentially the same cost. Hence, classical modular multiplication uses about $(2d^2, 2d^2)$ operations $(\times, +)$ in R. Additions modulo $\langle T \rangle$ take d operations.

Thus, a pure recursive approach for multiplication in $\mathbb{L}_2$ uses about $(4d_1^2 d_2^2, 4d_1^2 d_2^2)$ operations $(\times, +)$ in $\mathbb{K}$. Our PLAIN approach is less naive. We first perform a bivariate product in degrees (d_1, d_2). Then, we reduce all coefficients modulo $\langle T_1 \rangle$ and perform

Euclidean division in $\mathbb{L}_1[X_2]$, for a cost of about $(2d_1^2d_2^2, 2d_1^2d_2^2)$ operations. Hence, we can already make some advantage of fast FFT-based multiplication, since we traded $2d_1^2d_2^2$ base ring multiplications and as many additions for a bivariate product.

Using precomputations, the FAST approach performs 3 bivariate products in degrees about (d_1, d_2) and about $4d_2$ reductions modulo $\langle T_1 \rangle$. Even for moderate (d_1, d_2) such as in the range 20–30, bivariate products can already be done efficiently by FFT multiplication, for a cost much inferior to $d_1^2d_2^2$. Then, *even if reductions modulo $\langle T_1 \rangle$ are done by the* PLAIN *algorithm*, our approach performs better: the total cost of these reductions will be about $(4d_1^2d_2, 4d_1^2d_2)$, so we save a factor $\simeq d_2/2$ on them. This explains why we observe very low cut-offs in favor of the FAST algorithm.

6.4.2 Comparing implementations

Comparison with Magma. To evaluate the quality of our implementation of modular multiplication, we compared it with MAGMA v. 2-11 [16], which has set a standard of efficient implementation of low-level algorithms. We compared multiplication in $\mathbb{L}_3$ for the previous three patterns, in the same degree ranges. Figure 6.7 gives the timings for Pattern 3. The MAGMA code uses iterated `quo` constructs over `UnivariatePolynomial`'s, which was the most efficient configuration we found. For our code, we use the strategy PLAIN USING PRECOMPUTATIONS. On this example, our code outperforms MAGMA by factors up to 7.4; other patterns yield similar behavior.

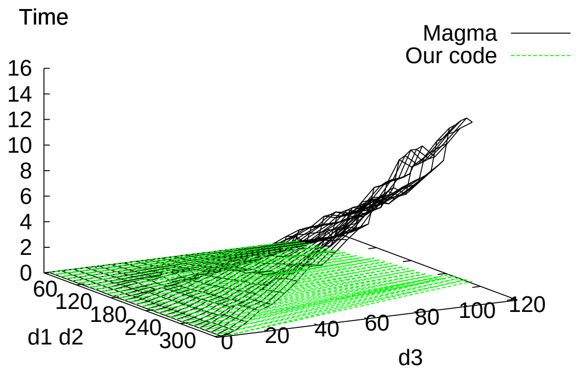


Figure 6.7: Multiplication in $\mathbb{L}_3$, pattern 3, Magma vs. our code.

Comparison with Maple. Our future goal is to obtain high-performance implementations of higher-level algorithms in higher-level languages, replacing built-in arithmetic by our C implementation. Doing it within MAPLE is not straightforward; our

MAPLE experiments stayed at the level of GCD and inversions in $\mathbb{L}_3$, for which we compared our code with MAPLE's `recden` library. We used the same degree patterns as before, but we were led to reduce the degree ranges to $4 \leq d_1d_2 \leq 204$ and $2 \leq d_3 \leq 20$. Our code uses the strategy FAST USING PRECOMPUTATIONS. The MAPLE `recden` library implements multivariate dense recursive polynomials and can be called from the MAPLE interpreter via the `Algebraic` wrapper library. Our MAPLE timings, however, do not include the necessary time for converting MAPLE objects into the `recden` format: we just measured the time spent by the function `invpoly` of `recden`. Figure 6.8 gives the timings for Pattern 3 (the other results are similar). There is a significant performance gap (our timing surface is very close the bottom). When using our PLAIN strategy, our code remains faster, but the ratio diminishes by a factor of about 4 for the largest configurations.

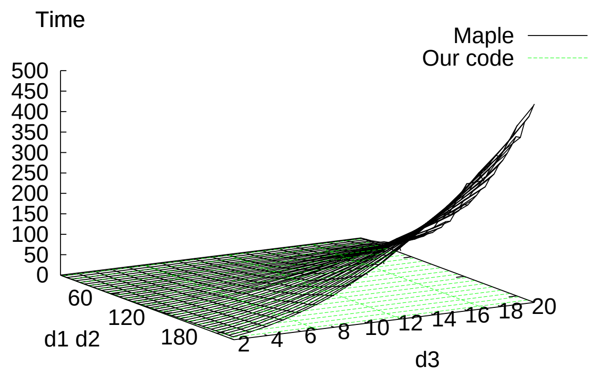


Figure 6.8: Inverse in $\mathbb{L}_3$, pattern 1, Maple vs. our code.

Comparison with AXIOM. Using our arithmetic in AXIOM is made easy by the C/GCL structure. In [65], the modular algorithm by van Hoeij and Monagan [71] was used as a driving example to show strategies for such multiple-level language implementations. This algorithm computes GCD's of univariate polynomials with coefficients in a number field by modular techniques. The coefficient field is described by a tower of simple algebraic extensions of $\mathbb{Q}$; we are thus led to compute GCD's modulo triangular sets over $\mathbb{F}_p$, for several primes p. We implemented the top-level algorithm in AXIOM. Then, two strategies were used: one relying on the built-in AXIOM modular arithmetic, and the other on our C code; the only difference between the two strategies at the top-level resides in which GCD function to call. The results are given in Figure 6.9. We use polynomials A, B in $\mathbb{Q}[X_1, X_2, X_3]/\langle T_1, T_2, T_3\rangle[X_4]$, with coefficients of absolute value bounded by 2. As shown in Figure 6.9 the gap is dramatic.

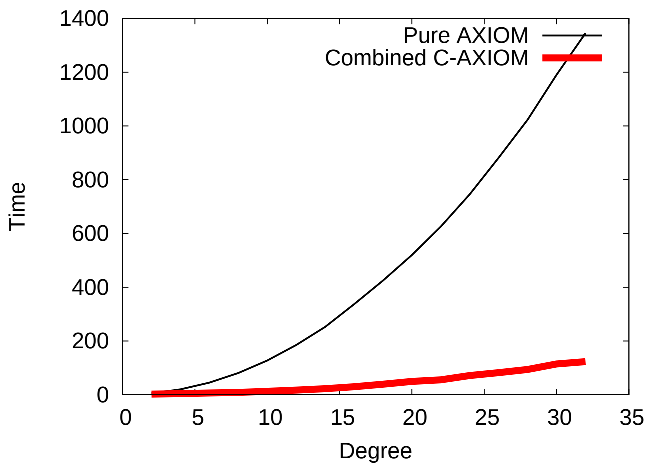


Figure 6.9: GCD computations $\mathbb{L}_3[X_4]$, pure AXIOM code vs. combined C-AXIOM code.

6.4.3 The deformation-based algorithm

We conclude with the implementation of the algorithm of Subsection 6.2.3, devoted to triangular sets made of univariate polynomials only. We focus on the most favorable case for this algorithm, when all degrees d_i are 2: in this case, in n variables, the cost reported in Proposition 6.2.4 becomes $O(2^n n \mathsf{M}(n))$. This extreme situation is actually potentially useful, see for instance an application to the addition of algebraic numbers in characteristic 2 in [85]. For most practical purposes, n should be in the range of about $1, \ldots, 20$; for such sizes, multiplication in degree n will rely on naive or at best Karatsuba multiplication; hence, a reasonable practical estimate for the previous bound is $O(2^n n^3)$, which we can rewrite as $O(\delta_T \log(\delta_T)^3)$. We compare in Figure 6.10 the behavior of this algorithm to the general one. As expected, the former behaves better: the general algorithm starts by multiplying the two input polynomials, before reducing them. The number of monomials in the product before reduction is $3^n = \delta_T^{\log_2(3)}$. Hence, for this family of problems, the general algorithm has a non-linear complexity.

6.5 Summary

We have provided new estimates for the cost of multiplication modulo a triangular set. The outstanding challenge for this question remains the suppression of ex-

variables	δ_T	general (Subsection 6.2.2)	specialized (Subsection 6.2.3)
3	8	0.000188	0.000043
4	16	0.001288	0.000126
5	32	0.007888	0.000337
6	64	0.045804	0.000983
7	128	0.254427	0.002720
8	256	1.434127	0.008141
9	512	7.682161	0.019928
10	1024	40.519331	0.052337
11	2048	204.719505	0.131778

Figure 6.10: General vs. specialized algorithm.

ponential overheads; a tempting approach is a higher-dimensional extension of the Cook-Sieveking-Kung idea, or the related Montgomery approach.

On the software level, our experiments show the importance of both fast algorithms and implementation techniques. While most of our efforts were limited to multiplication, the next steps are well-tuned inversion and GCD computations. Theory and practice revealed that, as far as multivariate multiplication is concerned, fast algorithms become faster than plain ones for very low degrees.

Chapter 7

Fast Algorithms for Regular GCD Computations and Regularity Test

Recall that in Chapters 3, 4 and 5 we have studies and implemented a set of asymptotically fast operations such as univariate/multivariate polynomial multiplication, division, GCD. In chapter 6 we have advanced our study further by considering operations modulo triangular sets, i.e. polynomial multiplication, inversion, GCD modulo a monic triangular set. In this and next chapters we develop new higher-level algorithms. They are fundamental subroutines for triangular decompositions based polynomial solving. Besides the algorithmic design, their high performance rely on the highly efficient implementations reported in previous chapters. In following sections, we report two new algorithms: polynomial GCDs modulo regular chains and regularize modulo saturated ideals.

`NOTE: This chapter is written based on the submitted Paper [67].`

7.1 Overview

A triangular decomposition of a set $F \subset \mathbb{K}[x_1, \ldots, x_n]$ is a list of polynomial systems $T_1, \ldots, T_e$, called *regular chains* (or regular systems, see Section 2.3) at Page 20 and representing the zero set $V(F)$ of F. Each regular chain T_i may encode several irreducible components of $V(F)$ provided that those share some properties (same dimension, same free variables, ...).

Triangular decomposition methods are based on an univariate and recursive vision of multivariate polynomials. Most of their routines manipulate polynomial remainder sequences (PRS). Moreover, these methods are usually "factorization free", which explains why two different irreducible components may be represented the same regular chain. An essential routine is then to check whether a hyper-surface $f = 0$ contains one of the irreducible components encoded by a regular chain T. This is achieved by testing whether the polynomial f is a zero-divisor modulo the so-called *saturated ideal* (see Section 2.3) of T. The univariate approach allows to perform this *regularity test* by means of GCD computations. However, since the saturated ideal of T may not be prime, the concept of a GCD used here is not standard.

The first formal definition of this type of GCDs was given by Kalkbrener in his PhD thesis [55]. However GCDs over non-integral domains were already used in several papers [34, 62, 46] since the introduction of the celebrated *D5 Principle* [30] by Della Dora, Dicrescenzo and Duval. Indeed, this brilliant and simple observation allows one to carry out over direct product of fields computations that are usually conducted over fields. For instance, computing univariate polynomial GCDs by means of the Euclidean Algorithm.

To define a polynomial GCD of two (or more) polynomials modulo a regular chain T, Kalkbrener refers to the irreducible components that T represents. In order to improve the practical efficiency of those GCD computations by means of subresultant techniques, Rioboo and the second author proposed a more abstract definition in [76]. Their GCD algorithm is, however, limited to regular chains with zero-dimensional saturated ideals.

While Kalkbrener's definition cover the positive dimensional case, his approach cannot support triangular decomposition methods solving polynomial systems incrementally, that is, by solving one equation after another. This is a serious limitation since incremental solving is a powerful way to control the complexity of intermediate computations and develop efficient sub-algorithms, by means of geometrical consideration. The first incremental triangular decomposition method was proposed by Lazard in [61], without proof nor a GCD definition. Another such method was presented and established by the second author in [75] together with a formal notion of GCD adapted to the needs of incremental solving. This concept, called *regular GCD*, is reviewed in Section 2.3.5 of this Chapter. It is stated there in the context of regular chains. A more abstract definition is as follows.

Let $\mathbb{A}$ be a commutative ring with unity. Let p, t, g be non-zero univariate polynomials in $\mathbb{A}[x]$. We say that g is a *regular GCD* of p, t if the following three conditions hold: (i) the leading coefficient of g in x is a regular element of $\mathbb{A}$,
(ii) g belongs to the ideal generated by p and t in $\mathbb{A}[x]$, and

(*iii*) if g has positive degree w.r.t. x, then g pseudo-divides both of p and t, that is, the pseudo-remainders $\text{prem}(p, g)$ and $\text{prem}(t, g)$ are null.

In the context of regular chains, the ring $\mathbb{A}$ is the residue class ring of a polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$ (over a field $\mathbb{K}$) by the saturated ideal $\text{sat}(T)$ of a regular chain T. Even if the leading coefficients of p, t are regular and $\text{sat}(T)$ is radical, the polynomials p, t may not necessarily admit a regular GCD (unless $\text{sat}(T)$ is prime). However, by splitting T into several regular chains $T_1, \ldots, T_e$ (in a sense specified in Section 2.3.5) one can compute a regular GCD of p, t over each of the ring $\mathbb{K}[x_1, \ldots, x_n]/\text{sat}(T_i)$, as shown in [75].

In this chapter, we propose an new algorithm for this task, together with a theoretical study and implementation report, providing significant improvements w.r.t. previous work [55, 75]. First, we aim at understanding when does a pair of polynomials p, t admit a regular GCD w.r.t. a regular chain T. In Section 7.3 of this Chapter we exhibit sufficient conditions for a subresultant of p, t (regarded as univariate polynomials in x) to be a regular GCD of p, t w.r.t. T. Some of these results are probably not new, but we could not find a reference for them, in particular when $\text{sat}(T)$ is not radical.

Secondly, we aim at making use of fast polynomial arithmetic and in particular FFT-based multivariate arithmetic. (Indeed, Euclidean-like algorithms tend to densify computations.) In addition, we observe that, when computing triangular decomposition, whenever a regular GCD of p and t w.r.t. T is needed, the resultant of p and t w.r.t. x is likely to be computed too. This suggests to organize calculations in a way that a PRS of p and t is computed only once. Moreover, we wish to follow a successful strategy introduced in [69]: compute in $\mathbb{K}[x_1, \ldots, x_n]$ instead of $\mathbb{K}[x_1, \ldots, x_n]/\text{sat}(T)$, as much as possible, while controlling expression swell. These three requirements targeting efficiency are satisfied by the regular GCD algorithm proposed in Section 7.4. The use of fast arithmetic for computing regular GCDs was proposed in [28] in the case of regular chains with zero-dimensional radical saturated ideals. However this method does not meet our two other requirements. Some complexity results for the algorithms of this chapter are given in Sections 7.5.1 and 7.5.2.

Efficient implementation is also a main objective of our work. We discuss our implementation techniques in Sections 7.5.1 and 7.5.3. In particular, we explain how we create opportunities for using modular methods and fast arithmetic in operations modulo regular chains, such as regular GCD computation and regularity test. The experimental results reported in Section 7.6 illustrate the high efficiency of our proposed algorithms. We obtain speed-up factors of several orders of magnitude w.r.t. the algorithms of [75] for regular GCD computations and regularize. In addition, our code compares and often outperforms packages with similar specifications in Maple and Magma.

7.2 Specification

In this chapter, we follow the notations used in Section 2.3:

- Let $\mathbb{K}$ be a field and let $\mathbb{K}[\mathbf{x}] = \mathbb{K}[x_1, \ldots, x_n]$ be the ring of polynomials with coefficients in $\mathbb{K}$, with ordered variables $x_1 \prec \cdots \prec x_n$.
- The *main variable* of $p \in \mathbb{K}[\mathbf{x}]$ is denoted by $\mathrm{mvar}(p)$.
- The *leading coefficient* of p in $x_i, i = 1 \ldots n$ is denoted by $\mathrm{lc}(p, x_i)$ in $lc(p,\ x_n)$.
- The *partial degree* of p in x_i is denoted by $\deg(p, x_i)$.
- The partial degree of p in its main variable is denoted by $\mathrm{mdeg}(p)$.
- The initial of p is $\mathrm{lc}(p, x_n)$ denoted by $\mathrm{init}(p)$.
- Given a triangular set T in $\mathbb{K}[\mathbf{x}]$, We denote by $\mathrm{sat}(T)$ the *saturated ideal* of T.
- Given $p \in \mathbb{K}[\mathbf{x}]$ the *pseudo-remainder* (resp. *iterated resultant*) of p w.r.t. T, denoted by $\mathrm{prem}(p, T)$.

We list below the specifications of the fundamental operations on regular chains used in this chapter. The names of these operations are the same as in the `RegularChains` library in Maple.

NormalForm. Let T be a zero-dimensional normalized regular chain, that is, a regular chain whose saturated ideal is zero-dimensional and whose initials are all in the base field $\mathbb{K}$. Observe that T is a lexicographic Gröbner basis. Then, for $p \in \mathbb{K}[\mathbf{x}]$, the operation $\mathrm{NormalForm}(p, T)$ returns the *normal form* of p w.r.t. T in the sense of Gröbner bases.

Normalize. Let T be a regular chain such that all variables occurring in T are algebraic w.r.t. T. Let $p \in \mathbb{K}[\mathbf{x}]$ a non-constant polynomial whose initial h is regular w.r.t. $\mathrm{sat}(T)$ and such that all variables occurring in h are algebraic w.r.t.

T. Then h is invertible modulo $\operatorname{sat}(T)$ and the operation $\operatorname{Normalize}(p, T)$ returns $\operatorname{NormalForm}(h^{-1}p, T)$ where h^{-1} is the inverse of h modulo $\operatorname{sat}(T)$.

RegularGcd. Let T be a regular chain and let $p, t \in \mathbb{K}[\mathbf{x}]$ be non-constant polynomials with $\operatorname{mvar}(p) = \operatorname{mvar}(t)$ and such that both $\operatorname{init}(p)$ and $\operatorname{init}(t)$ are regular w.r.t. $\operatorname{sat}(T)$. Then, the operation $\operatorname{RegularGcd}(p, t, T)$ returns a sequence of pairs $(g_1, T_1), \ldots, (g_e, T_e)$, called a *regular GCD sequence*, where $g_1, \ldots, g_e$ are polynomials and $T_1, \ldots, T_e$ are regular chains of $\mathbb{K}[\mathbf{x}]$, such that $T \longrightarrow (T_1, \ldots, T_e)$ holds and g_i is a regular GCD of p, t w.r.t. T_i for all $1 \leq i \leq e$.

Regularize. For a regular chain $T \subset \mathbb{K}[\mathbf{x}]$ and p in $\mathbb{K}[\mathbf{x}]$, the operation $\mathsf{Regularize}(p, T)$ returns regular chains $T_1, \ldots, T_e$ of $\mathbb{K}[\mathbf{x}]$ such that, for each $1 \leq i \leq e$, p is either zero or regular modulo $\operatorname{sat}(T_i)$ and we have $T \longrightarrow (T_1, \ldots, T_e)$.

7.3 Regular GCDs

Throughout this section, we assume $n \geq 2$ and we consider $p, t \in \mathbb{K}[x_1, \ldots, x_n]$ non-constant polynomials with the same main variable x_n and such that $\operatorname{mdeg}(t) \leq \operatorname{mdeg}(p)$ holds. We denote by r the resultant of p and t w.r.t. x_n. Let $T \subset \mathbb{K}[x_1, \ldots, x_{n-1}]$ be a non-empty regular chain such that $r \in \operatorname{sat}(T)$ and the initials of p, t are regular w.r.t. $\operatorname{sat}(T)$. We denote by $\mathbb{A}$ and $\mathbb{B}$ the ring of univariate polynomials in x_n with coefficients in $\mathbb{K}[x_1, \ldots, x_{n-1}]$ and $\mathbb{K}[x_1, \ldots, x_{n-1}]/\operatorname{sat}(T)$, respectively. Let Ψ be the canonical homomorphism from $\mathbb{A}$ to $\mathbb{B}$. For $0 \leq j \leq \operatorname{mdeg}(t)$, we denote by S_j the j-th subresultant of p, t in $\mathbb{A}[x_n]$.

Let d be an index in the range $1 \cdots \operatorname{mdeg}(t)$ such that $\operatorname{lc}(S_d, x_n)$ is regular modulo $\operatorname{sat}(T)$ and $S_j \in \operatorname{sat}(T)$ for all $0 \leq j < d$. Lemma 3, Lemma 4 and Corollary 1 exhibit conditions under which S_d is a regular GCD of p and t w.r.t. T.

