10-27-2021 2:30 PM

Cache-Friendly, Modular and Parallel Schemes For Computing Subresultant Chains

Mohammadali Asadi, *The University of Western Ontario*

Supervisor: Marc Moreno Maza, *The University of Western Ontario*

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

Recommended Citation

Asadi, Mohammadali, "Cache-Friendly, Modular and Parallel Schemes For Computing Subresultant Chains" (2021). *Electronic Thesis and Dissertation Repository*. 8205.
https://ir.lib.uwo.ca/etd/8205

Abstract

The RegularChains library in Maple offers a collection of commands for solving polynomial systems symbolically with taking advantage of the theory of regular chains. The primary goal of this thesis is algorithmic contributions, in particular, to high-performance computational schemes for subresultant chains and underlying routines to extend that of RegularChains in a C/C++ open-source library.

Subresultants are one of the most fundamental tools in computer algebra. They are at the core of numerous algorithms including, but not limited to, polynomial GCD computations, polynomial system solving, and symbolic integration. When the subresultant chain of two polynomials is involved in a client procedure, not all polynomials of the chain, or not all coefficients of a given subresultant, may be needed. Based on that observation, we design so-called speculative and caching strategies which yield great performance improvements within our polynomial system solver.

Our implementation of these techniques has been highly optimized. We have implemented optimized core arithmetic routines and multithreaded subresultant algorithms for univariate, bivariate and multivariate polynomials. We further examine memory access patterns and data locality for computing subresultants of multivariate polynomials, and study different optimization techniques for the fraction-free LU decomposition algorithm to compute subresultants based on determinant of Bézout matrices.

Our code is publicly available at www.bpaslib.org as part of the Basic Polynomial Algebra Subprograms (BPAS) library that is mainly written in C, with concurrency support and user interfaces written in C++.

Keywords: Subresultant chain, GCDs, modular arithmetic, Ducos' subresultant chain, Bézout matrix, polynomials system solving

Summary for Lay Audience

One of the most fundamental subjects in scientific computing is polynomial system solving; accordingly, almost all available computer algebra systems rely on polynomial system solvers. The `RegularChains` library in MAPLE offers a collection of commands for solving polynomial systems symbolically with taking advantage of the theory of regular chains. The primary goal of this thesis is algorithmic contributions, in particular, to high-performance computational schemes for subresultant chains and underlying routines to extend that of `RegularChains` as part of the Basic Polynomial Algebra Subprograms (BPAS) library that is mainly written in C, with concurrency support and user interfaces written in C++.

Subresultants are one of the most fundamental tools in computer algebra. They are at the core of numerous algorithms including, but not limited to, polynomial GCD computations, polynomial system solving, and symbolic integration. When the subresultant chain of two polynomials is involved in a client procedure, not all polynomials of the chain, or not all coefficients of a given subresultant, may be needed. Based on that observation, we design so-called speculative and caching strategies which yield great performance improvements within our polynomial system solver.

The notion of speculative algorithm is inspired by the definition of speculative execution on computer hardware. We mean an algorithm that would normally compute a sequence S of items but assumes that only a prescribed sub-sequence S' of those may be needed. It manages to increase performance by taking this observation into account, while being able to recover S from the calculations of S' if the whole S turns out to be needed. Thus, the cost of that recovery along with computing S' is essentially that of computing S directly.

Our implementation of these techniques has been highly optimized through developing asymptotically fast core arithmetic routines and multithreaded subresultant algorithms for univariate, bivariate and multivariate polynomials. We further examine memory access patterns and data locality for computing subresultants of multivariate polynomials. For matrices over multivariate polynomials, we make use of the Bareiss fraction-free LU decomposition and further optimize this algorithm with a so-called smart-pivoting technique to reduce the cost of the decomposition and compute subresultants based on determinant of Bézout and Hybrid Bézout matrices.

Acknowlegements

I would like to express my sincere appreciation to my supervisor Professor Marc Moreno Maza for his invaluable lessons, continuous inspiration and never-ending support throughout my studies.

While my name is the only one that appears on the author list of this thesis, there are several other people deserving recognition. I wish to extend my gratitude to my supervisor as I am the lucky beneficiary of his wide knowledge. I would also like to thank to my wonderful and insightful colleagues, Alex Brandt and Robert Moir for their great help in the past four years.

