FAST ALGORITHMS, MODULAR METHODS, PARALLEL APPROACHES
AND SOFTWARE ENGINEERING FOR SOLVING POLYNOMIAL SYSTEMS
SYMBOLICALLY

(Spine title: Contributions to Polynomial System Solvers)

(Thesis format: Monograph)

by

Yuzhen <u>Xie</u>

Graduate Program
in
Computer Science

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

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario, Canada
September 4, 2007

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Joint-Supervisor:

_____
Dr. Marc Moreno Maza

Joint-Supervisor:

_____
Dr. Stephen M. Watt

Examination committee:

_____
Dr. Rob Corless

_____
Dr. Erich Kaltofen

_____
Dr. Hanan Lutfiyya

_____
Dr. Sheng Yu

The thesis by

Yuzhen <u>Xie</u>

entitled:

**Fast Algorithms, Modular Methods, Parallel Approaches and Software
Engineering for Solving Polynomial Systems Symbolically**

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

Date _____   _____
                                  Chair of the Thesis Examination Board

Abstract

Symbolic methods are powerful tools in scientific computing. The implementation of symbolic solvers is, however, a highly difficult task due to the extremely high time and space complexity of the problem. In this thesis, we study and apply fast algorithms, modular methods, parallel approaches and software engineering techniques to improve the efficiency of symbolic solvers for computing triangular decomposition, one of the most promising methods for solving non-linear systems of equations symbolically.

We first adapt nearly optimal algorithms for polynomial arithmetic over fields to direct products of fields for polynomial multiplication, inversion and GCD computations. Then, by introducing the notion of equiprojectable decomposition, a sharp modular method for triangular decompositions based on Hensel lifting techniques is obtained. Its implementation also brings to the MAPLE computer algebra system a unique capacity for automatic case discussion and recombination.

A high-level categorical parallel framework is developed, written in the AL-DOR language, to support high-performance computer algebra on symmetric multi-processors and multicore processors. A component-level parallelization of triangular decompositions by the Triade algorithm is realized using this framework. Parallelism is created by applying modular methods, and task scheduling is guided by the geometric information discovered during the solving process.

By reviewing the `RegularChains` library in MAPLE, the challenges for the conception and implementation of triangular decompositions are analyzed. The software engineering techniques for developing a solver in three computer algebra systems targeting different communities of users are compared. We also prove and add two methods for efficiently computing irredundant triangular decompositions and for verifying symbolic solvers.

Our experimentation shows that the software developed, based on our approaches, helps solving application problems that are out of the scope of other comparable solvers. We believe that the algorithms and methods and the framework and our implementation techniques could benefit other areas of scientific computing.

Keywords: polynomial system solving, non-linear equations, triangular decomposition, equiprojectable decomposition, fast algorithm, modular method, multi-processed parallelism, component-level parallelization, categorical parallel framework, irredundant triangular decomposition, verification of solvers

Acknowledgments

While my name is the only one that appears on the author list of this thesis, there are several other people deserving recognition. My supervisors, Dr. Marc Moreno Maza and Dr. Stephen M. Watt, provided excellent research environment and support through my entire PhD study. I wish to extend my appreciation and gratitude to Dr. Marc Moreno Maza for introducing me to these interesting and challenging projects. I feel lucky to participate to these scientific adventures. I feel also honored to collaborate with my co-authors: Changbo Chen, Dr. Xavier Dahan, Dr. Franccois Lemaire, Wei Pan, Dr. Éric Schost, Dr. Ben Stephenson and Dr. Wenyuan Wu.

Sincere thanks and appreciation are extended to all the Professors and Staff members from ORCCA lab and the Computer Science Department for their invaluable teaching and assistance. I also wish to thank fellow graduates and friends for their help and wonderful friendship.

My sincere appreciation also goes to my family for their love and support throughout my life and especially the past few years.

Thank God for everything!

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1   Briefing of Polynomial System Solving

Solving systems of equations is a fundamental problem in mathematics, and is needed clearly for numerous applications in the sciences. Theoretical results and algorithms for this purpose have been accumulating since the ancient times. However, the space and time complexity of these algorithms, such as the exponential time algorithm for factoring univariate polynomials by Kronecker [83, 82], have limited their use until recent years. The development of computer systems has permitted the implementation of these algorithms. It has also stimulated the discovery of faster algorithms for solving systems of equations, such as the work of Berlekamp [14, 15], Zassenhaus [152], Lenstra et al. [93] and Kaltofen et al. [79] for factoring univariate polynomials, leading to polynomial time algorithms.

Algorithmic solutions for solving systems of equations can be classified into three categories: numeric, symbolic and hybrid numeric-symbolic. The decision about which technique should be used is based on the characteristics of the system of equations being solved. For instance, the decision depends on whether the coefficients are known exactly or are approximations obtained from experimental measurements. The choice also depends on the expected answers, which could be a complete description of all of the solutions, only the real solutions, or just one sample solution among many.

Symbolic solvers are powerful tools in scientific computing: they are well suited for problems where the desired output must be exact. They have been applied successfully in areas like digital signal processing, robotics, theoretical physics, cryptology, dynamical systems, with many important outcomes. An overview of these applications is presented in [67].

The implementation of symbolic solvers is, however, a highly difficult task. First, they implement sophisticated algorithms, which are generally at the level of on-going research. Moreover, in most computer algebra systems, the `solve` command involves nearly the entire set of libraries in the system, challenging the most advanced operations on matrices, polynomials, algebraic and modular numbers, etc.

Secondly, algorithms for solving systems of polynomial equations are, by nature, of exponential-time complexity. Consequently, symbolic solvers are extremely time-consuming when applied to large problems. Worse yet, intermediate expressions can grow to enormous size, which may halt the computation, even if the result is of moderate size. As such, the implementation of symbolic solvers requires techniques that go far beyond the manipulation of algebraic or differential equations including efficient memory management, data compression, and parallel and distributed computing, among others.

Last, but not least, the precise output specifications of a symbolic solver can be quite involved. Indeed, given an input polynomial system F, defining what a symbolic solver should return implies describing what the geometry of the solution set $V(F)$ of F can be. For an arbitrary F, the set $V(F)$ may consist of components of different natures and sizes: points, lines, curves, surfaces, etc. This leads to the great challenge of validating symbolic solvers.

Symbolic methods for systems of linear equations and systems of univariate or bivariate equations have received the attention of many researchers. They have obtained nearly optimal algorithms (see chapters 12, 14 and 15 in [57]) and highly efficient implementations [124, 114, 46]. The main reasons for this success are modular methods [61], the use of fast algorithms for polynomial arithmetic [57] and highly optimized code that makes efficient use of computer processor and memory resources [123, 73, 95].

Symbolic methods for non-linear systems of equations have also been investigated by many researchers who have proposed several approaches: Gröbner bases [1, 13, 26, 35], primitive element representations [63, 64, 88], and triangular decompositions [76, 87, 108, 138, 146]. Consider the following polynomial system F over the field $\mathbb{Q}$ of rational numbers:

$$\begin{cases} -z^5 - z^4 - 2z^3 - z^2 + 3z + y + x + 3 = 0 \\ -2z^5 - 4z^3 - z^2 + 6z + y^2 + 2y = 0 \\ z^3 + yz^2 - z - y = 0 \\ z^6 + z^4 - 4z^2 + 2 = 0. \end{cases}$$

3

It has lexicographical $(x > y > z)$ Gröbner basis G given by:

$$
\left\{
\begin{aligned}
&x^3 + 2x^2 - 2x + z^2 - 5 \\
-&3x^3 - 4x^2 + yx + zx + 8x + zy + y + z + 11 \\
-&x^3 + zx^2 - x^2 + 3x + y - 2z + 3 \\
&3x^3 + 4x^2 - 8x + y^2 - 11 \\
&x^3 + yx^2 + x^2 - 3x - 3y - 3 \\
&x^4 - 5x^2 + 6 \ ,
\end{aligned}
\right.
$$

which is a system of polynomials generating the same ideal as F and with algorithmic properties. For instance, given a polynomial equation $p(x, y, z) = 0$, one can decide from G whether this equation is a consequence of those of F, or not. (This is the ideal membership problem.)

A primitive element representation provides a parametric representation of the zero set of F:

$$
\left\{
\begin{aligned}
x &= -t^2 - 1 \\
y &= -t \\
z &= t
\end{aligned}
\right.
\quad \text{where} \quad t^4 + 2t^2 - 2 = 0 \ \text{ or,}
$$

$$
\left\{
\begin{aligned}
x &= t \\
y &= -t - 1 \\
z &= -1
\end{aligned}
\right.
\quad \text{where} \quad t^2 - 2 = 0 \ \text{ or,}
$$

$$
\left\{
\begin{aligned}
x &= t - 2 \\
y &= -t + 1 \\
z &= 1
\end{aligned}
\right.
\quad \text{where} \quad t^2 - 4t + 2 = 0.
$$

The triangular decomposition with the variable order of $(x > y > z)$ provides another representation of the zero set of F (non-parametric):

$$
\left\{
\begin{aligned}
z^4 + 2z^2 - 2 &= 0 \\
y + z &= 0 \\
x + z^2 + 1 &= 0
\end{aligned}
\right.
\quad \text{and} \quad
\left\{
\begin{aligned}
z^2 - 1 &= 0 \\
y^2 + 2y - 1 &= 0 \\
x + y + 1 &= 0.
\end{aligned}
\right.
$$

Primitive element representation and triangular decompositions are good at separating the different components of the set of solutions, and deciding if a component has real solutions (since these symbolic methods give all of the complex solutions). In addition, triangular decompositions are commonly used in differential algebra [20, 71]. As a matter of fact, triangular decomposition is a step toward obtaining irreducible components.

Moreover, sharp estimates [42] are known for the size of the triangular decomposi-

tion of a polynomial system. In addition, multivariate polynomial arithmetic during a triangular decomposition can be reduced to univariate operations that can be performed using asymptotically fast algorithms and efficient implementation techniques [54, 96, 97, 98, 99].

After an informal introduction to polynomial system solving and triangular decompositions, Chapter 2 recalls the fundamental notions used in these areas, namely Gröbner bases, algebraic varieties, triangular sets and regular chains. Chapter 2 outlines also an algorithm called Triade [108], for computing *TRIA*ngular *DE*compositions. The Triade algorithm has several important features for the topics discussed in this thesis. In particular, it manages the solving process as a tree of tasks, providing an initial framework toward a parallel algorithm. It also generates the (intermediate or output) components in decreasing order of dimension. The intermediate objects, regular chains and hypersurfaces, are structured and possess rich properties. Therefore, one can exploit geometrical information during the solving process, which provides good control over the intermediate computations. This mechanism allows the Triade algorithm to detect and cut redundant computing branches at an early stage, which is important because removing superfluous components is a major issue with all decomposition methods [86, 88, 126, 146]. In addition, the Triade algorithm has been implemented in the ALDOR language [3], in the computer algebra systems AXIOM [72] and MAPLE [105] as the `RegularChains` library [90]. This provides us with effective tools for conducting experiments.

1.2 Contributions of this Thesis

This thesis focuses on the challenges associated with developing efficient symbolic solvers. We address our research endeavor on four important, strongly-related techniques: *fast algorithms*, *modular methods*, *parallel approaches* and *software engineering* for solving polynomial systems symbolically by way of triangular decompositions.

Fast algorithms. The standard approach for computing with an algebraic number is through the data of its irreducible minimal polynomial over some base field k. However, in typical tasks such as polynomial system solving, which involve many algebraic numbers of high degree, following this approach will require using probably costly factorization algorithms. Jean Della Dora, Claire Dicrescenzo and Dominique Duval introduced "Dynamic Evaluation" (also termed "D5 Principle") [43] techniques as a means to compute with algebraic numbers while avoiding factorization. Roughly

speaking, this approach leads one to compute over *direct products of field extensions of k*, instead of only field extensions.

Many algorithms for solving polynomial systems symbolically need to perform standard operations, such as GCD computations, over coefficient rings that are direct products of fields rather than fields. In Chapter 3, we show how asymptotically fast algorithms for polynomials over fields can be adapted to this more general context. In particular, we show that they can be adapted to direct products of fields presented by *triangular sets*. In this context, we obtain nearly optimal (i.e. *quasi-linear*) algorithms for polynomial *quasi-inverse* and GCD computations. This joint work with Marc Moreno Maza, Xavier Dahan and Éric Schost is reported in [41].

Modular methods. Modular methods are extremely efficient tools for controlling the size of intermediate expressions, and hence, for reducing the space complexity of many algorithms for symbolic computation. Modular methods also provide opportunities to use fast polynomial arithmetic. This is a well developed approach for systems of linear equations. Standard applications are the resolution of systems over $\mathbb{Q}$ after specialization at a prime, and over the rational function field $k(Y_1, \ldots, Y_m)$ after specialization at a point $(y_1, \ldots, y_m)$. Applying modular methods to systems of non-linear equations remains an active research area.

Modular algorithms for Gröbner bases [5, 113, 134] and primitive element representations [64] have been developed by several researchers. Some software for computing Gröbner bases relies not only on efficient algorithms, but also on sophisticated implementation techniques [53]. However, classical algorithms for Gröbner bases do not make use of geometrical information. Instead they mainly rely on combinatorial arguments, such as Dixon's Lemma [36]. This makes a sharp modular method difficult to design.

Primitive element representation methods make use of geometric information but lack canonicity. (Indeed, a geometrical object may admit infinitely many parameterizations.) In some cases, two different primitive element representations may encode the same solution set and none of the algorithms are guaranteed to detect this situation.

Triangular decompositions do not have these drawbacks. However, when we started our research in this area, none of the many methods for computing triangular decompositions was making use of modular computations, restricting their practical efficiency. Moreover, all implementations of triangular decompositions available

were putting their main emphasis on the algorithms while failing to give efficient implementation techniques the attention they deserve.

In Chapter 4 we introduce the *equiprojectable decomposition* of a zero-dimensional algebraic variety. We show that the equiprojectable decomposition is canonical among all possible triangular decompositions of such variety, and that it has good computational properties. An efficient algorithm called `Split-and-Merge` is designed for computing the equiprojectable decomposition from any triangular decomposition. With height bound estimates and Hensel lifting techniques, this allows us to deduce an efficient probabilistic modular algorithm for solving non-linear systems with a finite number of complex solutions. This joint work with Xavier Dahan, Marc Moreno Maza, Éric Schost and Wenyuan Wu is published in [39].

We created a MAPLE implementation of this modular algorithm on top of the `RegularChains` library. Our implementation was released with MAPLE 11. Tests on benchmark systems from the Symbolic Data Project [128] reveal its strong features compared with two other MAPLE solvers (in version 10): *Triangularize*, from the `RegularChains` library, and *gsolve*, from the *Groebner* package. Our experimentation demonstrates the efficiency of this modular algorithm in reducing the size of the intermediate computations, and hence, its ability to solve difficult problems.

The deployment of these techniques based on the equiprojectable decomposition brings an extra flavor into MAPLE: the `MatrixTools` submodule. This module is designed for linear algebra over non-integral domains, allowing automatic case discussion and recombination.

Parallel approaches. Computer algebra involves complex algorithms, data structures and intensive computations. Parallelism is an important research topic in computer algebra as it was used to achieve efficient executions throughout the 1980s and 1990s. An overview of these developments is presented in the *Computer Algebra Handbook* [67]. The increasing availability of parallel computer architectures, from SMPs to multi-core laptops, has revitalized the need to develop parallel mathematical algorithms and software capable of exploiting these new computing resources. This need is even more dramatic in the case of symbolic computations which offer exciting, but highly complex challenges to computer scientists.

The parallelization of two other algorithms for solving polynomial systems symbolically have already been actively studied. The first one is Buchberger's algorithm for computing Gröbner bases for which parallel implementations are described in [7, 23, 27, 29, 52, 94]. The second one is the Characteristic Set method developed

by Wu [146] which is discussed in [2, 147, 148]. In all these works, the parallelized operation is polynomial reduction (or simplification). More precisely, given two polynomial sets A and B (with some conditions on A and B, depending on the algorithm) the reductions of the elements of A by those of B are executed in parallel.

In Chapter 5, we describe our *component-level* parallelization of triangular decompositions based on the Triade algorithm. Our long term goal is to achieve an efficient multi-level parallelism: coarse grained (component) for tasks computing geometric objects in the solution sets, and medium/fine grained for polynomial arithmetic such as GCD/resultant computation and polynomial reduction within each task.

Algorithms for triangular decompositions tend to split the input system into subsystems, and seem to be natural for component-level parallelization. However, a naive parallelization provides little benefit due to the following facts. Most polynomial systems $F \in \mathbb{Q}[X]$ with finitely many solutions are *equiprojectable*, that is, they can be represented by a single triangular set. This is verified both in theory (the Shape Lemma [12]) and in practice [128]. Therefore, there are very few opportunities for splitting the work with such polynomial systems at a coarse-grained level. Moreover, the tasks that appear when solving these systems are highly irregular in terms of both time and memory needs.

We create parallel opportunities by modular computations. Triangular decomposition methods are much more likely to split the computations evenly for polynomial systems over a prime field Z/pZ than over $\mathbb{Q}$. To take advantage of this, we use the modular algorithm described in Chapter 4. Moreover, the features of the Triade algorithm (in particular the fact that it generates the intermediate and output components by decreasing order of dimension) allow us to exploit the geometrical information and achieve load balancing. We have also strengthened the task model of Triade in order to estimate the costs of intermediate computations and thus to guide the parallel scheduling. This joint work with M. Moreno Maza is published in [111].

The implementation of this component-level parallel solver is yet another challenge. To serve this main purpose, we develop a high-level categorical parallel framework, written in the ALDOR language, to support high-performance computer algebra on symmetric multi-processors and multicore processors. We describe this work in detail in Chapter 6. This framework provides functions for dynamic process management and data communication and synchronization via shared memory segments. A simple interface for user-level scheduling is also provided. Packages are developed for serializing and de-serializing high-level ALDOR objects, such as sparse multivariate polynomials, into arrays of machine integers for data transfer. Our benchmark perfor-

mance results show this framework is efficient in practice for coarse-grained parallel symbolic computations. This joint work with M. Moreno Maza, B. Stephenson and S.M. Watt is reported in [110].

A component-level parallel solver for triangular decompositions has been realized by using the above framework in conjunction with the `BasicMath` library and the Triade sequential solver in ALDOR. Our experimentation shows promising speedups for some well-known problems. Indeed, for most systems this speedup is equal to the number of components with maximum degree, in the modular triangular decomposition. We expect that the speedup obtained at this component-level will add a multiplicative factor to the speedup of medium/fine grained level parallelization as parallel GCD and resultant computations.

Software engineering. There are special difficulties in the conception and implementation of polynomial solvers based on triangular decompositions. As such, software engineering plays an important role in the development of such solvers.

First of all, code validation and data structure design are extremely hard due to the sophisticated specifications and algorithms for computing triangular decompositions. For instance, the representation of a triangular decomposition is a list of lists of polynomials with special properties. The data structures and the techniques used to decide what information to cache can greatly affect the computation speed. It is also hard to specify the results since the same polynomial input has different possible outputs with varied benefits. Even worse, in the case of an infinite number of solutions there is no canonical form of the output. As mentioned earlier, in a computer algebra system, the *solve* command usually invokes almost all of the functions operating on matrices and polynomials in this software. Therefore, the prototyping and code validation of polynomial system solvers is a significant challenge.

On the other hand, different user communities use computer algebra systems for different purposes. Common uses include teaching, advanced research and high performance computing. The prototyping of algorithms and sub-routines, and their accessibility and ease of use for non-expert users are true challenges. These must consider the characteristics of the implementation environment, the level of expertise of the users, and the expectations of its user community.

In Chapter 7, we discuss our solutions and illustrate them with the implementation of the Triade algorithm in three computer algebra systems: AXIOM, ALDOR, and MAPLE. This joint work with F. Lemaire and M. Moreno Maza is published in [91] and [92]. In each case a different community of users was targeted. The

`RegularChains` library in MAPLE provides quite a large set of functions targeting a diverse group of users. It provides the ability to both solve and manipulate polynomials and regular chains, and the ability to perform computations modulo a set of algebraic equations. The functions with optional arguments are organized into two-level three modules, each providing user-friendly interfaces for both expert and non-expert users. The AXIOM implementation is general and very close to the mathematical theory, which is powerful and flexible, but more for experts. The ALDOR implementation focuses on high-performance with ease of interfacing with machine resources. All the implementations are equipped with large test suites and examples. We believe that these implementations of the same sophisticated mathematical algorithms for different communities of experts, advanced users, and non-experts is a unique experience in the area of symbolic computations which could benefit other algorithms in this field.

Triangular decompositions also have to face the problem of *removing redundant components*. As mentioned earlier, this is a common issue with all the symbolic decomposition algorithms such as those of [86, 88, 146] and in numerical approaches [126]. If the redundant computations can be detected efficiently and cut at an early stage of the solving process, performance will be improved. Removing redundant components is also a demand in the stability analysis of dynamical systems [137].

In Chapter 8, we present new functionality that we have added to the `RegularChains` library in MAPLE to efficiently compute irredundant triangular decompositions, and reports on the implementation of our different strategies. These strategies use inclusion tests of quasi-components, which rely on the `RegularChains` library, without computation of Gröbner basis. Unproved algorithms for this inclusion test are stated in [86] and [107]. They appear to be unsatisfactory in practice, since they rely on normalized regular chains, which tend to have much larger coefficients than non-normalized regular chains as verified experimentally in [9] and formally proved in [42]. We use a *divide and conquer* approach to efficiently remove the redundant components in a triangular decomposition. Our experiments show that, for difficult input systems, the computing time for removing redundant components can be reduced to a small portion of the total time needed for solving these systems. This joint work with M. Moreno Maza, Changbo Chen, F. Lemaire and Wei Pan is published in [31].

Another difficulty is the verification of the output of triangular decompositions. Given a polynomial system F and a set of components $C_1, \ldots, C_e$, it is hard, in

general, to tell whether the union of $C_1, \ldots, C_e$ corresponds exactly to the solution set $V(F)$ or not. Actually, solving this verification problem is generally (at least) as hard as solving the system F itself.

Because of the high complexity of symbolic solvers, developing verification algorithms and reliable verification software tools is clearly needed. However, this verification problem has received little attention in the literature. In Chapter 9 we present a new approach to the problem which computes the set theoretical differences between two constructable sets. The key idea is to verify the output of a solver by comparing it with the output of a known reliable solver.

Our method is implemented on top of the `RegularChains` library in MAPLE. We also realized a standard verification tool based on Gröbner basis computations. We provide comparative benchmarks of different verification procedures applied to four solvers on a large set of well-known polynomial systems. Standard verification techniques are highly resource consuming and apply only to polynomial systems which are easy to solve. The experimental results illustrate the high efficiency of our new approach. In particular, we are able to verify triangular decompositions of polynomial systems which are not easy to solve. This joint work with M. Moreno Maza, Changbo Chen and Wei Pan is published in [31].

In summary, this thesis contributes to polynomial system solving in four related fields of study: fast algorithms, modular methods, parallel approaches and software engineering. We have adapted fast algorithms for polynomial arithmetic over fields to direct products of fields represented by triangular sets and obtained a complexity study. These algorithms provide efficient low-level routines for computing triangular decompositions. The equiprojectable decomposition introduced here is canonical and has good computational properties. This makes it possible to design sharp modular methods for triangular decompositions, and it is the only such algorithm at the present time. A byproduct of this work is a series of functions implemented in MAPLE for linear algebra over non-integral domains with automatic case discussion and case recombination. This is a unique feature in the MAPLE computer algebra system. The component-level parallelization and the parallel framework developed in ALDOR take the first step toward efficient multi-level parallel computation for triangular decompositions on emerging architectures. The tools for efficient computation of irredundant components and verifying the output of solvers are useful for both developers and users. The techniques used to implement the Triade algorithm in three computer algebra systems for different user groups (i.e. general, advanced research and high-performance) could benefit other areas of scientific computing.

Chapter 2

Background

This chapter has two objectives. First, to introduce to the reader the notions of a triangular decomposition and of a regular chain. These are the fundamental concepts used through this thesis. Sections 2.1 and Section 2.2 introduce them in an informal manner and highlight their main properties in non-technical language.

The second objective is to define formally these two concepts and state their main properties. We also present important auxiliary notions such as algebraic variety (Section 2.3), Gröbner basis (Section 2.4), and triangular set (Section 2.5). Moreover, we review two fundamental algorithms for solving systems of polynomial equations: *Buchberger's Algorithm* and *Wu's Algorithm*. They are at the foundation of many of the other methods for this purpose. We will also refer to them later in this thesis when we discuss the parallelization of polynomial system solvers.

Regular Chains (Section 2.6) are triangular sets with additional properties. Section 2.7 presents an algorithm called Triade for computing `triangular decomposition` by means of regular chains. This algorithm is involved in almost all chapters of this thesis, which explains why we discuss its specifications, its main features and some of its sub-procedures. However, we refer to [108] for a complete exposition together with proofs.

2.1 Triangular decomposition: An introduction

For a given input system of equations, say from some high-school problem, it is desirable to write down explicit formulas for each of the solutions of this system. This objective has to be moderated by a variety of considerations. Let us give two fundamental ones. First, for univariate polynomial equations with degree higher than

4, solutions may not be always expressed by "explicit formulas" based on radicals. Second, a system of equations may have infinitely many solutions.

Symbolic computations offer different solutions to overcome these difficulties. One can replace the input system of equations F by another system of equations G such that both systems have the same set of solutions V and such that G reveals important information about V, such as its cardinality. This point of view is developed in the theory of *Gröbner bases*. One can also wish to group solutions into meaningful "components" described by "implicit formulas". This is the motivation in the theory of *triangular decompositions*.

Let us illustrate these two approaches from an example. Consider the polynomial system F_1 with ordered variable $x > y > z$. This ordering indicates the fact that we aim at expressing y as a function of z, and x as a function of y and z.

$$\begin{cases} x^3 - 3x^2 + 2x & = 0 \\ 2yx^2 - x^2 - 3yx + x & = 0 \\ zx^2 - zx & = 0 \end{cases}$$

A Gröbner basis of F_1 is:
$$\begin{cases} x^2 - xy - x \\ -xy + xy^2 \\ zxy \end{cases}$$

and triangular decomposition:

$$\left\{ x = 0 \right. \quad \bigcup \quad \begin{cases} x = 1 \\ y = 0 \end{cases} \quad \bigcup \quad \begin{cases} x = 2 \\ y = 1 \\ z = 0 \end{cases}$$

The geometric view of the triangular decomposition of the above polynomial system F_1 is illustrated by Figure 2.1. It is clearly shown that $V(F_1)$ consists of one point $(x = 2, y = 1, z = 0)$, one line $(x = 1, y = 0)$, and one plane $(x = 0)$. This example illustrates the fact that triangular decompositions can reveal geometric information about the solution set of a polynomial system. This is achieved by "grouping" solution points into meaningful components, such as points, curves and surfaces.

A given input polynomial system may admit different triangular decompositions. This leads to implementation challenges, in particular for systems with infinitely many solutions, as we shall discuss in Chapters 7, 8 and 9. However, for a system F with finitely many solutions, for a fixed variable ordering, there is a canonical triangular
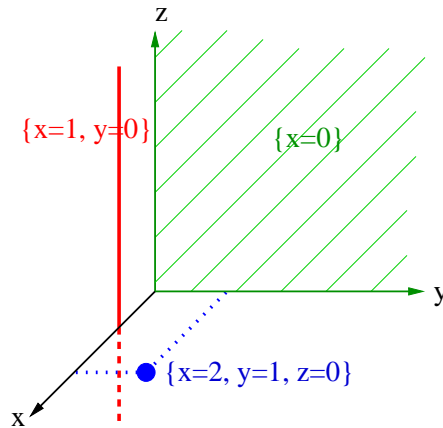
Figure 2.1: A Geometric View of the Triangular Decomposition of F_1

decomposition, called the *Equiprojectable Decomposition*. This will be discussed in Chapter 4.

Let us illustrate this remark with the following input polynomial system F_2,

$$F_2 : \begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases}$$

One possible triangular decomposition of the solution set of F_2 is:

$$\begin{cases} z = 0 \\ y = 1 \\ x = 0 \end{cases} \bigcup \begin{cases} z = 0 \\ y = 0 \\ x = 1 \end{cases} \bigcup \begin{cases} z = 1 \\ y = 0 \\ x = 0 \end{cases} \bigcup \begin{cases} z^2 + 2z - 1 = 0 \\ y = z \\ x = z \end{cases}$$

Another one is:

$$\begin{cases} z = 0 \\ y^2 - y = 0 \\ x + y = 1 \end{cases} \bigcup \begin{cases} z^3 + z^2 - 3z = -1 \\ 2y + z^2 = 1 \\ 2x + z^2 = 1 \end{cases}$$

Both results are valid. The second one is the equiprojectable decomposition. As a matter of fact, the second one can be computed from the first one by techniques explained in Chapter 4.

Although triangular decompositions display rich geometrical information, reading them requires some familiarity, especially when there is an infinite number of solutions. Let us consider the following polynomial system F_3 where the ordered variables are $s_1 > c_1 > s_2 > c_2 > b > a$:

$$F_3 : \begin{cases} c_1\,c_2 - s_1\,s_2 + c_1 - a &= 0 \\ s_1\,c_2 + c_1\,s_2 + s_1 - b &= 0 \\ c_1^2 + s_1^2 - 1 &= 0 \\ c_2^2 + s_2^2 - 1 &= 0 \end{cases}$$

This system is a particular case of the inverse kinematics problem in robotics [66]. The quantities a and b are positive real numbers whereas c_1, s_1, c_2, s_2 are unknown sines and cosines. A triangular decomposition of F_3 is given by the following two triangular systems T_1 and T_2:

$$T_1 : \begin{cases} (a^2 + b^2)\,s_1 + 2s_2c_1 - 2b &= 0 \\ (2a^2 + 2b^2)\,c_1 - 2bs_2 - a^3 - b^2a &= 0 \\ 4s_2{}^2 + a^4 + (2b^2 - 4)\,a^2 + b^4 - 4b^2 &= 0 \\ 2c_2 - a^2 - b^2 + 2 &= 0 \end{cases} ,$$

$$T_2 : \begin{cases} s_1{}^2 + c_1{}^2 - 1 &= 0 \\ s_2 &= 0 \\ c_2 + 1 &= 0 \\ a &= 0 \\ b &= 0 \end{cases}$$

Note that at this point we have not given yet a formal definition of a triangular decomposition or a triangular system. For the reader to be familiar with these objects, by triangular system, we mean quasi-component of a triangular set and by triangular decomposition we mean a finite set of triangular systems.

Let us return to our example and have a closer look at T_1. We see that we can determine from it the unknowns c_1, s_1, c_2, s_2 as functions of the parameters a and b. We define

$$C(a, b) = a^4 + (2b^2 - 4)\,a^2 + b^4 - 4b^2.$$

Since $4s_2{}^2 + C(a, b) = 0$ holds we must have $C(a, b) \leq 0$. Observe that $C(a, b)$ factorizes as follows:

$$C(a, b) = (a^2 + b^2 - 4)\,(a^2 + b^2).$$

Thus we get the inequality constraint: $a^2 + b^2 \leq 4$. Since each of the other three equations is linear w.r.t. the unknown it defines, we have proved that the triangular system T_1 has solutions with real coordinates if and only if $a^2 + b^2 \leq 4$ holds. It is natural to check the extreme cases where $a^2 + b^2 = 4$ and $a^2 + b^2 = 0$.

Let us start with $a^2 + b^2 = 4$. This particular case is actually covered by our

triangular system T_1. That is, adding this constraint to the input system F_3 or adding it to T_1 leads to the same triangular system, namely:

$$\left\{ \begin{array}{rcl} a^2 + b^2 - 4 & = & 0 \\ s_2 & = & 0 \\ c_2 + 1 & = & 0 \\ 2c_1 - a & = & 0 \\ 2s_1 - b & = & 0 \end{array} \right.$$

Let us continue with $a^2 + b^2 = 0$. Adding this constraints to T_1 and performing elementary transformations, we obtain:

$$\left\{ \begin{array}{rcl} a & = & 0 \\ b & = & 0 \\ s_2 & = & 0 \\ c_2 + 1 & = & 0 \end{array} \right.$$

This cannot be correct since the constraints on c_1 and s_1 have disappeared! On the contrary, if we add $a^2 + b^2 = 0$ to the input system, we obtain actually the triangular system T_2 of our initial triangular decomposition, that is:

$$T_2 : \left\{ \begin{array}{rcl} s_1{}^2 + c_1{}^2 - 1 & = & 0 \\ s_2 & = & 0 \\ c_2 + 1 & = & 0 \\ a & = & 0 \\ b & = & 0 \end{array} \right.$$

We are ready now to explain how to read our triangular decomposition $\{T_1, T_2\}$.

For the imposed variable ordering $s_1 > c_1 > s_2 > c_2 > b > a$, each polynomial equation should be regarded as a univariate one w.r.t. its largest variable. Hence, the first polynomial equation

$$\left(a^2 + b^2 \right) s_1 + 2s_2c_1 - 2b = 0.$$

from T_1 defines s_1 as

$$s_1 = \frac{-2s_2c_1 + 2b}{a^2 + b^2}$$

which requires, of course, the condition $a^2 + b^2 \neq 0$. Similarly, the second polynomial

equation

$$\left(2a^2 + 2b^2\right) c_1 - 2bs_2 - a^3 - b^2 a$$

from T_1 defines s_1 as

$$c_1 = \frac{2bs_2 + a^3 + b^2 a}{2a^2 + 2b^2}$$

under the same constraints. The last two equations from T_1 define s_2 and c_2 respectively, without any additional constraints. Therefore, we can conclude that

- for all values of the parameters a, b, satisfying $a^2 + b^2 \neq 0$ the unknowns c_2, s_2, c_1, s_1 are given by the 4th, the 3rd, the 2nd and the 1st equations from the triangular system:

$$T_1 : \begin{cases} \left(a^2 + b^2\right) s_1 + 2s_2 c_1 - 2b &= 0 \\ \left(2a^2 + 2b^2\right) c_1 - 2bs_2 - a^3 - b^2 a &= 0 \\ 4s_2{}^2 + a^4 + \left(2b^2 - 4\right) a^2 + b^4 - 4b^2 &= 0 \\ 2c_2 - a^2 - b^2 + 2 &= 0 \end{cases},$$

- under the constraints $a^2 + b^2 = 0$, the values of c_2, s_2, c_1, s_1 are given by the triangular system:

$$T_2 : \begin{cases} s_1{}^2 + c_1{}^2 - 1 &= 0 \\ s_2 &= 0 \\ c_2 + 1 &= 0 \\ a &= 0 \\ b &= 0 \end{cases}.$$

We make some additional remarks.

- In the first triangular system, the unknown s_2 is defined "implicitly" as one of the roots of a degree 2 polynomial.

- even if either $a^2 + b^2 < 0$ or $a^2 + b^2 > 4$ holds, the first triangular system provides values for c_2, s_2, c_1, s_1; however, some of these values are not real.

Before moving to more formal definitions, we would like to address informally the following question. Let T be a triangular system. Is it possible that T defines no values at all for the unknowns, for instance, because the inequality constraints are too restrict. As shown in the example below, this can happen. This means that not all triangular systems are good and that it is necessary to strengthen the notion of a triangular system in order to avoid such troubles. This leads to the notion of a *regular chain*.

Consider the following triangular systems for the ordering $x_1 < x_2 < x_3 < x_4$:

$$T : \begin{cases} x_2^2 - x_1 & = & 0 \\ x_3^2 - 2x_2x_3 + x_1 & = & 0 \\ (x_3 - x_2)x_4 & = & 0 \end{cases},$$

$$T_1 : \begin{cases} x_2^2 - x_1 & = & 0 \\ x_3 - x_2 & = & 0 \end{cases}.$$

Let us focus on T. The equation $x_2^2 - x_1 = 0$ defines x_2 as a square root (potentially complex or real) of x_1. Thus, this does not impose any constraints on x_1. Under the condition $x_2^2 = x_1$, the equation $x_3^2 - 2x_2x_3 + x_1 = 0$ becomes $x_3^2 - 2x_2x_3 + x_2^2 = 0$, that is, $(x_3 - x_2)^2 = 0$, which implies $x_3 - x_2 = 0$. The third equation, namely $(x_3 - x_2)x_4 = 0$, defines x_4 provided that $x_3 - x_2 \neq 0$ holds. Thus, the equations and inequalities of T are contradictory! Such a triangular system is called *inconsistent*.

Now, assume that T is not regarded as a triangular system but just as a system of equations. In this case $x_3 - x_2 \neq 0$ does not need to hold. What should be then a triangular decomposition of the solution set of T? Since $x_3 - x_2 = 0$ holds anyway, one can simply discard the third equation of T and obtain T_1. Clearly, this latter triangular system is not inconsistent and $\{T_1\}$ is a triangular decomposition of the solution set of T.

2.2 Regular Chains: An introduction

Performing calculations modulo a set of relations is a basic technique in algebra. For instance, computing the inverse of an integer modulo a prime integer or computing the inverse of the complex number $3 + 2\imath$ modulo the relation $\imath^2 + 1 = 0$ defining $\imath$.

Computing modulo a set F containing more than one relation can be much less simple. For instance, how to compute the inverse of $p = x + y$ modulo the following set F_4?

$$F_4 : \begin{cases} x^2 + y + 1 & = & 0 \\ y^2 + x + 1 & = & 0 \end{cases}$$

Things become much simpler when one realizes that this question is equivalent to computing the inverse of p modulo

$$C_0 : \begin{cases} x^4 + 2x^2 + x + 2 & = & 0 \\ y + x^2 + 1 & = & 0 \end{cases}$$

Indeed, we can transform F_4 into C_0, replacing y by $-x^2 - 1$ into $y^2 + x + 1 = 0$. Conversely, we can transform C_0 into F_4, replacing x^2 by $-y-1$ into $x^4 + 2x^2 + x + 2 = 0$. With the system C_0, inverting p becomes easier. Indeed, one can simplify p into $q := -x^2 + x - 1$ using the relation $y = -x^2 - 1$. Now p, or rather its simplified version q, is a univariate polynomial in x, which can be easily inverted modulo the relation $x^4 + 2x^2 + x + 2 = 0$. It suffices to use the extended Euclidean algorithm [57] and one can verify that $p^{-1} := -\frac{1}{2}x^3 - \frac{1}{2}x$ is the inverse of q modulo $x^4 + 2x^2 + x + 2 = 0$. The moral of this example is that the "triangular shape" of C_0 has made easier the inversion of p modulo F_4.

One commonly used mathematical structure for a set of algebraic relations is that of a *Gröbner basis* [13]. It is particularly well suited for deciding whether a quantity is null or not modulo a set of relations. For inverse computations, the notion of a *regular chain* is more adequate. To illustrate this remark, let us compute the inverse of $p = y + x$ modulo the set

$$C : \begin{cases} x^2 - 3x + 2 &= 0 \\ y^2 - 2x + 1 &= 0 \end{cases}$$

which is both a Gröbner basis and a regular chain for the variable order of $y > x$. Here, we cannot simplify p into a univariate polynomial as we did before. However, the theory of regular chains provides us with a general notion of GCD which can be used to solve our problem. Actually, one can compute a GCD of p and $C_y = y^2 - 2x + 1$ modulo the relation $C_x = 0$, with $C_x = x^2 - 3x + 2$. As we shall see in Chapter 3, such GCD modulo a regular chain is a piecewise function. In our example, it distinguishes the cases $x = 1$ and $x = 2$ since the result has a different shape in each case.

- For $x = 1$, the polynomials p and C_y simplify to $y + 1$ and $y^2 - 1$ respectively, leading to $y + 1 = p$ as GCD.

- For $x = 2$, the polynomials p and C_y simplify to $y + 2$ and $y^2 - 3$, respectively leading to 1 as a GCD, since $y + 2$ and $y^2 - 3$ have no common factors.

To summarize, we have:

$$\text{GCD}(p, C_y, \{C_x\}) = \begin{cases} p & \text{if} \quad x = 1 \\ 1 & \text{if} \quad x = 2 \end{cases}$$

This shows that p has no inverse if $x = 1$ and has an inverse (which can be computed and which is $-y + 2$) if $x = 2$.

The notion of a regular chain was introduced by *Kalkbrener* in [76] and extended that of a *triangular set* (as defined in [87]). It will be defined formally later in this chapter. Kalkbrener pointed out, "since every irreducible variety is uniquely determined by one of its generic points, we represent varieties by representing the generic points of their irreducible components. These generic points are given by certain polynomial subsets, so-called regular chains". The common roots of any set of multivariate polynomials F (with coefficients in a field $\mathbb{K}$) can be decomposed into a finite union of regular chains. Because of the triangular shape of a regular chain, such decomposition is called a *triangular decomposition.*

In 1987, Wen Tsun Wu [146] introduced the first method for solving systems of algebraic equations by means of triangular decompositions. The decompositions computed by Wu's method may contain inconsistent or redundant characteristic sets. That is, a triangular decomposition $\mathbf{T}$ of an input system F, produced by Wu's method, may contain a triangular system C which does not contribute anything (inconsistency) or which contribution is entirely contained in that of another triangular system C' of $\mathbf{T}$ (redundancy).

The inconsistency problem was solved by Kalkbrener [76] who proposed an algorithm for computing triangular decompositions by means of regular chains. The redundancy problem has been considered by Wang [138] and Lazard [86]. However, comparative experiments implemented in AXIOM [72] and reported in [9] show better performance for Kalkbrener's algorithm.

In [108], Moreno Maza proposed a new algorithm, called Triade, for computing triangular decompositions with an emphasis on the management of the intermediate computations. One strength of this approach is that redundant branches are easy to cut and can be cut at an early stage of the computations. In addition, this algorithm shows better performance than the other algorithms with similar specifications, as reported in [9, 32, 30].

2.3   Algebraic Varieties

Throughout this thesis, we consider a field $\mathbb{K}$ and an ordered set $X = x_1 < \cdots < x_n$ of n variables. We denote by $\mathbb{K}[x_1, \ldots, x_n]$ the ring of the polynomials with coefficients in $\mathbb{K}$ and variables in X. Let $\overline{\mathbb{K}}$ be an algebraic closure of $\mathbb{K}$. The reader may think of $\mathbb{K}$ and $\overline{\mathbb{K}}$ as $\mathbb{R}$ and $\mathbb{C}$, that is, the fields of real and complex numbers respectively. In practice, our polynomials will have coefficients in the field $\mathbb{Q}$ of rational numbers.

Sometimes, given a prime integer p, the coefficient field $\mathbb{K}$ will be the field $\mathbb{Z}/p\mathbb{Z}$ of integers modulo p.

Let $F \subset \mathbb{K}[x_1, \ldots, x_n]$ be a set of polynomials. A point $(\zeta_1, \ldots, \zeta_n)$ in $\overline{\mathbb{K}}^n$ is called a *common zero*, or *zero*, or *solution*, or *common root*, or *root* of F if every $f \in F$ evaluates to zero at $(\zeta_1, \ldots, \zeta_n)$. The set of all common roots of F is denoted by $V(F)$ and called the *zero set* of F, or *solution set* of F, or the *algebraic variety* defined by F.

It is well known [36], and not difficult to check, that the set of all algebraic varieties defined by polynomial sets of $F \subset \mathbb{K}[x_1, \ldots, x_n]$ are the closed sets of a topology called the *Zariski topology* of $\overline{\mathbb{K}}^n$ w.r.t. $\mathbb{K}$. Given a subset $W \subset \overline{\mathbb{K}}^n$ we denote by $\overline{W}$ the *Zariski closure* of W w.r.t. $\mathbb{K}$, that is, the closure of W for Zariski topology w.r.t. $\mathbb{K}$, that is, the intersection of all the $V(F)$ (for all $F \subset \mathbb{K}[x_1, \ldots, x_n]$) containing W.

Zariski topology plays an essential role in solving systems of polynomial equations, in particular when triangular decompositions are involved. Remember that the solution set $W(T)$ associated with a triangular system T is given by equations and inequalities. Thus $W(T)$ is not necessarily an algebraic variety and it is natural to manipulate its closure $\overline{W(T)}$. We discuss two examples below.

For $n = 3$ and $x_1 < x_2 < x_3$, consider the polynomial system F consisting of the single equation $x_1 x_3 + x_2 = 0$, with real coefficients. Let $V(F)$ be its algebraic variety in $\mathbb{C}^3$. As we shall see later, this set F is also a regular chain C. Because of the variable ordering, the zero set $W(C)$ consists of the points (x_1, x_2, x_3) for which $x_1 x_3 + x_2 = 0$ and $x_1 \neq 0$ hold.

We formally prove below that $W(C)$ cannot be an algebraic variety. The reader may rely on his or her geometrical intuition. The set $V(F)$ is an irreducible algebraic variety of dimension 2. (Indeed, the polynomial $x_1 x_3 + x_2$ is irreducible.) Hence any algebraic variety contained in $V(F)$ must have a smaller dimension, that is, 1 or 0. Therefore, if $W(C)$ is an algebraic variety, it must have dimension 1 or 0.

Observe that for each nonzero value ζ_1 of x_1 the set $W(C)$ contains the line defined by $\zeta_1 x_3 + x_2 = 0$. All these lines are irreducible algebraic varieties of dimension 1; none of them is contained in the union of the others; moreover they are all contained in $W(C)$. Since every algebraic variety can be decomposed into finitely many irreducible components, the set $W(C)$ cannot be an algebraic variety of dimension 1 or 0. Therefore, the set $W(C)$ is not an algebraic variety.

In fact, the algebraic variety $V(F)$ is the union of $W(C)$ and the line given by $x_1 = x_2 = 0$. In other words $W(C)$ is $V(F)$ minus that line. What is $\overline{W(C)}$ then?

By definition of the Zariski closure, we have

$$W(C) \subseteq \overline{W(C)} \subseteq V(C).$$

Since $\overline{W(C)}$ is a variety, it follows from our previous reasoning that it cannot have dimension 1 or 0. Since $V(F)$ is an irreducible variety of dimension 2 we must have $\overline{W(C)} = V(F)$.

More generally, if $V(F)$ is an irreducible variety of arbitrary dimension and F is a regular chain then we have $\overline{W(F)} = V(F)$. This would not hold if $V(F)$ were not irreducible. Consider now $n = 4$, $x_1 < x_2 < x_3 < x_4$ and

$$F = \{x_1 x_4 - x_2, x_2 x_3 - x_1, x_2^3 - x_1^2\}.$$

One can check that F is a regular chain C_1. However, the variety $V(F)$ is not irreducible. To understand this, let us consider first the points (x_1, x_2, x_3, x_4) of $V(F)$ for which $x_1 \neq 0$ holds. Then these points have the following form

$$(x_1, x_1^{2/3}, x_1/x_2, x_2/x_1) = (x_1, x_1^{2/3}, x_1^{1/3}, x_1^{-1/3}).$$

Hence, these points are contained in an algebraic curve V_1 parametrized by x_1. These points form $W(C_1)$ and thus we have $\overline{W(C_1)} \subseteq V_1$. (In fact, equality holds.) Now consider the points (x_1, x_2, x_3, x_4) of $V(F)$ for which $x_1 = 0$ holds. Observe that $x_1 = 0$ implies $x_2 = 0$ and that all polynomials in F vanish when $(x_1, x_2) = (0, 0)$ holds. Therefore, we have proved that $V(F)$ decomposes into two algebraic varieties

$$V(F) = V_1 \cup V_2$$

where V_2 is the linear variety of dimension 2 given by $x_1 = x_2 = 0$. Hence, the closure of $W(C_1)$ cannot equal $V(F)$ in this case. Indeed, we have observed that $W(C_1)$ was contained in an algebraic variety of dimension 1 (a curve) whereas $V(F)$ contains a variety of dimension 2.

These examples illustrate the importance of the concept of dimension. We shall not review here this algebraic notion and the reader should rely on her(his) intuition when this concept comes into play. However, we shall review in the next sections the notions of a Gröbner basis, a characteristic set and a regular chain. We refer to [36] for the former notion and to [8, 22] for the latter ones. These are the key objects

computed by the algorithms discussed in this thesis. We conclude this section by reviewing a fundamental result: *Hilbert's Theorem of Zeros.*

Let again $F = \{f_1, \ldots, f_m\}$ be an arbitrary finite set of polynomials in $\mathbb{K}[x_1, \ldots, x_n]$. 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 $\langle F \rangle$ is not equal to the entire polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$, then it is said to be a *proper ideal.*

Let $G = \{g_1, \ldots, g_s\}$ be another finite subset of $\mathbb{K}[x_1, \ldots, x_n]$. The following implication is easy to check:

$$\langle F \rangle = \langle G \rangle \quad \Rightarrow \quad V(F) = V(G). \tag{2.1}$$

What can we say about $\langle F \rangle$ and $\langle G \rangle$ when $V(F) = V(G)$ holds? One should expect that $\langle F \rangle = \langle G \rangle$ would not hold necessarily. Consider the following trivial example with $n = 1$, $F = \{x_1\}$ and $G = \{x_1^2\}$. Clearly every multiple of x_1^2 is a multiple of x_1 and we have $\langle G \rangle \subseteq \langle F \rangle$; but, clearly again, x_1 is not a multiple of x_1^2 and $\langle F \rangle \subseteq \langle G \rangle$ does not hold. Therefore, we need a notion that would be like the "square root" of an ideal.

The *radical* of the ideal generated by F, denoted by $\sqrt{\langle F \rangle}$, is the set of the 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 set $\sqrt{\langle F \rangle}$ is not obtained from $\langle F \rangle$ by simply removing repeated factors in the polynomials of F. These "multiplicities" can be hidden as shown below. Consider (again) for $n = 4$ the polynomial set $F = \{f_1, f_2\}$ with $f_1 = x_2^2 - x_1$, $f_2 = x_3^2 - 2x_2 x_3 + x_1$. It is not difficult to show (by means of degree considerations) that the polynomial $f_3 = x_3 - x_2$ cannot be expressed as a combination of the form $h_1 f_1 + h_2 f_2$ and thus $f_3 \notin \langle f_1, f_2 \rangle$. However we have

$$f_3^2 = (x_3 - x_2)^2 = x_3^2 - 2x_2 x_3 + x_2^2 = f_2 - f_1.$$

Hence, we have $f_3 \in \sqrt{\langle f_1, f_2 \rangle}$. Finally, one can check that we have

$$\sqrt{\langle f_1, f_2 \rangle} = \langle -x_3^2 + x_1, x_2 - x_3 \rangle.$$

This example suggests that computing $\sqrt{\langle F \rangle}$ can be far from trivial in general. In

fact, triangular decompositions can be used to perform such computations, thanks to *Hilbert's Strong Theorem of Zeros* that we can state now.

Theorem 2.3.1. *For all subsets $F, G \subseteq \mathbb{K}[x_1, \ldots, x_n]$ we have:*

$$\sqrt{\langle F \rangle} = \sqrt{\langle G \rangle} \quad \Longleftrightarrow \quad V(F) = V(G).$$

This theorem establishes a one-to-one correspondence between radical ideals of $\mathbb{K}[x_1, \ldots, x_n]$ and algebraic varieties of $\overline{\mathbb{K}}[x_1, \ldots, x_n]$. In abstract algebra textbooks [34], it is usually proved after establishing the results below, the first one being known as the *Hilbert's Strong Theorem of Zeros*. These statements are interesting for themselves and they are the basis of several algorithms. The first one says that the algebraic variety $V(F)$ is empty if and only 1 belongs to $\langle F \rangle$, the ideal generated by F. The second one implies that testing the membership of a polynomial h to $\sqrt{F}$ reduces to testing the membership of 1 to the ideal generated by F and $1 - yh$ where y is a new variable.

Theorem 2.3.2. *For all subset $F \subseteq \mathbb{K}[x_1, \ldots, x_n]$ we have:*

$$1 \in \langle F \rangle \quad \Longleftrightarrow \quad V(F) = \emptyset.$$

Theorem 2.3.3. *Let y a new variable. For all $h, f_1, \ldots, f_n \in \mathbb{K}[x_1, \ldots, x_n]$ we have:*

$$h \in \sqrt{\langle f_1, \ldots, f_n \rangle} \quad \Longleftrightarrow \quad \langle f_1, \ldots, f_n, 1 - yh \rangle = \mathbb{K}[x_1, \ldots, x_n, y].$$

2.4 Gröbner Bases

Recall that the variables are ordered: $x_1 < \cdots < x_n$. Let $M = \{x_1^{i_1} \ldots x_n^{i_n} \mid i_j \geq 0\}$ be the abelian monoid consisting of the monomials generated by X, that is, all possible products of the variables. We denote by 1 the neutral element of M.

A total order relation $\leq$ on M is an *admissible monomial order* on M, if it satisfies:

$$1 \leq u \quad \text{and} \quad u \leq v \Rightarrow uw \leq vw$$

for all $u, v, w \in M$. A fundamental example is the *lexicographical order* $\leq_{\text{lex}}$ defined as follows: we have

$$x_1^{i_1} \ldots x_n^{i_n} \leq_{\text{lex}} x_1^{j_1} \ldots x_n^{j_n}$$

if and only if there exists an integer e in the range $1..n$ such that we have $i_e < j_e$ and $i_k = j_k$ for all k in $(e+1)..n$. For $n = 2$, let us order a few monomials lexicographically:

$$x_1 \leq_{\text{lex}} x_1^2 \leq_{\text{lex}} \cdots \leq_{\text{lex}} x_2 \leq_{\text{lex}} x_1 x_2 \leq_{\text{lex}} x_1^2 x_2 \leq_{\text{lex}} \cdots \leq_{\text{lex}} x_2^2 \leq_{\text{lex}} x_1 x_2^2 \leq_{\text{lex}} x_1^2 x_2^2 \leq_{\text{lex}} \cdots$$

Let $f \in \mathbb{K}[x_1, \ldots, x_n]$ be a non-zero polynomial. We denote by $\text{lm}(f)$ the *leading monomial* of f, that is, the monomial of f with the highest order w.r.t. $\leq$. We denote by $\text{lc}(f)$ the *leading coefficients* of f, that is, the coefficient of $\text{lm}(f)$ in f. We define $\text{lt}(f) = \text{lc}(f)\text{lm}(f)$, called the *leading term* of F. For all $F \subset \mathbb{K}[X]$, we write $\text{lm}(F) = \{\text{lm}(f) \mid f \in F\}$.

We say that a polynomial $f \in \mathbb{K}[X]$ is *reduced* w.r.t. $g \in \mathbb{K}[X]$, with $g \neq 0$, if $\text{lm}(g)$ does not divide any monomial in f. Let $b_1, \ldots, b_k \in \mathbb{K}[X]$ be non-constant polynomials. If f is not reduced w.r.t. one polynomial among $b_1, \ldots, b_k \in \mathbb{K}[X]$, then one can "replace" f by a reduced polynomial r equal to f modulo the ideal generated by $b_1, \ldots, b_k \in \mathbb{K}[X]$. The following proposition states this fact more formally.

Proposition 2.4.1. *There exists an operation* Divide *such that* $\text{Divide}(f, \{b_1, \ldots, b_k\})$ *returns polynomials* $r, q_1, \ldots, q_k \in \mathbb{K}[X]$ *with the following properties:*

(*i*) $f = q_1 b_1 + \cdots + q_k b_k + r$,

(*ii*) r *is reduced w.r.t. all* $b_1, \ldots, b_k \in \mathbb{K}[X]$,

(*ii*) $\max(\text{lm}(q_1)\text{lm}(b_1), \ldots, \text{lm}(q_k)\text{lm}(b_k), \text{lm}(r)) = \text{lm}(f)$.

The polynomial r *is called a* remainder *of* f *w.r.t.* $\{b_1, \ldots, b_k\}$ *and* $q_1, \ldots, q_s$ *are the corresponding* quotients; *moreover we write:*

$$\begin{array}{c|c|c|c} f & b_1 & \cdots & b_k \\ \hline r & q_1 & \cdots & q_k \end{array}.$$

Algorithm 1 implements an operation Divide as specified by Proposition 2.4.1. When $n = 1$, that is, when there is only one variable, this operation is simply the usual polynomial division. In this case, this operation is uniquely defined. For $n > 1$, depending on the order of the polynomials $b_1, \ldots, b_k$, one can get different output, as shown by the following example, with $n = 2$, and the lexicographical order. With $f = y^2 x - x$, $g_1 = yx - y$ and $g_2 = y^2 - x$ one can check that we have

$$\begin{array}{c|c|c} f & g_1 & g_2 \\ \hline 0 & y & 1 \end{array} \quad \text{and} \quad \begin{array}{c|c|c} f & g_2 & g_1 \\ \hline x^2 - x & x & 0 \end{array},$$

This difficulty disappears when the set $\{b_1, \ldots, b_k\}$ is a Gröbner basis, which we shall define after a few more notions. First, we generalize the operation Divide: for

---

Algorithm 1 Multivariate Polynomial Division

Input $f, b_1, \ldots, b_k \in \mathbb{K}[X]$ such that $b_i \neq 0$ for all $i = 1, \ldots, k$.

Output $q_1, \ldots, q_k, r \in \mathbb{K}[X]$ such that $\begin{array}{c|c|c|c} f & b_1 & \cdots & b_k \\ \hline r & q_1 & \cdots & b_k \end{array}$.

$\mathbf{Divide}(f, \{b_1, \ldots, b_k\}) ==$
1: **for** i in $1, \ldots, s$ **do** $q_i \leftarrow 0$
2: $h \leftarrow f;\ \ r \leftarrow 0$
3: **while** $h \neq 0$ **do**
4:    $i \leftarrow 1$
5:    **while** $i \leq s$ **do**
6:       **if** $\text{lm}(b_i) \mid \text{lm}(h)$ **then**
7:          $t \leftarrow \dfrac{\text{lt}(h)}{\text{lt}(b_i)}$
8:          $q_i \leftarrow q_i + t;\ \ h \leftarrow h - tb_i;\ \ i \leftarrow 1$
9:       **else**
10:          $i \leftarrow i + 1$
11:       **end if**
12:    **end while**
13: **end while**
14: $r \leftarrow r + \text{lm}(h)$
15: $h \leftarrow h - \text{lm}(h)$
16: **return** $(q_1, \ldots, q_s, r)$

$a_1, \ldots, a_t, b_1, \ldots, b_k \in \mathbb{K}[X]$, the operation $\text{Reduce}(\{a_1, \ldots, a_t\}, \{b_1, \ldots, b_k\})$ returns the set of remainders of all the $\text{Divide}(a_i, \{b_1, \ldots, b_s\})$ for all $1 \leq i \leq t$.

A subset $B \subset \mathbb{K}[X]$ is said to be *autoreduced*, if for all $f \in B$ the polynomial f is reduced w.r.t. $B \setminus \{f\}$. Dickson's Lemma [35] states that every autoreduced set is finite. From now on, we assume that the elements of every autoreduced set $B = \{b_1, \ldots, b_k\}$ are sorted w.r.t. $\leq$, that is, $\text{lm}(b_1) < \ldots < \text{lm}(b_k)$. Let $B = b_1, \ldots, b_k$, $B' = b'_1, \ldots, b'_l$ be two sorted autoreduced sets. Then, we write $B \leq B'$, if

- either $\exists\, j \leq \min(k, l)$ s.t. $\text{lm}(b_i) = \text{lm}(b'_i)\ (1 \leq i < j)$ and $\text{lm}(b_j) < \text{lm}(b_j)'$,

- or $k \geq l$ and $\text{lm}(b_i) = \text{lm}(b'_i)\ (1 \leq i \leq l)$ holds.

The following holds, see [36]: Every family of autoreduced sets has a minimal element. When this family is finite, it is not difficult to design an algorithm computing such a minimal element. Hence, in Algorithm 2, we refer to an operation $\text{MinimalAutoreducedSubset}(F, \leq)$ returning a subset of F, which is minimal among all autoreduced subsets of F for the order $\leq$.

Definition 2.4.2. For all $F \subset \mathbb{K}[X]$, we call a *Gröbner basis* of F for the (admissible monomial) order $\leq$, any minimal autoreduced subset G contained in the ideal generated by F. In addition, a Gröbner basis G of F is said *reduced* if all leading coefficients in G are equal to 1.

Gröbner bases have numerous important properties, again established in [36].

- Any $F \subset \mathbb{K}[X]$ admits a Gröbner basis G for $\leq$; moreover we have $\langle F \rangle = \langle G \rangle$.

- Any $F \subset \mathbb{K}[X]$ admits a unique reduced Gröbner basis for $\leq$.

- For all $F \subset \mathbb{K}[X]$, for all polynomial $f \in \mathbb{K}[X]$, there exists $r \in \mathbb{K}[X]$ such that for all reduced Gröbner basis G of F w.r.t. $\leq$, we have $\text{Reduce}(f, G) = r$.

- For all $F \subset \mathbb{K}[X]$, the ideal generated by F is the whole ring $\mathbb{K}[X]$ if and only 1 belongs to a Gröbner basis of F.

- For all $F \subset \mathbb{K}[X]$, for all reduced Gröbner basis G of F, for all $p \in \mathbb{K}[X]$, the polynomial p belongs to the ideal generated by F if and only if $\text{Reduce}(p, G) = 0$.

In broad terms, the last point states that the "reduction" w.r.t. to a Gröbner basis is uniquely defined. We conclude this quick review of Gröbner bases with the fundamental algorithm for computing them: the *Buchberger Algorithm* [26], shown as Algorithm 2. This requires the concept of *S-polynomial*. For non-zero polynomials $f, g \in \mathbb{K}[X]$, let L be the least common multiple of $\text{lm}(f)$ and $\text{lm}(g)$. The polynomial

$$S(f, g) = \frac{L}{\text{lm}(f)} f - \frac{L}{\text{lm}(g)} g$$

is called the *S-polynomial* of f and g. By definition, the operation S_Polynomials(F) in Algorithm 2 returns all the $S(f, g)$ of all pairs $\{f, g\}$ of elements of F. The following theorem justifies the correctness of Algorithm 2; its termination follows from the properties of autoreduced sets.

Theorem 2.4.3. *For all $F \subset \mathbb{K}[X]$ a subset G of $\langle F \rangle$ is a Gröbner basis of F if and only if for all $f, g \in G$ we have $\text{Reduce}(S(f, g), G) = 0$.*

Each iteration of this algorithm consists of three steps:

(S)**elect** a candidate Gröbner basis B,

(R)**educe** the elements A w.r.t. B in order to check whether B is a Gröbner basis of F or not,

---

Algorithm 2 Buchberger Algorithm

    Input $F \subset \mathbb{K}[X]$ finite and an admissible monomial ordering $\leq$.

  Output G a reduced Gröbner basis w.r.t. $\leq$ of the ideal $\langle F \rangle$ generated by F.

BuchbergerAlgorithm(F) ==
 1: $B \leftarrow F$; $R \leftarrow F$
 2: **while** $R \neq \emptyset$ **do**
 3:    (S) $B \leftarrow$ MinimalAutoreducedSubset$(F, \leq)$
 4:    (R) $A \leftarrow$ S_Polynomials$(F) \cup F$ $R \leftarrow$ Reduce$(A, B, \leq)$
 5:    (U) $R \leftarrow R \setminus \{0\}$; $F \leftarrow F \cup R$
 6: **end while**
 7: **return** B

(U)**pdate** R and F for the possible next iteration.

When $R = \emptyset$ holds it follows from Theorem 2.4.3 that B is a Gröbner basis of F. Buchberger's Algorithm is a "completion algorithm" similar to many algorithms in the theory of formal languages, like the Knuth-Bendix Algorithm for word problems.

2.5   Triangular Sets

Gröbner bases provide a way to study systems of polynomial equations. In this theory, multivariate polynomials are regarded as combinations of monomials and polynomial sets are regarded as systems of generators of ideals.

    Another point of view is that of characteristic sets where multivariate polynomials are regarded as univariate polynomials with polynomial coefficients and where polynomial sets are regarded as descriptions of algebraic varieties. In this point of view, the notion of "reduction" is different from that in the case of Gröbner bases; it relies on the concept of pseudo-division that we review with the next result [**?**, Yap1993]

Proposition 2.5.1. *Let $\mathbb{A}$ be any commutative ring. Let $a, b \in A[x]$ be univariate polynomials such that b has a positive degree w.r.t. x and the leading coefficient of b is not a zero-divisor. We define $e = \min(0, \deg(a) - \deg(b) + 1)$. Then there exists a unique couple (q, r) of polynomials in $\mathbb{A}[x]$ such that we have:*

$$\operatorname{lc}(b)^e a = qb + r \quad \text{and} \quad (r = 0 \ \text{ or } \ \deg(r) < \deg(b)). \tag{2.2}$$

The polynomial q (resp. r) is called the the pseudo-quotient (the pseudo-remainder) of a by b and denoted by pquo(a, b) *(*prem(a, b)*). The map $(a, b) \longmapsto (q, r)$ is called the pseudo-division of a by b. In addition, Algorithm 3 computes this couple.*

Algorithm 3 Pseudo-division

    Input $a, b \in \mathbb{A}[x]$ with $b \notin \mathbb{A}$.

  Output $q, r \in \mathbb{A}[x]$ satisfying Relation (2.2) with $e = \min(0, \deg(a) - \deg(b) + 1)$.

pseudo-division$(a, b) ==$

  1: $r \leftarrow a; \, q \leftarrow 0$
  2: $e \leftarrow \max(0, \deg(a) - \deg(b) + 1)$
  3: **while** $r \neq 0$ **or** $\deg(r) \geq \deg(b)$ **do**
  4:     $d \leftarrow \deg(r) - \deg(b)$
  5:     $t \leftarrow \mathrm{lc}(r) y^d$
  6:     $q \leftarrow \mathrm{lc}(b) q + t$
  7:     $r \leftarrow \mathrm{lc}(b) r - tb$
  8:     $e \leftarrow e - 1$
  9: **end while**
 10: $r \leftarrow \mathrm{lc}(b)^e r$
 11: $q \leftarrow \mathrm{lc}(b)^e q$
 12: **return** (q, r)

In the above algorithm, for a non-zero polynomial $a \in \mathbb{A}[x]$ we denote by $\deg(a)$ and $\mathrm{lc}(a)$ the degree and the leading coefficient of a.