Lemma 1. *Under the above assumptions, the polynomial S_d is a non-defective subresultant of p and t over $\mathbb{A}$. Consequently, since $\operatorname{lc}(S_d, x_n)$ is regular modulo $\operatorname{sat}(T)$, $\Psi(S_d)$ is a non-defective subresultant of $\Psi(p)$ and $\Psi(t)$ in $\mathbb{B}[x_n]$.*

PROOF. When $d = \operatorname{mdeg}(t)$ holds, we are done. Hence, we assume $d < \operatorname{mdeg}(t)$. Suppose that S_d is defective, that is, $\deg(S_d, x_n) = e < d$. According to item (r_e) in the divisibility relations of subresultants, there exists a non-defective subresultant S_{d+1} such that

$$\operatorname{lc}(S_d, x_n)^{d-e} S_d = s_{d+1}^{d-e} S_e,$$

where s_{d+1} is the leading coefficient of S_{d+1} in x_n. By our assumptions, S_e belongs to $\operatorname{sat}(T)$, thus $\operatorname{lc}(S_d, x_n)^{d-e} S_d \in \operatorname{sat}(T)$ holds. It follows from the fact $\operatorname{lc}(S_d, x_n)$ is regular modulo $\mathrm{sat}(T)$ that S_d is also in $\mathrm{sat}(T)$. However the fact that $\mathrm{lc}(S_d, x_n) = \mathrm{init}(S_d)$ is regular modulo $\mathrm{sat}(T)$ also implies that S_d is regular modulo $\mathrm{sat}(T)$. A contradiction. $\square$

The following lemma justifies the assumption that $\mathrm{lc}(S_d, x_n)$ is regular modulo $\mathrm{sat}(T)$.

Lemma 2. *With the same setting as Lemma 1, if* $\mathrm{lc}(S_d, x_n)$ *is contained in* $\mathrm{sat}(T)$*, then all the coefficients of S_d regarded as a univariate polynomial in x_n are nilpotent modulo* $\mathrm{sat}(T)$*.*

PROOF. If the leading coefficient $\mathrm{lc}(S_d, x_n)$ is in $\mathrm{sat}(T)$, then $\mathrm{lc}(S_d, x_n) \in \mathfrak{p}$ holds for all the associated primes $\mathfrak{p}$ of $\mathrm{sat}(T)$. By the Block Structure Theorem of subresultants (Theorem 7.9.1 of [72]) over an integral domain $\mathbb{K}[x_1, \ldots, x_{n-1}]/\mathfrak{p}$, S_d must belong to $\mathfrak{p}$. Hence we have $S_d \in \sqrt{\mathrm{sat}(T)}$, since $\sqrt{I}$ equals the intersection of all associated primes of I for any ideal I. That is to say, S_d is nilpotent modulo $\mathrm{sat}(T)$. It follows from Exercise 2 of [8] that all the coefficients of S_d in x_n are also nilpotent modulo $\mathrm{sat}(T)$. $\square$

The above lemma says, when $\mathrm{lc}(S_d, x_n)$ is in $\mathrm{sat}(T)$, S_d will vanish on all the components after splitting $\mathrm{sat}(T)$ sufficiently. This is the key reason that Lemma 1 can be applied for computing regular GCD modulo $\mathrm{sat}(T)$. To be more precise, there are following cases:

(1) if $\mathrm{lc}(S_d, x_n)$ is regular modulo $\mathrm{sat}(T)$, then Lemma 1 directly applies;

(2) if $\mathrm{lc}(S_d, x_n)$ is in $\mathrm{sat}(T)$, then S_d must not be a regular GCD;

(3) if $\mathrm{lc}(S_d, x_n)$ is a zero-divisor modulo $\mathrm{sat}(T)$, then it reduces to case (1) or (2) after regularizing the leading coefficient of S_d w.r.t $\mathrm{sat}(T)$.

The subresultant S_d in Lemma 1 shall be referred as the *candidate regular GCD* of p and t modulo $\mathrm{sat}(T)$.

Example 1. *If* $\mathrm{lc}(S_d, x_n)$ *is not regular modulo* $\mathrm{sat}(T)$ *then S_d may be defective. Consider for instance the following polynomials p and t in* $\mathbb{Q}[x_1, x_2, x_3]$*.*

$$p = x_3^2 x_2^2 - x_1^4 \quad \text{and} \quad t = x_1^2 x_3^2 - x_2^4.$$

We have

$$\mathrm{prem}(p, -t) = (x_1^6 - x_2^6) \quad \text{and} \quad r = (x_1^6 - x_2^6)^2.$$

Let $T = \{r\}$. Then the last subresultant of p, t modulo $\mathrm{sat}(T)$ *is* $\mathrm{prem}(p, -t)$*, which has degree 0 w.r.t x_3, although its index is* 1*. Note that* $\mathrm{prem}(p, -t)$ *is nilpotent modulo* $\mathrm{sat}(T)$*.*

In what follows, we give sufficient conditions for the subresultant S_d to be a regular GCD of p and t w.r.t. T. When $\operatorname{sat}(T)$ is a radical ideal, Lemma 4 states that the assumptions of Lemma 1 are sufficient. This lemma validates the search for a regular GCD of p and t w.r.t. T in a bottom-up style, from S_0 up to S_d for some d. Corollary 1 covers the case where $\operatorname{sat}(T)$ is not radical and states that S_d is a regular GCD of p and t modulo T, provided that S_d satisfies the conditions of Lemma 1 and provided that, for all $d < k \leq \operatorname{mdeg}(t)$, the coefficient s_k of x_n^k in S_k is either null or regular modulo $\operatorname{sat}(T)$.

Lemma 3. *Under the assumptions of Lemma 1, assume further that, for all $d < j \leq \operatorname{mdeg}(t)$, the j-th subresultant S_j of p, t is either null modulo $\operatorname{sat}(T)$ or $\operatorname{lc}(S_j, x_n)$ is regular modulo $\operatorname{sat}(T)$. Then, S_d is a regular GCD of p, t w.r.t. T.*

PROOF. The assumptions and Lemma 1 imply that $T \cup \{S_d\}$ is a regular chain. Note also that, S_d is in the ideal generated by p, t, since S_d is a subresultant of these two polynomials. Hence, to prove that S_d is a regular GCD of p, t w.r.t. T, it suffices to check that both p and t belong to $\operatorname{sat}(T \cup S_d)$. When $d = \operatorname{mdeg}(t)$ holds, we conclude by applying Property (r_{q-1}) from the divisibility relations of subresultants over an integral domain. Hence, we assume $d < \operatorname{mdeg}(t)$. Let S_j be the non-zero subresultant of smallest index i such that $\operatorname{mdeg}(t) \geq j > d$. The divisibility relations (either $(r_{<q-1})$ or (r_{e-1})) imply that $\operatorname{prem}(S_j, S_d) \in \operatorname{sat}(T)$ holds, that is, $S_j \in \operatorname{sat}(T \cup S_d)$. If $j < \operatorname{mdeg}(t)$, let S_i be the non-zero subresultant of smallest j such that $i > j$. The divisibility relations imply now that $\operatorname{prem}(S_i, S_j) \in \operatorname{sat}(T \cup S_d)$ holds. By assumption $\operatorname{init}(S_j) = \operatorname{lc}(S_j, x_n)$ is regular modulo $\operatorname{sat}(T)$. Hence, we deduce $S_i \in \operatorname{sat}(T \cup S_d)$. Continuing in this manner, we obtained the desired result. □

Corollary 1. *We reuse the notations and assumptions of Lemma 1. Then S_d is a regular GCD of p and t modulo $\operatorname{sat}(T)$, if for all $d < k \leq \operatorname{mdeg}(t)$, the coefficient s_k of x_n^k in S_k is either null or regular modulo $\operatorname{sat}(T)$.*

PROOF. Let us assume that for all $d < k \leq \operatorname{mdeg}(t)$, the coefficient s_k is either null or regular modulo $\operatorname{sat}(T)$. It follows from Lemma 3 that we only need to prove that every defective subresultant $\Psi(S_j)$ of $\Psi(p)$ and $\Psi(t)$ in $\mathbb{B}[x_n]$ has a leading coefficient which is regular w.r.t. $\operatorname{sat}(T)$. So let $d < j < \operatorname{mdeg}(t)$ such that $\Psi(S_j) \neq 0$ and $\deg(\Psi(S_j), x_n) < j$ hold. Let $k = \deg(\Psi(S_j), x_n)$. The divisibility relations of subresultants over an arbitrary commutative ring, together with the assumption that $\operatorname{init}(t)$ is regular w.r.t. $\operatorname{sat}(T)$, imply that the non-zero subresultants $\Psi(S_{j+1})$ and $\Psi(S_k)$ are non-defective and we have:

$$\operatorname{lc}(\Psi(S_j))^{j-k}\Psi(S_j) = \Psi(s_{j+1})^{j-k}\Psi(S_k).$$

This implies that $\mathrm{lc}(\Psi(S_j))$ is regular modulo $\mathrm{sat}(T)$. □

Lemma 4. *Under the assumptions of Lemma 1, assume further that* $\mathrm{sat}(T)$ *is radical. Then,* S_d *is a regular GCD of* p, t *w.r.t.* T.

PROOF. As in the proof of Lemma 3, it suffices to check that both p and t belong to $\mathrm{sat}(T \cup \{S_d\})$. Let $\mathfrak{p}$ be any prime ideal associated with $\mathrm{sat}(T)$. Define $D = \mathbb{K}[x_1, \ldots, x_n]/\mathfrak{p}$ and let $\mathbb{L}$ be the fraction field of the integral domain D. Clearly S_d is the last subresultant of p, t in $D[x_n]$ and thus in $\mathbb{L}[x_n]$. Hence S_d is a GCD of p, t in $\mathbb{L}[x_n]$. Thus S_d divides p, t in $\mathbb{L}[x_n]$ and pseudo-divides p, t in $D[x_n]$. Therefore both $\mathrm{prem}(p, S_d)$ and $\mathrm{prem}(t, S_d)$ belong to $\mathfrak{p}$. Finally $\mathrm{prem}(p, S_d)$ and $\mathrm{prem}(t, S_d)$ belong to $\mathrm{sat}(T)$. Indeed, $\mathrm{sat}(T)$ being radical, it is the intersection of its associated primes. □

7.4 A Regular GCD Algorithm

Following the notations and assumptions of Section 7.3 we propose an algorithm for computing a regular GCD sequence of p, t w.r.t. T, as specified in Section 2.3.5. This algorithm is called RGSZR for *regular gcd sequence with zero resultant.* In Section 7.4.2 we show how to relax the assumption $r \in \mathrm{sat}(T)$.

There are three main ideas behind the RGSZR algorithm. Firstly, the subresultants of p, t in $\mathbb{A}[x_n]$ are assumed to be known. We shall explain in Section 7.5 how we compute them in our implementation. Secondly, we rely on the **Regularize** operation specified in Section 2.3.5. Lastly, we inspect the subresultant chain of p, t in $\mathbb{A}[x_n]$ in a bottom-up manner. Therefore, we view $S_1, S_2, \ldots$ has successive candidates and apply Lemma 4, if $\mathrm{sat}(T)$ is known to be radical, otherwise we apply Corollary 1.

7.4.1 Case where $r \in \mathrm{sat}(T)$: the algorithm RGSZR

Calling sequence. $\mathsf{RGSZR}(p, t, x_n, T)$

Input: p, t, x_n, T as in Section 7.3.

Output: Same output specification as $\mathrm{RegularGcd}(p, t, T)$, see Section 2.3.5

S1: ***Compute the subresultants of*** p ***and*** t ***in*** x_n***.*** See Section 7.5.1 for details.

S2: ***Initializing the search for a regular GCD.*** Let $i = 1$. The index i represents the smallest possible index of a subresultant S_i of p, t (regarded in $\mathbb{A}[x_n]$) such that $S_i \notin \mathrm{sat}(T)$. Recall that $S_0 = \mathsf{res}(p, t, x_n) \in \mathrm{sat}(T)$. The algorithm manages three sets *Tasks*, *Candidates* and *Results*. Define

$$Tasks = \{[i,\, T]\}, \quad Candidates = \emptyset, \quad Results = \emptyset.$$

Each item in $Tasks$ or $Candidates$ is a pair $[\ell, C]$ where ℓ is a subresultant index in the range $1 \cdots \mathrm{mdeg}(t)$ and where C is a regular chain such that $|T| = |C|$ and $\mathrm{sat}(T) \subseteq \mathrm{sat}(C)$ hold. Each item in $Tasks$ or $Candidates$ is the input data of some computation, whereas $Results$ is the value returned by the algorithm. Each task $[\ell, C] \in Tasks$ satisfies the following: for each $0 \leq j < \ell$ we have $S_j \in \mathrm{sat}(C)$.

S3: If $Tasks = \emptyset$ then go to **S6**, otherwise continue to **S4**.

S4: ***Searching for a candidate.*** Take an item $[\ell, C]$ out of $Tasks$. If $\ell = \mathrm{mdeg}(t)$ then set $j = \ell$ and go to **S5**. Otherwise, let $j \leq \ell$ be the smallest index of a subresultant S_j of p and t such that $S_j \notin \mathrm{sat}(C)$. Observe that j exists since $\mathrm{init}(t)$ regular w.r.t. $\mathrm{sat}(T)$ implies $t \notin \mathrm{sat}(C)$.

S5: ***Checking the candidate.*** Denote by c_u the leading coefficient of S_j in x_n. If $c_u \in \mathrm{sat}(C)$ holds, then for each $D \in \mathsf{Regularize}(S_j, C)$ do the following:

$$Tasks := Tasks \ \cup \ \{[j+1, \ D]\}.$$

If $c_u \notin \mathrm{sat}(C)$ holds, then for each $D \in \mathsf{Regularize}(c_u, C)$ do the following:

(a) if $c_u \notin \mathrm{sat}(D)$ then

$$Candidates := Candidates \ \cup \{[j, \ D]\};$$

(b) if $c_u \in \mathrm{sat}(D)$ then

$$Tasks := Tasks \ \cup \ \{[j, \ D]\}.$$

Go back to **S3**.

We make two comments. When $c_u \in \mathrm{sat}(C)$ holds, by Lemma 2, S_j is nilpotent modulo $\mathrm{sat}(C)$. Hence after regularizing S_j, S_j belongs to $\mathrm{sat}(D)$ for each D and we can proceed to the next level $j+1$. When $c_u \notin \mathrm{sat}(C)$, we split C by regularizing c_u. In case (a), the polynomial c_u is regular modulo $\mathrm{sat}(D)$ and, by Lemma 1, S_j is non-defective. We regard S_j as a candidate regular GCD of p, t w.r.t. D. In case (b), the polynomial c_u is in $\mathrm{sat}(D)$, we simply add it back to the task pool.

S6: ***Applying Lemma 4.*** If $\mathrm{sat}(T)$ is not known to be radical then go to **S7**. Otherwise, for all $[j, \ D] \in Candidates$ set

$$Results := Results \ \cup \ \{[S_j, \ D]\}$$

and return $Results$. Observe that for all $[j, \ D] \in Candidates$ the ideal $\mathrm{sat}(D)$ is radical too. Thus, Lemma 4 shows that S_j is a regular GCD of p, t w.r.t D.

S7: ***Applying Corollary 1.*** For each $[j, \ D]$ in $Candidates$,

(a) Set $Tasks = \{[j,\, D]\}$ and $Split = \emptyset$.

(b) while $Tasks \neq \emptyset$ do

(b.1) Take an element $[\ell,\, E]$ out of $Tasks$.

(b.2) Let $\ell < k \leq \mathrm{mdeg}(t)$ be the smallest index of a subresultant S_k such that s_k (the coefficient of S_k in x_n^k) is non-zero modulo $\mathrm{sat}(E)$.

(b.3) If $k = \mathrm{mdeg}(t)$ then $Split := Split\ \cup\ \{E\}$. Otherwise, for each $F \in$ $\mathsf{Regularize}(s_k, E)$ do $Tasks := Tasks\ \cup\ \{[\ell+1, F]\}$.

(c) For each regular chain $E \in Split$

$$Results := Results\ \cup\ \{[S_j, E]\}.$$

Finally, we return $Results$.

7.4.2 Case where $r \notin \mathrm{sat}(T)$

We explain how to relax the assumption $r \in \mathrm{sat}(T)$ and thus obtain a general algorithm for the operation $\mathsf{RegularGcd}$. The principle is straightforward. Let $r = \mathsf{res}(p, t, x_n)$. Then, we call $\mathsf{Regularize}(r, T)$ obtaining regular chains $T_1, \ldots, T_e$ such that $T \longrightarrow (T_1, \ldots, T_e)$. For each $1 \leq i \leq e$ we compute a regular GCD sequence of p and t w.r.t. T_i as follows: If $r \in \mathrm{sat}(T_i)$ holds then we call $\mathsf{RGSZR}(p, t, x_n, T_i)$; otherwise $r \notin \mathrm{sat}(T_i)$, the resultant r is actually a regular GCD of p and t w.r.t. T_i by the definition. Observe that in the case where $r \in \mathrm{sat}(T_i)$ holds the subresultant chain of p and t in x_n is used to compute their regular GCD w.r.t. T_i. This is one of the motivations for the implementation techniques described in Section 7.5.

7.5 Implementation and Complexity

In this section we address implementation techniques and complexity issues. We follow the notations introduced in Section 7.3. However we do not assume that $r = \mathsf{res}(p, t, x_n)$ belongs to the saturated ideal of the regular chain T.

In Section 7.5.1 we describe our encoding of the subresultant chain of p, t in $\mathbb{K}[x_1, \ldots, x_{n-1}][x_n]$. This representation is used in our implementation and complexity results. For simplicity our analysis is restricted to the case where $\mathbb{K}$ is a finite field whose "characteristic is large enough". The case where $\mathbb{K}$ is the field $\mathbb{Q}$ of rational numbers could be handled in a similar fashion, with the necessary adjustments.

One motivation for the design of the techniques presented in this chapter is the solving of systems of two equations, say $p = t = 0$. Indeed, this can be seen as a fundamental operation in incremental methods for solving systems of polynomial equations, such as the one of [75]. We make two simple key observations. Formula 2.25 p. 25 shows that solving this system reduces "essentially" to computing r and a regular GCD sequence of p, t modulo $\{r\}$, when r is not constant. This is particularly true when $n = 2$ since in this case the variety $V(h, p, t)$ is likely to be empty for "generic" polynomials p, t. The second observation is that, under the same genericity assumptions, a regular GCD g of p, t w.r.t. $\{r\}$ is likely to exist and to have degree one w.r.t. x_n. Therefore, once the subresultant chain of p, t w.r.t. x_n is calculated, one can obtain g "essentially" at no cost. Section 7.5.2 extends these observations with two complexity results.

In Section 7.5.3 an algorithm for the operation Regularize and its implementation are discussed. We show how to create opportunities for making use of fast polynomial arithmetic and modular techniques, bringing a significant improvement w.r.t. other algorithms for the same operation, as illustrated in Section 7.6.

7.5.1 Subresultant chain encoding

Following [23], we evaluate $(x_1, \ldots, x_{n-1})$ at sufficiently many points such that the subresultants of p and t (regarded as univariate polynomials in x_n) can be computed by interpolation. To be more precise, we need some notations. We denote by d_i the maximum of the degrees of p and t in x_i, for all $i = 1, \ldots, n$. Observe that $b_i := 2d_id_n$ is an upper bound for the degree of r (or any subresultant of p and t) in x_i, for all $i = 1, \ldots, n$. Let B be the product $(b_1 + 1) \cdots (b_{n-1} + 1)$.