Many thanks to the members of my committee Dr. Laureano Gonzalez-Vega, Dr. Roberto Solis-Oba, Dr. David John Jeffrey, and Dr. Robert M. Corless for their inspiration, comments and questions.

My special gratitude goes to my parents and my sister for their love and support in all aspects of my life, and for unwavering belief in me. I would like to show my gratitude and appreciation to my partner, Shokooh for her love and constant support, and for keeping me sane over the past few months.

Contents

Abstract **i**

Summary for Lay Audience **ii**

Acknowlegements **iii**

List of Figures **vii**

List of Tables **x**

1 Introduction **1**

1.1 A Brief History of Solving Systems of Equations 2

1.2 Triangular Decompositions of Polynomial Systems 4

1.3 Incremental Solving and the `Intersect` Algorithm 6

1.4 Subresultant Chains 8

1.4.1 The First Example 8

1.4.2 Theory of Subresultants (informally) 10

1.4.3 The Second Example 11

1.4.4 The Third Example 12

1.4.5 Subresultants in Theory and Practice 13

1.5 Objectives and Contributions 13

2 Primary Background **16**

2.1 `Triangularize` Algorithm 16

2.2 Theory of Subresultant Chain 19

2.3 BPAS Multithreaded Interface 23

3 Dense Univariate and Bivariate Polynomials **26**

3.1 Introduction 26

3.2 Basic Arithmetic 29

3.3 Fast Arithmetic over Prime Fields . . . 34
3.3.1 Montgomery Representation . . . 35
3.3.2 Univariate Polynomial Multiplication . . . 35
3.3.3 Division Operation . . . 37
3.3.4 Fast EEA (Half-GCD) Operation . . . 38
3.3.5 Evaluation-Interpolation Operations . . . 39
3.4 Speculative Subresultant Algorithms . . . 41
3.5 Parallel Subresultant Algorithms . . . 47
3.6 Experimentation . . . 51
3.6.1 Routines in $\mathbb{Z}_p[y]$ . . . 51
3.6.2 Subresultants in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$ . . . 54

4 Sparse Multivariate Polynomials **62**
4.1 Introduction . . . 62
4.2 Basic Arithmetic . . . 63
4.3 Multi-Divisor Division and Pseudo-Division . . . 67
4.3.1 Triangular Set Normal Form . . . 68
4.3.2 Triangular Set Pseudo-Division . . . 70
4.3.3 Experimentation . . . 74
4.4 Ducos' Subresultant Chain Algorithm . . . 76
4.4.1 Original Scheme . . . 77
4.4.2 Memory-Efficient Scheme . . . 79
4.4.3 Experimentation . . . 81

5 Bézout Matrix over Multivariate Polynomials **84**
5.1 Introduction . . . 84
5.2 Fraction-Free LU Decomposition . . . 86
5.2.1 Bareiss FFLU Algorithm . . . 87
5.2.2 Smart-Pivoting in FFLU Algorithm . . . 91
5.2.3 Parallel FFLU Algorithm . . . 92
5.2.4 Experimentation . . . 94
5.3 Bézout Subresultant Algorithms . . . 97
5.3.1 Bézout Matrix and Subresultants . . . 97
5.3.2 Speculative Bézout Subresultant Algorithms . . . 102
5.3.3 Experimentation . . . 105

6 Multivariate Power Series in MAPLE **109**
6.1 Introduction 109
6.2 Preliminary 111
6.2.1 Power Series 111
6.2.2 Univariate Polynomials over Power Series 112
6.3 An Overview of the User-Interface 113
6.4 Design Principles 118
6.4.1 Lazy Evaluation 118
6.4.2 MAPLE Data Structures and Built-in Functions 119
6.4.3 MAPLE Objects 120
6.5 Implementation of `MultivariatePowerSeries` 121
6.5.1 `PowerSeriesObject` 122
6.5.2 `UnivariatePolynomialOverPowerSeriesObject` 124
6.6 Experimentation 128