We return now to the case of multivariate polynomials in $\mathbb{K}[X]$. Let $p, q \in \mathbb{K}[X]$ with q non-constant. The greatest variable of q is denoted by $\mathrm{mvar}(q)$ and is called the *main variable* of q. Regarding q as a univariate polynomial w.r.t. $\mathrm{mvar}(q)$:

- the leading coefficient of q is called the *initial* of q,

- the degree of q (w.r.t. $\mathrm{mvar}(q)$) is denoted by $\mathrm{mdeg}(q)$,

- the monomial $\mathrm{mvar}(q)^{\mathrm{mdeg}(q)}$ is called the *rank* of q and is denoted by $\mathrm{rank}(q)$.

Assume that q is not constant, either. Then, we write $\mathrm{rank}(p) < \mathrm{rank}(q)$ and we say that p has a *smaller rank* than q if

- either $\mathrm{mvar}(p) < \mathrm{mvar}(q)$ holds or,

- $\mathrm{mvar}(p) = \mathrm{mvar}(q)$ and $\mathrm{mdeg}(p) < \mathrm{mdeg}(q)$ hold.

We denote by $\mathrm{prem}(p, q)$ the pseudo-remainder of p by q w.r.t. $\mathrm{mvar}(q)$, that is, if $\mathrm{mvar}(p) = x_\ell$, for some $1 \leq \ell \leq n$, the pseudo-remainder of p by q regarded in $\mathbb{A}[x_\ell]$ for $\mathbb{A} = \mathbb{K}[x_1, \ldots, x_{\ell-1}, x_{\ell+1}, \ldots, x_n]$. We say that p is *reduced* w.r.t q if its degree w.r.t. $\mathrm{mvar}(q)$ is less than $\mathrm{mdeg}(q)$, that is, if $\mathrm{prem}(p, q) = p$. Here's now a central definition in this thesis.

Definition 2.5.2. A subset $B = \{b_1, \ldots, b_k\}$ of non-constant polynomials of $\mathbb{K}[X]$ is a *triangular set* if the main variables of $b_1, \ldots, b_k$ are pairwise different. The triangular set $B = \{b_1, \ldots, b_k\}$ is *autoreduced* if for all $1 \leq i \leq s$ the polynomial b_i is reduced w.r.t. $B \setminus \{b_i\}$.

We denote by $\mathrm{mvar}(B)$ the set of the $\mathrm{mvar}(t)$ and by $\mathrm{rank}(B)$ the set of the $\mathrm{rank}(t)$, for all t in B. A variable from X is said *algebraic* w.r.t. B if it belongs to $\mathrm{mvar}(B)$.

From now on, we assume that the elements of every autoreduced triangular set $B = \{b_1, \ldots, b_k\}$ are sorted by increasing rank and we simply write $B = b_1, \ldots, b_k$.

For a polynomial $p \in \mathbb{K}[X]$ and an autoreduced triangular set $B = b_1, \ldots, b_k$ we define the pseudo-remainder of p w.r.t. B, denoted by $\mathrm{prem}(p, B)$, as follows:

$$\mathrm{prem}(p, B) = \begin{cases} \mathrm{prem}(p, b_1) & \text{if } k = 1 \\ \mathrm{prem}(\mathrm{prem}(p, b_2, \ldots, b_k), b_1) & \text{otherwise} \end{cases}$$

In Algorithm 4, the operation $\mathrm{PseudoReduce}(\{a_1, \ldots, a_t\}, \{b_1, \ldots, b_k\}, \leq)$ returns all the $\mathrm{prem}(a_i, \{b_1, \ldots, b_k\})$ for all $1 \leq i \leq t$, where $\leq$ records the variable ordering. The following proposition, sometimes called the *Remainder Formula* [8], plays for characteristic sets a role similar to Proposition 2.4.1 in the context of Gröbner bases.

Proposition 2.5.3. *Let $p \in \mathbb{K}[X]$ be a polynomial and $B = b_1, \ldots, b_k$ be an autoreduced triangular set. Denote by h the product $\mathrm{init}(b_1) \cdots \mathrm{init}(b_k)$ of the initials of B. Then, there exist polynomials $q_1, \ldots, q_k, r \in \mathbb{K}[X]$ and a non-negative integer e such that*

(i) $h^e p = q_1 b_1 + \cdots + q_k b_k + r$,

(ii) r *is reduced w.r.t. B.*

Recall that if $G \subseteq \mathbb{K}[X]$ is a Gröbner basis of G for some admissible monomial order, then for all $p \in \mathbb{K}[X]$ we have:

$$\mathrm{Reduce}(p, G) = 0 \quad \Longleftrightarrow \quad p \in \langle G \rangle.$$

If $B = b_1, \ldots, b_k$ is an autoreduced triangular set such that $\mathrm{prem}(p, B) = 0$ holds, what can we say about p and B? A first step to the answer goes through the following.

Definition 2.5.4. Let $B = \{b_1, \ldots, b_k\}$ be a (not necessarily autoreduced) triangular set and let h the product $\mathrm{init}(b_1) \cdots \mathrm{init}(b_k)$ of the initials of B. The *saturated ideal*

of B, denoted by $\mathrm{Sat}(B)$ is the set of all polynomials $p \in \mathbb{K}[X]$ such that there exist polynomial $q_1, \ldots, q_k, r \in \mathbb{K}[X]$ and a non-negative integer e satisfying

$$h^e p = q_1 b_1 + \cdots + q_k b_k.$$

Clearly $\mathrm{prem}(p, B) = 0$ implies $p \in \mathrm{Sat}(B)$. However, the converse implication holds if and only if B is a regular chain, as shown in [8]. Before discussing this latter notion in the next section, it is important to describe other properties of triangular sets. First, we give a "geometrical interpretation" of the saturated ideal of a triangular set, after the following definition.

Definition 2.5.5. Let $B \subset \mathbb{K}[X]$ be a triangular set and let h be the product of the initials of B. We call the *quasi-component* of T, denoted by $W(T)$, the subset of the zero-set $V(T)$ consisting of all the points which do not cancel h, that is,

$$W(T) = V(T) \setminus V(\{h\}).$$

Definition 2.5.5 formalizes the way we were reading the solutions of a triangular system in Sections 2.1, 2.2 and 2.3. The following theorem [8] states that the quasi-component of T is almost the zero-set of the saturated ideal of T.

Theorem 2.5.6. *For all triangular sets $B \subset \mathbb{K}[X]$ we have:*

$$\overline{W(B)} = V(\mathrm{Sat}(B)).$$

Remember that $\overline{W(B)}$ denotes the Zariski closure of $W(B)$, that is, the intersection of all algebraic varieties containing $W(B)$. Thus, $V(\mathrm{Sat}(B))$ is the smallest algebraic variety containing $W(B)$. Remember that $V(B)$ can contain components which are removed in $W(B)$. For instance, in Section 2.3 we saw that for $C_1 = x_1 x_4 - x_2, x_2 x_3 - x_1, x_2^3 - x_1^2$ we had $V(C_1) = \overline{W(C_1)} \cup V(x_1, x_2)$.

We now define a partial order on autoreduced triangular sets. This construction is similar to that of Section 2.4 which leads to the definition of a Gröbner basis. However the notions of rank and reduction are different. Let $B = b_1, \ldots, b_k$, $B' = b'_1, \ldots, b'_l$ be sorted autoreduced sets. By definition, we write $B \leq B'$ and we say that B has a *lower rank* than B' if

- either there exists $j \leq \min(k, l)$ s.t. $\mathrm{rank}(b_i) = \mathrm{rank}(b'_i)$ (for all $1 \leq i < j$) and $\mathrm{rank}(b_j) < \mathrm{rank}(b'_j)$ hold,

- or $k \geq l$ and $\mathrm{rank}(b_i) = \mathrm{rank}(b'_i)$ (for all $1 \leq i \leq l$) hold.

If neither $B \leq B'$ nor $B' \leq B$ hold, we say that B and B' have the same rank. The following holds [8]: Every family of autoreduced triangular sets has a minimal element. This leads to the following definition.

Definition 2.5.7. Let $F \subset \mathbb{K}[X]$. A *Ritt characteristic set* of F is a subset C of F such that either $C = \{a\}$ holds for some non-zero $a \in \mathbb{K}$ or C is minimal among all autoreduced triangular sets contained in F.

When F is finite, it is not difficult to design an algorithm computing such a minimal element. Hence, in Algorithm 4, for a finite set F, the operation MinimalAutoreducedSubset$(F, \leq)$ returns a Ritt Characteristic Set of F for the variable ordering $\leq$. When F is not finite, other assumptions on F (such as the ideal generated by F is prime) are needed in order to design a simple procedure. In fact, it is the purpose of triangular decompositions to reduce to cases where computing a Ritt characteristic set can be made "simple". The following theorem gives a first fundamental property of Ritt characteristic sets, see [8] for a proof. Please for polynomial ideals, refer to [34].

Theorem 2.5.8. *Let $F \subset \mathbb{K}[X]$, let $\mathcal{I}$ be the ideal generated by F and let C be an autoreduced triangular set contained in $\mathcal{I}$. Then, the following conditions are equivalent:*

(i) C is a Ritt characteristic set of $\mathcal{I}$.

(ii) For all $f \in \mathcal{I}$, we have $\mathrm{prem}(f, C) = 0$.

This theorem has two important corollaries that we prove for the reader to become more familiar with the notions in this chapter.

Corollary 2.5.9. *Let $F \subset \mathbb{K}[X]$ and let C be a Ritt characteristic set of $\langle F \rangle$. Then $V(F) = \emptyset$ if and only if C contains a non-zero constant.*

Let us prove this first corollary. Assume that C contains a non-zero constant, hence C is of the form $\{a\}$ for some non-zero $a \in \mathbb{K}$. Thus we have $V(C) = \emptyset$. Since $C \subset \langle F \rangle$, we deduce $V(F) \subseteq V(C) = \emptyset$. Assume now that C is an autoreduced triangular set. Assume by contradiction that $V(F) = \emptyset$ holds. Applying the Weak Theorem of Zeros (Theorem 2.3.2) we deduce that $1 \in \langle F \rangle$. Clearly $\mathrm{prem}(1, C) = 1$. If follows with Theorem 2.5.8 that C cannot be a Ritt characteristic set of $\langle F \rangle$, which is a contradiction. This concludes the proof of Corollary 2.5.9.

Corollary 2.5.10. *Let $F \subset \mathbb{K}[X]$, let $\mathcal{I}$ be the ideal generated by F and let C be a Ritt characteristic set of $\mathcal{I}$. Then, we have*

$$\overline{W(C)} \ \subseteq \ V(F) \ \subseteq \ V(C).$$

Let us prove this second corollary. We assume that C is a Ritt characteristic set of $\mathcal{I} = \langle F \rangle$. First, we observe that $C \subset F$ implies $V(F) \subseteq V(C)$. Next, Theorem 2.5.8 implies that for all $f \in \mathcal{I}$, we have $\mathrm{prem}(f, C) = 0$. It follows that $\langle F \rangle$ in contained in $\mathrm{Sat}(C)$. Thus the zero-set of $\mathrm{Sat}(C)$ is contained in that of $\langle F \rangle$. Since Theorem 2.5.6 implies $\overline{W(C)} = V(\mathrm{Sat}(C))$ we deduce $\overline{W(C)} \subseteq V(F)$. This concludes the proof.

This latter result implies that a Ritt characteristic set C of $\langle F \rangle$ provides an "approximation" of the zero-set of F. In fact, when $\langle F \rangle$ is prime the equality $\overline{W(C)} = V(F)$ holds, see [8]. Characteristic sets were introduced by J. F. Ritt [117] for "representing" prime ideals in the sense of the previous equality. For a non-prime ideal input ideal $\langle F \rangle$, the algorithm proposed by Ritt involves multivariate polynomial factorization over algebraic extensions of function fields. These computations are generally regarded as expensive. In [145, 146] W.T. Wu proposed an algorithm for "decomposing" the algebraic variety by means of quasi-components of triangular sets. His method does not require polynomial factorization and, essentially, relies only on pseudo-division. In order to achieve this, Wu's method does not compute Ritt characteristic sets of ideals and relies on the following weaker notion.

Definition 2.5.11. Let $F \subset \mathbb{K}[X]$. A *Wu characteristic set* of the ideal $\langle F \rangle$ is a non-empty subset C of F such that

- either $C = \{a\}$ for some non-zero constant $a \in \mathbb{K}$,

- or C is a Ritt characteristic set of a subset G of $\langle F \rangle$ such that $\langle G \rangle = \langle F \rangle$ holds.

It is easy to check that Corollary 2.5.10 holds also for Wu's characteristic sets. Unfortunately, we shall see that Corollary 2.5.9 does not generalize to Wu's characteristic sets. Let $F \subset \mathbb{K}[X]$. A Ritt characteristic set C of $\langle F \rangle$ is necessarily a Wu characteristic set of $\langle F \rangle$. Indeed, it suffices to choose $G = \langle F \rangle$ in Definition 2.5.11. The converse is false. Consider the autoreduced triangular set $T = f_1, f_2, f_3$ with

$$f_1 = x_2^2 - x_1, f_2 = x_1 x_3^2 - 2x_2 x_3 + 1 \ \ \text{and} \ \ f_3 = (x_3 x_2 - 1)x_4 + x_2^2.$$

Clearly, from Definition 2.5.11, choosing $G = T$, the set T is a Wu characteristic set of $\langle T \rangle$. Assume that T is also a Ritt characteristic set of $\langle T \rangle$. Observe that from

$f_1 = 0$ and $f_2 = 0$ we deduce $(x_3 x_2 - 1)^2 = 0$ and thus $x_3 x_2 - 1 = 0$. Hence, with $f_3 = 0$, we obtain $x_2 = 0$. With $f_1 = 0$, this brings $x_1 = 0$. Finally, with $f_2 = 0$, we have $1 = 0$. Therefore $V(T) = \emptyset$. Applying Corollary 2.5.10 we deduce that T contains a non-zero constant, which is a contradiction. Therefore, T cannot be a Ritt characteristic set of $\langle T \rangle$. This example shows an important drawback of the notion of a Wu characteristic set: A triangular set C can be a Wu characteristic set even when $V(C) = \emptyset$ holds. In other words, Corollary 2.5.9 does not generalize to Wu's characteristic sets.

We conclude this section by presenting a procedure for computing a Wu characteristic set of the ideal generated by some $F \subset \mathbb{K}[X]$. This algorithm was given by W. T. Wu in his paper [146]. Its structure is very similar to that of Algorithm 2: each loop iteration has three steps: select, reduce and update.

Algorithm 4 Computing a Wu Characteristic Set
  Input $F \subset \mathbb{K}[X]$ with a variable ordering $\leq$.

 Output C a Wu characteristic set of F.

WuCharSet$(F) ==$
1:   $B \leftarrow F$; $R \leftarrow F$
2:   **while** $R \neq \emptyset$ **do**
3:     (S) $B \leftarrow$ MinimalAutoreducedSubset$(F, \leq)$
4:     (R) $A \leftarrow F \setminus B$; $R \leftarrow$ PseudoReduce$(A, B, \leq)$
5:     (U) $R \leftarrow R \setminus \{0\}$; $F \leftarrow F \cup R$
6:   **end while**
7:   **return** B

From this procedure, W. T. Wu has derived in [146] an algorithm for decomposing any algebraic variety into quasi-components of triangular sets, leading to the following theorem and definition.

Theorem 2.5.12 (Wu, 1987). *For all $F \subset \mathbb{K}[X]$, there exist finitely many autoreduced triangular sets $T_1, \ldots, T_e \subset \mathbb{K}[X]$ such that we have*

$$V(F) = W(T_1) \ \cup \ \cdots \ \cup \ W(T_e).$$

Definition 2.5.13. Let $F \subset \mathbb{K}[X]$. A finite family $T_1, \ldots, T_e \subset \mathbb{K}[X]$ of triangular sets is a *triangular decomposition of F* (or a *triangular decomposition of $V(F)$*) *in the sense of Lazard* if we have

$$V(F) = W(T_1) \ \cup \ \cdots \ \cup \ W(T_e).$$

A finite family $T_1, \ldots, T_e \subset \mathbb{K}[X]$ of triangular sets is a *triangular decomposition of* $V(F)$ *in the sense of Kalkbrener* if we have

$$V(F) = \overline{W(T_1)} \ \cup \ \cdots \ \cup \ \overline{W(T_e)}.$$

In this thesis, when the sense is not specified, the one of Lazard is assumed. Note that it is stronger than that of Kalkbrener.

2.6 Regular Chains

The notion of a regular chain was introduced independently by Kalkbrener in [75, 76] and by Yang and Zhang in [149]. Moreover, Kalkbrener proposed an algorithm to represent algebraic varieties by means of regular chains.

Regular chains are triangular sets with additional properties. In particular, they do not have the drawback of Wu's characteristic sets described in the previous section. More precisely, if $T \subset \mathbb{K}[X]$ is a regular chain then its quasi-component $W(T)$ is not empty.

The theory of regular chains makes an intensive use of the notion of a zero-divisor modulo an ideal. Hence, we review this notion in the following.

Definition 2.6.1. Let $F \subset \mathbb{K}[X]$ and let $\mathcal{I}$ be the ideal generated by F. We say that a polynomial $p \in \mathbb{K}[X]$ is *regular* modulo $\mathcal{I}$ if p is neither null modulo $\mathcal{I}$, nor a zero-divisor modulo $\mathcal{I}$.

We can give now one of the central definitions for the topics discussed in this thesis.

Definition 2.6.2. Let $T = t_1, \ldots, t_s$ be a triangular set in $\mathbb{K}[X]$ where polynomials are sorted by increasing main variables. The triangular set T is a *regular chain* if for all $i = 2 \cdots s$ the initial of t_i is regular modulo the saturated ideal of $t_1, \ldots t_{i-1}$.

One can easily prove the following proposition which gives two simple criteria for a triangular set to be a regular chain.

Theorem 2.6.3. *Let* $T = t_1, \ldots, t_s$ *be a triangular set in* $\mathbb{K}[X]$. *Then we have:*

- *If* $s = 1$ *then* T *is a regular chain,*

- *If the initials of* $t_1, \ldots, t_s$ *are all constant, then* T *is a regular chain.*

Example 2.6.4. Consider $n = 2$ and the triangular set $T = \{f_1, f_2\}$ where

$$f_1 = x_1^2 - x_1 \quad \text{and} \quad f_2 = x_1 x_2^2 + 2x_2 + 1.$$

The triangular set $T_1 = \{f_1\}$ is a regular chain by virtues of Proposition 2.6.3. Moreover $\text{Sat}(T_1)$ is simply the ideal generated by f_1. This fact can be easily proved from the definition of a saturated ideal. However, the triangular set $T = \{f_1, f_2\}$ is not a regular chain: the initial of f_2, namely x_1, is a zero-divisor modulo $\text{Sat}(T_1)$. Indeed, we clearly have $x_1(x_1 - 1) \equiv 0 \mod x_1^2 - x_1$. Observe that this "irregularity" can be fixed by splitting $V(T)$:

$$V(T) = W(T) \ \cup \ W(x_1, 2x_2 + 1) \quad \text{and} \quad W(T) = V(\{x_1 - 1, x_2 + 1\}).$$

In other words, one zero of T is in $W(T)$ (the one which does not cancel the initial of f_2) and the other is not (since it cancels the initial of f_2).

Proposition 2.6.3 implies that $\{x_1, 2x_2 + 1\}$ is also a regular chain. (Indeed, the initial of $2x_2 + 1$ is just a number, namely 2.) Therefore, we have "replaced" the triangular set T, which is not a regular chain, by two regular chains. More generally, any triangular set can be "replaced" by a family of regular chains in a sense that we shall specify later.

Example 2.6.5. Consider $n = 2$ and the triangular set $T = \{f_1, f_2, f_3\}$ where

$$f_1 = x_2^2 - x_1, f_2 = x_1 x_3^2 - 2x_2 x_3 + 1 \quad \text{and} \quad f_3 = (x_3 x_2 - 1)x_4 + x_2^2.$$

The set $T_1 = \{f_1\}$ is a regular chain as is any polynomial set consisting of a single element. Moreover $\text{Sat}(T_1)$ is simply the ideal generated by f_1. This fact can be verified by means of Gröbner bases computations, see [9] for details.

The set $T_2 = \{f_1, f_2\}$ is a regular chain since the initial of f_2, namely x_1, is regular modulo $\text{Sat}(T_1)$. This fact is quite easy to understand: if x_1 were a zero-divisor modulo $\text{Sat}(T_1)$ there would exist a polynomial p such that

$$px_1 \equiv 0 \mod x_2^2 - x_1 \quad \text{and} \quad p \not\equiv 0 \mod x_2^2 - x_1.$$

Degree considerations show that these conditions are contradictory. Therefore, x_1 is regular modulo $\text{Sat}(T_1)$ and T_2 is a regular chain. In addition, one can verify, with Gröbner bases computations, that $\text{Sat}(T_2)$ is simply the ideal generated by T_2.

The set $T_3 = \{f_1, f_2, f_3\}$ is not a regular chain since the initial of f_3, namely

$x_3 x_2 - 1$, is not regular modulo $\mathrm{Sat}(T_2)$. Indeed, the polynomial $(x_3 x_2 - 1)^2$ belongs to $\mathrm{Sat}(T_2)$, i.e. it can be obtained by "combining" f_1 and f_2. Hence, the square of the initial of f_3 is null modulo $\mathrm{Sat}(T_2)$ and T_3 is not a regular chain.

Can we replace T by regular chains, as we did in the previous example? Yes, we can, just with the empty list of regular chains! Indeed, if a point (x_1, x_2, x_3, x_4) belongs to $V(T)$ it must satisfy $(x_3 x_2 - 1)^2 = 0$, and thus $x_3 x_2 - 1 = 0$. Together with $f_3 = 0$ this brings $x_2 = 0$, contradicting $x_3 x_2 - 1 = 0$. Therefore $V(T)$ is just empty! Remember that we saw that the same T was a Wu characteristic set, but not a Ritt characteristic set.

From Definition 2.6.3, it seems that checking whether a given triangular set is a regular chain requires the computation of saturated ideals. As mentioned earlier, systems of generators of saturated ideals can be computed by means of Gröbner bases. However, these computations can be fairly expensive.

In fact, it turns out that these computations can be completely avoided. In broad terms, the trick relies on two observations that we shall state more formally later:

- For a regular chain T with as many variables as equations, testing whether a polynomial p is regular w.r.t. T reduces to polynomial GCD computations (Proposition 2.6.6).

- The case of regular chain with less equations than variables reduces to the previous one. (Theorem 2.6.7).

First, consider the case of two variables x_1 and x_2. Let $T = \{f_1, f_2\}$ be a triangular set in $\mathbb{K}[x_1, x_2]$. Let h be the initial of f_2. We assume that h is not constant, otherwise we can conclude with Proposition 2.6.3. For simplicity, we assume also that T is autoreduced, which implies that the degree of h is less than that of f_1. To decide whether T is a regular chain it suffices to check whether the initial h of f_2 is regular modulo $\mathrm{Sat}(T_1) = \langle f_1 \rangle$. Let g be the GCD of h and f_1 and let u, v be their Bézout coefficients. Hence we have

$$uh + vf_1 = g.$$

If g is a constant, we assume that $g = 1$. In this case we have $uh \equiv 1 \mod f_1$. This shows that u is the inverse of h modulo $\langle f_1 \rangle$ and thus h is not a zero-divisor in this case. Assume now that g has positive degree. Since g divides both f_1 and h we have:

$$h \frac{f_1}{g} = \frac{h}{g} f_1 \equiv 0 \mod f_1.$$

Since $\deg(g) \leq \deg(h)$ and $\deg(h) < \deg(f_1)$ it follows that

$$\frac{f_1}{g} \not\equiv 0 \mod f_1.$$

Therefore, in this case, the polynomial h is a zero-divisor modulo f_1. Observe that in this case, we can factor f_1 into

$$f_1 = g\frac{f_1}{g}$$

To summarize, on this particular situation, we have observed that one can decide whether T is a regular chain simply by means of polynomial GCD computations. More generally, we have the following statement.

Theorem 2.6.6. *Let* $T \subset \mathbb{K}[x_1, \ldots, x_n]$ *be a regular chain with* n *elements* $T_1(x_1), T_2(x_1, x_2), \ldots, T_n(x_1, \ldots, x_n)$. *Let* $p \in \mathbb{K}[x_1, \ldots, x_n]$. *Then, we have:*

   (i) *The saturated ideal of* T *is equal to the ideal generated by* T.

   (ii) *The polynomial* p *is regular w.r.t.* $\langle T \rangle$ *if and only if there exists* $\overline{p} \in \mathbb{K}[x_1, \ldots, x_n]$ *such that* $p\overline{p} \equiv 1 \mod \langle T \rangle$

The second point states that p is regular modulo $\langle T \rangle$ if and only if p is invertible modulo $\langle T \rangle$. Checking invertibility modulo $\langle T \rangle$ can be achieved via polynomial GCD computations, as shown above in the case $n = 2$. Chapter 3 will cover the general case, using fast polynomial arithmetic.

For the question of testing regularity of an element w.r.t. to the saturated ideal of a regular chain, the following result [22] reduces the case of regular chains with less equations than variables to the case of as many equations as variables.

Theorem 2.6.7. *Let* d *be an integer such that* $1 \leq d < n$. *Recall that* $\mathbb{K}(x_1, \ldots, x_d)$ *denotes the field of rational functions with coefficients in* $\mathbb{K}$ *and variables in* $x_1, \ldots, x_d$. *Let* $T = \{T_{d+1}, \ldots, T_n\}$ *be a triangular set of* $\mathbb{K}[X]$. *Assume that* $\mathrm{mvar}(T_i) = x_i$ *for all* $d + 1 \leq i \leq n$ *and assume* $\mathrm{Sat}(T)$ *is a proper ideal of* $\mathbb{K}[X]$.

Let T_0 *be the image of* T *regarded as a triangular set in* $\mathbb{K}(x_1, \ldots, x_d)[x_{d+1}, \ldots, x_n]$, *that is, as a set of polynomials with variables in* $x_{d+1}, \ldots, x_n$ *and with coefficients in* $\mathbb{K}(x_1, \ldots, x_d)$. *(Formally speaking consider the localization by* $\mathbb{K}[x_1, \ldots, x_d] \setminus \{0\}$.)

Let $p \in \mathbb{K}[x_1, \ldots, x_n]$ *and* p_0 *its image in* $\mathbb{K}(x_1, \ldots, x_d)[x_{d+1}, \ldots, x_n]$. *Assume* p *non-zero modulo* $\mathrm{Sat}(T)$. *Then, the following conditions are equivalent:*

   (1) p *is regular w.r.t.* $\mathrm{Sat}(T)$,

(2) p_0 *is invertible w.r.t.* $\text{Sat}(T_0)$.

In particular T is a regular chain iff T_0 is a regular chain.

We conclude this section by a series of Theorems that play a central role in the algorithms for solving polynomial systems by means of regular chains.

Theorem 2.6.8 (Aubry, Lazard & Moreno Maza, 1997)**.** *Let T be an autoreduced triangular set in $\mathbb{K}[X]$. The following properties are equivalent.*

(*i*) $\text{Sat}(T)$ *is exactly the set of the polynomials p that reduces to 0 by pseudo-division w.r.t. T,*

(*ii*) T *is a regular chain,*

(*iii*) T *is a Ritt characteristic set of* $\text{Sat}(T)$.

This fundamental theorem provides an ideal membership test for saturated ideals of regular chains: if T is a regular chain and $p \in \mathbb{K}[X]$ we have

$$p \in \text{Sat}(T) \quad \Longleftrightarrow \quad \text{prem}(p, T) = 0.$$

Theorem 2.6.9. *If T is a regular chain, then every prime ideal associated with $\text{Sat}(T)$ has dimension $n - \mid T \mid$ where $\mid T \mid$ denotes the number of elements in T. Hence, the ideal $\text{Sat}(T)$ and the variety $\overline{W(T)}$ have dimension $n - \mid T \mid$. In particular $\text{Sat}(T)$ is a proper ideal of $\mathbb{K}[X]$ and $W(T) \neq \emptyset$.*

Let T be a regular chain and p be a polynomial. We denote by $Z(p, T)$ the intersection $V(p) \cap W(T)$, that is the set of the zeros of p that are contained in the quasi-component $W(T)$.

Theorem 2.6.10. *If p is regular modulo the saturated ideal of T, then $Z(p, T)$ is either empty or it is contained in a variety of dimension strictly less than the dimension of $\overline{W(T)}$.*

2.7   The Triade Algorithm

We discuss in this section the main features of the Triade algorithm [108] with an emphasis on those that are relevant to parallel execution. The notions of a *Task*, Definition 2.7.1 and that of a *delayed split*, Definition 2.7.3 play a central role. They are well-adapted to describe the relations between the intermediate computations

during the solving of a polynomial system. Algorithm 5 is the top-level procedure of the Triade algorithm: it manages a *task pool*. The tasks are transformed by means of a sub-procedure (Algorithms 6 and 7) which is dedicated to "simple tasks".

Definition 2.7.1. We call a *task* any couple $[F, T]$ where F is a finite subset of $\mathbb{K}[X]$ and $T \subset \mathbb{K}[X]$ is a regular chain. The task $[F, T]$ is *solved* if F is empty, otherwise it is *unsolved*. By *solving* a task, we mean computing regular chains $T_1, \ldots, T_e$ such that we have:

$$V(F) \cap W(T) \subseteq \cup_{i=1}^{e} W(T_i) \subseteq V(F) \cap \overline{W(T)}. \tag{2.3}$$

Most algorithms computing triangular decompositions consist of procedures that take a task $[F_0, T_0]$ as input and returns zero, or more tasks $[F_1, T_1], \ldots, [F_e, T_e]$. Then, solving an input polynomial system F_0 is achieved by calling one of these procedures with $[F_0, \emptyset]$ as input and obtaining "solved tasks" $[\emptyset, T_1], \ldots, [\emptyset, T_e]$ as output, such that $T_1, \ldots, T_e$ solves $[F_0, \emptyset]$ in the sense of Definition 2.7.1.

Therefore, given an algorithm A for computing triangular decompositions, it is natural to associate with each input polynomial system F_0 a *task tree* $G_A(F_0)$ whose vertices are tasks such that there is an arrow from any task $[F_i, T_i]$ to any task $[F_j, T_j]$ if task $[F_j, T_j]$ is among the output tasks of a procedure called on $[F_i, T_i]$; moreover, each internal node $[F_i, T_i]$ has a weight equal to the (estimated) running time for computing the children of $[F_i, T_i]$. The longest path (summing the weights along the path) from the root to a leaf, what is called a *critical path of* $G_A(F_0)$ and often denoted by T_∞, represents the minimum running time for a parallel execution of $A(F_0)$ on infinitely many processors. (Here we do not consider communication costs and scheduling overheads, for simplicity.) The sum of all the weights in $G_A(F_0)$, called the *work of* $G_A(F_0)$ and often denoted by T_1, represents the minimum running time for a sequential execution of $A(F_0)$.

It is well known that most algorithms decomposing polynomial systems into components (irreducible, equidimensional, ...) have to face the problem of *redundant components*, which may occur in the output or at intermediate stages of the solving process. This is a central question when computing triangular decompositions, see [9] for a discussion of this topic. Removing redundant components is also an important issue in other symbolic decomposition algorithms such as the one of [88] and also for numerical ones [126]. Being able to remove redundant components at an early stage of the computations helps reducing the work of $G_A(F_0)$ and, possibly its critical

path. One of the motivations in the design of the Triade algorithm [108] is to handle redundant components efficiently.

For any input task $[F, T]$ the main procedure of the Triade algorithm, called Triangularize(F, T), solves $[F, T]$ in the sense of Definition 2.7.1. This procedure reduces to the situation where F consists of a single polynomial p. One could expect that such an operation, say Decompose(p, T), should return regular chains $T_1, \ldots, T_\ell$ solving the task $[\{p\}, T]$. In fact, we shall explain now why this would not meet our requirement of handling redundant components efficiently.

Observe that $W(T_1) \subseteq W(T_2)$ implies $|T_2| \leq |T_1|$. It follows that, during the solving process, all the (final or intermediate) regular chains should be generated by increasing order of their cardinality, that is, by decreasing order of the dimension of their saturated ideals, in order to remove the redundant ones as soon as possible. Returning to the specifications of the operation Decompose(p, T), observe that $V(p) \cap W(T)$ could contain components of different dimension. (This will happen when $V(p)$ contains some of the irreducible components of $\overline{W(T)}$, but not all of them.) Therefore, it is not desirable for the operation Decompose(p, T) to solve the task $[\{p\}, T]$ in one step. Instead, Decompose(p, T) should compute the quasi-components of $V(p) \cap W(T)$ of maximum dimension and postpone the computation of the other quasi-components. This is made possible by a form of lazy evaluation, formalized by Definition 2.7.3, after Notation 2.7.2. Lazy evaluation is a common technique in functional programming language such as Lisp [151].

Notation 2.7.2. Let T_1, T_2 be two regular chains. We write $\mathrm{rank}(T_1) \prec \mathrm{rank}(T_2)$ whenever $\mathrm{rank}(T_2)$ is a proper subset of $\mathrm{rank}(T_1)$, or when $v_1^{d_1} \prec v_2^{d_2}$, where $v_1^{d_1}$ (resp. $v_2^{d_2}$) is the smallest element of $\mathrm{rank}(T_1) \setminus \mathrm{rank}(T_2)$ (resp. $\mathrm{rank}(T_2) \setminus \mathrm{rank}(T_2)$). When neither $\mathrm{rank}(T_1) \prec \mathrm{rank}(T_2)$ nor $\mathrm{rank}(T_2) \prec \mathrm{rank}(T_1)$ hold, we write $\mathrm{rank}(T_1) \simeq \mathrm{rank}(T_2)$. Let F_1, F_2 be finite subsets of $\mathbb{K}[X]$. We write $[F_1, T_1] \prec [F_2, T_2]$ either if $\mathrm{rank}(T_1) \prec \mathrm{rank}(T_2)$ holds, or if $\mathrm{rank}(T_1) \simeq \mathrm{rank}(T_2)$ holds and there exists $f_1 \in F_1$ such that $\mathrm{rank}(f_1) \prec \mathrm{rank}(f_2)$ for all $f_2 \in F_2$. Clearly, any sequence of tasks $[F_0, T_0], \ldots$, such that $[F_i, T_i] \prec [F_{i+1}, T_{i+1}]$ holds for all i, is finite.

Definition 2.7.3. The tasks $[F_1, T_1], \ldots, [F_e, T_e]$ form a *delayed split* of the task $[F, T]$ and we write

$$[F, T] \longmapsto_D [F_1, T_1], \ldots, [F_e, T_e]$$

if for all $1 \leq i \leq e$ we have $[F_i, T_i] \prec [F, T]$ and the following holds

$$V(F) \cap W(T) \subseteq \cup_{i=1}^e Z(F_i, T_i) \subseteq V(F) \cap \overline{W(T)}.$$

Below, we highlight the main features of the procedure $\mathsf{Decompose}(p, T)$ that are relevant to the rest of the work. First, for a polynomial p and a regular chain T, such that p is not zero modulo $\mathrm{Sat}(T)$, the procedure $\mathsf{Decompose}(p, T)$ returns a delayed split of the task $[\{p\}, T]$. Algorithm 5 implements the procedure $\mathsf{Triangularize}$ by means of the procedure $\mathsf{Decompose}$. Based on the specifications of $\mathsf{Decompose}$, the validity of this algorithm is easy to check and is established in [108]. Note that in our pseudo-code, we use indentation to denote blocks. Moreover, each of the Algorithms 5, 6 and 7. generates a sequence of items which are returned one by one in the output flow by means of **yield** statements.

Algorithm 5 Triangularize

    Input a task $[F, T]$.

  Output regular chains $T_1, \ldots, T_e$ solving $[F, T]$ in the sense of Definition 2.7.1

$\mathsf{Triangularize}(F, T) ==$ **generate**
  1: $task\_list \leftarrow [[F, T]]$
  2: **while** $task\_list \neq [\,]$ **do**
  3:      Choose and remove a task $[F_1, U_1]$ from $task\_list$
  4:      $F_1 = \emptyset \implies$ **yield** U_1
  5:      Choose a polynomial $p \in F_1$
  6:      $G_1 \leftarrow F_1 \setminus \{p\}$
  7:      **if** $p \equiv 0 \mod \mathrm{Sat}(U_1)$ **then**
  8:         $R \leftarrow \mathsf{cons}\ ([G_1, U_1], task\_list)$
  9:      **end if**
10:      **for** $[H, T] \in \mathsf{Decompose}(p, U_1)$ **do**
11:         $task\_list \leftarrow \mathsf{cons}\ ([G_1 \cup H, T], task\_list)$
12:      **end for**
13: **end while**

The key notion used by the procedure $\mathsf{Decompose}$ is that of a polynomial GCD modulo a regular chain, see Definition 2.7.4. This notion strengthens that introduced by Kalkbrener in [76] and extends that of a polynomial GCD modulo a triangular set introduced in [109].

Definition 2.7.4. Let p, t, g be non-zero polynomials and T be a regular chain. Assume that p and t are non-constant and have the same main variable v. Assume that $v \notin \mathrm{mvar}(T)$, that $\mathrm{init}(p)$ is regular w.r.t. $\mathrm{Sat}(T)$ and that $T \cup \{t\}$ is a regular chain. Then, the polynomial g is a *GCD of p and t w.r.t. T* if the following hold:

(G_1) g belongs to the ideal generated by p, t and $\mathrm{Sat}(T)$,

(G_2) the leading coefficient h_g of g w.r.t. v is regular modulo $\mathrm{Sat}(T)$,

(G_3) if mvar$(g) = v$ then p and t belong to Sat$(T \cup \{g\})$.

More generally, a sequence of pairs $G = (g_1, T_1), \ldots, (g_e, T_e)$, where $g_1, \ldots, g_e$ are polynomials and $T_1, \ldots, T_e$ are regular chains, is a *GCD sequence of p and t w.r.t. T* if the following properties hold:

(G_4) for all $1 \leq i \leq e$, if $|T_i| = |T|$ then g_i is a GCD of p and t modulo T_i,

(G_5) we have $W(T) \subseteq \cup_{i=1}^e W(T_i) \subseteq \overline{W(T)}$.

in Notation 2.7.5.

Notation 2.7.5. Four procedures of Triade are essential in Chapter 5.

- Let p, t, T be as in Definition 2.7.4. The procedure $\mathsf{GCD}(p, t, T)$ computes a GCD sequence of p and t *w.r.t. T*. These GCD computations allow testing whether a polynomial f is regular modular Sat(T).

- Given a polynomial f, the procedure $\mathsf{Regularize}(f, T)$ returns regular chains $T_1, \ldots, T_e$ such that for all $1 \leq i \leq e$, the polynomial f is either null or regular modulo Sat(T_i) and Relation (G_5) holds.

- Given a polynomial f, the procedure $\mathsf{RegularizeInitial}(f, T)$ returns regular chains $T_1, \ldots, T_e$ such that for all $1 \leq i \leq e$, the polynomial f is congruent to constant modulo Sat(T_i), or congruent to a non-constant polynomial f_i modulo Sat(T_i), whose initial init(f_i) is regular modulo Sat(T_i); moreover, Relation (G_5) holds.

- Given a triangular set $S \subset \mathbb{K}[X]$ the procedure $\mathsf{Extend}(S)$ returns regular chains $T_1, \ldots, T_e$ satisfying $W(S) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(S)}$.

- Finally, we denote by $\mathsf{Reduce}(f, T)$ the polynomial defined as follows: if $f \in \mathbb{K}$, then $\mathsf{Reduce}(f, T)$ is f; if $f \in \mathrm{Sat}(T)$, then $\mathsf{Reduce}(f, T)$ is 0 otherwise it is $\mathsf{Reduce}(h, T) \, \mathrm{mvar}(f) + \mathsf{Reduce}(g)$, where $h = \mathrm{init}(f)$ and $g = f - \mathrm{init}(f)\mathrm{rank}(f)$.

Algorithm 7 states the algorithm for $\mathsf{Decompose}(p, T)$. It relies on a sub-procedure given by Algorithm 6. The proof of Algorithms 6 and 7 relies fundamentally on Proposition 2.7.6. In broad words, this result states that the common zeros of p and t contained in $W(T)$ are "essentially" given by $W(T \cup \{g\})$, where g is a GCD of p and t w.r.t. T in the sense of Definition 2.7.4. See [108] for detail.

Algorithm 6 Algebraic Decompose

    Input p, T, t as in Definition 2.7.4.

  Output a delayed split of $[\{p\}, T \cup \{t\}]$.

AlgebraicDecompose$(p, T, t) ==$ **generate**

  1: $h_T \leftarrow$ product of the initials in T
  2: **for** $[g_i, T_i] \in \mathsf{GCD}(t, p, T)$ **do**
  3:     **if** $|T_i| > |T|$ **then**
  4:         **for** $T_{i,j} \in \mathsf{Extend}(T_i \cup \{h_T t\})$ **do**
  5:             **yield** $[p, T_{i,j}]$
  6:         **end for**
  7:     **end if**
  8:     **if** $g_i \in \mathbb{K}$ **then iterate**
  9:     **if** $\mathrm{mvar}(g_i) < v$ **then yield** $[\{g_i, p\}, T_i \cup \{t\}]$
 10:     **if** $\deg(g_i, v) = \deg(t, v)$ **then yield** $[\emptyset, T_i \cup \{t\}]$
 11:     **yield** $[\emptyset, T_i \cup \{g_i\}]$
 12:     **yield** $[\{\mathrm{init}(g_i), p\}, T_i \cup \{t\}]$
 13: **end for**

Algorithm 7 Decompose

    Input a polynomial p and a regular chain T such that $p \notin \mathrm{Sat}(T)$.

  Output a delayed split of $[\{p\}, T]$.

Decompose$(p, T) ==$ **generate**

  1: **for** $C \in \mathsf{RegularizeInitial}(p, T)$ **do**
  2:     $f := \mathsf{Reduce}(p, C)$
  3:     **if** $f = 0$ **then yield** $[\emptyset, C]$
  4:     **if** $f \in \mathbb{K}$ **then iterate**
  5:     $v := \mathrm{mvar}(f)$
  6:     **if** $v \notin \mathrm{mvar}(C)$ **then**
  7:         **yield** $[\{\mathrm{init}(f), p\}, C]$
  8:         **for** $D \in \mathsf{Extend}(C \cup \{f\})$ **do**
  9:             **yield** $[\emptyset, D]$
 10:         **end for**
 11:     **end if**
 12:     **for** $[F, E] \in \mathsf{AlgebraicDecompose}(f, C_{<v} \cup C_{>v}, C_v)$ **do**
 13:         **yield** $[F, E]$
 14:     **end for**
 15: **end for**

Proposition 2.7.6. *Let p, t, g, T be as in Definition 2.7.4. If g is a GCD of p and t w.r.t. T and $\mathrm{mvar}(g) = v$ holds, then we have*

$$[[\{p\}, T \cup \{t\}] \longmapsto_D [\emptyset, T \cup \{g\}], [\{h_g, p\}, T \cup \{t\}].$$

The following fundamental proposition is easily checked from the pseudo-code of Algorithm 6 and Algorithm 7.

Proposition 2.7.7. *Let $[F, E]$ be any task returned by Algorithm 7. Then we have:*

(H_1) *either $|E| = |T|$ and $F = \emptyset$,*

(H_2) *or $|E| = |T|$ and F contains a polynomial which is regular w.r.t. $\mathrm{Sat}(T)$,*

(H_3) *or $|E| > |T|$.*

Corollary 2.7.8. *Let $[F, E]$ be a task returned by Algorithm 7. If $\overline{W(E)}$ is a component of $V(p) \cap \overline{W(T)}$ with maximum dimension, then the task $[F, E]$ is solved, that is, $F = \emptyset$.*

Solving by decreasing order of dimension. It follows from Corollary 2.7.8 that the tasks in Algorithm 5 can be chosen such that the regular chains output by this algorithm are generated by increasing size. To do so, we assign to each task $[F, T] \in R$ in Algorithm 5 an upper bound $m([F, T])$ for the height of the regular chains solving $[F, T]$ in the sense of Definition 2.7.1. This upper bound is simply computed as follows. If a polynomial $f \in F$ has been shown to be regular w.r.t. T (See Proposition 2.7.7) then $m([F, T]) := |T| + 1$ otherwise $m([F, T]) := |T|$. Then, we say that a task $[F_1, T_1]$ has a *higher priority* than a task $[F_2, T_2]$ if either $m([F_1, T_1]) \leq m([F_2, T_2])$ holds, or $m([F_1, T_1]) = m([F_2, T_2])$ and $[F_1, T_1] \prec [F_2, T_2]$ hold. Sorting the tasks in the list R w.r.t. this ordering allows us to solve by decreasing order of dimension and therefore to handle redundant components efficiently. The performances of our inclusion test are reported in [31].

Chapter 3

Fast Polynomial Arithmetic over Direct Products of Fields

3.1   Introduction

The **D5 Principle** was introduced in 1985 by Jean Della Dora, Claire Dicrescenzo and Dominique Duval in their celebrated note "About a new method for computing in algebraic number fields". This innovative approach automatizes reasoning based on case discussion and is also known as "Dynamic Evaluation". Let us review this principle.

Let $\mathbb{K}$ be a field and let $\mathbb{L} = \mathbb{K}[X]/\langle p \rangle$ be an extension of $\mathbb{K}$, where p is a non-constant, univariate square-free, but not necessarily irreducible polynomial. Thus $\mathbb{L}$ is a direct product of fields. We want to decide whether some polynomial $q \in \mathbb{K}[X]$ is invertible in $\mathbb{L}$. Let g be the GCD of p and q. Then, we have:

- q is zero modulo g

- q is invertible modulo p/g.

Indeed, since p is square-free, the polynomials q and p/g are relatively prime. This is called a quasi-inverse computation. Up to splitting, it allows one to compute in $\mathbb{L}$ as if it were a field. Applications of Dynamic Evaluation have been made by many authors: [66], [65], [49], [100] and others. Many algorithms for polynomial system solving rely on the D5 principle; see, for instance, the work of [87], [76], [44], [108], [106], and [22].

When computing with polynomials over a direct product of fields $\mathbb{L}$, one may encounter a zero-divisor z. It is then natural to decompose $\mathbb{L}$ into two or more

components such that over each of these components z becomes zero or a unit. Then, depending on the context, one may wish to continue the computations over some or all components. In the works [21, 116, 77] computations continue only in one branch because the other branches are not interesting for the problem under study. In these other works [76, 86, 87, 48, 49, 65, 108] computations continue in all branches since one is interested in obtaining a result over $\mathbb{L}$.

The results reported in this chapter aims at filling the lack of complexity results for this approach. The addition and multiplication over a direct product of fields are easily proved to be *quasi-linear* (in a natural complexity measure). As for the inversion, it has to be replaced by *quasi-inversion* as in the above example: following the D5 philosophy, meeting zero-divisors in the computation will lead to *splitting* the direct product of fields into a family thereof. It is much more tricky to prove quasi-linear complexity estimate for quasi-inversion, because the algorithm relies on other algorithms, for which such an estimate has to be proved: the GCD and the splitting algorithms. In fact, this can be achieved by an inductive process that we will sketch at the end of this section. But, before that some setting is needed.

In this work, direct product of fields will be described using *triangular sets*. Those triangular sets will have additional properties in Definition 3.1.1 w.r.t. the general ones of Definition 2.5.2. In fact, the term *Lazard triangular set* should be used here, after the work of Lazard [87]. However, in this Chapter and in the next one, we will just say *triangular set* for short.

In what follows, we assume that the base field k is perfect. In practice k is either the field $\mathbb{Q}$ of rational numbers or a prime field $\mathbb{Z}/p\mathbb{Z}$, for a prime integer p. We reserve the notation $\mathbb{K}$ for extensions of the base field k.

Definition 3.1.1. A *triangular set* T is a family of n-variate polynomials over k:

$$T \;\; = (T_1(X_1) \;, \;\; T_2(X_1, X_2) \;, \;\; \dots \;, \;\; T_n(X_1, \dots, X_n)) \,,$$

which forms a reduced Gröbner basis for the lexicographic order induced by $X_n > \cdots > X_1$, and such that the ideal $\langle T \rangle$ generated by T in $k[X_1, \dots, X_n]$ is radical.

If T is a triangular set, the residue class ring $\mathbb{K}(T) := k[X_1, \dots, X_n]/\langle T \rangle$ is a direct product of fields. Hence, our questions can be basically rephrased as studying the complexity of operations (addition, multiplication, quasi-inversion) modulo triangular sets. The following notation helps us quantify the complexity of these algorithms.

Definition 3.1.2. We denote by $\deg_i(T)$ the degree of T_i in X_i, for all $1 \leq i \leq n$, and by $\deg(T)$ the product $\deg_1(T) \cdots \deg_n(T)$. We call it the *degree* of T.

Observe that $\langle T \rangle$ is zero-dimensional and that for all $1 \leq i \leq n$, the set $(T_1 \ldots, T_i)$ is a triangular set of $k[X_1, \ldots, X_i]$. The zero-set of T in the affine space $\mathbb{A}^n(\bar{k})$ has a particular feature: it is *equiprojectable* [10, 42]; besides, its cardinality equals $\deg(T)$. The notion of equiprojectability is discussed in Section 4.2.

Definition 3.1.3. A *triangular decomposition* of a zero-dimensional radical ideal $I \subset k[X_1, \ldots, X_n]$ is a family $\mathbf{T} = T^1, \ldots, T^e$ of triangular sets, such that $I = \langle T^1 \rangle \cap \cdots \cap \langle T^e \rangle$ and $\langle T^i \rangle + \langle T^j \rangle = \langle 1 \rangle$ for all $i \neq j$. A triangular decomposition $\mathbf{T}'$ of I *refines* another decomposition $\mathbf{T}$ if for every $T \in \mathbf{T}$ there exists a (necessarily unique) subset $\mathrm{decomp}(T, \mathbf{T}') \subseteq \mathbf{T}'$ which is a triangular decomposition of $\langle T \rangle$.

Let T be a triangular set, let $\mathbf{T} = T^1, \ldots, T^e$ be a triangular decomposition of $\langle T \rangle$, and define $\mathbb{K}(\mathbf{T}) := \mathbb{K}(T^1) \times \cdots \times \mathbb{K}(T^e)$. Then by the Chinese remainder theorem:

$$\mathbb{K}(T) \simeq \mathbb{K}(\mathbf{T}). \tag{3.1}$$

Now let $\mathbf{T}'$ be a refinement of $\mathbf{T}$. For each triangular set T^i in $\mathbf{T}$, denote by $U^{i,1}, \ldots, U^{i,e_i}$ the triangular sets in $\mathrm{decomp}(T^i, \mathbf{T}')$. We have the following e isomorphism:

$$\phi_i : \ \mathbb{K}(T^i) \simeq \mathbb{K}(U^{i,1}) \times \cdots \times \mathbb{K}(U^{i,e_i}), \tag{3.2}$$

which extend to the following e isomorphisms, where y is a new variable.

$$\Phi_i : \ \mathbb{K}(T^i)[y] \simeq \mathbb{K}(U^{i,1})[y] \times \cdots \times \mathbb{K}(U^{i,e_i})[y]. \tag{3.3}$$

Definition 3.1.4. For $\mathbf{h} = (h_1, \ldots, h_e) \in \mathbb{K}(T^1)[y] \times \cdots \times \mathbb{K}(T^e)[y]$, we call *projection* of $\mathbf{h}$ w.r.t. $\mathbf{T}$ and $\mathbf{T}'$, and write $\mathrm{project}(\mathbf{h}, \mathbf{T}, \mathbf{T}')$ the vector $(\Phi_1(h_1), \ldots, \Phi_e(h_e))$.

Note that if $g \in \mathbb{K}(T)[y]$, then we have

$$\mathrm{project}(g, \{T\}, \mathbf{T}') = \mathrm{project}(\mathrm{project}(g, \{T\}, \mathbf{T}), \mathbf{T}'). \tag{3.4}$$

For simplicity, we define:

$$\mathrm{project}(g, \mathbf{T}) = \mathrm{project}(g, \{T\}, \mathbf{T}). \tag{3.5}$$

We now introduce a fundamental notion, that of a *non-critical* decompositions. It is motivated by the following remark. Let $\mathbf{T} = T^1, \ldots, T^e$ be a family of triangular sets, with $T^j = (T_1^j, T_2^j, \ldots, T_n^j)$. For $1 \leq i \leq n$, we write $T_{\leq i}^j = T_1^j, T_2^j, \ldots, T_i^j$ and

define the family $\mathbf{T}_{\leq i}$ by:

$$\mathbf{T}_{\leq i} = \{T^j_{\leq i} \mid j \leq e\} \quad \text{(with no repetition allowed)}.$$

Even if $\mathbf{T}$ is a triangular decomposition of a 0-dimensional radical ideal $I \subset k[X_1, \ldots, X_n]$, the family $\mathbf{T}_{\leq i}$ is not necessarily a triangular decomposition of $I \cap k[X_1, \ldots, X_i]$. Indeed, with $n = 2$ and $e = 2$, consider $T^1 = ((X_1-1)(X_1-2), X_2)$ and $T^2 = ((X_1-1)(X_1-3), X_2-1)$. The family $\mathbf{T} = T^1, T^2$ is a triangular decomposition of the ideal $I = \langle T^1 \rangle \cap \langle T^2 \rangle$. However, the family of triangular sets

$$\mathbf{T}_{\leq 1} = \{T^1_1 = (X_1-1)(X_1-2), \ T^1_2 = (X_1-1)(X_1-3)\}$$

is not a triangular decomposition of $I \cap k[X_1]$ since $\langle T^1_1 \rangle + \langle T^2_1 \rangle = \langle X_1 - 1 \rangle$.

Definition 3.1.5. Let T be a triangular set in $k[X_1, \ldots, X_n]$. Two polynomials $a, b \in \mathbb{K}(T)[y]$ are *coprime* if the ideal $\langle a, b \rangle \subset \mathbb{K}(T)[y]$ equals $\langle 1 \rangle$.

Definition 3.1.6. Let $T \neq T'$ be two triangular sets, with $T = (T_1, \ldots, T_n)$ and $T' = (T'_1, \ldots, T'_n)$. The least integer ℓ such that $T_\ell \neq T'_\ell$ is called the *level* of the pair $\{T, T'\}$. The pair $\{T, T'\}$ is *critical* if T_ℓ and T'_ℓ are not coprime in $k[X_1, \ldots, X_{\ell-1}]/\langle T_1, \ldots, T_{\ell-1} \rangle[X_\ell]$. A family of triangular sets $\mathbf{T}$ is *non-critical* if it has no critical pairs, otherwise it is said to be *critical*.

The pair $\{T^1, T^2\}$ in the above example has level 1 and is critical. Consider $U^{1,1} = (X_1-1, X_2)$, $U^{1,2} = (X_1-2, X_2)$, $U^{2,1} = (X_1-1, X_2-1)$ and $U^{2,2} = (X_1-3, X_2-1)$. Observe that $\mathbf{U} = \{U^{1,1}, U^{1,2}, U^{2,1}, U^{2,2}\}$ is a non-critical triangular decomposition of I refining $\{T^1, T^2\}$ and that $\mathbf{U}_{\leq 1}$ is a triangular decomposition $I \cap k[X_1]$.

This notion of critical pair is fundamental. In fact, fast algorithms for the innocuous projection operations Φ_i of Equation (3.3) are not guaranteed for critical decompositions, as shown in the following extension of the previous example. Consider a third triangular set $T^3 = ((X_1-2)(X_1-3), X_2+X_1-3)$. One checks that $\mathbf{V} = \{T^1, T^2, T^3\}$ is a triangular decomposition of $T = ((X_1-1)(X_1-2)(X_1-3), X_2(X_2-1))$. However, projecting an element p from $\{T\}$ to $\mathbf{V}$ requires to compute

$$p \bmod (X_1-1)(X_1-2), \ p \bmod (X_1-1)(X_1-3), \ p \bmod (X_1-2)(X_1-3),$$

whence some redundancies. In general, these redundancies prevent the projecting computation from being quasi-linear w.r.t. $\deg(T)$. But if the triangular decom-

position is non-critical, then there is no more redundancy, and the complexity of projecting p can be hoped to be quasi-linear.

Removing critical pairs of a critical triangular decomposition in order to be able to project fast requires to delete the common factors between the polynomials involved in the decomposition. To do it fast, that is, in quasi-linear time, we use the *coprime factorization* or *GCD-free basis computation* algorithm. Of course to implement this algorithm over a direct product of fields, one first need to be able to compute GCD's over such a product in quasi-linear time.

Since $\mathbb{K}(T)$ is a direct product of fields, any pair of univariate polynomials $f, g \in \mathbb{K}(T)[y]$ admits a GCD h in $\mathbb{K}(T)[y]$, in the sense that the ideals $\langle f, g \rangle$ and $\langle h \rangle$ coincide, see [109]. However, even if f, g are both monic, there may not exist a monic polynomial h in $\mathbb{K}(T)[y]$ such that $\langle f, g \rangle = \langle h \rangle$ holds: consider for instance $f = y + \frac{a+1}{2}$ (assuming that 2 is invertible in k) and $g = y + 1$ where $a \in \mathbb{K}(T)$ satisfies $a^2 = a$, $a \neq 0$ and $a \neq 1$. GCD's with non-invertible leading coefficients are of limited practical interest here; this leads us to the following definition.

Definition 3.1.7. Let f, g be in $\mathbb{K}(T)[y]$. An *extended greatest common divisor (XGCD)* of f and g is a sequence $((h_i, u_i, v_i, T^i), 1 \leq i \leq e)$, where $\mathbf{T} = T^1, \ldots, T^e$ is a non-critical decomposition of T and for all $1 \leq i \leq e$, the elements h_i, u_i, v_i are polynomials in $\mathbb{K}(T^i)[y]$, such that the following holds. Let $f_1, \ldots, f_e = \text{project}(f, \{T\}, \mathbf{T})$ and $g_1, \ldots, g_e = \text{project}(g, \{T\}, \mathbf{T})$; then for $1 \leq i \leq e$, we have:

- h_i is monic or null,

- the inequalities $\deg u_i < \deg g_i$ and $\deg v_i < \deg f_i$ hold,

- h_i divides f_i and g_i in $\mathbb{K}(T^i)[y]$ and

- $h_i = u_i f_i + v_i g_i$ holds.

One easily checks that such XGCD's exists, and can be computed, for instance by applying the D5 Principle to the Euclidean algorithm. To compute GCD's in quasi-linear time over a direct product of fields, we will actually adapt the *Half-GCD* techniques [150] in Section 3.4.

Our last basic ingredient is the suitable generalization of the notion of inverse to direct products of fields.

Definition 3.1.8. A *quasi-inverse* of an element $f \in \mathbb{K}(T)$ is a sequence of couples $((u_i, T^i), 1 \leq i \leq e)$ where $\mathbf{T} = T^1, \ldots, T^e$ is a non-critical decomposition of T and

u_i is an element of $\mathbb{K}(T^i)$ for all $1 \leq i \leq e$, such that the following holds. Let $f_1, \ldots, f_e = \mathrm{project}(f, \{T\}, \mathbf{T})$; then for $1 \leq i \leq e$ we have either $f_i = u_i = 0$, or $f_i u_i = 1$.

Obtaining fast algorithms for GCD's, quasi-inverses and removal of critical pairs requires a careful inductive process that we summarize below, before a more detailed discussion.

- We first need complexity estimates for multiplication modulo a triangular set and projecting w.r.t. triangular decompositions. This is done in Section 3.3.

- Assuming that multiplications and quasi-inverse computations can be computed fast in $\mathbb{K}(T)$, and assuming that we can remove critical pairs from critical triangular decompositions of $\langle T \rangle$, we obtain in Section 3.4 a fast algorithm for computing GCD's in $\mathbb{K}(T)[y]$. Note that [85] states that GCD's over products of fields can be computed in quasi-linear time, but with no proof.

- Assuming that GCD's can be computed fast in $\mathbb{K}(T_1, \ldots, T_{n-1})[X_n]$, we present fast algorithms for quasi-inverses in $\mathbb{K}(T)$ (Section 3.5), coprime factorization for polynomials in $\mathbb{K}(T_1, \ldots, T_{n-1})[X_n]$ (Section 3.6) and refining a triangular decomposition $\mathbf{T}$ of T into a non-critical one (Section 3.7).

Figure 3.1 illustrates the **inductive process**. We comment this picture hereafter. We denote by $\mathbb{L}_i$ the residue class ring $\mathbb{K}[X_1, \ldots, X_i]/\langle T_1, \ldots, T_i \rangle$ and by δ_i the degree of the extension from $\mathbb{K}$ to $\mathbb{L}_i$, that is, the dimension of $\mathbb{L}_i$ as a vector space over $\mathbb{K}$. Here are the steps of this inductive process depicted above.

1. Assuming that GCDs can be computed fast in $\mathbb{L}_{n-1}[X_n]$, we obtain

   (a) fast coprime factorization in $\mathbb{L}_{n-1}[X_n]$,

   (b) fast computations of quasi-inverses in $\mathbb{L}_n$.

2. Based on fast coprime factorization in $\mathbb{L}_{n-1}[X_n]$, we obtain fast removal of critical pairs in $\mathbb{L}_n$, and, then, fast evaluation of the projection map.

3. Based on all the previous fast operations in $\mathbb{L}_n$, we adapt the Half-GCD algorithms in $\mathbb{L}_n[X_{n+1}]$ and preserve its complexity class as if $\mathbb{L}_n$ were a field.

The theorems below are the basic blocks for our inductive process, which yields our main results:
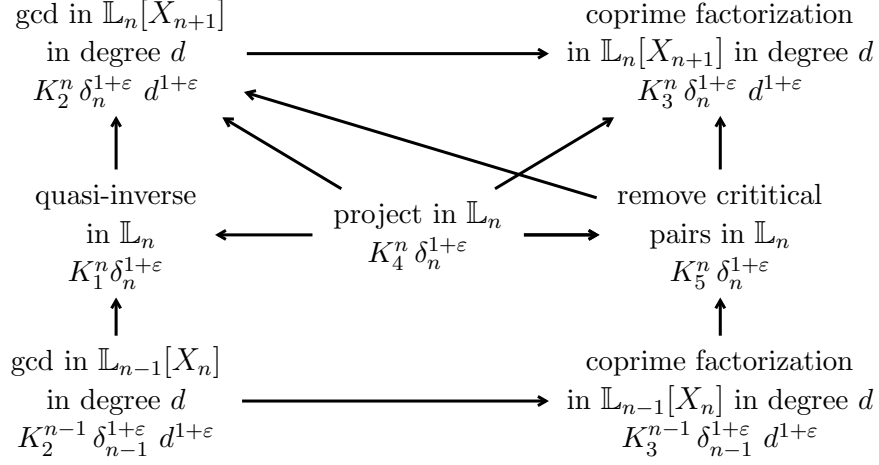
Figure 3.1: A View of the Inductive Process

Theorem 3.1.9. *For any $\varepsilon > 0$, we have $A_\varepsilon > 0$ such that addition, multiplication and quasi-inversion in $\mathbb{K}(T)$ can be computed in $A_\varepsilon^n \deg(T)^{1+\varepsilon}$ operations in k.*

Theorem 3.1.10. *There exists $G > 0$, and for any $\varepsilon > 0$, there exists $A_\varepsilon > 0$, such that one can compute an extended greatest common divisor of polynomials in $\mathbb{K}(T)[y]$, with degree at most d, using at most $G\, A_\varepsilon^n\, d^{1+\varepsilon} \deg(T)^{1+\varepsilon}$ operations in k.*

This is a joint work with Xavier Dahan, Marc Moreno Maza and Éric Schost. This chapter is an extended version of our prepublication [41]: it appears in the proceedings of *Transgressive Computing 2006*, a conference in honor of Jean Della Dora, one of the three inventors of the D5 Principle.

3.2 Complexity Notions

We start by recalling basic results for operations on univariate polynomials.

Definition 3.2.1. A *multiplication time* is a map $\mathsf{M} : \mathbb{N} \to \mathbb{R}$ 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 inequalities $\frac{\mathsf{M}(d)}{d} \leq \frac{\mathsf{M}(d')}{d'}$ and $\mathsf{M}(dd') \leq \mathsf{M}(d)\mathsf{M}(d')$ hold.

Note that in particular that the inequalities $\mathsf{M}(d) \geq d$ and $\mathsf{M}(d) + \mathsf{M}(d') \leq \mathsf{M}(d + d')$ hold for all d, d'. Using the result of [28], that follows the work of Schönhage and Strassen, we know that there exists $c \in \mathbb{R}$ such that the function

$d \mapsto c\,d\,\mathrm{logp}(d)\,\mathrm{logp}\,\mathrm{logp}(d)$ is a multiplication time. In what follows, the function logp is defined by $\mathrm{logp}(x) = 2\log_2(\max\{2, x\})$: this function turns out to be more convenient than the classical logarithm for handling inequalities.

Fast polynomial multiplication is the basis of many other fast algorithms: Euclidean division, computation of the subproduct tree (see Chapter 10 in [57] and Section 3.6 of this article), and multiple remaindering.

Proposition 3.2.2. *There exists a constant $C \geq 1$ such that the following holds over any ring R. Let M be a multiplication time. Then:*

1. *Dividing in $R[X]$ a polynomial of degree less than $2d$ by a monic polynomial of degree at most d requires at most $5\mathsf{M}(d) + O(d) \leq C\,\mathsf{M}(d)$ operations $(+, \times)$ in R.*

2. *Let F be a monic polynomial of degree d in $R[X]$. Then additions and multiplications in $R[X]/F$ requires at most $6\,\mathsf{M}(d) + O(d) \leq C\,\mathsf{M}(d)$ operations $(+, \times)$ in R.*

3. *Let $F_1, \ldots, F_s$ be non-constant monic polynomials in $R[X]$, with sum of degrees d. Then one can compute the subproduct tree associated to $F_1, \ldots, F_s$ using at most $\mathsf{M}(d)\,\mathrm{logp}(d)$ operations $(+, \times)$ in R.*

4. *Let $F_1, \ldots, F_s$ be non-constant monic polynomials in $R[X]$, with sum of degrees d. Then given A in $R[X]$ of degree less than d, one can compute $A \bmod F_1, \ldots, A \bmod F_s$ within $11\,\mathsf{M}(d)\,\mathrm{logp}(d) + O(d\,\mathrm{logp}(d)) \leq C\,\mathsf{M}(d)\,\mathrm{logp}(d)$ operations $(+, \times)$ in R.*

5. *Assume that R is a field. Then, given two polynomials in $R[X]$ of degree at most d, computing their monic GCD and their Bézout coefficients can be done in no more than $33\,\mathsf{M}(d)\,\mathrm{logp}(d) + O(d\,\mathrm{logp}(d)) \leq C\,\mathsf{M}(d)\,\mathrm{logp}(d)$ operations $(+, \times, /)$ in R.*

6. *Assume that R is a field and that F is a monic squarefree polynomial in $R[X]$ of degree d. Then, computing a quasi-inverse modulo F of a polynomial $G \in R[X]$ of degree less than d can be done in no more than $71\,\mathsf{M}(d)\,\mathrm{logp}(d) + O(d\,\mathrm{logp}(d)) \leq C\,\mathsf{M}(d)\,\mathrm{logp}(d)$ operations $(+, \times, /)$ in R.*

PROOF. The first point is proved in Theorem 9.6 of [57] and implies the second one. The third and fourth points are proved in Lemma 10.4 and Theorem 10.15 of the same book. The fifth point is reported in Theorem 11.5 of that book (with a

better constant), and is a particular case of Section 3.4 of this article. If F has no multiple factors in $R[X]$, a quasi-inverse of G modulo F can be obtained by at most two extended GCD computations and one division with entries of degree at most d. Using estimates for the GCD leads to the result claimed in point 6. $\square$

We now define our key complexity notion, arithmetic time for triangular sets.