We proceed by evaluation/interpolation; our sample points are chosen on an $(n-1)$-dimensional rectangular grid. We call "Scale" the evaluation of the subresultant chain of p, t on this grid, which is how the subresultants of p, t are encoded in our implementation. Of course, the validity of this approach requires that our evaluation points cancel no leading term in p or t. Even though finding such points deterministically is a difficult problem, this created no issue in our implementation. Whenever possible (typically, over suitable finite fields), we choose roots of unity as sample points, so that we can use FFT (or van der Hoeven's Truncated Fourier Transform [51]); otherwise, the standard fast evaluation/interpolation algorithms are used. We have $O(d_n)$ evaluations and $O(d_n^2)$ interpolations to perform. Since our evaluation points lie on a grid, the total cost becomes

$$O\left(Bd_n^2\sum_{i=1}^{n-1}\log(b_i)\right) \quad \text{or} \quad O\left(Bd_n^2\sum_{i=1}^{n-1}\frac{\mathsf{M}(b_i)\log(b_i)}{b_i}\right),$$

depending on the choice of the sample points (see e.g. [78] for similar estimates). Here, as usual, $\mathsf{M}(b)$ stands for the cost of multiplying polynomials of degree less than b, see [43, Chap. 8]. Using the estimate $\mathsf{M}(b) \in O(b\log(b)\log\log(b))$ from [21], this respectively gives the bounds

$$O(d_n^2 B\log(B)) \quad \text{and} \quad O(d_n^2 B\log^2(B)\log\log(B)).$$

These estimates are far from optimal. A first improvement (present in our code) consists in interpolating only the *leading coefficients* of the subresultants in a first time, and recover all other coefficients when needed. This is sufficient for the algorithms of Section 7.3. For instance, in the FFT case, the cost is reduced to

$$O(d_n^2 B + d_n B\log(B)).$$

Another desirable improvement would of course consist in using fast arithmetic based on *Half-GCD* techniques [43], with the goal of reducing the total cost to $O\tilde{}(d_n B)$, which is the best known bound for computing the resultant, or a given subresultant. However, as of now, we do not have such a result, due to the possible splittings.

7.5.2 Solving two equations

Our goal now is to estimate the cost of computing the polynomials r and g in the context of Formula 2.25 p. 25. We propose an approach where the computation of g essentially comes free, once r has been computed. This is a substantial improvement compared to traditional methods, such as [56, 75], which compute g without recycling the calculation of r. With the assumptions and notations of Section 7.5.1, we saw that the resultant r can be computed in at most $O(d_n B\log(B) + d_n^2 B)$ operations in $\mathbb{K}$. In many cases (typically with random systems), g has degree one in $v = x_n$. Then, the GCD g can be computed within the same bound as the resultant. Besides, in this case, one can use the Half-GCD approach instead of computing all subresultants of p and t. This leads to the following result in the bivariate case; we omit its proof here.

Corollary 2. *With $n = 2$, assuming that $V(h, p, t)$ is empty, and assuming*

$\deg(g, v) = 1$, *solving the input system* $p = t = 0$ *can be done in* $O^{\sim}(d_2^2 d_1)$ *operations in* $\mathbb{K}$.

7.5.3 Implementation of Regularize

Regularizing a polynomial w.r.t regular chain is a fundamental operation in methods computing triangular decompositions. It has been used in the algorithms presented in Section 7.4 and its specification can be found in Section 2.3.5. Algorithms for this operation appear in [56, 75].

The purpose of this section is to show how to realize efficiently this operation. For simplicity, we restrict ourselves to regular chains with zero-dimensional saturated ideals, in which case the separate operation of [56] and the regularize operation [75] are similar. For such a regular chain T in $\mathbb{K}[\mathbf{x}]$ and a polynomial $p \in \mathbb{K}[\mathbf{x}]$ we denote by RegularizeDim0(p, T) the function call Regularize(p, T). In broad terms, it "separates" the points of $V(T)$ that cancel p from those which do not. The output is a set of regular chains $\{T^1, \ldots, T^e\}$ such that the points of $V(T)$ which cancel p are given by the T^i's modulo which p_i is null.

Algorithm 1 differs from those with similar specification in [56, 75] by the fact it creates opportunities for using modular methods and fast polynomial arithmetic. Our first trick is based on the following result (Theorem 1 in [22]): the polynomial p is invertible modulo T if and only if the iterated resultant of p with respect to T is non-zero. The correctness of Algorithm 1 follows from this result, the specification of the algorithm of RGSZR and an inductive process. Similar proofs appear in [56, 75].

The main novelty of Algorithm 1 is to employ the fast evaluation/interpolation strategy described in Section 7.5.1. In our implementation of Algorithm 1, at Step (6), we compute the "Scube" representing the subresultant chain of q and C_v. This allows us to compute the resultant r and then to compute the regular GCDs (g, E) at Step (12) from the same "Scube". In this way, intermediate computations are recycled. Moreover, fast polynomial arithmetic is involved through the manipulation of the "Scube".

Algorithm 1.

Input: T *a normalized zero-dimensional regular chain and* p *a polynomial, both in* $\mathbb{K}[x_1, \ldots, x_n]$.

Output: *See specification in Section 2.3.5.*

```
RegularizeDim0$(p, T)$ ==
(1)  $Results := \emptyset$;
(2)  for $(q, C) \in$ RegularizeInitDim0$(p, T)$ do
(3)      if $q \in \mathbb{K}$ then
(4)          $Results := \{C\} \cup Results$
(5)      else $v := \mathrm{mvar}(q)$
(6)          $r := \mathsf{res}(q, C_v, v)$
(7)          for $D \in$ RegularizeDim0$(r, C_{<v})$ do
(8)              $s := \mathrm{NormalForm}(r, D)$
(9)              if $s \neq 0$ then
(10)                 $U := \{D \cup \{C_v\} \cup C_{>v}\}$
(11)                 $Results := \{U\} \cup Results$
(12)             else for $(g, E) \in \mathrm{RegularGcd}(q, C_v, D)$ do
(13)                 $g := \mathrm{NormalForm}(g, E)$
(14)                 $U := \{E \cup \{g\} \cup D_{>v}\}$
(15)                 $Results := \{U\} \cup Results$
(16)                 $c := \mathrm{NormalForm}(\mathrm{quo}(C_v, g), E)$
(17)                 if $\deg(c, v) > 0$ then
(18)                     $Results :=$
                         RegularizeDim0$(q, E \cup c \cup C_>v)$
                         $\cup Results$
(19) return $Results$
```

In Algorithm 1, a routine RegularizeInitialDim0 is called, whose specification and pseudo-code are given below. Briefly speaking, this routine splits a regular chain T according to the initial of a polynomial p such that p either is a constant or has a regular initial over each component of $\mathrm{sat}(T)$.

Algorithm 2.

Input: *T a normalized zero-dimensional regular chain and p a polynomial, both in $\mathbb{K}[x_1, \ldots, x_n]$.*

Output: *A set of pairs* $\{(p_i, T_i) \mid i = 1 \cdots e\}$*, in which* p_i *is a polynomial and* T_i *is a regular chain, such that either* p_i *is a constant or its initial is regular modulo* $\mathrm{sat}(T_i)$*, and* $p \equiv p_i \bmod \mathrm{sat}(T_i)$ *holds.*

```
RegularizeInitDim0($p, T$) ==
(1)  $p := \mathrm{NormalForm}(p, T)$
(2)  $Tasks := \{(p, T)\}$
(3)  $Results := \emptyset$
(4)  while $Tasks \neq \emptyset$ do
(5)      Take a pair $(q, C)$ out of $Tasks$
(6)      if $q \in \mathbb{K}$ then
(7)          $Results := \{(q, C)\} \cup Results$
(8)      else for $D \in$ RegularizeDim0($\mathrm{init}(q), C$) do
(9)          $t := \mathrm{NormalForm}(\mathrm{tail}(q), D)$
(10)         $h := \mathrm{NormalForm}(\mathrm{init}(q), D)$
(11)         if $h \neq 0$ then
(12)             $Results := \{(h\,\mathrm{rank}(q) + t, D)\} \cup Results$
(13)         else $Tasks := \{(t, D)\} \cup Tasks$
(14) return $Results$
```

7.6 Experimentation

We have implemented in C language all the algorithms reported in the previous sections. The new implementations rely on the set of asymptotically fast polynomial arithmetic operations from our *modpn* library [68] as their base level sub-routines. We also provide a Maple interface FastArithmeticTools calling these new implementations and our previous ones reported in [68]. In this section, we compare the performance of our algorithms and their implementation with Maple's and/or Magma's existing counterparts. For Maple, we use its latest distribution version 13; For Magma we ordered its latest version V2.15-4 however the performance for the algorithms we have benchmarked on such as *TriangularDecomposition* and *Saturation* is slower than the ones in the previous version, thus we still use Magma's Version V2.14-8. We focus on *Resultant and GCD* in Section 7.3 and *Regularize* in Section 7.5.3. All the benchmarks are conducted on Intel Pentium VI, Quad CPU 2.40 GHZ machines with 4 MB cache and 3 GB main memory.

7.6.1 Resultant and GCD

In Figure 7.1 we benchmark our *Resultant and GCD* algorithm. The “degree” shown in the figure is the partial degree of each input polynomial in its main variable. The input polynomials are random dense polynomial in two variables and each of them has a totally degree of the square of “degree” (see the first line of definition of “degree”). This is one of the so-called “internal” benchmarks. Namely we compare two flavor of implementations of our *Resultant and GCD* algorithm. One is based on the subproduct-tree interpolation method, the other is based on the DFT interpolation. Obviously the DFT based approach is faster in this benchmark. However the subproduct-tree is more generally applicable since it does not require the characteristic p to be a Fourier prime. Figures 7.2 and 7.3 have the same setting except they are the 3-variable and 4-variable cases respectively.

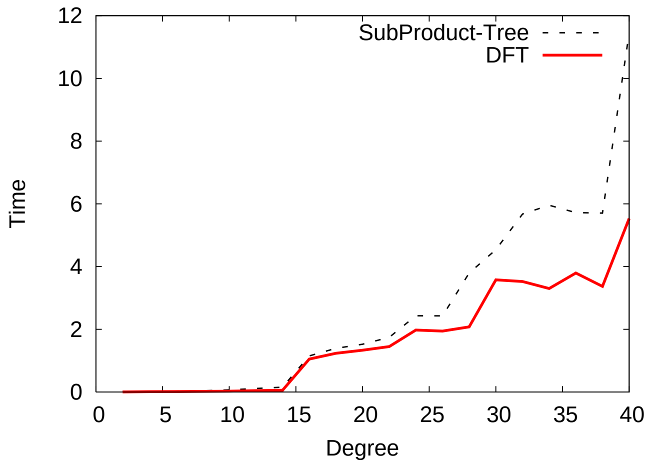


Figure 7.1: Resultant and GCD random dense 2-variable.

Figure 7.4 is one of so-called “external” benchmarks. We are comparing our *Resultant and GCD* algorithm with Magma’s counterpart. In Figure 7.4 we use the same “degree” as defined in previous *Resultant and GCD* benchmark. As shown our performance is way beyond Magma’s.

d_1.	d_2	Regularize	Fast Regularize	Magma
2	2	0.000	0.004	0.000
4	6	0.044	0.000	0.010
6	10	1.256	0.012	0.020
8	14	6.932	0.020	0.070
10	18	35.242	0.048	0.160
12	22	$>$ 100.000	0.052	0.370
14	26	$>$ 100.000	0.100	0.900
16	30	$>$ 100.000	0.132	1.760
18	34	$>$ 100.000	0.240	3.260
20	38	$>$ 100.000	0.472	6.400
22	42	$>$ 100.000	0.428	11.150
24	46	$>$ 100.000	0.668	18.890
26	50	$>$ 100.000	1.304	29.120
28	54	$>$ 100.000	1.052	44.770
30	58	$>$ 100.000	1.260	74.450
32	62	$>$ 100.000	2.408	97.380
34	66	$>$ 100.000	3.768	183.930

Table 7.1: Random dense 2-variable case.

d_1	d_2	d_3	Regularize	Fast Regularize	Magma
2	2	3	0.032	0.004	0.010
3	4	6	0.160	0.016	0.020
4	6	9	0.404	0.024	0.060
5	8	12	>100	0.129	0.330
6	10	15	>100	0.272	1.300
7	12	18	>100	0.704	5.100
8	14	21	>100	1.276	14.530
9	16	24	>100	5.836	40.770
10	18	27	>100	9.332	107.280
11	20	30	>100	15.904	229.950
12	22	33	>100	33.146	493.490

Table 7.2: Random dense 3-variable case.

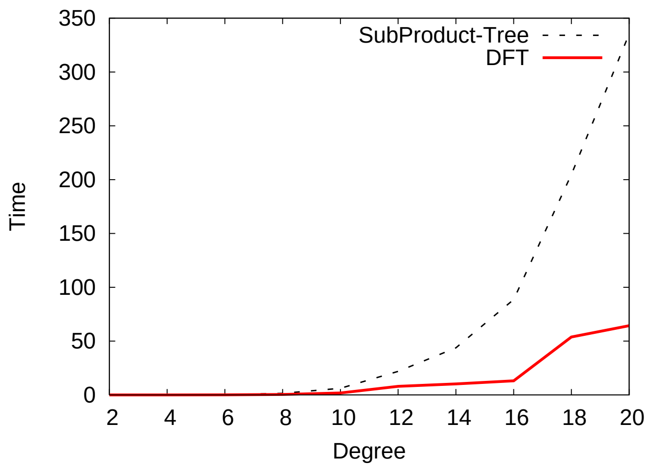


Figure 7.2: Resultant and GCD random dense 3-variable.

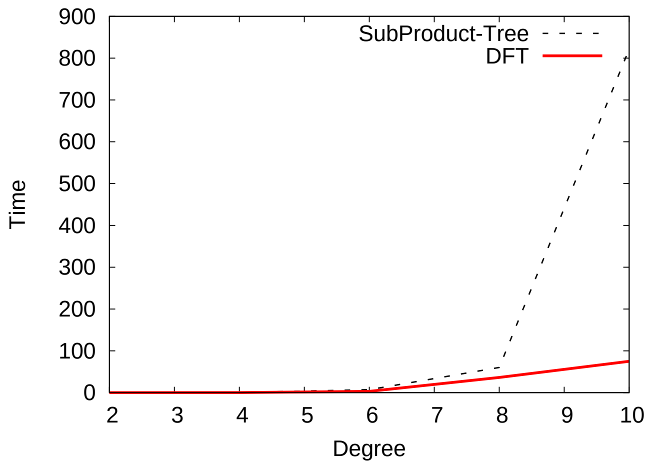


Figure 7.3: Resultant and GCD random dense 4 variable.

7.6.2 Regularize

In the following benchmarks (Tables 7.1, 7.2, and 7.4), we compare our fast regularize algorithm with "Regularize" from Maple RegularChains library and Magma's counterpart. Namely, in Magma we first saturate the ideal generated by the triangular set with an input polynomial by using the *Saturation* command. Then we use *TriangularDecomposition* command to decompose the output from the first step. The total degree of the input polynomial i is d_i. In Table 7.1, we generate two random dense polynomials with 2 variables for each, thus we are generally in the equiprojectable case and the "split" step in terms of triangular decomposition rarely happen. Similarly in Table 7.2, we generate three random dense polynomials with 3 variables for

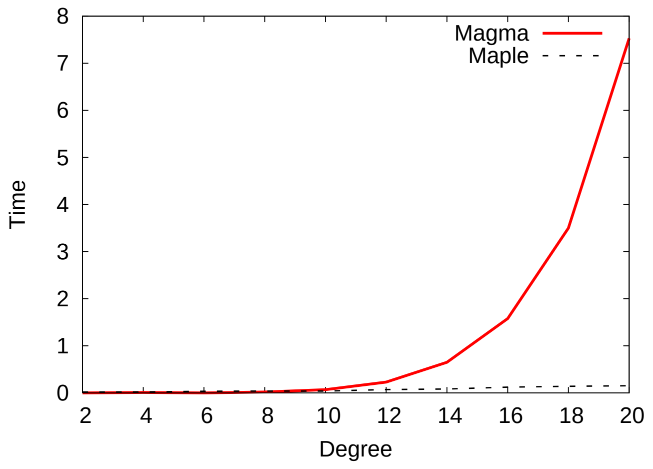


Figure 7.4: Resultant and GCD random dense 3-variable.

each. In this "non-splitting" (equiprojectable) case, our fast regularize algorithm is significantly faster than the other two implementations. For the three variables case, we are more than 150 times faster than both Magma's and RegularChains "Regularize" for the larger input examples. However, in the "splitting" (non-equiprojectable) case where we design the input systems with large number of "split" in terms of triangular decomposition, our fast regularize is slightly slower than Magma's counterpart, but still much faster than "Regularize" from RegularChains. Table 7.3 shows the run time of the "split" case with two input bivariate polynomials. Table 7.4 shows the run time of the "split" case with three input trivariate polynomials.

7.7 Summary

The concept of a regular GCD extends the usual notion of polynomial GCD from polynomial rings over fields to polynomial rings modulo saturated ideals of regular chains. Regular GCDs play a central role in triangular decomposition methods. Traditionally, regular GCDs are computed in a top-down manner, by adapting standard PRS techniques (Euclidean algorithm, subresultant algorithms, etc.).

In this chapter, we have examined the properties of regular GCDs of two polynomials w.r.t a regular chain. With the Algorithm RGSZR presented in Section 7.3, our main theoretical result, one can proceed in a bottom-up manner. This has three benefits described in Section 7.5. Firstly, this algorithm is well-suited to employ modular methods and fast polynomial arithmetic. Secondly, we avoid the repetition of (potentially expensive) intermediate computations. Lastly, we avoid, as much as

d_1.	d_2	Regularize	Fast Regularize	Magma
2	2	0.024	0.004	0.000
4	6	0.232	0.012	0.000
6	10	1.144	0.016	0.010
8	14	7.244	0.040	0.030
10	18	25.281	0.080	0.050
12	22	$>$ 100.000	0.176	0.090
14	26	$>$ 100.000	0.340	0.250
16	30	$>$ 100.000	0.516	0.280
18	34	$>$ 100.000	1.196	0.630
20	38	$>$ 100.000	1.540	0.920
22	42	$>$ 100.000	2.696	1.450
24	46	$>$ 100.000	3.592	2.540
26	50	$>$ 100.000	4.328	4.700
28	54	$>$ 100.000	6.536	4.790
30	58	$>$ 100.000	10.644	6.570
32	62	$>$ 100.000	10.028	9.360
34	66	$>$ 100.000	15.648	11.540

Table 7.3: Non-equiprojectable 2-variable case.

d_1	d_2	d_3	Regularize	Fast Regularize	Magma
2	2	3	0.292	0.012	0.000
3	4	6	1.732	0.028	0.010
4	6	9	68.972	0.072	0.030
5	8	12	328.296	0.204	0.150
6	10	15	>1000	0.652	0.370
7	12	18	>1000	2.284	1.790
8	14	21	>1000	5.108	2.890
9	16	24	>1000	18.501	10.950
10	18	27	>1000	31.349	19.180
11	20	30	>1000	55.931	56.850
12	22	33	>1000	101.642	76.340

Table 7.4: Non-equiprojectable 3-variable case.

possible, computing modulo regular chains and use polynomial computations over the base field instead, while controlling expression swell. The experimental results reported in Section 7.6 illustrate the high efficiency of our algorithms.

Chapter 8

The Modpn Library: Bringing Fast Polynomial Arithmetic into MAPLE

8.1 Overview

In Chapter 7 at Page 91, we have reported our new algorithms for the *Regular GCD* and *Regularize* operations. The latter can be regarded as an application of the former. We also mentioned briefly another application of *Regular GCD*, i.e *two-equation solver*. In this chapter, besides explaining in greater details for the *two-equation solver*, we report two other based on the operation *Regular GCD* based algorithms: *Bivariate Solver* and *Invertibility Test*. We are restricted to the two variable case for *Bivariate Solver*, thus more specialized tricks can be applied as described in Section 8.3 at Page 118. *Invertibility Test* is also a specialized algorithm with respect to *Regularize* since it assumes that the input regular chain is zero-dimensional and generates a radical ideal.

Besides the theoretical result, we are more interested in the implementation strategy for computations modulo regular chains. Therefore, while reporting the new algorithms, we will combine the practical programming consideration. Moreover, we have also conducted new experimentation in terms of programming environment. Recall that in Chapter 3 at Page 26, Chapter 4 at Page 45, Chapter 5 at Page 57, and Chapter 6 at Page 65, we use AXIOM as the experimentation environment. In this chapter, we investigate the integration of fast arithmetic operations implemented in C into MAPLE. Most of MAPLE library functions are high-level interpreted code such as the *Regularchains* library. Our objective is to let these high-level triangular composition library benefit from our C-level fast routines. However, to reach this goal, we have to handle the following facts in a careful manner. To our knowledge, the standard method to connect C code into Maple is simple but quite rudimentary. The only structured data which can be recognized by the both sides are the simple ones such as strings, arrays, tables. This leads to potential conversion overheads. Indeed, generally, Maple polynomials are represented by sparse data structures whereas those used by fast arithmetic operations are dense. Thus, we have to convert Maple sparse object into our dense object. This situation implies a second downside factor: Since conversions from Maple to C objects must be performed on the Maple side as interpreted code, the overhead of conversion is significant. Clearly, one would like to implement them on the C side, as compiled and optimized code. However, this requires a lot expertise of *OpenMaple* (see Maple help page) which is huge amount of efforts. The third disadvantage is that the Maple language does not enforce "modular programming" or "generic programming" compared to AXIOM integration. Only providing a Maple *connection-package* capable of calling our C routines will not be sufficient to speed up all Maple triangular decomposition libraries. Clearly, high-level Maple code also needs to be carefully rewritten to call this connection-package in a delicate manner. The "top-level" algorithms such as bivariate solver, two-equation solver, invertibility test, are written in Maple and relies on our C routines of different tasks such as the computation of subresultant chain, normal form of a polynomial w.r.t. a zero-dimensional regular chain, etc. These three applications are actually part of the new module of the `RegularChains` library, called `FastArithmeticTools`, which provides operations on regular chains (in prime characteristic and mainly in dimensions zero or one) based on modular methods and fast polynomial arithmetic. Therefore, these three applications are well representatives and simple enough such that their performance can be sharply evaluated.