7 Conclusion and Future Work **132**
7.1 Future Work 134

Bibliography **136**

Curriculum Vitae **142**

List of Figures

1.1 A circle and an hyperbola center at the point of coordinates (1,1). 2

3.1 Computing the subresultant chain of $a, b \in \mathbb{Z}[x, y]$ using modular arithmetic, evaluation-interpolation and CRT algorithms where $(t_0, \ldots, t_N)$ is the list of evaluation points, $(p_0, \ldots, p_i)$ is the list of distinct primes, $\bar{a} = a \bmod p_i$, and $\bar{b} = b \bmod p_i$. 46

3.2 A one-dimensional C-array representing polynomial $a = \sum_{i=0}^{n} a_i y^i$ of $n+1$ terms showing GMP trees and GMP heads as a_0, a_1, . . . , a_n. 51

3.3 Comparing plain, Karatsuba, and FFT-based multiplication in BPAS with the wrapper `mul` method in NTL to compute ab for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$. 52

3.4 Comparing Euclidean and fast division algorithms in BPAS with the division method in NTL to compute $\mathrm{rem}(a, b)$ and $\mathrm{quo}(a, b)$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = 2(\deg(b) - 1) = d$. 53

3.5 Comparing Euclidean-based GCD and Half-GCD-based GCD algorithms in BPAS with the GCD algorithm in NTL to compute $\gcd(a, b) = 1$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$. 54

3.6 Comparing EEA, modular subresultant, and Half-GCD-based subresultant (`BPAS_specSRC`, $\rho = 0, 2$), in BPAS for dense polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$. 55

3.7 Comparing (optimized) Ducos' subresultant chain algorithm, modular subresultant chain, and speculative subresultant for $\rho = 0, 2$, algorithms in BPAS with Ducos' subresultant chain algorithm in MAPLE for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$. 56

3.9 Comparing Opt. Ducos' algorithm (the top surface) and modular subresultant chain (the bottom surface) to compute the entire chain for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$. 56

3.10 Comparing modular subresultant chain using FFT (the top surface), and speculative subresultant ($\rho = 0$) (the bottom surface) for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$. 56

3.8 Comparing (optimized) Ducos' subresultant chain, modular subresultant chain, and speculative subresultant for $\rho = 0, 2, 4, 6$, in BPAS with Ducos' algorithm in Maple for dense polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = 50$ and $\deg(a, x) = \deg(b, x) + 1 = d$. 57

3.11 The parallel-speed-up of the `Triangularize` algorithm in BPAS when we use parallel subresultant algorithms (Section 3.5) for a test suite of more than 3000 polynomial systems. 59

4.1 Comparing normal form algorithms over $\mathbb{Q}$ where polynomials are in $\mathbb{Q}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit. 75

4.2 Comparing normal form algorithms over $\mathbb{Z}$ where polynomials are in $\mathbb{Z}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit. 75

4.3 Comparing triangular set pseudo-division algorithms over $\mathbb{Q}$ where polynomials are in $\mathbb{Q}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit. . . 76

4.4 Comparing triangular set pseudo-division algorithms over $\mathbb{Z}$ where polynomials are in $\mathbb{Z}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit. . . 76

5.1 Comparing Algorithm 24 (`BPAS_FFLU_Serial`) and Algorithm 25 (`BPAS_FFLU_Parallel`) for $n \times n$ matrices filled with random non-zero univariate polynomials with integer coefficients and degree 1. 96

6.1 List of the commands of `MultivariatePowerSeries`. 113

6.2 Creating power series from a polynomial or an anonymous function. . . . 114

6.3 Creating a univariate polynomial over power series 115

6.4 Controlling the output format of a multivariate power series. 115

6.5 Factoring univariate polynomials using `WeierstrassPreparation`. 116

6.6 Factoring univariate polynomials using `HenselFactorize`. 117

6.7 Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_1 = 1 + X_1 + X_2$. 129

6.8 Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_2 = 1 + X_1 + X_2 + X_3$. 129

6.9 Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_3 = 2 + \frac{1}{3}(X_1 + X_2)$. 130