Definition 3.2.3. An *arithmetic time* is a function $T \mapsto \mathsf{A}_n(T)$ with real positive values and defined over all triangular sets in $k[X_1, \ldots, X_n]$ such that the following conditions hold.

(E_0) For every triangular decomposition $\mathbf{T} = T^1, \ldots, T^e$ of T, we have $\mathsf{A}_n(T^1) + \cdots + \mathsf{A}_n(T^e) \leq \mathsf{A}_n(T)$.

(E_1) Every addition or multiplication in $\mathbb{K}(T)$ can be done in at most $\mathsf{A}_n(T)$ operations in k.

(E_2) Every quasi-inverse in $\mathbb{K}(T)$ can be computed in at most $\mathsf{A}_n(T)$ operations in k.

(E_3) Given a triangular decomposition $\mathbf{T}$ of T, one can compute a *non-critical* triangular decomposition $\mathbf{T}'$ which refines $\mathbf{T}$, in at most $\mathsf{A}_n(T)$ operations in k.

(E_4) For every $\alpha \in \mathbb{K}(T)$ and every non-critical triangular decomposition $\mathbf{T}$ of T, one can compute $\mathrm{project}(\alpha, \{T\}, \mathbf{T})$ in at most $\mathsf{A}_n(T)$ operations in k.

Our main goal in this work is then to give estimates for arithmetic times. This is done through an inductive proof; the following proposition gives such a result for the base case, triangular sets in one variable.

Proposition 3.2.4. *If $n = 1$, then $T \in k[X_1] \mapsto C\,\mathsf{M}(\deg T)\log\mathsf{p}(\deg T)$ is an arithmetic time.*

PROOF. A triangular set in one variable is simply a squarefree monic polynomial in $k[X_1]$. Hence, (E_1), (E_2) and (E_4) respectively follow from points 2, 6 and 4 in Proposition 3.2.2. Property (E_0) is clear. Since $n = 1$, all triangular decompositions are non-critical, and (E_3) follows. $\square$

3.3 Basic Complexity Results: Multiplication and Projection

This section is devoted to give first complexity results for triangular sets: we give upper bounds on the cost of multiplication, and projection. In general, we do not know how to perform this last operation in quasi-linear time; however, when the decomposition is non-critical, quasi-linearity can be reached.

Proposition 3.3.1. *Let* M *be a multiplication function, and let* C *be the constant from Proposition 3.2.2. Let* T *be a triangular set in* $k[X_1, \ldots, X_n]$. *Then:*

- *Additions and multiplications modulo* T *can be done in at most* $C^n \prod_{i \leq n} \mathsf{M}(\deg_i T)$ *operations in* k.

- *If* $\mathbf{T}$ *is a non-critical decomposition of* T, *then for any* h *in* $\mathbb{K}(T)$, *one can compute* $\mathrm{project}(h, \{T\}, \mathbf{T})$ *in at most* $n\, C^n \prod_{i \leq n} \mathsf{M}(\deg_i T) \operatorname{logp}(\deg_i T)$ *operations in* k.

PROOF. The first part of the proposition is easy to deal with: the case of additions is obvious, using the inequality $\mathsf{M}(d) \geq d$; as to multiplication, an easy induction using point (1) in Proposition 3.2.2 gives the result. The end of the proof uses point (4) in Proposition 3.2.2; the non-critical assumption is then used through the following lemma. $\square$

Lemma 3.3.2. *Consider a non-critical decomposition* $\mathbf{T}$ *of the triangular set* $T = (T_1, \ldots, T_n)$. *Write* $\mathbf{T}_{\leq n-1} = \{U^1, \ldots, U^s\}$, *and, for all* $i \leq s$, *denote by* $T^{i,1}, \ldots, T^{i,e_i}$ *the triangular sets in* $\mathbf{T}$ *such that* $T^{i,j} \cap k[X_1, \ldots, X_{n-1}] = U^i$ *(thus* $\mathbf{T}$ *is the set of all* $T^{i,j}$, *with* $i \leq s$ *and* $j \leq e_i$). *Then* $\mathbf{T}_{\leq n-1}$ *is a non-critical decomposition of the triangular set* $(T_1, \ldots, T_{n-1})$. *Moreover, for all* $i \leq s$, *we have:*

$$\sum_{j \leq e_i} \deg_n T^{i,j} = \deg_n T.$$

PROOF. Let U^i and U^j be in $\mathbf{T}_{\leq n-1}$, and T^i and T^j be in $\mathbf{T}$ such that $U^i = T^i \cap k[X_1, \ldots, X_{n-1}]$ and $U^j = T^j \cap k[X_1, \ldots, X_{n-1}]$. Since U^i and U^j differ, the level ℓ of T^i and T^j is at most $n-1$. Then, coprimality at level ℓ for T^i and T^j implies coprimality at level ℓ for U^i and U^j. Therefore, $\mathbf{T}_{\leq n-1}$ is a non-critical decomposition of the triangular set $(T_1, \ldots, T_{n-1})$.

We prove the second claim. Let $i \leq s$. The pairwise coprimality of $T^{(i,1)}, \ldots, T^{(i,e_i)}$ modulo $\langle U^{(i)} \rangle$ implies that

$$\bigcap_{j \leq e_i} \langle T^{(i,j)} \rangle = \langle U^i \rangle + \langle T^{(i,1)} \cdots T^{(i,e_i)} \rangle.$$

We write $A_i = \langle U^i \rangle + \langle T^{(i,1)} \cdots T^{(i,e_i)} \rangle$. From the definition of a triangular decomposition, we have the equality between ideals in $k[X_1, \ldots, X_n]$:

$$\langle T \rangle = \bigcap_{i \leq s} \bigcap_{j \leq e_i} \langle T^{(i,j)} \rangle = \bigcap_{i \leq s} A_i. \tag{3.6}$$

On the other hand, by definition of $\mathbf{T}_{\leq n-1}$ we have:

$$\langle T \rangle = \bigcap_{i \leq s} \langle U^i \rangle + \langle T_n \rangle. \tag{3.7}$$

Since $\mathbf{T}_{\leq n-1}$ is a triangular decomposition of the triangular set $(T_1, \ldots, T_{n-1})$, the ideals $\langle U^i \rangle$ are pairwise coprime. We deduce that $A_i = \langle U^i \rangle + \langle T_n \rangle$ holds. The conclusion follows. $\square$

As an illustration, consider again, for $n = 2$, the triangular sets

$$
\begin{aligned}
T^1 &= ((X_1 - 1)(X_1 - 2),\ X_2) \\
T^2 &= ((X_1 - 1)(X_1 - 3),\ X_2 - 1) \\
\text{and } T^3 &= ((X_1 - 2)(X_1 - 3),\ X_2 + X_1 - 3).
\end{aligned}
$$

These triangular sets form a critical decomposition $\mathbf{T}$ of the ideal $\langle T^1 \rangle \cap \langle T^2 \rangle \cap \langle T^3 \rangle$, which is also generated by $T = ((X_1 - 1)(X_1 - 2)(X_1 - 3), X_2(X_2 - 1))$.

Here, $\mathbf{T}_{\leq 1}$ is given by $\{U^1, U^2, U^3\} = \{(X_1 - 1)(X_1 - 2), (X_1 - 1)(X_1 - 3), (X_1 - 1)(X_1 - 3)\}$, so that $s = 3$. Take for instance $U^1 = (X_1 - 1)(X_1 - 2)$; then we have $e_1 = 1$ and $T^{1,e_1} = T^1$. Note then that $\deg_2 T^{1,e_1} = 1$ differs from $\deg_2 T = 2$, so the conclusion of the previous lemma is indeed violated.

3.4 Fast GCD Computations Modulo Triangular Sets

GCD's of univariate polynomials over a field can be computed in quasi-linear time by means of the *Half-GCD* algorithm [24, 150]. We show how to adapt this tech-

nique over the direct product of fields $\mathbb{K}(T)$ and how to preserve its complexity class. Throughout this section, we consider an arithmetic time $T \mapsto \mathsf{A}_n(T)$ for triangular sets in $k[X_1, \ldots, X_n]$.

Proposition 3.4.1. *For all $a, b \in \mathbb{K}(T)[y]$ with $\deg a$, $\deg b \leq d$, one can compute an extended greatest common divisor of a and b in $O(\mathsf{M}(d)\log(d))\mathsf{A}_n(T)$ operations in k.*

We prove this result by describing our GCD algorithm over the direct product of fields $\mathbb{K}(T)$ and its complexity estimate. We start with two auxiliary algorithms.

Monic forms. Any polynomial over a field can be made monic by division through its leading coefficient. Over a product of fields, this division may induce splittings. We now study this issue.

Definition 3.4.2. A *monic form* of $f \in \mathbb{K}(T)[y]$ with degree d is a sequence of quadruples $((u_i, v_i, m_i, T_i), 1 \leq i \leq e)$, where $\mathbf{T} = T^1, \ldots, T^e$ is a non-critical decomposition of T, u_i, v_i are in $\mathbb{K}(T^i)$ and m_i is in $\mathbb{K}(T^i)[y]$ for all $1 \leq i \leq e$, and such that the following holds.
  Let $f_1, \ldots, f_e = \mathrm{project}(f, \{T\}, \mathbf{T})$. Denote by $\mathrm{lc}(f_i)$ the leading coefficient of f_i. Then, for all $1 \leq i \leq e$ we have $u_i = \mathrm{lc}(f_i)$, and $m_i = v_i f_i$, and either $u_i = v_i = 0$ or $u_i v_i = 1$.

Observe that for all $1 \leq i \leq e$, the polynomial m_i is monic or null.

  The following algorithm shows how to compute a monic form. This function uses a procedure $\mathsf{quasiInverse}(\mathbf{f}, \mathbf{T})$. This procedure takes as input a triangular decomposition $\mathbf{T} = T^1, \ldots, T^e$ of T and a sequence $\mathbf{f} = f_1, \ldots, f_e$ in $\mathbb{K}(T^1)[y] \times \cdots \times \mathbb{K}(T^e)[y]$ and returns a sequence $(((f_{ij}, T^{ij}), 1 \leq j \leq e_i), 1 \leq i \leq e)$ where $((f_{ij}, T^{ij}), 1 \leq j \leq e_i)$ is a quasi-inverse of f_i modulo T^i and such that $(T^{ij}, 1 \leq j \leq e_i, 1 \leq i \leq e)$ is a non-critical refinement of $\mathbf{T}$. Its complexity is studied in Section 3.5.
  The number at the end of a line, multiplied by $\mathsf{A}_n(T)$, gives an upper bound for the total time spent at this line. Therefore, the following algorithm computes a monic form of f in at most $(6d + 4)\mathsf{A}_n(T)$ operations in k.

Division with monic remainder. The previous notion can then be used to compute Euclidean divisions, producing *monic* remainders: they will be required in our fast Euclidean algorithm for XGCD's.

Definition 3.4.3. Let $f, g \in \mathbb{K}(T)[y]$ with g monic. A *division with monic remainder* of f by g is a sequence of tuples $((g_i, q_i, v_i, u_i, r_i, T^i), 1 \leq i \leq e)$ such that $\mathbf{T} =$

Algorithm 8 Monic Form

$\mathsf{monic}(f, T) ==$

1: $\mathbf{T} \leftarrow \{T\};\quad \mathbf{v} \leftarrow (0);\quad g \leftarrow f$
2: **while** $g \neq 0$ **do**
3: $\quad \mathbf{u} \leftarrow \mathrm{project}(\mathrm{lc}(g), \{T\}, \mathbf{T})$ $\hfill [d+1]$
4: $\quad (\mathbf{w}, \mathbf{T}') \leftarrow \mathsf{quasiInverse}(\mathbf{u}, \mathbf{T})$ $\hfill [d+1]$
5: $\quad \mathbf{v} \leftarrow \mathrm{project}(\mathbf{v}, \mathbf{T}, \mathbf{T}')$ $\hfill [d+1]$
6: $\quad$ **for** $1 \leq i \leq \#\mathbf{v}$ **do**
7: $\quad\quad$ **if** $v_i = 0$ **then** $v_i \leftarrow w_i$ $\hfill [d+1]$
8: $\quad$ **end for**
9: $\quad \mathbf{T} \leftarrow \mathbf{T}'$
10: $\quad g \leftarrow g - \mathrm{leadingTerm}(g)$
11: **end while**
12: $\mathbf{f} \leftarrow \mathrm{project}(f, \{T\}, \mathbf{T})$ $\hfill [d]$
13: $\mathbf{u} \leftarrow \mathrm{lc}(\mathbf{f})$
14: $\mathbf{m} \leftarrow \mathbf{v} \cdot \mathbf{f}$ $\hfill [d]$
15: **return** $((u_i, v_i, m_i, T^i), 1 \leq i \leq \#\mathbf{T})$

$T^1, \ldots, T^e$ is a non-critical decomposition of T, and, for all $1 \leq i \leq e$, we have $u_i, v_i \in \mathbb{K}(T^i)$ and $g_i, q_i, r_i, \in \mathbb{K}(T^i)[y]$, and such that the following holds.

Let $f_1, \ldots, f_e = \mathrm{project}(f, \{T\}, \mathbf{T})$ and $g_1, \ldots, g_e = \mathrm{project}(g, \{T\}, \mathbf{T})$. Then, for all $1 \leq i \leq e$, the polynomial r_i is null or monic, we have either $u_i = v_i = 0$ or $u_i v_i = 1$, and the polynomials q_i and $u_i r_i$ are the quotient and remainder of f_i by g_i in $\mathbb{K}(T^i)[y]$.

The following algorithm computes a division with monic remainder of f by g and requires at most $(5\mathsf{M}(d) + O(d))\mathsf{A}_n(T)$ operations in k. We write $(q, r) = \mathrm{div}(f, g)$ for the quotient and the remainder in the (standard) division with remainder in $\mathbb{K}(T)[y]$.

Algorithm 9 Division with Monic Remainder

$\mathsf{mdiv}(f, g, T) ==$

1: $(q, r) \leftarrow \mathrm{div}(f, g)$ $\hfill [5\mathsf{M}(d) + O(d)]$
2: $((u_i, v_i, r_i, T^i), 1 \leq i \leq \#\mathbf{T}) \leftarrow \mathsf{monic}(r, T)$ $\hfill [O(d)]$
3: $(q_i, 1 \leq i \leq \#\mathbf{T}) \leftarrow \mathrm{project}(q, \{T\}, \mathbf{T})$ $\hfill [d+1]$
4: $(g_i, 1 \leq i \leq \#\mathbf{T}) := \mathrm{project}(g, \{T\}, \mathbf{T})$ $\hfill [d]$
5: **return** $((g_i, q_i, r_i, u_i, v_i, T^i), 1 \leq i \leq \#\mathbf{T})$

Most of our operations comes in two flavors: one takes input polynomials $f, g, \ldots$ in a ring $\mathbb{K}(T)[y]$ for a given triangular set T, the other one takes projections $\mathbf{f} = f_1, \ldots, f_e$ and $\mathbf{g} = g_1, \ldots, g_e$ of these polynomials on a triangular decomposition $\mathbf{T}$ of T. In Section 3.5 we detail the case of quasi-inversions with Algorithms 11 and 12.

In the second algorithm, the dominant cost is the the call to the first one. Hence, the two algorithms have essentially the same running time.

For monic remainder computations, the two flavors are also needed. Algorithm 9 takes as input a triangular set T and polynomials $f, g \in \mathbb{K}(T)[y]$. The "projection flavor" is built on top of Algorithm 9:

- it takes as input a triangular decomposition $\mathbf{T} = T^1, \ldots, T^e$ of T, together with $\mathbf{f} = f_1, \ldots, f_e$ and $\mathbf{g} = g_1, \ldots, g_e$, which are sequences of polynomials in $\mathbb{K}(T^1)[y], \ldots, \mathbb{K}(T^e)[y]$;

- it proceeds with e calls to **mdiv** of Algorithm 9, namely $\mathbf{mdiv}(f_1, g_1, T^1)$, $\ldots$, $\mathbf{mdiv}(f_e, g_e, T^e)$.

- this leads to additional splits, which requires removal of critical pairs.

The dominant cost is the e calls to **mdiv**. Therefore, in each situation, the total cost is still bounded by $O\big(\mathsf{M}(d) + d\big)\mathsf{A}_n(T)$.

XGCD's. We are now ready to generalize the *Half-GCD* method as exposed in [150]. We introduce the following operations. For $a, b \in \mathbb{K}(T)[y]$ with $0 < \deg b < \deg a = d$, each of the following algorithms $\mathsf{M}_{\mathbf{gcd}}(a, b, T)$ and $\mathsf{M}_{\mathbf{hgcd}}(a, b, T)$ returns a sequence $((M_1, T^1), \ldots, (M_e, T^e))$ where

(s_1) $\mathbf{T} = T^1, \ldots, T^e$ is a non-critical triangular decomposition of T,

(s_2) M_i is a square matrix of order 2 with coefficients in $\mathbb{K}(T^i)[y]$,

such that, if we define $(a_1, \ldots, a_e) = \mathrm{project}(a, \{T\}, \mathbf{T})$ and $(b_1, \ldots, b_e) = \mathrm{project}(b, \{T\}, \mathbf{T})$, then, for all $1 \leq i \leq e$, defining $(t_i, s_i) = (a_i, b_i) \, {}^t M_i$, we have

(s_3) in the case of $\mathrm{M}_{\mathrm{gcd}}$, the polynomial t_i is a GCD of a_i, b_i and $s_i = 0$ holds,

(s_3') in the case of $\mathrm{M}_{\mathrm{hgcd}}$, the ideals $\langle t_i, s_i \rangle$ and $\langle a_i, b_i \rangle$ of $\mathbb{K}(T^i)[y]$ are identical, and $\deg s_i < \lceil d/2 \rceil \leq \deg t_i$ holds.

The algorithm below implements $\mathsf{M}_{\mathbf{gcd}}(a, b, T)$, and is an extension of the analogue algorithm known over fields. Observe that if the input triangular set T is not decomposed during the algorithm, in particular if $\mathbb{K}(T)$ is a field, then the algorithm yields generators of the ideal $\langle a, b \rangle$. If T is decomposed, then the lines from 23 to 31 guarantee that $\mathsf{M}_{\mathbf{gcd}}(a, b, T)$ generates a non-critical triangular decomposition of T.

Algorithm 10 Half-GCD Modulo a Triangular Set

$\mathsf{M}_{\mathbf{g}cd}$(a,b,T) ==

1: $\mathbf{G} \leftarrow [\,]; \quad \mathbf{T} \leftarrow [\,]$

2: $((M_i, T^i), 1 \le i \le e) \leftarrow \mathsf{M}_{\mathsf{h}gcd}(a, b, T)$ 　　　　　　　　　　　$[H(d)]$

3: $(a_1, \ldots, a_e) \leftarrow \text{project}(a, (T^i, 1 \le i \le e))$ 　　　　　　　　　$[O(d)]$

4: $(b_1, \ldots, b_e) \leftarrow \text{project}(b, (T^i, 1 \le i \le e))$ 　　　　　　　　　$[O(d)]$

5: **for** i in $1 \cdots e$ **do**

6: 　　$(t_i, s_i) \leftarrow (a_i, b_i) \; {}^t M_i$ 　　　　　　　　　　　　　　$[4\,\mathsf{M}(d) + O(d)]$

7: 　　**if** $s_i = 0$ **then**

8: 　　　　$\mathbf{G} \leftarrow \mathbf{G}, (M_i, T^i); \quad \mathbf{T} \leftarrow \mathbf{T}, T^i$

9: 　　**end if**

10: 　　$((s_{ij}, q_{ij}, r_{ij}, u_{ij}, v_{ij}, T^{ij}), 1 \le j \le e_i) \leftarrow \mathsf{mdiv}(t_i, s_i, T^i)$ 　　$[\frac{5}{2}\mathsf{M}(d) + O(d)]$

11: 　　$(M_{ij}, 1 \le j \le e_i) \leftarrow \text{project}(M_i, (T^{ij}, 1 \le j \le e_i))$ 　　　$[O(d)]$

12: 　　**for** j in $1 \cdots e_i$ **do**

13: 　　　　$M_{ij} \leftarrow \begin{pmatrix} 0 & 1 \\ v_{ij} & -q_{ij}v_{ij} \end{pmatrix} M_{ij}$ 　　　　　　$[2\,\mathsf{M}(d) + O(d)]$

14: 　　　　**if** $r_{ij} = 0$ **then** $\mathbf{G} \leftarrow \mathbf{G}, (M_{ij}, T^i); \quad \mathbf{T} \leftarrow \mathbf{T}, T^{ij}$

15: 　　　　$((N_{ijk}, T^{ijk}), 1 \le k \le e_{ij}) \leftarrow \mathsf{M}_{\mathbf{g}cd}(s_{ij}, r_{ij}, T^{ij})$ 　　$[G(d/2)]$

16: 　　　　$(M_{ijk}, 1 \le k \le e_{ij}) \leftarrow \text{project}(M_{ij}, (T^{ijk}, 1 \le k \le e_{ij}))$ 　$[O(d)]$

17: 　　　　**for** k in $1 \cdots e_{ij}$ **do**

18: 　　　　　　$M_{ijk} \leftarrow N_{ijk} M_{ijk}$ 　　　　　　　　　　　$[8\,\mathsf{M}(d) + O(d)]$

19: 　　　　　　$\mathbf{G} \leftarrow \mathbf{G}, (M_{ijk}, T^{ijk}); \quad \mathbf{T} \leftarrow \mathbf{T}, T^{ijk}$

20: 　　　　**end for**

21: 　　**end for**

22: **end for**

23: $\mathbf{T}' \leftarrow \mathsf{RemoveCriticalPairs}(\mathbf{T})$ 　　　　　　　　　　　　　　$[1]$

24: $\mathbf{Res} \leftarrow [\,]$

25: **for** $(M, T) \in \mathbf{G}$ **do**

26: 　　$\mathsf{U} \leftarrow \text{decomp}(T, \mathbf{T}')$

27: 　　$(M_\ell, 1 \le \ell \le \#\mathsf{U}) \leftarrow \text{project}((\,)M, \{T\}, \mathsf{U})$ 　　　　　$[O(d)]$

28: 　　**for** $1 \le \ell \le \#\mathsf{U}$ **do**

29: 　　　　$\mathbf{Res} \leftarrow \mathbf{Res}, (M_i, U^i)$

30: 　　**end for**

31: **end for**

32: **return Res**

Similarly, the Half-GCD algorithm can be adapted to $\mathbb{K}(T)[y]$, leading to an implementation of $\mathsf{M}_{\mathsf{h}gcd}(a, b, T)$. It has a structure very similar to $\mathsf{M}_{\mathsf{g}cd}(a, b, T)$, see [150] for details in the case when the coefficients lie in a field.

Now, we give running time estimates for $\mathsf{M}_{\mathsf{h}gcd}(a, b, T)$ and $\mathsf{M}_{\mathsf{g}cd}(a, b, T)$. For $0 < \deg b < \deg a = d$, we denote by $G(d)$ and $H(d)$ respective upper bounds for the running time of $\mathsf{M}_{\mathsf{g}cd}(a, b)$ and $\mathsf{M}_{\mathsf{h}gcd}(a, b)$, in the sense that both operations can be done in respective times $G(d)\mathsf{A}_n(T)$ and $H(d)\mathsf{A}_n(T)$.

The number at the end of a line, multiplied by $\mathsf{A}_n(T)$, gives an upper bound of the running time of this line. These estimates follow from the super-linearity of the arithmetic time for triangular sets, the running time estimates of the operation $\mathrm{mdiv}(f, g, T)$ and classical degree bounds for the intermediate polynomials in the Extended Euclidean Algorithms; see for instance Chapter 3 in [57]. Therefore, counting precisely the degrees appearing, we have: $G(d) \leq G(d/2) + H(d) + (33/2)\mathsf{M}(d) + O(d)$. The operation $\mathsf{M}_{\mathsf{h}gcd}(a, b, T)$ makes two recursive calls with input polynomials of degree at most $d/2$, leading to $H(d) \leq 2H(d/2) + (33/2)\mathsf{M}(d) + O(d)$. The superlinearity of M implies

$$H(d) \leq \frac{33}{2}\mathsf{M}(d)\log d + O(d \log d) \ \text{ and } \ G(d) \leq 2H(d) + 2\mathsf{M}(d) + O(d).$$

This leads to the result reported in Proposition 3.4.1.

We conclude with a specification of a function used in the remaining sections. For a triangular decomposition $\mathbf{T} = T^1, \ldots, T^e$ of T, two sequences $\mathbf{f} = f_1, \ldots, f_e$ and $\mathbf{g} = g_1, \ldots, g_e$ of polynomials in $\mathbb{K}(T^1)[y], \ldots, \mathbb{K}(T^e)[y]$, the operation $\mathsf{xgcd}(\mathbf{f}, \mathbf{g}, \mathbf{T})$ returns a sequence $(((g_{ij}, u_{ij}, v_{ij}, T^{ij}), 1 \leq j \leq e_i), 1 \leq i \leq e)$ where $((g_{ij}, u_{ij}, v_{ij}, T^{ij}), 1 \leq j \leq e_i)$ is an extended greatest common divisor of f_i and g_i and such that $(T^{ij}, 1 \leq j \leq e_i, 1 \leq i \leq e)$ is a non-critical refinement of $\mathbf{T}$.

Proposition 3.4.1 implies that if $f_1, \ldots, f_e, g_1, \ldots, g_e$ have degree at most d then $\mathsf{xgcd}(\mathbf{f}, \mathbf{g}, \mathbf{T})$ runs in at most $O(\mathsf{M}(d)\log(d))\mathsf{A}_n(T)$ operations in k.

3.5 Fast Computation of Quasi-inverses

Throughout this section, we consider an arithmetic time A_{n-1} for triangular sets in $n - 1$ variables. We explain how a quasi-inverse can be computed fast with the algorithms *project*, *xgcd*, and *RemoveCriticalPairs*.

Proposition 3.5.1. *Let $T = (T_1, \ldots, T_n)$ be a triangular set with $\deg_i(T) = d_i$ for*

all $1 \leq i \leq n$. Let f be in $\mathbb{K}(T)$. Then one can compute a quasi-inverse of f modulo T in $O\big(\mathsf{M}(d_n)\log(d_n)\big)\mathsf{A}_{n-1}(T_{<n})$ operations in k.

With Algorithm 11, we consider first the case where f is a non-constant polynomial and its degree w.r.t. X_n is positive and less than d_n; the algorithm is followed by the necessary explanations. Here, the quantity at the end a line, once multiplied by $\mathsf{A}_{n-1}(T_{<n})$, gives the total amount of time spent at this line. At the end of this section, we briefly discuss the other cases to be considered for f.

Algorithm 11 Quasi-inverse

$\mathsf{quasiInverse}(f, T) ==$

1: $((g_i, u_i, v_i, T^i_{<n}), 1 \leq i \leq e) \leftarrow \mathsf{xgcd}(f, T_n, T_{<n})$ $\qquad\qquad\big[O\big(\mathsf{M}(d_n)\log(d_n)\big)\big]$

2: $(T^i_n, \ldots, T^e_n) \leftarrow \mathrm{project}(T_n, \{T_{<n}\}, \{T^1_{<n}, \ldots, T^e_{<n}\})$ $\qquad\qquad [O(d_n)]$

3: $(f_i, \ldots, f_e) \leftarrow \mathrm{project}(f, \{T_{<n}\}, \{T^1_{<n}, \ldots, T^e_{<n}\})$ $\qquad\qquad [O(d_n)]$

4: $\mathbf{T} \leftarrow \{\}; \quad \mathbf{C} \leftarrow \{\}; \quad \mathbf{result} \leftarrow \{\}$

5: **for** $i = 1 \ldots e$ **do**

6: $\quad$ **if** $\deg(g_i) = 0$ **then**

7: $\qquad \mathbf{C} \leftarrow \mathbf{C}, \ (u_i, T^i_{<n} \cup T^i_n); \quad \mathbf{T} \leftarrow \mathbf{T}, \ T^i_{<n} \cup T^i_n$

8: $\quad$ **else if** $\deg(g_i) > 0$ **then**

9: $\qquad \mathbf{C} \leftarrow \mathbf{C}, \ (0, T^i_{<n} \cup g_i); \quad \mathbf{T} \leftarrow \mathbf{T}, \ T^i_{<n} \cup g_i$

10: $\qquad q_i \leftarrow \mathrm{quotient}(T^i_n, g_i)$ $\qquad\qquad [5\mathsf{M}(d_n) + O(d_n)]$

11: $\qquad ((g_{ij}, u_{ij}, v_{ij}, T^{ij}_{<n}), 1 \leq j \leq e_i) \leftarrow \mathsf{xgcd}(f_i, q_i, T^i_{<n})$ $\qquad \big[O\big(\mathsf{M}(d_n)\log(d_n)\big)\big]$

12: $\qquad (T^{i1}_n, \ldots, T^{ie_i}_n) \leftarrow \mathrm{project}(q_i, \{T^i_{<n}\}, \{T^{i1}_{<n}, \ldots, T^{ie_i}_{<n}\})$ $\qquad [O(d_n)]$

13: $\qquad$ **for** $j = 1 \ldots e_i$ **do**

14: $\qquad\quad \mathbf{C} \leftarrow \mathbf{C}, \ (u_{ij}, T^{ij}_{<n} \cup T^{ij}_n); \quad \mathbf{T} \leftarrow \mathbf{T}, \ T^{ij}_{<n} \cup T^{ij}_n$

15: $\qquad$ **end for**

16: $\quad$ **end if**

17: **end for**

18: $\mathbf{T}'_{<\mathbf{n}} \leftarrow \mathsf{RemoveCriticalPairs}(\mathbf{T}_{<\mathbf{n}})$ $\qquad\qquad [O(1)]$

19: **for** $(u, S) \in \mathbf{C}$ **do**

20: $\quad (R^1, \ldots, R^l) \leftarrow \mathrm{decomp}(S_{<n}, \mathbf{T}'_{<\mathbf{n}})$

21: $\quad (S^1_n, \ldots, S^l_n) \leftarrow \mathrm{project}(S_n, \{S_{<n}\}, \{R^1, \ldots, R^l\})$ $\qquad\qquad [O(d_n)]$

22: $\quad (u_1, \ldots, u_l) \leftarrow \mathrm{project}(u, \{S_{<n}\}, \{R^1, \ldots, R^l\})$ $\qquad\qquad [O(d_n)]$

23: $\quad \mathbf{result} \leftarrow \mathbf{result}, \ ((u_k, R^k \cup S^k_n), 1 \leq k \leq l)$

24: **end for**

25: **return result**

We first calculate an extended greatest common divisor of f and T_n modulo the triangular set $T_{<n} = (T_1, \ldots, T_{n-1})$. This induces a non-critical decomposition $\{T^1_{<n}, \ldots, T^e_{<n}\}$ of $T_{<n}$. For further operations, we compute the images of T_n and f over this decomposition.

Let $1 \le i \le e$. If the value of g_i is 1, then u_i is the inverse of f modulo $\{T^i_{<n} \cup T^i_n\}$. Otherwise, $\deg g_i > 0$, and the computation needs to be split into two branches.

In one branch, at line 9, we build the triangular set $\{T^i_{<n} \cup g_i\}$, modulo which f reduces to zero. In the other branch, starting from line 10, we build the triangular set as $\{T^i_{<n} \cup q_i\}$, modulo which f is invertible. Indeed since the triangular set $\{T^i_{<n} \cup q_i\}$ generates a radical ideal, T^i_n is squarefree modulo $\{T^i_{<n}\}$, and $\gcd(f, q_i)$ must be 1 modulo $\{T^i_{<n} \cup q_i\}$. Therefore we can simply use the *xgcd* (step 11) once to compute the quasi-inverse of f modulo $\{T^i_{<n} \cup q_i\}$.

After collecting all the quasi-inverses, we remove the critical pairs in the new family of triangular sets. Since no critical pairs are created at level n in the previous computation, the removal of critical pairs needs only to perform below level n. At the end, we project the inverses and the top polynomials w.r.t the last non-critical decomposition.

We also need quasi-inverse computations in two other different situations. One is when f may not have the same main variable as the triangular set T. This case is essentially covered by induction. We need also to compute quasi-inverses in the sense of quasiInverse$(\mathbf{f}, \mathbf{T})$ introduced in Section 3.4 where $\mathbf{T} = T^1, \ldots, T^e$ is a triangular decomposition of T, and $\mathbf{f} = f_1, \ldots, f_e$ is a sequence of polynomials in $k[X_1, \ldots, X_n]$. As shown by Algorithm 12, This is simply built on top Algorithm 11, with additional splits and removal of critical pairs. The dominant cost is the two xgcd calls. Therefore, in each situation, the total cost is bounded by $O\big(\mathsf{M}(d_n) \log(d_n)\big) \mathsf{A}_{n-1}(T_{<n})$.

3.6 Coprime Factorization

We present first in this section a quasi-linear time algorithm for coprime factorization of univariate polynomials over a field. Other fast algorithms for this problem are given by [60], with a concern for parallel efficiency, and in [17], in a wider setting, but with a slightly worse computation time. Remark that the research announcement [16] has a time complexity that essentially matches ours.

Following the ideas presented in Section 3.4, we then give an adaptation of this algorithm over a direct product of fields given by a triangular set. We will use this tool in Section 3.7 for computing non-critical refinements of a triangular decomposition (see the example in the introduction for a motivation of this idea).

Algorithm 12 Refining Quasi-inverse

quasiInverse($\mathbf{f}, \mathbf{T}$) ==
1: $\mathbf{T}' \leftarrow \{\}; \quad \mathbf{C} \leftarrow \{\}; \mathbf{Res} \leftarrow \{\}$
2: **for** i in $1 \ldots e$ **do**
3: $\quad ((u_{ij}, T^{ij}), 1 \le j \le e_i) \leftarrow$ quasiInverse(f_i, T^i) $\qquad \left[O\big(\mathsf{M}(d_n)\log(d_n)\big)\right]$
4: $\quad$ **for** j in $1 \ldots e_i$ **do**
5: $\quad\quad \mathbf{C} \leftarrow \{\mathbf{C}, (u_{ij}, T^{ij})\}$
6: $\quad\quad \mathbf{T}' \leftarrow \{\mathbf{T}', T^{ij}\}$
7: $\quad$ **end for**
8: **end for**
9: $\mathbf{T}'_{<\mathbf{n}} \leftarrow$ RemoveCriticalPairs($\mathbf{T}'_{<\mathbf{n}}$) $\qquad [O(1)]$
10: **for** (u, S) in $\mathbf{C}$ **do**
11: $\quad (R^k, 1 \le k \le l) \leftarrow \mathrm{decomp}(S_{<n}, \mathbf{T}'_{<\mathbf{n}})$
12: $\quad (S_n^k, 1 \le k \le l) \leftarrow \mathrm{project}(S_n, \{S_{<n}\}, \{R^1, \ldots, R^l\})$ $\qquad [O(d_n)]$
13: $\quad (u_k, 1 \le k \le l) \leftarrow \mathrm{project}(u, \{S_{<n}\}, \{R^1, \ldots, R^l\})$ $\qquad [O(d_n)]$
14: $\quad$ **for** k in $1 \ldots l$ **do**
15: $\quad\quad \mathbf{Res} \leftarrow \{\mathbf{Res}, (u_k, R^k \cup S_n^k)\}$
16: $\quad$ **end for**
17: **end for**
18: **return Res**

3.6.1 GCD-Free Basis

Throughout this section, we consider an arithmetic time A_n for triangular sets in n variables. Definition 3.6.1 deals with the case of univariate polynomials over the base field k whereas Definition 3.6.3 handles the case of coefficients in a direct product of fields.

Definition 3.6.1. Let $A = a_1, \ldots, a_s$ be squarefree polynomials in $k[x]$. Some polynomials $b_1, \ldots, b_t$ in $k[x]$ are a *GCD-free basis* of the set A if $\gcd(b_i, b_j) = 1$ for $i \ne j$, each a_i can be written (necessarily uniquely) as a product of some of the b_j, and each b_j divides one of the a_i. The associated *coprime factorization* of A consists in the factorization of all polynomials a_i in terms of the polynomials $b_1, \ldots, b_t$.

We shall establish the following result.

Proposition 3.6.2. *Let d be the sum of the degrees of $A = a_1, \ldots, a_s$. Then a coprime factorization of A can be computed in $O(\mathsf{M}(d)\log\mathrm{p}(d)^3)$ operations in k.*

For brevity's sake, we will only prove how to compute a GCD-free basis of A, assuming without loss of generality that all a_i have positive degree. Deducing the coprime factorization of A involves some additional bookkeeping operations, keeping

track of divisibility relations; it induces no new arithmetic operations, and thus has no consequence on complexity.

Definition 3.6.3. Let $T \subset k[X_1, \ldots, X_n]$ be a triangular set and $\mathbf{a} = a_1, \ldots, a_s$ be squarefree, monic polynomials in $\mathbb{K}(T)[y]$. A *GCD-free basis of $a_1, \ldots, a_s$* **over** $\mathbb{K}(T)$ is the datum of monic, pairwise coprime, polynomials $\mathbf{b} = b_1, \ldots, b_t$ in $\mathbb{K}(T)[y]$ such that:

- each of the polynomials a_i can be expressed as a product of some of the polynomials b_j;

- each of the polynomials b_j divides one of the polynomials a_i.

Note in particular that the sum of the degrees of the polynomials $\mathbf{b}$ is less than or equal to that of polynomials $\mathbf{a}$. Moreover, Definitions 3.6.1 and 3.6.3 are very similar. However, in the latter case, such a GCD-free basis need not exist in general, even though it does when $\mathbb{K}(T)$ is a field. As for GCD's, the workaround is to take into account possible splittings of $\mathbb{K}(T)$. Let thus $\mathbf{U} = U_1, \ldots, U_e$ be a triangular decomposition of T, for which we write $(a_{i,1}, \ldots, a_{i,e}) = \text{project}(a_i, \mathbf{U})$ for all $i = 1, \ldots, s$. Then, a *GCD-free basis* of $\mathbf{a}$ over $\mathbf{U}$ consists in families of pairwise coprime, monic polynomials $\mathbf{b}_j$ in $\mathbb{K}(U_j)[y]$, for $j \leq e$, such that each $\mathbf{b}_j$ forms a coprime factorization of $a_{1,j}, \ldots, a_{s,j}$ over $\mathbb{K}(U_j)$.

The goal of this section is to give complexity estimates for this task, namely Theorem 3.6.4. To this effect, we will use the following convention concerning the big-Oh notation: *the cost of an algorithm is said to be in $O(g(d)\mathsf{A}(T))$, for some function $g : \mathbb{N} \to \mathbb{R}$, if there exists a constant K independent of d and T, and such that this cost is at most $Kg(d)\mathsf{A}_n(T)$ for any $d \in \mathbb{N}$ and any triangular set T.*

Theorem 3.6.4. *Let T be a triangular set, and $\mathbf{a} = a_1, \ldots, a_s$ be squarefree, monic polynomials in $\mathbb{K}(T)[y]$. One can compute as a GCD-free basis of $\mathbf{a}$ over $\mathbf{U}$,*

$$O\big(\mathsf{M}(d)\log^3(d)\,\mathsf{A}_n(T)\big)$$

operations in k, where $d = \sum_{i \leq s} \deg a_i$.

In Sections 3.6.2, 3.6.3, 3.6.4, 3.6.5 and 3.6.1, we define the subroutines required for our GCD-free basis algorithm. Recall that the cost at given any line in our pseudo-code denotes the total time spent at this line; for simplicity, in what follows, we omit the $O(\ )$ in the complexity estimates attached to the pseudo-code.

3.6.2 Subproduct Tree Techniques

The subproduct tree is a useful construction to devise fast algorithms with univariate polynomials, in particular GCD-free basis. We review this notion briefly and refer to [57] for more details.

Let $m_1, \ldots, m_r$ be monic, non-constant polynomials in $k[x]$. We define a procedure, subProductTree, to generate a subproduct tree Sub associated to $m_1, \ldots, m_r$. If $r = 1$, then Sub is a single node, labeled by the polynomial m_1. Otherwise, let $r' = \lceil r/2 \rceil$, and let Sub_1 and Sub_2 be the subProductTree associated to $m_1, \ldots, m_{r'}$ and $m_{r'+1}, \ldots, m_r$ respectively. Let p_1 and p_2 be the polynomials at the roots of Sub_1 and Sub_2. Then Sub is the tree whose root is labeled by the product $p_1 p_2$ and has children Sub_1 and Sub_2. A *row* of the tree consists in all nodes lying at some given distance from the root. The *depth* of the tree is the number of its non-empty rows. Let $d = \sum_{i=1}^{r} \deg(m_i)$; then the sum of the degrees of the polynomials on any row of the tree is at most d, and its depth is at most $\log_p(d)$.

Subproduct tree techniques are used in the proof of the fourth claim of Proposition 3.2.2 for the operation called *fast simultaneous remainder*.

This statement holds for non-constant monic polynomials over an arbitrary commutative ring with units. Thus, in particular, it holds over a direct product of fields.

In the case of non-necessarily monic polynomials, with coefficients in some $\mathbb{K}(T)$ for a triangular set T, one needs to split the computations. Algorithm 14 covers that generalization. Note that in this case, the nodes of the tree have to be labeled with projections of polynomials over a triangular decomposition of T. Algorithm 14 will be used later only for the case of input monic polynomials whereas Algorithm 13 gives a convenient subroutine to be reused in other algorithms.

Algorithm 13 Refining Project

> **Input** $((F^i, C^i)_{1 \leq i \leq r}, U)$ where F^i is a set of polynomials $\in \mathbb{K}(C^i)[y]$, T is a triangular set, and both $\{C^1, \ldots, C^r\}$ and U are non-critical triangular decompositions of T, such that the latter one refines the former one.

Output all $\text{project}(F^i, \{C^i\}, \text{decomp}(C^i, U))$ for $1 \leq i \leq r$.

refineProject$((F^i, C^i)_{1 \leq i \leq r}, U) ==$
1: $result \leftarrow \{\}$
2: **for** i from 1 to r **do**
3: $\quad result \leftarrow result \cup \text{project}(F^i, \{C^i\}, \text{decomp}(C^i, U))$
4: **end for**
5: **return** $result$

Algorithm 14 Fast Simultaneous Remainder Modulo a Triangular Set

    Input T is a triangular set, and $p, a_1, \ldots, a_e$ are polynomials in $\mathbb{K}(T)[y]$. Let $d = \sum deg(a_i)$.

    Output $((R^1, W^1), \ldots, (R^l, W^l))$ where $\{W^1, \ldots, W^l\}$ is a non-critical triangular decomposition of T. $R^i = (r^i_j)$ where r^i_j is the remainder of p by a_i in $\mathbb{K}(W^i)[y]$.

multiRemModT$(p, (a_1, \ldots, a_e), T) ==$

  1: $tree \leftarrow$ subProductTree$((a_1, \ldots, a_e), T)$; $result \leftarrow \{\}$
  2: $f \leftarrow$ root of $tree$; $W \leftarrow \{\}$
  3: **if** $deg(p) \geq deg(f)$ **then**
  4:     $(r_i, T^i)_{1 \leq i \leq s} \leftarrow$ mdiv(p, f, T)
  5:     Label f by $(r^i, T^i)_{1 \leq i \leq s}$; $W \leftarrow \{T^1, \ldots, T^s\}$
  6: **else**
  7:     Label f by (p, T); $W \leftarrow \{T\}$
  8: **end if**
  9: **for** every node N in $tree$ starting from its root top-down and left-right **do**
 10:    **if** N is not a leaf **then**
 11:       $((r_i, U^i)_{1 \leq i \leq \#U}) \leftarrow$ label of N
 12:       $f_1 \leftarrow$ leftChild(N); $f_2 \leftarrow$ rightChild(N)
 13:       **for** f in $\{f_1, f_2\}$ **do**
 14:          $((r'_j, W^j)_{1 \leq j \leq \#W}) \leftarrow$ refineProject$((r_i, U^i)_{1 \leq i \leq \#U}, W)$
 15:          $((f'_j, W^j)_{1 \leq j \leq \#W}) \leftarrow$ project$(f, \{T\}, W)$
 16:          $((rem_k, E^k)_{1 \leq k \leq \#E}) \leftarrow$ mdiv$((r'_j, f'_j, W^j)_{1 \leq j \leq \#W})$
 17:          Label f by $(rem_k, E^k)_{1 \leq k \leq \#E}$
 18:          $W \leftarrow \{E^1, \ldots, E^{\#E}\}$
 19:       **end for**
 20:    **end if**
 21: **end for**
 22: $W \leftarrow$ RemoveCriticalPair(U_f)
 23: **for** every node N in leaves **do**
 24:    $((r'_i, U^i)_{1 \leq i \leq \#U}) \leftarrow$ label of N
 25:    $((r_j, W^j)_{1 \leq j \leq \#W}) \leftarrow$ refineProject$((r'_i, U^i)_{1 \leq i \leq \#U}, W)$
 26:    $result \leftarrow result \cup (r_j, W^j)_{1 \leq j \leq \#W}$
 27: **end for**
 28: $result \leftarrow$ group the result into $(((r^i_1, \ldots, r^i_e), W^i)_{1 \leq i \leq \#W})$
 29: **return** $result$

3.6.3 Multiple GCD's

In this section, we present our first sub-algorithm involved in the computations of GCD-free bases. We start with the case of univariate polynomials over a field, leading to Algorithm 15. It takes as input p and $(a_1, \ldots, a_e)$ in $k[x]$, and outputs the sequence of all $\gcd(p, a_i)$. The idea of this algorithm is to first reduce p modulo all a_i using fast simultaneous reduction, and then take the GCD's of all remainders with the polynomials a_i (see also Exercise 11.4 in [57]).

We make the assumption that all a_i are non-constant in the pseudo-code below, so as to apply the results of Proposition 3.2.2. To cover the general case, it suffices to introduce a wrapper function, that strips the input sequence $(a_1, \ldots, a_e)$ from its constant entries, and produces 1 as corresponding GCD's; this function induces no additional arithmetic cost. Finally, we write $d = \sum_{i=1}^{e} \deg a_i$.

Algorithm 15 Multiple GCDs over a Field

$\mathsf{multiGcd}(p, (a_1, \ldots, a_e)) ==$

1: $tree \leftarrow \mathtt{subProductTree}((a_1, \ldots, a_e), T); \quad f \leftarrow \text{root of } tree$
2: **if** $\deg p \geq d$ **then**
3:     $p \leftarrow p \mod f$ $[\mathsf{M}(\deg p) + \mathsf{M}(d) \operatorname{logp}(d)]$
4: **end if**
5: $p_1 \leftarrow p \mod \text{leftChild}(f); \quad p_2 \leftarrow p \mod \text{rightChild}(f)$
6: Continue the operation from top to the leaves of $tree$, until
    $(q_1, \ldots, q_e) \leftarrow (p \mod a_1, \ldots, p \mod a_e)$ $[\mathsf{M}(d) \operatorname{logp}(d)]$
7: **return** $(\gcd(q_1, a_1), \ldots, \gcd(q_e, a_e))$ $\left[\sum_i \mathsf{M}(\deg a_i) \operatorname{logp}(\deg a_i)\right]$

The cost of lines 3 and 6 follows from Proposition 3.2.2. The function $d \mapsto \mathsf{M}(d) \operatorname{logp}(d)$ is super-additive, so the complexity at line 7 fits in $O(\mathsf{M}(d) \operatorname{logp}(d))$. Hence, the total cost of this algorithm is in $O(\mathsf{M}(\deg p) + \mathsf{M}(d) \operatorname{logp}(d))$.

Algorithm 17 is an adaptation of Algorithm 15 for the case of univariate polynomials over $\mathbb{K}(T)$, for some triangular set T. The principle is the same. However, each GCD computation can refine the current triangular decomposition of T. It is, therefore, preferable to define an operation dedicated to computing GCDs of several pairs of polynomials in $\mathbb{K}(T)[y]$. This is the purpose of Algorithm 16. The dominant cost in this latter algorithm comes from the GCD computations. Indeed projections and removal of critical pairs have lower cost. Therefore, the complexity analysis of Algorithm 17 is similar to that of Algorithm 15 and we have:

Proposition 3.6.5. *Algorithm 17 runs within*

$$O\big((\mathsf{M}(\deg p) + \mathsf{M}(d)\log(d))\,\mathsf{A}_n(T)\big)$$

operations in k where $d = \sum_{i \leq e} deg(a_i)$.

Algorithm 16 List of GCDs Modulo a Triangular Set

    Input T is a triangular set, $a_1, \ldots, a_e, b_1, \ldots, b_e$ are polynomials in $\mathbb{K}(T)[y]$.

  Output $((G^1, W^1), \ldots, (G^m, W^m))$ where $\{W^1, \ldots, W^m\}$ is a triangular decompositions of T. $G^i = (g_1^i, \ldots, g_e^i)$ where g_j^i is a GCD of a_j and b_j in $\mathbb{K}(W^i)[y]$.

$\mathsf{listGcdModT}((a_i, b_i)_{1 \leq i \leq e}, T) ==$

1: $((g_1^j, U^j)_{1 \leq j \leq s}) \leftarrow \mathsf{xgcd}(a_1, b_1, T)$
2: $temp \leftarrow ((g_1^i, U^i)_1 \leq i \leq s); \quad W \leftarrow \{U^1, \ldots, U^s\}$
3: **for** i in $2 \ldots e$ **do**
4:     $((a_i^j, W^j)_{1 \leq j \leq \#W}) \leftarrow \mathrm{project}(a_i, \{T\}, W)$
5:     $((b_i^j, W^j)_{1 \leq j \leq \#W}) \leftarrow \mathrm{project}(b_i, \{T\}, W)$
6:     $H \leftarrow \{\}$
7:     **for** j in $1 \ldots \#W$ **do**
8:         $((g_i^k, E^k)_{1 \leq k \leq t}) \leftarrow \mathsf{xgcd}(a_i^j, b_i^j, W^j)$
9:         $temp \leftarrow temp \cup ((g_i^k, E^k)_{1 \leq k \leq t})$
10:       $H \leftarrow H \cup \{E^1, \ldots, E^t\}$
11:     **end for**
12:     $W \leftarrow \mathsf{RemoveCriticalPair}(H)$
13: **end for**
14: $result \leftarrow \{\}$
15: **for** $(g^l, E^l)_{1 \leq l \leq \#E}$ in $temp$ **do**
16:     $((g'^m, W^m)_{1 \leq m \leq \#W}) \leftarrow \mathsf{refineProject}((g^l, E^l)_{1 \leq l \leq \#E}, W)$
17:     $result \leftarrow result \cup ((g'^m, W^m)_{1 \leq m \leq \#W})$
18: **end for**
19: Group $result$ as $((g_1^m, \ldots, g_e^m), W^m)_{1 \leq m \leq \#W}$ and **return** $result$

3.6.4   All Pairs of GCD's

The second sub-algorithm involved in the computations of GCD-free bases performs the following. On input, we take two families of polynomials $(a_1, \ldots, a_e)$ and $(b_1, \ldots, b_s)$, where all a_i (resp. all b_i) are squarefree and pairwise coprime. Algorithms 18 and 19 compute all $\gcd(a_i, b_j)$. Algorithm 18 covers the case of polynomials over the base field k and Algorithm 19 deals with polynomials in $\mathbb{K}(T)[y]$.

As above, we suppose that all a_is are non-constant; to handle the general case, it suffices to introduce a wrapper function, with arithmetic cost 0, that removes each

---

Algorithm 17 Multiple GCDs Modulo a Triangular Set

    Input T is a triangular set, and $p, a_1, \ldots, a_e$ are polynomials in $\mathbb{K}(T)[y]$ with $a_1, \ldots, a_e$ **monic**.

  Output $((G^1, W^1), \ldots, (G^s, W^s))$ where $\{W^1, \ldots, W^l\}$ is a triangular decompositions of T. $G^i = (g_1^i, \ldots, g_e^i)$ where g_j^i is a GCD of p and a_j in $\mathbb{K}(W^i)[y]$.

$\mathsf{multiGcdModT}(p, (a_1, \ldots, a_e), T) ==$
1: $(((r_1, \ldots, r_e), T)) \leftarrow \mathsf{multiRemModT}(p, (a_1, \ldots, a_e), T)$
2: **return** $\mathsf{listGcdModT}((r_j, a_j)_{1 \leq j \leq e}, T)$

---

constant a_i from the input, and adds the appropriate sequence $(1, \ldots, 1)$ in the output. Here, we write $d = \max(\sum_i \deg a_i, \sum_j \deg b_j)$.

Algorithm 18 computes the GCD's of $(b_1, \ldots, b_s)$ with all polynomials in the subproduct tree associated with $(a_1, \ldots, a_e)$; the requested output can be found at the leaves of the tree. To give the complexity of this algorithm, one proves that the total number of operations along each row is in $O(\mathsf{M}(d)\,\mathrm{logp}(d))$, whence a total cost in $O(\mathsf{M}(d)\,\mathrm{logp}(d)^2)$.

Algorithm 19 follows the same strategy as Algorithm 18. However, each call to `listGcdModT` can further split the current triangular decomposition W of T, leading to projection and removal of critical pairs. However, their costs are dominated by the calls to `listGcdModT`. Therefore, we have:

Proposition 3.6.6. *Algorithm 19 runs within*

$$O\big(\,\mathsf{M}(d)\,\mathrm{logp}(d)^2\,\mathsf{A}_n(T)\,\big)$$

operations in k where $d = \max(\sum_i \deg a_i, \sum_j \deg b_j)$.

3.6.5   Merging GCD-Free Bases

The input of our third subroutine are sequences of polynomials $(a_1, \ldots, a_e)$ and $(b_1, \ldots, b_s)$, where all a_i (resp. all b_i) are squarefree, **monic** and pairwise coprime. We compute a GCD-free basis of $(a_1, \ldots, a_e, b_1, \ldots, b_s)$. Algorithm 20 deals with the case of univariate polynomials over the base field k whereas Algorithm 23 covers the case of univariate polynomials over $\mathbb{K}(T)$, for some triangular set T. This is done by computing all $\gcd(a_i, b_j)$, as well as the quotients $\delta_i = a_i / \prod_j \gcd(a_i, b_j)$ and $\gamma_j = b_j / \prod_i \gcd(a_i, b_j)$.

Algorithm 18 All Pairs of GCDs over a Field

$\mathsf{pairsOfGcd}((a_1, \ldots, a_e), (b_1, \ \ldots, \ b_s)) ==$

1:   $tree \leftarrow \mathsf{subProductTree}(a_1, \ldots, a_e); \ \ f = \mathrm{rootOf}(tree)$ $[\mathsf{M}(d)\log\mathrm{p}(d)]$

2:   Label the root of $tree$ by $\mathsf{multiGcd}(f, (b_1, \ldots, b_s))$ $[\mathsf{M}(d)\log\mathrm{p}(d)]$

3: **for** every node $N \in tree$, going top-down **do**

4:    **if** N is not a leaf and has label **g then**

5:      $f_1 \leftarrow \mathrm{leftChild}(N); \ \ f_2 \leftarrow \mathrm{rightChild}(N)$

6:      Label f_1 by $\mathsf{multiGcd}(f_1, \mathbf{g})$ $[\mathsf{M}(d)\log\mathrm{p}(d)^2]$

7:      Label f_2 by $\mathsf{multiGcd}(f_2, \mathbf{g})$ $[\mathsf{M}(d)\log\mathrm{p}(d)^2]$

8:    **end if**

9: **end for**

10: **return** labels of the leaves

The fact that the polynomials $a_1, \ldots, a_e, b_1, \ldots, b_s$ and their GCD's are monic makes Algorithm 23 relatively simple. For clarity, it uses two sub-procedures Algorithm 21 and Algorithm 22 for simplicity.

We denote by $\mathrm{removeConstants}(L)$ a subroutine that removes all constant polynomials from a sequence L (such a function requires no arithmetic operation, so its cost is zero in our model). In the complexity analysis, we still write $d = \max(\sum_i \deg a_i, \sum_j \deg b_j)$.

The validity of Algorithm 20 is easily checked. The estimates for the cost of lines 4, 5, 9 and 10 come for the cost necessary to build a subproduct tree and perform Euclidean division, together with the fact that β_j (resp. α_i) divides b_j (resp. a_i). The total cost is thus in $O(\mathsf{M}(d)\log\mathrm{p}(d)^2)$. Similarly, Algorithm 23 has total cost $O\big(\mathsf{M}(d)\log\mathrm{p}(d)^2\,\mathsf{A}_n(T)\big)$.

3.6.6   Computing GCD-Free Bases

We finally give an algorithm for computing GCD-free bases. As input, we take square-free, non-constant polynomials $a_1, \ldots, a_e$, with $d = \sum_{i \leq e} \deg a_i$. We need a construction close to the subproduct tree: we form a subproduct tree $tree$ whose nodes will be labeled by sequences of polynomials. Initially the leaves contain the sequences of length 1, $(a_1), \ldots, (a_e)$, and all other nodes are empty. Then, we go up the tree; at a node N, we use the subroutines of Section 3.6.5 in order to compute a GCD-free basis of the sequences labeling the children of N.

Algorithm 24 deals with the case of univariate polynomials over the base field k whereas Algorithm 25 covers the case of univariate polynomials over $\mathbb{K}(T)$, for some triangular set T.

Algorithm 19 All Pairs GCDs Modulo a Triangular Set

  Input T is a triangular set, and $a_1, \ldots, a_e, b_1, \ldots, b_s$ are polynomials in $\mathbb{K}(T)[y]$.

  Output $((G^k, W^k)_{1 \leq k \leq r})$ where $\{W^1, \ldots, W^r\}$ is a triangular decomposition of T. $G^k = (g_{a_i, b_j}{}^k)_{1 \leq i \leq e, 1 \leq j \leq s}$ is a GCD of a_i and b_j over $\mathbb{K}(W^k)[y]$.

allPairsGcdsModT$((a_1, \ldots, a_e), (b_1, \ldots, b_s), T) ==$

1:   $result \leftarrow \{\}; \;\; W \leftarrow \{T\}$
2:   $tree \leftarrow$ subProductTree$(a_1, \ldots, a_e); \;\; f \leftarrow$ rootOf$(tree)$
3:   $(((g^i_{f,1}, \ldots, g^i_{f,s}), U^i)_{1 \leq i \leq t}) \leftarrow$ multiGcdModT$(f, (b_1, \ldots, b_s), T)$
4:   Label the root of $tree$ by $(((g^i_{f,1}, \ldots, g^i_{f,s}), U^i)_{1 \leq i \leq t})$
5:   **for** every node N in $tree$ starting from its root top-down and left-right **do**
6:     **if** N is not a leaf **then**
7:       $(((g^i_{N,b_1}, \ldots, g^i_{N,b_s}), U^i)_{1 \leq i \leq t}) \leftarrow$ label of N
8:       $f_1 \leftarrow$ leftChild$(N); \;\; f_2 \leftarrow$ rightChild(N)
9:       **for** f in $\{f_1, f_2\}$ **do**
10:         $((f^j, W^j)_{1 \leq j \leq \#W}) \leftarrow$ project(f, T, W)
11:         $(((g^j_{N,b_1}, \ldots, g^j_{N,b_s}), W^j)_{1 \leq j \leq \#W}) \leftarrow$
            project$((((g^i_{N,b_1}, \ldots, g^i_{N,b_s}), U^i)_{1 \leq i \leq t}), W)$
12:         $U_f \leftarrow \{\}; \;\; result_f \leftarrow \{\}$
13:         **for** j in $1 \ldots \#W$ **do**
14:           $(((g^k_{N,b_1}, \ldots, g^k_{N,b_s}), E^k)_{1 \leq k \leq \#E}) \leftarrow$
            multiGcdModT$(f^j, (g^j_{N,b_1}, \ldots, g^j_{N,b_s}), W^j)$
15:           $result_f \leftarrow result_f \cup (((g^k_{N,b_1}, \ldots, g^k_{N,b_s}), E^k)_{1 \leq k \leq \#E})$
16:           $U_f \leftarrow U_f \cup \{E^1, \ldots, E^k\}$
17:         **end for**
18:         Label f by $result_f; \;\; W \leftarrow$ RemoveCriticalPair(U_f)
19:       **end for**
20:     **end if**
21:   **end for**
22:   **for** every node N in leaves of $tree$ **do**
23:     $(((g^k_{N,b_1}, \ldots, g^k_{N,b_s}), E^k)_{1 \leq k \leq \#E}) \leftarrow$ label of N
24:     $(((g^r_{N,b_1}, \ldots, g^r_{N,b_s}), W^r)_{1 \leq r \leq \#W}) \leftarrow$
      refineProject$((((g^k_{N,b_1}, \ldots, g^k_{N,b_s}), E^k)_{1 \leq k \leq \#E}, W)$
25:     $result \leftarrow result \cup (((g^r_{N,b_1}, \ldots, g^r_{N,b_s}), W^r)_{1 \leq r \leq \#W})$
26:   **end for**
27:   $result \leftarrow$ group the result into $((g_{a_i, b_j}{}^r{}_{1 \leq i \leq e, 1 \leq j \leq s}, W^r)_{1 \leq r \leq \#W}$
28:   **return** $result$

Algorithm 20 Merge GCD-Free Bases over a Field

---

mergeGCDFreeBases$((a_1, \ldots, a_e), (b_1, \ldots, b_s)) ==$

1: $(g_{i,j})_{1 \leq i \leq e, 1 \leq j \leq s} := \mathsf{pairsOfGcd}((a_1, \ldots, a_e), (b_1, \ldots, b_s))$ $\qquad [\mathsf{M}(d) \log \mathsf{p}(d)^2]$
2: **for** i in $1 \ldots e$ **do**
3: $\quad L_i \leftarrow \mathrm{removeConstants}(g_{i,1}, \ldots, g_{i,s})$
4: $\quad \alpha_i \leftarrow \prod_{\ell \in L_i} \ell$ $\qquad\qquad [\mathsf{M}(d) \log \mathsf{p}(d)]$
5: $\quad \delta_i \leftarrow a_i$ quo α_i $\qquad\qquad [\mathsf{M}(d)]$
6: **end for**
7: **for** j in $1 \ldots s$ **do**
8: $\quad L_j \leftarrow \mathrm{removeConstants}(g_{1,j}, \ldots, g_{e,j})$
9: $\quad \beta_j \leftarrow \prod_{\ell \in L_j} \ell$ $\qquad\qquad [\mathsf{M}(d) \log \mathsf{p}(d)]$
10: $\quad \gamma_j \leftarrow b_j$ quo β_j $\qquad\qquad [\mathsf{M}(d)]$
11: **end for**
12: **return** $\mathrm{removeConstants}(g_{1,1}, \ldots, g_{i,j}, \ldots, g_{e,s}, \gamma_1, \ldots, \gamma_s, \delta_1, \ldots, \delta_e)$

Algorithm 21 Coprime Factors Modulo a Triangular Set

Input a triangular set T, two sets $A = (a_1, \ldots, a_e)$, $B = (b_1, \ldots, b_s)$ each of which is contained in $\mathbb{K}(T)[y]$ and is a GCD-free basis of itself, together with $G = \{g_{i,j} | 1 \leq i \leq e, 1 \leq j \leq s\}$ where $g_{i,j}$ is a monic GCD of a_i and b_j.

Output (C, T) where C is a GCD-free basis of $A \cup B$.

coprimeFactorsModT$((a_1, \ldots, a_e), (b_1, \ldots, b_s), (g_{i,j})_{1 \leq i \leq e, 1 \leq j \leq s}, T)$

1: **for** i in $1 \ldots e$ **do**
2: $\quad L_i \leftarrow \mathrm{removeConstants}(g_{i,1}, \ldots, g_{i,s})$
3: $\quad \alpha_i \leftarrow \prod_{\ell \in L_i} \ell$
4: $\quad \delta_i \leftarrow a_i$ quo α_i
5: **end for**
6: **for** j in $1 \ldots s$ **do**
7: $\quad L_j \leftarrow \mathrm{removeConstants}(g_{1,j}, \ldots, g_{e,j})$
8: $\quad \beta_j \leftarrow \prod_{\ell \in L_j} \ell$
9: $\quad \gamma_j \leftarrow b_j$ quo β_j
10: **end for**
11: $C \leftarrow \mathrm{removeConstants}(g_{1,1}, \ldots, g_{i,j}, \ldots, g_{e,s}, \gamma_1, \ldots, \gamma_s, \delta_1, \ldots, \delta_e)$
12: **return** (C, T)

Algorithm 22 Merge Two GCD-Free Bases Modulo a Triangular Set

    Input T a triangular set and two sets $A = (a_1, \ldots, a_e)$, $B = (b_1, \ldots, b_s)$ each of which is contained in $\mathbb{K}(T)[y]$ and is a GCD-free basis of itself.

    Output $(F^\ell, W^\ell)_{1 \leq \ell \leq r}$ where $W = \{W^1, \ldots, W^r\}$ is a non-critical triangular decomposition of T and, for all $1 \leq \ell \leq r$, F^ℓ is a GCD-free basis of $A \cup B$ modulo W^ℓ.