After the success of this experimentation, we have collected selectively all our past C level implementation as a complete library called `modpn` (Multivariate Polynomial Arithmetic Modulo a prime number with N variables). As mentioned in Section 1.2, this library is in features of asymptotically fast polynomial arithmetic and their highly efficient implementation. This library targets on supporting symbolic polynomial solving via triangular decomposition techniques. `modpn` has already been accepted and integrated into the latest Maple distribution, version 13 (at the time writing this thesis).

The outline of this chapter is as following: In Section 8.2 at Page 114, we investigate the integration of asymptotically fast arithmetic operations implemented in C into Maple. In Sections 8.3 at Page 118 and 8.4 at Page 124, we present our new algorithms **Two-equation Solver** and **Invertibility Test** and their implementation. In Section 8.5 at Page 127, we show the performance result of our new algorithms. We demonstrate that with suitable implementation strategies, our new algorithms are highly effective methods.

`NOTE: This chapter is written based on the published Paper [68].`

8.2 A Compiled-Interpreted Programming Environment

Our library, `modpn`, contains two levels of implementation: MAPLE code (interpreted) and C code (compiled); our purpose is to reach high performance while spending a reasonable amount of development time. Relying on asymptotically fast algorithms and code optimization, the C level routines are very solid result. The "core" operations consist of modular multiplication/inversion 6, lifting techniques [84]), GCD's, resultants and fast interpolation, etc. At the MAPLE level, we write more abstract algorithms; typically, they are higher level polynomial solvers. The major trade-off between two levels is language abstraction and high performance.

We use multiple polynomial data encoding at each level, showed in Figure 8.1. The *Maple-Dag* and *Maple-Recursive-Dense* polynomials are MAPLE built-in types; the *C-Dag*, *C-Cube* and *C-2-Vector* polynomials are written in C by us. Each encoding will be used in certain computation; for instance *C-Cube* will be used in the fast dense computation at C level and *Maple-Dag* will be used in regular chain computation at MAPLE level. Our polynomial solving algorithms are each composed by such different computations. Therefore, at run time in the same algorithm a polynomial may need to be represented differently. Consequently, how to efficiently map one encoding to another, especially from MAPLE level ones to C level ones (or vice versa) is highly important.

For the four questions regarding C/MAPLE integration mentioned in Section 1.2, we try to answer the first two in Sections 8.2.1, 8.2.2 and 8.2.3:

- To what extent triangular decomposition algorithms can take advantage of fast polynomial arithmetic implemented in C?
- What is a good design for a hybrid C-MAPLE application?

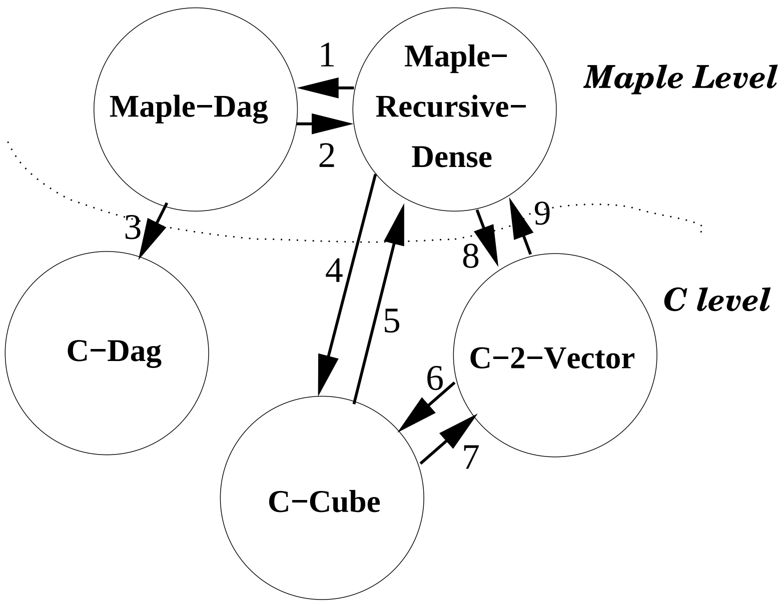


Figure 8.1: The polynomial data representations in `modpn`.

8.2.1 The C level

Primarily, our C code targets on the best performance. All operations are based on asymptotically fast algorithms rooted at fast Fourier transform (FFT) and its variant truncated Fourier transform (TFT) [51]. These operations are optimized with respect to crucial features of hardware architecture: memory hierarchy, instruction pipe-lining, and vector instructions. As reported in Chapters 5, 6, and 7 (or see Papers [69, 65]), our C library often outperforms the best known implementations such as MAGMA and NTL [5, 6].

Large portion of the C code is dedicated to regular chain operations modulo a machine size prime number, mainly in dimension zero. Such computation typically generates dense polynomials in the middle stages; thus, we use multidimensional arrays as the canonical encoding for polynomials, and we call them C-Cube*s* This encoding is the most appropriate one for FFT-based modular multiplication, inversion, interpolation, etc. For this encoding, we can pre-allocate the working buffer since all the partial degrees of a polynomial are bounded by the given regular chain. Then, in-place operations can be easy conducted on these buffers whenever they are applicable. Moreover, tracing coefficients and degrees also becomes trivial constant operations.

Besides C-Cube, we have another polynomial encoding called C-Dag. It's designed for triangular lifting algorithms [84, 27]. in which we use a Directed Acyclic Graph (DAG) to encode a polynomial. Actually, DAG polynomials is the default data presentation in Maple. Our C-Dag polynomials are used at C level only. This data representation has its unique properties, such as by setting flags in the nodes of these Dags, we can track their visibility and aliveness in constant time.

In addition to C-Cube and C-Dag, we have implemented a third data structure at C level called C-2-Vector. At the beginning of this chapter, we mentioned that the overhead of data conversion between Maple and C can be significant. Thus, we designed C-2-Vector to ease this problem (see 8.2.3 for explanation).

8.2.2 The Maple level

Many complex algorithms for triangular decompositions are highly abstract, so it is sensible to implement them in a well equipped high-level language environment like Maple. First, the implementation effort is much less intensive than that in C or C++; Second, Maple has a comprehensive mathematical library, so it is possible to directly use other existing algorithms to verify our results. In our case, we use Maple `RegularChains` library [63] to verify the result of our new algorithms and their implementation. At the Maple level, we use two types of polynomials: Maple Dags and `RecDen` (recursive dense) polynomials. As mentioned previously, Dags are the default data representation for polynomials in Maple. For example, Maple `RegularChains` library uses it uniformly. Thus, for the hybrid Maple/C implementation, we need to convert the C level polynomials to Maple Dag's (vice versa).

`RecDen` is an efficient Maple library for doing dense polynomial computation. It has its own data representation for polynomials; we call it `RecDen` polynomials. In our hybrid implementation we use some `RecDen` operations, thus we need the data representation conversion.

8.2.3 Maple and C cooperation

When designing polynomial solving algorithms such as the ones reported in Section 8.3 at Page 118 and 8.4 at Page 124, we try to rely on the fast arithmetic in our C library. Recall our first question: is this an effective approach? Our answer is a conditional yes: if the code integration process is careful, our C code provides a large speed-up to the Maple code. This has been demonstrated in Section 8.5 at Page 127. However, if the overall overhead of data conversion between C and Maple is significant this might not be a good approach. This observation naturally leads we to investigate this overhead and the methods to reduce it.

For general users, Maple `ExternalCalling` package is the only standard way to link in externally C functions. The procedure of linking is not complicated: the user just needs to carefully map MAPLE level data onto C level ones. For example, a MAPLE `rtable` type can be directly mapped to a C level array. However, if the MAPLE data encoding is very different from the C one, the data conversion might be an issue. Actually, there are only a small group of simple MAPLE data structures, such as integers, floats or tables, can be automatically converted into C responding ones. For other compound data structures, such as converting from a MAPLE Dag polynomial to a C Dag polynomial, we have to manually pack the data into a MAPLE `rtable`, and unpack it at C level. In other words, we need to "encode" the data at MAPLE level and "decode" it at the C level. This encoding/decoding process maybe expensive especially at the MAPLE end. There are two major ways to reduce this overhead:

1. to minimize the amount of conversions at the algorithm design level,

2. to minimize the amount of time for each conversion at the implementation level.

The amount of conversions is application dependent; it turns out that it happens quite often in the implementation of our new algorithms. Many conversions are "voluntary": namely, we are willing to conduct them, expecting that better algorithms or better implementations can be used after converting to suitable data representation. For example, in the triangular lifting algorithm we use C-Dag as the default representation since it is more efficient for the sub operations such as differentiation, variable substitution and variable lifting. However, we need to convert the C-Dag polynomials into C-Cube polynomials in the middle stage to use our FFT based fast arithmetic. We are willing to pay this overhead since the speed-up from FFT outweighs the extra cost from the data conversion. However, some conversions are "involuntary". Indeed, we would like all the computational intensive operations are implemented at the C level. However, this is unrealistic due to the complexity of implementation. Thus, there are often cases that we have to convert polynomials from C to MAPLE to use MAPLE level operations. As mentioned previously, the data conversion of polynomials might be very expensive. Therefore, we need to carefully study both the "voluntary" and "involuntary" conversions and decide 2 things: (1) what kind polynomial arithmetic or which sub-algorithm should be used. (2) which portion of the code should rely on MAPLE code or instead on the C code.

The amount of time for each conversion can be reduced by carefully designed data converters. For example, as mentioned previously we designed a so-called C-2-Vector polynomial representation: one vector we called `degree vector` recursively encodes the degrees of all `polynomial coefficients`, and the other vector we called `number coefficients vector` encodes all the base number coefficients. Two vectors use the same traversal order to encode information. To be specific, the recursive dense polynomial representation [64] uses a tree structure to encode a multivariate polynomial. The root itself represents the given polynomial. Its children nodes are its coefficients which may have their own children nodes, i.e. their coefficients. The leaves in the tree are numbers from the base ring. We call the nodes between the root and the leaves are `polynomial coefficients`. Therefore, by choosing a fixed tree traversal order we encode the degrees of those `polynomial coefficients` into the `degree vector`. Then accordingly, we use the same traversal order to encode the number coefficients into the `number coefficients vector`.

This data representation in our library does not participate to any real computation: it is specifically designed for facilitating the data conversion from C-Cube to `RecDen` encoding. The C-2-Vector encoding has the same recursive structure as `RecDen`, so the data conversion become easier. Moreover, the C-2-Vector encoding use flattened polynomial tree structures (a tree encodes in an 1-dimensional array), which are convenient to pass from C to MAPLE.

8.3 Bivariate Solver

The first application we used to evaluate our framework is the solving of bivariate polynomial systems by means of triangular decompositions. We consider two bivariate polynomials F_1 and F_2, with ordered variables $X_1 < X_2$ and with coefficients in a field $\mathbb{K}$. We assume that $\mathbb{K}$ is perfect; in our experimentation $\mathbb{K}$ is a prime field whose characteristic is a machine word size prime.

We rely on an algorithm introduced in [80] and based on the following well-known fact [11]. The common roots of F_1 and F_2 over an algebraic closure $\overline{\mathbb{K}}$ of $\mathbb{K}$ are “likely” to be described by the common roots of a system with a triangular shape:

$$\left\{ \begin{array}{rcl} T_1(X_1) & = & 0 \\ T_2(X_1, X_2) & = & 0 \end{array} \right.$$

such that the leading coefficient of T_2 w.r.t. X_2 is invertible modulo T_1; moreover the degree of T_2 w.r.t. X_2 is “likely” to be 1. For instance, the system

$$\left\{ \begin{array}{rcl} X_1^2 + X_2 + 1 & = & 0 \\ X_1 + X_2^2 + 1 & = & 0 \end{array} \right.$$

is *solved* by the triangular system

$$\begin{cases} X_1^4 + 2X_1^2 + X_1 + 2 &= 0 \\ X_2 + X_1^2 + 1 &= 0. \end{cases}$$

In general, though, more complex situations can arise, where more than one triangular system is needed. The goal of this section is to show that this algorithm can easily be implemented in our framework while providing high-performance. Section 8.3.2 at Page 121 and Section 8.3.3 at Page 122 contain the algorithm and the corresponding code, respectively.

8.3.1 Subresultant sequence and GCD sequence

In Sections 2.3.4 and 2.3.5 at Page 22, We have studied *subresultant* theory and *regular GCD*. Here we define *subresultant sequence* and GCD sequence in the bivariate case.

Subresult sequence. In Euclidean domains such as $\mathbb{K}[X_1]$, polynomial GCD's can be computed by the Euclidean algorithm and by the subresultant algorithm (we refer here to the algorithm presented in [32]). Consider next more general rings, such as $\mathbb{K}[X_1, X_2]$. Assume F_1, F_2 are non-constant polynomials with $\deg(F_1, X_2) \geq \deg(F_2, X_2)$, and $\deg(F_2, X_2) = q$. The polynomials computed by the subresultant algorithm form a sequence, called the *subresultant chain* of F_1 and F_2 and denoted by $\mathrm{src}(F_1, F_2)$. This sequence consists of $q+1$ polynomials, starting at $\mathrm{lc}(F_2, X_2)^{\delta}\, F_2$, with $\delta = \deg(F_1, X_2) - \deg(F_2, X_2)$, and ending at $R_1 := \mathsf{res}(F_1, F_2)$, the resultant of F_1 by F_2 w.r.t. X_2. We write this sequence $S_q, \ldots, S_0$ where the polynomial $S_j := S_j(F_1, F_2)$ is called the *subresultant* (*of* F_1, F_2) *of index* j. Let j be an index such that $0 \leq j \leq q$. If S_j is not zero, it turns out that its degree is at most j and S_j is said *regular* when $\deg(S_j, X_2) = j$ holds.

The subresultant chain of F_1 and F_2 satisfies a fundamental property, called the *block structure*, which implies the following fact: if the subresultant S_j of index j, with $j < \deg(F_2, X_2) - 1$, is not zero and not regular, then there exists a non-zero subresultant S_i with index $i < j$ such that S_i is regular, has the same degree as S_j and for all $i < \ell < j$ the subresultant S_ℓ is null.

The subresultant chain of F_1 and F_2 satisfies another fundamental property, called the *specialization property*, which plays a central in our algorithm. Let Φ be a homomorphism from $\mathbb{K}[X_1, X_2]$ to $\overline{\mathbb{K}}[X_2]$, with $\Phi(X_1) \in \overline{\mathbb{K}}$. Assume $\Phi(a) \neq 0$ where

$a = \text{lc}(f_1, X_2)$. Then we have:

$$\Phi(S_j(F_1, F_2)) = \Phi(a)^{q-k} S_j(\Phi(F_1), \Phi(F_2)) \tag{8.1}$$

where $q = \deg(F_2, X_2)$ and $k = \deg(\Phi(F_2), X_2)$.

GCD sequence Let $T_1 \in \mathbb{K}[X_1] \setminus \mathbb{K}$ and $T_2 \in \mathbb{K}[X_1, X_2] \setminus \mathbb{K}[X_1]$ be two polynomials. Note that T_i has a positive degree in X_i, for $i = 1, 2$. The pair $\{T_1, T_2\}$ is a *regular chain* if the leading coefficient $\text{lc}(T_2, X_2)$ of T_2 in X_2 is invertible modulo T_1. By definition, the set $\{T_1\}$ is also a regular chain. For simplicity, we will require T_1 to be squarefree, which has the following benefit: the residue class ring $\mathbb{L} = \mathbb{K}[X_1]/\langle T_1 \rangle$ is a direct product of fields. For instance, with $T_1 = X_1(X_1 + 1)$, we have:

$$\begin{array}{rcl} \mathbb{K}[X_1]/\langle T_1 \rangle & \simeq & \mathbb{K}[X_1]/\langle X_1 \rangle \oplus \mathbb{K}[X_1]/\langle X_1 + 1 \rangle \\ & \simeq & \mathbb{K} \oplus \mathbb{K}. \end{array}$$

Let $F_1, F_2, G \in \mathbb{K}[X_1 X_2]$ be non-zero. We say G is a *regular GCD* of F_1, F_2 modulo T_1 if the following conditions hold:

1. $\text{lc}(G, X_2)$ is invertible modulo T_1,
2. there exist $A_1, A_2 \in \mathbb{K}[X_1, X_2]$ such that $G \equiv A_1 f_1 + A_2 f_2 \mod T_1$,
3. if $\deg(G, X_2) > 0$ then G divides F_1 and F_2 in $\mathbb{L}[X_2]$.

The polynomials F_1, F_2 may not have a regular GCD in the previous sense. However the following holds.

Proposition 7. *There exists polynomials $A_1, \ldots, A_e$ in $\mathbb{K}[X_1]$ and polynomials $B_1, \ldots, B_e$ in $\mathbb{K}[X_1, X_2]$ such that the following properties hold:*

- *the product $A_1 \cdots A_e$ equals T_1,*
- *for all $1 \le i \le e$, the polynomials B_i is a regular GCD of F_1, F_2 modulo A_i.*

The sequence $(A_1, B_1), \ldots, (A_e, B_e)$ is called a GCD sequence *of F_1 and F_2 modulo T_1.*

Consider for instance $T_1 = X_1(X_1 + 1)$,

$$F_1 = X_1 X_2 + (X_1 + 1)(X_2 + 1) \quad \text{and} \quad F_2 = X_1(X_2 + 1) + (X_1 + 1)(X_2 + 1).$$

Then $(X_1, X_2 + 1), (X_1 + 1, 1)$ is a GCD sequence of F_1 and F_2 modulo T_1.

8.3.2 Algorithm

Recall that we aim at computing the set $V(F_1, F_2)$ of the common roots of F_1 and F_2 over $\overline{\mathbb{K}}$. For simplicity, we assume that both F_1 and F_2 have a positive degree in X_2; we define $h_1 = \mathrm{lc}(f_1, X_2)$, $h_2 = \mathrm{lc}(f_2, X_2)$ and $h = \mathrm{gcd}(h_1, h2)$. Recall also that R_1 denotes the resultant of F_1 and F_2 in X_2. Since h divides R_1, we define R_1' to be the quotient of the squarefree part of R_1 by the squarefree part of h. Our algorithm relies on the following observation.

Theorem 1. *Assume that $V(F_1, F_2)$ is finite and not empty. Then R_1' is not constant. Moreover, for any GCD sequence $(A_1, B_1), \ldots, (A_e, B_e)$ of F_1 and F_2 modulo R_1', we have*

$$V(F_1, F_2) = \bigcup_{i=1}^{i=e} V(A_i, B_i) \cup V(h, F_1, F_2). \tag{8.2}$$

and for all $1 \le i \le e$ the polynomial B_i has a positive degree in X_2 and thus $V(A_i, B_i)$ is not empty.

This theorem implies that the points of $V(F_1, F_2)$ which do not cancel h can be computed by means of one GCD sequence computation. This is the purpose of Algorithm 9. The entire set $V(F_1, F_2)$ is computed by Algorithm 10.

Algorithm 9 Modular Generic Solve

Input: F_1, F_2 as in Theorem 1.

Output: $(A_1, B_1), \ldots, (A_e, B_e)$ as in Theorem 1.

```
ModularGenericSolve2($F_1, F_2, h$) ==
(1)  Compute src($F_1, F_2$)
(2)  Let ${R_1}'$ be as in Theorem 1
(3)  $i := 1$
(4)  while ${R_1}' \notin \mathbb{K}$ repeat
(5)      Let $S_j \in$ src($F_1, F_2$) regular with $j \ge i$ minimum
(6)      if $\mathrm{lc}(S_j, X_2) \equiv 0 \mod {R_1}'$
             then $i := i + 1$; goto (5)
(7)      $G := \mathrm{gcd}({R_1}', \mathrm{lc}(S_j, X_2))$
(8)      if $G \in \mathbb{K}$
             then output $({R_1}', S_j)$; exit
(9)      output $({R_1}'$ quo $G, S_j)$
(10)     ${R_1}' := G$; $i := i + 1$
```

The following comments justify Algorithm 9 and are essential in view of our implementation. In Step (1) we compute the subresultant chain of F_1, F_2 in the following lazy fashion:

1. $B := 2d_1d_2$ is a bound for the degree of R_1, where $d_1 = \max(\deg(F_i, X_1))$ and $d_2 = \max(\deg(F_i, X_2))$. We evaluate F_1 and F_2 at $B+1$ different values of X_1, say $x_0, \ldots, x_B$, such that none of these specializations cancels $\mathrm{lc}(F_1, X_2)$ or $\mathrm{lc}(F_2, X_2)$.