6.10 Computing $f = \sum_{i=1}^{k} \frac{1}{1-x-y}$ using m-ary and binary addition. 130

6.11 Computing Weierstrass preparation of $f_1 = \frac{1}{1+X_1+X_2}{X_3}^k + {X_3}^{k-1} + \cdots + X_2X_3 + X_1 \in \mathbf{k}[\![X_1, X_2]\!][X_3]$. 130
6.12 Computing Weierstrass preparation of $f_2 = \frac{1}{1+X_1+X_2}{X_3}^k + X_2{X_3}^{k-1} + \cdots + X_3 + X_1 \in \mathbf{k}[\![X_1, X_2]\!][X_3]$. 130
6.13 For $f = \prod_{i=1}^{k}(X_2 - i) + X_1({X_2}^{k-1} + X_2)$, computing HenselFactorize(f). 131
6.14 For $f = \prod_{i=1}^{k}(X_2 - i) + X_1({X_2}^{k-1} + X_2)$, computing TaylorShift($f, 1$). 131

List of Tables

3.1 Comparing the execution time (in seconds) of subresultant implementations for constructed bivariate systems in Listing 3.1. Column headings follow Table 3.3, and `FFTBlockSize` is block size in the FFT-based evaluation and interpolation algorithms. 58

3.2 Comparing the execution time (in seconds) of subresultant algorithms for constructed bivariate systems in Listing 3.2. Column headings follow Table 3.1. 59

3.3 Comparing the execution time (in seconds) of subresultant algorithms on the BPAS `Triangularize` solver for well-known bivariate systems in the literature. deg(src[idx]) shows a list of minimum main degrees of the computed subresultants in each subresultant call and `Indexes` indicates a list of requested subresultant indexes. 60

4.1 Ducos' test examples in [37, Section 5]. 81

4.2 Comparing memory usage (GB) of Ducos' subresultant chain algorithms for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$ in Figure 3.7 in $\mathbb{Z}[y]$. 82

4.3 Comparing the execution time (in seconds) of Ducos' subresultant chain algorithms for polynomials in Table 4.1. 83

4.4 Comparing memory usage (MB) of Ducos' subresultant chain algorithms for extended Ducos' Tests in Table 4.1 where $\deg(a, x) = deg(b, x) + 1 = d$. 83

5.1 Compare the execution time (in seconds) of fraction-free LU decomposition algorithms for Bézout matrix of randomly generated, non-zero and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_5]$ with $x_5 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1)+1 = d$, $\deg(a, x_2) = \deg(b, x_2)+1 = 5$, $\deg(a, x_3) = \deg(b, x_3) = 1$, $\deg(a, x_4) = \deg(b, x_4) = 1$, $\deg(a, x_5) = \deg(b, x_5) = 1$. 95

5.2 Ratios of FFLU algorithms for polynomials in Table 5.1. 95

5.3 Comparing the execution time (in seconds) of Algorithm 24 and Algorithm 25 for $n \times n$ matrices filled with random non-zero univariate polynomials with integer coefficients and degree 1. 96
5.4 Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_6, x_5, \ldots, x_1]$ with $x_6 < x_5 < \cdots < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 6$. 105
5.5 Comparing the execution time (in seconds) of speculative subresultant algorithms for polynomials in Table 5.4. 107
5.6 Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_7]$ with $x_7 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 7$. 107
5.7 Comparing the execution time (in seconds) of speculative and caching subresultant algorithms for polynomials in Table 5.6. 107
5.8 Comparing the execution time (in seconds) of systems with `nvar` ≥ 5 from the pool of more than 3000 polynomial systems [10] so that their running-times ($t \in \mathbb{R}$) using *OptDucos* take $1 \leq t \leq 50$ seconds. 108
5.9 Comparing the execution time (in seconds) of systems with `nvar` ≥ 5 from the pool of more than 3000 polynomial systems [10] so that their running-times using *OptDucos* take > 50 seconds. 108

Chapter 1

Introduction

Polynomials system solving is one of the most fundamental and important subjects in mathematical and computational sciences. Almost all available computer algebra systems, academic or commercial, rely on polynomial system solvers. We begin with a review of some of the basic features of polynomial system solving through a series of examples.