$\mathsf{mergeTwoGcdFreeBasesModT}((a_1, \ldots, a_e), (b_1, \ldots, b_s), T) ==$

1: $bases \leftarrow \{\}$
2: $(((g_{a_i, b_j}{}^k)_{1 \leq i \leq e, 1 \leq j \leq s}, W^k)_{1 \leq k \leq r}) \leftarrow \mathsf{allPairsGCDsModT}((a_1, \ldots, a_e), (b_1, \ldots, b_s), T)$
3: $((a_1^k, \ldots, a_e^k), W^k)_{1 \leq k \leq r} \leftarrow \mathrm{project}((a_1, \ldots, a_e), \{T\}, \{W^1, \ldots, W^r\})$
4: $((b_1^k, \ldots, b_s^k), W^k)_{1 \leq k \leq r} \leftarrow \mathrm{project}((b_1, \ldots, b_s), \{T\}, \{W^1, \ldots, W^r\})$
5: **for** k in $1 \ldots r$ **do**
6:    $((b_u^k)_{1 \leq u \leq \#b^k}, W^k)_{1 \leq k \leq r} \leftarrow$
   $\mathsf{coprimeFactorsModT}((a_1, \ldots, a_e), (b_1, \ldots, b_s), ((g_{a_i, b_j}{}^m)_{1 \leq i \leq e, 1 \leq j \leq s}, W^k)$
7:    $bases \leftarrow bases \cup ((b_u^k)_{1 \leq u \leq \#b^k}, W^k)_{1 \leq k \leq r}$
8: **end for**
9: **return** $bases$

Algorithm 23 Merge GCD-Free Bases Modulo a Triangular Set

    Input T a triangular set, $((A^i, B^i, U^i)_{1 \leq i \leq t})$ a sequence where $U = \{U^1, \ldots, U^t\}$ is a non-critical triangular decomposition of T and where, for $1 \leq i \leq t$ each of A^i and B^i is a finite set of non-constant, monic, squarefree and pairwise coprime polynomials of $\mathbb{K}(U^i)[y]$.

    Output $(F^k, W^k)_{1 \leq k \leq r}$ where $W = \{W^1, \ldots, W^r\}$ is a non-critical triangular decomposition of T refining U and for all $1 \leq k \leq r$, the finite set $F^k \subset \mathbb{K}(W^k)[y]$ is a GCD-free basis of $A^i \cup B^i$ modulo W^k where $W^k \in \mathrm{decomp}(U^i, W)$, which determines i uniquely.

$\mathsf{mergeGcdFreeBasesModT}((A^i, B^i, U^i)_{1 \leq i \leq t}) ==$

1: $U \leftarrow \{\}; \;\; temp \leftarrow \{\}; \;\; bases \leftarrow \{\}$
2: **for** i in $1 \ldots t$ **do**
3:    $(((b_u^k)_{1 \leq u \leq \#b^k}, E^k)_{1 \leq k \leq \#E}, E) \leftarrow \mathsf{mergeTwoGcdFreeBasesModT}(A^i, B^i, U^i)$
4:    $temp \leftarrow temp \cup ((b_u^k)_{1 \leq u \leq \#b^k}, E^k)_{1 \leq k \leq \#E}$
5:    $U \leftarrow U \cup \{E^1, \ldots, E^{\#E}\}$
6: **end for**
7: $W \leftarrow \mathsf{RemoveCriticalPair}(U)$
8: **for** $((b_u^k)_{1 \leq u \leq \#b^k}, E^k)_{1 \leq k \leq \#E}$ in $temp$ **do**
9:    $bases \leftarrow bases \cup \mathsf{refineProject}(((b_u^k)_{1 \leq u \leq \#b^k}, E^k)_{1 \leq k \leq \#E}, W)$
10: **end for**
11: **return** $(bases)$

Algorithm 24 GCD-Free Bases over a Field

---

gcdFreeBasis($\{a_1, \ldots, a_e\}$) ==

1: $tree \leftarrow$ subProductTree($a_1, \ldots, a_e$)
2: **for** every node $N \in tree$ **and** from bottom-up **do**
3:    **if** N is not a leaf **then**
4:       $f_1 \leftarrow$ leftChild(N); $f_2 \leftarrow$ rightChild(N)
5:       Label N by mergeGcdFreeBases(f_1, f_2) $[\mathsf{M}(d) \operatorname{logp}(d)^3]$
6:    **end if**
7: **end for**
8: **return** label of rootOf($tree$)

Consider first the case of Algorithm 24. The total number of operations at a node N of the subset tree is $O(\mathsf{M}(d_N) \operatorname{logp}(d_N)^2)$, where d_N is sum of the degrees of the polynomials lying at the two children of N. Summing over all nodes, using the tree structure, the total cost is seen to be in $O(\mathsf{M}(d) \operatorname{logp}(d)^3)$ operations, as claimed. Algorithm 25 being essentially the same as With Algorithm 24, we deduce Theorem 3.6.4.

Algorithm 25 GCD-Free Basis Modulo a Triangular Set

   Input $(a_1, \ldots, a_e) \in \mathbb{K}(T)[y]$ and T is a triangular set.

  Output $(F^k, W^k)_{1 \leq k \leq r}$ where $\{W^1, \ldots, W^r\}$ is a triangular decomposition of T. $F^k = (f_j^k)_{1 \leq j \leq \#F^k}$ is a GCD-free basis of $(a_1, \ldots, a_e)$ modulo W^k.

gcdFreeBasisModT($(a_1, \ldots, a_e), T$) ==

1: $tree \leftarrow$ subProductTree($(a_1, \ldots, a_e), T$); $W \leftarrow \{T\}$
2: **for** every node $N \in tree$ at each level left-right and bottom-up **do**
3:    **if** N is not a leaf **then**
4:       $((F^i, C^i)_{1 \leq i \leq r}) \leftarrow$ Label of leftChild(N)
5:       $((G^i, D^i)_{1 \leq i \leq s}) \leftarrow$ Label of rightChild(N)
6:       $((A^i, W^i)_{1 \leq i \leq t}) \leftarrow$ refineProject($(F^i, C^i)_{1 \leq i \leq r}, W$)
7:       $((B^i, W^i)_{1 \leq i \leq t}) \leftarrow$ refineProject($(G^i, D^i)_{1 \leq i \leq s}, W$)
8:       $((H^i, E^i)_{1 \leq i \leq u}) \leftarrow$
      mergeGcdFreeBasesModT($(A^i, B^i, W^i)_{1 \leq i \leq t}$)
9:       Label of $N \leftarrow ((H^i, E^i)_{1 \leq i \leq u})$
10:      $W \leftarrow \{E^1, \ldots, E^u\}$
11:    **end if**
12: **end for**
13: **return** Label of rootOf($tree$)

3.7 Removing Critical Pairs

We next show how to remove critical pairs. This is an inductive process, whose complexity is estimated in the following proposition and its corollary. We need to extend the notion of "refining" introduced previously. Extending Definition 3.1.3, we say that a family of triangular sets $\mathbf{T}'$ refines another family $\mathbf{T}$ if for every $T \in \mathbf{T}$, there exists a subset of $\mathbf{T}'$ that forms a triangular decomposition of $\langle T \rangle$. Note the difference with the initial definition: we do not impose that the family $\mathbf{T}$ forms a triangular decomposition of some ideal I. In particular, the triangular sets in $\mathbf{T}$ do not have to generate coprime ideals.

Proposition 3.7.1. *We have a constant K such that the following holds. Let $\mathsf{A}_1, \ldots, \mathsf{A}_{n-1}$ be arithmetic times for triangular sets in $1, \ldots, n-1$ variables.*

Let T be a triangular set in n variables, and let $\mathbf{U}$ be a triangular decomposition of $\langle T \rangle$. Then for all $j = 1, \ldots, n$, the following holds: given $\mathbf{U}_{\leq j}$, one can compute a non-critical triangular decomposition $\mathbf{W}$ of $T_{\leq j}$ that refines $\mathbf{U}_{\leq j}$ using a_j operations in k, where a_j satisfies the recurrence inequalities, $a_0 = 0$ and for $j = 0, \ldots, n-1$,

$$a_{j+1} \leq 2a_j + K\mathsf{M}(d_{j+1} \cdots d_n)\,\mathrm{logp}(d_{j+1} \cdots d_n)^3 \mathsf{A}_j(T_{\leq j}),$$

and where $d_j = \deg_j T$ for $j = 1, \ldots, n$.

Before discussing the proof of this assertion, let us give an immediate corollary, which follows by a direct induction.

Corollary 3.7.2. *Given a triangular decomposition $\mathbf{U}$ of $\langle T \rangle$, one can compute a non-critical triangular decomposition $\mathbf{W}$ of $\langle T \rangle$ that refines $\mathbf{U}$ in time*

$$K\left(2^{n-1}\mathsf{M}(d_1 \cdots d_n)\,\mathrm{logp}(d_1 \cdots d_n)^3 + \cdots + \mathsf{M}(d_n)\,\mathrm{logp}(d_n)^3 \mathsf{A}_{n-1}(T_{\leq n-1})\right).$$

PROOF. We only sketch the proof of the proposition. Let thus j be in $0, \ldots, n-1$ and let $\mathbf{U} = U^1, \ldots, U^e$ be a triangular decomposition of $\langle T \rangle$; we aim at removing the critical pairs in $\mathbf{U}_{\leq j+1}$. Let $\mathbf{V}$ be obtained by removing the critical pairs in $\mathbf{U}_{\leq j}$. Thus, $\mathbf{V}$ consists in triangular sets in $k[X_1, \ldots, X_j]$, and has no critical pair.

Let us fix $i \leq e$, and write $U^i = (U_1^i, \ldots, U_n^i)$. By definition, there exists a subset $\mathbf{V}_i = V^{i,1}, \ldots, V^{i,e_i}$ of $\mathbf{V}$ which forms a non-critical decomposition of $(U_1^i, \ldots, U_j^i)$. Our next step is to compute

$$U_{j+1}^{i,1}, \ldots, U_{j+1}^{i,e_i} = \mathrm{project}(U_{j+1}^i, (U_1^i, \ldots, U_j^i), \mathbf{V}_i).$$

Consider now a triangular set V in $\mathbf{V}$. There may be several subsets $\mathbf{V}_i$ such that $V \in \mathbf{V}_i$. Let $S_V \subset \{1, \ldots, e\}$ be the set of corresponding indices; thus, for any $i \in S_V$, there exists $\ell(i)$ in $1, \ldots, e_i$ such that $V = V^{i,e_{\ell(i)}}$. We will then compute a coprime factorization of all polynomials $U_{j+1}^{i,e_{\ell(i)}}$ in $\mathbb{K}(V)[X_{j+1}]$, for $i \in S_V$, and for all V.

This process will refine the family $\mathbf{V}$, creating possibly new critical pairs: we get rid of these critical pairs, obtaining a decomposition $\mathbf{W}$. It finally suffices to project all polynomials in the coprime factorization obtained before from $\mathbf{V}$ to $\mathbf{W}$ to conclude. The cost estimates then takes into account the cost for the two calls to the same process in j variables, hence the term $2a_j$, and the cost for coprime factorization and projecting. Studying the degrees of the polynomials involved, this cost can be bounded by

$$K\mathsf{M}(d_{j+1} \cdots d_n) \operatorname{logp}(d_{j+1} \cdots d_n)^3 \mathsf{A}_j(T_{\leq j})$$

for some constant K, according to the results in the last section. $\qquad\square$

3.8   Concluding the Proof

All ingredients are now present to give the proof of the following result, which readily implies the main theorems stated in the introduction.

Theorem 3.8.1. *We have a constant L such that, writing*

$$\mathsf{A}_n(d_1, \ldots, d_n) = L^n \prod_{i \leq n} \mathsf{M}(d_i) \operatorname{logp}(d_i)^3,$$

the function $T \mapsto \mathsf{A}_n(\deg_1 T, \ldots, \deg_n T)$ is an arithmetic time for triangular sets in n variables, for all n.

PROOF.   The proof requires to check that taking L big enough, all conditions defining arithmetic times are satisfied. We do it by induction on n; the case $n = 1$ is settled by Proposition 3.2.4, taking L larger than the constant C in that proposition, and using the fact that $\operatorname{logp}(x) \geq 1$ for all x.

Let us now consider index n; we can thus assume that the function A_j is an arithmetic time for triangular sets in j variables, for $j = 1, \ldots, n-1$. Then, at index n, condition (E_0) makes no difficulty, using the super-additivity of the function M. Addition and multiplication (condition (E_1)) and projecting (condition (E_4)) follow from Proposition 3.3.1, again as soon as the condition $L \geq C$ holds. The computation of quasi-inverses (condition (E_2)) is taken care of by Proposition 3.5.1,

---

Algorithm 26 Remove Critical Pair

    Input $[U^1, \ldots, U^e]$: a triangular decomposition of a triangular set $\langle T \rangle$, where $U^i = \{U^i_1, \ldots, U^i_n\}$.

  Output a non-critical decomposition of T.

RemoveCriticalPair$(U^1, \ldots, U^e) ==$

1: **if** $n \geq 1$ **then**

2:    $((b^1_1, \ldots, b^1_{\#b^1}), \{\}) \leftarrow$ gcdFreeBasisModT$(\{U^1_1, \ldots, U^e_1\}, \{\})$

3:    $V^1 \leftarrow \{V^1_i \mid V^1_i = \{b^1_i\}, 1 \leq i \leq \#b^1\}$

4:    **if** $n = 1$ **then return** V^1

5:    **for** j from 1 to $n-1$ **do**

6:      **for** i from 1 to e **do**

7:        $V_i = \{V^{i,k}, 1 \leq k \leq \#V_i\} \leftarrow \{V \in V^j \mid V \in U^i_{<j+1}\}$

8:        $\{U^{i,1}_{j+1}, \ldots, U^{i,\#V_i}_{j+1}\} \leftarrow$ project$(U^i_{j+1}, \{U^i_1, \ldots, U^i_j\}, V_i)$

9:      **end for**

10:    $R^{V^j} \leftarrow [\ ]; \quad b^{V^j} \leftarrow [\ ]$

11:    **for** s from 1 to $\#V^j$ **do**

12:      $p^{V^j_s} \leftarrow \{U^{i,k}_{j+1} \mid V^{i,k} = V^j_s \text{ for } V^{i,k} \in V_i, 1 \leq i \leq e, 1 \leq k \leq \#V_i\}$

13:      $((f^{V^j_s,m}_l)_{1 \leq l \leq \#(f^{V^j_s,m})}, R^{V^j_s,m})_{1 \leq m \leq \#R^{V^j_s}} \leftarrow$ gcdFreeBasisModT$(p^{V^j_s}, V^j_s)$

14:      $R^{V^j} \leftarrow R^{V^j} \cup \{R^{V^j_s,1}, \ldots, R^{V^j_s,\#R^{V^j_s}}\}$

15:    **end for**

16:    $\{W^{V^j}_1, \ldots, W^{V^j}_{\#W^{V^j}}\} \leftarrow$ RemoveCriticalPair(R^{V^j})

17:    $V^{j+1} \leftarrow [\ ]$

18:    **for** s from 1 to $\#V^j$ **do**

19:      $W^{V^j_s} \leftarrow \{W^{V^j}_k \mid W^{V^j}_k \in V^j_s, 1 \leq k \leq \#W^{V^j}\}$

20:      $((b^{V^j_s,r}_l)_{1 \leq l \leq \#(b^{V^j_s,r})}, W^{V^j_s,r})_{1 \leq r \leq \#W^{V^j_s}} \leftarrow$
       refineProject$(((f^{V^j_s,m}_l)_{1 \leq l \leq \#(f^{V^j_s,m})}, R^{V^j_s,m})_{1 \leq m \leq \#R^{V^j_s}}, W^{V^j_s})$

21:      $V^{j+1} \leftarrow V^{j+1} \cup \{W^{V^j_s,r}, b^{V^j_s,r}_l\}_{1 \leq l \leq \#(b^{V^j_s,r}), 1 \leq r \leq \#W^{V^j_s}}$

22:    **end for**

23:    **end for**

24: **end if**

25: **return** V^n

---

using our induction assumption on A, as soon as L is large enough to compensate the constant factor hidden in the $O(\ )$ estimate of that proposition.

The cost for removing critical pairs is given in the previous section. In view of Corollary 3.7.2, and using the condition $\mathsf{M}(dd') \leq \mathsf{M}(d)\mathsf{M}(d')$, after a few simplifications, to satisfy condition (E_3), L must satisfy the inequality

$$K(2^{n-1} + 2^{n-2}L + \cdots + L^{n-1}) \leq L^n,$$

where K is the constant introduced in Corollary 3.7.2. This is the case for $L \geq K + 2$. $\square$

Chapter 4

A Modular Method for Triangular Decomposition

This chapter presents a modular method for solving polynomial systems of zero-dimensional varieties by means of triangular decompositions. Among all possible triangular decompositions, we introduce a canonical one, and call it the equiprojectable decomposition. We show that it has good computational properties. This allows us to develop a sharp modular method for triangular decomposition based on the Hensel lifting techniques.

4.1   Introduction

Modular methods for computing polynomial GCDs and solving linear algebra problems have been well-developed for several decades, see [57] and the references therein. Without these methods, the range of problems accessible to symbolic computations would be dramatically limited. Such methods, in particular Hensel lifting, also apply to solving polynomial systems. Standard applications are the resolution of systems over $\mathbb{Q}$ after specialization at a prime, and over the rational function field $k(Y_1, \ldots, Y_m)$ after specialization at a point $(y_1, \ldots, y_m)$. These methods have already been put to use for Gröbner bases [134, 5] and primitive element representations, starting from [63], and refined notably in [64].

  Triangular decompositions are well-suited to many practical problems: see some examples in [19, 55, 120]. In addition, these techniques are commonly used in differential algebra [20, 71]. Triangular decompositions of polynomial systems can be obtained by various algorithms [76, 87, 108] but none of them uses modular compu-

tations, restricting their practical efficiency. Our goal in this work is to discuss such techniques, extending the preliminary results of [120].

Let us introduce the notation used below. If k is a perfect field (e.g., $\mathbb{Q}$ or a finite field), a *triangular set* is a family $T_1(X_1), T_2(X_1, X_2), \ldots, T_n(X_1, \ldots, X_n)$ in $k[X_1, \ldots, X_n]$ which forms a reduced Gröbner basis for the lexicographic order $X_n > \cdots > X_1$ and generates a radical ideal (so T_i is monic in X_i). The notation $T^1, \ldots, T^s$ denotes a family of s triangular sets, with $T^i = T_1^i, \ldots, T_n^i$. Then, any 0-dimensional variety V can be represented by such a family, such that $I(V) = \cap_{i \leq s} \langle T^i \rangle$ holds, and where $\langle T^i \rangle$ and $\langle T^{i'} \rangle$ are coprime ideals for $i \neq i'$; we call it a *triangular decomposition* of V. This decomposition is not unique: the different possibilities are obtained by suitably recombining the triangular sets describing the irreducible components of V.

In this work, we consider 0-dimensional varieties defined over $\mathbb{Q}$. Let thus $F = F_1, \ldots, F_n$ be a polynomial system in $\mathbb{Z}[X_1, \ldots, X_n]$. Since we have in mind to apply Hensel lifting techniques, we will only consider the *simple roots* of F, that is, those where the Jacobian determinant J of F does not vanish. We write $Z(F)$ for this set of points; by the Jacobian criterion [51, Ch. 16], $Z(F)$ is finite, even though the whole zero-set of F, written $V(F)$, may have higher dimension.

Let us assume that we have at hand an oracle that, for any prime p, outputs a triangular decomposition of $Z(F \bmod p)$. Then for a prime p, a rough sketch of an Hensel lifting algorithm could be: (1) Compute a triangular decomposition $t^1, \ldots, t^s$ of $Z(F \bmod p)$, and (2) Lift these triangular sets over $\mathbb{Q}$. However, without more precautions, this algorithm may fail to produce a correct answer. Indeed, extra factorizations or recombinations can occur modulo p. Thus, we have no guarantee that there exist triangular sets $T^1, \ldots, T^s$ defined over $\mathbb{Q}$, that describe $Z(F)$, and with $t^1, \ldots, t^s$ as modular images. Furthermore, if we assume no control over the modular resolution process, there is little hope of obtaining a quantification of primes p of "bad" reduction.

Consider for instance the variety $V \subset \mathbb{C}^2$ defined by the polynomials sys $= \{326X_1 - 10X_2^6 + 51X_2^5 + 17X_2^4 + 306X_2^2 + 102X_2 + 34,\ X_2^7 + 6X_2^4 + 2X_2^3 + 12\}$. For the order $X_2 > X_1$, the only possible description of V by triangular sets with rational coefficients corresponds to its irreducible decomposition, that is, $T^1 : (\ X_1-1,\ X_2^3+6\ )$ and $T^2 : (\ X_1^2 + 2,\ X_2^2 + X_1\ )$. Now, the following triangular sets describe the zeros of (sys mod 7), which are not the reduction modulo 7 of T^1 and T^2;

$$
t^1 \left|
\begin{array}{l}
X_2^2 + 6X_2X_1^2 + 2X_2 + X_1 \\
X_1^3 + 6X_1^2 + 5X_1 + 2
\end{array}
\right.
\quad \text{and} \quad
t^2 \left|
\begin{array}{l}
X_2 + 6 \\
X_1 + 6
\end{array}
\right. ,
$$

A lifting algorithm should discard t^1 and t^2, and replace them by the better choice $t'^1 : (X_1+6, X_2^3+6)$ and $t'^2 : (X_1^2+2, X_2^2+X_1)$, which are the reduction of T^1 and T^2 modulo 7. In [120], this difficulty was bypassed by restricting to *equiprojectable* varieties, *i.e.* varieties defined by a single triangular set, where no such ambiguity occurs. However, as this example shows, this assumption discards simple cases. Our main concern is to relax this limitation, thus extending these techniques to handle *triangular decompositions*.

Our answer consists in using a canonical decomposition of a 0-dimensional variety V, its *equiprojectable decomposition*, described as follows. Consider the map $\pi : V \subset \mathbb{A}^n(\overline{k}) \to \mathbb{A}^{n-1}(\overline{k})$ that forgets the last coordinate. To x in V, we associate $N(x) = \#\pi^{-1}(\pi(x))$, that is, the number of points lying in the same π-fiber as x. Then, we split V into the disjoint union $V_1 \cup \cdots \cup V_d$, where for all $i = 1, \ldots, d$, V_i equals $N^{-1}(i)$, *i.e.*, the set of points $x \in V$ where $N(x) = i$. This splitting process is applied recursively to all $V_1, \ldots, V_d$, taking into account the fibers of the successive projections $\mathbb{A}^n(\overline{k}) \to \mathbb{A}^i(\overline{k})$, for $i = n - 1, \ldots, 1$. In the end, we obtain a family of pairwise disjoint, equiprojectable varieties, whose reunion equals V, which form the *equiprojectable decomposition* of V. As requested, each of them is representable by a triangular set with coefficients in the definition field of V.

Looking back at the example, both $Z(\mathtt{sys})$ and $Z(\mathtt{sys} \bmod 7)$ are described on the leftmost picture below (forgetting the actual coordinates of the points). Representing $Z(\mathtt{sys})$ by T^1 and T^2, as well as $Z(\mathtt{sys} \bmod 7)$ by t'^1 and t'^2 amounts to grouping the points as on the central picture; this is the equiprojectable decomposition. The rightmost picture shows the description of $Z(\mathtt{sys} \bmod 7)$ by t^1 and t^2.
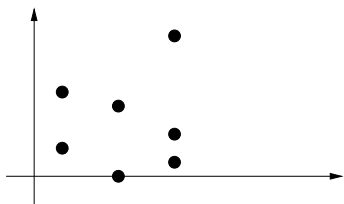
Figure 4.1: Description of both $Z(\mathtt{sys})$ and $Z(\mathtt{sys} \bmod 7)$
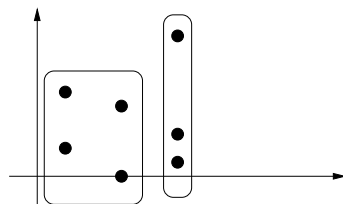
Figure 4.2: The Equiprojectable Decomposition: Representing $Z(\mathtt{sys})$ by T^1 and T^2, and Representing $Z(\mathtt{sys} \bmod 7)$ by t'^1 and t'^2
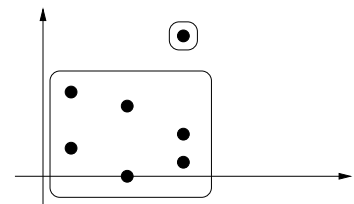
Figure 4.3: Description of $Z(\mathtt{sys} \bmod 7)$ by t^1 and t^2 for $\mathtt{sys}$

The above algorithm sketch is thus improved by applying lifting only after computing the equiprojectable decomposition of the modular output. Theorem 4.1.1 shows how to control the primes of bad reductions for the equiprojectable decomposition, thus overcoming the limitation that we pointed out previously. In Theorem 4.1.1 [39], the height of $x \in \mathbb{Z}$ is defined as $\mathbf{h}x = \log |x|$; the height of $f \in \mathbb{Z}[X_1, \ldots, X_n]$ is the maximum of the heights of its coefficients; that of $p/q \in \mathbb{Q}$, with $\gcd(p, q) = 1$, is $\max(\mathbf{h}p, \mathbf{h}q)$.

Theorem 4.1.1. *Let $F_1, \ldots, F_n$ have degree $\leq d$ and height $\leq h$. Let $T^1, \ldots, T^s$ be the triangular description of the equiprojectable decomposition of $Z(F)$. There exists $A \in \mathbb{N} - \{0\}$, with $\mathbf{h}A \leq \mathfrak{a}(n, d, h)$, and, for $n \geq 2$,*

$$\mathfrak{a}(n, d, h) = 2n^2 d^{2n+1}(3h + 7\log(n+1) + 5n\log d + 10),$$

and with the following property. If a prime p does not divide A, then p cancels none of the denominators of the coefficients of $T^1, \ldots, T^s$, and these triangular sets reduced mod p define the equiprojectable decomposition of $Z(F \bmod p)$.

Thus, the set of bad primes is finite and we have an explicit control on its size. Since we have to avoid some "discriminant locus", it is natural, and probably unavoidable, that the bound should involve the square of the Bézout number.

A second question is the coefficient size of the output. In what follows, we write $\deg V$ and $\mathbf{h}V$ for the *degree* and *height* of a 0-dimensional variety V defined over $\mathbb{Q}$: the former denotes its number of points, and the later estimates its arithmetic complexity; see [81] and references therein for its definition. Let then $T^1, \ldots, T^s$ be the triangular sets that describe the equiprojectable decomposition of $Z = Z(F)$. In [42], it is proved that all coefficients in $T^1, \ldots, T^s$ have height in $O(n^{O(1)}(\deg Z + \mathbf{h}Z)^2)$. However, better estimates are available, through the introduction of an alternative representation denoted by $N^1, \ldots, N^s$. For $i \leq s$, $N^i = N_1^i, \ldots, N_n^i$ is obtained as follows. Let $D_1^i = 1$ and $N_1^i = T_1^i$. For $2 \leq \ell \leq n$ and $1 \leq i \leq s$, define

$$D_\ell^i = \prod_{1 \leq j \leq \ell-1} \frac{\partial T_j^i}{\partial X_j} \quad \text{and} \quad N_\ell^i = D_\ell^i T_\ell^i \bmod (T_1^i, \ldots, T_{\ell-1}^i).$$

It is proved in [42] that all coefficients in $N^1, \ldots, N^s$ have height in $O(n^{O(1)}(\deg Z + \mathbf{h}Z))$. Since $T^1, \ldots, T^s$ are easily recovered from $N^1, \ldots, N^s$, our algorithm will compute the latter, their height bounds being the better.

Theorem 4.1.2 below states our main result regarding lifting techniques for trian-

gular decompositions; in what follows, we say that an algorithm has a *quasi-linear* complexity in terms of some parameters if its complexity is linear in all of these parameters, up to polylogarithmic factors. We need the following assumptions:

- For any $C \in \mathbb{N}$, let $\Gamma(C)$ be the sets of primes in $[C + 1, \ldots, 2C]$. We assume the existence of an oracle O_1 which, for any $C \in \mathbb{N}$, outputs a random prime in $\Gamma(C)$, with the uniform distribution.

- We assume the existence of an oracle O_2, which, given a system F and a prime p, outputs the representation of the equiprojectable decomposition of $Z(F \bmod p)$ by means of triangular sets. We give in Section 4.2.2 an algorithm to convert any triangular decomposition of $Z(F \bmod p)$ to the equiprojectable one; its complexity analysis is subject of current research.

- For F as in Theorem 4.1.1, we write $\mathfrak{a}_F = \mathfrak{a}(n, d, h)$, $\mathfrak{h}_F = nd^n(h + 11 \log(n+3))$ and $\mathfrak{b}_F = 5(\mathfrak{h}_F + 1)\log(2\mathfrak{h}_F + 1)$. The input system is given by a straight-line program of size L, with constants of height at most h_L.

- $\mathsf{C} \in \mathbb{N}$ is such that for any ring R, any $d \geq 1$ and monic $t \in R[X]$ of degree d, all operations $(+, -, \times)$ in $R[X]/t$ can be computed in $\mathsf{C}d \log d \log \log d$ operations in R [57, Ch. 8,9]. Then all operations $(+, -, \times)$ modulo a triangular set T in n variables can be done in quasi-linear complexity in C^n and $\deg V(T)$.

Theorem 4.1.2. *Let $\varepsilon > 0$. There exists an algorithm which, given F, satisfying*

$$\frac{4\mathfrak{a}_F + 2\mathfrak{b}_F}{\varepsilon} + 1 < \frac{1}{2}\exp\left(2\mathfrak{h}_F + 1\right),$$

computes $N^1, \ldots, N^s$ defined above. The algorithm uses two calls to O_1 with $C = 4\mathfrak{a}_F + 2\mathfrak{b}_F/\varepsilon$, two calls to O_2 with p in $[C+1, \ldots, 2C]$, and its bit complexity is quasi-linear in $L, h_L, d, \log h, \mathsf{C}^n, \deg Z, (\deg Z + \mathbf{h}Z), |\log \varepsilon|$. The algorithm is probabilistic, with success probability $\geq 1 - \varepsilon$.

To illustrate these estimates, suppose e.g. that we have $n = 10, d = 4, h = 100$, hence potentially 1048576 solutions; to ensure a success probability of 99%, the primes should have only about 20 decimal digits, hence can be generated without difficulty. Thus, even for such "large" systems, our results are quite manageable. Besides, computing the polynomials N^i instead of T^i enables us to benefit from their improved height bounds.

In the sequel, we use the following notation. For $n \in \mathbb{N}$, for $1 \leq j \leq i \leq n$ and any field k, we denote $\pi_j^i : \mathbb{A}^i(\overline{k}) \rightarrow \mathbb{A}^j(\overline{k})$ the map $(x_1, \ldots, x_i) \mapsto (x_1, \ldots, x_j)$. The cardinality of a finite set G is written $\#G$.

4.2 Equiprojectable Decomposition of Zero-dimensional Varieties

We start by introducing the notion of equiprojectable decomposition of a 0-dimensional variety V, reported in [38]. Then, in preparation for the modular algorithm of Section 4.2.2, we present an algorithm for computing this decomposition, given an arbitrary triangular decomposition of V. We call it *Split-and-Merge*, after its two phases: the *splitting* of what we call *critical pairs* (which is achieved by GCD computations) and the *merging* of what we call *solvable families* (which is performed by Chinese remaindering). The complexity analysis of the Split-and-Merge algorithm is work in progress [40]. From our preliminary study reported in [37], we believe that suitable improvements of the Split-and-Merge algorithm can run in quasi-linear time in the degree of V.

4.2.1 Notion of Equiprojectable Decomposition

Let k be a perfect field and $\overline{k}$ one of its algebraic closures. Following [10], we first define the notion of equiprojectability.

Equiprojectable variety. Let $V \subset \mathbb{A}^n(\overline{k})$ be a 0-dimensional variety over k. For $1 \leq i \leq n$, the variety V is *equiprojectable* on $\pi_i^n(V)$ if all fibers of the restriction $\pi_i^n : V \rightarrow \pi_i^n(V)$ have the same cardinality. Then, for $1 \leq i \leq n$, V is *i-equiprojectable* if it is equiprojectable on all $\pi_j^n(V)$, $i \leq j \leq n$. Thus, any 0-dimensional variety is n-equiprojectable. Finally, V is *equiprojectable* if it is 1-equiprojectable. It is the case if and only if its defining ideal is generated by a triangular set $T_1, \ldots, T_n$ with coefficients in k. In this case, k being perfect, all fibers of the projection $\pi_i^n(V) \rightarrow \pi_{i-1}^n(V)$ share the same cardinality, which is the degree of T_i in X_i.

The pictures in Figure 4.4 illustrate the definition of equiprojectable variety by an example. Given a zero-dimensional variety V in (x, y, z) consisting of 12 points, which are described in the leftmost picture. We use projection to check. Projecting to the (x, y) plane is shown in the middle picture. It results in 4 fibers. All of them have the same cardinality of 3. Then we project to the (x) axis, shown in the rightmost

picture. This projection has 2 fibers, and both have the same cardinality of 2. For this variety V, all the fibers in each projection have the same cardinality. Therefore, V is an equiprojectable variety.
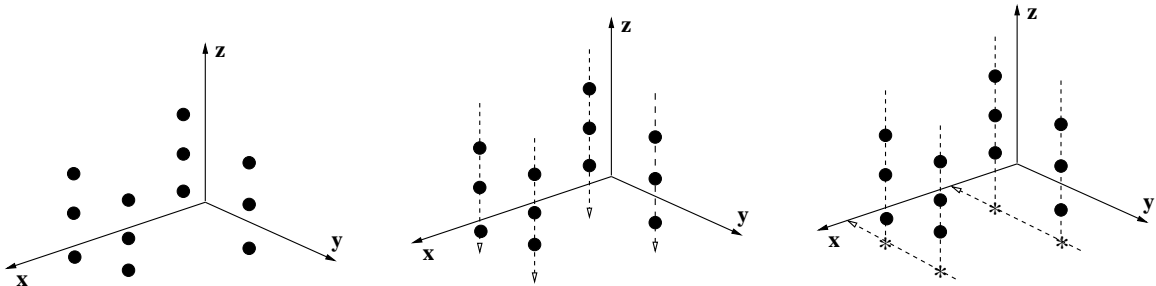
Figure 4.4: Definition of an Equiprojectable Variety

The variety V can be decomposed as the disjoint union of equiprojectable ones, in possibly several ways. Any such decomposition amounts to represent V as the disjoint union of the zeros of some triangular sets. The equiprojectable decomposition is a canonical way of doing so, defined by combinatorial means.

Equiprojectable decomposition. Let first W be a 0-dimensional variety in $\mathbb{A}^i(\overline{k})$, for some $1 \leq i \leq n$. For x in $\mathbb{A}^{i-1}(\overline{k})$, we define the preimage

$$\mu(x, W) = (\pi_{i-1}^i)^{-1}(x) \cap W;$$

for any $d \geq 1$, we can then define

$$A(d, W) = \left\{ x \in W \mid \#\mu(\pi_{i-1}^i(x), W) = d \right\}.$$

Thus, x is in $A(d, W)$ if W contains exactly d points x' such that $\pi_{i-1}^i(x) = \pi_{i-1}^i(x')$ holds. Only finitely many of the $A(d, W)$ are not empty and the non-empty ones form a partition of W. Let $1 \leq i \leq n$. Writing $W = \pi_i^n(V)$, we define

$$B(i, d, V) = \{ x \in V \mid \pi_i^n(x) \in A(d, W) \}.$$

Thus, $B(i, d, V)$ is the preimage of $A(d, W)$ in V, so these sets form a partition of V. If V is i-equiprojectable, then all $B(i, d, V)$ are $(i-1)$-equiprojectable. We then define inductively $B(V) = V$, and, for $1 < i \leq n$, $B(d_i, \ldots, d_n, V) =$

$B(i, d_i, B(d_{i+1}, \ldots, d_n, V))$. All $B(d_i, \ldots, d_n, V)$ are $(i - 1)$-equiprojectable, only finitely many of them are not empty, and the non-empty ones form a partition of V.

The *equiprojectable decomposition* of V is its partition into the family of all non-empty $B(d_2, \ldots, d_n, V)$. All these sets being equiprojectable, they are defined by triangular sets. Let us illustrate by the example in Figure 4.5. The leftmost picture shows a variety consisting of 12 points in (x, y, z). First, we project to the (x, y) plane, as described in the middle picture. This projection has 5 fibers. We will put the fibers with the same cardinality into the same set. Here, the 5 fibers are partitioned into 2 sets: round dots and square dots. The set of round dots contains 2 fibers with cardinality of 3, and the set of square dots contains 3 fibers with cardinality of 2. Next in the leftmost picture, we project from (x, y) to the (x) axis and we apply the same rule. The set of square dots is further split into 2 sets: the square dots and the triangular dots. Now the original variety is partitioned into 3 sets. Each of them is an equiprojectable variety. We call this process the equiprojectable decomposition.
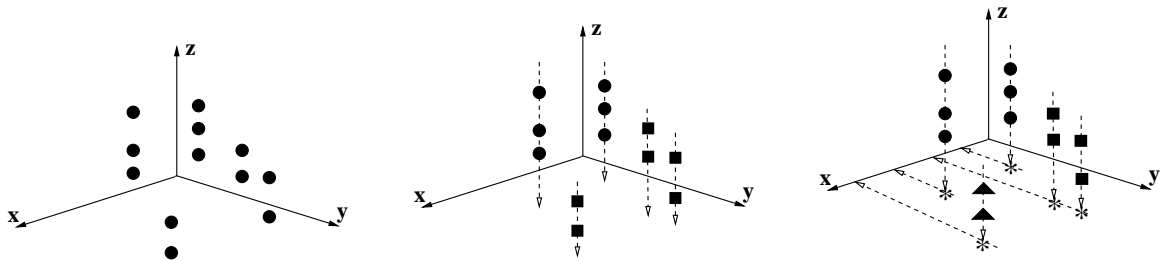
Figure 4.5: Definition of an Equiprojectable Decomposition

4.2.2 Split-and-Merge Algorithm

Recall that the *equiprojectable decomposition* of V is its partition into the family of all non-empty $B(d_2, \ldots, d_n, V)$. All these sets being equiprojectable, they are defined by triangular sets. Note that we have not proved yet that the $B(d_2, \ldots, d_n, V)$ are defined over the same field as V. This will come as a by-product of the algorithms of this section. To do so, we introduce first the notions of a *critical pair* and a *solvable pair*.

Critical and solvable pairs. Let $T \neq T'$ be two triangular sets. The least integer ℓ such that $T_\ell \neq T'_\ell$ is called the *level* of the pair T, T'. If $\ell = 1$ we let $K_\ell = k$, otherwise

we define $K_\ell = k[X_1, \ldots, X_{\ell-1}]/\langle T_1, \ldots, T_{\ell-1} \rangle$. Since a triangular set generates a radical ideal, the residue class ring K_ℓ is a direct product of fields. Therefore, every pair of univariate polynomials with coefficients in K_ℓ has a GCD in the sense of [109]. The pair T, T' is *critical* if T_ℓ and T'_ℓ are not relatively prime in $K_\ell[X_\ell]$. If T, T' is not critical, it is *certified* if $U, U' \in K_\ell[X_\ell]$ such that $UT_\ell + U'T'_\ell = 1$ are known. The pair T, T' is *solvable* if it is not critical and if for all $\ell < j \leq n$ we have $\deg_{X_j} T_j = \deg_{X_j} T'_j$.

Introducing the notion of a certified solvable pair is motivated by efficiency considerations. Indeed, during the splitting step, solvable pairs are discovered. Then, during the merging step, the Bézout coefficients U, U' of these solvable pairs will be needed for Chinese Remaindering.

Solvable families. We extend the notion of *solvability* from a pair to a family of triangular sets. A family $\mathfrak{T}$ of triangular sets is *solvable* (resp. *certified solvable*) at level ℓ if every pair $\{T, T'\}$ of elements of $\mathfrak{T}$ is solvable (resp. certified solvable) of level ℓ.

The following proposition shows how to recombine such families. When this is the case, we say that all T in $\mathfrak{T}$ *divide* S. In what follows, we write $V(\mathfrak{T})$ for $\cup_{T \in \mathfrak{T}} V(T)$.

Proposition 4.2.1. *If $\mathfrak{T}$ is certified solvable at level ℓ, one can compute a triangular set S such that $V(S) = V(\mathfrak{T})$, using only multiplications in $K_\ell[X_\ell]$.*

PROOF. First, we assume that $\mathfrak{T}$ consists of the pair $\{T, T'\}$. We construct S as follows. We set $S_i = T_i$ for $1 \leq i < \ell$ and $S_\ell = T_\ell T'_\ell$. Let $\ell < i \leq n$. For computing S_i, we see T_i and T'_i in $K_\ell[X_\ell][X_{\ell+1}, \ldots, X_i]$. We apply Chinese remaindering to the coefficients in T_i and T'_i of each monomial in $X_{\ell+1}, \ldots, X_i$ occurring in T_i or T'_i: since the Bézout coefficients U, U' for T_ℓ, T'_ℓ are known, this can be done using multiplications in $K_\ell[X_\ell]$ only. It follows from the Chinese Remaindering Theorem that the ideal $\langle S \rangle$ is equal to $\langle T \rangle \cap \langle T' \rangle$; for $i > \ell$, the equality $\deg_{X_i} T_i = \deg_{X_i} T'_i$ shows that S is monic in X_i, as requested.

Assume that $\mathfrak{T}$ consists of $s > 2$ triangular sets $T^1, \ldots, T^s$. First, we apply the case $s = 2$ to T^1, T^2, obtaining a triangular set $T^{1,2}$. Observe that every pair $T^{1,2}, T^j$, for $3 \leq j \leq s$, is solvable but not certified solvable: we obtain the requested Bézout coefficient by updating the known ones. Let us fix $3 \leq j \leq s$. Given $A_1, A_2, B_1, B_j, C_2, C_j \in K_\ell[X_\ell]$ such that $A_1 T^1_\ell + A_2 T^2_\ell = B_1 T^1_\ell + B_j T^j_\ell = C_2 T^2_\ell + C_j T^j_\ell = 1$ hold in $K_\ell[X_\ell]$, we let $\alpha = B_1 C_2 \mod T^j_\ell$ and $\beta = A_1 C_j T^1_\ell + A_2 B_j T^2_\ell \mod T^1_\ell T^2_\ell$. Then, $\alpha T^{1,2}_\ell + \beta T^j_\ell = 1$ in $K_\ell[X_\ell]$, as requested. Proceeding by induction ends the proof. $\square$

Splitting critical pairs. Let now V be a 0-dimensional variety over k. Proposition 4.2.3 below encapsulates the first part of the *Split-and-Merge* algorithm: given

any triangular decomposition $\mathfrak{T}$ of V, it outputs another one, without critical pairs. We first describe the basic splitting step.

Proposition 4.2.2. *Let $\mathfrak{T}$ be a triangular decomposition of V which contains critical pairs. Then one can compute a triangular decomposition $\mathsf{Split}(\mathfrak{T})$ of V which has cardinality larger than that of $\mathfrak{T}$.*

PROOF. Let T, T' be a critical pair of $\mathfrak{T}$ of level ℓ and let G be a GCD of T_ℓ, T'_ℓ in $K_\ell[X_\ell]$. First, assume that G is monic, in the sense of [109]; let Q and Q' be the quotients of T_ℓ and T'_ℓ by G in $K_\ell[X_\ell]$. We define the sets

$$
\begin{aligned}
A &= T_1, \ldots, T_{\ell-1}, G, T_{\ell+1}, \ldots, T_n, \\
B &= T_1, \ldots, T_{\ell-1}, Q, T_{\ell+1}, \ldots, T_n, \\
A' &= T_1, \ldots, T_{\ell-1}, G, T'_{\ell+1}, \ldots, T'_n, \\
B' &= T_1, \ldots, T_{\ell-1}, Q', T'_{\ell+1}, \ldots, T'_n.
\end{aligned}
$$

We let $\mathsf{Split}(\mathfrak{T}) = \{A, B, A', B'\}$, excluding the triangular sets defining the empty set. Since the pair T, T' is critical, $V(A)$ and $V(B)$ are non-empty. Since T_ℓ and T'_ℓ are not associate in $K_\ell[X_\ell]$, at least Q or Q' is not constant. Thus, $\mathsf{Split}(\mathfrak{T})$ has cardinality at least 3. Since $\langle T \rangle$ and $\langle T' \rangle$ are radical, if $Q \notin K_\ell$, G and Q are coprime in $K_\ell[X_\ell]$, so $V(T)$ is the disjoint union of $V(A)$ and $V(B)$. The same property holds for A' and B'. Thus, the proposition is proved.

Assume now that T_ℓ, T'_ℓ have no monic GCD in $K_\ell[X_\ell]$. Then, there exist triangular sets $C^1, \ldots, C^s, D^1 \ldots D^s$ such that $V(T)$ is the disjoint union of $V(C^1)$, ..., $V(C^s)$, $V(T')$ is the disjoint union of $V(D^1)$, ..., $V(D^s)$, at least one pair C^i, D^j is critical and C^i_ℓ, D^j_ℓ admits a monic GCD in $K_\ell[X_\ell]$. These triangular sets are obtained by the algorithms of [109] when computing a GCD of T_ℓ, T'_ℓ in $K_\ell[X_\ell]$. Then the results of the monic case prove the existence of $\mathsf{Split}(\mathfrak{T})$. $\square$

Proposition 4.2.3. *Let $\mathfrak{T}$ be a triangular decomposition of V. One can compute a triangular decomposition $\mathfrak{T}'$ of V with no critical pairs, and where each pair of triangular sets is certified.*

PROOF. Write $\mathfrak{T}_0 = \mathfrak{T}$, and define a sequence $\mathfrak{T}_i$ by $\mathfrak{T}_{i+1} = \mathsf{Split}(\mathfrak{T}_i)$, if $\mathfrak{T}_i$ contains critical pairs, and $\mathfrak{T}_{i+1} = \mathfrak{T}_i$ otherwise. Testing the presence of critical pairs is done by GCD computations, which yields the Bézout coefficients in case of coprimality. Let D be the number of irreducible components of V. Any family $\mathfrak{T}_i$ has cardinality at most D, so the sequence $\mathfrak{T}_i$ becomes stationary after at most D steps. $\square$

Thus, we can now suppose that we have a triangular decomposition $\mathfrak{T}$ of V without critical pairs, and where every pair is certified, such as the one computed in Proposition 4.2.3. We describe the second part of the *Split-and-Merge* algorithm: merging solvable families in a suitable order, to obtain the equiprojectable decomposition of V.

For $0 \leq \kappa \leq n$, we say that $\mathfrak{T}$ satisfies property P_κ if for all $T, T' \in \mathfrak{T}$ the pair $\{T, T'\}$ is certified, has level $\ell \leq \kappa$ and for all $\kappa < i \leq n$ satisfies $\deg_{X_i} T_i = \deg_{X_i} T'_i$. Observe that if $\mathsf{P}_0(\mathfrak{T})$ holds, then $\mathfrak{T}$ contains only one triangular set, and that the input family $\mathfrak{T}$ satisfies P_n.

The basic merging algorithm. Let $1 \leq \kappa \leq n$. We now define the procedure Merge_κ, which takes as input a family $\mathfrak{T}_\kappa$ of triangular sets which satisfies P_κ, and outputs several families of triangular sets, whose reunion defines the same set of points, and all of which satisfy $\mathsf{P}_{\kappa-1}$. First, we partition $\mathfrak{T}_\kappa$ using the equivalence relation $T \equiv T'$ if and only if $T_1, \ldots, T_{\kappa-1} = T'_1, \ldots, T'_{\kappa-1}$. Assumption P_κ shows that each equivalence class is certified and solvable of level κ. We then let $\mathfrak{S}^{(\kappa)}$ be the family of triangular sets obtained by applying Proposition 4.2.1 to each equivalence class.

Lemma 4.2.4. *Let $S \neq S'$ in $\mathfrak{S}^{(\kappa)}$. The pair $\{S, S'\}$ is non-critical, certified, of level $\ell < \kappa$.*

PROOF. Let $T, T' \in \mathfrak{T}$, which respectively divide S and S'. Due to assumption P_κ, there exists $0 \leq \ell \leq \kappa$ such that $T_1, \ldots, T_{\ell-1} = T'_1, \ldots, T'_{\ell-1}$ and $(T_1, \ldots, T_\ell)$ and $(T'_1, \ldots, T'_\ell)$ have no common zero. Then, $\ell < \kappa$, since $T \not\equiv T'$. Thus, $T_1, \ldots, T_\ell = S_1, \ldots, S_\ell$ and $T'_1, \ldots, T'_\ell = S'_1, \ldots, S'_\ell$. Since $\{T, T'\}$ is certified of level $\ell < \kappa$, $\{S, S'\}$ is also. $\square$

We partition $\mathfrak{S}^{(\kappa)}$ some more, into the classes of the equivalence relation $S \equiv' S'$ if and only if $\deg_{X_\kappa} S_\kappa = \deg_{X_\kappa} S'_\kappa$. Let $\mathfrak{S}^{(\kappa)}_1, \ldots, \mathfrak{S}^{(\kappa)}_\delta$ be the equivalence classes, indexed by the common degree in X_κ; we define $\mathsf{Merge}_\kappa(\mathfrak{T}_\kappa)$ as the data of all these equivalence classes.

Lemma 4.2.5. *Each family $\mathfrak{S}^{(\kappa)}_d$ satisfies $\mathsf{P}_{\kappa-1}$.*

PROOF. Let $S \neq S'$ in $\mathfrak{S}^{(\kappa)}_d$, and let T, T' be as in the proof of Lemma 4.2.4; we now prove the degree estimate. For $\kappa < i \leq n$, we have $\deg_{X_i} T_i = \deg_{X_i} S_i$ and $\deg_{X_i} T'_i = \deg_{X_i} S'_i$; assumption P_κ shows that $\deg_{X_i} S_i = \deg_{X_i} S'_i$ for $\kappa < i \leq n$. Since $\deg_{X_\kappa} S_\kappa = \deg_{X_\kappa} S'_\kappa = d$, the lemma is proved. $\square$

Proposition 4.2.6. $V(\mathfrak{S}_d^{(\kappa)}) = B(\kappa, d, V(\mathfrak{T}_\kappa))$ *for all d.*

PROOF. We know that $V(\mathfrak{T}_\kappa)$ is the union of the $V(\mathfrak{S}_d^{(\kappa)})$. Besides, both families $\{V(\mathfrak{S}_d^{(\kappa)})\}$ and $\{B(\kappa, d, V(\mathfrak{T}))\}$ form a partition of $V(\mathfrak{T}_\kappa)$. Thus, it suffices to prove that for x in $V(\mathfrak{T}_\kappa)$, $x \in V(\mathfrak{S}_d^{(\kappa)})$ implies that $\pi_\kappa^n(x) \in A(d, W)$, with $W = \pi_\kappa^n(V(\mathfrak{T}_\kappa))$. First, for S in $\mathfrak{S}^{(\kappa)}$, write $W_S = \pi_\kappa^n(S)$. Then Lemma 4.2.4 shows that the W_S form a partition of W, and that their images $\pi_{\kappa-1}^\kappa(W_S)$ are pairwise disjoint.

Let now $x \in V(\mathfrak{S}_d^{(\kappa)})$ and $y = \pi_\kappa^n(x)$. There exists a unique $S \in \mathfrak{S}^{(\kappa)}$ such that $x \in V(S)$. The definition of $\mathfrak{S}_d^{(\kappa)}$ shows that there are exactly d points y' in W_S such that $\pi_{\kappa-1}^\kappa(y) = \pi_{\kappa-1}^\kappa(y')$. On the other hand, for any $y \in W_{S'}$, with $S' \neq S$, the above remark shows that $\pi_{\kappa-1}^\kappa(y) \neq \pi_{\kappa-1}^\kappa(y')$. Thus, there are exactly d points y' in W such that $\pi_{\kappa-1}^\kappa(y) = \pi_{\kappa-1}^\kappa(y')$; this concludes the proof. $\square$

The main merging algorithm. We can now give the main algorithm. We start from a triangular decomposition $\mathfrak{T}$ of V without critical pairs, and where every pair is certified, so it satisfies P_n. Let us initially define $\mathfrak{T}_n = \{\mathfrak{T}\}$; note that $\mathfrak{T}_n$ is a *set of families of triangular sets*. Then, for $1 \leq \kappa \leq n$, assuming $\mathfrak{T}_\kappa$ is defined, we write $\mathfrak{T}_{\kappa-1} = \cup_{\mathfrak{U}^{(\kappa)} \in \mathfrak{T}_\kappa} \mathsf{Merge}_\kappa(\mathfrak{U}^{(\kappa)})$. Lemma 4.2.5 shows that this process is well-defined; note that each $\mathfrak{T}_\kappa$ is a set of families of triangular sets as well.

Let $\mathfrak{U}$ be a family of triangular sets in $\mathfrak{T}_0$. Then $\mathfrak{U}$ satisfies P_0, so by the remarks made previously, $\mathfrak{U}$ consists of a single triangular set. Proposition 4.2.6 then shows that the triangular sets in $\mathfrak{T}_0$ form the equiprojectable components of V.

An example. Re-consider the example sys in Section 4.1. $Z(\mathsf{sys})$ is represented by its irreducible decomposition T^1 and T^2, shown in Figure 4.2. One possible representation of $Z(\mathsf{sys} \bmod 7)$ is by triangular sets t^1 and t^2, described in Figure 4.3, which are not the reduction modulo 7 of T^1 and T^2. We explain below how to compute t'^1 and t'^2, the equiprojectable decomposition of $Z(\mathsf{sys} \bmod 7)$, from t^1 and t^2 by the *Split-and-Merge* algorithm.

Recall the pair of triangular sets t^1 and t^2, where

$$t^1 \left| \begin{array}{l} t_2{}^1 = X_2^2 + 6X_2X_1^2 + 2X_2 + X_1 \\ t_1{}^1 = X_1^3 + 6X_1^2 + 5X_1 + 2 \end{array} \right. \quad \text{and} \quad t^2 \left| \begin{array}{l} t_2{}^2 = X_2 + 6 \\ t_1{}^2 = X_1 + 6 \end{array} \right.$$

We re-illustrate t^1 and t^2 in the leftmost picture of Figure 4.6. It shows that t^1 contains 6 points, and t^2 has 1 point. By examining this pair of components, it is not hard to see that $t_1{}^1$ and $t_1{}^2$ are not co-prime. Their GCD is $X_1 + 6$. This finding indicates that t^1 and t^2 is a *critical pair*. Thus, t^1 is split into two triangular sets, $t^{1,1}$

Algorithm 27 Merge

Input $[T^1, \ldots, T^n]$: a triangular decomposition in $k[X_1, \ldots, X_n]$ of a 0-dimensional variety V. No pair $((T^i, T^j), i \neq j)$ is critical.

BezoutCoeffsTable: a table with key of $[T_{<\ell}{}^{i,j}, X_l, T_\ell{}^i, T_\ell{}^j]$ and value of $[g^{i,j} = 1, A_i, A_j]$ (normalized). $T_{<\ell}{}^{i,j}$ is the regular chain below the level of l, and $A_i T^i + A_j T^j = g^{i,j} \bmod T_{<\ell}{}^{i,j}$.

$lm = [M^1, \ldots, M^n]$: (optional) a list of matrices corresponding to $[T^1, \ldots, T^n]$.

Output The equiprojectable decomposition of V.

Merge($[T^1, \ldots, T^n]$, *BezoutCoeffsTable*, lm) ==

1: $rc\_list \leftarrow [T^1, \ldots, T^n]$
2: **if** lm is not empty **then** $m\_list \leftarrow [M^1, \ldots, M^n]$
3: **for** X_ℓ in $X_n, \ldots, X_1$ **do**
4:    $E \leftarrow$ GetSolvableEquivalenceClasses(rc_list, X_ℓ)
5:    **for** E^i in E **do**
6:       $rc\_list \leftarrow rc\_list \setminus T^i$ for all T^i in E^i
7:       **while** cardinality of $E^i > 1$ **do**
8:          $T^m, T^k \leftarrow$ Pop out two regular chains from E^i
9:          Get $g^{m,k}, A_m, A_k$ from *BezoutCoeffsTable* by key of $T_{<\ell}{}^{m,k}, X_\ell, T_\ell{}^m, T_\ell{}^k$
10:         $T^{m,k} \leftarrow$ MergeSolvablePair($T^m, T^k, g^{m,k}, A_m, A_k, X_\ell$)
11:         **if** cardinality of $E^i > 0$ **then**
12:            **for** T^i in E^i **do**
13:               Get A_i, B_m, B_i, C_k, C_i from *BezoutCoeffsTable* s.t.
$A_m T_\ell{}^m + A_k T_\ell{}^k = B_m T_\ell{}^m + B_i T_\ell{}^i = C_k T_\ell{}^k + C_i T_\ell{}^i = 1$;
14:               $\alpha \leftarrow B_m C_k \bmod T_\ell{}^i; \beta \leftarrow (A_m C_i T_\ell{}^m + A_k B_i T_\ell{}^k) \bmod T_\ell{}^m T_\ell{}^k$;
15:               Add $[1, \alpha, \beta]$ for $[T_{<\ell}{}^{m,k}, X_\ell, T_\ell{}^{m,k}, T_\ell{}^i]$ to *BezoutCoeffsTable*
16:            **end for**
17:         **end if**
18:         **if** lm is not empty **then**
19:            $M^m, M^k \leftarrow$ Pop out two matrices related to T^m, T^k from m_list
20:            $M^{m,k} \leftarrow$ MergeMatrixPair($T^m, T^k, g^{m,k}, A_m, A_k, X_\ell, M^m, M^k$)
21:            $M^{m,k} \leftarrow$ NormalFormMatrix($T^{m,k}, M^{m,k}$)
22:            $m\_list \leftarrow m\_list \cup M^{m,k}$
23:         **end if**
24:         $E^i \leftarrow E^i \cup T^{m,k}$
25:       **end while**
26:       $rc\_list \leftarrow rc\_list \cup E^i$
27:    **end for**
28: **end for**
29: **if** lm is not empty **then return** $[rc\_list, m\_list]$
30: **return** rc_list

Algorithm 28 Get Solvable Equivalence Classes

    Input $[T^1, \ldots, T^n]$: a list of regular chains in $k[X_1, \ldots, X_n]$.

            X_ℓ: a variable.

            BezoutCoeffsTable: a table with key of $[T_{<\ell}{}^{i,j}, X_\ell, T_\ell^i, T_\ell^j](1 \le \ell < n)$ and
            value of $[g^{i,j} = 1, A_i, A_j]$ (normalized).

   Output Equivalence solvable classes at level ℓ.

GetSolvableEquivalenceClasses($[T^1, \ldots, T^n], X_\ell$, *BezoutCoeffsTable*)
==

1: $rc\_list \leftarrow [T^1, \ldots, T^n]$; $\ell\_list \leftarrow [\ ]$; $E \leftarrow [\ ]$
2: **while** rc_list not empty **do**
3:    $T^t \leftarrow$ Pop out one regular chain from rc_list
4:    $c\_list \leftarrow [\ ]$; $t\_list \leftarrow [\ ]$; $in\_class \leftarrow$ false
5:    **for** T^i in rc_list **do**
6:      **if** $[T^t_{<X_{\ell-1}}, X_\ell, T^t_{X_\ell}, T^i_{X_\ell}]$ has value in *BezoutCoeffsTable*
        (i.e. T^i and T^t are certified solvable at level ℓ) **then**
7:        **if** $in\_class =$false **then**
8:          $c\_list \leftarrow c\_list \cup T^t$; $in\_class \leftarrow$true
9:        **end if**
10:       $c\_list \leftarrow c\_list \cup T^i$; $t\_list \leftarrow t\_list \cup T^i$
11:     **end if**
12:    **end for**
13:    $\ell\_list \leftarrow \ell\_list \cup [c\_list]$; $rc\_list \leftarrow rc\_list \setminus t\_list$;
14: **end while**
15: **while** ℓ_list is not empty **do**
16:   $equi\_list \leftarrow$ Pop out one list from ℓ_list
17:   **while** $equi\_list$ is not empty **do**
18:     $rc_1 \leftarrow$ Pop out one regular chain from $equi\_list$
19:     $E^1 \leftarrow [rc_1]$
20:     $ds \leftarrow$ degree sequence of $X_n, \ldots, X_{\ell+1}$ of rc_1
21:     **for** T^i in $equi\_list$ **do**
22:       **if** degree sequence of T^i is the same as ds **then**
23:         $E^1 \leftarrow E^1 \cup T^i$
24:       **end if**
25:     **end for**
26:     $equi\_list \leftarrow equi\_list \setminus E^1$
27:     $E \leftarrow E \cup [E^1]$
28:   **end while**
29: **end while**
30: **return** E

Algorithm 29 Merge Solvable Pair

Input T^1, T^2: two regular chains in $k[X_1, \ldots, X_n]$.

X_ℓ: a variable.

$g^{1,2} = 1, A_1, A_2$: gcd and Bezóut coefficients s.t. $A_1 T^1_{X_\ell} + A_2 T^2_{X_\ell} = g^{1,2} = 1$.

Output A regular chain merged from T^1 and T^2.

MergeSolvablePair$(T^1, T^2, X_\ell, g^{1,2}, A_1, A_2) ==$

1: $f \leftarrow T^1 \times T^2$
2: $new\_rc \leftarrow$ Construct a regular chain from $T^{1,2}_{<X_\ell}$ and f
3: **if** $l = n$ **then**
4:     **return** new_rc
5: **end if**
6: $A_1' \leftarrow A_1 \times T_{X_\ell}{}^1$; $A_2' \leftarrow A_2 \times T_{X_\ell}{}^2$
7: **for** i from $\ell + 1$ to n **do**
8:     $T_i^{1,2} \leftarrow$MergePolynomialPair$(T^1_{X_i}, T^2_{X_i}, A_1', A_2', \{X_{\ell+1}, \ldots, X_i\}, T^{1,2}_{<X_\ell})$
9:     $new\_rc \leftarrow$Construct a regular chain from new_rc and $T_i^{1,2}$
10: **end for**
11: **return** new_rc;

Algorithm 30 Merge Polynomial Pair

Input f_1, f_2: two polynomials with the same main variable X_i and same main degree.

fs, gt: Bezóut relations s.t. $fs + gt \equiv 1 \bmod rc$.

$\{X_i, \ldots, X_{\ell+1}\}$: a variable set.

rc: a regular chain below level of l.

Output A polynomial f merged from f_1 and f_2 by CRT s.t. $f' = f_1 \bmod rc + g$ and $f' = f_2 \bmod rc + f$.

MergePolynomialPair$(f_1, f_2, fs, gt, \{X_i, \ldots, X_{\ell+1}\}, rc) ==$

1: $D \leftarrow \mathbf{lcm}(\text{init}(f_1), \text{init}(f_2))$
2: $p_1 \leftarrow f_1 \times D/\text{init}(f_1)$
3: $p_2 \leftarrow f_2 \times D/\text{init}(f_2)$
4: $f' \leftarrow 0$;
5: **for** each monomial $t = X_{\ell+1}{}^{y_{\ell+1}} \ldots X_i{}^{y_i}$ $(0 \leq y_{\ell+1} \leq d_{\ell+1}, \ldots, 0 \leq y_i \leq d_i)$ **do**
6:     Merge the corresponding coefficients by
    $C_t \leftarrow fs \times (\mathbf{Coefficient}(t) \text{ of } p_1) + gt \times (\mathbf{Coefficient}(t) \text{ of } p_2)$
7:     $C_t' \leftarrow \mathbf{NormalForm}(C_t, rc)$
8:     $f' \leftarrow f' + C_t' \times t$
9: **end for**
10: **return** f'

Algorithm 31 Merge Matrix Pair

Input T^1, T^2: two regular chains in $k[X_1, \ldots, X_n]$.
$\quad\quad\quad$ X_ℓ: a variable.
$\quad\quad\quad$ $g^{1,2} = 1, A_1, A_2$: gcd and Bezóut coefficients s.t. $A_1 T_{X_\ell}^1 + A_2 T_{X_\ell}^2 = g^{1,2} = 1$.
$\quad\quad\quad$ M^1, M^2: two matrices with the same dimension $r \times c$.

Output A matrix M merged from M^1 and M^2 by CRT s.t. $M \equiv M^1 \bmod T^1$ and
$\quad\quad\quad\quad$ $M \equiv M^2 \bmod T^2$.

MergeMatrixPair($T^1, T^2, X_\ell, g^{1,2}, A_1, A_2, M^1, M^2$) ==
1: $M \leftarrow$ zero matrix of dimension $r \times c$
2: $A_1' \leftarrow A_1 \times T_{X_\ell}{}^1$; $A_2' \leftarrow A_2 \times T_{X_\ell}{}^2$
3: **for** i from 1 to r **do**
4: $\quad$ **for** j from 1 to c **do**
5: $\quad\quad$ $M[i,j] \leftarrow$
$\quad\quad$ MergePolynomialPair($M^1[i,j], M^2[i,j], A_1', A_2', \{X_{\ell+1}, \ldots, X_n\}, T_{<X_\ell}^{1,2}$)
6: $\quad$ **end for**
7: **end for**
8: **return** M

---

and $t^{1,2}$. $t^{1,1}$ contains 4 points and $t^{1,2}$ contains 2 points. The result is shown in the middle picture of Figure 4.6.

After splitting, we obtain 3 triangular sets, $t^{1,1}$, $t^{1,2}$ and t^2. They are free of critical pairs. Next we merge all the solvable pairs based on the Chinese Remaindering Theorem. In this example, $t^{1,2}$ and t^2 is a solvable pair. They are merged into component t'^2. The result is described in the rightmost picture of Figure 4.6, where $t^{1,1}$ is renamed as t'^1. It can be seen that t'^1 has 2 fibers with the same cardinality 2, and t'^2 has 1 fiber with cardinality 3. Hence, we obtain the equiprojectable decomposition of $Z(\text{sys} \bmod 7)$.
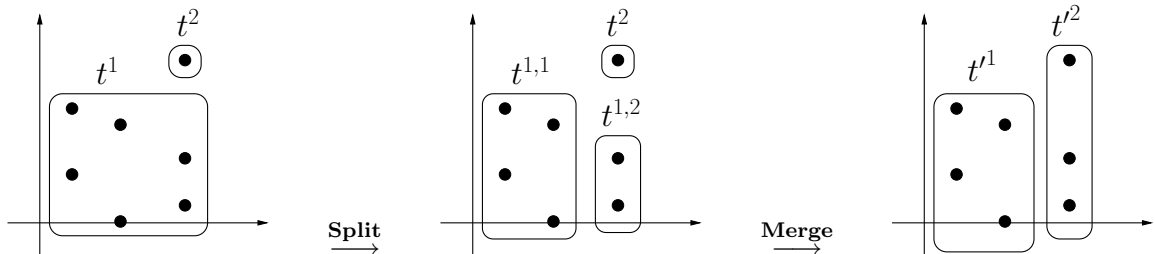
Figure 4.6: An Example for the *Split-and-Merge* Algorithm

4.3 A Modular Algorithm for Triangular Decompositions

We now give the details of our lifting algorithm for triangular decompositions based on the *Split-and-Merge* algorithm and the height bound given by Theorem 4.1.1, see its proof in [39]. Given a polynomial system F, it outputs a triangular representation of its set of simple solutions $Z = Z(F)$, by means of the polynomials $N^1, \ldots, N^s$ defined in the introduction. First of all, we describe the required subroutines, freely using the notation of Theorem 4.1.2, and that preceding it. We do not give details of the complexity estimates; they are similar to those of [120].