2. For each $i = 0, \ldots, B$, we compute the subresultant chain of $F_1(X_1 = x_i, X_2)$ and $F_2(X_1 = x_i, X_2)$.

3. We interpolate the resultant R_1 and do not interpolate any other subresultants in $\mathrm{src}(F_1, F_2)$.

In Step (5) we consider S_j the regular subresultant of F_1, F_2 with minimum index j greater or equal to i. We view S_j as a "candidate GCD" of F_1, F_2 modulo R_1' and we interpolate its leading coefficient w.r.t. X_2 only. In Step (6) we test whether $\mathrm{lc}(S, X_2)$ is null modulo R_1'; if this is the case, then it follows from the block structure property that S_j is null modulo R_1' and we go to the next candidate. In Step (8), if $G \in \mathbb{K}$ then we have proved that S_j is a GCD of F_1, F_2 modulo R_1'; in this case we interpolate S_j completely and return the pair (R_1', S_j). In Steps (9)-(10) $\mathrm{lc}(S_j, X_2)$ has been proved to be a zero-divisor. Since R_1' is squarefree, we apply the *D5 Principle* and the computation splits into two branches:

1. $\mathrm{lc}(S_j, X_2)$ is invertible modulo R_1' quo G, so we output the pair $(R_1' \text{ quo } G, S_j)$

2. $\mathrm{lc}(S, X_2) = 0 \bmod G$; we go to the next candidate.

The following comments justify Algorithm 10. Recall that $V(F_1, F_2)$ is assumed to be non-empty and finite. Steps (1)-(2) handle the case where one input polynomial is univariate in X_1; the only motivation of the trick used here is to keep pseudo-code simple. Step (4) computes the points of $V(F_1, F_2)$ which do not cancel h. From Step (6) one computes the points of $V(F_1, F_2)$ which do cancel h, so we replace F_1, F_2 by their reductums w.r.t. X_2. In Steps (8)-(10) we filter out the solutions computed at Step (7), discarding those which do not cancel h.

8.3.3 Implementation

We explain now how Algorithms 9 and 10 are implemented in MAPLE interpreted code, using the functions of the `modpn` library. We start with Algorithm 9. The

Algorithm 10 Modular Solve

Input: F_1, F_2 as in Theorem 1.

Output: regular chains $(A_1, B_1), \ldots, (A_e, B_e)$ such that $V(F_1, F_2) = \bigcup_{i=1}^{i=e} V(A_i, B_i)$.

```
 ModularSolve2($F_1, F_2$) ==
(1)  **if** $F_1 \in \mathbb{K}[X_1]$ **then return** ModularSolve2($F_1 + F_2, F_2$)
(2)  **if** $F_2 \in \mathbb{K}[X_1]$ **then return** ModularSolve2($F_1, F_2 + F_1$)
(3)  $h := \gcd(\mathrm{lc}(F_1, X_2), \mathrm{lc}(F_2, X_2))$
(4)  $G :=$ ModularGenericSolve2($F_1, F_2, h$)
(5)  **if** $h = 1$ **return** $G$
(6)  $(F_1, F_2) := (\mathrm{reductum}(F_1, X_2), \mathrm{reductum}(F_2, X_2))$
(7)  $D :=$ ModularSolve2($F_1, F_2$)
(8)  **for** $(A(X_1), B(X_1, X_2)) \in D$ **repeat**
(9)      $g := \gcd(A, h)$
(10)     **if** $\deg(g, X_1) > 0$ **then** $G := G \cup \{(g, B)\}$
(11) **return** $G$
```

dominant cost is at Step (1) and it is desirable to perform this step entirely at the C level in one "function call". On the other hand the data computed at Step (1) must be accessible on the Maple side, in particular at Step (5). Recall that the only structured data that the C and Maple levels can share are arrays. Fortunately, there is a natural efficient method for implementing Step (1) under these constraints:

- We represent F_1 (resp. F_2) by a $(B + 1) \times d_2$ array (or "cube") C_1 (resp. C_2) where $C_1[i, j]$ (resp. $C_2[i, j]$) is the coefficient of F_1 (resp. F_2) of X_2^i evaluated at x_j; if F_1 (resp. F_2) is given over the monomial basis of $\mathbb{K}[X_1, X_2]$, then the "cube" C_1 (resp. C_2) is obtained by fast evaluation techniques.

- For each $i = 0, \ldots, B$, the subresultant chain of $F_1(X_1 = x_i, X_2)$ and $F_2(X_1 = x_i, X_2)$ is computed and stored in an $(B + 1) \times d_2 \times d_2$ array, that we call "Scube"; this array is allocated on the Maple side and is available at the C level without any data conversions.

- The resultant R_1 of (F_1 and F_2 w.r.t. X_2) is obtained from the "Scube" by fast interpolation techniques.

In Step (5) the "Scube" is passed to a C function which computes the index j and interpolates the leading coefficient $\mathrm{lc}(S_j, X_2)$ of S_j, the candidate GCD. Testing whether $\mathrm{lc}(S_j, X_2)$ is zero or invertible modulo R_1' is done at the Maple level using the `RecDen` module. Finally, in Step (8), when $\mathrm{lc}(S_j, X_2)$ has been proved to be invertible modulo R_1', the "Scube" is passed to a C function in order to interpolate S_j.

The implementation of Algorithm 10 is much more straightforward, since the operation ModularSolve2 consists mainly of recursive calls and calls to ModularGenericSolve2. The only place where computations take place "locally" is at Step (9) where the `RecDen` module is called for performing GCD computations.

8.4 Two-equation Solver and Invertibility Test

In this section, we present the two other applications used to evaluate the framework reported in Section 8.2. In Subsection 8.4.1, we specify the main subroutines on which these algorithms rely; we also include there the specifications of the invertibility test for convenience. The top-level algorithms are presented in Subsections 8.4.2 and 8.4.3.

As we shall see in Section 8.5 at Page 127, under certain circumstances, the data conversions implied by the calling of subroutines can become a bottleneck. It is thus useful to have a clear picture of these subroutines.

In this chapter, however, we do not assume a preliminary knowledge on triangular decomposition algorithms. To this end, the presentation of our bivariate solver in Section 8.3 at Page 118 was relatively self-contained, while omitting proofs; this was made easy by the bivariate nature of this application. In this section, we deal with polynomials with an arbitrary number of variables. In Section 2.3 at Page 2.3 we have introduced the notion of a *regular chain* and that of a *regular GCD (modulo a regular chain)* for bivariate polynomials. In the sequel, we rely on "natural" generalizations of these notions: we recall them briefly and refer to [9, 22] for introductory presentations.

8.4.1 Subroutines

Note that we restrict ourselves here to zero-dimensional regular chains. In this setting, observe that a normalized regular chain is a lexicographical Gröbner basis. In the specification of our subroutines below, we denote by T a normalized regular chain and p, q polynomials in $\mathbb{K}[X_1, \ldots, X_n]$. We reuse the notations $\mathrm{mvar}(p)$, *initp*, $\mathrm{NormalForm}(p, T)$, $\mathrm{Normalize}(p, T)$ and $\mathrm{RegularGcd}(p, q, T)$ as defined in Section 7.2 at Page 94 and add two more notations as following:

$\mathrm{IsInvertible}(p, T)$**:** returns pairs $(p_1, T^1), \ldots, (p_e, T^e)$ where $p_1, \ldots, p_e$ are polynomials and $T^1, \ldots, T^e$ are normalized regular chains, such that $V(T) = V(T^1) \cup \cdots \cup$

$V(T^e)$ holds and such that for all $i = 1, \ldots, e$, the polynomial p_i is either null or invertible modulo T^i and $p \equiv p_i \mod T^i$. The algorithm and implementation of this operation are described in Section 8.4.3 at Page 126.

$T_{<v}, T_v, T_{>v}$: these denote respectively the polynomials in T with main variable less than v, the polynomial in T with main variable v and the polynomials in T with main variable greater than v, where $v \in \{X_1, \ldots, X_n\}$.

8.4.2 Two-equation solver

Let $F_1, F_2 \in \mathbb{K}[X_1, \ldots, X_n]$ be non-constant polynomials with MainVariable(F_1) = MainVariable$(F_2) = X_n$. We assume that $R_1 = \mathsf{res}(F_1, F_2, X_n)$ is non-constant. Algorithm 11 below is simply the adaptation of Algorithm 9 to the case where F_1, F_2 are n-variate polynomials instead of bivariate polynomials. The relevance of Algorithm 11 to our study is based on the following observation.

As we shall see in Section 8.5, the implementation of Algorithm 9 at Page 121 in our framework is quite successful. It is, therefore, natural to check how these results are affected when some of its parameters are modified. A natural parameter is the number of variables. Increasing it makes some routine calls more expensive and could raise some overheads. In broad terms, Algorithm 11 computes the "generic solutions" of F_1, F_2. Formally speaking, it computes regular chains $T^, \ldots, T^e$ such that we have

$$V(F_1, F_2) = \overline{W(T^1)} \cup \cdots \cup \overline{W(T^e)} \cup V(F_1, F_2, h_1 h_2) \tag{8.3}$$

where $h_1 h_2$ is the product Initial(F_1)Initial(F_2) and where $\overline{W(T^i)}$ denotes the Zariski closure of the quasi-component of T^i. It is out of the scope of this chapter to expand on the theoretical background of Algorithm 11; this can be found in [75]. Instead, as mentioned above, our goal is to measure how Algorithm 9 scales when the number of variable increases.

The implementation plan of Algorithm 11 is exactly the same as that of Algorithm 9. In particular, the computations of squarefree parts, primitive parts and the GCDs at Steps (1) and (7) are performed on the MAPLE side, whereas the subresultant chain src(F_1, F_2) is computed on the C side. In the complexity analysis of Algorithm 11 the dominant cost is given by src(F_1, F_2) and a natural question is whether this is verified experimentally. If this is the case, this will be a positive point for our framework.

Algorithm 11 Modular Generic Solve N-variable

Input: $F_1, F_2 \in \mathbb{K}[X_1, \ldots, X_n]$ with $\deg(F_1, X_n) > 0, \deg(F_2, X_n) > 0$ and $\mathsf{res}(F_1, F_2, X_n) \notin \mathbb{K}$.

Output: $T^1 = (A_1, B_1), \ldots, T^e = (A_e, B_e)$ as in (8.3).

```
ModularGenericSolveN($F_1, F_2$) ==
(1)  **Compute** $\mathrm{src}(F_1, F_2)$; $R_1 := \mathsf{res}(F_1, F_2, X_n)$
     $h := \gcd(\mathrm{Initial}(F_1), \mathrm{Initial}(F_2))$
(2)  $R_1' :=$ squarefreePart($R_1$) quo squarefreePart($h$)
     $v :=$ MainVariable($R_1$);
     $R_1' :=$ primitivePart($R_1, v$)
(3)  $i := 1$
(4)  **while** $\deg(R_1', v) > 0$ **repeat**
(5)       **Let** $S_j \in \mathrm{src}(F_1, F_2)$ regular with $j \geq i$ minimum
(6)       **if** $\mathrm{lc}(S_j, X_n) \equiv 0 \mod R_1'$
               **then** $i := i + 1$; **goto** (5)
(7)       $G := \gcd(R_1', \mathrm{lc}(S_j, X_2))$
(8)       **if** $\deg(G, v) = 0$
               **then output** $(R_1', S_j)$; **exit**
(9)       **output** ($R_1'$ quo $G, S_j$)
(10)      $R_1' := G$; $i := i + 1$
```

8.4.3 Invertibility test

Invertibility test modulo a regular chain is a fundamental operation in algorithms computing triangular decompositions. The precise specification of this operation has been given in Section 8.4.1 at Page 124. In broad terms, for a regular chain $T = T_1(X_1), \ldots, T_n(X_1, \ldots, X_n)$ and a polynomial p the call IsInvertible(p, T) "separates" the points of $V(T)$ that cancel p from those which do not. The output is a list of pairs $(p_1, , T^1), \ldots, (p_e, T^e)$ where $p_1, \ldots, p_e$ are polynomials and $T^1, \ldots, T^e$ are normalized regular chains: the points of $V(T)$ which cancel p are given by the T^i's such that p_i is null.

Algorithm 12 is in the spirit of those in [76, 75] implementing this invertibility test. However, it offers more opportunities for using modular methods and fast polynomial arithmetic. The trick is based on the following result (Theorem 1 in [22]): the polynomial p is invertible modulo T if and only if the iterated resultant of p with respect to T is non-zero. Iterated resultants can be computed efficiently by evaluation and interpolation, following the same implementation techniques as those of Algorithm 9. Our implementation of Algorithm 12 employs this strategy. In particular the resultant r (computed at Step (4)) and the regular GCDs (g, D) (computed at Step (7)) are obtained from the same "Scube".

The calls to NormalForm(p, T) (Step (1)), NormalForm$(\mathrm{quo}(T_v, g), D)$ (Step (10)) and Normalize(g, D) (Step (8)) are performed on the C side: they require the conversions of regular chains encoded by Maple polynomials to regular chains encoded by C-Cube polynomials. If the call to RegularGcd(p, T_v, C) (Step (7)) outputs many cases, that is, if computations split in many branches, these conversions could become a bottleneck as we shall see in Section 8.5. Finally, for simplicity, we restrict Algorithm 12 to the case of (zero-dimensional) regular chains generating radical ideals.

Algorithm 12 Invertibility Test

Input: T a normalized regular chain generating a radical ideal and p a polynomial, both in $\mathbb{K}[X_1, \ldots, X_n]$.

Output: See specification in Section 8.4.1 at Page 124.

```
IsInvertible($p, T$) ==
(1)  $p$ := NormalForm($p, T$)
(2)  if $p \in \mathbb{K}$ then return $[p, T]$
(3)  $v$ := mvar($p$)
(4)  $r$ := res($p, T_v, v$)
(5)  for $(q, C) \in$ IsInvertible($r, T_{<v}$) repeat
(6)      if $q \neq 0$ then output $[p, C \cup T_v \cup T_{>v}]$
(7)      else for $(g, D) \in$ RegularGcd($p, T_v, C$) repeat
(8)          $g$ := Normalize($g, D$)
(9)          output $[0, D \cup g \cup T_{>v}]$
(10)         $q$ := NormalForm(quo($T_v, g$), $D$)
(11)         if $\deg(q, v) \neq 0$ then output $[p, D \cup q \cup T_{>v}]$
```

8.5 Experiments

We discuss here the last two questions mentioned in the Section 3:

- Can our implementation based on the Maple/C hybrid model outperforms other highly efficient systems?
- Does the performance of the implementation of the new algorithms comply with the theoretical complexity?

Our answer for the first one is "yes, if the application is well suitable for our framework". As shown below, we have improved the performance of triangular decomposition based computation in MAPLE. On the example of the invertibility test, our code is competitive with MAGMA and often outperforms it. The answer to the last question is "yes, the performance does comply with the complexity analysis", though there are some interferences due to the overhead of the data conversion as discussed in Section 8.2.3 at Page 116.

We report two sets of statistic data. For the first set, we compare the performance of our new implementations with their existing counterparts in MAPLE or MAGMA (see Subsections 8.5.1, 8.5.2 and 8.5.3). For the second set, we profile the implementation of our new polynomial solving algorithms to determine for which kind of algorithms our framework is the most suitable one. The profiling information for invertibility test is reported in the Section 8.5.3; for the solvers is reported in Section 8.5.3. In all examples, the base field is $\mathbb{Z}/p\mathbb{Z}$, where p is a machine-word size FFT prime. In the profiling samples, we only calculate the MAPLE side conversion time and ignore the C side since the latter one is mostly negligible.

8.5.1 Bivariate solver

In Figures 8.2, 8.3, 8.4 and 8.5, we consider two bivariate polynomials F_1 and F_2, with ordered variables $X_1 < X_2$ and with coefficients in a field $\mathbb{K}$. In our experimentation $\mathbb{K}$ is a prime field whose characteristic is a prime number, and its size is less than 32 bit.

The benchmark shown in Figure 8.2 is comparing the performance of libraries all from Maple: "Triangularize" is the solver from Maple *RegularChains* library; "Basis" is the solver from the Maple *Groebner* library; "Fast Triangularize" is the solver from our Maple FastArithmeticTools library. Actually we have also tested the solver "Solve" from the *Groebner* library which is significantly slower than the other ones. Thus, we list its data and all the previous ones in Table 8.1. The "degree" in Figure 8.2 (also "deg." in Table 8.1) is the total degree of each input random dense polynomial. We compare the computational time. To make the figure more readable, we extract the comparison between "Basis" and our fast solver into Figure 8.3. In Table 8.1, "Basis", "Solve", "Triang" and "FTriang" are short for "Basis from Groebner", "Solve from Groebner", "Triangularize from RegularChains" and "Fast Triangularize from FastArithmeticTools". As shown, our solver from FastArithmeticTools library is the fastest one. It approximately 20 times faster than lex "Basis" on our biggest input

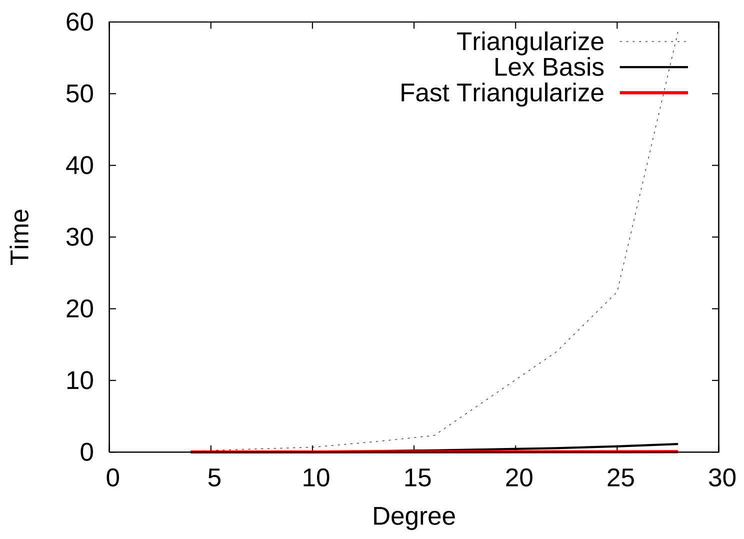


Figure 8.2: Bivariate solver dense case.

example. While the input size increasing, the ratio of speed-up is more significant. Recall that the major sub-algorithms of the bivariate solver are *subresultant chain* and *regular gcd*, thus the high performance is also relying the implementation of these two sub-algorithms.

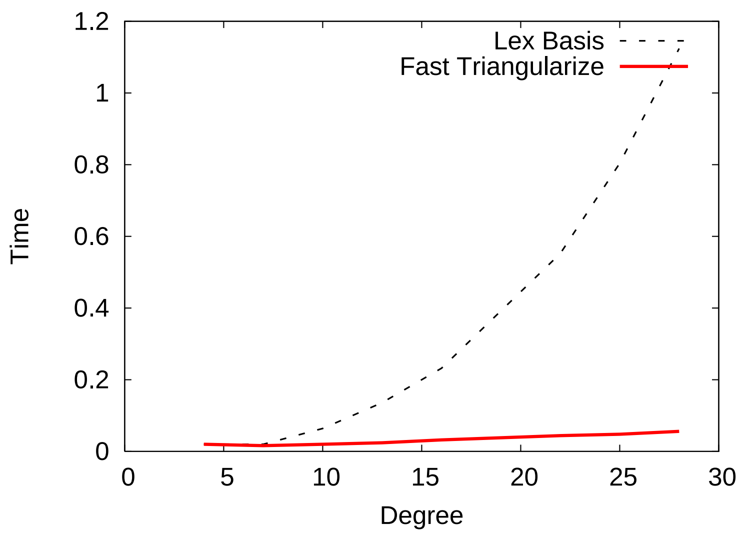


Figure 8.3: Bivariate solver dense case.