Consider $a = a_2x^2+a_1x+a_0$ a univariate polynomial in x with real number coefficients a_2, a_1, a_0 with $a_2 \neq 0$. From high-school mathematics, we know that this polynomial has two complex solutions and we use the well-known quadratic formula for calculating them:

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_2}.$$

This is in fact a symbolic solution for a univariate polynomial of degree 2. For a polynomial of degree 5 or more, Galois theory tells us that there is no general formula which expresses the roots of that polynomial in radicals. For example, the code below shows MAPLE [71] representation for the solutions of $a = x^5 - x + 1$. Notice that MAPLE is unable to compute explicit representations of the roots and instead returns 5 instances of `RootOf(_Z^5-_Z+1)`, where `_Z` is a generic variable.

```
> a := x^5-x+1:
> solve(a, x); # solve x^5-x+1 with respect to x
  RootOf(_Z^5-_Z+1,index=1), RootOf(_Z^5-_Z+1,index=2), RootOf(_Z^5-_Z
    +1,index=3), RootOf(_Z^5-_Z+1,index=4), RootOf(_Z^5-_Z+1,index=5)
```

Now, let $a(x_1, \ldots, x_n)$ be a multivariate polynomial with variables ordered as $x_1 < \cdots < x_n$. We can see a as a univariate polynomial in x_n with degree d:

$$a = a_d(x_1, \ldots, x_{n-1})x_n^d + \cdots + a_1(x_1, \ldots, x_{n-1})x_n + a_0(x_1, \ldots, x_{n-1}),$$

where x_n is referred to as the main variable and $a_i(x_1, \ldots, x_{n-1})$ are multivariate polynomials in $x_1, \ldots, x_{n-1}$. Hence, by the *fundamental theorem of algebra*, for any $x_1, \ldots, x_{n-1}$ such that $a_d(x_1, \ldots, x_{n-1}) \neq 0$, a has exactly d complex solutions in x_n, counted with multiplicities. For instance, consider the set of polynomial equations below:

$$\begin{cases} f_1 &= {x_2}^2 + {x_1}^2 - 2\,x_2 - 2\,x_1 - 2 \\ f_2 &= x_2\,x_1 - x_2 - x_1 \end{cases}, \tag{1.1}$$

each of them has two complex roots in x_2 for any real value of x_1. Figure 1.1 shows a plot of the two planar curves $f_1 = 0$ and $f_2 = 0$, they are coloured blue and red respectively. We shall return later to the question of computing the common solutions of $f_1 = f_2 = 0$. First, we will illustrate how the common solutions of a set of polynomial equations may look like.

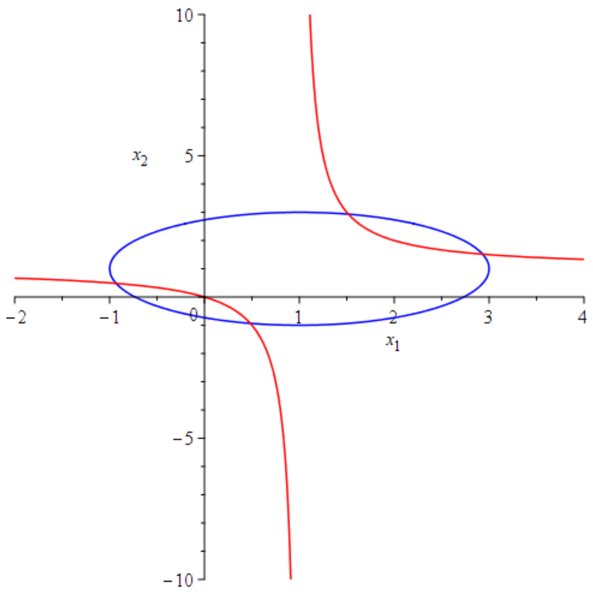


Figure 1.1: A circle and an hyperbola center at the point of coordinates (1,1).

1.1 A Brief History of Solving Systems of Equations

For an arbitrary system of polynomials, it is important to distinguish between the *linear case* and the *non-linear case*. In the former, any polynomial is actually a linear form, that is, a linear combination of the unknowns, $x_1, x_2, \ldots, x_n$. In this case, the set of common solutions is a linear space. In particular, a linear system has either a unique solution, no solution, or infinitely many solutions. In the non-linear case, terms can contain products of the unknowns and the set of common solutions is a so-called algebraic set. Examples of algebraic sets are algebraic curves (e.g. circles, hypoerbola, parabola, etc.) and algebraic surfaces (spheres, hyperboloids, paraboloids, etc.).