- `EquiprojDecomposition` takes as input a polynomial system F and outputs the equiprojectable decomposition of $Z(F)$, encoded by triangular sets. This routine is called here for systems defined over finite fields. For the experiments in the next section, we applied the triangularization algorithm of [108], followed by the Split-and-Merge algorithm of Section 4.2.2, modulo a prime. Studying the complexity of this task is left to the forthcoming [41]; hence, we consider this subroutine as an oracle here, which is called O_2 in Theorem 4.1.2.

- `Lift` applies the Hensel lifting algorithm of [120], but this time to a *family of triangular sets*, defined first modulo a prime p_1, to triangular sets defined modulo the successive powers $p_1^{2^\kappa}$. From [120], one easily sees that the κth lifting step has a bit complexity quasi-linear in $(L, h_L, \mathsf{C}^n, \sum_{i \leq s} \deg V(T^i), 2^\kappa, \log p_1)$, *i.e.* in $(L, h_L, \mathsf{C}^n, \deg Z, 2^\kappa, \log p_1)$.

- `Convert` computes the polynomials N^i starting from the polynomials T^i. Only multiplications modulo triangular sets are needed to perform this operation, so its complexity is negligible before that of `Lift`.

- `RationalReconstruction` does the following. Let $a = p/q \in \mathbb{Q}$, and $m \in \mathbb{N}$ with $\gcd(q, m) = 1$. If $\mathbf{h}m \geq 2\mathbf{h}a + 1$, given $a \bmod m$, this routine outputs a. If $\mathbf{h}m < 2\mathbf{h}a + 1$, the output may be undefined, or differ from a. We extend this notation to the reconstruction of all coefficients of a family of triangular sets. Using the fast Euclidean algorithm [57, Ch 5,11], its complexity is negligible before that of `Lift`.

- We do not consider the cost of prime number generation. We see them as input here; formally, in Theorem 4.1.2, this is handled by calls to oracle O_1.

Algorithm 32 Lifting a Triangular Decomposition

    Input The system F, primes p_1, p_2.

  Output The polynomials $N^1, \ldots, N^s$.

1: $T^{1,0}, \ldots, T^{s,0} \leftarrow$ EquiprojDecomposition($Z(F \bmod p_1)$);
2: $u^1, \ldots, u^{s'} \leftarrow$ EquiprojDecomposition($Z(F \bmod p_2)$);
3: $m^1, \ldots, m^{s'} \leftarrow$ Convert($u^1, \ldots, u^{s'}$);
4: $\kappa \leftarrow 1$;
5: **while** not Stop **do**
6:    $T^{1,\kappa}, \ldots, T^{s,\kappa} \leftarrow$ Lift($T^{1,\kappa-1}, \ldots, T^{s,\kappa-1}$) $\bmod p_1^{2^\kappa}$;
7:    $N^{1,\kappa}, \ldots, N^{s,\kappa} \leftarrow$ Convert($T^{1,\kappa}, \ldots, T^{s,\kappa}$);
8:    $N_{\mathbb{Q}}^{1,\kappa}, \ldots, N_{\mathbb{Q}}^{s,\kappa} \leftarrow$ RationalReconstruction($N^{1,\kappa}, \ldots, N^{s,\kappa}$);
9:    **if** $\{m^1, \ldots, m^{s'}\}$ Equals $\{N_{\mathbb{Q}}^{1,\kappa}, \ldots, N_{\mathbb{Q}}^{s,\kappa}\} \bmod p_2$ **then**
10:      $Stop = true$;
11:    **end if**
12:    $\kappa \leftarrow \kappa + 1$;
13: **end while**
14: **return** $N_{\mathbb{Q}}^{1,\kappa-1}, \ldots, N_{\mathbb{Q}}^{s,\kappa-1}$;

We still use the notation and assumption of Theorem 4.1.2. From [42, Th. 1], all coefficients of $N^1, \ldots, N^s$ have height in $n^{O(1)}(\deg Z + \mathbf{h}Z)$, which can explicitly be bounded by $\mathfrak{h}_F$. For $p_1 \leq \exp(2\mathfrak{h}_F + 1)$, define

$$\mathfrak{d} = \mathfrak{d}(p_1) = \left\lceil \log_2\left(\frac{2\mathfrak{h}_F + 1}{\log p_1}\right) \right\rceil.$$

Then, $p_1^{2^{\mathfrak{d}(p_1)}}$ has height at least $2\mathfrak{h}_F + 1$. In view of the prerequisites for rational reconstruction, $\mathfrak{d}(p_1)$ bounds the number of lifting steps. From an intrinsic viewpoint, at the last lifting step, 2^κ is in $O(n^{O(1)}(\deg Z + \mathbf{h}Z))$.

Suppose that p_1 does not divide the integer A of Theorem 4.1.1. Then, Hensel lifting computes approximations $T^{1,\kappa}, \ldots, T^{s,\kappa} = T^1, \ldots, T^s$ modulo $p_1^{2^\kappa}$. At the κth lifting step, let $N^{1,\kappa}, \ldots, N^{s,\kappa}$ be the output of Convert applied to $T^{1,\kappa}, \ldots, T^{s,\kappa}$, computed modulo $p_1^{2^\kappa}$; let $N_{\mathbb{Q}}^{1,\kappa}, \ldots, N_{\mathbb{Q}}^{s,\kappa}$ be the same polynomials after rational number reconstruction, if possible. By construction, they have rational coefficients of height at most $2^{\kappa-1} \log p_1$. Supposing that p_2 does not divide the integer A of Theorem 4.1.1, failure occurs only if for some κ in $0, \ldots, \mathfrak{d}-1$, and some j in $1, \ldots, s$, $N_{\mathbb{Q}}^{j,\kappa}$ and N^j differ, but coincide modulo p_2. For this to happen, p_2 must divide some non-zero number of height at most $\mathfrak{h}_F + 2^{\kappa-1} \log p_1 + 1$. Taking all κ into account, this shows that for any prime p_1, there exists a non-zero integer B_{p_1} such that $\mathbf{h}B_{p_1} \leq (\mathfrak{h}_F + 1)\mathfrak{d} + 2^{\mathfrak{d}} \log p_1$,

and if p_2 does not divide B_{p_1}, the lifting algorithm succeeds. One checks that the above bound can be simplified into $\mathbf{h}B_{p_1} \leq \mathfrak{b}_F$.

Let $C \in \mathbb{N}$ be such that

$$C = \left\lceil \frac{4\mathfrak{a}_F + 2\mathfrak{b}_F}{\varepsilon} \right\rceil, \quad \text{so that } C \leq \frac{1}{2} \exp\left(2\mathfrak{h}_F + 1\right);$$

let Γ be the set of pairs of primes in $[C+1, \ldots, 2C]^2$ and γ be the number of primes in $C+1, \ldots, 2C$; note that $\gamma \geq C/(2 \log C)$ and that $\#\Gamma = \gamma^2$. The upper bound on C shows that all primes p less than $2C$ satisfy the requested inequality $\log p \leq 2\mathfrak{h}_F + 1$. We can then estimate how many choices of (p_1, p_2) in Γ lead to failure. There are at most $\mathfrak{a}_F/\log C$ primes p_1 in $C+1, \ldots, 2C$ which divide the integer A of Theorem 4.1.1, discriminating at most $\gamma\mathfrak{a}_F/\log C$ pairs (p_1, p_2). For any other value of p_1, there are at most $(\mathfrak{a}_F + \mathfrak{b}_F)/\log C$ choices of p_2 which divide A and B_{p_1}. This discriminates at most $\gamma(\mathfrak{a}_F + \mathfrak{b}_F)/\log C$ pairs (p_1, p_2). Thus the number of choices in Γ leading to failure is at most $\gamma(2\mathfrak{a}_F + \mathfrak{b}_F)/\log C$. The lower bound on γ shows that if (p_1, p_2) is chosen randomly with uniform probability in Γ, the probability that it leads to failure is at most

$$\frac{\gamma(2\mathfrak{a}_F + \mathfrak{b}_F)}{\#\Gamma \log C} = \frac{\gamma(2\mathfrak{a}_F + \mathfrak{b}_F)}{\gamma^2 \log C} = \frac{2\mathfrak{a}_F + \mathfrak{b}_F}{\gamma \log C} \leq \frac{4\mathfrak{a}_F + 2\mathfrak{b}_F}{C},$$

which is at most ε, as requested.

To estimate the complexity of this algorithm, note that since we double the precision at each lifting step, the cost of the last lifting step dominates. From the previous discussion, the number of bit operations cost at the last step is quasi-linear in $(L, h_L, \mathbf{C}^n, \deg Z, 2^\kappa, \log p_1)$. The previous estimates show that at this step, 2^κ is in $O(n^{O(1)}(\deg Z + \mathbf{h}Z))$, whereas $\log p_1$ is quasi-linear in $|\log \varepsilon|, \log h, d, \log n$. Putting all these estimates gives the proof of Theorem 4.1.2.

4.4   Experimental Results

We realized a first MAPLE 9.5 implementation of our modular algorithm on top of the `RegularChains` library [90]. Tests on benchmark systems [128] reveal its strong features, compared with two other MAPLE solvers, `Triangularize`, from the `Regular-Chains` library, and `gsolve`, from the `Groebner` library. Note that the triangular decompositions modulo a prime, that are needed in our algorithm, are performed by `Triangularize`. This function is a generic code: essentially the same code is used

over $\mathbb{Z}$ and modulo a prime. Thus, `Triangularize` is not optimized for modular computations.

Our computations are done on a 2799 MHz Pentium 4. For the time being our implementation handles square systems that generate radical ideals. We compare our algorithm called `TriangularizeModular` with `gsolve` and `Triangularize`;

For each benchmark system, Table 4.1 lists the numbers $n, d, h, \mathfrak{h}$ and Table 4.2 lists the prime p_1, the *a priori* and actual number of lifting steps ($\mathfrak{d}$ and a) and the maximal height of the output coefficients (C_a). Table 4.3 gives the time of one call to `Triangularize` modulo p_1 (Δ_p), the equiprojectable decomposition (E_p), and the lifting (Lift.) in seconds — the first two steps correspond to the "oracle calls" O_2 mentioned in Theorem 4.1.2, which is a work in progress in [40]. Table 3 gives also the total time, the total memory usage and output size for `TriangularizeModular`, whereas Table 4.4 gives that data for `Triangularize` and `gsolve`.

The maximum time is set up to 10800 seconds; we set the probability of success to be at least 90%.

`TriangularizeModular` solves 12 of the 14 test systems before the timeout, while `Triangularize` succeeds with 7 and `gsolve` with 6. Among most of the problems which `gsolve` can solve, `TriangularizeModular` shows less time consumed, less memory usage, and smaller output size. Noticeably, quite a few of the large systems can be solved by `TriangularizeModular` with time extension: system 13 is solved in 18745 seconds. Another interesting system is Pinchon-1 (from the FRISCO project), for which $n = 29, d = 16, h = 20, \mathfrak{h} = 1409536095e + 29$, which we solve in 64109 seconds. Both `Triangularize` and `gsolve` fail these problems due to memory allocation failure. Our modular method demonstrates its efficiency in reducing the size of the intermediate computations, whence its ability to solve difficult problems.

We observed that for every test system, for which E_p can be computed, the Hensel lifting always succeeds, *i.e.* the equiprojectable decomposition over $\mathbb{Q}$ can be reconstructed from E_p. In addition, `TriangularizeModular` failed `chemkin` at the Δ_p phase rather than at the lifting stage. Furthermore, the time consumed in the equiprojectable decomposition and the Hensel lifting is rather insignificant comparing with that in triangular decomposition modulo a prime. For every tested example the Hensel lifting achieves its final goal in less steps than the theoretical bound. In addition, the primes derived from our theoretical bounds are of quite moderate size, even on large examples.

Sys	Name	n	d	h	$\mathfrak{h}$
1	Cyclohexane	3	4	3	4395
2	Fee_1	4	4	2	24464
3	fabfaux	3	3	13	2647
4	geneig	6	3	2	116587
5	eco6	6	3	0	105718
6	Weispfenning-94	3	5	0	7392
7	Issac97	4	2	2	1511
8	dessin-2	10	2	7	358048
9	eco7	7	3	0	387754
10	Methan61	10	2	16	450313
11	Reimer-4	4	5	1	55246
12	Uteshev-Bikker	4	3	3	7813
13	gametwo5	5	4	8	159192
14	chemkin	13	3	11	850088102

Table 4.1: Features of the Polynomial Systems for Modular Method

Sys	p_1	$\mathfrak{o}$	a	C_a
1	4423	7	2	15
2	24499	8	4	70
3	2671	7	5	110
4	116663	10	5	162
5	105761	10	3	40
6	7433	7	3	31
7	1549	6	5	102
8	358079	11	7	711
9	387799	11	4	89
10	450367	11	6	362
11	55313	9	2	19
12	7841	7	5	125
13	159223	10	-	-
14	850088191	18	-	-

Table 4.2: Data for the Modular Algorithm

Sys	Δ_p	E_p	Lift	Total	Mem.	Output size
1	1	0.3	2	7	5	243
2	3	1	9	20	6	4157
3	8	0.4	6	22	7	5855
4	5	1	5	18	6	4757
5	12	1.5	6	35	6	2555
6	16	1.5	11	43	7	3282
7	66	0.4	4	133	8	4653
8	47	9	232	427	13	122902
9	1515	9	35	2873	11	9916
10	2292	6	82	4686	25	50476
11	3507	1	9	5569	38	2621
12	4879	2	22	8796	63	12870
13	∞	-	-	-	-	-
14	-	-	-	-	fail	-

Table 4.3: Results from our Modular Algorithm

Sys	Triang.	Mem.	Size	gsolve	Mem.	Size
1	0.4	4	169	0.2	3	239
2	2	6	1680	504	18	34375
3	512	275	6250	1041	34	27624
4	2.5	4	743	-	fail	-
5	5	5	3134	9	5	2236
6	3000	250	2695	4950	66	34932
7	-	fail	-	1050	31	31115
8	-	fail	-	-	error	-
9	1593	18	55592	-	fail	-
10	∞	-	-	-	fail	-
11	-	fail	-	-	fail	-
12	-	fail	-	-	fail	-
13	-	fail	-	∞	-	-
14	-	fail	-	-	fail	-

Table 4.4: Results from Triangularize and gsolve

4.5 An Application of Equiprojectable Decomposition: Automatic Case Distinction and Case Recombination

In the Maple session below, we first compute the lower echelon form and the inverse (when it exists) of two matrices `m` modulo a regular chain `rc`. This regular chain defines a tower of simple extensions with zero-divisors. Each matrix `m` leads to two cases. The first matrix `m` is invertible in one case, but not in the other. The second matrix `m` is invertible in both cases and the corresponding answers are recombined. This recombination feature of the `RegularChains` library is a by-product of the notion of the *equiprojectable decomposition*. In addition, this examples illustrate the capabilities of the `RegularChains` library for solving linear systems over non-integral domains. The detailed algorithms are listed afterwards.

```
>   R := PolynomialRing([y,z]);
```

$$R := polynomial\_ring$$

```
>   rc := Chain([z^4 +1, y^2 -z^2], Empty(R), R);
```

$$rc := regular\_chain$$

```
>   Equations(rc, R);
```

$$[y^2 - z^2, z^4 + 1]$$

```
>   m := Matrix([[1, y+z], [0, y-z]]);
```

$$m := \begin{bmatrix} 1 & y+z \\ 0 & y-z \end{bmatrix}$$

```
>   lm := LowerEchelonForm(m, rc, R);
```

$$lm := [[\begin{bmatrix} -2z & 0 \\ 0 & y-z \end{bmatrix}, regular\_chain], [\begin{bmatrix} 0 & 0 \\ 1 & y+z \end{bmatrix}, regular\_chain]]$$

```
>   Equations(lm[1][2], R); Equations(lm[2][2], R);
```

$$[y + z, z^4 + 1]$$
$$[y - z, z^4 + 1]$$

```
>   MatrixInverse(m,rc,R);
```

$$[[\begin{bmatrix} 1 & 0 \\ 0 & 1/2\,z^3 \end{bmatrix}, regular\_chain]], [[\text{``no Inverse''}, \begin{bmatrix} 1 & y+z \\ 0 & y-z \end{bmatrix}, regular\_chain]]]$$

```
>   m := Matrix([[1, y+z], [2, y-z]]);
```

$$m := \begin{bmatrix} 1 & y+z \\ 2 & y-z \end{bmatrix}$$

```
>  lm := LowerEchelonForm(m, rc, R);
```

$$lm := [[\begin{bmatrix} -2\,z & 0 \\ 2 & y-z \end{bmatrix}, regular\_chain], [\begin{bmatrix} 4\,z & 0 \\ 1 & y+z \end{bmatrix}, regular\_chain]]$$

```
>  Equations(lm[1][2], R); Equations(lm[2][2], R);
```

$$[y+z, z^4+1]$$
$$[y-z, z^4+1]$$

```
>  mim := MatrixInverse(m,rc,R);
```

$$mim := [[[\begin{bmatrix} 1 & 0 \\ -z^3 & 1/2\,z^3 \end{bmatrix}, regular\_chain], [\begin{bmatrix} 0 & 1/2 \\ -1/2\,z^3 & 1/4\,z^3 \end{bmatrix}, regular\_chain]], []]$$

```
>  m1 := mim [1][1][1]:   rc1 := mim [1][1][2]:   Equations(rc1, R);
```

$$[y+z, z^4+1]$$

```
>  m2 := mim [1][2][1]:   rc2 := mim [1][2][2]:   Equations(rc2, R);
```

$$[y-z, z^4+1]$$

```
>  mc := MatrixCombine([rc1, rc2], R, [m1, m2]);
```

$$mc := [[\begin{bmatrix} 1/2\,z^3 y + 1/2 & -1/4\,z^3 y + 1/4 \\ -3/4\,z^3 + 1/4\,z^2 y & 3/8\,z^3 - 1/8\,z^2 y \end{bmatrix}, regular\_chain]]$$

```
>  Equations(mc[1][2], R);
```

$$[y^2 - z^2, z^4+1]$$

```
>  MatrixMultiply(mc[1][1], m, mc[1][2], R);
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

It is likely to have several cases in the output of `MatrixCombine`. In the example below, we generate four random matrices. When re-combining the four cases, we cannot obtain a single recombination. Since the two ideals generated by `rc1` and `rc2` are obviously relatively prime (no common roots in `z`), if we try to recombine them, we create a polynomial in `y` with a zero-divisor as a leading coefficient. This is forbidden by the properties of a regular chain, as shown by the followed check.

```
>  R := PolynomialRing([x,y,z]);
```

$$R := polynomial\_ring$$

```
>  sys := {x^2 + y + z - 1, x + y^2 + z - 1, x + y + z^2 - 1};
```

$$sys := \{x + y + z^2 - 1, x + y^2 + z - 1, x^2 + y + z - 1\}$$

```
>  lrc := Triangularize(sys,R, normalized=yes);
>  map(Equations, lrc, R);
```

$$lrc := [regular\_chain, regular\_chain, regular\_chain, regular\_chain]$$

$$[[x - 1, y, z], [x, y - 1, z], [x, y, z - 1], [x - z, y - z, z^2 + 2z - 1]]$$

```
>  lm := [seq( Matrix ( [ seq([ seq(randpoly([x,y,z],degree=1),j=1..2)],
i=1..2)]), k=1..4)];
```

$$lm := [\begin{bmatrix} -7 + 22\,x - 55\,y - 94\,z & 87 - 56\,x - 62\,z \\ 97 - 73\,x - 4\,y - 83\,z & -10 + 62\,x - 82\,y + 80\,z \end{bmatrix},$$

$$\begin{bmatrix} -44 + 71\,x - 17\,y - 75\,z & -10 - 7\,x - 40\,y + 42\,z \\ -50 + 23\,x + 75\,y - 92\,z & 6 + 74\,x + 72\,y + 37\,z \end{bmatrix},$$

$$\begin{bmatrix} -23 + 87\,x + 44\,y + 29\,z & 98 - 23\,x + 10\,y - 61\,z \\ -8 - 29\,x + 95\,y + 11\,z & -49 - 47\,x + 40\,y - 81\,z \end{bmatrix},$$

$$\begin{bmatrix} 91 + 68\,x - 10\,y + 31\,z & -51 + 77\,x + 95\,y + z \\ 1 + 55\,x - 28\,y + 16\,z & 30 - 27\,x - 15\,y - 59\,z \end{bmatrix}]$$

```
>  clr := MatrixCombine(lrc, R, lm);
```

$$clr := [[\begin{bmatrix} -76\,y + 15 & -81\,y + 31 \\ y + 24 & 26\,y + 52 \end{bmatrix}, regular\_chain],$$

$$[\begin{bmatrix} 178 - 85\,z - 87\,z^2 & -17/2 + 88\,z - \frac{85}{2}z^2 \\ \frac{43}{2} + 2\,z - \frac{41}{2}z^2 & \frac{119}{2} - 160\,z - \frac{59}{2}z^2 \end{bmatrix}, regular\_chain]]$$

```
>  rc1 := clr[1][2]; Equations(rc1, R); rc2 :=
clr[2][2];Equations(rc2, R);
```

$$rc1 := regular\_chain$$

$$[x + y - 1, y^2 - y, z]$$

$$rc2 := regular\_chain$$

$$[2\,x + z^2 - 1, 2\,y + z^2 - 1, z^3 + z^2 - 3\,z + 1]$$

```
>  Rz := PolynomialRing([z]); rc := Empty(Rz); rc1 :=
Chain([z],rc,Rz); rc2 := Chain([z^3 + z^2 - 3*z + 1], rc, Rz);
m1 := Matrix([[y^2 - y]]); m2 := Matrix([[2*y +z^2  -1]]); clr :=
MatrixCombine([rc1, rc2], Rz, [m1, m2]);
```

```
>  m := clr[1][1]; rc := clr[1][2]; p := m[1,1];
   init := Initial(p,R); invInit:=Inverse(init,rc,Rz);
   Equations(invInit[1][1][3],R);map(Equations, invInit[2], R);
```

$$Rz := polynomial\_ring$$

$$rc := regular\_chain$$

$$rc1 := regular\_chain$$

$$rc2 := regular\_chain$$

$$m1 := \left[\ y^2 - y\ \right]$$

$$m2 := \left[\ 2\,y + z^2 - 1\ \right]$$

$$clr := \left[\left[\left[\ 9\,yz + z^3 - 3\,z - 3\,yz^3 - 3\,yz^2 + 2\,z^2 + z^3y^2 + z^2y^2 - 3\,zy^2 + y^2 - y\ \right],\right.\right.$$
$$regular\_chain]]$$

$$m := \left[\ 9\,yz + z^3 - 3\,z - 3\,yz^3 - 3\,yz^2 + 2\,z^2 + z^3y^2 + z^2y^2 - 3\,zy^2 + y^2 - y\ \right]$$

$$rc := regular\_chain$$

$$p := 9\,yz + z^3 - 3\,z - 3\,yz^3 - 3\,yz^2 + 2\,z^2 + z^3y^2 + z^2y^2 - 3\,zy^2 + y^2 - y$$

$$init := z^3 + z^2 - 3\,z + 1$$

$$invInit := [[[1, 1, regular\_chain]], [regular\_chain, regular\_chain]]$$

$$[z]$$

$$[[z^2 + 2\,z - 1], [z - 1]]$$

Algorithm 33 Matrix Combine

Input rc_list: list of 0-dimensional and strongly normalized regular chains in $k[X_1, \ldots, X_n]$.

lm: list of matrices with polynomial entries in $k[X_1, \ldots, X_n]$, and they have the same dimension.

Output the equiprojectable decomposition of the variety given by rc_list, and the corresponding combined matrices.

MatrixCombine(rc_list, lm) ==

1: $(lrc, BezoutCoeffsTable) \leftarrow$ RemoveCriticalPair(rc_list);

   #Bézout coefficients are recorded when gcds are computed for removing critical pairs by Algorithm 26

2: $lm' \leftarrow$ project(lm, lrc);

   #reduce each corresponding matrix w.r.t its refined regular chains

3: $(new\_lrc, new\_lm) \leftarrow$ Merge($lrc, BezoutCoeffsTable, lm'$);

   #merged by Algorithm 27

4: **return** (new_lrc, new_lm);

Algorithm 34 Lower Echelon Form Modulo a Regular Chain

    Input rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
           A: matrix of dimension $m \times c$ with polynomial entries in $k[X_1, \ldots, X_n]$.

   Output list of cases $[\ldots, [B_i, T_i], \ldots]$ such that B_i is the lower echelon form of A
        modulo T_i.

LowerEchelonForm$(rc, A) ==$

1: $case\_list \leftarrow [["todoEchelon", rc, A]]; result \leftarrow [\ ];$
2: **while** case_list not empty **do**
3:    $case \leftarrow$ Pop out one element from $case\_list$;
4:    $flag \leftarrow case[1]; rc' \leftarrow case[2]; A' \leftarrow case[3];$
5:    **if** $flag =$"$doneEchelon$" **then**
6:      $result \leftarrow result \ \cup \ [A', rc'];$
7:    **else if** $flag =$"$todoEchelon$" **then**
8:      $intercase\_list \leftarrow$ innerLowerEchelonFormModuloRC$(rc', A');$
9:      $case\_list \leftarrow case\_list \ \cup \ intercase\_list;$
10:    **end if**
11: **end while**
12: **return** $result$;

Algorithm 35 Normal Form of a Matrix

    Input rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
           A: matrix of dimension $m \times c$ with polynomial entries in $k[X_1, \ldots, X_n]$.

   Output fraction normal form of A modulo the saturated ideal of rc.

NormalFormMatrix$(rc, A) ==$

1: $B \leftarrow$ zero matrix of dimension $m \times c$;
2: **for** i from 1 to m **do**
3:    **for** j from 1 to c **do**
4:      $B[i, j] \leftarrow$ NormalForm$(B[i, j], rc);$
5:    **end for**
6: **end for**
7: **return** B;

Algorithm 36 Matrix Inverse Modulo a Regular Chain

Input A: square matrix of dimension m with polynomial entries in $k[X_1, \ldots, X_n]$.
rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.

Output list of cases $[A_i, T_i]$ or $["noInverse", B_i, rc_i]$ such that A_i is the inverse of A modulo T_i, or B_i has no inverse under rc_i.

MatrixInverse(rc, A) ==
1: $inverse\_list \leftarrow [\,]$; $noinverse\_List \leftarrow [\,]$; $echelon\_result \leftarrow [\,]$;
2: $case\_list \leftarrow [["todoEchelon", rc, A]]$;
3: **while** $case\_list$ is not empty **do**
4:    $case \leftarrow$ Pop up one element from $case\_list$;
5:    $flag \leftarrow case[1]$; $rc' \leftarrow case[2]$; $A' \leftarrow case[3]$;
6:    **if** $flag$ is "$doneEchelonInverse$", or "$noInverse$", or "$zeroDivisorInEchelon$" **then**
7:       $echelon\_result \leftarrow echelon\_result \cup case$;
8:    **else if** flag is "$todoEchelon$" **then**
9:       $echcase\_list \leftarrow$ LowerEchelonFormForInverseModuloRC(rc', A');
10:      $case\_list \leftarrow case\_list \cup echcase\_list$;
11:    **else if** flag is "$todoEchelonInverse$" **then**
12:      $A_e \leftarrow case[4]$; $A_t \leftarrow case[5]$;
13:      $invEchcase\_list \leftarrow$ InverseLowerEchelonFormModuloRC(rc', A', A_e, A_t);
14:      $case\_list \leftarrow case\_list \cup invEchcase\_list$;
15:    **end if**
16: **end while**
17: **while** $echelon\_result$ list is not empty **do**
18:    $case \leftarrow$ Pop up one case from $echelon\_result$ list;
19:    **if** $case$'s flag is "$noInverse$", or "$zeroDivisorInEchelon$" **then**
20:      $noinverse\_List \leftarrow noinverse\_List \cup case$;
21:    **else**
22:      $rc' \leftarrow case[2]$; $invA_e \leftarrow case[3]$; $A_t \leftarrow case[4]$;
23:      $inverse\_A \leftarrow$ MatrixMatrixMultiplyModuloRC($rc', invA_e, A_t$);
24:      $inverse\_list \leftarrow inverse\_list \cup [inverse\_A, rc']$;
25:    **end if**
26: **end while**
27: **return** $[inverse\_list, noinverse\_List]$;

Algorithm 37 Inner Lower Echelon Form Modulo a Regular Chain

> **Input** rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
> A: matrix of dimension $m \times c$ with polynomial entries in $k[X_1, \ldots, X_n]$.
>
> **Output** list of cases as $[flag, T_i, A_i]$; $flag = "todoEchelon"$, or $"doneEchelon"$. If $flag = "doneEchelon"$, A_i is the lower echelon form of A mod T_i.

innerLowerEchelonFormModuloRC$(rc, A) ==$

1: #use Bareiss single step fraction-free Gaussian elimination, but computation may split by the D5 principle
2: $case\_list \leftarrow [\ ]$; $A_e \leftarrow$ copy of A;
3: $r \leftarrow m$; $sign \leftarrow 1$; $divisor\_inverse \leftarrow 1$;
4: **for** k from c to 1 by -1 **do**
5:     **if** $r < 1$ **then**
6:       **break**;
7:     **else**
8:       $hasNonZeroPivot, sign, A_e \leftarrow$ GetNonZeroPivot$(rc, A_e, k, sign)$;
9:       **if** $hasNonZeroPivot = true$ **then**
10:         $(inverse\_list,\ zerodivisor\_list) \leftarrow$ InverseModuloRC$(A_e[r, k], rc)$;
           #may split by the D5 principle
11:         **if** number of $inverse\_list = 1$ and $zerodivisor\_list$ is empty **then**
12:           **for** i from $r - 1$ to 1 by -1 **do**
13:             **for** j from $k - 1$ to 1 by -1 **do**
14:               $A_e[i, j] \leftarrow (A_e[r, k]A_e[i, j] - A_e[r, j]A_e[i, k]) \times divisor\_inverse$;
15:               $A_e[i, j] \leftarrow$ NormalForm$(A_e[i, j], rc)$;
16:             **end for**
17:             $A_e[i, k] \leftarrow 0$;
18:           **end for**
19:           Get inv from $inverse\_list$ s.t. inv is the inverse of $A_e[r, k]$ mod rc;
20:           $divisor\_inverse \leftarrow$ NormalForm(inv, rc);
21:           $r \leftarrow r - 1$;
22:         **else**
23:           **for** regular chain rc' in $inverse\_list$ or in $zerodivisor\_list$ **do**
24:             $case\_list \leftarrow case\_list \cup ["todoEchelon", rc', A]$;
25:           **end for**
26:           **return** $case\_list$;
27:         **end if**
28:       **end if**
29:     **end if**
30: **end for**
31: $case\_list \leftarrow [["doneEchelon", rc, A_e]]$;
32: **return** $case\_list$;

Algorithm 38 Lower Echelon Form for Inverse Modulo a Regular Chain

    Input A: square matrix of dimension m with polynomial entries in $k[X_1, \ldots, X_n]$.
            rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.

   Output list of cases; a case can be ["$noInverse$", A, rc], or ["$todoEchelon$", A, T_i],
           or ["$todoEchelonInverse$", rc, A, A_e, A_t] where A_e is the lower echelon form
           of A modulo rc, A_t is the companion matrix following the transformation for
           generating A_e.

LowerEchelonFormForInverseModuloRC$(rc, A) ==$

1: #use Bareiss single step fraction-free Gaussian elimination, but computation may split by the D5 principle
2: $case\_list \leftarrow [\ ]$; $A_e \leftarrow$ copy of A; $A_t \leftarrow$ identity matrix of dimension m;
3: $r \leftarrow m$; $sign \leftarrow 1$; $divisor\_inverse \leftarrow 1$;
4: **for** k from m to 1 by -1 **do**
5:    **if** $r < 1$ **then**
6:      break;
7:    **else**
8:      $hasNonZeroPivot, sign, A_e =$ GetNonZeroPivot$(rc, A_e, k, sign)$;
9:      **if** $hasNonZeroPivot = false$ **then** [["$noInverse$", A, rc]];
10:     $(inverse\_list, \ zerodivisor\_list) \leftarrow$ InverseModuloRC$(A_e[r, k], rc)$;
          #may split by the D5 principle
11:     **if** number of $inverse\_list = 1$ and $zerodivisor\_list$ is empty **then**
12:       **for** i from $r - 1$ to 1 by -1 **do**
13:         **for** j from $k - 1$ to 1 by -1 **do**
14:           $A_e[i, j] \leftarrow (A_e[r, k] \times A_e[i, j] - A_e[r, j] \times A_e[i, k]) \times divisor\_inverse$;
15:           $A_e[i, j] \leftarrow$ NormalForm$(A_e[i, j], rc)$;
16:         **end for**
17:         $A_e[i, k] \leftarrow 0$;
18:         **for** s from m to 1 by -1 **do**
19:           $A_t[i, s] \leftarrow (A_e[r, k] \times A_t[i, s] - A_t[r, s] \times A_e[i, k]) \times divisor\_inverse$;
20:           $A_t[i, s] \leftarrow$ NormalForm$(A_t[i, s], rc)$;
21:         **end for**
22:       **end for**
23:       Get inv from $inverse\_list$ s.t. inv is the inverse of $A_e[r, k] \mod rc$;
24:       $divisor\_inverse \leftarrow$ NormalForm(inv, rc);
25:       $r \leftarrow r - 1$;
26:     **else**
27:       **for** regular chain rc' in $inverse\_list$ or in $zerodivisor\_list$ **do**
28:         $case_list \leftarrow case_list \ \cup \ $["$todoEchelon$", A, rc'];
29:       **end for**
30:       **return** $case\_list$;
31:     **end if**
32:    **end if**
33: **end for**
34: $case\_list \leftarrow$ ["$todoEchelonInverse$", rc, A, A_e, A_t];
35: **return** $case\_list$;

Algorithm 39 Inverse Lower Echelon Form Modulo a Regular Chain

Input rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
A: square matrix of dimension m with polynomial entries in $k[X_1, \ldots, X_n]$.
A_e: lower echelon form of A modulo rc.
A_t: companion matrix following the transformation for generating A_e.

Output list of tasks as $["zeroDivisorInEchelon", T_i, A]$, or $["todoEchelon", T_i, A]$, or $["doneEchelonInverse", rc, invA_e, A_t]$ where $invA_e$ is the inverse of A_e modulo rc.

InverseLowerEchelonFormModuloRC(rc, A, A_e, A_t) ==
1: $invA_e \leftarrow$ zero matrix of dimension m;
2: **for** i from 1 to m **do**
3:    $(inverse\_list,\ zerodivisor\_list) \leftarrow$ InverseModuloRC$(A_e[i, i], rc)$;
     #may split by the D5 principle
4:    **if** number of $inverse\_list = 1$ and $zerodivisor\_list$ is empty **then**
5:       Get inv from $inverse\_list$ s.t. inv is the inverse of $A_e[i, i]$ mod rc;
6:       $invA_e[i, i] \leftarrow$ NormalForm(inv, rc);
7:    **else**
8:       **for** regular chain rc' in $inverse\_list$ or in $zerodivisor\_list$ **do**
9:         $case\_list \leftarrow case\_list \cup ["todoEchelon", rc', A]$;
10:      **end for**
11:      **return** $case\_list$;
12:    **end if**
13: **end for**
14: **for** j from 1 to $m - 1$ **do**
15:    **for** i from $j + 1$ to m **do**
16:      $temp \leftarrow 0$;
17:      **for** k from j to $i - 1$ **do**
18:        $temp \leftarrow temp + A_e[i, k] \times invA_e[k, j]$;
19:      **end for**
20:      $invA_e[i, j] \leftarrow -invA_e[i, i] \times temp$;
21:      $invA_e[i, j] \leftarrow$ NormalForm$(invA_e[i, j], rc)$
22:    **end for**
23: **end for**
24: **return** $["doneEchelonInverse", rc, invA_e, A_t]$;

Algorithm 40 Matrix Matrix Multiply Modulo a Regular Chain

> **Input** rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
> A: matrix of dimension $m \times r$ with polynomial entries in $k[X_1, \ldots, X_n]$.
> B: matrix of dimension $r \times l$ with polynomial entries in $k[X_1, \ldots, X_n]$.

> **Output** product of A and B modulo the saturated ideal of rc

MatrixMatrixMultiplyModuloRC$(rc, A, B) ==$
1: $D \leftarrow$ zero matrix of dimension $m \times l$;
2: **for** i from 1 to m **do**
3:   **for** k from 1 to l **do**
4:     $f \leftarrow 0$;
5:     **for** j from 1 to r **do**
6:       $f_i \leftarrow A[i, j] \times B[j, k]$;
7:       $f \leftarrow f + f_i$;
8:     **end for**
9:     $D[i, k] \leftarrow$ NormalForm(f, rc);
10:   **end for**
11: **end for**
12: **return** D;

Algorithm 41 Get Non-Zero Pivot

> **Input** rc: 0-dimensional regular chain in $k[X_1, \ldots, X_n]$.
> A: matrix of dimension $m \times c$ with polynomial entries in $k[X_1, \ldots, X_n]$.
> k: the number of starting column.
> $sign$: effect of switching rows.

> **Output** $[flag, sign', A']$; $flag$ is true if non-zero pivot is found, and $A'[k, k]$ is the non-zero pivot; otherwise $flag$ is false.

GetNonZeroPivot$(rc, A, k, sign) ==$
1: $hasNonZeroPivot \leftarrow true$; $A' \leftarrow$ copy of A;
2: **if** $A'[k, k]$ is zeor modulo rc **then**
3:   $hasNonZeroPivot \leftarrow false$;
4:   **for** p from k to 1 by -1 **do**
5:     **if** $A'[p, k]$ is nonzero modulo rc **then**
6:       $hasNonZeroPivot \leftarrow true$;
7:       $A' \leftarrow$ switch the pth row and the kth row of A';
8:       **if** $p <> k$ **then**
9:         $sign' \leftarrow -sign$;
10:       **end if**
11:     **end if**
12:   **end for**
13: **end if**
14: **return** $[hasNonZeroPivot, sign', A']$;

4.6   Summary

We have presented a modular algorithm for triangular decompositions of 0-dimensional varieties over $\mathbb{Q}$ and have demonstrated the feasibility of Hensel lifting in computing triangular decompositions of non-equiprojectable varieties. Experiments show the capacity of this approach to improve the practical efficiency of triangular decomposition. The implementation of these methods brings to the Maple algebra system a unique module for automatic case discussion and case recombination.

By far, the bottleneck is the modular triangularization phase. This is quite encouraging, since it is the part for which we relied on generic, non-optimized code. The next step is to extend these techniques to specialize variables as well during the modular phase, following the approach initiated in [63] for primitive element representations, and treat systems of positive dimension.

Chapter 5

Component-level Parallelization of Triangular Decompositions

In this chapter, we study the parallelization of algorithms for solving polynomial systems symbolically by way of triangular decompositions. We introduce a component-level parallelism for which the number of processors in use depends on the geometry of the solution set of the input system. Our long-term goal is to achieve an efficient multi-level parallelism: coarse grained (component) level for tasks computing geometric objects in the solution sets, and medium/fine grained level for polynomial arithmetic such as GCD/resultant computation within each task.

Component-level parallelization of triangular decompositions belongs to the class of dynamic irregular parallel applications, which leads us to address the following question: How to exploit geometrical information at an early stage of the solving process that would be favorable to parallelization? We report on the effectiveness of the approaches that we have applied, including "modular methods", "solving by decreasing order of dimension", "task pool with dimension and rank guided scheduling". We have extended the ALDOR programming language to support multiprocessed parallelism on SMPs and realized a preliminary implementation. Our experimentation shows promising speedups for some well-known problems and proves that our component-level parallelization is efficient in practice. We expect that this speedup would add a multiplicative factor to the speedup of medium/fine grained level parallelization as parallel GCD and resultant computations.

5.1   Introduction

This work aims at investigating new directions in the parallelization of symbolic solvers for polynomial systems. Ideally, one would like that each component of the solution set of a polynomial system could be produced by an independent processor, or a set of processors. In practice, the input polynomial system, which is hiding those components, requires some transformations in order to split the computations into subsystems and, then, lead to the desired components. The efficiency of this approach depends on its ability to detect and exploit geometrical information during the solving process. Its implementation, which involves parallel symbolic computations, is yet another challenge.

Several symbolic algorithms provide a decomposition of the solution set of any system of algebraic equations into components (which may be irreducible or with weaker properties): primary decompositions [62, 122], comprehensive Gröbner bases [144], triangular decompositions [146, 76, 86, 108, 140] and others. These algorithms tend to split the input polynomial system into subsystems and, therefore, seem to be natural candidates for a *component-level* parallelization.

Unfortunately, such a parallelization is very likely to be unsuccessful, bringing no practical speedup w.r.t. comparable sequential implementations of the same algorithms. Indeed, even if computations split into sub-problems which can be processed concurrently, the computing cost of the corresponding tasks are extremely irregular. Even worse: for input polynomial systems with coefficients in the field $\mathbb{Q}$ of rational numbers, a single heavy task may dominate the whole solving process, leading essentially to no opportunities for component-level parallel execution. This phenomenon follows from the following observation. For most polynomial systems with coefficients in $\mathbb{Q}$ that arise in theory or in practice, see for instance `www.SymbolicData.org`, the solution set can be described by a single component! The theoretical justification is given by the celebrated *Shape Lemma* [12] for systems with finitely many solutions.

We show, in this work, how to achieve a successful *component-level* parallelization for polynomial systems including for the case of rational number coefficients. Among the algorithms that decompose the solution set of a polynomial system into components, we consider Triade [108] for computing triangular decompositions. This algorithm was presented in Section 2.7.

The first reason for this choice is that, triangular decompositions of polynomial systems with coefficients in $\mathbb{Q}$ can be reduced to triangular decompositions of polynomial systems modulo a prime number, as discussed in Chapter 4. This brings

rich opportunities for parallel execution. The second reason is that the Triade Algorithm has been implemented in the ALDOR language [3] and in the computer algebra systems AXIOM [72] and MAPLE [105] as the `RegularChains` library [90]. This provides us with useful tools for our experimentation work. The third and main reason is that the Triade Algorithm can generate the (intermediate or output) components by decreasing order of dimension. As we show in Section 5.2, this allows us to exploit the opportunities for parallel execution created by modular techniques, leading to successfully component-level parallel execution.

Our objective is to develop a parallel solver for which the number of processors in use depends on the *intrinsic complexity* of the input system, that is, on the geometry of its solution set. This approach is not aimed to bring scalability. For instance, for systems over $\mathbb{Z}/p\mathbb{Z}$ with finitely many solutions, if the output consists of s components with similar degrees, we cannot expect a speed-up much larger than s by relying only on a component-level parallelism. We do not aim either at replacing the previous parallel approaches at the level of polynomial reduction (or simplification) in Buchberger's Algorithm for computing Gröbner bases and in the Characteristic Set Method of Wu. On the contrary, we aim at adding an extra level of parallelism.

The parallelization of two other algorithms for solving polynomial systems symbolically have already been actively studied. First, Buchberger's Algorithm (Algorithm 2 p. 27) for computing Gröbner bases, see for instance [23, 27, 29, 52, 7, 94]. Second, the Characteristic Set Method (Algorithm 4 p. 33) of Wu [146], see [2, 147, 148]. In all these works, the parallelized operation is polynomial reduction (or simplification). In both Algorithm 2 and Algorithm 4, this is Step (R). More precisely, given two polynomial sets A and B (with some conditions on A and B, depending on the algorithm) the reductions of the elements of A by those of B are executed in parallel.

The Triade algorithm also has a *polynomial simplification level* which relies on polynomial GCDs and resultants. The parallelization of such computations is reported in [115, 70]. The addition of this second level to the Triade algorithm is work in progress. We also plan to integrate a third and fine grained level, for polynomial arithmetic operations modulo a triangular set. In [97] the authors report on a successful multithreaded implementation of arithmetic operations modulo a triangular set. This would be a good foundational library for the linear algebra, and thus the polynomial GCDs and resultants needed in triangular decompositions.

In Section 2.7, we presented the *task model* employed by the Triade algorithm. We review also how this algorithm makes use of geometrical information discovered during the computations.

We describe in Section 5.2 the techniques that we have developed to create and exploit component-level parallelism. Our heuristically efficient *Task Pool with Dimension and Rank Guided scheduling* (TPDRG) is reported in Section 5.3. In the remaining of this chapter, we report on our preliminary implementation and experimentation. We have extended the ALDOR programing language for multi-processed parallel programing on SMPs and realized a preliminary implementation of this component-level parallel algorithm based on the `BasicMath` library [132]. We have conducted an intensive experimentation on some well-known problems. A comparison on the practical efficiency between our TPDRG scheduling and the generally good *Greedy* scheduling has also been performed. These help in evaluating the efficiency of our implementation and reveal its limitation as well. In the conclusion, we discuss the potential to extend this work to achieve efficient multi-level parallelization for triangular decompositions.

5.2   Parallelization

In Section 2.7, we discussed the main features of the Triade algorithm [108] with an emphasis on those that are relevant to parallel execution. The notions of a *Task*, Definition 2.7.1 and that of a *delayed split*, Definition 2.7.3 play a central role. Algorithm 5 is the top-level procedure of the Triade algorithm: it manages a *task pool*. The tasks are transformed by means of a sub-procedure (Algorithms 6 and 7) which is dedicated to "simple tasks". The execution of such a simple task can be highly irregular and dynamic. It can also generate other tasks.

One could think of deriving a parallel scheme from Algorithm 5 by running the procedure Triangularize(F, T) on one processor and running each call to Decompose on any other available processors, following a greedy scheduling. As mentioned in the Introduction, such parallelization is very likely to be unsuccessful, bringing no practical speed-up w.r.t. comparable sequential implementations of the same algorithms. In characteristic zero, this mainly follows from the fact, for most polynomial systems, the solution set can be described by a single component, though not necessarily irreducible. In prime characteristic, however, even if a single component suffices, it is more likely that polynomials factorize and thus that components split.

Using modular techniques. The previous remark suggests the use of modular techniques for computing triangular decompositions. We rely on the algorithm proposed in [39]. For a given input square system $F \subset \mathbb{Q}[x_1, \ldots, x_n]$ this algorithm computes the simple points of $V(F)$ in four steps:

Sys	Name	n	d	p	Degrees
1	eco6	6	3	105761	[1,1,2,4,4,4]
2	eco7	7	3	387799	[1,1,1,1,4,2, 4,4,4,4,4,2]
3	CassouNogues2	4	6	155317	[8]
4	CassouNogues	4	8	513899	[8,8]
5	Nooburg4	4	3	7703	[18,6,6,3,3,4, 4,4,4,2,2,2, 2,2,2,2,2,1, 1,1,1,1]
6	UteshevBikker	4	3	7841	[1,1,1,1,2,30]
7	Cohn2	4	6	188261	[3,5,2,1,2,1,1, 16,12,10,8,8, 4,6,4,4,4,4,2, 1,1,1,1,1,1,1, 1,1,1,1,1,1,1]

Table 5.1: Polynomial Examples and Effect of Modular Computation

(S_1) compute a prime number p such that $V(F)$ can be reconstructed, with high probability, from $V_p := V(F \mod p)$, the zero-set of F regarded in $\mathbb{Z}/p\mathbb{Z}[x_1,\ldots,x_n]$,

(S_2) compute a triangular decomposition of V_p,

(S_3) compute the equiprojectable decomposition of $_p$,

(S_4) reconstruct by Hensel lifting the equiprojectable decomposition of $V(F)$.

As reported in [39], the second step has the dominant cost. Therefore, we focus on computing triangular decompositions of polynomial systems with coefficients in a finite field.

Our test suite. Table 5.1 contains data about 7 well-known test systems that we use through the experiments reported in this article. All of them are polynomial systems over $\mathbb{Q}$: for each we give its number of n equations, its total degree d, the prime number p provided by the above Step (S_1) and the list of the degrees of the triangular decomposition computed at Step (S_2) by the Triade algorithm. We emphasize the fact that each of these systems, except for Cohn2, is equiprojectable, that is, its equiprojectable decomposition consists of a single component. Hence, for a direct computation over $\mathbb{Q}$, the computations may not split. Therefore, our modular approach has created opportunities for parallel execution.

Regularizing initials for controlling task irregularity. A call to the procedure Decompose as given by Algorithm 7 may result in unpredictable amount of work. Indeed, since the initial of p may not be regular w.r.t. $\mathrm{Sat}(T)$, the polynomial f computed at Line 2 may have a different main variable than p. Hence we cannot predict the main variables and degrees of the input polynomials in the calls to AlgebraicDecompose and Extend. It could be the case that these calls lead to inexpensive operations, say polynomial GCDs of univariate polynomials of low degrees, whereas the regular chain contains very large polynomials in many variables. Therefore, Decompose(p, T) may lead to inexpensive computations but expensive data communication. In order to control this phenomenon, we strengthen the notion of a task in Definition 5.2.1: the **initial** of every polynomial $f \in F$ in a task $[F, T]$ must be **regular** w.r.t. $\mathrm{Sat}(T)$. The motivation of this Definition is twofold. First, we want to anticipate which operations will be performed by Algorithm 7. Second, we want to force light-load calls to Decompose(p, T) to be performed inside heavily-load calls. Reaching the former goal is discussed after Definition 5.2.1 while the latter one is achieved by the *Split-by-height* strategy presented at the end of this section.

Definition 5.2.1. The task $[F, T]$ is *standard* if for all $f \in F$, when modulo $\mathrm{Sat}(T)$, f is not constant and its initial is regular w.r.t. $\mathrm{Sat}(T)$.

Estimating the cost of tasks. Assume from now on that every task in Algorithm 5 is standard. When the polynomial p is chosen at Line 6, we know which operations will be performed by the call Decompose(p, U_1). Indeed, if the initial of p is regular w.r.t. $T := U_1$, Line 1 in Algorithm 7 is useless and we know that $f = p$ holds. Let v be mvar(p). Therefore, two cases arise: either v is algebraic w.r.t. T and $\mathsf{GCD}(C_v, p, C_{<v})$ is called and its cost can be estimated (see for instance [41] for complexity estimates); or v is not algebraic w.r.t. T and Extend($T \cup \{p\}$) is called, leading again to GCD computations with predictable costs.

The *Split-by-height* strategy. Let $[F, E]$ be a task. We introduce a new procedure, called SplitByHeight(F, E), returning a delayed split of $[F, E]$ with the following requirement: If $[G, U]$ is a task returned by SplitByHeight(F, T) and $|U| = |T|$ holds then $G = \emptyset$ holds. An algorithm for SplitByHeight(F, T) is easily derived from Algorithm 5 and Proposition 2.7.7, leading to Algorithm 42 below.

Based on SplitByHeight we derive a new implementation of Triangularize(F, T), given as Algorithm 43.

Two benefits are obtained from Algorithm 43 in view of parallelization. Assume that at each iteration of the **while** loop all tasks with maximum priority are executed

---

Algorithm 42 Split by Height

> **Input** a task $[F, T]$.

> **Output** a delayed split of $[F, T]$ such that for all output task $[G, U]$ either $|U| > |T|$ holds, or both $|U| = |T|$ and $G = \emptyset$ hold.

SplitByHeight$(F, T) == $ **generate**

1: $R := [[F, T]]$ # R is a list of tasks
2: **while** $R \neq [\,]$ **do**
3:     $[F_1, U_1] \leftarrow$ choose and remove a task from R
4:     **if** $|U_1| > |T|$ **then yield** $[F_1, U_1]$
5:     **if** $F_1 = \emptyset$ **then yield** $[F_1, U_1]$
6:     Choose a polynomial $p \in F_1$; $G_1 := F_1 \setminus \{p\}$
7:     **if** $p \equiv 0 \mod \mathrm{Sat}(U_1)$ **then**
8:        $R := \mathsf{cons}\,([G_1, U_1], R)$
9:     **end if**
10:     **for** $[H, T] \in \mathsf{Decompose}(p, U_1)$ **do**
11:        $R := \mathsf{cons}\,([G_1 \cup H, T], R)$
12:     **end for**
13: **end while**

Algorithm 43 Parallel Triangularize

> **Input** a task $[F, T]$.

> **Output** regular chains $T_1, \ldots, T_e$ solving $[F, T]$ in the sense of Definition 2.7.1

Triangularize$(F, T) == $ **generate**

1: $R := [[F, T]]$, # R is a list of tasks
2: **while** $R \neq [\,]$ **do**
3:     Choose and remove $[F_1, U_1] \in R$ with max priority
4:     **if** $F_1 = \emptyset$ **then yield** U_1
5:     **for** $[H, T] \in \mathsf{SplitByHeight}(F_1, U_1)$ **do**
6:        $R := \mathsf{cons}\,([H, T], R)$
7:     **end for**
8:     Sort R by decreasing priority
9: **end while**

---

concurrently. Consequently, at most n (the number of variables) iterations are needed. Indeed, after each call to SplitByHeight, and thus after each parallel step, the minimum height of a regular chain in any unsolved tasks of R has increased at least by one. Therefore, the depth of the task tree is at most n. Moreover, at each node, with high probability, the work load has increased in a significant manner. The *Split-by-height* strategy also respects the original scheme of solving by decreasing order of dimension in favor of removing redundant computations at early stages of the solving process.

5.3   Preliminary Implementation and Experimentation

In the previous section, we showed how to create parallel opportunities at a coarse-grained level by making use of modular methods. Then, we introduced different techniques (standard tasks in Definition 5.2.1 in order to estimate costs, the *Split-by-height* strategy in order to "factorize" the task tree) so as to limit the irregularity of tasks and thus to avoid cheap computations combined expensive data communications.

In this section, we first briefly introduce the framework based on ALDOR [110] that supports this implementation. Then, we present our dynamic "task farming" parallel scheme and our *Task Pool with Dimension and Rank Guided dynamic scheduling* (TPDRG) method, for achieving both load balancing and for removing redundant computing branches at early stages. In the end, we report our experimentation on some well-known problems.

5.3.1   Implementation Scheme

Our preliminary implementation is realized in the high-performance categorical parallel framework reported in [110]. This framework provides a support of multi-processed parallelism in ALDOR on symmetric multiprocessor machines and multi-core processors. It has mechanisms to support dynamic task management, and offers functions for data communication via shared memory segments for parametric data types such as SparseMultivariatePolynomial by serialization. Further more, a sequential implementation [92] of the Triade algorithm has been developed together with the BasicMath library for high performance computing. Many of the categories, domains and packages in this sequential implementation (such as polynomial arithmetic, polynomial GCD and resultant over an arbitrary ring) can be reused or extended for our

purpose. These provide us qualified support for realizing a preliminary implementation of the parallel algorithm in a reasonable period of time.

As discussed in the previous sections, this component-level parallelization of triangular decompositions is dynamic and irregular. We propose to manage the dynamic tasks by a "task farming" scheme, where a *Manager* processor distributes tasks to worker processors. The Manager owns an identifier 0, and it also assigns a unique identifier (`TID`) to each task generated at run time. When a task needs to be processed and a processor is available, the Manager will launch a worker (i.e. process) and pass the `TID` as a command line argument to the worker. The worker takes the task's `TID` as its virtual process identifier to guide its communication with the Manager, as described in [110]. When a worker finishes processing an input task, it sends back to the Manager all output unsolved tasks and writes its solved tasks to its standard output.

Our *task pool with dimension and rank guided dynamic scheduling* is depicted in Figure 5.1. The *task pool* (which can be seen as an implementation of the list R in Algorithms 43) is managed by a *Manager* processor. The Manager first preprocesses the input task $[F, T]$ which generates child tasks. Then, the Manager selects unsolved tasks with maximum priority (See the last paragraph in Section 2.7) determined by the dimension information of the tasks, say $Task1.1$, $Task1.2$ and $Task1.3$, and estimates the cost of each of the selected tasks (See Section 5.2), denoted by $Cost1.1$, $Cost1.2$ and $Cost1.3$, and sorts them decreasingly, say $Cost1.1 \geq Cost1.2 \geq Cost1.3$. Now the manager will launch worker processes if there are processors available and distributes these tasks following this order. Scheduling tasks by the order of decreasing cost aims at obtaining the best trade off between the scheduling overhead and balanced workload, as reported in [129]. When there is only one task in the selection, the Manager will process it by itself. It will process on its own the tasks with very low estimated cost. (See paragraph on *Estimating the cost of tasks* in Section 5.2.)
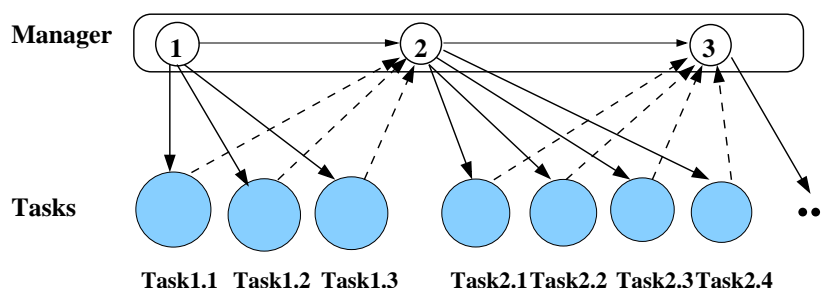

Figure 5.1: Task Pool with Dimension and Rank Guided Dynamic Scheduling

The Manager then proceeds to step 2 to receive the results from the workers for removing redundant components by inclusion test, and then starts another selection, say the $Task2$'s shown in Figure 5.1. The overall solving process follows the decreasing order of dimension, which indicates that the dimension of $Task2$'s is lower than the dimension of $Task1$'s. The Manager repeats this scheduling rule until all the tasks are solved. Other than scheduling, the work load of the Manager is very light and it cannot be a bottleneck. The benefits of using standard tasks (to facilitate cost estimates) and solving by decreasing order of dimension have been discussed near the end of Section 5.2.

To evaluate the effectiveness of our *task pool with dimension and rank guided dynamic scheduling*, we compare below its practical efficiency in our implementation with the *Greedy scheduling* method [68]. In our case, it works as: whenever there is an unsolved task and a free processor, a process is spawned to work on this task. Theoretically, a greedy scheduler is always within a factor of 2 of optimal. However, it cannot ensure the removal of redundant components at an early stage of the solving process.

5.3.2 Experimentation

Our experimentation was accomplished on **Silky** in Canada's Shared Hierarchical Academic Research Computing Network (SHARCNET). Silky is a SGI Altix 3700 Bx2 SMP cluster having 128 Itanium2 Processors (1.6GHz). It is a heavy-loaded multiprogrammed computing resource. The system schedules multi-user's job to run. Usually more than 95% memory is in use and almost all the CPUs are used up. This situation does not allow us to test examples that consumes large amount of memory and explains the level of difficulty of our test-examples.

For each problem listed in Table 5.1, its sequential running time with and without the *regularized initial* condition, imposed by the use of standard tasks (See Definition 5.2.1), are listed in column `noregSeq` and column `regSeq` respectively in Table 5.2. The sequential runs are given by the `Triade` solver [92]. Column `slowBy` is the ratio between these two timings. This result shows that the cost for maintaining the property of standard tasks is negligible (0.01%). Column #P records the number of processors which can give significant speedup to the example's run, that is, beyond it, the increase in the number of processors can not influence significantly its execution time any more. The parallel execution time using this number of processors is recorded in column `SigPar`. The speedup ratio (`SPD`) is calculated by comparing

the parallel execution time with respect to the comparable sequential running time `regSeq`.

Table 5.3 reports on the parallel execution time (wall time) of each problem on a varied number of processors, from 3 to 21. For each run, one processor is always used by the Manager. Thus, given P processors, there are actually $P - 1$ which can be scheduled for workers. The corresponding speedups are reported in Table 5.4. By the scheduling policy of Silky, the number of processors that a program requests are used exclusively by itself. The timing reported here is based on one execution, since all of our testing runs give very close results, with a standard deviation of about 0.001 (s).

These results demonstrate that, for these small and medium-sized problems, our component-level (coarse grained) parallel triangular decomposition implemented in a high-level categorical programming language can gain a speedup from 2 to 6, using a considerably small number of processors (from 5 to 9). This is an encouraging result. Unfortunately, this level of parallelization does not show good scalability. For all these small examples, beyond some limit, the speedup cannot increase when adding more processors. The theoretical results by Attardi and Traverso [7] reveal similar nature for coarse grained parallel Gröbner basis computations. For instance, their theoretical speedup of `cyclic7(-last)` is 11.08 by using 136 processors. Although our parallelism is very different from theirs in terms of mathematical operations, the performance of component-level parallelism for triangular decompositions also depends on the geometrical property of the input system. The speed-up factor is "essentially" bounded by the number of components with "large degrees" in the output of our modular decompositions. For our Systems 1, 2, 4, 6, this number is clearly 3, 6, 2, 1 respectively, which is close to the corresponding speed-up factors 2.1, 6.1, 2.3 and 1.9. For Systems 5 and 7, which have larger ranges of output component degrees, our claim needs to be refined but still gives a good first approximation. System 3 is more subtle: several inconsistent branches explain why we obtain a speed-up of 2.3 with only 1 output component.

In Table 5.5, for each of the problems, we show the minimum parallel running time (in column `TPDRG`) of our parallel implementation using our *TPDRG* scheduling method and the number of processors used for gaining it, denoted by column `#P (A)`. To evaluate the efficiency of our *TPDRG scheduling*, we also implemented a parallel version using the *Greedy scheduling* for comparison. To reveal the influence of the number of processors, we investigate two timings for the *Greedy scheduling* technique. One is using the same number of processors as used in our best parallel run that we noticed. The second one is using 2 more processors than that in column

`#P (A)`. Except for the very small example *eco6*, all other examples show a better timing for the *TPDRG* scheduling method. This proves that our *TPDRG* scheduling is heuristically efficient. It helps effectively removing redundant components, and hence using less CPU time by avoiding working on redundant tasks. On the contrary, the *Greedy scheduling* cannot ensure removing all redundant tasks, in particular, at an early stage of the solving process.

5.4   Summary

We have introduced a component-level parallel algorithm for solving non-linear polynomial systems symbolically by way of triangular decompositions. By using modular methods, we have created opportunities for coarse-grained parallel solving of polynomial systems with rational number coefficients. To exploit these opportunities, we have transformed the Triade algorithm. We have strengthened its notion of a task and replaced the operation Decompose by SplitByHeight in order to reduce the depth of the task tree, create more work at each node, and be able to estimate the cost of each task within each parallel step. This allows us to design a *task pool with dimension and rank guided scheduling* scheme and obtain a heuristically efficient parallelization.

Our preliminary implementation and experimentation demonstrate good performance gain with respect to the comparable sequential solver. We have shown that our multi-processed parallel framework in ALDOR is practically efficient for coarse-grained parallel symbolic computations.

Our long-term goal is to achieve an efficient multi-level parallelism: *coarse grained (component)* level for tasks computing geometric objects in the solution sets, and *medium/fine grained* level for polynomial arithmetic such as GCD/resultant computation within each task. We expect that the speedup in the component level would add a multiplicative factor to the speedup of medium/fine grained level parallelization as parallel GCD/resultant computations. Parallel arithmetic for univariate polynomials over fields is well-developed. We need to extend these methods to multivariate case over more general domains with potential of automatic case discussion. A preliminary work in this direction is reported in [97].

Sys	noregSeq (s)	regSeq (s)	slowBy	#P	SigPara (s)	SPD
1	3.63	4.00	0.01	5	1.94	2.1
2	707.53	727.95	0.01	9	119.44	6.1
3	463.02	476.16	0.01	9	207.29	2.3
4	2132.87	2162.40	0.01	9	905.24	2.4
5	4.10	4.14	0.01	9	1.79	2.3
6	866.27	866.20	-	9	455.21	1.9
7	298.33	305.24	0.01	9	96.70	3.2

Table 5.2: Wall Time (s) for Sequential (with vs without Regularized Initial) and Parallel Solving

#P	Sys 1	Sys 2	Sys 3	Sys 4	Sys 5	Sys 6	Sys 7
3	3.14	355.08	278.70	1401.43	2.10	622.88	104.98
5	1.94	225.29	214.24	1004.69	2.10	481.73	98.44
7	1.91	142.74	209.17	939.40	1.91	470.18	97.19
9	1.91	119.44	207.29	905.25	1.79	455.21	96.70
11	1.95	119.48	207.08	894.27	1.63	453.13	96.38
13	-	119.09	206.38	874.53	1.61	451.93	96.42
17	-	120.01	211.70	865.51	1.63	451.57	96.20
21	-	119.17	-	852.49	-	451.36	96.54

Table 5.3: Parallel Timing (s) vs #Processor

#P	Sys1	Sys2	Sys3	Sys4	Sys5	Sys6	Sys7
3	1.3	2.1	1.7	1.5	2.0	1.4	2.9
5	2.1	3.2	2.2	2.2	2.0	1.8	3.1
7	2.1	5.1	2.3	2.3	2.2	1.8	3.1
9	2.1	6.1	2.3	2.4	2.3	1.9	3.2
11	2.0	6.1	2.3	2.4	2.6	1.9	3.2
13	-	6.1	2.3	2.4	2.5	1.9	3.2

Table 5.4: Speedup vs #Processor

System	TPDRG (best)	#P (A)	Greedy (A)	#P (B)	Greedy (B)
1	1.91	7	1.79	9	1.78
2	119.09	13	120.51	15	120.52
3	206.38	13	213.21	15	213.35
4	852.49	20	896.79	22	939.62
5	1.61	13	1.63	15	1.63
6	451.36	20	500.50	22	469.35
7	96.20	17	100.78	19	96.17

Table 5.5: Best *TPDRG* Timing vs *Greedy Scheduling (s)*

Chapter 6

Multiprocessed Parallelism Support in ALDOR on SMPs and Multicores

6.1 Introduction

Throughout the 1980s and 1990s the subject of parallel computer algebra was an active area of research. Many researchers have contributed to this area, both through the invention of parallel algorithms and the development and implementation of parallel systems. An excellent overview of these developments is provided in the *Computer Algebra Handbook* [67]. Over the past decade the area has received less intense attention, but recent developments in widely available computer hardware make the subject now more relevant than ever: Current hardware improvements have focused on increasing the number of computations that can be performed in parallel rather than on increasing clock speed alone. This change in focus has brought multi-core workstations to the desktop, expanding interest in parallel algorithms and revitalizing research in parallel computer algebra.

In this work we describe a *high-level categorical* parallel framework in ALDOR [3, 142] that supports high-performance computer algebra. This framework provides multiprocess parallelism support in ALDOR on symmetric multiprocessor (SMP) and multi-core architectures. Our work complements previous work in the area. Gautier and Mannhart previously developed a system known as $\prod^{IT}$[58, 103], which utilized MPI features in ALDOR to provide computer algebra on *distributed* parallel architectures. Their approach was to develop general facilities and demonstrate their use with

sample computer algebra problems. We have come at the problem from the opposite direction: Our design has been motivated by a particular family of challenging problems in the computation of triangular decompositions, and we have tried to generalize for broader applications. From a different perspective, Ashby, Kennedy and O'Boyle have used ALDOR for data-parallel QCD computations [6].

We have chosen ALDOR as our implementation language because it provides both facilities to support high-level mathematical abstraction as well as efficient access to low-level machine control. ALDOR is an extension of the AXIOM computer algebra system which focuses on interoperability with other languages and high-performance computing. The designers of ALDOR and AXIOM overcame many of the challenges associated with providing an environment for implementing the extremely rich relationships that exists among mathematical structures. This was accomplished by providing high-level categorical support which allows for the development of generic algorithms for solving computer algebra problems. Some of the features currently provided by ALDOR are being introduced into Fortress, a language which is presently being developed by Sun. Fortress is specifically designed for both high performance computing and high programmability [127].

Our framework uses ALDOR's low-level access to the machine to make effective use of shared memory multiprocessor and multi-core architectures, providing simple yet powerful supercomputing support for computer algebra. We support the communication of high-level objects between processes without requiring knowledge of low level details, allowing researchers to concentrate on implementing mathematical algorithms.

The remainder of this chapter describes our framework in detail. It is organized in the following manner. Section 6.2 introduces our parallel computation framework. Section 6.3 describes how data is communicated efficiently between the processes and concurrency control. This is followed by a discussion of our serialization tools for high-level ALDOR objects in Section 6.4. Dynamic process management and user-level scheduling techniques is discussed in Section 6.5. Benchmark performance results are presented in Section 6.6. We present our summary in Section 6.7.

6.2   Overview of the Parallel Framework

Our goal is to develop a high level, categorical, parallel framework for high-performance computer algebra that effectively exploits the parallel features of modern multi-core and multiprocessor computer architectures. In order to accomplish

this goal, we have introduced multiple process parallelism in ALDOR by providing a mechanism for spawning an arbitrary number of new processes dynamically at run-time (within the limits imposed by the operating system). When multiple processors or cores are available, these processes will execute in parallel. Furthermore, we have also implemented the mechanisms necessary to coordinate the execution of these processes and communicate data between them efficiently. A high level description of our framework is presented in the remainder of this section. The technical details of each component are presented in the sections that follow.

The ALDOR programming language is a type-complete, strongly-typed, imperative programming language. It uses a two-level object model consisting of *categories* and *domains*. In many respects these concepts are analogous to *interfaces* and *classes* in Java. ALDOR provides a type system that provides the programmer with the flexibility to build new types by creating new categories and domains, as well as the flexibility to extend existing categories and domains. For example, new categories and domains can be implemented to model algebraic structures (e.g. rings) and their members (e.g. polynomial domains). Pervasive use of dependent types allows static checking of dynamic objects and provides object-oriented features such as parametric polymorphism.

FRISCO (A Framework for Integrated Symbolic/Numeric Computation) was a project funded by the European Commission under the Esprit Reactive LTR Scheme from 1996 to 1999. It resulted in the creation of a library, `BasicMath`, for polynomial arithmetic and a sequential polynomial solver, `triade`, developed in ALDOR. Even today, `Triade` outperforms three solvers in Maple: `Triangularize`, `RegSer` and `SimSer` [32]. Many of the categories, domains and packages of this sequential implementation (such as polynomial arithmetic, polynomial greatest common divisor and resultant over an arbitrary ring) can be reused or extended for general purpose parallel symbolic computations.

ALDOR source code can be compiled into a variety of formats. These include native operating system executables; native operating system object files that can be linked with each other, or with C or Fortran code to form other applications; portable bytecode libraries; and C or Lisp source code [25]. Aggressive code optimizations produce code that performs comparably to hand-optimized C [143]. This makes it possible to build executables formed from source files written in several languages. Furthermore, by compiling ALDOR code to an object file, it can be linked into many different executables that will be run as independent parallel processes. ALDOR also provides a primitive, `run(...)`, for initiating a new program `P` from within a

program `Q`. The `run(...)` primitive is a wrapper for the C *exec(...)* function which launches the target application as a separate process. These features give us the basic functionality necessary for dynamic process management.

Because our parallel workloads will execute as separate processes, it is necessary to establish a mechanism for interprocess communication in ALDOR. We rely on *shared memory segments* from the standard set of UNIX System V interprocess communication tools. A shared memory segment is a block of memory that can be accessed by several processes. Shared memory segments are provided by most flavors of UNIX, providing us with portability across many platforms. The number of shared memory segments available on a system is normally confined to a small value. However, this value can usually be increased. Shared memory segments are commonly regarded as an effective way to communicate large amounts of data efficiently. The shared memory segments are accessed in ALDOR through a newly developed ALDOR domain called `SharedMemorySegment`. Our domain uses interoperability with the C programming language to provide the required functionality.

Our implementation of shared memory segments in ALDOR allows an array of integers to be communicated between processes. High-level ALDOR types can be communicated by converting them into an array of integers and then copying the integers into a shared memory segment. Similarly, the process receiving the data constructs a new instance of the ALDOR high-level data type from the provided array of integers. This serialize/unserialize process is necessary because the data types that represent sparse multivariate polynomials and dense multivariate polynomials are implemented with pointers. Consequently, it is not possible to copy instances of these ALDOR domains directly from one process to another because the pointers will not necessarily be valid in the destination process.

An additional shared memory segment is used to ensure that the serialized data is communicated between processes in a synchronized manner. We will refer to the second shared memory segment, which is a single integer value, as the *tag*. A protocol was developed requiring the destination process to check the tag value before accessing the polynomial data in the data shared memory segment. Our protocol exploits the fact that writing a single integer value to a shared memory segment (or reading a single integer value from a shared memory segment) is an atomic operation when the integer is word aligned. The specification for `shmat()`, the function used to attach to a shared memory segment, can be asked to return a pointer that is rounded down to the nearest multiple of the segment low boundary address multiple [133], giving an address that is guaranteed to be page aligned (and as such, will also be word aligned).

Consequently, because the tag value resides at the start of a shared memory segment, it will reside at an address that is word aligned. As such, it will not span multiple cache blocks, and read and write operations will be performed atomically. By following the memory access ordering restrictions of our protocol and exploiting the atomic nature of reads and writes, we are able to ensure that our data is accessed in a consistent manner in the presence of parallel execution.

Since parallel applications in computer algebra are generally dynamic and irregular, we must provide scheduling mechanism for processes in ALDOR to achieve load balancing. This parallel framework supports dynamic process management, and it is also feasible for users to apply scheduling techniques.

This section has provided a high level overview of the general concepts used to implement our parallel computer algebra framework. The low-level technical details are discussed in the following sections.

6.3   Data Communication and Synchronization

Our framework uses UNIX System V shared memory segments for both data communication and synchronization between parallel processes. We developed a new ALDOR domain, `SharedMemorySegment`, to provide easy access to shared memory segments from ALDOR source code. In order to transfer information from one ALDOR process to another, the information must be serialized into a primitive array of machine integers. The destination process unserializes the data, constructing a new high-level ALDOR datatype, before the data is utilized for other purposes.

Our SharedMemorySegment domain calls C functions in order to gain access to shared memory segments. Prototypes are shown below for the most frequently used functions:

```
key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

In our framework, we generally utilize these functions in the following manner:

1. A key is constructed for the shared memory segment using `ftok`, the path to a file, and an integer.
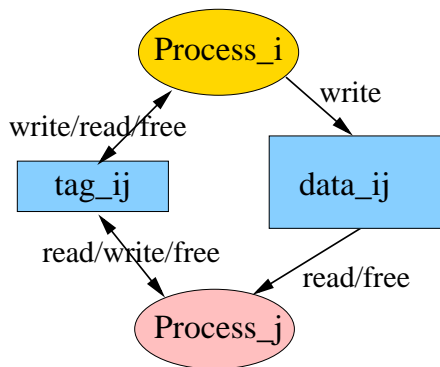
Figure 6.1: `Process_i` Sending Data to `Process_j`

2. The shared memory segment associated with the key is created (or connected to if it has already been created by another process) using `shmget`. The read/write permission of the segment are set by passing an appropriate value for *shmflg*.