The benchmark shown in Figure 8.4 uses the same parameter as defined in Figure 8.2. Namely the "degree" is the total degree of each input polynomial. However

deg	Basis	Solve	Triang	FTriang
4	0.020	0.040	0.152	0.020
7	0.020	0.580	0.424	0.016
10	0.064	3.892	0.680	0.020
13	0.136	16.557	1.424	0.024
16	0.232	55.939	2.324	0.032
22	0.552	416.466	13.972	0.044
25	0.804	1116.045	22.346	0.048
28	1.124	2162.271	58.695	0.056

Table 8.1: Bivariate solver dense case.

instead of using dense random polynomials, we generate specific "split" examples in terms of non-equiprojectability in triangular decomposition. As shown in the figures, our fast solver is significant faster than the other two. We also provide the data in Table 8.2 for this benchmark. At the total degree 23 our fast solver is approximately 100 times faster than the "Lex Basis" which is the second fastest one.

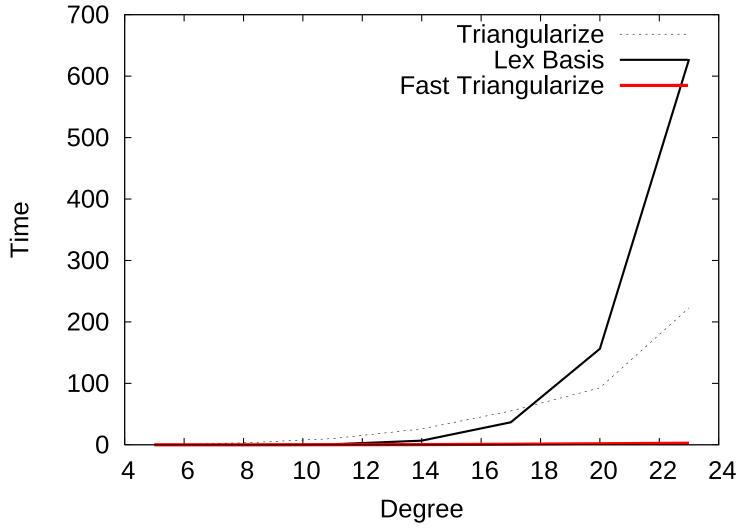


Figure 8.4: Bivariate solver non-equiprojectable case.

Figure 8.5 is generated based on the data from Table 8.3. Here we compare our *Fast Regularize* (FTriang in the table) with Magma's implementation: one is *Gröbner Basis* (Abbr. GB in the table); the other one is *Triangular Decomposition* (Abbr. Triang in the table). The input polynomials which generate a zero-dimensional ideal are designed with many split steps during the solving. Again, our solver is the fastest

deg	Basis	Solve	Triang	FTriang
5	0.014	0.080	0.616	0.016
8	0.152	3.004	3.200	0.048
11	0.908	44.407	10.049	0.124
14	6.837	246.839	25.902	0.428
17	36.581	1266.958	55.014	0.938
20	156.245	6296.301	92.662	1.740
23	627.551	21758.120	222.897	2.625

Table 8.2: Bivariate solver non-equiprojectable, us vs. Maple.

one in terms of running time. For the non-equiprojectable examples, our solver outperforms Magma's even more significantly.

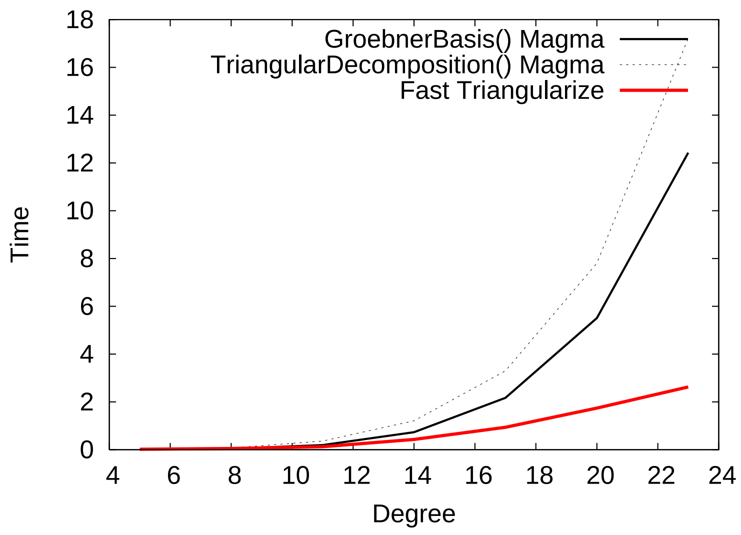


Figure 8.5: Bivariate solver non-equiprojectable case.

8.5.2 Two-equation solver

We consider now the solver of Algorithm 11. For a machine-word size FFT prime p, we consider a pair of trivariate polynomials $F_1, F_2 \in \mathbb{Z}/p\mathbb{Z}[X_1, X_2, X_3]$ of total degrees d_1, d_2. We compare our code for `ModularGenericSolveN` (Algorithm 11) to the `Triangularize` function of `RegularChains` library. In Magma there are several ways to obtain similar outputs: either by a triangular decomposition in $\mathbb{K}(X_1)[X_2, X_3]$ (triangular decompositions in Magma require the ideal to have dimension zero) or

deg	GB (Magma)	Triang (Magma)	FTriang (Maple)
5	0.010	010	0.016
8	0.040	070	0.048
11	0.190	0.360	0.124
14	0.730	1.210	0.428
17	2.170	3.300	0.938
20	5.510	7.810	1.740
23	12.430	17.220	2.625

Table 8.3: Bivariate solver non-equiprojectable case.

by computing the GCD of the input polynomials modulo their resultant (assuming that this resultant is irreducible).

d_1	d_2	Maple		Magma	
		`Triangularize`	`ModularGenericSolveN`	Tr. dec.	Resultant + GCD
2	4	0.3	0.06	0.03	0.01
4	4	1.4	0.15	0.03	0.3
6	4	25	0.27	0.7	12
8	4	257	0.52	6.9	155
10	4	1933	1.02	46.7	1012

Table 8.4: Solving two equations in three variables

Table 8.4 summarizes the timings (in seconds) obtained on random dense polynomials by the approaches above (in the same order). Our new code performs significantly better than all other ones. For completeness, we add that on these examples, computing a lexicographic Gröbner basis in $\mathbb{K}[X_1, X_2, X_3]$ in Magma takes time similar to that of the triangular decomposition.

8.5.3 Invertibility test

We continue with the operation IsInvertible. Designing good test suites for this algorithm is not easy: one of the main reasons for the high technicality of these algorithms is that various kinds of degeneracies need to be handled. Using random systems, one typically does not meet such degeneracies: a random polynomial is invertible modulo a random regular chain. Hence, if we want our test suite to address more than the generic case of our algorithms, the examples must be constructed ad-hoc.

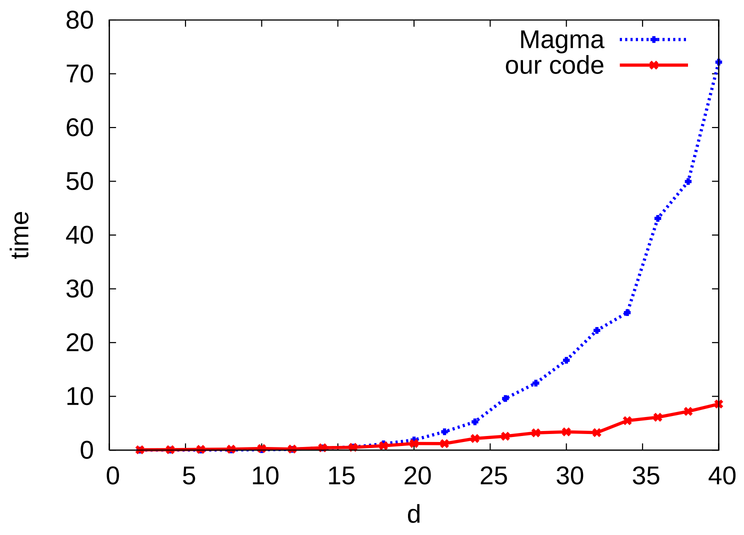


Figure 8.6: Bivariate case: timings, $p = 0.98$.

Here, we report on such examples for bivariate and trivariate systems. We construct our regular chain T by Chinese Remaindering, starting from smaller regular chains $T^{(i)}$ of degree 1 or 2. Then, we interpolate a function f from its values $f^{(i)} = f \bmod T^{(i)}$, these values being chosen at random. The probability p that $f^{(i)} \neq 0$ is a parameter of our construction. We generated families of examples with $p = 0.5$, for which we expect that the invertibility test of f will generate a large number of splittings. Other families have $p = 0.98$, for which few splittings should occur.

The bivariate case. Figure 8.6 gives results for bivariate systems with $p = 0.98$ and $d = d_1 = d_2$ in abscissa. We compare our implementation with Magma's counterpart, that relies on the functions `TriangularDecomposition` and `Saturation` (in general, when using Magma, we always choose the fastest available solution). We also tested the case $p = 0.5$ in Figure 8.7. Figure 8.8 profiles the percentage of the conversion time with respect to the total computation time, for the same set of samples. With $p = 0.98$, IsInvertible spends less time on conversions (around 60%) and has fewer calls to the Maple operations than with $p = 0.5$ (the conversion ratio with $p = 0.5$ reaches 83%).

The trivariate case. Table 8.5 uses trivariate polynomials as the input for IsInvertible, with $p = 0.98$; Table 8.6 has $p = 0.5$. Figure 8.9 profiles the conversion time spent on these samples. The conversion time increases dramatically along the input size. For the largest example, the conversion time reaches 85% of the total computation time. More than 5% of the time is spent on other Maple computations, so that

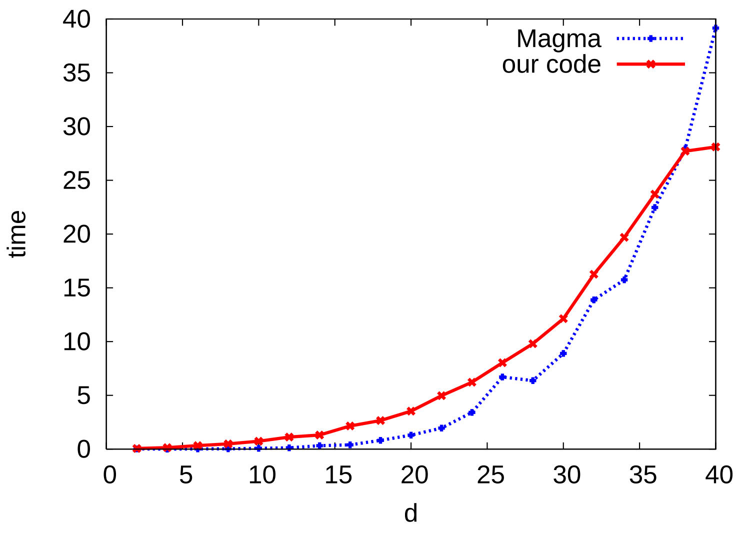


Figure 8.7: Bivariate case: timings, $p = 0.5$.

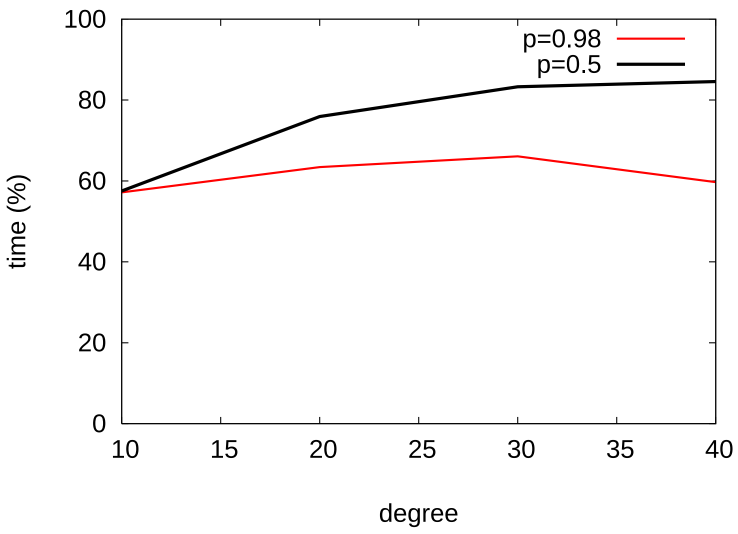


Figure 8.8: Bivariate case: time spent in conversions.

the real C computation costs less than 5%. We also provide the timing of the operation Regularize from the Maple `RegularChains` library. The pure Maple code, with no fast arithmetic, is several hundred times slower than our implementation.

The 5 variable case. We performed further tests between the Maple Regularize operation and our IsInvertible function, using random dense polynomials in 5 variables. IsInvertible is significantly faster than Regularize; the speedup reaches a factor of 300. Similar experiments with sparse polynomials give a speed-up of 100.

d_1d_2	d_3	Magma	Maple	
			Regularize	IsInvertible
4	3	0.000	1.199	0.091
12	6	0.020	6.569	0.281
24	9	0.050	24.312	0.509
40	12	0.170	73.905	1.293
60	15	0.550	172.931	1.637
84	18	1.990	450.377	5.581
112	21	5.130	871.280	9.490
144	24	12.830	1956.728	12.624
180	27	30.510	3621.394	23.564
220	30	62.180	6457.538	32.675
264	33	129.900	7980.241	89.184

Table 8.5: Trivariate case: timings, $p = 0.98$.

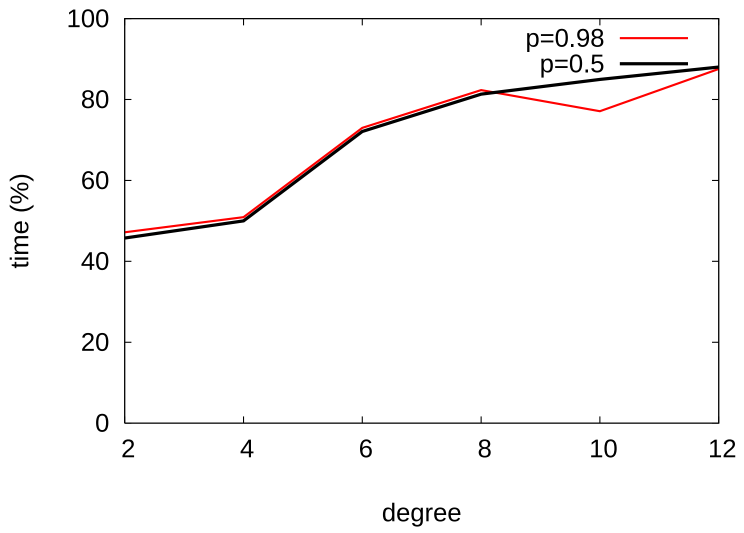


Figure 8.9: Trivariate case: time spent in conversions.

d_1d_2	d_3	Magma	Maple	
			Regularize	IsInvertible
4	3	0.010	0.773	0.199
12	6	0.020	4.568	0.531
24	9	0.040	17.663	1.082
40	12	0.150	47.767	2.410
60	15	0.480	126.629	5.023
84	18	1.690	284.697	10.405
112	21	4.460	632.539	19.783
144	24	10.960	1255.980	42.487
180	27	26.070	2328.012	69.736
220	30	58.700	4170.468	109.667
264	33	106.140	7605.915	191.514

Table 8.6: Trivariate case: timings, $p = 0.5$.

8.5.4 Profiling information for the solvers

We conclude this section with profiling information for the bivariate solver and the two-equation solver. The differences between these algorithms have noticeable consequences regarding profiling time.

Bivariate solver. For this algorithm, there is no risk of data duplication. The amount of data conversion is bounded by the size of the input plus the size of the output; hence we expect that data conversions cannot be a bottleneck. Third, the calls to Maple interpreted code simply perform univariate operations, thus we do not expect them to become a bottleneck either.

Table 8.7 confirms this expectation, by giving the profiling information for this algorithm. The input system is dense and contains 400 solutions. The computation using the `RecDen` package costs 49% of the total computation time. The C level subresultant chain computation spends around 34%, and the conversion time is less than 11%. With larger input systems, the conversion time reduces. For systems with 2,500 and 10,000 solutions, the C computation takes about 40% of the time; `RecDen` computations takes roughly 50%; other Maple functions take 5% and the conversion time is less than 5%.

The profiling information in Figure 8.10 also concerns the bivariate solver; there, the sample input intends to generate many splittings (we take $p = 0.5$, as in the examples in the previous subsection). The conversion time slowly increases but does not become the bottleneck (28% to 38%).

Operation	calls	time	time (%)
Subresultant chain	1	0.238	33.85
`Recden`	41	0.344	48.93
Conversions	17	0.076	10.81

Table 8.7: Bivariate solver: profiling, $p = 0.98$.

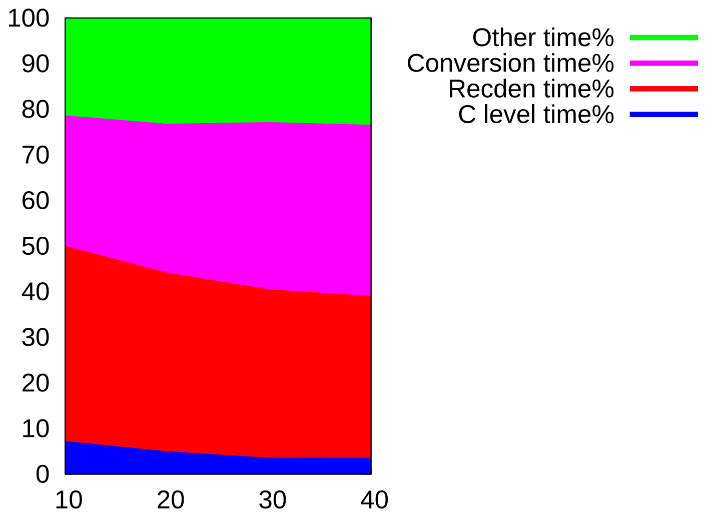


Figure 8.10: Bivariate solver: profiling, $p = 0.5$.

Two-equation solver. This algorithm has properties similar to the bivariate solver, except that the calls to interpreted code can be expensive since it involves multivariate arithmetic. Hence, we expect that the overhead of conversion is quite limited. Indeed, in Table 8.5.4, N is the number of variables and d_1, d_2 are the degrees of T_1, T_2 respectively 8.3. The C level computation is the major factor of the total computation time; it reaches 91% in case $N = 4,\ d_1 = 5,\ d_2 = 5$.

N	d_1	d_2	C (%)	MAPLE (%)	Conversion (%)
3	5	5	56.47	12.96	30.57
4	5	5	91.54	2.64	5.82
8	2	2	83.67	8.02	8.31

Table 8.8: Two-equation solver: profiling.

8.6 Summary

The answers to our main questions are mostly positive: we have obtained large performance improvements over existing Maple implementations, and often outperform Magma's. Still, some triangular decomposition algorithms are not perfectly suited to our framework. For instance, we implemented the efficiency-critical operations of IsInvertible in C, but the main algorithm itself in Maple. This algorithm may generate large amounts of "external" calls to the C functions, so the data conversion between Maple and C becomes a dominant cost. For this kind of algorithms, we suggest either to implement them purely in C or tune the algorithmic structure to avoid intensive data conversion;

Chapter 9

Multithreaded Parallel Implementation of Arithmetic Operations Modulo a Triangular Set

9.1 Overview

In Chapter 6 at Page 65 we have studied arithmetic operations for triangular families of polynomials, concentrating on multiplication in dimension zero what we called *modular multiplication*. As reported previously, this algorithm consists of two major operations: (1) polynomial multiplication, (2) modular reduction what we called *normal form* for convenience. In this chapter, we discuss the parallelization of these two operations.

When computing modulo a triangular set, multivariate polynomials are regarded recursively as univariate ones. This recursive data structure leads to several challenges for obtaining a high performance parallel implementation. The serial modular multiplication algorithm is reported in Chapter 6. Based the serial one, we have developed a parallel version of multi-dimensional fast Fourier transform to perform the polynomial multiplication step. We have also developed several versions of parallel *normal form*. Each parallel algorithm and its implementation will be reported in details in following sections.

The outline of this chapter is as following. In Section 9.2 at Page 140, we review the top-level algorithm **modular multiplication**. In Section 9.3 at Page 141, we first specify all the subroutines of the serial version of **modular multiplication**. Then, we develop their variants which is still in the serial mode whereas can better expose the parallelism. Finally, we illustrate the parallelization techniques of each sub-routine. In Section 9.4 at Page 148, we provide the benchmark result between the serial and parallel implementation of modular multiplication algorithm. The parallelized code has satisfactory speed-up, though still potential to be further tuned.

`NOTE: This chapter is written based on the published paper [66].`

9.2 Algorithms

In this section, we give more simplified definition of *modular multiplication* algorithm (see Chapter 6 the more detailed version).