Consider the following system of linear equations:

$$L = \begin{cases} 3x + y + z - 1 = 0 \\ x + 3y + z - 1 = 0 \\ x + y + 3z - 1 = 0 \end{cases} .$$

Using the well-known method of *Gaussian elimination*, one can transform L to obtain the following equivalent system: $\{x - 1/5 = 0, y - 1/5 = 0, z - 1/5 = 0\}$. Solving linear systems has been studied intensively for centuries [74]. There are fast and optimized implementations in nearly every modern computer algebra software, e.g. [35, 47]. Counted as the number of arithmetic operations on the coefficients, the complexity of such an algorithm is cubic (and thus polynomial) in the number of unknowns. However, for non-linear systems, the time and space complexities are not polynomial in general. The time complexity of solving polynomials systems grows exponentially with the number of variables [59]. As a result, there are still very active research questions regarding the theory, algorithms, and implementation techniques for solving (non-linear) polynomial systems.

Before the age of computers, the theoretical question of solving polynomial systems was solved by the notion of a *primary decomposition* of a polynomial ideal, with the work of Emmanuel Lasker [57], Emmy Nöther [84], David Hilbert, Bartel Leendert van der Waerden, and Wolfgang Gröbner. While there exists algorithms and software implementations for primary decomposition today, this is not regarded as a tool of practical interest. First, because of its very high algebraic complexity, and second, because the information provided by primary decomposition is not well-suited to the needs of most applications of polynomial system solving. With the revolution of computers, the quest for effective algorithms solving polynomial systems began. In 1965, Buchberger [25] introduced the first such algorithm based on the concept of a *Gröbner basis.*

In general, algorithms for solving non-linear polynomial systems can be split into three categories: symbolic, numeric, and hybrid (using both symbolic and numeric techniques). Among them, symbolic solvers, like the Buchberger algorithm, are powerful tools in scientific computing with many applications, including cryptography, dynamical systems, and robotics [4, 5, 89]. Before the work of Buchberger [25], there were other algorithmic approaches to solve particular types of algebraic and differential polynomial systems symbolically. However, those algorithms could be neither implemented nor performed manually because of their high complexity. The *characteristic set method*, originally introduced by Ritt in [88] – for systems of ordinary differential equations – is the first "implementable" symbolic method for polynomial system solving. However, this method requires a relatively high computational cost due to its intensive use of polynomial factorization. In the early 1980's, Wu [101], relaxed the dependence of the characteristic set method on polynomial factorization, thus yielding the first *factorization-free* algorithm for decomposing polynomial systems. It should be noted that both Buchberger and Wu actually implemented their algorithms on computers.

Gröbner bases and characteristic sets are also representations of the solutions of a system of polynomials. There are fast implementations of system solvers based on Gröbner bases that can solve large and difficult polynomial systems [16, 38, 39]. On the other hand, characteristic set methods reveal more information about the geometry of the input systems than Gröbner bases and have a variety of applications. *Regular chains*, to be defined in the next section, extend characteristic sets to improve computational efficiency. Complexity results in [32] suggest that representing algebraic varieties as regular chains is nearly optimal in terms of space complexity. Hence, the development of fast algorithms and optimized software implementations based on characteristic sets and regular chains are important in the study of symbolic solutions of non-linear polynomial systems.

1.2 Triangular Decompositions of Polynomial Systems

Consider the following system of non-linear equations:

$$F = \begin{cases} x^2 + \ y + \ z = 1 \\ x + y^2 + \ z = 1 \\ x + \ y + z^2 = 1 \end{cases} .$$

We can use the command `Basis` from the library `Groebner` in MAPLE in order to compute a Gröbner basis of F under the lexicographic order $x < y < z$ (see [100, Chapter 24]) as follows:

```
> F := [x^2+y+z-1, y^2+x+z-1, z^2+x+y-1]:
> Groebner:-Basis(F, plex(z, y, x));
  [x^6-4x^4+4x^3-x^2, x^4+2x^2y-x^2, -x^2+y^2+x-y, x^2+y+z-1]
```

To obtain the following list of polynomial equations. The `factor` command can be used to further reveal the structure of the set of common solutions:

$$\begin{cases} x^2+y+z-1 & = 0 \\ (x+y-1)(y-x) & = 0 \\ x^2(x^2+2y-1) & = 0 \\ x^2(x^2+2x-1)(x-1)^2 & = 0 \end{cases}. \tag{1.2}$$

In the literature, a set of polynomials with pairwise different main variables is called a *triangular set*, by analogy to the case of linear systems. The equations defined by such a triangular set forms a *triangular set.* So, the above list of equations is not a triangular set since y appears as the main variable of the second and the third equations.