3. A pointer to the shared memory is acquired by attaching to the shared memory segment using `shmat()`.

4. Read and write operations are performed on the shared memory by using the pointer returned by `shmat()`.

5. Each process detaches itself from the shared memory using `shmdt`.

6. The shared memory segment is deleted by providing the appropriate command flags to `shmctl()`.

Each shared memory segment is uniquely identified by its key and its size. Any process can access the shared memory segment if it knows both the key values and the size of the segment, provided that the shared memory segment is world readable. Note that it is imperative that applications free each shared memory segment that is allocated, either by invoking `shmctl()` before the application terminates or by using the `ipcrm` utility after the application completes because shared memory segments are not automatically released when a program terminates.

In our framework, each ALDOR process is assigned a unique virtual identifier, or *VPID*, when it is created. These VPIDs guide the flow of information between ALDOR processes, serving a similar role to process ranks in MPI's data communication model. Let `Process_i` and `Process_j` be processes with VPIDs i and j respectively. Figure 6.1 illustrates the strategy used by `Process_i` to send data to `Process_j`.

Our communication protocol uses the `tag_ij` shared memory segment to ensure that the `data_ij` segment is accessed in a safe manner. The keys for both the tag

and data segments are created using path names that are unique across our parallel framework. In particular, the key for the tag segment is created using the path */tmp/tag_ij* and the key for the data segment is created using the path */tmp/data_ij*. As a result, we are assured that these shared memory segments will only be accessed by `Process_i` and `Process_j`. Using a separate block of shared memory for each pair of communicating processes eliminates both the need for complex synchronization operations and the bottlenecks that can occur when many processes try to access a shared memory segment at the same time.

The tag segment consists of a single integer. It is used by the sending process, `Process_i`, to inform the receiving process, `Process_j`, of the size of the data that is being transferred[1]. Initially, the value of the shared memory segment is 0, denoting that no data is available to be read by `Process_j`. We do not need to explicitly write this 0 value to the shared memory segment because every byte in a shared memory segment is automatically initialized to zero as part of the allocation process. This initialization behavior is defined in IEEE Standard 1003.1 [133]. Once the polynomial data shared memory segment has been created and the data being transferred has been placed in the data segment by `Process_i`, it changes the value in the tag segment to the size of the data being transferred. `Process_i` is not permitted to change any value in either shared memory segment until it reads a 0 value from the tag segment.

When `Process_j` sees a value greater than zero in the tag segment it is assured that the required information is present in the data shared memory segment. `Process_j` writes the value -1 to the tag to indicate that it is currently accessing the shared memory segment. Once `Process_j` has successfully unserialized the data and freed the data segment, it writes the value 0 to the tag segment, indicating that it is done with that set of data. By freeing the shared memory segments immediately after use, we reduce the overall memory footprint of our framework.

If the value initially read by `Process_j` is -2 then `Process_j` knows that the data block being transmitted is empty. Consequently, it immediately writes a 0 back to the tag acknowledging that there was no data to receive.

Regardless of the tag value read initially, `Process_j` is not permitted to access the data segment after writing 0 to the tag segment until the value of the tag changes to a positive integer. Following this protocol ensures that `Process_j` will always read data that is complete and that `Process_i` will never overwrite data that is still needed by `Process_j`. The tag shared memory segment is released by either `Process_i` or

---

[1]The value -2 is used to indicate the transmission of an empty data block.

`Process_j` before it terminates. Note that `Process_i` is not permitted to release the tag shared memory unless it reads a 0 value from the tag.

The complete algorithm followed by `Process_i` and `Process_j` is described below:

- `Process_i` **(Sending)**

    1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"

    2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively

    3. Create or connect to the tag segment, setting the permission to allow both reads and writes

    4. Repeat until the value of the tag segment is 0

    5. Create the data segment with sufficient size to hold the values being sent to `Process_j`

    6. Write the data to the data shared memory segment

    7. Detach from the data segment

    8. Write the size of the data to the tag segment

- `Process_j` **(Receiving)**

    1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"

    2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively

    3. Create or connect to the tag segment, setting the permission to allow both reads and writes

    4. Repeat until the value of the tag segment, t, is greater than 0

    5. Write -1 to *tag_ij*

    6. Detach from the tag segment

    7. Connect to the data segment using key *data_ij*

    8. Read t integers from the data segment

    9. Detach from the data segment

   10. Delete the data segment

    11. Write 0 to the tag segment

We developed an ALDOR domain named `SharedMemorySegment` to provide easy access to shared memory segments from ALDOR source code. This domain used AL-DOR's interoperability with C in order to call the shared memory functions described earlier in this section. The domain has methods for creating and connecting to a shared memory segment, reading and writing values to and from the segment, and detaching from and deleting a shared memory segment. While these methods do not provide the full power of shared memory segments, they successfully hide many of the cumbersome details while being sufficient to implement our communication protocol.

The `InterProcessSharedMemoryPackage` domain provides a further level of abstract with the functions *Send(i,j,data)* and *Receive(i,j)*. *Send(i,j,data)* performs all of the operations described previously for `Process_i` while *Receive(i,j)* performs all of the operations described previously for `Process_j`. Using this domain allows the programmer to concentrate on solving computer algebra problems rather than the details of shared memory segments and interprocess communication.

6.4 Serialization of High-Level Objects

On shared memory computer, it would be ideal if we could copy ALDOR objects directly between processes. Unfortunately, a direct copy cannot be performed because ALDOR objects are represented using pointers, and because each process executes in its own address space. The memory usage within each address space will differ because the specific problem being solved by each process differs. Consequently, a pointer that is valid in one address space will not necessarily reference the correct piece of memory when it is copied to another address space. Consequently, it is necessary to serialize ALDOR objects before they can be transferred to another process using shared memory segments. This serialization process converts the high-level object into an array of machine integers before it is placed in the shared memory segment. The destination process uses the array of machine integers to reconstruct the high-level ALDOR objects and then performs additional computations using the objects.

We have developed a package named `Serialization` that serializes two polynomial types in the ALDOR `BasicMath` library. Presently, serialization is available for `SparseMultivariatePolynomial`, abbreviated SIMPLY, and `Distributed-MultivariatePolynomial`, abbreviated DMPOLY. `BasicMath` includes other polynomial representations such as `DenseRecursiveMultivariatePolynomial` and `Spare-AlternatedArrayMultivariatePolynomial`. We plan to address the serialization of these polynomial types in the future.
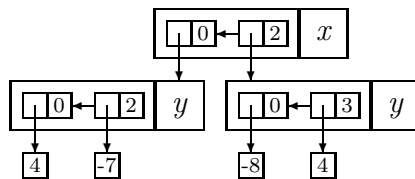
Figure 6.2: Sparse Multivariate Polynomial (SMPOLY) Representation of g
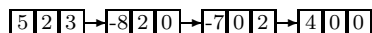
Figure 6.3: Distributed Multivariate Polynomial (DMPOLY) Representation of g

Both SMPOLY and DMPOLY are designed for the efficient representation and manipulation of sparse multivariate polynomials. Solvers have been developed in ALDOR, such as `triade` [92] based on the algorithm presented in [108], which are primarily designed for solving large systems with many variables [55]. In `triade`, a polynomial system is solved by way of triangular decomposition. Triangular decomposition requires a recursive vision of multivariate polynomials. Consequently, this solver employs the SMPOLY polynomial domain constructor because the representation is sparse and recursive. Thus a SMPOLY is a univariate polynomial whose coefficients are polynomials themselves. In broad terms, a SMPOLY is represented by a tree. The interior nodes are non-constant polynomials while the leaves are coefficients from the base ring. For example, the polynomial $g = 5x^2y^3 - 8x^2 - 7y^2 + 4$ with a variable order of $x > y$ is viewed as $(4 - 7y^2) + (-8 + 5y^3)x^2$. Figure 6.2 shows the SMPOLY representation of g.

A DMPOLY is represented by a list of terms, where each term is an exponent vector together with a coefficient. This data structure is flat. As such, it is more efficient for accessing the monomials in a polynomial, making it a suitable representation for the polynomial arithmetic involved in Gröbner basis computations. Figure 6.3 illustrates the DMPOLY representation of g.

Our `Serialization` package provides a function, `SerializeDMP()` which converts a DMPOLY into a primitive array of integers by traversing the values in the list of terms. Two functions are provided for converting a SMPOLY into an array of integers. One is called `SerializeSMPbyKronecker()` which turns a SMPOLY into a primitive array of machine integers via a univariate polynomial using Kronecker substitution [57]. A corresponding function is provided named `UnserializeSMPbyKronecker()`. It constructs a SMPOLY from a primitive array of machine integers. While we have successfully used Kronecker substitution here in the context of spare multivariate polynomials, one would expect this representation to be more suitable for representing

dense multivariate polynomials. As a result, our work on `SerializeSMPbyKroneck-er()` represents preliminary work for a future investigation in the serialization of `DenseRecursiveMultivariatePolynomial`.

Another function provided by the serialization package is `SerializeSMPbyDMP()`. It converts a SMPOLY into DMPOLY, and then into a primitive array of machine integers. A corresponding function, named `UnserializeSMPbyDMP()`, constructs a SMPOLY from a primitive array of machine integers. In addition, we provide functions that convert a list of multivariate polynomials for some list of variables with the ring characteristic into a primitive array of machine integers.

For our example polynomial g, shown previously, the `SerializeSMPby-Kronecker()` function will return an array consisting of $\{5, 0, 0, 0, -7, 0, 0, 0, -8, 0, 4\}$. Using `SerializeSMPbyDMP()` for g gives $\{5, 2, 3, 8, 2, 0, 7, 0, 2, 4, 0, 0\}$.

6.5   Dynamic Process Management

Since parallel applications in computer algebra are usually dynamic and irregular, dynamic process management adds flexibility and eases dynamic task management. Earlier research has suggested the importance of dynamic process management in computer algebra [141]. In this section, we describe the dynamic process management mechanism used in our parallel framework. In addition, we show that it is convenient to use in user's programs. Using dynamic process management reduces the complexity of data communication and user-level scheduling for load balancing.

We created a new ALDOR function, `Spawn(command, argument)`, which allows an ALDOR program to create a new process. The *command* is the name (with path) of a program to execute as a new process while *argument* is a list of arguments to the command. We implemented our Spawn function using ALDOR's `run()` primitive. Internally, `run()` uses the `system()` function provided as part of the standard C library on most UNIX platforms. As a result, we are able to control how many new processes we spawn, and the order in which the processes are created. Once the new processes are created, their parallel execution is managed by the operating system's scheduler.

We define a **task** to be a program that can be executed independently that performs some function or processes some data. In our framework, it is the the user's responsibility to pass an integer as part of the argument list which is the VPID of the spawned process. This VPID is used to allow the process to communicate with other spawned processes as was discussed previously in Section 6.3.

This is analogous to what a user must do when using MPI, where a procedure being distributed to a processor is identified by the processor's rank. Using its rank, a procedure can communicate with other procedures executed by other processors.

By building on the `run()` primitive, our `Spawn()` function can be used within a process (say running program A) to launch one or more additional processes that will run other programs independently. What processor each of these processes will execute on depends on the user-level scheduler and the operating system's scheduler.

The key issue that a user needs to pay attention to is the organization of the unique VPIDs within a parallel application. We provide a solution to a general computing model described by a directed acyclic graph (DAG) for two common schemes: "task farming" and "dynamic fully-strict task processing".

The solution for the task farming scheme is simple. A manager process with VPID 0 starts the main program and spawns worker processes. The manager also sends the data to each worker process. When a worker completes its job, it sends its result back to the manager and then it terminates. If the manager schedules tasks so that the maximum number of worker processes can run in parallel is bounded by $ncpu$, then the manager needs to maintain two integer variables: process counter and VPID counter, abbreviated PC and $VPIDC$ respectively. In addition, a list data structure, $listVPID$, is needed to hold the VPIDs of the active worker processes. PC is initialized to 0, and $VPIDC$ is initialized to 1. Initially $listVPID$ is empty. When a task needs a worker process, the Manager will check if $PC < ncpu$. If the result of the comparison is true then the manager will launch a new worker for the task. The manager will pass the value of $VPIDC$ as the VPID argument to this worker and the manager will send the data needed as well. Then the Manager will add $VPIDC$ to $listVPID$ and increment PC and $VPIDC$. The Manager will repeat this procedure if there are additional tasks and $PC < ncpu$. Otherwise, the Manger will traverse $listVPID$, checking if each worker is done. If a worker has completed, the manager will collect the worker's result, remove its VPID from $listVPID$ and decrement PC. These activities will be repeated until all of the tasks are solved. All data communication between the manager and a worker is achieved by the techniques described in Section 6.3.

The scheduling algorithm in this solution corresponds exactly to the *greedy* scheduling method [68]. A greedy scheduler attempts to do as much work as possible at every step for a given number of processors, P. If there are at least P tasks ready to run, it selects any P of them and runs them. When there are strictly less than P tasks that are ready to run, the greedy scheduler runs them all. Given P pro-
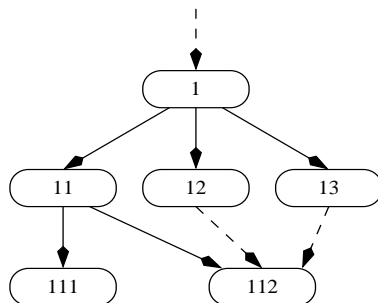
Figure 6.4: Dynamic Fully-Strict Task Processing

cessors, a greedy scheduler executes any computation DAG in time: $T_p \leq T_1/P + T_\infty$, where T_p is the minimum running time on P processors, T_1 is the minimum running time on 1 processor, and T_∞ is the critical path length of a DAG. A greedy scheduler is always within a factor of 2 of optimal. It is generally a good scheduler.

A variation based on this scheme is the *task pool with dimension and rank guided dynamic scheduling* designed and implemented in ALDOR for a parallel solver [111]. An implementation of the *greedy* scheduling method has also been realized in this solver for performance evaluation.

Another scheme we provide a solution to is dynamic fully-strict task processing, where tasks are generated dynamically and processed accordingly. In general, this problem can be modeled by a DAG. Part of an example DAG is shown in Figure 6.4. This type of problem corresponds to a problem known as *fully strict (i.e. well-structured) multithreaded computations*, originally published in [18]. A solution for the organization of unique VPIDs for using multiprocessed parallelism by this framework is illustrated in Figure 6.4.

In Figure 6.4, the dotted arrows denote data dependencies, while the solid arrows show when new processes were spawned. The initial process has VPID 1, and starts with the input data. Then the initial process spawns three new processes with VPIDs of 11, 12 and 13 respectively. The following rule is used to generate and assign VPIDs to any new processes it spawns. Let the process' VPID be k. Let the number of new processes it will spawn be n. For $1 \leq i \leq n$, the ith new process is given a VPID of ki, which is obtained by appending i to k. For instance, the first new process is given a VPID of $k1$, the second new process is given a VPID of $k2$, etc. Constructing the VPID as ki guarantees uniqueness within this application because the prefix, k, was unique in its parent's list of spawned processes and i is unique for each process spawned from this process. We note that this solution is akin to the scheme for handling the rank of spawned processes in MPICH2 [4].

The performance of dynamic process management in this parallel framework for coarse grained parallel computations is reported in Section 6.6.

6.6 Experimentation

This parallel framework has been used successfully to implement a parallel symbolic triangular decomposition solver [111]. It utilizes both the dynamic task management and data communication techniques we have described previously to communicate lists of multivariate polynomials to workers which execute in parallel. We describe the performance gains that we have achieved using our framework. In addition, we provide a detailed analysis of the overhead costs introduced such as the cost of process spawning and data communication/serialization for both the Kronecker substitution and `DMPOLY` serialization techniques. Results are presented for several well known examples, allowing the performance and overhead of our parallel framework to be compared with techniques developed by other authors.

Our experimental results were gathered on *Silky*, one of several multiprocessor clusters and SMP systems that make up Canada's Shared Hierarchical Academic Research Computing Network (SHARCNET). Silky is classified by SHARCNET as a mid-range symmetric multiprocessing (SMP) cluster. It is a SGI Altix 3700 Bx2, equipped with 128 Itanium2 processors clocked at 1.6GHz. The cluster has 256GB of main memory with a 6MB cache and runs SUSE Linux Enterprise Server 10 (ia64). Unfortunately Silky is a shared machine accessed by a large number of researchers in areas from computer algebra to computational physics. It is common for the active processes to occupy over 95 percent of the main memory and all 128 processors. As a result, we were unable to acquire sufficient resources on the machine to run large memory examples such as Virasoro.

The parallel solver was developed based on a triangular decomposition algorithm called Triade [108]. It makes use of the ALDOR `BasicMath` library for polynomial arithmetic over an arbitrary ring. We compare the performance of our framework with the sequential `triade` solver in ALDOR which uses the same underlying algorithm. Thus, the performance results we present here represent how well we are able to use the parallel resources available to us rather than the difference between two distinct algorithms.

The Triade algorithm uses "incremental solving", organizing the computation into a dynamic task tree. A `task` is any pair $[F, T]$ where F is a finite system of equations to solve and T is a solved system. Splitting of a task is based on the `D5 Principle`

[43] and case distinction of the form $f = 0 \; or \; f \neq 0$. The parallelization of this algorithm exploits the parallel opportunities created by the task splitting based on the `D5 Principle` combined with modular techniques [39].

Parallelizing the problem in this manner provides a coarse grained division of the work into separate processes. The parallelization is highly dynamic, with the final shape of the task tree being determined only when a solution to the system is achieved. The parallelization is also highly irregular, varying greatly with respect to both the input system of equations and the size of the task represented by each node in the tree. The implementation for this algorithm uses a "task farming" scheme and a cost-guided scheduling algorithm. A manager process preprocesses the input system and distributes intermediate tasks to the worker processes. If only one task needs to be solved then the manager completes the task itself. The manager also processes trivial tasks on its own.

When there is a task to perform, and an available processor, the manager spawns a new worker process. The manager then forwards the data for the task to that worker. The worker will process the task and send the intermediate results back to the manager unless the task is solved. Once the intermediate results or solution are determined the worker process terminates. In our example, each task consists of a list of ALDOR `SMPOLY` polynomials.

Table 6.1 lists the number of variables, n, and the total degree, d for each of the examples. It also lists the prime number for modular computation and the time required to reach a solution using the sequential solver Triade. Table 6.2 records the time required to reach a solution in our parallel framework for the two serialization techniques discussed previously. The number of CPUs used and the speedup relative to the sequential algorithm are also reported in this table.

In Table 6.3 and Table 6.4 we report additional details about the behavior of the Kronecker and DMPOLY serialization methods respectively. These details include the number of workers, number of tags, number of integers read and written and the percentage of zeros in the integers transferred. In every case, values for reads and writes are reported from the perspective of the worker tasks. Note that each worker process is created dynamically to process a specific task. The worker terminates when it completes its task. Consequently, the total number of workers used is equal to the total number of tasks being executed in parallel.

Table 6.5 and Table 6.6 show the time spent spawning workers, synchronizing their execution and communicating data to and from them. The reading time includes both the time required to read the array of machine integers from shared memory

Sys	Name	n	d	p	Sequential (s)
1	eco6	6	3	105761	4.00
2	eco7	7	3	387799	727.95
3	CNogues2	4	6	155317	476.16
4	CNogues	4	8	513899	2162.40
5	Nooburg4	4	3	7703	4.14
6	UBikker	4	3	7841	866.20
7	Cohn2	4	6	188261	305.24

Table 6.1: Polynomial Examples and Sequential Timing

Sys	CPUs	Kron. (s)	DMP (s)	Kron. Speedup	DMP Speedup
1	5	1.94	1.91	2.1	2.1
2	9	119.44	117.41	6.1	6.2
3	9	207.29	215.28	2.3	2.2
4	9	905.25	1002.56	2.4	2.2
5	9	1.79	1.81	2.3	2.3
6	9	455.21	463.24	1.9	1.8
7	9	96.70	102.55	3.2	3.0

Table 6.2: Parallel Timing on two Serializing Methods

Sys	Workers (#)	Tags (#)	Read (#int*)	Write (#int)	Total (#int)	$Zeros$ (%)
1	9	9	4131	3586	7717	59
2	24	24	29307	27382	56689	72
3	32	32	57106	55696	112802	73
4	42	42	216000	214217	430217	83
5	14	14	13307	0	13307	72
6	49	49	128983	125162	254145	55
7	44	44	39146	38280	77426	39

Table 6.3: Dissection of Workers' Overhead for Kronecker (* One int has 8 bytes)

Sys	Workers (#)	Tags (#)	Read (#int)	Write (#int)	Total (#int)	Zeros (%)
1	9	9	5069	4449	9518	55
2	24	24	36893	35184	72077	57
3	32	32	64106	64106	127304	39
4	42	42	168178	167186	335364	39
5	14	14	12681	0	12681	44
6	49	49	271845	267761	539606	42
7	44	44	104486	103534	208020	40

Table 6.4: Dissection of Workers' Overhead for DMPOLY

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Over-head (%)
1	358	1067	492	76	3.80	24.4
2	579	3414	1264	184	660.54	0.3
3	773	4887	9682	623	469.48	2.4
4	1695	7221	68737	491	2164.62	3.3
5	452	1940	488	0	3.57	26.4
6	1558	7773	21762	823	871.04	2.8
7	925	6014	2378	369	289.15	1.3

Table 6.5: Dissection of Workers' Time for Kronecker (Wall Time)

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Over-head (%)
1	314	1211	347	79	3.71	20.0
2	685	3498	1345	623	611.16	0.4
3	1435	4676	1813	683	474.16	0.8
4	1723	7524	80490	2360	2134.71	3.8
5	552	2224	764	0	3.65	36.2
6	1994	7847	52157	5242	886.59	6.7
7	1110	6673	5881	2063	282.16	3.2

Table 6.6: Dissection of Workers' Time for DMPOLY (Wall Time)

and the time required to reconstruct the high-level ALDOR object. Similarly, the writing time includes both the time spent serializing the high level objects and the time spent to copy the serialized data into a shared memory segment. In addition, we report the total net amount of work performed, which is the time spent by workers excluding the time spent spawning processes and performing synchronization and data communication. The percentage overhead is the ratio of the sum of the time spent on overhead divided by the net amount of work performed. Table 6.7 and Table 6.8 go on to show the average cost per worker spawned, per synchronization tag used, per integer read and unserialized and per integer serialized and written for the Kronecker and DMPOLY serialization methods respectively.

Examining the total parallel execution time reveals that there is little variation between the Kronecker and DMPOLY serialization techniques. Similar performance was observed because the examples presented here transfer data that is not very sparse, as is indicated in the percent zeros column.

Most of the examples considered in this study show a low amount of parallel overhead. Exceptions to this general pattern are Sys 1 and Sys 5 which are both

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	40	118	119	21
2	24	142	43	6
3	24	152	169	11
4	40	172	318	2
5	32	138	36	-
6	32	158	168	6
7	21	136	60	9
AVG	30	145	130	9

Table 6.7: Analysis of Workers' Overhead for Kronecker

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	35	134	68	17
2	29	146	36	18
3	45	146	180	21
4	41	179	478	14
5	39	158	59	-
6	41	160	192	19
7	26	151	56	20
AVG	37	153	152	18

Table 6.8: Analysis of Workers' Overhead for DMPOLY

smaller examples. Even though the level of overhead is low, our results show that this parallel framework is only suitable for coarse grained parallel symbolic computations. The granularity of this parallelization framework is very coarse, as shown by the total number of workers (tasks) that are executed in parallel and the total number of workers used in total. The average cost of spawning a process is approximately 35 milliseconds. Reading one integer and unserializing it to reconstruct the high-level ALDOR object normally takes between 36 and 192 microseconds on average. The cost in Sys 4 is outside of this range, likely due to the relative high degree of the polynomials in the data being transferred. We plan to investigate this phenomenon further in the future.

We also observed that serializing and writing costs are much lower than the reading and unserialization costs. This difference occurs because constructing a new high-level object is expensive, involving numerous memory allocations to represent the object's complex structure. In contrast, serializing the object does not require any memory allocation or deallocation operations. The time complexity of serialization via either

Kronecker substitution or DMPOLY is linear. Interestingly, the cost associated with our synchronization tags was small in comparison to the other sources of overhead in our parallel framework.

We also observed that the per integer serialization cost for a SMPOLY via DM-POLY almost doubles compared to using Kronecker substitution. This doubling reveals a drawback in our implementation, which requires that the SMPOLY must be converted to a DMPOLY twice, once to determine the size of the shared memory segment, and then a second time to write the serialized data into the shared memory segment. We expect that it will be easy to remove this inefficiency by improving our implementation.

Finally, we wish to point out that the dynamic nature of our task management technique is particularly advantageous in a shared computing environment such as SHARCNET. While there is overhead associated with spawning and terminating processes, our technique ensures that a process only exists when it has useful work to perform. There will not be any processes left idling, waiting for something to do. This helps ensure that shared computing resources are used effectively. Furthermore, removing idle processes ensures that we do not spend unnecessary time communicating with or synchronizing such processes. Because of this, we have found, both in theory and in practice, that our communication costs do not increase with number of CPUs utilized.

6.7 Summary

We have reported on a high-level categorical parallel framework to support high performance computer algebra on SMPs and multicores. We have used the ALDOR programming language as our implementation vehicle since it has high-level categorical support for generic algorithms in computer algebra, while providing the necessary low-level access for high performance computing.

Our framework provides functions for dynamic process management and synchronized data communication for high-level ALDOR objects such as sparse multivariate polynomials via the shared memory segments. This framework is complementary to $\prod^{IT}$ [103] which targeted distributed architectures.

Throughout the design and implementation of the framework, we have kept the solution of multivariate polynomial systems as a motivating example. Indeed, this framework has been used for the successful implementation of a sophisticated parallel symbolic solver. Our evaluation of the parallel overhead has shown that this parallel

framework is efficient for coarse-grained parallel symbolic computations. More experimentation on the granularity of parallelism supported by this framework is work in progress.

We plan to develop a model for threads in ALDOR to support finer grained parallelization, as has been done for SACLIB [84, 121] and KAAPI [74, 59], and support automatic scheduling based on "work stealing" [18, 56]. In this setting, ALDOR's system of parametric types provides opportunities for elegant problem formulation. As a practical matter, it will be necessary to review the ALDOR run-time system to take advantage of threads and modify it in a few places for thread safety. In particular, one current investigation is modification to use "localized tracing" [33], in the garbage collector to allow it to make use of multiple threads.

Chapter 7

Overview of the `RegularChains` Library in Maple

The `RegularChains` library provides facilities for symbolic computations with systems of polynomial equations. In particular, it allows to compute modulo a set of algebraic relations. It also allows automatic case discussion (and recombination) handling zero-divisors and parameters, which permits triangular decomposition of polynomial equations.

The `RegularChains` library in Maple was developed based on an algorithm for triangular decompositions, called Triade [108]. The Triade algorithm has been realized, during the last eight years, in three computer algebra systems: AXIOM, Aldor, and Maple, targeting different communities of users. In this chaper, we also summarize the challenges in implementing triangular decompositions, and presents a comparison between three implementations, and highlights their advantages and weaknesses in terms of efficiency, ease of use and targeted audience.

7.1 Organization of the `RegularChains` Library in Maple

The `RegularChains` library is a collection of commands for solving systems of algebraic equations symbolically and studying their solutions. The field $\mathbb{K}$ of coefficients can be $\mathbb{Q}$, a prime field, or a field of multivariate rational functions over $\mathbb{Q}$ or a prime field. In addition, a remarkable feature of the `RegularChains` library is that it also provides functions for computing modulo regular chains, based on the algorithms of [108, 109].

The most frequently used functions are accessible at the top level module in the library. Two submodules, `ChainTools` and `MatrixTools`, contain additional commands to manipulate regular chains and triangular decompositions.

7.1.1 The Top Level Module

In the top level module of `RegularChains`, the main function is `Triangularize`. For a set F of polynomials, the command `Triangularize` computes the common roots of F in an algebraic closure $\mathbb{L}$ of $\mathbb{K}$ in a form of a triangular decomposition (If $\mathbb{K}$ is $\mathbb{Q}$, then $\mathbb{L}$ is the field of the complex numbers.). By default, the sense of Kalkbrener is used. An option for solving in the sense of Lazard is also available. The operation `PolynomialRing` allows the user to define the polynomial ring $\mathbb{R}$ in which the computations take place, together with the order of the variables.

One very useful function is `RegularGcd`, which computes gcds of two polynomials p_1 and p_2 with common main variable v modulo a regular chain `rc`. It returns a list of pairs $[g_i, rc_i]$ where g_i is a polynomial and rc_i is a regular chain. For each pair, the polynomial g_i is a `gcd` of p_1 and p_2 modulo the saturated ideal of rc_i. Moreover, the leading coefficient of the polynomial g_i w.r.t. v is regular modulo the saturated ideal of rc_i. Finally, the returned regular chains rc_i form a triangular decomposition of `rc` (in the sense of Kalkbrener). See the example in Section 7.2. Other useful functions are `NormalForm` (which applies only to strongly normalized regular chains) and `SparsePseudoRemainder` (which can be used with any regular chains) for "reducing" a polynomial modulo a regular chain.

7.1.2 The ChainTools Submodule

The `ChainTools` submodule is a collection of commands to manipulate regular chains. These commands split into different categories. The commands `Empty`, `ListConstruct`, `Construct`, `Chain` create regular chains from lists of polynomials and other regular chains. Other commands allow to inspect the properties of a regular chain, such as `IsZeroDimensional` and `IsStronglyNormalized`.

The commands `DahanSchostTransform` and `Lift` perform transformation on a regular chain, whereas the commands `EquiprojectableDecomposition`, `SeparateSolutions`, `Squarefree` perform transformation on a triangular decomposition.

The commands `IsInSaturate` and `IsInRadical` compare one polynomial p and one regular chain T. The first one decides whether p belongs to the saturated ideal

I of T. This is done without computing a system of generators of I, just by pseudo-division. In fact, the `RegularChains` module never computes explicitly a system of generators for a saturated ideal. The second command decides whether p belongs to the radical of I; this is achieved simply by gcd computations.

The commands `EqualSaturatedIdeals` and `IsIncluded` compare two regular chains T_1 and T_2. More precisely, the first one can decide whether the saturated ideals of T_1 and T_2 are equal or not. If the second command returns true, then the saturated ideal of T_1 is contained in that of T_2. However, if it returns false, nothing can be concluded.

7.1.3 The MatrixTools Submodule

The `MatrixTools` submodule is a collection of commands to manipulate matrices of polynomials modulo regular chains, including `IsZeroMatrix`, `JacobianMatrix`, `LowerEchelonForm`, `MatrixInverse`, `MatrixMultiply`, `MatrixOverChain`, and `MatrixCombine`.

The main purpose of the commands of the `MatrixTools` submodule is to compute the inverse of the Jacobian matrix of a polynomial system modulo (the saturated ideal of) a regular chain. This question arises for instance in Hensel lifting techniques for triangular sets [119]. The commands of the `MatrixTools` submodule are quite standard, such as multiplication of matrices, computation of inverse or lower echelon of a matrix. The lower echelon form of a matrix is computed following the standard *fraction-free* Gaussian elimination [61]. However, these commands are considered here in a non-standard context. Indeed, the coefficients of these matrices are polynomials and the computations are performed modulo (the saturated ideal of) a regular chain. In case of a zero-divisor, following the D5 principle [43], the computations split into branches, where in each branch the zero-divisor becomes either zero or a regular element. The `MatrixCombine` command is an application of the equiprojectable decomposition. It generates a canonical output for automatic case discussion. See the examples and algorithms about these functions in Chapter 4 Section 4.5.

7.2 The RegularChains Keynote Features

We present here an overview of the `RegularChains` library by means of a series of examples. The first few ones are for non-experts in symbolic computation whereas the next ones require some familiarity with this area.

7.2.1 Solving Polynomial Systems Symbolically

In this first example, we show how the `RegularChains` library can solve systems of algebraic equations symbolically. After loading the library in our MAPLE session, we define the ring of the polynomials of the system to be solved. Indeed, most operations of the `RegularChains` library requires such polynomial ring as argument. This is where one specifies the variable ordering. In our example we choose `x > y > z`. Other arguments passed to the `PolynomialRing` command could be a set of parameters or the characteristic of the ground field. By default, there are no parameters and the characteristic is zero. Hence, in our example below, the polynomial ring is $\mathbb{Q}[x, y, z]$, that is the ring of polynomials in x, y, z with rational number coefficients.

> `R:=PolynomialRing([x,y,z]);`
$$R := polynomial\_ring$$

Then we define a set of polynomials of `R` by

> `sys := {x^2 + y + z - 1, x + y^2 + z - 1, x + y + z^2 - 1};`
$$sys := \{x^2 + y + z - 1, x + y^2 + z - 1, x + y + z^2 - 1\}$$

Ideally, one would like to decompose the solutions of `sys` into a list of points. This is what `Triangularize` does using symbolic expressions. However, some points are grouped because they share some properties. These groups are precisely regular chains.

> `dec := Triangularize(sys, R);`
$$dec := [regular\_chain, regular\_chain, regular\_chain, regular\_chain]$$

Because regular chains may involve large expressions, by default the output is not displayed. However, one may ask to view them! The command `Equations` displays the list of polynomials of a regular chain.

> `map(Equations, dec, R);`
$$[[x - 1, y, z], [x, y - 1, z], [x, y, z - 1], [x - z, y - z, z^2 + 2z - 1]]$$

The first three regular chains are very simple: each of them clearly corresponds to a point in the space. Let us have a closer look at the last one. The polynomial in `z` has two solutions. Each of them corresponds to a point in the space.

Since the above system has finitely many solutions, its equiprojectable decomposition (the canonical form of output) is obtained by the following call.

```
> decepd := EquiprojectableDecomposition(dec, R); map(Equations,
  decepd, R);
```

$$decepd := \{regular\_chain\}$$

$$\{[x+y-1, -y+y^2, z], [2\,x+z^2-1, 2\,y+z^2-1, z^3+z^2-3\,z+1]\}$$

`Triangularize` can also handle inequations. Below we impose the condition $x-z \neq 0$. Then, two points from the original decomposition are removed.

```
> decn := Triangularize(sys,[x-z],R); map(Equations, decn, R);
```

$$decn := [regular\_chain, regular\_chain]$$

$$[[x-1, y, z], [x, y, z-1]]$$

The option *'probability'='prob'* of `Triangularize` is used to compute an equiprojectable decomposition of the input system using the probabilistic modular algorithm in [39]. This algorithm applies only to square systems in characteristic zero. For square input systems generating radical zero-dimensional ideals, this modular and probabilistic algorithm is asymptotically faster than the general (and generic) algorithm implemented by `RegularChains:-Triangularize`. The following is an example.

```
> sys1 := [x*y^4+y*z^4-2*x^2*y-3, y^4+x*y^2*z+x^2-2*x*y+y^2+z^2,
  -x^3*y^2+x*y*z^3+y^4+x*y^2*z-2*x*y];
```

```
> decepd := Triangularize(sys1, R, probability = 0.9);
```

$$decepd := [regular\_chain]$$

Below shows an example for solving over a polynomial ring with a prime characteristic of 3. The ring is created by the value 3 to the optional parameter for prime characteristic of `PolynomialRing`. Triangular decomposition will be performed over this ring.

```
> R := PolynomialRing([x ,y ,z], 3);
```

$$R := polynomial\_ring$$

```
> decp := Triangularize(sys, R); map(Equations, decp, R);
```

$$decp := [regular\_chain, regular\_chain, regular\_chain, regular\_chain]$$

$$[[x+2, y, z], [x, y+2, z], [x, y, z+2], [x+2\,z, y+2\,z, z^2+2\,z+2]]$$

7.2.2 Solving Polynomial Systems with Parameters

The `RegularChains` library can solve systems of equations with parameters. To illustrate this feature, let us consider a "generic" linear system with 2 unknowns and 2 equations. First we declare `x,y,a,b,c,d,g,h` all as variables.

```
>  R:=PolynomialRing([x,y,a,b,c,d,g,h]):   sys:={a*x+b*y-g,
c*x+d*y-h};
```

$$sys \; := \; \{ax + by - g, cx + dy - h\}$$

In this setting, with more unknowns than equations, our system has an infinite number of solutions. There are two ways of solving such systems. First, one can describe its "generic solutions", which is done by computing a triangular decomposition in the sense of Kalkbrener. This is the default behavior of the `Triangularize` command. Observe that the cases where the determinant $-cb + ad$ vanishes are not explicitly described.

```
>  dec := Triangularize(sys, R); map(Equations, dec, R);
```

$$dec \; := \; [regular\_chain]$$
$$[[cx + dy - h, (-cb + ad)\,y + cg - ah]]$$

Now let us compute all the solutions (generic or not), that is to compute a triangular decomposition in the sense of Lazard. To do so, we use the option `output=lazard` of the `Triangularize` command.

```
>  dec := Triangularize(sys, R, output=lazard);
```

$$dec \; := \; [regular\_chain, regular\_chain, regular\_chain, regular\_chain, regular\_chain,$$
$$regular\_chain, regular\_chain, regular\_chain, regular\_chain, regular\_chain]$$

When a regular chain encodes an infinite number of solutions, these solutions are the values canceling any of the polynomials returned by the `Equations` command and none of the polynomials returned by the `Inequations` command. In the command below, for each regular chain of `dec`, we display on the same line its list of equations eq and its list of inequations $ineq$. For instance, the solutions given by the first regular chain in `dec` satisfy simultaneously $cx + dy - h = 0$, $(-cb + ad)y + cg - ah = 0$, $-cb + ad \neq 0$ and $c \neq 0$.

```
>  [seq([eq=Equations(dec[i],R), ineq=Inequations(dec[i],R)],
i=1..nops(dec))];
```

$$
\begin{array}{llll}
eq &=& \{cx + dy - h, (-cb + ad)y + cg - ah\} & ineq &=& \{-cb + ad, c\} \\
eq &=& \{ax + by - g, dy - h, c\}, & ineq &=& \{a, d\} \\
eq &=& \{cx + dy - h, -cb + ad, -dg + hb\}, & ineq &=& \{h, c, d\} \\
eq &=& \{cx - h, -cg + ah, b, d\}, & ineq &=& \{h, c\} \\
eq &=& \{dy - h, a, -dg + hb, c\}, & ineq &=& \{h, d\} \\
eq &=& \{cx + dy, -cb + ad, g, h\}, & ineq &=& \{c, d\} \\
eq &=& \{by - g, a, c, d, h\}, & ineq &=& \{b\} \\
eq &=& \{x, b, d, g, h\}, & ineq &=& \{\} \\
eq &=& \{y, a, c, g, h\}, & ineq &=& \{\} \\
eq &=& \{a, b, c, d, g, h\}, & ineq &=& \{\}
\end{array}
$$

Now, we change our polynomial ring in order to specify that g and h are parameters. This means that we consider now the ring of polynomials in variables x,y,a,b,c,d with coefficients in the field of rational functions $\mathbb{Q}(g, h)$. When solving our input system in the sense of Lazard with this new polynomial ring, the last five cases above are discarded since $g = 0$ or $h = 0$ cannot hold anymore.

```
>  R2 := PolynomialRing([x,y,a,b,c,d],{g,h}):
>  dec := Triangularize(sys, R2, output=lazard):
>  [seq([eq=Equations(dec[i],R2), ineq=Inequations(dec[i],R2)],
i=1..nops(dec))];
```

$$
\begin{array}{llll}
eq &=& \{cx + dy - h, (-cb + ad)y + cg - ah\} & ineq &=& \{-cb + ad, c\} \\
eq &=& \{ax + by - g, dy - h, c\}, & ineq &=& \{a, d\} \\
eq &=& \{cx + dy - h, -cb + ad, -dg + hb\}, & ineq &=& \{c, d\} \\
eq &=& \{cx - h, -cg + ah, b, d\}, & ineq &=& \{c\} \\
eq &=& \{dy - h, a, -dg + hb, c\}, & ineq &=& \{d\}
\end{array}
$$

To summarize:

- one can specify (in advance) a set of variables to be viewed as parameters (this was done with the latter call to Triangularize obtaining 5 cases)

- or one can discover the largest set of variables which can be viewed as parameters (this was done with the first call to Triangularize, leading to the generic points, in the sense of Kalkbrener)

- or one can view all variables as unknowns (as in the second call to Triangularize, returning 10 cases).

7.2.3 Computation over Non-integral Domains

The `RegularChains` library provides linear algebra and polynomial computations over towers of simple extensions. These algebraic structures, which appear naturally when solving polynomial systems, may possess zero-divisors. Below, we construct a regular chain `rc` with two simple algebraic extensions. The first one is the extension of the field of rational numbers by $\sqrt{2}$. The second one is not a field extension but introduces *zero-divisors*. Indeed, its defining polynomial $y^2 - y + x - 2$ factorizes as $(y - x)(y + x - 1)$ modulo the defining polynomial $x^2 - 2$ of the first extension.

```
> R := PolynomialRing([z,y,x]);
```
$$R := \ polynomial\_ring$$
```
> rc := Chain([x^2-2, y^2 -y + x -2], Empty(R), R);
```
$$rc := \ regular\_chain$$
```
> Equations(rc,R);
```
$$[y^2 - y + x - 2, x^2 - 2]$$

Let us compute the gcd of polynomials `p1` and `p2` below w.r.t. `rc`. The example is made such that *splitting* is needed.

```
> p1 := (y-x)*z+(y+x-1)*(z+1);
```
$$p1 := (y - x)\,z + (y + x - 1)\,(z + 1)$$
```
> p2 := (y-x)*1+(y+x-1)*(z+1);
```
$$p2 := y - x + (y + x - 1)\,(z + 1)$$
```
> g:= RegularGcd(p1,p2,z,rc,R,normalized=yes);
```
$$g := [[2\,y + zy + zx - z - 1, regular\_chain], [3\,y^2 - 2\,yx - 2\,y - x^2 + 2\,x, regular\_chain]]$$
```
> rc1 := g[1][2]:  Equations(rc1, R);
```
$$[y - x, x^2 - 2]$$
```
> rc2 := g[2][2]:  Equations(rc2, R);
```
$$[y + x - 1, x^2 - 2]$$

We obtain two cases. This case discussion comes from the following fact. Modulo the regular chain `rc1`, the `gcd` of `p1` and `p2` has degree 1 w.r.t. `z`, whereas it has degree 0 modulo `rc2`. In general, the output of a gcd computation w.r.t. a regular chain is a list of "cases". Indeed, such gcd is computed by applying the *D5 principle*.

7.2.4   Controlling the Properties and the Size of the Output

Solving systems of equations by means of regular chains can help in reducing the size of the coefficients in the output. Even when no splitting arises! In the example below, due to Barry Trager, we compare the size of the output of `Triangularize` with the lexicographical Gröbner basis for the same variable ordering. Here, we do not print the Gröbner basis nor the regular chain, only the size (as number of characters in the output) of which is printed.

```
>  R := PolynomialRing([x,y,z]);
```
$$R := polynomial\_ring$$
```
>  sys := [-x^5 + y^5 -3*y -1, 5*y^4 -3, -20*x + y -z];
```
$$sys := [-x^5 + y^5 - 3\,y - 1, 5\,y^4 - 3, -20\,x + y - z]$$
```
>  dec := Triangularize(sys, R);
```
$$dec := [regular\_chain]$$
```
>  length(convert(map(Equations,dec,R),string));
```
$$654$$
```
>  gb := Groebner:-gbasis(sys,plex(x,y,z)):
>  length(convert(gb,string));
```
$$8672$$

On the contrary to the polynomial set `gb`, the regular chain `dec[1]` is not a reduced Gröbner basis of the input system. However, the set `gb` is a regular chain and can be obtained such as by using the option `normalized=yes` of `Triangularize`. In addition, it is possible to obtain from this normalized regular chain (also called "triangular set" in [42, 38, 87]) a more compact regular chain using the *transformation* of Dahan and Schost [42], as shown below. Again, we only show the sizes.

```
>  dec := Triangularize(sys, R, normalized=yes);
```
$$dec := [regular\_chain]$$
```
>  length(convert(map(Equations,dec,R),string));
```
$$8674$$
```
>  dec2 := map(DahanSchostTransform, dec, R);
```
$$dec2 := [regular\_chain]$$
```
>  length(map(Equations,dec2,R),string);
```
$$1692$$

7.3 Challenges in Implementing Triangular Decompositions

This section presents the main difficulties arising during the conception and the implementation of a polynomial system solver based on triangular decompositions. Among those are:

- the sophisticated notions and rich properties attached to triangular decompositions,

- the prototyping of the algorithms, and their sub-routines for computing triangular decompositions

- the validation and user-interface of such a solver.

Depending on the implementation environment, these difficulties must be treated differently, depending on:

- the strengths and weaknesses of this environment,

- the level of expertise and expectation of its community of users.

In general, triangular decompositions can reveal geometrical information of the solution sets better than other symbolic descriptions of polynomial systems such as Gröbner bases. However, the specifications and algorithms for computing triangular decompositions are quite sophisticated, which impose great challenges on their implementation in mathematical software environments, their accessibility and ease of use for users with various interests.

For an input system of polynomials F with rational coefficients, both a Gröbner basis and a triangular decomposition of F give the full set of the complex solutions of F. Consider the polynomial system F_1 with the variables $x > y > z$:

$$\begin{cases} x^3 - 3x^2 + 2x & = & 0 \\ 2yx^2 - x^2 - 3yx + x & = & 0 \\ zx^2 - zx & = & 0 \end{cases}$$

It has lexicographical Gröbner basis:

$$\begin{cases} x^2 - xy - x \\ -xy + xy^2 \\ zxy \end{cases}$$

and triangular decomposition:

$$\{ x = 0 \} \ \bigcup \ \begin{cases} x = 1 \\ y = 0 \end{cases} \ \bigcup \ \begin{cases} x = 2 \\ y = 1 \\ z = 0 \end{cases}$$

It is clearly shown that it consists of one point $(x = 2, y = 1, z = 0)$, one line $(x = 1, y = 0)$, and one plane $(x = 0)$.

This example reveals the first challenge in implementing triangular decompositions, that is, the representation of triangular decompositions. It is a list of lists of polynomials with special properties, instead of just a list of polynomials as for Gröbner basis.

In addition, for the same input polynomial system, there are different possible output triangular decompositions. However, there is a canonical form of output (if the system has finitely many solutions), called the equiprojectable decomposition [39], which can be computed from another triangular decomposition, if needed. In practice these different outputs are of varied benefits, but this makes it harder to specify the results.

Let us illustrate by an example. For the following input polynomial system F_2,

$$F_2 : \begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases}$$

One possible triangular decomposition of the solution set of F_2 is:

$$\begin{cases} z = 0 \\ y = 1 \\ x = 0 \end{cases} \ \bigcup \ \begin{cases} z = 0 \\ y = 0 \\ x = 1 \end{cases} \ \bigcup \ \begin{cases} z = 1 \\ y = 0 \\ x = 0 \end{cases} \ \bigcup \ \begin{cases} z^2 + 2z - 1 = 0 \\ y = z \\ x = z \end{cases}$$

Another one is:

$$\begin{cases} z = 0 \\ y^2 - y = 0 \\ x + y = 1 \end{cases} \ \bigcup \ \begin{cases} z^3 + z^2 - 3z = -1 \\ 2y + z^2 = 1 \\ 2x + z^2 = 1 \end{cases}$$

Both results are valid. The second one is the equiprojectable decomposition. As a matter of fact, the second one can be computed from the first one by the `RegularChains:-EquiprojectableDecomposition` function based on the techniques explained in [39]. However, there is no canonical form of output yet when there is an infinite number of solutions. Besides, different solvers may produce different valid forms.

Although triangular decompositions display rich geometrical information, the solutions can be hard to read, especially when there is an infinite number of solutions; This problem has not been properly solved yet. See below for an example, which also illustrates the differences between two kinds of triangular decompositions: one is in the sense of Kalkbrener, and another one is in the sense of Lazard.

Given a polynomial system F_1 having two polynomials with a variable order of $x > y > a > b > c > d > e > f$:

$$\begin{cases} ax + cy - e &= 0 \\ bx + dy - f &= 0 \end{cases}$$

The triangular decomposition of F_1 in the sense of Kalkbrener consists of one regular chain, which is

$$\begin{cases} bx + dy - f \\ (da - cb)\, y - fa + eb \end{cases}$$

The triangular decomposition of F_1 in the sense of Lazard consists of eleven regular chains, which are

$$\begin{cases} bx + dy - f \\ (da - cb)\, y - fa + eb \end{cases}, \begin{cases} ax + cy - e \\ dy - f \\ b \end{cases}, \begin{cases} bx + dy - f \\ da - cb \\ fc - ed \end{cases}, \begin{cases} dy - f \\ a \\ b \\ fc - ed \end{cases}, \begin{cases} bx - f \\ fa - eb \\ c \\ d \end{cases},$$

$$\begin{cases} ax + cy - e \\ b \\ d \\ f \end{cases}, \begin{cases} bx + dy \\ da - cb \\ e \\ f \end{cases}, \begin{cases} cy - e \\ a \\ b \\ d \\ f \end{cases}, \begin{cases} y \\ a \\ b \\ e \\ f \end{cases}, \begin{cases} x \\ c \\ d \\ e \\ f \end{cases}, \begin{cases} a \\ b \\ c \\ d \\ e \\ f \end{cases}.$$

Due to the above reasons, code validation and output verification for such complex symbolic solvers is extremely difficult. Moreover, the software packages are very large. For example, the `RegularChains` Library in MAPLE consists of $50,000$ lines of code. The command for solving involves almost all the functions in the library and invokes the sophisticated operations on matrices and polynomials, etc.

In 1987, Wen-Tsün Wu [146] introduced the first algorithm computing triangular decompositions of systems of algebraic equations, by means of the so-called "characteristic sets". Kalkbrener [76] provided an algorithm where he considered particular characteristic sets, namely regular chains, leading to theoretical and practical improvements. See also the work of Wang [140], and the work of Lazard and his students [8].

Our study employs the algorithm called Triade. This algorithm relies more intensively on geometrical considerations than the previous ones for computing triangular decompositions, leading to an efficient management of the intermediate computations and control of expression swell. Lazy evaluation techniques and a *task manager* paradigm are also essential tools in this algorithm.

The implementation challenges on Triade are summarized as follows. The first challenge is the prototyping. Indeed, most operations rely on automatic case discussion and the computation may split into sub-cases; see Section 7.4.1 for this point. Secondly, it has to be decided which functionalities will be provided to the end users, and this will affect the ease of use of the package. This choice depends on the computer algebra system and the different communities of users. Thirdly, code validation is particularly difficult, because checking the computations in the case of triangular decomposition is much harder than that for Gröbner bases computations. Therefore, we need more advanced techniques, such as validating packages, comparison with other softwares, and large test suites. We dedicate Chapter 9 to our solution for the verificaition of polynomial system solvers. Finally, performance and optimization of the implementations have special features. Again, they rely largely on the underlying computer algebra system and the requirements of different groups of users. For instance, the data representation used for polynomials and regular chains affects the efficiency.

7.4 Comparison between Three Implementations

This section presents a comparison between three implementations of Triade in three computer algebra systems: AXIOM, ALDOR, and MAPLE, with focus on their advantages and weaknesses in terms of efficiency, ease of use and targeted audience.

AXIOM has been designed to express the extremely rich and complex variety of structures and algorithms in computer algebra; the AXIOM implementation by Moreno Maza (65,000 lines of code) of the Triade algorithm matches its theoretical specifications; it is meant for researchers in the area of symbolic computation, and is available in open source.

ALDOR is an extension of the AXIOM system with a focus on interoperability with other languages and high-performance computing. In ALDOR, the Triade implementation (110,000 lines of code) is available in the form of specialized servers which can solve polynomial systems with frequently used rings of coefficients; these servers have been successfully used by researchers in theoretical physics and algebraic problems. Today, a parallel implementation of Triade in ALDOR is under development on multiple processor machines with shared memory.

MAPLE is a general purpose computer algebra system with users from broad areas (students, engineers, researchers, etc.). The MAPLE implementation of the Triade algorithm (50,000 lines of code) is available since the release 10 of MAPLE as the `RegularChains` library. Non-expert users can access easily a first group of easy-to-use functionalities for computing triangular decompositions and studying the properties of the solutions of polynomial systems. Expert users can take advantage of many options of these functionalities. In addition, sub-modules of the `RegularChains` provide advanced features such as automatic case discussion in parametric problems, and linear algebra over non-integral domains.

7.4.1 The AXIOM Implementation

The AXIOM designers attempted to overcome the challenges of providing an environment for implementing the extremely rich relationships among mathematical structures [72]. Hence, their design is of somewhat different direction from that of other computer algebra systems.

The AXIOM computer algebra system possesses an interactive mode for user interactions and the SPAD language for building library modules. This language has a two-level object model of *categories* and *domains*, which 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 categories and domains, as is usually required in algebra.

The SPAD language has also a functional programming flavor: 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, one can write a SPAD function q which takes as arguments a commutative ring R and an element $p \in R$ such that $q(R, p)$ implements the quotient ring R/pR.

These features allowed us to implement the Triade algorithm in its full generality, that is without any restrictions w.r.t. the theory presented in [108]. In particular, our code can be used with any multivariate polynomial data-type over any field of coefficients available in AXIOM.

One important characteristic of the algorithms producing triangular decompositions is the fact that the intermediate computations require many polynomial coefficient types leading to potentially many type conversions. More precisely, the typical procedure, say `proc`,

- takes as input a quotient ring Q of the form $\mathbb{K}[X]/\mathcal{I}$, where $\mathcal{I}$ is an ideal of $\mathbb{K}[X]$, and elements of Q, say f, g, and

- returns a list of pairs $(Q_1, h_1), \ldots, (Q_s, h_s)$ where Q_j is a quotient ring $\mathbb{K}[X]/\mathcal{I}_j$ and h_j is an element of Q_j, for all $1 \leq j \leq s$.

In the *Dynamic Evaluation* packages in AXIOM [65, 49] the signature of a function implementing `proc` would match the specializations of `proc` precisely; in particular, the types $Q, Q_1, \ldots, Q_s$ would be instantiated at run-time. In the implementation of the Triade algorithm the quotient rings $Q, Q_1, \ldots, Q_s$ are not built explicitly; instead, they are represented by the ideals $\mathcal{I}, \mathcal{I}_1, \ldots, \mathcal{I}_s$, and the polynomials $f, g, h_1, \ldots, h_s$ are encoded by representatives in $\mathbb{K}[X]$. This latter approach may look less elegant than the former one. However, as reported in [107], it brings performance improvement, by avoiding type instantiations and conversions; in addition, it offers more opportunities for optimizations, by homogenizing the type of the intermediate quantities. This approach was reused in the ALDOR and MAPLE implementations of the Triade algorithm.

As discussed in Section 7.3, another challenge in implementing triangular decompositions is code validation. Because a given input system $F \subset \mathbb{K}[X]$ may admit different triangular decompositions, it is hard to use one implementation of these decompositions to validate another. The safest approach, as mentioned in Appendix

A, is through computations based on Gröbner bases. However, this leads to computations which, in practice and in theory, are much more expensive than those of triangular decompositions. This difficulty was resolved by interfacing AXIOM with a high-performance software package [53] for computing Gröbner bases; see [107, 9] for details.

The AXIOM implementation of the Triade algorithm has been integrated in 1998 in the release 2.2 of AXIOM [131]. Experiments reported in [9] show that it often outperforms comparable solvers. Moreover, combined with another symbolic solver [116], it provides functionalities for isolating real roots of polynomial systems symbolically: AXIOM was the first general purpose computer algebra system offering this feature, which we illustrate by the fragment of AXIOM session shown in Appendix B.

In contrast to other general purpose computer algebra system such as MAPLE, AXIOM is primarily destined for the community of researchers in computer algebra: it requires good programming skills and a strong background in algebra. In particular, every user is potentially an expert and a code developer. As a consequence, the logical organization of the library modules relies simply on the algebraic hierarchies of categories and domains; thus, there is less concern with "ease of use" than in MAPLE.

To summarize, the AXIOM implementation of the Triade algorithm has reached its goals: providing a generic, reliable and quite efficient polynomial system solver by means of triangular decompositions.

7.4.2 The ALDOR Implementation

The ALDOR language was designed to be an extension language for the AXIOM computer algebra system. In addition, an ALDOR program can be compiled into: stand-alone executable programs; object libraries in native operating system formats (which can be linked with one another, or with C or Fortran code to form application programs); portable byte code libraries; and C or Lisp source [25]. Aggressive code optimizations by techniques such as program specialization, cross-file procedural integration and data structure elimination, are performed at intermediate stages of compilation [143]. This produces code that is comparable to hand-optimized C.

For these reasons we have used ALDOR to develop high-performance implementations of the Triade algorithm since 1999. More recently, we have realized a parallel implementation [101, 102] on a multiprocessor machine using shared memory for data-communication.

Our ALDOR implementation is much less generic than our AXIOM implementation. First, it is limited to particular, and frequently used, coefficient fields, such as $\mathbb{Q}$, the field of rational numbers, and finite fields. Secondly, it is available in form of executable binary programs, like an operating system command. These "servers" are quite easy to use, but they perform only very specific tasks; in particular, they offer a very limited user interaction. However, their computational power outperforms the AXIOM implementation and they were used to solve difficult problems in theoretical physics [55] and in invariant theory [80].