Let $L_0 = \mathbb{K}$ be a commutative ring with a unit. Let B be a univariate polynomial in $\mathbb{K}[x]$, non-constant, monic and with degree $d > 1$. We aim at computing modulo B the product $A \in \mathbb{K}[x]$ of two polynomials reduced w.r.t. B, that is, with degree less than d. So, for simplicity, let us assume that A has degree $2d - 2$.

The quotient Q and the remainder R in the division of A by B can be computed as follows, using the trick of Cook-Sieveking-Kung [24, 87, 59]. We summarize this trick and refer to [43] for details. Let B^{-1} be the inverse of the reversal of B modulo x^{d-1}. Let $\overline{Q}$ be the product $\overline{A}B^{-1}$ computed modulo x^{d-1}, where $\overline{A}$ is the reversal of A. Then Q is the reversal of $\overline{Q}$ and we have $R = A - BQ$.

Consider now $\mathbf{T} = (T_1, \ldots, T_s)$ a set of non-constant polynomials in $\mathbb{K}[x_1, \ldots, x_s]$. Let d_i be the degree of T_i w.r.t. x_i, for all i. We say that T is a *triangular set* if for all i, the polynomial T_i lies in $\mathbb{K}[x_1, \ldots, x_i]$, is monic in x_i and is reduced with respect to $T_1, \ldots, T_{i-1}$, that is, for all $j = 1, \ldots, i-1$ the degree of T_i w.r.t. x_j is less than of d_j.

Let $1 \leq i \leq s$ and let $P \in \mathbb{K}[x_1, \ldots, x_s]$. The *normal form* of P w.r.t. $T_1, \ldots, T_i$, denoted by $\mathrm{NF}_i(P)$, is the the unique polynomial $R \in \mathbb{K}[x_1, \ldots, x_s]$ which is reduced w.r.t. $T_1, \ldots, T_i$, and congruent to P modulo the ideal $\langle T_1, \ldots, T_i \rangle$. Moreover, we define $\mathrm{NF}_0(P) = P$.

For $i = 1, \ldots, s$ we define $L_i = \mathbb{K}[x_1, \ldots, x_i]/\langle T_1, \ldots, T_i \rangle$, the residue class ring of $\mathbb{K}[x_1, \ldots, x_i]$ modulo $\langle T_1, \ldots, T_i \rangle$.

Our main goal is to implement arithmetic operations in all L_i, leading to *normal form* computations for polynomials in $\mathbb{K}[x_1, \ldots, x_s]$ modulo $\langle T_1, \ldots, T_i \rangle$. We summarize the algorithm in Chapter 6. We assume that, for all $1 \leq i \leq s$, the inverse T_i^{-1} of the reversal of T_i in $L_{i-1}[x_i]/\langle x_i^{d_i - 1} \rangle$ has been precomputed. Let $P \in \mathbb{K}[x_1, \ldots, x_s]$

be such that the degree of P w.r.t. x_i is at most $2d_i - 2$ for all $1 \leq i \leq s$. Then we compute $\mathrm{NF}_s(P)$ as follows:

Step 1 Let $P' := \mathrm{NF}_{s-1}(P)$.

Step 2 Let $\overline{P'}$ be the reversal of P' in $L_{s-1}[x_s]$. Let $\overline{P'} := \overline{P'} \mod x_s^{d_s-1}$ and let $\overline{Q} := \overline{P'}T_s^{-1} \mod x_s^{d_s-1}$.

Step 3 Let $\overline{Q} := \mathrm{NF}_{s-1}(\overline{Q})$.

Step 4 Let Q be the reversal of $\overline{Q}$ in $L_{s-1}[x_s]$. Let $R := P - QT_s$.

Step 5 Return $\mathrm{NF}_{s-1}(R)$.

For a polynomial F in $\mathbb{K}[x_1, \ldots, x_s]$, with positive degree w.r.t. x_s, we compute $\mathrm{NF}_{s-1}(F)$ as a "map" on its coefficients w.r.t. x_s.

We parallelize the computation of $\mathrm{NF}_s(P)$ at two levels. First, for degrees large enough, we perform the products in **Step 2** and **Step 4** by means of a parallel multidimensional FFT algorithm (see Section 2.1 at Page 8). From now on, let us regard these products as atomic operations. Secondly, we focus on the calls to the NF_{s-1} function performed at **Step 1**, **Step 3** and **Step 5**. Let G be the *task graph* or *instruction stream DAG* [15] associated with $\mathrm{NF}_s(P)$. One can use either a *depth-first* traversal or a *bottom-up level-by-level* traversal for G, leading to the two parallel schemes detailed in Section 9.3.3 at Page 146. Note that our task graph G is not a fork-join graph and the special techniques developed for this kind of task graphs, see for instance [89], do not apply here.

In fact, the structure of the algorithm implies several "global synchronisations". More precisely, before starting each of **Step 2**, **Step 3**, **Step 4** and **Step 5**, all threaded computations of the previous step must be completed. These constraints make the parallelization of our *normal form* computations more challenging than for more standard "divide & conquer" algorithms. See also [77] on this topic.

9.3 Implementation

9.3.1 Multidimensional FFT

We have already studied the serial multidimensional FFT algorithm in Section 2.1 at Page 8. Multidimensional FFT is a very nice application to parallelize on a SMP architecture, since the "small" DFTs/IDFTs performed on a given dimension have no data dependency to each other. Therefore, instead of computing these "small" DFTs/IDFTs one by one in a sequential setting, we create multiple threads and each of the threads will be in charge of an amount of "small" DFTs/IDFTs' computations. Since all the "small" DFTs/IDFTS have similar amount of workloads in a dense polynomial application, each thread will be in charge of a similar number of "small" DFTS/IDFTS.

In fact, when implementing multivariate polynomial multiplication in our sequential mode, we used two approaches. One is the above mentioned multidimensional FFT, the other is based on Kronecker's substitution. In this latter method, two input multivariate polynomials are mapped to univariate ones. Then, univariate FFT can be used to compute the polynomial multiplication. This implies that parallelizing Kronecker's substitution based FFT multiplication is actually, parallelizing a univariate FFT. We didn't try this direction based on three reasons. First, the multidimensional FFT is much easier to parallelize as we described before. Second, we implemented Truncated Fourier transform (TFT) [51], and replaced multidimensional FFT by multidimensional TFT in our package. This brings us a significant improvement of performance comparing to Kronecker's substitution method as reported in Section 9.4 at Page 148. Moreover, the multidimensional TFT has the same code structure as the multidimensional FFT, thus is easy to implement. Third, multidimensional FFT/TFT is more cache friendly comparing to Kronecker's substitution method for certain range of input [69].

Therefore, on a multi-processor architecture we prefer multidimensional FFT to Kronecker's substitution method. In addition, matrix transposition in multidimensional FFT also can be parallelized. We leave it as a future work, since the computation time of matrix transposition is generally a small portion of the whole computation.

9.3.2 Two traversal methods for normal form

By using the names defined in our pseudo-code in this section, we describe the *normal form* operation as follows. The *normal form* operation consists of two major operations **UniFastMod** and **NormalForm**. **NormalForm** is the "main" function which recursively reduces the coefficients of the input polynomial $f \in \mathbb{K}[x_1, x_2, \cdots, x_s]$. TS is the given triangular set, and s is the number of variables. In addition, we have following definition of operations for all pseudo-code in this section.

- $\mathrm{rev}_n(f)$ returns ${x_s}^n f(\frac{1}{x_s})$, where x_s is the main variable of f and $n \geq deg(f)$.

- $deg(f)$ returns the degree of f.
- $degree(f,\ i)$ returns the partial degree of f in x_i.
- $coef\ (f,\ i)$ fetches the i-th coefficient of f.

  In this chapter, *normal form* operation only applies to a dense multivariate polynomial f who is encoded in an one dimensional array. we define this operation as following. For the input polynomial f, we use a data representation based on Kronecker substitution [65, 69]. Namely, a dense multivariate polynomial will be encoded in an one dimensional array. The Kronecker map $U(f)$ is an array of element of $\mathbb{K}$.

$$\begin{array}{ll} U: & (x_1,\ x_2, \cdots,\ x_s) \longmapsto (x_1,\ {x_1}^{\delta_2}, \cdots,\ {x_1}^{\delta_s}) \\ where & \delta_1 = 1, \\ & \delta_i = \ \prod_{j=1}^{i-1} (degree(f,\ j)\ +\ 1) \end{array} \tag{9.1}$$

  Thus, $coef\ (f,\ i,\ s)$ returns the i-th slot of $U(f)$ regarded as an array where each slot has size of δ_s.

Algorithm 13 Normal Form

```
NormalForm $(f,\ TS,\ s)$

Input: $f \in \mathbb{K}[x_1, x_2, \cdots, x_s]$, $TS = \{T_1, T_2, \cdots, T_s\}$, with $T_i$ is monic.

Output: The normal form of $f$ w.r.t. $TS$.

1 if $(s == 0)$ return $f$
2 $d = deg(f)$
3 RC $(f,\ 0,\ d,\ TS,\ s-1)$
4 $f =$ UniFastMod$(f,\ TS,\ s)$
```

Each reduction step is performed by calling **UniFastMod**, namely a fast univariate division in $L_{s-1}[x_s]$. The function **RC** means to reduce each coefficient of a polynomial by calling **NormalForm** and it is an in-place operation.

As we mentioned above, a multivariate polynomial can be encoded by a tree structure. When reducing its coefficients, we need to have a tree traversal. The nested recursion in **NormalForm** performs a depth-first tree traversal. The other way is what we called "bottom-up level-by-level" (BULL) traversal. The pseudo functions

Algorithm 14 Fast Univariate Division

```
UniFastMod ($f$, $TS$, $s$)

1  $n \longleftarrow \deg f$
2  $m \longleftarrow \deg T_s$
3  if $n < m$ then
4    $q \longleftarrow 0$
5    $r \longleftarrow f$
6  else
7    $q \longleftarrow \mathrm{rev}_n(f)\, T_s^{-1} \mod x^{n-m+1}$
8    $q \longleftarrow \mathrm{rev}_{n-m}(q)$
9    RC($q$, 0, $n-m$, $TS$, $s-1$)
10   $w = T_s\, q$
11   RC($w$, 0, $n-m$, $TS$, $s-1$)
12   $r \longleftarrow f - w$
13 return $r$
```

Algorithm 15 Fast Coefficients Reduction

```
RC ($f$, $start$, $end$, $TS$, $s$)

1 for $i$ from start to end do
2   $coef$ ($f$, $i$)=NormalForm ($coef$ ($f$, $i$), $TS$, $s$)
```

Algorithm 16 Normal Form 2

```
NormalForm2 ($f$, $TS$, $s$)

1 if ($s == 0$) return $f$
2 $size = \prod_{j=1}^{s} (degree(f,\ j) + 1)$
3 $i = 2$
4 while ($i \leq s$) do
5   $ss = size \,/\, \prod_{j=1}^{i} (degree(f,\ j) + 1)$
6   RS ($f$, 0, $ss-1$, $TS$, $i$)
7   $i = i + 1$
```

Algorithm 17 Iterative Reduction

```
RS ($f$, $start$, $end$, $TS$, $s$)

1 for $i$ from start to end do
2   $coef(f,i,s)$=UniFastMod2($coef(f,i,s),TS,s$)
```

RS, **NormalForm2**, **UniFastMod2**, and **RC2** describe the computational steps for this method.

In brief, we suppose the input multivariate polynomial f is encoded in an one dimensional array by the Kronecker map U. The *size* of the array is $\prod_{j=1}^{s}(degree(f,\ j)\ +\ 1)$. We start the reduction steps at level 1. That is we view the given array as an array with *size* / $(degree(f,\ 1)\ +\ 1)$ slots. Each slot has size of $degree(f,\ 1)\ +\ 1$. Each slot actually is encoding an univariate polynomial in L_1. Then we reduce all slots by calling **UniFastMod2**. Then we continue the reduction steps on level 2, 3, $\cdots$, i, $\cdots$, s. On level i, the given array is viewed as an array with *size* / $\prod_{j=1}^{i}(degree(f,\ j)\ +\ 1)$ slots. Each slot has size $\prod_{j=1}^{i}(degree(f,\ j)\ +\ 1)$. We iteratively conduct the reduction steps from level 1 to level s by calling function **RS**. In this way, we compute a *normal form* in a BULL traversal.

Algorithm 18 Fast Univariate Division 2

```
UniFastMod2 ($f,\ TS,\ s$)

1  $n \longleftarrow \deg f$
2  $m \longleftarrow \deg T_s$
3  if $n < m$ then
4    $q \longleftarrow 0$
5    $r \longleftarrow f$
6  else
7    $q \longleftarrow \mathrm{rev}_n(f)\, T_s^{-1} \mod x^{n-m+1}$
8    $q \longleftarrow \mathrm{rev}_{n-m}(q)$
9    RC2($q,\ 0,\ n-m,\ TS,\ s-1$)
10   $w = T_s\, q$
11   RC2($w,\ 0,\ n-m,\ TS,\ s-1$)
12   $r \longleftarrow f - w$
13 return $r$
```

Algorithm 19 Iterative Reduction 2

```
RC2 ($f,\ start,\ end,\ TS,\ s$)

1 for $i$ from start to end do
2   $coef(f,i)$=NormalForm2($coef(f,i), TS, s$)
```

9.3.3 Parallelizing normal form

Both approaches based on either depth-first or bottom-up level-by-level tree traversal are nice applications to parallelize. In our setting, we suppose input polynomial are dense, thus the workload of each coefficient reduction is close. We describe our parallelization strategies as following.

Parallelism in Depth-first Method

Algorithm 20 Parallel Normal Form

NormalForm_Para (f, TS, s)

```
1 if ($s == 0$) return
2 $d = deg(f)$
3 for $i$ from 0 to d do
4    Task=NormalForm_Para($coef(f,i)$,$TS$,$s$-1)
5    CreateThread (Task)
6 DumpThreadPool()
7 $f$ =UniFastMod($f$, $TS$, $s-1$)
```

In the depth-first method we cursively create a thread for each coefficient reduction which we called a "**task**". All threads will live in a **thread_pool**. When the **thread_pool** is full. We will force all threads to finish up before inserting a new one. To force all threads to finish, we use the function **DumpThreadPool**. Function **NormalForm_Para** in above pseudo-code is the parallelized version of the depth-first multivariate reduction.

Algorithm 21 Creating Tasks

CreateThread (**Task**)

```
1 Creat a thread for Task in thread_pool
2 if thread_pool is full.
3    DumpThreadPool(thread_pool)
```

Algorithm 22 Dump Thread-pool

DumpThreadPool(**thread_pool**)

```
1 Force all threads in thread_pool to finish.
```

In the BULL traversal, we have two slightly different sub-methods. One is that at each level we create C threads to handle all reductions on this level in parallel, where C is a constant. Then, we wait them to finish and destroy these C threads before go to next level. Therefore, the total number of threads being created is parametrized by the number of variables of the input. This sub-method is presented by function **NormalForm2_Para_1** in above pseudo-code. In the other sub-method, we will create a fixed number of threads and put them into sleep at the beginning. Then we start the BULL traversal. When there is a reduction on demanding, we will push it onto a task queue and send a signal to wake up some thread. The waken thread will go to fetch a task from the task queue and handle it immediately. If there are multiple tasks have been pushed on the task queue, multiple threads will be waken up and run in parallel. After finishing a task, the thread will go back to sleep or continue to handle another task. This sub-method is presented by function **NormalForm2_Para_2**.

Algorithm 23 Parallelism in Bottom-up Level-by-level Method

```
NormalForm2_Para_1 ($f$, $TS$, $s$)

1  if ($s == 0$) return $f$
2  $size = \prod_{j=1}^{s} (degree(f, j) + 1)$
3  $i = 2$
4  while ($i \leq s$) do
5    $ss = size / \prod_{j=1}^{i} (degree(f, j) + 1)$
     // suppose NoOfCPU divides ss.
6    $q = ss$ / NoOfCPU
7    for $j$ from 0 to NoOfCPU-1 repeat
8      Task = RS ($f$, $jq$, $(j+1)q$, $TS$, $i$)
9      CreateThread ( Task )
10   $i = i + 1$
11   DumpThreadPool()
```

The first sub-method is very easy to implement. But the overhead of creating and destroying many threads maybe burdensome in large input cases. The second sub-method takes a little more coding effort for tasks management and threads synchronization. But it is advantageous by avoiding the potential overhead happened in the first sub-method.

We used `pthread` library to implement the parallelization. We tested the performance on a AMD 4 processor machine. We observed a factor of 3.5 speed-up

Algorithm 24 Parallelism in Bottom-up Level-by-level Method Variant.

```
NormalForm2_Para_2 ($f$, $TS$, $s$)

1  Create $C$ threads and put them into sleep.
2  if ($s == 0$) return $f$
3  $size = \prod_{j=1}^{s} (degree(f, j) + 1)$
4  $i = 2$
5  while ($i \le s$) do
6    $ss = size / \prod_{j=1}^{i} (degree(f, j) + 1)$
     // suppose NoOfCPU divides ss.
7    $q = ss$ / NoOfCPU
8    for $j$ from 0 to NoOfCPU-1 repeat
9      Task = RS ($f$, $jq$, $(j+1)q$, $TS$, $i$)
10     Wake up a thread to handle Task.
11   $i = i + 1$
12 Finish and terminate all threads.
```

when the input size is sufficiently large. The experimentation results are reported in Section 9.4.

9.4 Benchmarks

In Section 9.3, several parallelization strategies have been described. We provide benchmark results for these methods. The tested operation is modular multiplication itself and the tested strategies are summarized in below list.

0	Sequential algorithm.
1	Depth-first traversal with a thread pool.
2	BULL traversal with a thread pool.
3	BULL traversal with sleep/wake-up threads.

Table 9.1: List of parallel strategies.

We conducted our benchmark on a AMD Opteron 850 4-Processor machine with CPU MHZ 2391.537 and cache size 1024 KB for each processor. The input dense polynomials are randomly generated. The benchmark data can well reflect the performance in real world computation.

We benchmarked 2, 3 and 4 variable cases. We observe a factor of $2 \sim 3$ speed-up in those examples. Here, we only report the data we collected from the 4 variable example. In this example, we fixed the partial degrees in x_3 and x_4 at 4, i.e. the number of processors. Then by increasing partial degrees in x_1 and x_2, we obtain a timing surface for each methods listed in above table. Namely, Figure 9.1 is the benchmark between the sequential method and the Depth-first traversal parallelization method with a global thread pool. Figure 9.2 is the benchmark between sequential method and the BULL traversal parallelization method with a global thread pool. Figure 9.3 is the benchmark between sequential method and the BULL traversal parallelization method with threads sleep/wake-up strategy. And Table 9.4, 9.4, 9.4 and 9.4 are the selected data point from Figure 9.1, 9.2, 9.3 and 9.4 respectively.

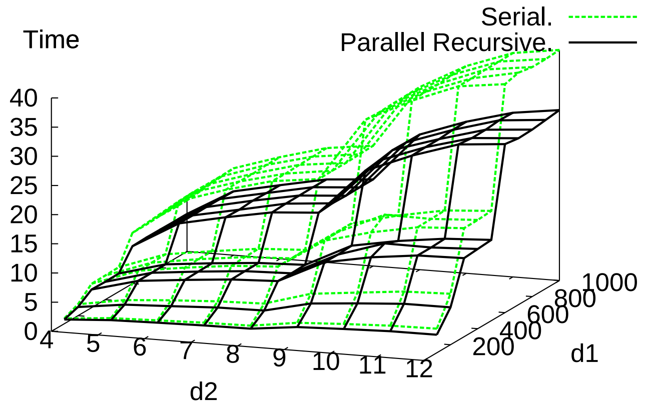


Figure 9.1: Method 0 vs. method 1

d_4	d_3	d_2	d_1	method 0 (sec)	method 1 (sec)
4	4	4	100	0.926028	0.736449
4	4	6	300	8.104279	6.015184
4	4	8	500	9.642438	7.084307
4	4	10	800	35.232581	25.746897
4	4	12	1000	39.521405	29.216119

Table 9.2: Selected data points from Figure 9.1

According to the benchmark result, the depth-first method does not improve the performance by big factors w.r.t to the number of processors. The main reason is that when the coarser grain parallelization is well balanced and processors have been well

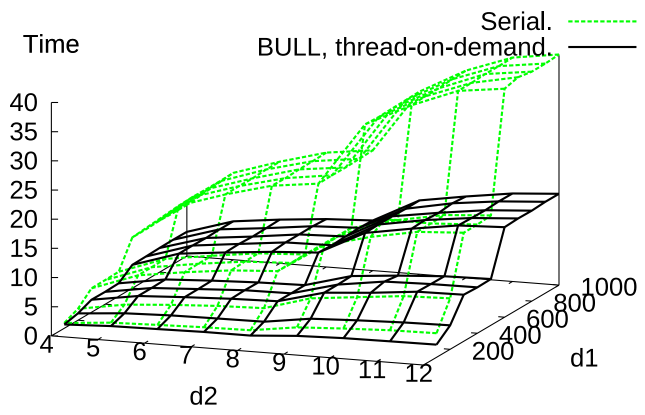


Figure 9.2: Method 0 vs. method 2

d_4	d_3	d_2	d_1	method 0 (sec)	method 2 (sec)
4	4	4	100	0.926028	0.659218
4	4	6	300	8.104279	3.844373
4	4	8	500	9.642438	4.391355
4	4	10	800	35.232581	13.915399
4	4	12	1000	39.521405	15.650396

Table 9.3: Selected data points from Figure 9.2

utilized, it's insensible to keep generating finer grain sub-threads recursively for the sub-tasks, especially when the sub-tasks are small in terms of workload. On the other hand, the bottom-up level-by-level approach has a factor of $2 \sim 3$ speed up based on the input size accordingly. The examples with larger degrees have better speed-up than the smaller ones. The main reason for this is that the overhead generated by threads and tasks management is still not negligible for smaller input.