Performing elementary algebraic manipulations over the Gröbner basis (1.2) that is derived from F, we can transform F into a set of triangular sets so that every point of coordinates (x, y, z) is a solution of F if and only if it is a solution of one of the four systems below:

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

Furthermore, each of the above triangular sets is a so-called *regular chain* and these 4 regular chains together form a *triangular decomposition* of F. The following listing shows how to use the triangular decomposition algorithm as the command `Triangularize` from the `RegularChains` library [64] in Maple, to compute a triangular decomposition directly from the input system.

```
> with(RegularChains):
> F := {z^2+x+y-1, y^2+x+z-1, x^2+z+y-1}:
> R := PolynomialRing([z, y, x]):
> Triangularize(F, R);
  [regular_chain, regular_chain, regular_chain, regular_chain]
> map(Equations, %, R);
  [[z-1, y, x], [z, y-1, x], [z, y, x-1], [z-x, y-x, x^2+2x-1]]
```

The notion of a *regular chain* was introduced independently by Kalkbrener [53], and by Yang and Zhang [102]. The main purpose of this notion is to provide a mechanism to decide whether a triangular set is consistent (that is, has at least one solution) or not, what Wu's characteristic set method can not do. While this question of consistency may seem theoretical, it has important practical implications. For instance, the following triangular set, with $x < y$, is not a regular chain, while it is the triangular set produced by Wu's characteristic set method:

$$\begin{cases} y(x-1)+1 & = & 0 \\ x^2 & = & 1 \end{cases}.$$

The reason is because for the solution $x = 1$ of the second equation, the first equation has no y-solution, while for $x = -1$, one obtains $y = \frac{1}{2}$. Therefore, back substitution fails (for half of the x values) on the above system. To define the notion of a regular chain in an informal way, one can say that this is a triangular set where the back substitution process always succeeds.

In Kalkbrener's decomposition algorithm, given an input polynomial system F, the output regular chains represent *generic zeros* (in the sense of van der Waerden [99]) of the *irreducible components* of algebraic set $V(F)$. We will not try to define here the concepts of *generic zeros* and *irreducible components.* However, for a system F with finitely many solutions, the concept of *generic zeros* coincides with the familiar concept of "common solution".

1.3 Incremental Solving and the `Intersect` Algorithm

In the realm of methods for solving polynomial systems, further algorithmic improvements were made with the principle of *incremental solving* [58, 60]. With this principle, one equation is solved against each component (regular chain) produced by the previously solved equations. This allows one to control the properties and the size of the algebraic entities (curves, surfaces, etc.) that are produced at each computational step. Lazard proposed incremental solving for computing triangular decompositions in [58], which was improved and extended by Moreno Maza [81, 27] and others.

The incremental triangular decompositions algorithms in [81, 27] are based on a procedure, named `Intersect`, which computes the intersection of

- a hyper-surface and,
- the quasi-component of a regular chain denoted by $W(T)$ where T is the regular chain.

To rephrase that formal statement in loose terms, the procedure `Intersect` computes the common solutions of a triangular set and a polynomial equation. See [27] for algorithms.