To summarize, our ALDOR implementation of the Triade algorithm is reaching its main objective: high-performance computing.

7.4.3   The RegularChains Library in MAPLE

MAPLE [105] is a general-purpose computer algebra system. It offers an interpreted, dynamically typed programming language. MAPLE has a very large audience among the world. It is used by engineers, researchers as well as students, in much different topics such as engineering, finance, statistics, education, etc. MAPLE is shipped with a wide variety of libraries dealing, for instance, with linear algebra, differential equations solving, numerical computations. MAPLE is intended to be powerful and easy to use for the high end user. We have realized a MAPLE implementation of the Triade algorithm, the `RegularChains` library [91], which is shipped with the MAPLE software since the version 10 of MAPLE, released in 2005.

In contrast to AXIOM and ALDOR, the MAPLE programming language does not have a strong object-oriented flavor. Code organization and validation are, therefore, even more challenging in this context. So, we describe our effort in these directions for implementing the `RegularChains` library. MAPLE libraries are usually organized as follows:

- a user-interface level providing functionalities accessible to the end-user in the interactive mode; those functions usually check the input specifications,

- an internal level providing functionalities accessible only to the library programmers; they are called by the user-interface functionalities for doing the actual computations.

The data structures are quite straightforward. The most complex data used are the multivariate polynomials. We have chosen the native MAPLE polynomials which are directed acyclic graphs (DAG). This choice has been made for simplicity reasons.

Indeed, all MAPLE directives manipulating polynomials handle DAG. All other objects, as regular chains, have been implemented with lists. Moreover, all structures have been enriched with extra information of two different kinds. The first kind are cached results which are computed frequently (for example the leading variable of a polynomial). The second kind are flags that help optimizing certain functions (for example, knowing that a regular chain represents an irreducible component helps speeding-up computations).

The source code organization is rather standard too. We have split the library source into different files, each one representing a different class of objects. The objective was to mimic the AXIOM/ALDOR organization into categories and domains. This split is very handy, because it emulates some kind of generic programming. Indeed, if we want to use a different representation for polynomials, we only need to change the file implementing polynomials. This makes it easy to compare the efficiency of two different polynomial representations. As for prototyping, the internal functions have been organized similarly to AXIOM/ALDOR, as discussed in Section 7.4.1.

The `RegularChains` user-interface has been designed to provide ease of use to the non-expert and advanced functionalities to the expert. The library offers numerous primitives for computing and manipulating triangular decompositions. For instance, it provides a rich variety of coefficient fields: $\mathbb{Q}$, the field of rational functions, prime fields, fields of rational functions over $\mathbb{Q}$ and prime fields. This is an additional challenge in the MAPLE framework, which has limited support for generic programming.

Combining ease of use and variety of advanced functionalities is achieved by a two-level organization of the user-interface. The first level provides the basic functionalities easy to use for the non-expert. Those functionalities allow to compute triangular decompositions and manipulate polynomials. The second level of the user-interface provides more technical functionalities that are available through optional arguments of the basic functionalities and through two submodules, called `ChainTools` and `MatrixTools`. Those two sub-libraries respectively provide tools for manipulating triangular decompositions and regular chains, and for doing linear algebra over non-integral domains. This makes MAPLE the unique computer algebra system offering automatic case discussion and recombination, as illustrated by the fragment of MAPLE session in Appendix B.

The code validation is made through the MAPLE library test suites. A test suite for `RegularChains` checks all the user interface functions, in order to validate any changes that would be made to the code and the user-interface. Also, the primitive for

computing triangular decomposition, which is a crucial functionality, is tested through a large set of problems. The outputs are partially checked in positive dimension by checking that the radical of the input system is included in the radical of (the saturated ideal of) each regular chain in the output. This ensures that we haven't lost any solution. A complete check in positive dimension has not been done as for those of the AXIOM or ALDOR implementations. However, the checking in zero dimension has been done very thoroughly. Indeed, the output decomposition is processed in a special way. We first make the decomposition radical (each regular chain of the output is made radical), which removes multiple roots. Then the decomposition is processed in such a way that all regular chains of the decomposition have distinct roots. This ensures that the total number of solutions n_o without multiplicity of the decomposition is exactly the sum of the number of solutions of each regular chain (which is just the product of the leading degrees). Thus, if the input system is reduced to zero by each regular chain of the decomposition, we know that the solutions of the input are solutions of the decomposition. Therefore, if we know the number of solutions of the input in advance, and if it is equal to n_o, we are sure that no solutions have been lost or added, which means that the decomposition is correct.

7.5   Summary

Here are some highlights and additional comments regarding three implementations of the Triade algorithm in AXIOM, ALDOR and MAPLE.

The AXIOM implementation has been developed in a very general manner in the sense the design is very close to the mathematical theory. This makes it powerful and flexible. The drawback is that it is not suitable for high-performance, and hard to use for non-experts.

The ALDOR implementation is less general than the AXIOM implementation but has several advantages. First of all, the ALDOR compiler produces binaries which can act as servers or regular applications. This makes it easier for interfacing the Triade solver with other software. Moreover, ALDOR provides an efficient interface with the machine resources leading to higher performances.

Both ALDOR and AXIOM implementations are organized into categories and domains, and lots of functionalities can be used and extended. Therefore, they are well adapted for expert users who aim at developing new algorithms and performing advanced experimentations.

The MAPLE implementation `RegularChains` is different from the ALDOR and

AXIOM ones, in numerous ways. First of all, MAPLE has a larger audience of users, and is aimed at being user friendly. `RegularChains` is written in this spirit and is very easy to use for non-experts. Advanced users are still able to make more complicated computations by using optional arguments and the two submodules `ChainTools` and `MatrixTools`. Secondly, the MAPLE programming language is interpreted and dynamically typed. The language syntax is straightforward and thus not difficult to write MAPLE code. However, the code validation and maintenance is much harder because type errors are only detected at execution. Therefore, coding requires a lot of care and discipline.

Despite of this difficulty, it appears in practice that contributions from students and collaborators are usually made to the MAPLE implementation rather than to its ALDOR and AXIOM counterparts. This is clearly due to the ease of use of the `RegularChains` library. Consequently, some recent and efficient algorithms have been implemented only in the `RegularChains` library. For instance, the modular algorithm presented in Chapter 4 was implemented in MAPLE but not in ALDOR. This is why, on some test problems, the `RegularChains` library can outperform the ALDOR and AXIOM implementations of the Triade algorithm.

Chapter 8

Efficient Computation of Irredundant Triangular Decompositions

This chapter presents new functionalities that we have added to the `RegularChains` library in MAPLE to efficiently compute irredundant triangular decompositions, and reports on the implementation of different strategies. Our experiments show that, for difficult input systems, the computing time for removing redundant components can be reduced to a small portion of the total time needed for solving these systems.

8.1 Introduction

Efficient symbolic solving of parametric polynomial systems is an increasing need in robotics, geometric modeling, stability analysis of dynamical systems and other areas. Triangular decomposition provides a powerful tool for these systems. However, for parametric systems, and more generally for systems in positive dimension, these decompositions have to face the problem of *removing redundant components*. This problem is not limited to triangular decompositions and is also an important issue in other symbolic decomposition algorithms such as those of [140, 88] and in numerical approaches [126].

When decomposing a polynomial system, removing redundant components at an early stage of the solving process is necessary to avoid redundant computations and hence improves the performance. This matter is discussed in [9].

Removing redundant components is also a requirement for solving certain prob-

lems. For instance, with the following question: given an integer n and a parametric system $F \in \mathbb{Q}[U][X]$, determine the values of the parameters U for which F has exactly n real roots. Such problems arise in the stability analysis of dynamical systems [137].

In this work, different criteria and algorithms for deciding whether a quasi-component is contained in another are studied and compared. Then, based on these tools, we obtain several algorithms for removing redundant components in a triangular decomposition. The implementation of these different solutions are realized within the `RegularChains` library [91].

Among the new functionalities that we have added to the `RegularChains` library for solving and manipulating quasi-algebraic systems, we focus in this report on the inclusion test of quasi-components. Section 8.2 provides different algorithmic solutions for this problem. Different strategies for performing the removal of redundant components are described in Section 8.3. A *divide and conquer* approach is used for efficiently removing the redundant components in a triangular decomposition (i.e. a set of regular chains). The experimentation and comparison with these strategies is summarized in Section 8.4.

8.2   Inclusion Test of Quasi-components

It is well-known that inclusion of quasi-algebraic sets reduces to radical membership of polynomial ideals, see for instance [125]. For polynomial ideals given by generators, this latter problem reduces to a Gröbner basis computation, see for instance [36]. When the ideal under consideration is the saturated ideal of the regular chain T, checking whether the polynomial p belongs to the radical of $\mathrm{Sat}(T)$ or not reduces to GCD computations and pseudo-division [76].

In this section we describe our strategies for the inclusion test of quasi-components based on the `RegularChains` library. We refer to [8, 108, 91] for the notion of a regular chain, its related concepts, such as initial, saturated ideals, quasi-components and the related operations.

Let $T, U \subset \mathbb{K}[X]$ be two regular chains. Let h_T and h_U be the respective products of their initials. In this section we discuss how to decide whether the inclusion $W(T) \subseteq W(U)$ holds or not. The proofs of the propositions stated here are standard and similar, so we provide only the one of the most significant result. We aim at relying on the `RegularChains` library, which implies avoiding Gröbner basis computations. Unproved algorithms for this inclusion test are stated in [86] and [107]; they

appeared not to be satisfactory in practice, since they are relying on normalized regular chains, which tend to have much larger coefficients than non-normalized regular chains as verified experimentally in [9] and formally proved in [42].

Proposition 8.2.1. *The inclusion* $W(T) \subseteq W(U)$ *holds if and only if the following both statements hold*

(C_1) *for all* $p \in U$ *we have* $p \in \sqrt{\mathrm{Sat}(T)}$,

(C_2) *we have* $W(T) \cap V(h_U) = \emptyset$.

If $\mathrm{Sat}(T)$ is radical, then condition (C_1) can be replaced by:

(C_1') *for all* $p \in U$ *we have* $p \in \mathrm{Sat}(T)$,

which is easier to check. Checking (C_2) can be approached in different ways, depending on the computational cost that one is willing to pay. Recall in Chapter 10.1 we describe an operation $\mathsf{Intersect}(p, T)$ which takes a polynomial p and a regular chain T and returns regular chains $T_1, \ldots, T_e$ such that we have

$$V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}.$$

A call to Intersect can be seen as relatively cheap, since $\mathsf{Intersect}(p, T)$ exploits the fact that T is a regular chain. Checking

(C_h) $\mathsf{Intersect}(h_U, T) = \emptyset$,

is a good criterion for (C_2).

However, when $\mathsf{Intersect}(h_U, T)$ does not return the empty list, we cannot conclude. To overcome this limitation, we rely on Proposition 8.2.2 and the operation Triangularize of the `RegularChains` library. For a polynomial system $F \subset \mathbb{K}[X]$, $\mathsf{Triangularize}(F)$ returns regular chains $T_1, \ldots, T_e$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$.

Proposition 8.2.2. *The inclusion* $W(T) \subseteq W(U)$ *holds if and only if the following both statements hold*

(C_1) *for all* $p \in U$ *we have* $p \in \sqrt{\mathrm{Sat}(T)}$,

(C_2') *for all* $S \in \mathsf{Triangularize}(T \cup \{h_U\})$ *we have* $h_T \in \sqrt{\mathrm{Sat}(S)}$.

PROOF. Let us denote $V(T), V(h_T), V(U), V(h_U)$ by A, B, C, D respectively. The complement of a subset E of $\overline{\mathbb{K}}^n$ is denoted by E^c. The inclusion $W(T) \subseteq W(U)$ rewrites to $A \setminus B \subseteq C \setminus D$, that is, $A \setminus B \cap (C \setminus D)^c = \emptyset$, that is finally: $A \cap B^c \cap C^c = \emptyset$ and $A \cap B^c \cap D = \emptyset$. Translating back $A \cap B^c \cap C^c = \emptyset$ in terms of $V(T), V(h_T), V(U), V(h_U)$ we retrieve the condition (C_1). Checking the condition $A \cap D \cap B^c = \emptyset$ is equivalent to check whether $V(T) \cap V(h_U) \subseteq V(h_T)$ holds, that is $h_T \in \sqrt{\langle T, h_U \rangle}$. The conclusion follows. □

Remark 8.2.3. Proposition 8.2.2 provides an effective algorithm for testing the inclusion $W(T) \subseteq W(U)$. However, the cost for computing $\mathsf{Triangularize}(T \cup \{h_U\})$ is clearly higher than that for $\mathsf{Intersect}(h_U, T)$, since the former operation cannot take advantage of the fact that T is a regular chain. In $\mathsf{Triangularize}(T \cup \{h_U\})$, T is just a polynomial set, and $\mathsf{Triangularize}(T, h_U)$ will decompose the whole variety $V(T) \cap V(h_U)$, not just $W(T) \cap V(h_U)$. This approach clearly leads to a computational overhead. Indeed, the ideal generated by $\langle T \rangle$ may have a *more complex structure* than $\mathrm{Sat}(T)$. This happens, in particular, when $\langle T \rangle$ is not equidimensional. Consider for instance with $n = 4$ the regular chain $T = X_1 - X_2, X_2 X_3 - X_1, X_1 X_4 - X_2$. The ideal generated by T has two associated primes $\mathcal{P}_1 = \langle X_1, X_2 \rangle$ and $\mathcal{P}_2 = \langle X_4 - 1, -1 + X_3, -X_1 + X_2 \rangle$. We have in fact $\langle T \rangle = \mathcal{P}_1 \cap \mathcal{P}_2$, whereas $\mathrm{Sat}(T) = \mathcal{P}_2$.

8.3 Removing Redundant Components in Triangular Decompositions

Let $F \subset \mathbb{K}[X]$ and let $\mathcal{T} = T_1, \ldots, T_e$ be a *triangular decomposition* of $V(F)$, that is, a set of regular chains such that we have $V(F) = W(T_1) \cup \cdots \cup W(T_e)$. We aim at removing every T_i such that there exists T_j, with $i \neq j$ and $W(T_i) \subseteq W(T_j)$.

Remark 8.3.1. Let $T_{i,1}, \ldots, T_{i,e_i}$ be regular chains such that $W(T_i) \subseteq W(T_{i,1}) \cup \cdots \cup W(T_{i,e_i}) \subseteq \overline{W(T_i)}$ holds. Then, replacing T_i by $T_{i,1}, \ldots, T_{i,e_i}$ in $T_1, \ldots, T_e$ leads again to a triangular decomposition of $V(F)$.

Based on the results of Section 8.2 and Remark 8.3.1, we have developed three strategies: two heuristic ones and a deterministic one. Let $T, U \subset \mathbb{K}[X]$ be two regular chains. We describe these strategies below.

`heuristic-no-split:` It checks whether (C_1') and (C_h) hold. If both hold, then $W(T) \subseteq W(U)$ has been established and $[\mathtt{true}, T, U]$ is returned. Otherwise, no conclusions can be made and $[\mathtt{false}, T, U]$ is returned.

`heuristic-with-split`: It tests the conditions (C_1) and (C_h). Checking (C_1) is achieved by means of the operation Regularize [91, 108]: for a polynomial p and a regular chain T, $\mathsf{Regularize}(p, T)$ returns regular chains $T_1, \ldots, T_e$ such that we have

- $W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(T)}$,

- for each $1 \leq i \leq e$ the polynomial p is either 0 or regular modulo $\mathrm{Sat}(T_i)$,

- no drop of dimension in any of $T_1, \ldots, T_e$.

Therefore, Condition (C_1) holds iff for all T_i returned by $\mathsf{Regularize}(p, T)$ we have $p \equiv 0 \bmod \mathrm{Sat}(T_i)$. For those T_i for which this does not hold we return $[\texttt{false}, T_i, U]$. For the others: if $\mathsf{Intersect}(h_u, T_i)$ returns the empty list (which implies $W(T_i) \cap V(h_u) = \emptyset$) then we return $[\texttt{true}, T_i, U]$ otherwise we return $[\texttt{false}, T_i, U]$ (which does not imply $W(T_i) \cap V(h_u) = \emptyset$).

`certified`: It checks conditions (C_1) and (C_2'). If both hold, then $W(T) \subseteq W(U)$ has been established and $[\texttt{true}, T, U]$ is returned. If at least one of the conditions (C_1) or (C_2') does not hold, then the inclusion $W(T) \subseteq W(U)$ does not hold either and $[\texttt{false}, T, U]$ is returned.

Divide and Conquer Approach. Let `inclusion-test` be one of the tests `heuristic-no-split`, `heuristic-with-split` or `certified`. In order to describe the general mechanism by which redundant components are removed, we define a function $\mathsf{RCCompare}$ based on `inclusion-test` and working as follows: On input (T, U), passed to `inclusion-test`, it returns all the T_i from the tuples $[\texttt{false}, T_i, U]$ and discard those from the $[\texttt{true}, T_i, U]$. For removing the redundant components in the triangular decomposition of $V(F)$ which is a set of regular chains, we use a *divide and conquer* approach. See Figure 8.1 for a sketch of the algorithm.
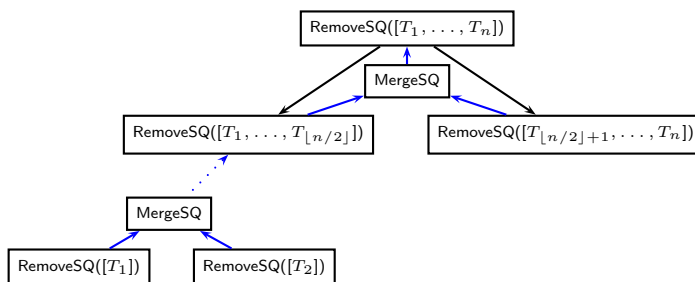
Figure 8.1: Divide and Conquer Approach for Removing Redundant Components in a Triangular Decomposition

1. RemoveSQ($\mathcal{T}$) takes as input a triangular decomposition $\mathcal{T}$ of some quasi-algebraic set Q and returns a triangular decomposition $\mathcal{S}$ of Q such that the quasi-components of $\mathcal{S}$ are pairwise noninclusive.

2. MergeSQ($\mathcal{T}_1, \mathcal{T}_2$) takes as input two triangular decompositions $\mathcal{T}_1$ and $\mathcal{T}_2$ of quasi-algebraic sets Q_1 and Q_2, where each of them is already irredundant; then returns a triangular decomposition $\mathcal{S}$ of $Q_1 \cup Q_2$ such that the quasi-components of $\mathcal{S}$ are pairwise noninclusive.

3. Compare($\mathcal{T}_1, \mathcal{T}_2$) takes as input two list of regular chains $\mathcal{T}_1$ and $\mathcal{T}_2$, where each of them is already noninclusive; then returns the regular chains from $\mathcal{T}_1$ which are not included in $\mathcal{T}_2$.

Now we give a short explanation about the base case of the algorithm and highlight some key points. Concrete comparison work is done only in cases that the input lists contain at most two regular chains. If one regular chain splits when applying inclusion test w.r.t another one, we guarantee the output is noninclusive. To merge two lists of noninclusive regular chains, first we compare in one direction and then update the input for comparing in the other direction.

Figure 8.3 presents such a common situation in which a quasi-component $W(T_1)$ is the union of two quasi-components $W(T_A)$ and $W(T_B)$ and a quasi-component $W(T_2)$ is the union of the quasi-components $W(T_B)$ and $W(T_C)$. The data flow of the algorithm is pictured in Figure 8.3 where the dotted red arrow highlights the updating step. All real computations are done in *MergeSQ* function which intrigues the *RCCompare* function. Note that the final result relies on the order of the input list of regular chains. Here the output is $[T_A, T_2]$, whereas if we compare $[T_2, T_1]$ first then the output would be $[T_B, T_1]$. In both cases, we obtain irredundant representations of the quasi-variety $W(T_1) \cup W(T_2)$.
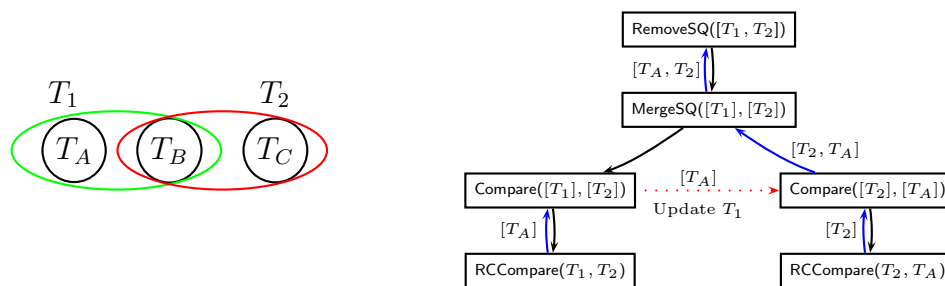
Figure 8.2: Base Case: Removing Redundant Components in Two Triangular Sets

The details of the divider and conquer algorithms are described in Algorithms 44, 45 and 46 .

Algorithm 44 Remove Redundant Quasi-components

Input $\mathcal{T} = [T_1, \ldots T_n]$: a triangular decomposition of a quasi-algebraic set Q.

Output a triangular decomposition $\mathcal{S}$ of Q such that the quasi-components of $\mathcal{S}$ are pairwise noninclusive.

RemoveSQ$(\mathcal{T}) ==$
1: **if** $n < 2$ **then**
2:    **Return** $\mathcal{T}$;
3: **else**
4:    $z := \llcorner n/2 \lrcorner$;
5:    **return** MergeSQ(RemoveSQ$(\mathcal{T}[1..z])$, RemoveSQ$(\mathcal{T}[z+1.. -1]))$;
6: **end if**

Algorithm 45 Merge Quasi-components

Input $\mathcal{T}_1 = \{T_1, \ldots T_m\}$: an irredundant triangular decomposition of a quasi-algebraic set Q_1.
        $\mathcal{T}_2 = \{T'_1, \ldots T'_n\}$: an irredundant triangular decomposition of a quasi-algebraic set Q_2.

Output a triangular decomposition $\mathcal{S}$ of $Q_1 \cup Q_2$ such that the quasi-components of $\mathcal{S}$ are pairwise noninclusive.

MergeSQ$(\mathcal{T}_1, \mathcal{T}_2) ==$
1: $\mathcal{T}_1 NotIn \mathcal{T}_2 \leftarrow$ Compare$(\mathcal{T}_1, \mathcal{T}_2)$;
2: $\mathcal{T}_2 NotIn \mathcal{T}_1 \leftarrow$ Compare$(\mathcal{T}_2, \mathcal{T}_1 NotIn \mathcal{T}_2)$;
3: **return** $\mathcal{T}_1 NotIn \mathcal{T}_2 \cup \mathcal{T}_2 NotIn \mathcal{T}_1$;

8.4   Experimental Results

This experimentation deals with the removal of redundant (quasi-)components in triangular decompositions of algebraic varieties. Its first objective is to test new functionalities of the `RegularChains` library for computing with quasi-algebraic systems. In particular, we want to challenge their efficiency. The second objective is to compare the different strategies described in Section 8.3.

The input polynomial systems are well-known systems which can be found at [128]. The system names appear in the second column of Table 8.1. For each of them, the

---

Algorithm 46 Compare Quasi-components

    Input $\mathcal{T}_1 = \{T_1, \ \ldots \ T_m\}$: an irredundant triangular decomposition of a quasi-algebraic set Q_1.

           $\mathcal{T}_2 = \{T'_1, \ \ldots \ T'_n\}$: an irredundant triangular decomposition of a quasi-algebraic set Q_2.

  Output the regular chains from $\mathcal{T}_1$ which are not included in $\mathcal{T}_2$

Compare$(\mathcal{T}_1, \mathcal{T}_2)$ ==

1:   $result \leftarrow [\ ]$;
2: **for** $rc1$ in $\mathcal{T}_1$ **do**
3:     **for** $rc2$ in $\mathcal{T}_2$ **do**
4:       $result \leftarrow result \cup \mathsf{RCCompare}(rc1, rc2)$;
5:     **end for**
6: **end for**
7: **output** $result$;

---

zero set has dimension at least one. Indeed, redundant components appear rarely when computing triangular decompositions of zero-dimensional polynomial systems.

The third column of Table 8.1 reports on the number of components and running time with Triangularize **without** removal of redundant components, obtaining a decomposition $\mathcal{T}(F)$ for each input system F.

The fourth column of Table 8.1 (called *Certification*), the second column of Table 8.2 (called *heuristic removal without split*) and the fourth column of Table 8.2 (called *heuristic removal with split*) correspond to the removal of the redundant components performed by the procedure described in Section 8.3 applied to $\mathcal{T}(F)$ and using respectively the inclusion tester `certified`, `heuristic-no-split` and `heuristic-with-split`.

The third and the fifth column of Table 8.2 reports on the number of components and running time when applying the deterministic removal of the redundant components to the output produced by the heuristic removal without split and to the output produced by the heuristic removal with split. Therefore, these data allow us to compare three strategies for a complete (or certified) removal of the redundant components:

- by performing directly a certified removal (fourth column of Table 8.1),

- by performing first a heuristic removal without split, followed by a certified removal (second and third columns of Table 8.2),

Sys	Name	Triangularize (No removal)		Certification Proposition 8.2.2	
		♯ RC	time(s)	♯ RC	time(s)
1	genLinSyst-3-2	20	1.684	17	1.182
2	Butcher	15	9.528	7	0.267
3	MacLane	161	12.733	27	7.144
4	neural	10	14.349	4	8.948
5	Vermeer	6	27.870	5	58.396
6	Liu-Lorenz	23	29.044	16	121.793
7	chemical	7	71.364	5	7.727
8	Pappus	393	37.122	120	141.702
9	Liu-Lorenz-Li	22	1796.622	9	96.364
10	KdV572c11s21	41	8898.024	7	6.980

Table 8.1: Triangularize without Removal and with Certified Removal

- by performing first a heuristic removal with split, followed by a certified removal (fourth and fifth columns of Table 8.2).

We make the following observations:

1. The heuristic removal without split performs very well. First, for all examples, except sys 8, it discovers all redundant components. Second, for all examples, except sys 8, its running time is a relatively small portion of the solving time (third column of Table 8.1).

2. Theoretically, the heuristic removal with split can eliminate more *redundancies* than the other strategies. Indeed, it can discover that a quasi-component is contained in the union of two others, meanwhile these three components are pairwise noninclusive.

3. In practice, the heuristic removal with split does not generate more irredundant components than the heuristic removal without split, except for systems 5 and 6. However, the running time overhead is large (roughly, it multiplies by 3 the solving time).

4. The direct deterministic removal is also quite expensive on several systems (5, 6, 8). Unfortunately, the heuristic removal without split, used as pre-cleaning process does not really improve the cost of a certified removal of the redundant components.

Sys	Heuristic (C_1') and (C_h) (without split)		Certification (Deterministic)		Heuristic (C_1) and (C_h) (with split)		Certification (Deterministic)	
	♯ RC	time(s)	♯ RC	time(s)	♯ RC	time(s)	♯ RC	time(s)
1	17	0.382	17	1.240	17	0.270	17	1.214
2	7	0.178	7	0.259	7	0.147	7	0.325
3	27	3.437	27	8.470	27	3.358	27	8.239
4	4	1.881	4	8.353	4	6.429	4	14.045
5	5	0.771	5	60.108	8	54.455	8	109.928
6	16	1.937	16	123.052	18	96.492	18	203.937
7	5	0.243	5	7.828	5	5.180	5	12.842
8	124	42.817	120	135.780	124	48.756	120	148.341
9	9	8.186	9	101.668	10	105.598	10	217.837
10	7	4.878	7	6.688	7	5.881	7	7.424

Table 8.2: Heuristic Removal, without and with Split, followed by Certification

8.5   Summary

In this work, new functionalities have been added to the `RegularChains` library in MAPLE for solving and manipulating quasi-algebraic systems. This allows the user to compute triangular decompositions without redundant components. Components contained in the union of several other ones can also be removed.

Our implementation has demonstrated the efficiency of these methods. Indeed, the complete removal of the redundant components can always be achieved in a reasonable amount of time comparing to the solving time of the input system. In addition, the heuristic removal without split is very efficient in practice.

Chapter 9

Verification of Polynomial System Solvers

In this chapter, we discuss the verification of mathematical software solving polynomial systems symbolically by way of triangular decomposition. Standard verification techniques are highly resource consuming and apply only to polynomial systems which are easy to solve. We exhibit a new approach which manipulates constructible sets represented by regular systems. We provide comparative benchmarks of different verification procedures applied to four solvers on a large set of well-known polynomial systems. Our experimental results illustrate the high efficiency of our new approach. In particular, we are able to verify triangular decompositions of polynomial systems which are not easy to solve.

9.1   Introduction

Solving systems of non-linear, algebraic or differential equations is a fundamental problem in mathematical science. It has been studied for centuries and has stimulated many research developments. Algorithmic solutions can be classified into three categories: numeric, symbolic and hybrid numeric-symbolic. The choice for one of them depends on the characteristics of the system of equations to solve. For instance, it depends on whether the coefficients are known exactly or are approximations obtained from experimental measurements. This choice depends also on the expected answers, which could be a complete description of all the solutions, or only the real solutions, or just one sample solution among all of them.

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 areas like digital signal processing, robotics, theoretical physics, cryptology, dynamical systems, with many important outcomes. See [67] for an overview of these applications.

Symbolic solvers are also highly complex software. First, they implement sophisticated algorithms, which are generally at the level of on-going research. Moreover, in most computer algebra systems, the `solve` command involves nearly the entire set of libraries in the system, challenging the most advanced operations on matrices, polynomials, algebraic and modular numbers, polynomial ideals, etc.

Secondly, algorithms for solving systems of polynomial equations are by nature of exponential-space complexity. Consequently, symbolic solvers are extremely time-consuming when applied to large examples. Even worse, intermediate expressions can grow to enormous size and may halt the computations, even if the result is of moderate size. The implementation of symbolic solvers, then, requires techniques that go far beyond the manipulation of algebraic or differential equations, such as efficient memory management, data compression, parallel and distributed computing, etc.

Last, but not least, the precise output specifications of a symbolic solver can be quite involved. Indeed, given an input polynomial system F, defining what a symbolic solver should return implies describing what the geometry of the solution set $V(F)$ of F can be. For an arbitrary F, the set $V(F)$ may consist of components of different natures and sizes: points, lines, curves, surfaces. This leads to the following difficult challenge.

Given a polynomial system F and a set of components $C_1, \ldots, C_e$, it is hard, in general, to tell whether the union of $C_1, \ldots, C_e$ corresponds exactly to the solution set $V(F)$ or not. Actually, solving this verification problem is generally (at least) as hard as solving the system F itself.

Because of the high complexity of symbolic solvers, developing verification algorithms and reliable verification software tools is a clear need. However, this verification problem has received little attention in the literature. In this chapter, we present new techniques for verifying a large class of symbolic solvers. We also report on intensive experimentation illustrating the high efficiency of our approach w.r.t. known techniques.

We assume that each component of the solution set $V(F)$ is given by a so-called *regular system*. This is a natural assumption in symbolic computations, well-developed in the literature under different terminologies, see [8, 135] and the references therein. In broad words, a regular system consists of several polynomial equations

with a triangular shape

$$p_1(x_1) = p_2(x_1, x_2) = \cdots = p_i(x_1, x_2, \ldots, x_n) = 0$$

and a polynomial inequality

$$h(x_1, \ldots, x_n) \neq 0$$

such that there exists (at least) one point $(a_1, \ldots, a_n)$ satisfying the above equations and inequality. Note that these polynomials may contain parameters.

Let us consider the following well-known system F taken from [47].

$$\begin{cases} x^{31} - x^6 - x - y &= 0 \\ x^8 - z &= 0 \\ x^{10} - t &= 0 \end{cases}$$

We aim at solving this system for $x > y > z > t$, that is, perversely expressing x as a function of y, z, t, then y as a function of z, t and z as a function of t. One possible decomposition is given by the three regular systems below:

$$\begin{cases} (t^4 - t)\,x - ty - z^2 &= 0 \\ t^3 y^2 + 2t^2 z^2 y + (-t^6 + 2t^3 + t - 1)\,z^4 &= 0 \\ z^5 - t^4 &= 0 \\ t^4 - t &\neq 0 \end{cases}, \quad \begin{cases} x^2 - z^4 &= 0 \\ y + t^2 z^2 &= 0 \\ z^5 - t &= 0 \\ t^3 - 1 &= 0 \end{cases}, \quad \begin{cases} x &= 0 \\ y &= 0 \\ z &= 0 \\ t &= 0 \end{cases}$$

Another decomposition is given by these other three regular systems:

$$\begin{cases} (t^4 - t)\,x - ty - z^2 &= 0 \\ tzy^2 + 2z^3 y - t^8 + 2t^5 + t^3 - t^2 &= 0 \\ z^5 - t^4 &= 0 \\ z\,(t^4 - t) &\neq 0 \end{cases}, \quad \begin{cases} zx^2 - t &= 0 \\ ty + z^2 &= 0 \\ z^5 - t &= 0 \\ t^3 - 1 &= 0 \\ tz &\neq 0 \end{cases}, \quad \begin{cases} x &= 0 \\ y &= 0 \\ z &= 0 \\ t &= 0 \end{cases}$$

These two decompositions look slightly different (in particular, in the second components) and one could think that, if each of them was produced by a different solver, then at least one of these solvers has a bug. In fact, both decompositions are valid, but proving respectively that they encode the solution set $V(F)$ is not feasible without computer assistance. However, proving that they define the same set of points can be achieved by an expert hand without computer assistance. This is an important observation that we will guide us in this work.

Let us consider now an arbitrary input system F and a set of components $C_1, \ldots, C_e$ encoded by regular systems $S_1, \ldots, S_e$ respectively. The usual approach for verifying that $C_1, \ldots, C_e$ correspond exactly to the solution set $V(F)$ is as follows.

(1) First, one checks that each candidate component C_i is actually contained in $V(F)$. This essentially reduces to substitute the coordinates of the points given by C_i into the polynomials of F: if all these polynomials vanish at these points, then C_i is a component of $V(F)$, otherwise, (and up to technical details that we will skip in this introduction) C_i is not a component of $V(F)$.

(2) Secondly, one checks that $V(F)$ is contained in the union of the candidate components $C_1, \ldots, C_e$ by:

    (2.1) computing a polynomial system G such that $V(G)$ corresponds exactly to $C_1, \ldots, C_e$, and

    (2.2) checking that every solution of $V(F)$ cancels the polynomials of G.

Steps (2.1) and (2.2) can be performed using standard techniques based on computations of Gröbner bases, as we discuss in Section 9.6.1. These calculations are very expensive, as shown by our experimentation, reported in Section 9.7.

In this work, we propose a different approach, summarized in non-technical language in Section 9.2. The main idea is as follows. Instead of comparing a candidate set of components $C_1, \ldots, C_e$ against the input system F, we compare it against the output $D_1, \ldots, D_f$ produced by another solver. Both this solver and the comparison process are assumed to be validated. Hence, the candidate set of components $C_1, \ldots, C_e$ corresponds exactly to the solution set $V(F)$ if and only if the comparison process shows that $D_1, \ldots, D_f$ and $C_1, \ldots, C_e$ define the same solution set.

The technical details of this new approach are given in Sections 9.3, 9.4, 9.5 and 9.6. In Section 9.3, we review the fundamental algebraic concepts and operations involved in our work. In particular, we specify the kind of solvers that we consider in this study, namely those solving polynomial systems by means of *triangular decompositions*.

The key computational concept behind these triangular decomposition computed is that of a *constructible set*, so we dedicate Section 9.4 to it. Section 9.5 is a formal and complete presentation of our process for comparing triangular decompositions. In Section 9.6, we summarize the different verification procedures that are available for triangular decompositions, including our new approach. In Section 9.7, we report

on experimentation with these verification procedures. Our data illustrate the high efficiency of our new approach.

9.2 Methodology

Let us consider again an arbitrary input polynomial system F and a set of components $C_1, \ldots, C_e$ encoded by regular systems $S_1, \ldots, S_e$ respectively. As mentioned in the Introduction, checking whether $C_1, \ldots, C_e$ corresponds exactly to the solution set $V(F)$ of F can be done by means of Gröbner bases computations. This verification process is quite simple, see Section 9.6, and its implementation is straightforward. Thus, if the underlying Gröbner bases engine is *reliable*, such verification tool can be regarded as safe. See [9] for details.

Unfortunately, this verification process is highly expensive. Even worse, as shown by our experimental results in Section 9.7, this verification process is unable to check many triangular decompositions that are easy to compute.

We propose a new approach in order to overcome this limitation. Assume that we have at hand a reliable solver computing triangular decompositions of polynomial systems. We believe that this reliability can be acquired over time by combining several features.

- Checking the solver with a verification tool based on Gröbner bases for input systems of moderate difficulty.

- Using the solver for input systems of higher difficulty where the output can be verified by theoretical arguments, see [11] for an example of such input system.

- Involving the library supporting the solver in other applications.

- Making the solver widely available to potential users.

Suppose that we are currently developing a new solver computing triangular decompositions. In order to verify the output of this new solver, we can take advantage of the reliable solver.

This may sound natural and easy in the first place, but this is actually not. Indeed, as shown in the Introduction, two different solvers can produce two different, but valid, triangular decompositions for the same input system. Checking that these two triangular decompositions encode the same solution set boils down to computing the differences of two constructible sets. This is a non-trivial operation, see the survey paper [125].

The first contribution of our work is to provide a relatively simple, but efficient, procedure for computing the set-theoretical differences between two constructible sets. See Section 9.5. Such a procedure can be used to develop a verification tool for our new solver by means of our reliable solver. Moreover, this procedure is sufficiently straightforward to implement that it can be trusted after a relatively short period of testing, as the case for the verification tool based on Gröbner bases computations.

The second contribution of our work is to illustrate the high efficiency of this new verification tool. In Section 9.7, we consider four solvers computing triangular decomposition of polynomial systems:

- the command *Triangularize* of the *RegularChains* library [91] in MAPLE

- the Triade solver of the *BasicMath* library [132] in ALDOR

- the commands *RegSer* and *SimSer* of the *Epsilon* library [136] in MAPLE.

We have run these four solvers on a large set of well-known input systems from the data base [104, 128, 140]. For those systems for which this is feasible, we have verified their computed triangular decompositions with a verification tool based on Gröbner bases computations. Then, for each input system, we have compared all its computed triangular decompositions by means of our new verification tool.

Based on our experimentation data reported in Section 9.7 we make the following observations.

- All computed triangular decompositions, that could be checked via Gröbner bases computations, are correct.

- However, the verification tool based on Gröbner bases computations failed to check many examples by running out of computer memory.

- For each input system F, most pairs of triangular decompositions of F could be compared successfully by our new verification tool.

- Moreover, for any system F to which all verification tools could be applied, our new approach runs much faster.

This suggests that the four solvers and our new verification tool have a good level of reliability. Moreover, our verification tool allows to process cases that were previously out of reach.

9.3 Preliminaries

In this section we introduce notations and review fundamental results in the theory of regular chains and regular systems [8, 22, 76, 108, 135, 140].

We shall use some notions from commutative algebra (such as the dimension of an ideal) and refer for instance to [118] for this subject.

9.3.1 Basic Notations and Definitions

Let $\mathbb{K}[Y] := \mathbb{K}[Y_1, \ldots, Y_n]$ be the polynomial ring over the field $\mathbb{K}$ in variables $Y_1 < \cdots < Y_n$. Let $p \in \mathbb{K}[Y]$ be a non-constant polynomial. The *leading coefficient* and the *degree* of p regarded as a univariate polynomial in Y_i will be denoted by $\mathrm{lc}(p, Y_i)$ and $\deg(p, Y_i)$ respectively. The greatest variable appearing in p is called the *main variable* denoted by $\mathrm{mvar}(p)$. The degree, the leading coefficient, and the leading monomial of p regarding as a univariate polynomial in $\mathrm{mvar}(p)$ are called the *main degree*, the *initial*, and the *rank* of p; they are denoted by $\mathrm{mdeg}(p)$, $\mathrm{init}(p)$ and $\mathrm{rank}(p)$ respectively.

Let $F \subset \mathbb{K}[Y]$ be a finite polynomial set. Denote by $\langle F \rangle$ the ideal it generates in $\mathbb{K}[Y]$ and by $\sqrt{\langle F \rangle}$ the radical of $\langle F \rangle$. Let h be a polynomial in $\mathbb{K}[Y]$, the *saturated ideal* $\langle F \rangle : h^\infty$ of $\langle F \rangle$ w.r.t h, is the set

$$\{q \in \mathbb{K}[Y] \mid \exists m \in \mathbb{N} \text{ s.t. } h^m q \in \langle F \rangle\},$$

which is an ideal in $\mathbb{K}[Y]$.

A polynomial $p \in \mathbb{K}[Y]$ is a *zerodivisor* modulo $\langle F \rangle$ if there exists a polynomial q such that pq is zero modulo $\langle F \rangle$, and q is not zero modulo $\langle F \rangle$. The polynomial is *regular* modulo $\langle F \rangle$ if it is neither zero, nor a zerodivisor modulo $\langle F \rangle$. Denote by $V(F)$ the *zero set* (or solution set, or algebraic variety) of F in $\overline{\mathbb{K}}^n$. For a subset $W \subset \overline{\mathbb{K}}^n$, denote by $\overline{W}$ its closure in the Zariski topology, that is the intersection of all algebraic varieties $V(G)$ containing W for all $G \subset \mathbb{K}[Y]$.

Let $T \subset \mathbb{K}[Y]$ be a *triangular set*, that is a set of non-constant polynomials with pairwise distinct main variables. Denote by $\mathrm{mvar}(T)$ the set of main variables of $t \in T$. A variable in Y is called *algebraic* w.r.t. T if it belongs to $\mathrm{mvar}(T)$, otherwise it is called *free* w.r.t. T. For a variable $v \in Y$ we denote by $T_{<v}$ (resp. $T_{>v}$) the subsets of T consisting of the polynomials t with main variable less than (resp. greater than) v. If $v \in \mathrm{mvar}(T)$, we say T_v is defined. Moreover, we denote by T_v the polynomial in T whose main variable is v, by $T_{\leqslant v}$ the set of polynomials in T with main variables

less than or equal to v and by $T_{\geqslant v}$ the set of polynomials in T with main variables greater than or equal to v.

Definition 9.3.1. Let p, $q \in \mathbb{K}[Y]$ be two nonconstant polynomials. We say $\mathrm{rank}(p)$ is smaller than $\mathrm{rank}(q)$ w.r.t *Ritt ordering* and we write, $\mathrm{rank}(p) <_r \mathrm{rank}(q)$ if one of the following assertions holds:

- $\mathrm{mvar}(p) < \mathrm{mvar}(q)$,

- $\mathrm{mvar}(p) = \mathrm{mvar}(q)$ and $\mathrm{mdeg}(p) < \mathrm{mdeg}(q)$.

Note that the partial order $<_r$ is a well ordering. Let $T \subset \mathbb{K}[Y]$ be a triangular set. Denote by $\mathrm{rank}(T)$ the set of $\mathrm{rank}(p)$ for all $p \in T$. Observe that any two ranks in $\mathrm{rank}(T)$ are comparable by $<_r$. Given another triangular set $S \subset \mathbb{K}[Y]$, with $\mathrm{rank}(S) \neq \mathrm{rank}(T)$, we write $\mathrm{rank}(T) <_r \mathrm{rank}(S)$ whenever the minimal element of the symmetric difference $(\mathrm{rank}(T)\backslash\mathrm{rank}(S)) \cup (\mathrm{rank}(S)\backslash\mathrm{rank}(T))$ belongs to $\mathrm{rank}(T)$. By $\mathrm{rank}(T) \leqslant_r \mathrm{rank}(S)$, we mean either $\mathrm{rank}(T) < \mathrm{rank}(S)$ or $\mathrm{rank}(T) = \mathrm{rank}(S)$. Note that any sequence of triangular sets, of which ranks strictly decrease w.r.t $<_r$, is finite.

Given a triangular set $T \subset \mathbb{K}[Y]$, denote by h_T be the product of the initials of T (throughout the work we use this convention and when T consists of a single element g we write it in h_g for short). The *quasi-component* $W(T)$ of T is $V(T) \setminus V(h_T)$, in other words, 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: if T is empty then $\mathrm{Sat}(T)$ is defined as the trivial ideal $\langle 0 \rangle$, otherwise it is the ideal $\langle T \rangle : h_T^\infty$.

Let $h \in \mathbb{K}[Y]$ be a polynomial and $F \subset \mathbb{K}[Y]$ a set of polynomials, we write

$$Z(F, T, h) := (V(F) \cap W(T)) \setminus V(h).$$

When F consists of a single polynomial p, we use $Z(p, T, h)$ instead of $Z(\{p\}, T, h)$; when F is empty we just write $Z(T, h)$. By $Z(F, T)$, we denote $V(F) \cap W(T)$.

Given a family of pairs $S = \{[T_i, h_i] \mid 1 \leq i \leq e\}$, where $T_i \subset \mathbb{K}[Y]$ is a triangular set and $h_i \in \mathbb{K}[Y]$ is a polynomial. We write

$$Z(S) := \bigcup_{i=1}^{e} Z(T_i, h_i).$$

We conclude this section with some well known properties of ideals and triangular sets. For a proper ideal $\mathcal{I}$, we denote by $\dim(V(\mathcal{I}))$ the dimension of $V(\mathcal{I})$.

Lemma 9.3.2. *Let $\mathcal{I}$ be a proper ideal in $\mathbb{K}[Y]$ and $p \in \mathbb{K}[Y]$ be a polynomial regular w.r.t $\mathcal{I}$. Then, either $V(\mathcal{I}) \cap V(p)$ is empty or we have:* $\dim(V(\mathcal{I}) \cap V(p)) \leq \dim(V(\mathcal{I})) - 1$.

Lemma 9.3.3. *Let T be a triangular set in $\mathbb{K}[Y]$. Then, we have*

$$\overline{W(T)} \setminus V(h_T) = W(T) \quad \text{and} \quad \overline{W(T)} \setminus W(T) = V(h_T) \cap \overline{W(T)}.$$

PROOF.   Since $W(T) \subseteq \overline{W(T)}$, we have

$$W(T) = W(T) \setminus V(h_T) \subseteq \overline{W(T)} \setminus V(h_T).$$

On the other hand, $\overline{W(T)} \subseteq V(T)$ implies

$$\overline{W(T)} \setminus V(h_T) \subseteq V(T) \setminus V(h_T) = W(T).$$

This proves the first claim. Observe that we have:

$$\overline{W(T)} = \left(\overline{W(T)} \setminus V(h_T) \right) \cup \left(\overline{W(T)} \cap V(h_T) \right).$$

We deduce the second one. □

Lemma 9.3.4 ([8, 22])**.** *Let T be a triangular set in $\mathbb{K}[Y]$. Then, we have*

$$V(\mathrm{Sat}(T)) = \overline{W(T)}.$$

Assume furthermore that $W(T) \neq \emptyset$ holds. Then $V(\mathrm{Sat}(T))$ is a nonempty unmixed algebraic set with dimension $n - |T|$. Moreover, if N is the free variables of T, then for every prime ideal $\mathcal{P}$ associated with $\mathrm{Sat}(T)$ we have

$$\mathcal{P} \cap \mathbb{K}[N] = \langle 0 \rangle.$$

9.3.2   Regular Chain and Regular System

Definition 9.3.5 (Regular Chain)**.** A triangular set $T \subset \mathbb{K}[Y]$ is a *regular chain* if one of the following conditions hold:

- either T is empty,

- or $T \setminus \{T_{\max}\}$ is a regular chain, where $T_{\max}$ is the polynomial in T with maximum rank, and the initial of $T_{\max}$ is regular w.r.t. $\mathrm{Sat}(T \setminus \{T_{\max}\})$.

It is useful to extend the notion of regular chain as follows.

Definition 9.3.6 (Regular System)**.** A pair $[T, h]$ is a regular system if T is a regular chain, and $h \in \mathbb{K}[Y]$ is regular w.r.t $\operatorname{Sat}(T)$.

Remark 9.3.7. A regular system in a stronger sense was presented in [135]. For example, consider a polynomial system $[T, h]$ where $T = [Y_1 Y_4 - Y_2]$ and $h = Y_2 Y_3$. Then $[T, h]$ is still a regular system in our sense but not a regular system in Wang's sense. Also we do not restrict the main variables of polynomials in the inequality part. At least our definition is more convenient for our purpose in dealing with zerodivisors and conceptually clear as well. We also note that in zero-dimension case (no free variable exist) the notion of regular chain and that of a regular set in [135] are the same, see [8, 135] for details.

There are several equivalent characterizations of a regular chain, see [8]. In this chapter, we rely on the notion of *iterated resultant* in order to derive a characterization which can be checked by solving a polynomial system.

Proposition 9.3.8. *For every regular system $[T, h]$ we have $Z(T, h) \neq \emptyset$.*

PROOF. Since T is a regular chain, by Lemma 9.3.4 we have $V(\operatorname{Sat}(T)) \neq \emptyset$. By definition of regular system, the polynomial $h h_T$ is regular w.r.t $\operatorname{Sat}(T)$. Hence, by Lemma 9.3.2, the set $V(h h_T) \cap V(\operatorname{Sat}(T))$ either is empty, or has lower dimension than $V(\operatorname{Sat}(T))$. Therefore, the set

$$V(\operatorname{Sat}(T)) \setminus V(h h_T) = V(\operatorname{Sat}(T)) \setminus (V(h h_T) \cap V(\operatorname{Sat}(T)))$$

is not empty. Finally, by Lemma 9.3.3, the set

$$Z(T, h) = W(T) \setminus V(h) = \overline{W(T)} \setminus V(h h_T) = V(\operatorname{Sat}(T)) \setminus V(h h_T)$$

is not empty. $\qquad \square$

Notation 9.3.9. For a regular system $R = [T, h]$, we define $\operatorname{rank}(R) := \operatorname{rank}(T)$. For a set $\mathcal{R}$ of regular systems, we define

$$\operatorname{rank}(\mathcal{R}) := \max\{\operatorname{rank}(T) \mid [T, h] \in \mathcal{R}\}.$$

For a pair of regular systems (L, R), we define $\operatorname{rank}((L, R)) := (\operatorname{rank}(L), \operatorname{rank}(R))$. For a pair of lists of regular systems, we define

$$\operatorname{rank}((\mathcal{L}, \mathcal{R})) = (\operatorname{rank}(\mathcal{L}), \operatorname{rank}(\mathcal{R})).$$

For triangular sets $T, T_1, \ldots, T_e$ we write $W(T) \xrightarrow{D} (W(T_i), i = 1 \ldots e)$ if one of the following conditions holds:

- either $e = 1$ and $T = T_1$,

- or $e > 1$, $\mathrm{rank}(T_i) < \mathrm{rank}(T)$ for all $i = 1 \ldots e$ and

$$W(T) \subseteq \bigcup_{i=1}^{e} W(T_i) \subseteq \overline{W(T)}.$$

9.3.3 Triangular Decompositions

Definition 9.3.10. Given a finite polynomial set $F \subset \mathbb{K}[Y]$, a *triangular decomposition* of $V(F)$ is a finite family $\mathcal{T}$ of regular chains of $\mathbb{K}[Y]$ such that

$$V(F) = \bigcup_{T \in \mathcal{T}} W(T).$$

For a finite polynomial set $F \subset \mathbb{K}[Y]$, the Triade algorithm [108] computes a triangular decomposition of $V(F)$. We list below the specifications of the operations from Triade that we use in this work.

Let p, p_1, p_2 be polynomials, and let T, C, E be regular chains such that $C \cup E$ is a triangular set (but not necessarily a regular chain).

- Regularize(p, T) returns regular chains $T_1, \ldots, T_e$ such that

    - $W(T) \xrightarrow{D} (W(T_i), i = 1 \ldots e)$,
    - for all $1 \leq i \leq e$ the polynomial p is either 0 or regular modulo $\mathrm{Sat}(T_i)$.

- For a set of polynomials F, Triangularize(F, T) returns regular chains $T_1, \ldots, T_e$ such that we have

$$V(F) \cap W(T) \ \subseteq \ W(T_1) \cup \cdots \cup W(T_e) \ \subseteq \ V(F) \cap \overline{W(T)}.$$

    and for $1 \leq i \leq e$ we have $\mathrm{rank}(T_i) < \mathrm{rank}(T)$.

- Extend$(C \cup E)$ returns a set of regular chains $\{C_i \mid i = 1 \ldots e\}$ such that we have $W(C \cup E) \xrightarrow{D} (W(C_i), i = 1 \ldots e)$.

- Assume that p_1 and p_2 are two non-constant polynomials with the same main variable v, which is larger than any variable appearing in T, and assume that

the initials of p_1 and p_2 are both regular w.r.t. $\mathrm{Sat}(T)$. Then, $\mathsf{GCD}(p_1, p_2, T)$ returns a sequence

$$([g_1, C_1], \ldots, [g_d, C_d], [\emptyset, D_1], \ldots, [\emptyset, D_e]),$$

where g_i are polynomials and C_i, D_i are regular chains such that the following properties hold:

- $W(T) \xrightarrow{D} (W(C_1), \ldots, W(C_d), W(D_1), \ldots, W(D_e))$,
- $\dim V(\mathrm{Sat}(C_i)) = \dim V(\mathrm{Sat}(T))$ and $\dim V(\mathrm{Sat}(D_j)) < \dim V(\mathrm{Sat}(T))$, for all $1 \leqslant i \leqslant d$ and $1 \leqslant j \leqslant e$,
- the leading coefficient of g_i w.r.t. v is regular w.r.t. $\mathrm{Sat}(C_i)$,
- for all $1 \leqslant i \leqslant d$ there exists polynomials u_i and v_i such that we have $g_i = u_i p_1 + v_i p_2 \mod \mathrm{Sat}(C_i)$,
- if g_i is not constant and its main variable is v, then p_1 and p_2 belong to $\mathrm{Sat}(C_i \cup \{g_i\})$.

9.4 Representations of Constructible Sets

The constructible set [50, 69] is a classical concept in elimination theory. In this section, we present two types of representations for constructible sets in $\mathbb{K}^n$.

Definition 9.4.1 (Constructible set)**.** A constructible subset of $\mathbb{K}^n$ is a subset which can be represented by a finite union of the intersection of an open algebraic subset with a closed algebraic subset.

Let $\mathcal{F}$ be the set of all constructible subsets of $\mathbb{K}^n$ w.r.t K_0. From Exercise 3.18 in [69], we have

- all open algebraic sets are in $\mathcal{F}$;

- the complement of an element in $\mathcal{F}$ is in $\mathcal{F}$;

- the intersection of two elements in $\mathcal{F}$ is in $\mathcal{F}$.

Moreover, these three properties describe *exactly* all constructible sets.

Given a set of polynomial F and $f \in \mathrm{P}_n$, we denote $D(F, f)$ the difference of $V(F)$ and $V(f)$, which is also called a *basic constructible set*. If F is the empty set, then we write $D(f)$ for short. Note that for a regular system in [135], $D(T, h) = Z(T, h)$ holds.

Gröbner basis representation Now Gröbner bases have become a standard tool to deal with algebraic sets; and they can be applied to manipulate constructible sets as well. Given a constructible set C, according to the definition, one can represent C by a unique sequence of closed algebraic sets whose defining ideals naturally can be characterized by their reduced Gröbner bases [112].

However, the constructible set is a geometrical object intrinsically. We pay extra cost to manipulate them, since it is very hard to compute the intersection of two ideals and even to compute the radical ideal of an ideal. However, there exist effective algorithms to manipulate constructible sets. We shall use regular systems to do the same jobs in a more efficient manner.

Regular system representation In this section, we show that (Theorem 9.4.3) every constructible set C can be represented by a finite set of regular systems $\{[T_i, h_i] \mid i = 1 \ldots e\}$, that is,

$$C = \bigcup_{i=1}^{e} Z(T_i, h_i).$$

Combining with Lemma 9.3.8, we know that if a regular system representation of a constructible set is empty then C is an empty set. This fact leads to an important application of verifying polynomial system solver. The proof of Theorem 9.4.3 is partially constructible which relies on an algorithm called Difference. As a consequence, we reach the following theorem by means of Difference.

Theorem 9.4.2. *Given two regular systems $[T, h]$ and $[T', h']$, there is an algorithm to compute the regular system representations of:*

(1) *the difference $Z(T, h) \setminus Z(T', h')$;*

(2) *the intersection $Z(T, h) \bigcap Z(T', h')$.*

PROOF. (1) follows from the algorithm Difference. Note that given two sets A and B, $A \cap B = A \setminus (A \setminus B)$. (2) follows from a successive call to Difference. □

Theorem 9.4.3. *Every constructible set can be represented by a finite set of regular systems.*

PROOF. Consider the following family $\widetilde{\mathcal{F}}$ of subsets of $\mathbb{K}^n$:

$$\widetilde{\mathcal{F}} = \{S \mid S = \bigcup_{i=1}^{e} Z(T_i, p_i)\},$$

where $[T_i, p_i]$ are regular systems. First, every open subset can be decomposed into the finite union of open subsets $D(f)$ which can be represented by the empty regular chain and f. Hence $\widetilde{\mathcal{F}}$ contains all open subsets. Secondly, consider two elements S and T in $\widetilde{\mathcal{F}}$; and assume that

$$S = \bigcup_{i=1}^{e} Z(S_i, p_i) \quad \text{and} \quad T = \bigcup_{j=1}^{f} Z(T_j, q_j).$$

We have

$$S \bigcap T = \bigcup_{i=1}^{e}\bigcup_{j=1}^{f} \left(Z(S_i, p_i) \bigcap Z(T_j, q_j) \right).$$

By Theorem 9.4.2, $S \bigcap T$ has a regular system representation, that is to say, $S \bigcap T \in \widetilde{\mathcal{F}}$. By induction, any finite intersection of elements of $\widetilde{\mathcal{F}}$ is in $\widetilde{\mathcal{F}}$. Finally, we shall prove that the complement of an element in $\widetilde{\mathcal{F}}$ is in $\widetilde{\mathcal{F}}$. Essentially, we only need to show that for each $1 \le i \le e$, $Z(S_i, p_i)^c$ is in $\widetilde{\mathcal{F}}$. Indeed,

$$Z(S_i, p_i)^c = W(S_i)^c \bigcup V(p_i) = V(S_i)^c \bigcup V(p_i h_{S_i})$$

is in $\widetilde{\mathcal{F}}$, since both $V(S_i)^c$ and $V(p_i h_{S_i})$ have regular system representations. $\qquad\square$

9.5 Difference Algorithms

In this section, we present an algorithm to compute the set theoretical difference of two constructible sets given by regular systems. As mentioned in the Introduction, a naive approach appears to be very inefficient in practice. Here we contribute a more sophisticated algorithm, which heavily exploits the structure and properties of regular chains.

Two procedures, Difference and DifferenceLR, are involved in order to achieve this goal. Their specifications and pseudo-codes can be found below. The rest of this section is dedicated to proving the correctness and termination of these algorithms. For the pseudo-code, we use the MAPLE syntax. However, each of the two functions below returns a sequence of values. Individual value or sub-sequences of the returned sequence are thrown to the flow of output by means of an **output** statement. Hence an **output** statement does not cause the termination of the function execution.

Algorithm 47 Difference($[T, h], [T', h']$)

> **Input** $[T, h]$, $[T', h']$ two regular systems.

Output Regular systems $\{[T_i, h_i] \mid i = 1 \ldots e\}$ such that

$$Z(T, h) \setminus Z(T', h') = \bigcup_{i=1}^{e} Z(T_i, h_i),$$

and $\operatorname{rank}(T_i) \leqslant_r \operatorname{rank}(T)$.

Algorithm 48 DifferenceLR$(\mathcal{L}, \mathcal{R})$

    Input $\mathcal{L} := \{[L_i, f_i] \mid i = 1 \ldots r\}$ and $\mathcal{R} := \{[R_j, g_j] \mid j = 1 \ldots s\}$ two lists of regular systems.

    Output Regular systems $\mathcal{S} := \{[T_i, h_i] \mid i = 1 \ldots e\}$ such that

$$\left(\bigcup_{i=1}^{r} Z(L_i, f_i) \right) \setminus \left(\bigcup_{j=1}^{s} Z(R_j, g_j) \right) = \bigcup_{i=1}^{e} Z(T_i, h_i),$$

with $\operatorname{rank}(\mathcal{S}) \leqslant_r \operatorname{rank}(\mathcal{L})$.

To prove the termination and correctness of above two algorithms, we present a series of technical lemmas.

Lemma 9.5.1. *Let p and h be polynomials and T a regular chain. Assume that $p \notin \operatorname{Sat}(T)$. Then there exists an operation* Intersect(p, T, h) *returning a set of regular chains $\{T_1, \ldots, T_e\}$ such that*

(i) *h is regular w.r.t $\operatorname{Sat}(T_i)$ for all i;*

(ii) *$\operatorname{rank}(T_i) <_r \operatorname{rank}(T)$;*

(iii) *$Z(p, T, h) \subseteq \cup_{i=1}^{e} Z(T_i, h) \subseteq (V(p) \cap \overline{W(T)}) \setminus V(h)$;*

(iv) *Moreover, if the product of initials h_T of T divides h then*

$$Z(p, T, h) = \bigcup_{i=1}^{e} Z(T_i, h).$$

PROOF. Let

$$\mathcal{S} = \mathsf{Triangularize}(p, T),$$
$$\mathcal{R} = \bigcup_{C \in \mathcal{S}} \mathsf{Regularize}(h, C).$$

Algorithm 47 Difference of Two Regular Systems

Difference($[T, h], [T', h']$) ==

1: **if** $\mathrm{Sat}(T) = \mathrm{Sat}(T')$ **then**
2:     **output** Intersect($h'h_{T'}, T, hh_T$)
3: **else**
4:     Let v be the largest variable s.t. $\mathrm{Sat}(T_{<v}) = \mathrm{Sat}(T'_{<v})$
5:     **if** $v \in \mathrm{mvar}(T')$ and $v \notin \mathrm{mvar}(T)$ **then**
6:       $p' \leftarrow T'_v$;   **output** $[T, hp']$
7:       **output** DifferenceLR(Intersect(p', T, hh_T), $[T', h']$)
8:     **else if** $v \notin \mathrm{mvar}(T')$ and $v \in \mathrm{mvar}(T)$ **then**
9:       $p \leftarrow T_v$;   **output** DifferenceLR($[T, h]$, Intersect($p, T', h'h_{T'}$))
10:     **else**
11:       $p \leftarrow T_v$; $\mathcal{G} \leftarrow$ GCD($T_v, T'_v, T_{<v}$)
12:       **if** $|\mathcal{G}| = 1$ **then**
13:         Let $(g, C) \in \mathcal{G}$
14:         **if** $g \in \mathbb{K}$ **then**
15:           **output** $[T, h]$
16:         **else if** $\mathrm{mvar}(g) < v$ **then**
17:           **output** $[T, gh]$
18:           **output** DifferenceLR(Intersect(g, T, hh_T), $[T', h']$)
19:         **else if** $\mathrm{mvar}(g) = v$ **then**
20:           **if** $\mathrm{mdeg}(g) = \mathrm{mdeg}(p)$ **then**
21:             $D'_p \leftarrow T'_{<v} \cup \{p\} \cup T'_{>v}$;   **output** Difference($[T, h], [D'_p, h'h_{T'}]$);
22:           **else if** $\mathrm{mdeg}(g) < \mathrm{mdeg}(p)$ **then**
23:             $q \leftarrow \mathrm{pquo}(p, g, C)$; $D_g \leftarrow C \cup \{g\} \cup T_{>v}$; $D_q \leftarrow C \cup \{q\} \cup T_{>v}$
24:             **output** Difference($[D_g, hh_T], [T', h']$)
25:             **output** Difference($[D_q, hh_T], [T', h']$)
26:             **output** DifferenceLR(Intersect(h_g, T, hh_T), $[T', h']$)
27:          **end if**
28:         **end if**
29:       **else if** $|\mathcal{G}| \geq 2$ **then**
30:         **for** $(g, C) \in \mathcal{G}$ **do**
31:           **if** $|C| > |T_{<v}|$ **then**
32:             **for** $E \in$ Extend($C, T_{\geqslant v}$) **do**
33:               **for** $D \in$ Regularize(hh_T, E) **do**
34:                 **if** $hh_T \notin \mathrm{Sat}(D)$ **then output** Difference($[D, hh_T], [T', h']$)
35:               **end for**
36:             **end for**
37:             **else**
38:               **output** Difference($[C \cup T_{\geqslant v}, hh_T], [T', h']$)
39:             **end if**
40:           **end for**
41:         **end if**
42:       **end if**
43: **end if**

Algorithm 48 Difference of a List of Regular Systems

DifferenceLR$(L, R) ==$

  1: **if** $L = \emptyset$ **then**
  2:     **output** $\emptyset$
  3: **else if** $R = \emptyset$ **then**
  4:     **output** L
  5: **else if** $|R| = 1$ **then**
  6:     Let $[T', h'] \in R$
  7:     **for** $[T, h] \in L$ **do**
  8:         **output** Difference$([T, h], [T', h'])$
  9:     **end for**
10: **else**
11:     **while** $R \neq \emptyset$ **do**
12:         Let $[T', h'] \in R$, $R \leftarrow R \setminus \{ [T', h'] \}$
13:         $S \leftarrow \emptyset$
14:         **for** $[T, h] \in L$ **do**
15:            $S \leftarrow S \cup$ Difference$([T, h], [T', h'])$
16:         **end for**
17:         $L \leftarrow S$
18:     **end while**
19:     **output** L
20: **end if**

We then have

$$V(p) \cap W(T) \ \subseteq \ \bigcup_{R \in \mathcal{R}} \ \subseteq \ V(p) \cap \overline{W(T)}.$$

This implies

$$Z(p, T, h) \subseteq \bigcup_{R \in \mathcal{R}, \ h \notin \mathrm{Sat}(R)} Z(R, h) \subseteq (V(p) \cap \overline{W(T)}) \setminus V(h).$$

Rename the regular chains $\{R \mid R \in \mathcal{R}, \ h \notin \mathrm{Sat}(R)\}$ as $\{T_1, \ldots, T_e\}$. By the specification of Regularize we immediately conclude that $(i), (iii)$ hold. Since $p \notin \mathrm{Sat}(T)$, by the specification of Triangularize, (ii) holds. By Lemma 9.3.3, (iv) holds. $\qquad\square$

Lemma 9.5.2. *Let $[T, h]$ and $[T', h']$ be two regular systems. If $\mathrm{Sat}(T) = \mathrm{Sat}(T')$, then $h'h_{T'}$ is regular w.r.t $\mathrm{Sat}(T)$ and*

$$Z(T, h) \setminus Z(T', h') = Z(h'h_{T'}, T, hh_T).$$

    PROOF.   Since $\mathrm{Sat}(T) = \mathrm{Sat}(T')$ and $h'h_{T'}$ is regular w.r.t $\mathrm{Sat}(T')$, $h'h_{T'}$ is

regular w.r.t $\mathrm{Sat}(T)$. By Lemma 9.3.3 and Lemma 9.3.4, we have

$$
\begin{aligned}
Z(T, hh'h_{T'}) &= W(T) \setminus V(hh'h_{T'}) \\
&= \overline{W(T)} \setminus V(hh'h_T h_{T'}) \\
&= \overline{W(T')} \setminus V(hh'h_T h_{T'}) \\
&= W(T') \setminus V(hh'h_T) \\
&= Z(T', hh'h_T).
\end{aligned}
$$

Then, we can decompose $Z(T, h)$ into the disjoint union

$$
Z(T, h) = Z(T, hh'h_{T'}) \bigsqcup Z(h'h_{T'}, T, hh_T).
$$

Similarly, we have:

$$
Z(T', h') = Z(T', hh'h_T) \bigsqcup Z(hh_T, T', h'h_{T'}).
$$

The conclusion follows from the fact that

$$
Z(T, hh'h_{T'}) \setminus Z(T', hh'h_T) = \emptyset \quad \text{and} \quad Z(h'h_{T'}, T, hh_T) \cap Z(T', h') = \emptyset.
$$

$\square$

Lemma 9.5.3. *Assume that* $\mathrm{Sat}(T_{<v}) = \mathrm{Sat}(T'_{<v})$. *We have*

(i) *If* $p' := T'_v$ *is defined but not* T_v, *then* p' *is regular w.r.t* $\mathrm{Sat}(T)$ *and*

$$
Z(T, h) \setminus Z(T', h') = Z(T, hp') \bigsqcup \left(Z(p', T, hh_T) \setminus Z(T', h') \right).
$$

(ii) *If* $p := T_v$ *is defined but not* T'_v, *then* p *is regular w.r.t* $\mathrm{Sat}(T')$ *and*

$$
Z(T, h) \setminus Z(T', h') = Z(T, h) \setminus Z(p, T', h'h_{T'}).
$$

PROOF. (i) As $\mathrm{init}(p')$ is regular w.r.t $\mathrm{Sat}(T'_{<v})$, it is also regular w.r.t $\mathrm{Sat}(T_{<v})$. Since T_v is not defined, we know $v \notin \mathrm{mvar}(T)$. Therefore, p' is also regular w.r.t $\mathrm{Sat}(T)$. On the other hand, we have a disjoint decomposition

$$
Z(T, h) = Z(T, hp') \bigsqcup Z(p', T, hh_T).
$$

$\square$

By the definition of p', $Z(T', h') \subseteq V(p')$ which implies

$$Z(T, hp') \bigcap Z(T', h') = \emptyset.$$

The conclusion follows.