For the comparison between methods 2 and 3, we observe that method 2 outperforms method 3 for smaller input. The main reason is that in methods 3 the overhead of managing task queues and synchronizing signals is more expensive than the one in method 2. When the input is small, the overhead has bigger impact on the overall computational time. Whereas, method 3 will only generate fixed number of threads. Thus, the scheduling becomes much simpler. The overhead of creating /destroying threads in the middle steps has been avoided as well. Thus, for larger input method 3 outperforms method 2 according to our results, though the gap is not big.

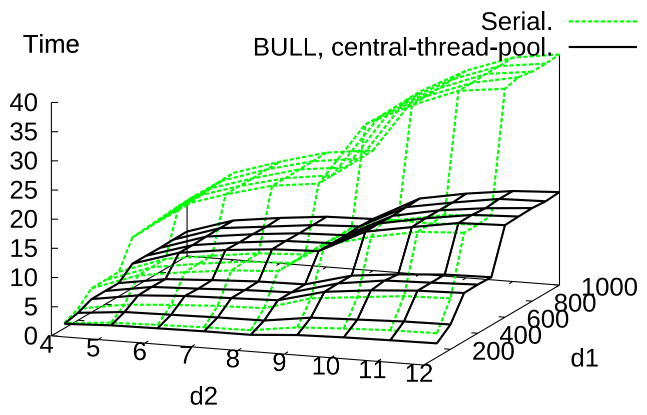


Figure 9.3: Method 0 vs. method 3

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
4	4	4	100	0.926028	0.778774
4	4	6	300	8.104279	4.031646
4	4	8	500	9.642438	4.531477
4	4	10	800	35.232581	13.335127
4	4	12	1000	39.521405	14.952662

Table 9.4: Selected data points from Figure 9.3

Figure 9.4 shows an improved version of method 3. The speed-up is yielded by replacing all Fast Fourier Transform by Truncated Fourier Transform (TFT). Although this improvement seems unrelated to parallelism, the better multiple cache behavior deserves to be counted in. Namely, TFT requires less memory to store the intermediate results than FFT. There is a larger chance that these results will be kept in cache and used in later computation steps on the same processor.

Above benchmarks only show a factor of $2 \sim 3$ speed up on a 4 processor machine. This is not a satisfying result with considering that polynomials in our applications are dense ones. Dense polynomial computations usually provide a good opportunity for work-load balance. However, we have identified the major bottle-neck that impedes the perform in our benchmark examples. Recall that in previous benchmarks we set the partial degrees of x_4 and x_3 as a constant number 4. This leads a situation that in some of the sub-algorithms such as *Coefficient Reduction*, there is no enough work-load to be scheduled evenly to all 4 processors by our current scheduling method.

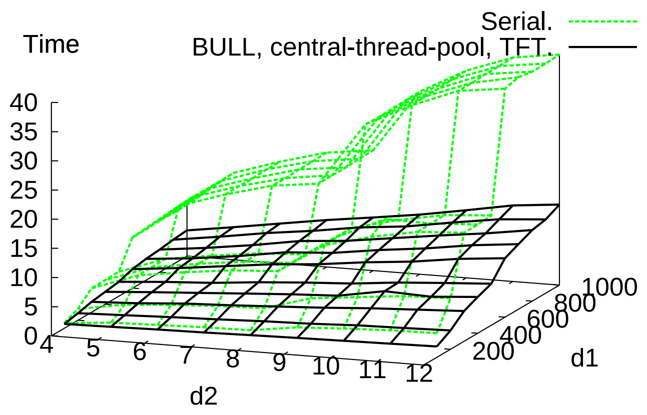


Figure 9.4: Method 0 vs. method 3 with TFT implementation.

d_4	d_3	d_2	d_1	method 0 (sec)	TFT (sec)
4	4	4	100	0.926028	0.755583
4	4	6	300	8.104279	2.732532
4	4	8	500	9.642438	4.831472
4	4	10	800	35.232581	10.011660
4	4	12	1000	39.521405	13.816763

Table 9.5: Selected data points from Figure 9.4

Therefore, we increase the degrees of x_3 and x_4 to be 8. Then, we observe a factor $3.2 \sim 3.3$ speed-up between method 1 and method 3. In Table 9.4 we list a few timing points from the new benchmark result.

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
8	8	8	100	13.770629	4.321261
8	8	8	300	96.117776	18.458235
8	8	8	1000	132.304345	39.757645
8	8	8	1600	277.367573	82.651414

Table 9.6: Larger benchmark 1.

When we increase the partial degrees of x_3 and x_4 to be $16, 24, 32, \cdots$. we observe a factor of $3.4 \sim 3.6$ speed-up between method 1 and method 3 (see Table 9.4).

To summarize, for the larger examples, especially when we increase the partial

d_4	d_3	d_2	d_1	method 0 (sec)	method 3 (sec)
16	16	16	16	15.303748	4.567856
16	16	24	24	56.612566	16.479111
16	16	32	32	63.762428	18.359758
16	16	40	40	236.199680	67.175220
16	16	48	48	252.753472	71.237213
16	16	56	56	265.966837	74.979127

Table 9.7: Larger benchmark 2.

degrees of x_3 and x_4 in 4-variable case, the performance is reasonably better. By profiling information, we know the top level division in BULL method is often a dominant factor. Thus, increasing the degrees of top level variables to some extend with respect to the number of processors allows a more balanced work-load assignment thus a better performance. Although, our experiments are conducted on a 4 processor machine. We believes that our approach will scale on larger parallel SMP system. Actually, the number of threads in application has been parametrized such that it can be easily adjusted according to the number of processors or other cut-offs.

9.5 Summary

In conclusion, we studied multithreaded versions of multivariate polynomial arithmetic modulo a triangular set. In this report, we focused on the *normal form* operation. We obtain parallelism from two procedures: a multidimensional FFT algorithm and our *normal form* algorithm. Due to the intrinsic data-dependency inside these operations, we observe a factor of 2$\sim$3 speed up on a 4 processor machine. One major issue remains: detecting cut-offs between the different possible strategies. This is a highly complicated task. A cut-off in our application is parametrized by the type of architectures, the number of processors, the number of variables of the input, and the shape of the given triangular set, etc.

Chapter 10

Conclusion

This thesis has been devoted to the design and implementation of polynomial system solvers based on symbolic computation. Driven by this motivations, we have developed new algorithms and implementations to support the technique of triangular decompositions for polynomial solving.

As reported in Chapters 3, 4 and 5, we have investigated and demonstrated that with suitable implementation techniques, FFT-based asymptotically fast polynomial arithmetic in practice can outperform the corresponding classical algorithms in a significant manner. By integrating our C-level implementation of fast polynomial arithmetic into AXIOM, the AXIOM higher level existing related libraries has been sped up in large scale. By using the same implementation technique, we have demonstrated in Chapter 8 that MAPLE higher level libraries such as *RegularChains* have also been dramatically improved in terms of performance. We have reported in Chapters 6, 7 and 8 our new asymptotically fast algorithms, i.e. *fast integer reduction trick*, *modular multiplication*, *regular GCD*, *bivariate solver*, *two-equation solver* and *regularity test*. In Chapter 9, we have investigated the potential parallelism inside fast algorithms modulo regular chains. All our reported new implementations and algorithms from this thesis have been finalized as a commercial software library *Modpn* (see Section 8.2)

In this research, we have focused on algorithms modulo regular chains in dimension-zero. Higher dimensional asymptotically fast triangular decompositions algorithms can be developed and implemented based on these results. Therefore, we expect that the generic triangular decompositions based polynomial solvers can yield high-performance.

Bibliography

[1] ALDOR: a computer algebra system. `http://www.aldor.org`.

[2] AXIOM: a general-purpose commercial computer algebra system. `http://page.axiom-developer.org`.

[3] GCL: GNU Common Lisp. `http://www.gnu.org/software/gcl`.

[4] GMP: GNU Multiple Precision Arithmetic library. `http://swox.com/gmp/`.

[5] MAGMA: the computational algebra system for algebra, number theory and geometry. `http://magma.maths.usyd.edu.au/magma/`.

[6] NTL: the Number Theory Library. `http://www.shoup.net/ntl`.

[7] Linbox: exact computational linear algebra. `http://www.linalg.org/`.

[8] M. Atiyah and L. G. Macdonald. *Introduction to Commutative Algebra*. Addison-Wesley, 1969.

[9] P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *J. Symb. Comp.*, 28(1-2):105–124, 1999.

[10] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[11] E. Becker, T. Mora, M. G. Marinari, and C. Traverso. The shape of the shape lemma. In *Proc. of the international symposium on Symbolic and algebraic computation*, pages 129–133, New York, NY, USA, 1994. ACM Press.

[12] D. Bini. Relations between exact and approximate bilinear algorithms. Applications. *Calcolo*, 17(1):87–97, 1980.

[13] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Inf. Proc. Lett.*, 8(5):234–235, 1979.

[14] D. Bini, G. Lotti, and F. Romani. Approximate solutions for the bilinear form computational problem. *SIAM J. Comput.*, 9(4):692–697, 1980.

[15] R. D. Blumofe, M. Frigo, C. F. Joerg, and C. E. Leiserson. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. SPAA'96*, pages 297–308. ACM Press, 1996.

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

[17] A. Bostan and É. Schost. On the complexities of multipoint evaluation and interpolation. *Theor. Comput. Sci.*, 329:223–235, 2004.

[18] R. Brent. Algorithms for matrix multiplication. Master's thesis, Stanford University, 1970. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/.

[19] R. Brent, F. Gustavison, and D. Yun. Fast solution of Toeplitz systems of equations and computations of Padé approximants. *Journal of Algorithms*, 1:259–295, 1980.

[20] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, 1965.

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

[22] C. Chen, F. Lemaire, O. Golubitsky, M. Moreno Maza, and W. Pan. *Comprehensive Triangular Decomposition*, volume 4770 of *Lecture Notes in Computer Science*, pages 73–101. Springer Verlag, 2007.

[23] G. Collins. The calculation of multivariate polynomial resultants. *Journal of the ACM*, 18(4):515–532, 1971.

[24] S. Cook. *On the minimum computation time of functions.* PhD thesis, Harvard University, 1966.

[25] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier +series. *Math. Comp.*, 19:297–301, 1965.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2002.

[27] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.

[28] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *Proc. of* Transgressive Computing 2006, Granada, Spain, 2006.

[29] X. Dahan and É. Schost. Sharp estimates for triangular sets. In *ISSAC 04*, pages 103–110. ACM, 2004.

[30] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL 85 Vol. 2*, volume 204 of *Lect. Notes in Comp. Sci.*, pages 289–290. Springer-Verlag, 1985.

[31] L. Ducos. *Effectivité en théorie de Galois. Sous-résultants.* PhD thesis, Université de Poitiers, 1997.

[32] L. Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145:149–163, 2000.

[33] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. *ISSAC 02*, pages 63–74, 2002.

[34] D. Duval. *Questions Relatives au Calcul Formel avec des Nombres Algébriques.* Université de Grenoble, 1987. Thèse d'État.

[35] M. El Kahoui. An elementary approach to subresultants theory. *J. Symb. Comp.*, 35:281–292, 2003.

[36] I. Z. Emiris and V. Y. Pan. Fast Fourier transform and its applications. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computations.* CRC Press Inc, 1999.

[37] T. Färnqvist. Number theory meets cache locality: efficient implementation of a small prime FFT for the GNU Multiple Precision arithmetic library. Master's thesis, Stockholms Universitet, 2005.

[38] R. J. Fateman. Vector-based polynomial recursive representation arithmetic. `http://www.norvig.com/ltd/test/poly.dylan`, 1999.

[39] J.-C. Faugère. *Résolution des systèmes d'équations algébriques.* PhD thesis, Université Paris 6, 1994.

[40] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases. *J. Pure and Appl. Algebra*, 139(1-3):61–88, 1999.

[41] 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.

[42] M. Frigo and S. G. Johnson. Fftw. http://www.fftw.org/.

[43] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[44] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 2003.

[45] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992.

[46] T. Gómez Díaz. *Quelques applications de l'évaluation dynamique.* PhD thesis, Université de Limoges, 1994.

[47] L. González Vega, H. Lombardi, T. Recio, and M. Roy. Spécialisation de la suite de sturm et sous-résultants. *Informatique Théorique et Applications*, 24(6):561–588, 1990.

[48] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook.* Springer, 2003.

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

[50] M. v. Hoeij and M. Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In T. Mora, editor, *Proc. ISSAC 2002*, pages 109–116. ACM Press, July 2002.

[51] J. Hoeven. Truncated Fourier transform. In *Proc. ISSAC'04*. ACM Press, 2004.

[52] R. D. Jenks and R. S. Sutor. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992. AXIOM is a trade mark of NAG Ltd, Oxford UK.

[53] J. R. Johnson, W. Krandick, K. Lynch, K. G. Richardson, and A. D. Ruslanov. High-performance implementations of the descartes method. In *ISSAC'06*, pages 154–161. ACM, 2006.

[54] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical taylor shift by 1. In *ISSAC'05*, pages 200–207. ACM, 2005.

[55] M. Kalkbrener. *Three contributions to elimination theory.* PhD thesis, Johannes Kepler University, Linz, 1991.

[56] M. Kalkbrener. A generalized euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143–167, 1993.

[57] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, (7):595–596, 1963.

[58] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1999.

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

[60] L. Langemyr. Algorithms for a multiple algebraic extension. In *Effective methods in algebraic geometry (Castiglioncello, 1990)*, volume 94 of *Progr. Math.*, pages 235–248. Birkhäuser Boston, 1991.

[61] D. Lazard. A new method for solving algebraic systems of positive dimension. *Discr. App. Math*, 33:147–160, 1991.

[62] D. Lazard. Solving zero-dimensional algebraic systems. *J. Symb. Comp.*, 15:117–132, 1992.

[63] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In I. S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[64] X. Li. *Efficient Management of Symbolic Computations with Polynomials.* 2005. University of Western Ontario.

[65] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In *ICMS'06*, pages 12–23. Springer, 2006.

[66] X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. In *Proc. PASCO'07*, pages 53–59, New York, NY, USA, 2006. ACM Press.

[67] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. Submitted to ISSAC'09, 2009.

[68] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The `modpn` library: Bringing fast polynomial arithmetic into MAPLE. In *MICA'08*, 2008.

[69] 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.

[70] X. Li, M. Moreno Maza, and É. Schost. On the virtues of generic programming for symbolic computation. In *ICCS'07*, volume 4488 of *Lecture Notes in Computer Science*, pages 251–258. Springer, 2007.

[71] M. van Hoeij and M. Monagan. A modular GCD algorithm over number fields presented with multiple extensions. In *ISSAC'02*, pages 109–116. ACM, 2002.

[72] B. Mishra. *Algorithmic Algebra*. Springer-Verlag, New York, 1993.

[73] R. T. Moenck. Practical fast polynomial multiplication. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148, New York, NY, USA, 1976. ACM Press.

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

[75] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. Presented at the MEGA-2000 Conference, Bath, England.

[76] M. Moreno Maza and R. Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*, pages 365–382. Springer, 1995.

[77] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Proc. PLDI'07*, 2007.

[78] V. Y. Pan. Simple multivariate polynomial multiplication. *J. Symb. Comp.*, 18(3):183–186, 1994.

[79] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gavcić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc' IEEE*, 93(2):232–275, 2005.

[80] R. Rasheed. Modular methods for solving polynomial systems. Master's thesis, 2007. University of Western Ontario.

[81] J. F. Ritt. *Differential Equations from an Algebraic Standpoint*, volume 14. American Mathematical Society, New York, 1932.

[82] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977.

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

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

[85] É. Schost. Multivariate power series multiplication. In *ISSAC'05*, pages 293–300. ACM, 2005.

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

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

[88] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik.*, 13:354–356, 1969.

[89] S. Varette and J.-L. Roch. Probabilistic certification of divide & conquer algorithms on global computing platforms. application to fault-tolerant. In *Proc. PASCO'07*. ACM Press, 2007.

[90] S. M. Watt. The $A^{\#}$ programming language and its compiler. Technical report, IBM Research, 1993.

[91] W. T. Wu. A zero structure theorem for polynomial equations solving. *MM Research Preprints*, 1:2–12, 1987.

[92] L. Yang and J. Zhang. Searching dependency between algebraic equations: an algorithm applied to automated reasoning. Technical Report IC/89/263, International Atomic Energy Agency, Miramare, Trieste, Italy, 1991.

[93] C. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1993.

Curriculum Vitae

Name:	**Xin Li**
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada Ph.D. Computer Algebra, Apr. 2009 (expected date)
	The University of Western Ontario London, Ontario, Canada M.Sc. Computer Science, Sept. 2005
	Beijing Information Science & Technology University Beijing, China B.E. in Automation, Sept. 1997
Selected Honors and Awards:	Best Novel Use of Mathematics in Technology Transfer 2009 from MITACS (See Refereed Software).
	NSERC PGS Scholarship, 2007 - 2009
	OGSST Scholarship, 2006
Working Experience:	IBM Canada Ltd. Compiler backend developer associated student. Apr.2008 - Apr.2009

Refereed Software:

X. Li, M. Moreno Maza, *The* `Modpn` *library and its Maple wrapper package* `FastArithmeticTools` *have been integrated in the* MAPLE `RegularChains` *library and will be distributed with Maple version 13, 2009.*

This software library has won the national level award *Best Novel Use of Mathematics in Technology Transfer 2009* from MITACS (Mathematics of Information Technology and Complex Systems).

Refereed Papers:	X. Li, M. Moreno Maza and W. Pan *Computations modulo Regular Chains.* Accepted by ISSAC' 2009, Korea Institute for Advanced Study.
	X. Li, M. Moreno Maza, R. Rasheed and É Schost, *The Modpn library: Bringing Fast Polynomial Arithmetic into Maple (extended version).* Submitted to the Journal of Symbolic Computation. 2008.
	X. Li, M. Moreno Maza and É Schost, *Fast Arithmetic for Triangular Sets: from Theory to Practice (extended version).* Journal of Symbolic Computation (to appear). 2009.
	X. Li, M. Moreno Maza, R. Rasheed and É Schost, *High-Performance Symbolic Computation in a Hybrid Compiled-Interprered Programming Environment.* In proc. of CASA'2008, Perugia, Italy, IEEE Press.
	X. Li , M. Moreno Maza, R. Rasheed and É Schost, *The Modpn library: Bringing Fast Polynomial Arithmetic into Maple.* In proc. of MICA'2008, Stonehaven Bay, Trinidad and Tobago.
	X. Li and M. Moreno Maza, *Multithreaded Parallel Implementation of Arithmetic Operations Modulo a Triangular Set.* In proc. of PASCO' 2007, UWO, Canada, ACM Press.
	X. Li, M. Moreno Maza and É Schost, *Fast Arithmetic for Triangular Sets: From Theory to Practice.* In proc. of ISSAC' 2007, Waterloo, Canada, ACM Press.
	X. Li, M. Moreno Maza and É Schost, *On the Virtues of Generic Programming for Symbolic Computation.* In proc. of CASA' 2007, Beijing, China, ACM Press.
	X. Li and M. Moreno Maza, *Efficient Implementation of Polynomial Arithmetic in a Multiple-level Programing Environment.* In proc. of ICMS' 2006, Spain, ACM Press.
	A. Filatei, X. Li, M. Moreno Maza and É Schost, *Implementation Techniques for Fast Polynomial Arithmetic in a High-level Programming Environment.* In proc. of ISSAC'2006, Italy, ACM Press.