(ii) Similarly, we know p is regular w.r.t $\text{Sat}(T')$. By the disjoint decomposition

$$Z(T', h') = Z(T', h'p) \bigsqcup Z(p, T', h'h_{T'}),$$

and $Z(T, h) \cap Z(T', h'p) = \emptyset$, we have

$$Z(T, h) \setminus Z(T', h') = Z(T, h) \setminus Z(p, T', h'h_{T'}),$$

from which the conclusion follows.

Lemma 9.5.4. *Assume that* $\text{Sat}(T_{<v}) = \text{Sat}(T'_{<v})$ *but* $\text{Sat}(T_{\leqslant v}) \neq \text{Sat}(T'_{\leqslant v})$ *and that* v *is algebraic w.r.t both* T *and* T'. *Define*

$$\mathcal{G} = \mathsf{GCD}(T_v, T'_v, T_{<v});$$
$$\mathcal{E} = \bigcup_{(g,C)\in\mathcal{G},\ |C|>|T_{<v}|} \mathsf{Extend}(C, T_{\geqslant v});$$
$$\mathcal{R} = \bigcup_{E\in\mathcal{E}} \mathsf{Regularize}(hh_T, E).$$

Then we have

(i)

$$Z(T, h)$$

$$= \left(\bigcup_{R\in\mathcal{R},\ hh_T\notin\text{Sat}(R)} Z(R, hh_T) \right) \bigcup \left(\bigcup_{(g,C)\in\mathcal{G},\ |C|=|T_{<v}|} Z(C \cup T_{\geqslant v}, hh_T) \right).$$

(ii) $\text{rank}(R) <_r \text{rank}(T)$, *for all* $R \in \mathcal{R}$.

(iii) *Assume that* $|C| = |T_{<v}|$. *Then*

(iii.a) $C \cup T_{\geqslant v}$ *is a regular chain and* hh_T *is regular w.r.t it.*

(iii.b) *If* $|\mathcal{G}| > 1$, *then* $\text{rank}(C \cup T_{\geqslant v}) <_r \text{rank}(T)$.

PROOF. By the specification of **GCD** we have

$$W(T_{<v}) \subseteq \bigcup_{(g,C)\in\mathcal{G}} W(C) \subseteq \overline{W(T_{<v})}.$$

That is,

$$W(T_{<v}) \xrightarrow{D} (W(C), (g,C) \in \mathcal{G}).$$

From the specification of Extend we have: for each $(g,C) \in \mathcal{G}$ such that $|C| > |T_{<v}|$,

$$W(C \cup T_{\geqslant v}) \xrightarrow{D} (W(E), E \in \mathsf{Extend}(C \cup T_{\geqslant v})).$$

From the specification of Regularize, we have for all $(g,C) \in \mathcal{G}$ such that $|C| > |T_{<v}|$ and all $E \in \mathsf{Extend}(C \cup T_{\geqslant v})$,

$$W(E) \xrightarrow{D} (W(R), \ R \in \mathsf{Regularize}(hh_T, E)).$$

Therefore, by applying the Lifting Theorem [108] we have:

$$\begin{aligned}
W(T) &= W(T_{<v} \cup T_{\geqslant v}) \\
&\subseteq \left(\bigcup_{R\in\mathcal{R}} W(R)\right) \bigcup \left(\bigcup_{(g,C)\in\mathcal{G},\ |C|=|T_{<v}|} W(C \cup T_{\geqslant v})\right) \\
&\subseteq \overline{W(T_{<v} \cup T_{\geqslant v})} \\
&= \overline{W(T)},
\end{aligned}$$

which implies,

$$\begin{aligned}
Z(T, h) &= Z(T, hh_T) \\
&\subseteq \left(\bigcup_{R\in\mathcal{R},\ hh_T\notin\mathrm{Sat}(R)} Z(R, hh_T)\right) \bigcup \left(\bigcup_{(g,C)\in\mathcal{G},\ |C|=|T_{<v}|} Z(C \cup T_{\geqslant v}, hh_T)\right) \\
&\subseteq \overline{W(T)} \setminus V(hh_T) = Z(T, h).
\end{aligned}$$

So (i) holds. When $|\mathcal{G}| > 1$, by Notation 9.3.9, (ii) and $(iii.b)$ hold.

If $|C| = |T_{<v}|$, by Proposition 5 of [108], we conclude that $(iii.a)$ holds. $\qquad\square$

Lemma 9.5.5. *Assume that* $\mathrm{Sat}(T_{<v}) = \mathrm{Sat}(T'_{<v})$ *but* $\mathrm{Sat}(T_{\leqslant v}) \neq \mathrm{Sat}(T'_{\leqslant v})$ *and that*

v is algebraic w.r.t both T and T'. Define $p = T_v$, $p' = T'_v$ and

$$\mathcal{G} = \mathsf{GCD}(p, p', T_{<v}).$$

If $|\mathcal{G}| = 1$, let $\mathcal{G} = \{(g, C)\}$. Then the following properties hold

(i) $C = T_{<v}$.

(ii) *If $g \in \mathbb{K}$, then*
$$Z(T, h) \setminus Z(T', h') = Z(T, h).$$

(iii) *If $g \notin \mathbb{K}$ and $\mathrm{mvar}(g) < v$, then g is regular w.r.t $\mathrm{Sat}(T)$ and*

$$
\begin{aligned}
&Z(T, h) \setminus Z(T', h') \\
&= Z(T, gh) \bigsqcup \left(Z(g, T, hh_T) \setminus Z(T', h') \right).
\end{aligned}
$$

(iv) *Assume that $\mathrm{mvar}(g) = v$.*

    (iv.a) *If $\mathrm{mdeg}(g) = \mathrm{mdeg}(p)$, defining*

$$
\begin{aligned}
q' &= \mathrm{pquo}(p', p, T'_{<v}) \\
D'_p &= T'_{<v} \cup \{p\} \cup T'_{>v} \\
D'_{q'} &= T'_{<v} \cup \{q'\} \cup T'_{>v},
\end{aligned}
$$

    then we have

$$Z(T, h) \setminus Z(T', h') = Z(T, h) \setminus Z(D'_p, h'h_{T'}),$$

    $\mathrm{rank}(D'_p) < \mathrm{rank}(T')$ *and $h'h_{T'}$ is regular w.r.t $\mathrm{Sat}(D'_p)$.*

    (iv.b) *If $\mathrm{mdeg}(g) < \mathrm{mdeg}(p)$, defining*

$$
\begin{aligned}
q &= \mathrm{pquo}(p, g, T_{<v}) \\
D_g &= T_{<v} \cup \{g\} \cup T_{>v} \\
D_q &= T_{<v} \cup \{q\} \cup T_{>v},
\end{aligned}
$$

    then we have: D_g and D_q are regular chains such that $\mathrm{rank}(D_g) < \mathrm{rank}(T)$,

$$\mathrm{rank}(D_q) < \mathrm{rank}(T), \; hh_T \text{ is regular w.r.t } \mathrm{Sat}(D_g) \text{ and } \mathrm{Sat}(D_q), \text{ and}$$

$$Z(T, h) = Z(D_g, hh_T) \bigcup Z(D_q, hh_T) \bigcup Z(h_g, T, hh_T).$$

PROOF. Since $|\mathcal{G}| = 1$, by the specification of the operation GCD and Notation 1, (i) holds. Therefore we have

$$\mathrm{Sat}(C) = \mathrm{Sat}(T_{<v}) = \mathrm{Sat}(T'_{<v}) \tag{9.1}$$

There exist polynomials A and B such that

$$g \equiv Ap + Bp' \mod \mathrm{Sat}(C). \tag{9.2}$$

From (9.2), we have

$$V(\mathrm{Sat}(C)) \subseteq V(g - Ap - Bp') \tag{9.3}$$

Therefore, we deduce

$$
\begin{aligned}
& W(T) \bigcap W(T') \\
&= W(T_{<v} \cup p \cup T_{\geqslant v}) \bigcap W(T'_{<v} \cup p' \cup T'_{\geqslant v}) \\
&\subseteq (W(T_{<v}) \cap V(p)) \bigcap (W(T'_{<v}) \cap V(p')) \\
&\subseteq V(\mathrm{Sat}(T_{<v})) \bigcap V(p) \bigcap V(p') && \text{by (9.1)} \\
&\subseteq V(g - Ap - Bp') \bigcap V(p) \bigcap V(p') && \text{by (9.3)} \\
&\subseteq V(g).
\end{aligned}
$$

that is

$$W(T) \bigcap W(T') \subseteq V(g). \tag{9.4}$$

Now we prove (ii). When $g \in \mathbb{K}$, $g \neq 0$, from (9.4) we deduce

$$W(T) \bigcap W(T') = \emptyset. \tag{9.5}$$

Thus we have

$$Z(T, h) \setminus Z(T', h')$$
$$= (W(T) \setminus V(h)) \setminus (W(T') \setminus V(h'))$$
$$= (W(T) \setminus V(h)) \qquad\qquad \text{by (9.5)}$$
$$= Z(T, h).$$

Now we prove (iii). Since $C = T_{<v}$ and $\mathrm{mvar}(g)$ is smaller than or equal to v, by the specification of GCD, g is regular w.r.t $\mathrm{Sat}(T)$. We have following decompositions

$$Z(T, h) = Z(T, gh) \bigsqcup Z(g, T, hh_T),$$
$$Z(T', h') = Z(T', gh') \bigsqcup Z(g, T', h'h_{T'}).$$

On the other hand,

$$Z(T, gh) \bigcap Z(T', gh')$$
$$= (W(T) \cap V(gh)^c) \bigcap (W(T') \cap V(gh')^c)$$
$$\subseteq (W(T) \cap V(g)^c) \bigcap (W(T') \cap V(g)^c)$$
$$= (W(T) \cap W(T')) \bigcap V(g)^c$$
$$= \emptyset \qquad \text{by (9.4).}$$

Therefore,

$$Z(T, h) \setminus Z(T', h')$$
$$= (Z(T, gh) \setminus Z(T', gh')) \bigsqcup (Z(g, T, hh_T) \setminus Z(T', h'))$$
$$= Z(T, gh) \bigsqcup (Z(g, T, hh_T) \setminus Z(T', h')).$$

Now we prove $(iv.a)$. First, both h' and h'_T are regular w.r.t $\mathrm{Sat}(C) = \mathrm{Sat}(T_{<v}) = \mathrm{Sat}(T'_{<v})$. From the construction of D'_p, we have $h'h_{T'}$ is regular w.r.t $\mathrm{Sat}(D'_p)$.

Assume that $\mathrm{mvar}(g) = v$ and $\mathrm{mdeg}(g) = \mathrm{mdeg}(p)$. We note that $\mathrm{mdeg}(p') > \mathrm{mdeg}(p)$ holds. Otherwise we would have $\mathrm{mdeg}(g) = \mathrm{mdeg}(p) = \mathrm{mdeg}(p')$ which implies:

$$p \in \mathrm{Sat}(T'_{\geqslant v}) \text{ and } p' \in \mathrm{Sat}(T_{\geqslant v}). \qquad (9.6)$$

Thus

$$\begin{aligned}
\operatorname{Sat}(T_{\leqslant v}) = \langle T_{\leqslant v}\rangle : h_{T_{\leqslant v}}^{\infty} &= \langle T_{<v} \cup p\rangle : h_{T_{\leqslant v}}^{\infty} \\
&\subseteq \operatorname{Sat}(T'_{\leqslant v}) : h_{T_{\leqslant v}}^{\infty} \qquad\qquad \text{by (9.6)} \\
&= \operatorname{Sat}(T'_{\leqslant v}),
\end{aligned}$$

that is $\operatorname{Sat}(T_{\leqslant v}) \subseteq \operatorname{Sat}(T'_{\leqslant v})$. Similarly, $\operatorname{Sat}(T'_{\leqslant v}) \subseteq \operatorname{Sat}(T_{\leqslant v})$ holds. So we have $\operatorname{Sat}(T'_{\leqslant v}) = \operatorname{Sat}(T_{\leqslant v})$, a contradiction.

Hence, $\operatorname{mvar}(q') = v$.

By Lemma 6 [108], we know that D'_p and $D'_{q'}$ are regular chains. Then with Theorem 7 [108] and Lifting Theorem [108], we know

$$\begin{aligned}
Z(T', h') \subseteq Z(D'_p, h') &\bigcup Z(D'_{q'}, h') \bigcup Z(h_p, T', h') \\
&\subseteq \overline{W(T')} \setminus V(h').
\end{aligned}$$

By Lemma 9.3.3, we have

$$Z(T', h') = Z(D'_p, h'h_{T'}) \bigcup Z(D'_{q'}, h'h_{T'}) \bigcup Z(h_p, T', h'h_{T'}).$$

Since

$$\begin{aligned}
Z(D'_{q'}, h'h_{T'}) &= Z(D'_{q'}, h_p h'h_{T'}) \bigcup Z(h_p, D'_{q'}, h'h'_T) \\
&= Z(D'_{q'}, ph_p h'h_{T'}) \bigcup Z(p, D'_{q'}, h_p h'h'_T) \bigcup Z(h_p, D'_{q'}, h'h'_T)
\end{aligned}$$

and

$$\begin{aligned}
Z(p, D'_{q'}, h_p h'h'_T) &\subseteq Z(D'_p, h'h_{T'}) \\
Z(h_p, D'_{q'}, h'h'_T) &\subseteq Z(h_p, T', h'h_{T'}),
\end{aligned}$$

we deduce

$$Z(T', h') = Z(D'_p, h'h_{T'}) \bigsqcup Z(D'_{q'}, ph'h_{T'}) \bigsqcup Z(h_p, T', h'h_{T'}).$$

Now observe that

$$Z(T,h) \bigcap Z(D'_{q'}, ph'h_{T'}) = \emptyset, \text{ and}$$
$$Z(T,h) \bigcap Z(h_p, T', h'h_{T'}) = \emptyset.$$

We obtain

$$Z(T,h) \setminus Z(T',h') = Z(T,h) \setminus Z(D'_p, h'h_{T'}).$$

Finally we prove $(iv.b)$. We assume that $\mathrm{mvar}(g) = v$ and $\mathrm{mdeg}(g) < \mathrm{mdeg}(p)$; this implies $\mathrm{mvar}(q) = v$. Applying Lemma 6 in [108] we know that D_g and D_q are regular chains and satisfy the desired rank condition. Then by Theorem 7 [108] and Lifting Theorem [108] we have

$$Z(T,h) = Z(D_g, hh_T) \bigcup Z(D_q, hh_T) \bigcup Z(h_g, T, hh_T).$$

This completes the whole proof. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Definition 9.5.6. Given two pairs of ranks $(\mathrm{rank}(T_1), \mathrm{rank}(T'_1))$ and $(\mathrm{rank}(T_2), \mathrm{rank}(T'_2))$, where T_1, T_2, T'_1, T'_2 are triangular sets. We define the product order $<_p$ of Ritt order $<_r$ on them as follows

$$(\mathrm{rank}(T_2), \mathrm{rank}(T'_2)) <_p (\mathrm{rank}(T_1), \mathrm{rank}(T'_1))$$
$$\Longleftrightarrow \begin{cases} \mathrm{rank}(T_2) <_r \mathrm{rank}(T_1) & \text{or} \\ \mathrm{rank}(T_2) = \mathrm{rank}(T_1), & \mathrm{rank}(T'_2) <_r \mathrm{rank}(T'_1). \end{cases}$$

In the following theorems, we prove the termination and correctness separately. Along with the proof of Theorem 9.5.7, we show the rank conditions are satisfied which is part of the correctness. The remained part, say zero set decomposition, will be proved in Theorem 9.5.8.

Theorem 9.5.7. *Algorithms* Difference *and* DifferenceLR *terminate and satisfy the rank conditions in their specifications.*

PROOF. The following two statements need to be proved

(i) Difference terminates with $\mathrm{rank}(\mathsf{Difference}([T,h],[T',h'])) \leqslant_r \mathrm{rank}([T,h])$,

(ii) DifferenceLR terminates with $\mathrm{rank}(\mathsf{DifferenceLR}(\mathcal{L},\mathcal{R})) \leqslant_r \mathrm{rank}(\mathcal{L})$.

We prove them by induction on the product order $<_p$.

(1) Base case: there are no recursive calls to Difference or DifferenceLR. The termination of both algorithms is clear. By line $2, 18$ of the algorithm Difference, $\mathrm{rank}(\mathsf{Difference}([T, h], [T', h'])) \leqslant_r \mathrm{rank}([T, h])$. By line $2, 4$ of the algorithm DifferenceLR, $\mathrm{rank}(\mathsf{DifferenceLR}(\mathcal{L}, \mathcal{R})) \leqslant_r \mathrm{rank}(\mathcal{L})$.

(2) Induction hypothesis: assume that both (i) and (ii) hold with inputs whose ranks are smaller than the rank of $([T, h], [T', h'])$ w.r.t. $<_p$.

(3) By (1), if no recursive calls occur in one branch, then (i) and (ii) already hold. When recursive calls occur, by line $8, 11, 21, 25, 30, 31, 32, 41, 46$ and Lemma 9.5.1, 9.5.3, 9.5.4, 9.5.5, we know the inputs of recursive calls to both Difference and DifferenceLR have smaller ranks than $\mathrm{rank}(([T, h], [T', h']))$ w.r.t $<_p$. By induction hypothesis, (i) holds. Finally, by line $8, 15$ of algorithm DifferenceLR and (i), (ii) holds.

□

Theorem 9.5.8. *Both* Difference *and* DifferenceLR *satisfy their specifications.*

PROOF. By Theorem 9.5.7, Difference and DifferenceLR terminate and satisfy their rank conditions. So it suffices to prove the correctness of Difference and DifferenceLR, that is

(i) $Z(T, h) \setminus Z(T', h') = Z(\mathsf{Difference}([T, h], [T', h']))$,

(ii) $Z(\mathcal{L}) \setminus Z(\mathcal{R}) = Z(\mathsf{DifferenceLR}(\mathcal{L}, \mathcal{R}))$.

We prove them by induction on the product order $<_p$.

(1) Base case: no recursive calls to Difference and DifferenceLR occur. First, by line $2, 15$ of the algorithm Difference and Lemma 9.5.1, 9.5.2, 9.5.5, (i) holds. Second, by line $2, 4$ of the algorithm DifferenceLR, (ii) holds.

(2) Induction hypothesis: assume that both (i) and (ii) hold with inputs whose ranks are smaller than the rank of $([T, h], [T', h'])$ w.r.t. $<_p$.

(3) By (1), if no recursive calls occur, (i) and (ii) already hold. When there are recursive calls, we first show (i) holds. From the proof of Theorem 9.5.7, in Difference, the inputs of recursive calls to Difference and DifferenceLR will have smaller ranks w.r.t. the product order $<_p$. Therefore, by (2), line 6, 7, 9, 17, 18, 21, 24, 25, 26, 34, 38 and Lemma 9.5.1, 9.5.3, 9.5.4, 9.5.5, (i) holds.

Finally, by (i) and line $5 - 18$ of algorithm DifferenceLR, (ii) holds.

□

9.6 Verification of Triangular Decompositions

In this section, we describe how to verify the output from a triangular decomposition. Verification in Kalkbrener's sense is still unknown whether we can circumvent Gröbner basis computations. However, in Lazard's sense, we will present both Gröbner basis and triangular decomposition methods.

9.6.1 Verification with Gröbner bases

The following two lemmas state the Gröbner basis methods to verify whether two basic constructible sets are equal or not.

Lemma 9.6.1. *Let $\{F, f\}$ and $\{G_0, g_0\}$ be two polynomial systems. The following statements are equivalent*

1. $D(F, f) \setminus D(G_0, g_0) \subseteq \bigcup_{i=1}^{r} D(G_i, g_i)$.

2. *For every $\{i_1, \ldots, i_s\} \subseteq \{0, \ldots, r\}$, $0 \leq s \leq r$,*

$$\sqrt{\langle F \cup \{g_{i_1}, \ldots, g_{i_s}\} \rangle} \supseteq \prod_{k \in \{0, \ldots, r\} \setminus \{i_1, \ldots, i_s\}} \langle f \rangle \langle G_k \rangle. \qquad (9.7)$$

PROOF. (1) is equivalent to $D(F, f) \subseteq \bigcup_{i=0}^{r} D(G_i, g_i)$.

$$D(F, f) \bigcap \left(\bigcap_{i=0}^{e} D(G_i, g_i)^c \right) = \emptyset.$$

Using the distributive property, we deduce that (1) is equivalent to

$$\left(D(F, f) \bigcap V(g_{i_1}, \ldots, g_{i_s}) \right) \bigcap \left(\bigcap_{k \in \{0, \ldots, r\} \setminus \{i_1, \ldots, i_s\}} V(G_k)^c \right) = \emptyset,$$

for all subsets $\{i_1, \ldots, i_s\}$ of $\{0, \ldots, r\}$. The proof easily follows. □

Lemma 9.6.2. *Let $\{F, f\}$ and $\{G, g\}$ be two polynomial systems. The following statements are equivalent*

1. $D(F, f) \setminus D(G, g) \supseteq \bigcup_{i=1}^{r} D(H_i, h_i)$.

2. *For all $1 \leq i \leq r$, we have*

$$h_i g \in \sqrt{\langle H_i \cup G \rangle}, h_i \in \sqrt{\langle H_i, f \rangle}, \text{ and } \langle h_i \rangle \langle F \rangle \subset \sqrt{\langle H_i \rangle}. \qquad (9.8)$$

PROOF. (1) holds if and only if for each $1 \leq i \leq r$ we have

$$\begin{cases} D(H_i, h_i) \bigcap D(F, f)^c &= \emptyset, \\ D(H_i, h_i) \bigcap D(G, g) &= \emptyset, \end{cases}$$

which holds if and only if

$$\begin{cases} V(H_i) \bigcap V(h_i)^c \bigcap V(F)^c &= \emptyset, \\ V(H_i) \bigcap V(h_i)^c \bigcap V(f) &= \emptyset, \\ V(H_i) \bigcap V(h_i)^c \bigcap V(G) \bigcap V(g)^c &= \emptyset. \end{cases}$$

The proof easily follows. $\square$

9.6.2 Verification with Triangular Decompositions

Given two Lazard's triangular decompositions $\{T_i \mid i = 1 \ldots e\}$ and $\{S_j \mid j = 1 \ldots f\}$. Checking $\cup_{i=1}^{e} W(T_i) = \cup_{j=1}^{f} W(S_j)$ amounts to checking both

$$\left(\bigcup_{i=1}^{e} W(T_i) \right) \setminus \left(\bigcup_{j=1}^{f} W(S_j) \right) \text{ and } \left(\bigcup_{j=1}^{f} W(S_j) \right) \setminus \left(\bigcup_{i=1}^{e} W(T_i) \right)$$

being empty, after computing the regular system representations of above two constructible sets. According to Lemma 9.3.8, we solve the verification problem with the algorithm DifferenceLR in Lazard's sense.

9.7 Experimentation

We have implemented a verifier, named *Diff-verifier*, according to the DifferenceLR algorithm proposed in Section 9.5, and it has been implemented in MAPLE 11 based on the `RegularChains` library. To verify the effectiveness of our Diff-verifier, we have also implemented another verifier, named *GB-verifier*, applying Lemma 9 and 10, on top of the *PolynomialIdeals* package in MAPLE 11.

We use these two verifiers to examine four polynomial system solvers herein. They are the *Triangularize* function in the *RegularChains* library [91], the Triade server in ALDOR, written with the *BasicMath library* [132], the *RegSer* function and the *SimSer* function in *Epsilon* [136] implemented in MAPLE. The first two solvers solve a polynomial system into regular chains by means of the Triade algorithm [108]. They can work in both *Lazard's* sense and *Kalkbrener's* sense. In this work, we use the

options for solving in *Lazard's* sense. The *RegSer* function decomposes a polynomial system into regular systems in the sense of [135], and the *SimSer* function decomposes a polynomial system into simple systems, as in [139].

The problems used in this benchmark are chosen from [104, 128, 136]. In Table 9.1, for each system, we give the *dimension* sequence of the triangular decomposition computed in *Kalkbrener's* sense by the Triade algorithm. The number of variables is denoted by n, and d is the maximum degree of a monomial in the input. We also give the number of components in the solution set for each of the methods we are studying.

Table 9.2 gives the timing of each problem solved by the four methods. In this study, due to the current availability of *Epsilon*, the timings obtained by the *RegSer* and the *SimSer* commands are performed in MAPLE 8 on Intel Pentium 4 machines (1.60GHz CPU, 513MB memory and Red Hat Linux 3.2.2-5). All the other timings are run on Intel Pentium 4 (3.20GHz CPU, 2.0GB total memory, and Red Hat 4.0.0-9), and the Maple version used is 11. The Triade server is a stand-alone executable program compiled from a program in ALDOR.

Table 9.3 summarizes the timings of GB-verifier for verifying the solutions of the four methods. Table 9.3 illustrates the timings of Diff-verifier for checking the solutions by MAPLE *Triangularize* against ALDOR Triade server, MAPLE *Triangularize* against Epsilon *RegSer*, and Epsilon *RegSer* against Epsilon *SimSer*. For the case where there is a time, the verifying result is also true. The $'-'$ denotes the case where the test stalls by either reaching the time limit of 43200 seconds or causing a memory failure.

This experimentation results illustrate that verifying a polynomial solver is a truly difficult task. The GB-verifier is very costly in terms of CPU time and memory. It only succeeds for some easy examples. Assuming that the GB-verifier is reliable, for the examples it succeeds, the Diff-verifier agrees with its results by pair-wise checking, while it takes much less time. This shows the efficiency of our Diff-verifier. Further more, the tests also show that the Diff-verifier can verify more difficult problems by pair-wise checking. The tests indicate that all of the four methods are solving tools with a high probability of correctness, since the checking results would not agree with each other otherwise.

					Number of Components			
					MAPLE	ALDOR	Epsilon	Epsilon
Sys	Name	n	d	Dimension	Triangularize	Triade server	RegSer	SimSer
1	Montes S1	4	2	[2,2,1]	3	3	3	3
2	Montes S2	4	3	[0]	1	1	1	1
3	Montes S3	3	3	[1,1]	2	2	2	3
4	Montes S4	4	2	[0]	1	1	1	1
5	Montes S6	4	3	[2,2,2]	3	3	3	3
6	Montes S7	4	3	[1]	2	2	3	6
7	Montes S8	4	12	[2,1]	2	2	6	6
8	Alonso	7	4	[3]	3	3	3	4
9	Raksanyi	8	3	[4]	4	4	4	10
10	YangBaxter Rosso	6	3	[4,3,3,1,1,1,1]	7	7	4	13
11	l-3	4	3	[0,0,0,0,0,0,0, 0,0,0,0,0,0]	25	13	8	8
12	Caprasse	4	4	[0,0,0,0,0]	15	5	4	4
13	Reif	16	3	[]	0	0	0	0
14	Buchberger WuWang	5	3	[2]	3	3	3	4
15	DonatiTraverso	4	31	[1]	6	3	3	3
16	Wu-Wang.2	13	3	[1,1,1,1,1]	5	5	5	5
17	Hairer-2-BGK	13	4	[2]	4	4	5	6
18	Montes S5	8	3	[4]	4	4	4	10
19	Bronstein	4	3	[1]	4	2	4	9
20	Butcher	8	4	[3,3,3,2,2,0]	7	6	6	6
21	genLinSyst-2-2	8	2	[6]	11	11	11	11
22	genLinSyst-3-2	11	2	[8]	17	18	18	18
23	Gerdt	7	4	[3,2,2,2,1,1]	7	6	10	10
24	Wang93	5	3	[1]	5	4	6	7
25	Vermeer	5	5	[1]	5	4	12	14
26	Gonnet	5	2	[3,3,3]	3	3	9	9
27	Neural	4	3	[1,1]	4	3	–	–
28	Noonburg	4	3	[1,1]	4	3	–	–
29	KdV	1	0	[12,12,11, 11,11,11,11]	7	7	–	–
30	Montes S12	8	2	[4]	22	17	23	–
31	Pappus	12	2	[6,6,6,6,6, 6,6,6,6]	124	129	156	–

Table 9.1: Features of the Polynomial Systems for Verification

Sys	MAPLE Triangularize	ALDOR Triade server	Epsilon RegSer	Epsilon SimSer
1	0.104	0.164	0.01	0.03
2	0.039	0.204	0.03	0.02
3	0.069	0.06	0.019	0.111
4	0.510	0.072	0.049	0.03
5	0.052	0.096	0.03	0.03
6	0.150	0.06	0.09	5.14
7	0.376	0.072	0.2	1.229
8	0.204	0.065	0.109	0.16
9	0.460	0.066	0.141	0.481
10	1.252	0.108	0.069	0.21
11	5.965	0.587	1.53	2.91
12	2.426	0.167	1.209	2.32
13	123.823	1.886	1.979	2.36
14	0.2	0.101	0.049	0.109
15	2.641	0.08	0.439	0.7
16	105.835	1.429	5.49	6.14
17	23.453	0.688	1.76	1.679
18	0.484	0.078	0.13	0.471
19	0.482	0.071	0.24	1.000
20	9.325	0.442	1.689	2.091
21	0.557	0.096	0.13	0.21
22	1.985	0.173	0.431	0.411
23	4.733	0.499	3.5	4.1
24	7.814	5.353	2.18	30.24
25	26.533	0.580	4.339	60.65
26	3.983	0.354	2.18	2.48
27	15.879	1.567	–	–
28	15.696	1.642	–	–
29	9245.442	49.573	–	–
30	17.001	0.526	2.829	–
31	79.663	4.429	11.78	–

Table 9.2: Solving Timings in Seconds of the Four Methods

	GB-verifier timing(s)				Diff-verifier timing(s)		
	MAPLE Triangularize	ALDOR Triade server	Epsilon RegSer	Epsilon SimSer	M.T. vs	M.T. vs	E.R. vs
sys	(M.T.)	(A.T.)	(E.R.)	(E.S.)	A.T.	E.R.	E.S.
1	0.556	0.526	0.518	0.543	0.58	0.439	0.445
2	0.128	0.127	0.129	0.131	0.039	0.02	0.013
3	0.584	0.575	0.585	2.874	0.182	0.108	0.427
4	0.104	0.133	0.139	0.137	0.037	0.027	0.023
5	1.484	1.472	1.457	1.469	0.591	0.339	0.356
6	76.596	72.374	71.853	–	7.204	5.268	15.334
7	0.616	0.601	4.501	4.536	0.573	0.758	1.017
8	–	–	–	–	1.196	1.564	2.618
9	–	–	–	–	5.442	9.837	18.252
10	–	–	–	–	10.888	22.638	22.649
11	–	–	–	–	14.652	4.541	3.585
12	–	58.332	33.469	35.213	2.52	2.398	3.113
13	–	–	–	–	0	0	0
14	1.96	1.937	2.165	5.739	0.924	0.915	1.155
15	330.317	–	–	–	2.244	4.782	4.201
16	10466.587	–	–	–	4.34	4.408	3.207
17	–	–	–	–	6.348	6.109	15.719
18	–	–	–	–	5.32	10.485	17.897
19	1.544	0.717	5.046	–	7.838	7.986	43.506
20	–	–	–	–	13.04	10.218	9.978
21	–	–	–	–	10.872	15.098	11.048
22	–	–	–	–	61.147	48.865	32.184
23	–	–	–	–	11.144	15.981	16.222
24	–	–	–	–	1564.654	1918.968	870.962
25	–	–	–	–	2144.726	–	2182.401
26	–	–	–	–	3.839	6.041	9.550
27	11383.335	–	–	–	1088.563	–	–
28	–	–	–	–	1119.449	–	–
29	–	–	–	–	30.016	–	–
30	–	–	–	–	–	–	–
31	–	–	–	–	–	–	–

Table 9.3: Timings of GB-verifier and Diff-verifier

Chapter 10

Conclusions and Future Work

10.1 Conclusions

By fast algorithms, modular methods, parallel approaches and software engineering, we aim at improving the theoretical and practical efficiency for solving non-linear polynomial systems symbolically by way of triangular decompositions.

We have extended fast algorithms for polynomial addition, multiplication, GCD and quasi-inverse over field to direct products of fields presented by triangular sets. Our complexity analysis proves that they have quasi-linear time complexity (i.e. nearly-optimal). They are fundamental operations for polynomial arithmetic for computing triangular decompositions. If they are implemented efficiently, it will speedup significantly the computations for triangular decompositions.

The equiprojectable decomposition that we introduced here is canonical and has good computational properties. Our modular method based on Hensel lifting techniques is designed for solving systems with a finite number of solutions. The complexity is proved to be quasi-linear. The implementation and experimental results show its capability for solving difficult problems which are impossible for other solvers. This work also provides functionalities (as a byproduct) for linear algebra over non-integral domains and automatic case discussion. The next step is to extend this modular method to treat systems with infinite number of solutions.

We believe that our implementation experience in making a symbolic solver available to different communities of users can benefit other algorithms in this area. The tools for efficiently computing irredundant triangular decompositions and verifying the outputs of solvers are useful for both developers and users.

Symbolic methods are powerful tools in scientific computing. The implementation of these methods is still a highly difficult task. Using parallel and distributed comput-

ing to gain practical efficiency for solving polynomial systems is a promising direction. Our implementation for the component-level parallelization of the Triade algorithm demonstrates a certain degree of speedup for solving some well-known problems, however we have noticed the limited parallelism by nature in this coarse grained level. Our implementation also indicates that the parallel overhead of multi-processing and data communication cannot be neglected. One of our research plans is to achieve efficient multi-level parallelization of triangular decompositions and develop libraries for fast and parallel polynomial arithmetic. Some details of this plan are reported in the next section.

10.2 Towards Efficient Multi-level Parallelization of Triangular Decompositions

The field of computer algebra has reached the stage where algorithms for polynomial systems on high performance architectures are of great interest. This contrasts sharply with the state of affairs of only a few years ago, where the computational complexity of known algorithms made treatment of large problems infeasible. Our proposed research in the near future is to extend the approach of this thesis into three promising, strongly-related areas:

- to design a suitable high-level categorical framework for computer algebra that can be efficiently specialized to important parallel environments and, within this,

- to explore techniques for polynomial arithmetic on clusters of multi-core processors and determine a suitable set of primitives for multivariate polynomials, akin to BLAS [130] and LinBox [114] and parallel systems [45, 78] for linear algebra, to allow effective definitions on different parallel architectures, and to use this

- to develop algorithms with multiple levels of parallelization for multivariate systems.

We shall use the ALDOR programming language as the implementation vehicle, since it has high-level categorical support for generic algorithms in computer algebra, while providing the necessary low-level access for high performance computing. These features are some of what Fortress is attempting to build for Fortran.

In this thesis study, we have noticed that the heavy overhead from dynamic process management and data communication can be a bottleneck for an efficient parallel execution. In fact, this is an important challenge for parallel symbolic computations in general. It is also the case that our coarse-grained tasks are highly irregular and problem dependent. In many occasions a few heavy tasks dominate the computing time.

The first objective of our future work is to adapt a thread model for ALDOR, as has been done for SACLIB [84, 121] and KAAPI [74, 59]. Threads allow the right level of granularity for parallel symbolic computation, but the ALDOR runtime system must be altered to take advantage of threads where possible (e.g. in its garbage collector [33]), and must be revised in some places for thread safety. Other than concurrency control issues, we must ensure that parametric types, such as polynomial data types, are properly treated.

Since parallel applications in computer algebra are generally dynamic and irregular, we must provide scheduling mechanism for threads in ALDOR to achieve load balancing. We will apply the provably efficient "work-stealing" scheduling algorithm based on the "work first principle" [18, 56] to support automatic scheduling for threads in ALDOR, as it has been used in Cilk [89] and KAAPI. Of course, we shall also provide support for data locality and adaptivity on multiprogrammed environment.

The second objective is to use this platform to develop a library for parallel and fast polynomial arithmetic, and to use it to investigate multi-level parallel algorithms for triangular decompositions of polynomial systems. We plan to investigate combining coarse grained level for tasks to compute geometric of the solution sets, and medium/fine grained level for polynomial arithmetic such as multiplication, GCD/resultant, and factorization. Parallel polynomial arithmetic over fields for linear algebra and univariate polynomials are well-developed. We need to extend these methods to the multivariate case over more general domains with potential of automatic case discussion.

The software resulting from this work should help solve application problems usually out of reach for other symbolic solvers. The parallel framework and the library for multivariate polynomial arithmetic will benefit other algorithms in symbolic computations in adapting emerging architectures.

Bibliography

[1] W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases.* American Mathematical Society, 1994.

[2] I. A. Ajwa. *Parallel algorithms and implementations for the Gröbner Bases Algorithm and the Characteristic Set Method.* PhD thesis, Kent State University, Kent, Ohio, 1998.

[3] aldor.org. *The* Aldor *compiler web site.* University of Western Ontario, Canada, 2007. http://www.aldor.org.

[4] Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[5] E. A. Arnold. Modular algorithms for computing Gröbner bases. *J. Symb. Comp.*, 35(4):403–419, 2003.

[6] T. Ashby and A. K. M. O'Boyle. A modular iterative solver package in a categorical language. In *LNCS vol.47*, pages 123–132, 1993.

[7] G. Attardi and C. Traverso. Strategy-accurate parallel Buchberger algorithms. *Journal of Symbolic Computation*, 21(4):411–425, 1996.

[8] P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *J. Symb. Comp.*, 28(1-2):105–124, 1999.

[9] P. Aubry and M. Moreno Maza. Triangular sets for solving polynomial systems: A comparative implementation of four methods. *J. Symb. Comp.*, 28(1-2):125–154, 1999.

[10] P. Aubry and A. Valibouze. Using Galois ideals for computing relative resolvents. *J. Symb. Comp.*, 30(6):635–651, 2000.

[11] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor, *Proc. ISSAC'91*, pages 103–111. ACM, 1991.

[12] E. Becker, T. Mora, M. G. Marinari, and C. Traverso. The shape of the shape lemma. In *Proc. of ISSAC94*, pages 129–133, New York, NY, USA, 1994. ACM Press.

[13] T. Becker and V. Weispfenning. *Gröbner Bases: a computational approach to commutative algebra*, volume 141 of *Graduate Texts in Mathematics*. Springer Verlag, 1991.

[14] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46:1853–1859, 1967.

[15] E. R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24(7):713–735, 1970.

[16] D. J. Bernstein. Faster factorization into coprimes.   .
`http://cr.yp.to/papers.html$\#$dcba2`, 2004.

[17] D. J. Bernstein. Factoring into coprimes in essentially linear time. *J. Algorithms*, 54(1):1–30, 2005.

[18] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE FOCS94*, 1994.

[19] F. Boulier, L. Denis-Vidal, T. Henin, and F. Lemaire. Lépisme. In *International Conference on Poynomial System Solving*. University of Paris 6, France, 2004.

[20] F. Boulier and F. Lemaire. Computing canonical representatives of regular differential ideals. In *proceedings of ISSAC 2000*, pages 37–46, St Andrews, Scotland, 2000. ACM Press.

[21] F. Boulier, F. Lemaire, and M. Moreno Maza. PARDI ! Technical Report LIFL 2001–01, Université Lille I, LIFL, 2001.

[22] F. Boulier, F. Lemaire, and M. Moreno Maza. Well known theorems on triangular systems and the D5 principle. In *Proc. of* Transgressive Computing 2006, Granada, Spain, 2006.

[23] R. Bradford. A parallelization of the Buchberger algorithm. In *Proc. of ISSAC90*, New York, NY, USA, 1990. ACM Press.

[24] 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.

[25] P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morisson, J. M. Steinbach, R. S. Sutor, and S. M. Watt. *AXIOM Library Compiler User Guide*. NAG, The Numerical Algorithms Group Limited, Oxford, United Kingdom, 1st edition, November 1994. AXIOM is a registered trade mark of NAG.

[26] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.

[27] R. Bündgen, M. Göbel, and W. Küchlin. A fine-grained parallel completion procedure. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 269–277, New York, NY, USA, 1994. ACM Press.

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

[29] S. Chakrabarti and K. Yelick. Distributed data structures and algorithms for Gröbner basis computation. *LISP AND SYMBOLIC COMPUTATION: An International Journal*, 7:147–172, 1994.

[30] C. Chen, F. Lemaire, O. Golubitsky, M. Moreno Maza, and W. Pan. Comprehensive triangular decomposition. In *Proc. of CASC'07*. Springer, 2007.

[31] C. Chen, F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. Proc. of CASA2007, 2007.

[32] C. Chen, M. M. Maza, W. Pan, and Y. Xie. On the verification of polynomial system solvers. In *Proceedings of AWFS 2007*, 2007.

[33] Y. Chicha and S. M. Watt. A localized tracing scheme applied to garbage collection. In *APLAS 2006*, 2006.

[34] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Spinger-Verlag, 1st edition, 1992.

[35] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Spinger-Verlag, 2nd edition, 1997.

[36] D. Cox, J. Little, and D. O'Shea. *Using Algebraic Geometry.* Graduate Text in Mathematics, 185. Springer-Verlag, New-York, 1998.

[37] X. Dahan, M. Moreno Maza, W. W. É. Schost, and Y. Xie. On the complexity of the D5 principle. Poster, ISSAC 2005, 2005.

[38] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Equiprojectable decompositions of zero-dimensional varieties. In A. Valibouze, editor, *International Conference on Poynomial System Solving.* University of Paris 6, France, 2004.

[39] 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.

[40] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. The complexity of the Split-and-Merge algorithm. In preparation.

[41] 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.

[42] X. Dahan and É. Schost. Sharp estimates for triangular sets. In *ISSAC 04*, pages 103–110. ACM, 2004.

[43] 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.

[44] S. Dellière. *Triangularisation de systèmes constructibles. Application à l'évaluation dynamique.* PhD thesis, Université de Limoges, 1999.

[45] A. Díaz, M. Hitz, E. Kaltofen, A. Lobo, and T. Valente. Process scheduling in DSC and the large sparse linear systems challenge. *JSC*, 19(1–3):269–282, 1995.

[46] A. Díaz and E. Kaltofen. FoxBox: a system for manipulating symbolic objects in Black Box representation. In *Proc. ISSAC'98*, pages 30–37, 1998.

[47] L. Donati and C. Traverso. Experimenting the Gröbner basis algorithm with the ALPI system. In *Proc. ISSAC'89*, pages 192–198. ACN Press, 1989.

[48] D. Duval. *Questions Relatives au Calcul Formel avec des Nombres Algébriques.* Université de Grenoble, 1987. Thèse d'État.

[49] D. Duval. Algebraic numbers: an example of dynamic evaluation. *J. Symb. Comp.*, 18(5):429–446, November 1994.

[50] D. Eisenbud. *Commutative Algebra.* Graduate Text in Mathematics, 150. Springer-Verlag, New-York, 1994.

[51] D. Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry.* Springer-Verlag, New York-Berlin-Heidelberg, 1995.

[52] J.-C. Faugère. Parallelization of Gröbner bases. In *Proc. PASCO'94.* World Scientific Publishing Company, 1994.

[53] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases. *J. Pure and Appl. Algebra*, 139(1-3):61–88, 1999.

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

[55] M. Foursov and M. Moreno Maza. On computer-assisted classification of coupled integrable equations. *J. Symb. Comp.*, 33:647–660, 2002.

[56] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.

[57] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[58] T. Gautier and N. Mannhart. Parallelism in Aldor – the communication library $\prod^{IT}$ for parallel, distributed computation. In *ACPC-2, LNCS vol.734*, pages 204–218, 1998.

[59] T. Gautier, J. Roch, and F. Wagner. Fine grained distributed implementation of a language with provable performance. In *Proceedings of ICCS2007/PAPP2007*, May 2007.

[60] T. Gautier and J.-L. Roch. NC^2 computation of gcd-free basis and application to parallel algebraic numbers computation. In *PASCO '97: Proceedings of the*

second international symposium on parallel symbolic computation, pages 31–37. ACM Press, 1997.

[61] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1999.

[62] P. Gianni, B. Trager, and G. Zacharias. Gröbner bases and primary decomposition of polynomial ideals. *J. Symb. Comp.*, 6:149–167, 1988.

[63] M. Giusti, J. Heintz, J. E. Morais, and L. M. Pardo. When polynomial equation systems can be solved fast? In *AAECC-11*, volume 948 of *LNCS*, pages 205–231. Springer, 1995.

[64] M. Giusti, G. Lecerf, and B. Salvy. A Gröbner free alternative for polynomial system solving. *J. Complexity*, 17(1):154–211, 2001.

[65] T. Gómez Díaz. *Quelques applications de l'évaluation dynamique*. PhD thesis, Université de Limoges, 1994.

[66] M. González-López and T. Recio. The ROMIN inverse geometric model and the dynamic evaluation method. In A. M. Cohen, editor, *Proc. of the 1991 SCAFI Seminar, Computer Algebra in Industry*. Wiley, 1993.

[67] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook*. Springer, 2003.

[68] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(3):416–429, March 1969.

[69] R. Hartshorne. *Algebraic Geometry*. Springer-Verlag, New-York, 1997.

[70] H. Hong and H. W. Loidl. Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In B. Buchberger and J. Volkert, editors, *Proc. of CONPAR 94–VAPP VI, Springer LNCS 854.*, pages 325–336. Springer Verlag, September 1994.

[71] É. Hubert. Notes on triangular sets and triangulation-decomposition algorithms. I. Polynomial systems. In *Symbolic and numerical scientific computation (Hagenberg, 2001)*, volume 2630 of *LNCS*, pages 1–39. Springer, 2003.

[72] 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.

[73] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical taylor shift by 1. In *ISSAC'05*, pages 200–207. ACM Press, 2005.

[74] KAAPI Group. KAAPI: Kernel for Adaptive, Asynchronous Parallel and Interactive programming. http://kaapi.gforge.inria.fr/.

[75] M. Kalkbrener. *Three contributions to elimination theory.* PhD thesis, Johannes Kepler University, Linz, 1991.

[76] M. Kalkbrener. A generalized euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143 – 167, 1993.

[77] E. Kaltofen. Fast parallel absolute irreducibility testing. *JSC*, 1(1):57–67, 1985. Misprint corrections: JSC. vol.9, p.320 (1989).

[78] E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24(3–4):331–348, July–Aug. 1999. Special Issue on "Coarse Grained Parallel Algorithms".

[79] E. Kaltofen and B. Trager. Computing with polynomials given by Black Boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *JSC*, 9(3):301–320, 1990.

[80] I. A. Kogan and M. Moreno Maza. Computation of canonical forms for ternary cubics. In T. Mora, editor, *Proc. ISSAC 2002*, pages 151–160. ACM Press, July 2002.

[81] T. Krick, L. M. Pardo, and M. Sombra. Sharp estimates for the arithmetic Nullstellensatz. *Duke Math. J.*, 109(3):521–598, 2001.

[82] L. Kronecker. Die Zerlegung der ganzen Grössen eines natürlichen Rationalitäts-Bereichs in ihre irreductibeln Factoren. *J. für die Reine und Angewandte Mathematik*, 94:344–348, 1897.

[83] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *J. für die Reine und Angewandte Mathematik*, 92:1–122, 1897.

[84] W. W. Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In *AAECC-8, LNCS vol.508*, pages 341–353, 1990.

[85] 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.

[86] D. Lazard. A new method for solving algebraic systems of positive dimension. *Discr. App. Math*, 33:147–160, 1991.

[87] D. Lazard. Solving zero-dimensional algebraic systems. *J. Symb. Comp.*, 15:117–132, 1992.

[88] G. Lecerf. Computing the equidimensional decomposition of an algebraic closed set by means of lifting fibers. *J. Complexity*, 19(4):564–596, 2003.

[89] C. E. Leiserson and M. Frigo. The Cilk Project . http://supertech.csail.mit.edu/cilk/, 2007.

[90] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In *Maple 10*, Maplesoft, Canada, 2005. Refereed software.

[91] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[92] F. Lemaire, M. Moreno Maza, and Y. Xie. Making a sophisticated symbolic solver available to different communities of users. In *Proc. of* Asian Technology Conference in Mathematics*'06*, 2006.

[93] A. K. Lenstra, J. H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen.*, 261:515–534, 1982.

[94] A. Leykin. On parallel computation of Gröbner bases. In *ICPP Workshops*, pages 160–164, 2004.

[95] X. Li. Efficient management of symbolic computations with polynomials, 2005. Masters Thesis, University of Western Ontario.

[96] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In A. Iglesias and N. Takayama, editors, *Proc. Mathematical Software - ICMS 2006*, pages 12–23. Springer, 2006.

[97] X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. In *Proc. ISSAC'07*, pages 53–59, New York, NY, USA, 2006. ACM Press.

[98] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: From theory to practice. In *Proc. ISSAC'07*, pages 269–276, New York, NY, USA, 2007. ACM Press.

[99] X. Li, M. Moreno Maza, and É. Schost. On the virtues of generic programming for symbolic computation. In *Proc. of the International Conference on Computational Science (2)*, pages 251–258. Springer, 2007.

[100] H. Lombardi. Structures algébriques dynamiques, espaces topologiques sans points et programme de hilbert. *Ann. Pure Appl. Logic*, 137(1-3):256–290, 2006.

[101] M. Moreno Maza and Y. Xie. An implementation report for parallel triangular decompositions on a shared memory multiprocessor. In *Proc. of* Symposium on Parallelism in Algorithms and Architectures*'06*. ACM Press, 2006.

[102] M. Moreno Maza and Y. Xie. Parallelization of triangular decompositions. In *Proc. of Algebraic Geometry and Geometric Modelling'06*, pages 96–100, Barcelona, Spain, 2006.

[103] N. Mannhart. $\prod^{IT}$: *a portable communication library for distributed computer algebra*. PhD thesis, Swiss Federal Institute of Technology, 1997.

[104] M. Manubens and A. Montes. Improving DISPGB algorithm using the discriminant ideal (extended abstract). In *In A3L-2005 Proceedings*, 2005.

[105] Maplesoft. *Maple 10*. http://www.maplesoft.com/, 2005.

[106] T. Mora. *Solving Polynomial Equation Systems I. The Kronecker-Duval Philosophy.* Number 88 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[107] M. Moreno Maza. *Calculs de Pgcd au-dessus des Tours d'Extensions Simples et Résolution des Systèmes d'Équations Algébriques*. PhD thesis, Université Paris 6, 1997.

[108] 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. http://www.csd.uwo.ca/~moreno.

[109] M. Moreno Maza and R. Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*, pages 365–382. Springer, 1995.

[110] M. Moreno Maza, B. Stephenson, S. M. Watt, and Y. Xie. Multiprocessed parallelism support in Aldor on SMPs and multicores. In *Proc. PASCO'07*, pages 60–68, New York, NY, USA, 2006. ACM Press.

[111] M. Moreno Maza and Y. Xie. Component-level parallelization of triangular decompositions. In *Proc. PASCO'07*, pages 69–77, New York, NY, USA, 2006. ACM Press.

[112] J. O'Halloran and M. Schilmoeller. Gröbner bases for constructible sets. *Journal of Communications in Algebra*, 30(11), 2002.

[113] F. Pauer. On lucky ideals for Gröbner basis computations. *J. Symb. Comp.*, 14(5):471–482, Nov. 1992.

[114] Project LinBox. *Exact computational linear algebra*. http://www.linalg.org/.

[115] M. O. Rayes, P. S. Wang, and K. Weber. Parallelization of the sparse modular gcd algorithm for multivariate polynomials on shared memory multiprocessors. In *Proc. of the international symposium on Symbolic and algebraic computation*, pages 66–73, New York, NY, USA, 1994. ACM Press.

[116] R. Rioboo. Real algebraic closure of an ordered field, implementation in AXIOM. In *Proc. ISSAC'92*, pages 206–215. ISSAC, ACM Press, 1992.

[117] J. F. Ritt. *Differential Equations from an Algebraic Standpoint*, volume 14. American Mathematical Society, New York, 1932.

[118] P. Samuel and O. Zariski. *Commutative algebra*. D. Van Nostrand Company, INC., 1967.

[119] É. Schost. Degree bounds and lifting techniques for triangular sets. In T. Mora, editor, *Proc. ISSAC 2002*, pages 238–245. ACM Press, July 2002.

[120] É. Schost. Complexity results for triangular sets. *J. Symb. Comp.*, 36(3-4):555–594, 2003.

[121] W. Schreiner and H. Hong. The design of the PACLIB kernel for parallel algebraic computation. In *ACPC-2, LNCS vol.734*, pages 204–218, 1993.

[122] T. Shimoyama and K. Yokoyama. Localization and primary decomposition of polynomial ideals. *J. Symb. Comput.*, 22(3):247–277, 1996.

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

[124] V. Shoup. The number theory library, 1996–2006.

[125] W. Sit. Computations on quasi-algebraic sets. In R. Liska, editor, *Electronic Proceedings of IMACS ACA'98*, 1998.

[126] A. Sommese, J. Verschelde, and C. Wampler. Numerical decomposition of the solution sets of polynomial systems into irreducible components. *SIAM J. Numer. Anal.*, 38(6):2022–2046, 2001.

[127] Sun Fortress project group. The Sun Fortress Project. fortress.sunsource.net, 2007.

[128] *The SymbolicData Project.* http://www.SymbolicData.org, 2000–2006.

[129] L. N. T. Tzen. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1), 1993.

[130] The BLAS Project Group. BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/, 2007.

[131] The Computational Mathematics Group. AXIOM 2.2. NAG Ltd, Oxford, UK, 1998.

[132] The Computational Mathematics Group. The BasicMath Library. NAG Ltd, Oxford, UK, 1998. http://www.nag.co.uk/projects/FRISCO.html.

[133] The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2004 Edition. http://www.opengroup.org/onlinepubs/009695399/.

[134] W. Trinks. On improving approximate results of Buchberger's algorithm by Newton's method. In *EUROCAL 85*, volume 204 of *LNCS*, pages 608–611. Springer, 1985.

[135] D. Wang. Computing triangular systems and regular systems. *Journal of Symbolic Computation*, 30(2):221–236, 2000.

[136] D. Wang. *Elimination Practice: Software Tools and Applications.* Imperial College Press, London, UK, UK, 2004.

[137] D. Wang and B. Xia. Stability analysis of biological systems with real solution classfication. In M. Kauers, editor, *Proc. 2005 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 354–361, New York, 2005. ACM Press.

[138] D. M. Wang. An elimination method for polynomial systems. *J. Symb. Comp.*, 16:83–114, 1993.

[139] D. M. Wang. Decomposing polynomial systems into simple systems. *J. Symb. Comp.*, 25(3):295–314, 1998.

[140] D. M. Wang. *Elimination Methods*. Springer, Wein, New York, 2000.

[141] S. M. Watt. A system for parallel computer algebra programs. In *LNCS Vol.204*, pages 537–538, 1985.

[142] S. M. Watt. Aldor. In J. Grabmeier, E.Kaltofen, and V. Weispfenning, editors, *Computer Algebra Handbook*, pages 265 – 270. Springer, 2003.

[143] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. A first report on the $A^{\#}$ compiler. In *Proceedings of ISSAC '94*, New York, NY, USA, 1994. ACM Press.

[144] V. Weispfenning. Canonical comprehensive Gröbner bases. In *ISSAC 2002*, pages 270–276. ACM Press, 2002.

[145] W. T. Wu. On zeros of algebraic equations – an application of Ritt principle. *Kexue Tongbao*, 31(1):1–5, 1986.

[146] W. T. Wu. A zero structure theorem for polynomial equations solving. *MM Research Preprints*, 1:2–12, 1987.

[147] Y. Wu, W. Liao, D. Lin, and P. Wang. Local and remote user interface for ELIMINO through OMEI. Technical report, Kent State University, Kent, Ohio, 2003. http://icm.mcs.kent.edu/reports/.

[148] Y. Wu, G. Yang, H. Yang, W. Zheng, and D. Lin. A distributed computing model for Wu's method. *Journal of Software (in Chinese)*, 16(3), 2005.

[149] 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.

[150] C. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1993.

[151] T. Yuasa, M. Hagiya, and W. F. Schelter. *GNU Common Lisp.* `http://www.gnu.org/software/gcl`.

[152] Zassenhaus, H. On Hensel factorization I. *Journal of Number Theory*, 1:291–311, 1969.

Curriculum Vitae

Name: **Yuzhen Xie**

Post-Secondary   The University of Western Ontario
Education and   London, Ontario, Canada
Degrees: Ph.D. Computer Algebra
                       September 2007

                       The University of Western Ontario
                       London, Ontario, Canada
                       M.Sc. and B.Sc. Computer Science
                       May 2002

                       B.E.Sc. in Systems Modeling
                       Civil & Environmental Engineering
                       Tsinghua University, 1988

Selected Honors   NSERC PDF, 09/2007 - 08/2009
and Awards:

                       UWO Thesis Award, 2007

                       CS Publications Incentive Award, 2005 - 2006
                       Computer Science Department, UWO

                       Ontario Graduate Scholarship (OGS)
                       2004, 2005 and 2006

                       ISSAC'2005 Distinguished Student Author Award, 2005

                       ISSAC'2005 Best Poster Award, 2005

                       Deans Honor List, UWO, 2001

Selected Honors and Awards Continued:	Canadian New Millennium Scholarship, 2000
	Nominee of Award of Excellence in Teaching Assistance UWO, 1999
	Western Graduate Research Scholarship UWO, 1998, 1999, 2002 and 2003
	1st Prize for National Academic Progress National Education Committee of China, 1993
	Honor of China National Scholarship Peking University, 1991
	Graduate Award of Academic Excellence Tsinghua University, 1988
Related Work Experience:	Research Assistant & Teaching Assistant Computer Science Department University of Western Ontario 2002 - 2007
	Software and Systems Developer FAG Aerospace Canada 2000 - 2001
	Research Assistant & Teaching Assistant Civil & Environmental Engineering Department University of Western Ontario 1998 - 1999
	Engineer Design and Planning Division China Urban Planning and Design Institute 1991 - 1997

Refereed Papers:

Moreno Maza, M. and **Xie, Y.** (2007) Component-level parallelization of triangular decompositions. Proceedings of Parallel Symbolic Computation (PASCO), p69-77, London, Canada.

Moreno Maza, M. and **Xie, Y.** (2007) Multiprocessed parallelism support in ALDOR on SMPs and multicores. Proceedings of PASCO2007, p60-68, London, Canada.

Chen, C., Moreno Maza, M., Pan, W. and **Xie, Y.** (2007) On the verification of polynomial system solvers. Proceedings of Fifth Asian Workshop on Foundations of Software (AWFS), p116-144, Xiamen, China.

Chen, C., Lemaire, F., Moreno Maza, M., Pan, W. and **Xie, Y.** Efficient computations of irredundant triangular decompositions with the `RegularChains` library. Proceedings of the Fifth International Workshop on Computer Algebra Systems and Applications, p268-271, Beijing.

Lemaire, F., Moreno Maza, M. and **Xie, Y.** (2006) Making a sophisticated symbolic solver available to different communities of users. Proceedings of 11th Asian Technology Conference in Mathematics, 10 pages, Hong Kong.

Moreno Maza, M. and **Xie, Y.** (2006) Parallelization of triangular decompositions. Proceedings of Algebraic Geometry and Geometric Modeling 2006 (AGGM), p96-100, Barcelona University, Spain.

Moreno Maza, M. and **Xie, Y.** (2006) Brief Announcement: An implementation report for parallel triangular decompositions. Proceedings of 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), p235, MIT, US.

Dahan, X., Moreno Maza, M., Schost, É. and **Xie, Y.** (2006) On the complexity of D5 principle. Proceedings of Transgressive Computing, p149-167, Granada, Spain.

Refereed Papers Continued:

Dahan, X., Moreno Maza, M., Schost, É., Wu, W. and **Xie, Y.** (2005) Lifting techniques for triangular decompositions. Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC), p108-115, Beijing. (**ISSAC'2005 Distinguished Student Author Award**)

Lemaire, F., Moreno Maza, M. and **Xie, Y.** (2005) The `RegularChains` library in MAPLE, MAPLE Conference, p355-368, Waterloo, Canada.

Dahan, X., Moreno Maza, M., Schost, É., Wu, W. and **Xie, Y.** (2004) Equiprojectable decomposition of zero-dimensional varieties. Proceedings of International Conference on Polynomial System Solving, p69-71, Univ. of Paris 6, France.

Watt, S. M. and **Xie, Y.** (2002) A family of modular XML schemas for MathML. Proceedings of Internet Accessible Mathematical Computation Conference, 8 pages, Lille, France.

Watt, S., Padovani, L. and **Xie, Y.** (2002) A Lisp subset based on MathML. Proceedings of MathML 2002 Conference, 8 pages, Chicago, US.

Rowe, R.K., Shang, J.Q. and **Xie, Y.** (2002) Effect of permeating solutions on complex permittivity of compacted clay, Canadian Geotechnical Journal, 39: 1016-1025.

Rowe, R.K., Shang, J.Q. and **Xie, Y.** (2001) Complex permittivity measurement system for detecting soil contamination, Canadian Geotechnical Journal, 38: 498-506.

Refereed Conference Posters:

Chen, C., Moreno Maza, M., Pan, W. and **Xie, Y.** (2007) On the verification of polynomial system solvers. ISSAC2007, Waterloo, Canada.

Moreno Maza, M. and **Xie, Y.** (2006) Parallel triangular decompositions. The Mathematics of Information Technology and Complex Systems (MITACS) 7th Annual Conference, Toronto, Canada.

Moreno Maza, M. and **Xie, Y.** (2006) Solving polynomial systems symbolically and in parallel, ISSAC2006, Genova, Italy.

Dahan, X., Moreno Maza, M., Schost, É., Wu, W., and **Xie, Y.** (2005) On the complexity of D5 principle. ISSAC2005, Beijing. (**ISSAC'2005 Best Poster Award**)

Lemaire, F., Moreno Maza, M., Wu, W., and **Xie, Y.** (2005) The `RegularChains` library in MAPLE 10, ISSAC'2005, Beijing.

Dahan, X., Moreno Maza, M., Schost, É., Wu, W., and **Xie, Y.** (2005) Equiprojectable decomposition of zero-dimensional varieties. MITACS 6th Annual Conference, Calgary, Alberta.

Software Packages:	Chen, C., Moreno Maza, M., Pan, W. and **Xie, Y.** (2007) A verifier for symbolic solvers (on top of the `RegularChains` library). Prepared to be released with MAPLE 12.

Moreno Maza, M. and **Xie, Y.** (2007) Parallel solver in ALDOR for computing triangular decompositions (with parallel framework). Available on request.

Dahan, X., Moreno Maza, M., Schost, É., Wu, W. and **Xie, Y.** (2006) Modular methods for triangular decompositions in the `RegularChains` library. Distributed in MAPLE 11, Maplesoft, Waterloo, Canada, August 2006

Lemaire, F., Moreno Maza, M. and **Xie, Y.** (2005) The `RegularChains` library. Shipped with MAPLE since Version 10, Maplesoft, Waterloo, Canada, May 2005. http://www.maplesoft.com

Xie, Y. (2002) "XML-Scheme Interpreter" software package, ORCCA, London, Ontario. http://www.orcca.on.ca/mathml/

Xie, Y. (2001) "enttran" program (to perform conversion between the types of entity references in an XML file or HTML file), ORCCA, London, Ontario. http://www.orcca.on.ca/mathml/

Xie, Y. (2000) "Design and Process List (DPL) Project Manager", enterprise distributed software, FAG Aerospace Canada (Industrial development) |