Returning to a formal presentation, let $\mathbf{k}[x_1, \ldots, x_n]$ be the ring of multivariate polynomials with coefficients in a field $\mathbf{k}$ (say $\mathbb{Q}$ the field of rational numbers) and with variables $x_1 < \cdots < x_n$. For a polynomial $f \in \mathbf{k}[x_1, \ldots, x_n]$ and a regular chain T, the function call `Intersect`(f, T) returns regular chains $T_1, \ldots, T_e \subset \mathbf{k}[x_1, \ldots, x_n]$ such that we have:

$$V(f) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(f) \cap \overline{W(T)},$$

where $V(f)$ denotes the zero set of f, $W(T)$ denotes the common solutions of the polynomials of T and $\overline{W(T)}$ denotes the topological closure of $W(T)$ for the topology of Zariski. The reader may be surprised to see that $V(f) \cap W(T)$ is not given by an equality. Instead, `Intersect`(f, T) computes an "approximate decomposition" $W(T_1) \cup \cdots \cup W(T_e)$ which contains $V(f) \cap W(T)$ and is contained in $V(f) \cap \overline{W(T)}$. Since $V(f) \cap W(T)$ and $V(f) \cap \overline{W(T)}$ are topologically very close, this approximation is sharp. Moreover, it turns out that when solving a system of polynomial equations, say

$$F = \left\{ \begin{array}{ccc} f_1 & = & 0 \\ f_2 & = & 0 \\ \vdots & & \vdots \\ f_m & = & 0 \end{array} \right. ,$$

by repeated applications of the procedure `Intersect` with $f_1, f_2 \ldots, f_m$ produces regular chains $\{T_1, \ldots, T_v\} \subset \mathbf{k}[x_1, \ldots, x_n]$ such that:

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

Using the polynomial system in Equation (1.1):

$$\left\{ \begin{array}{ccl} f_1 & = & {x_2}^2 + {x_1}^2 - 2\,x_2 - 2\,x_1 - 2 \\ f_2 & = & x_2\,x_1 - x_2 - x_1 \end{array} \right. ,$$

with the variable ordering $x_1 < x_2$, we shall illustrate how the incremental solving process of [27] works. This process starts by taking $\{\emptyset\}$ as the initial triangular decomposition. The empty set is indeed a regular chain, the solution set of which is simply the entire ambient space, since $\emptyset$ has no constraints. The first call `Intersect`(f, T) is then with $f = f_1$ and $T = \emptyset$ which simply returns the single regular chain $\{f_1\}$. Next, the second call `Intersect`(f, T) is then with $f = f_2$ and $T = \{f_1\}$. Since f_1 and f_2 share the variables x_1 and x_2, the set $\{f_1, f_2\}$ is not a triangular set. A natural question is to determine the values v of x_1 so that when x_1 is replaced by v in both f_1 and f_2 then these two specialized polynomials have at least one common solution in x_2. On this particular example, one can determine those values of x_1 by a simple series of substitutions. Indeed, from $f_2 = 0$, we have:

$$(x_1 - 1)(x_2 - 1) = 1.$$

Hence, assuming $x_1 \neq 1$, we have $x_2 = \frac{1}{x_1 - 1} + 1$. Replacing x_2 with $\frac{1}{x_1 - 1} + 1$ into $f_1 = 0$ leads to ${x_1}^4 - 4\,{x_1}^3 + 2\,{x_1}^2 + 4\,x_1 - 2 = 0$. This latter equation has four distinct real solutions:

$$\left\{1 - {}^{\sqrt{3}\sqrt{2}}/_{2} + {}^{\sqrt{2}}/_{2}, 1 + {}^{\sqrt{3}\sqrt{2}}/_{2} - {}^{\sqrt{2}}/_{2}, 1 - {}^{\sqrt{3}\sqrt{2}}/_{2} - {}^{\sqrt{2}}/_{2}, 1 + {}^{\sqrt{3}\sqrt{2}}/_{2} + {}^{\sqrt{2}}/_{2}\right\},$$

that one could anticipate from Figure 1.1, where the two curves have four distinct common points. Substituting these 4 values of x_1 into $f_2 = 0$ yields the corresponding values of x_2:

$$\left\{{}^{\left(\sqrt{3}\sqrt{2}-\sqrt{2}-2\right)\sqrt{2}}/_{2\sqrt{3}-2}, {}^{\left(\sqrt{3}\sqrt{2}-\sqrt{2}+2\right)\sqrt{2}}/_{2\sqrt{3}-2}, {}^{\left(\sqrt{3}\sqrt{2}+\sqrt{2}-2\right)\sqrt{2}}/_{2+2\sqrt{3}}, {}^{\left(\sqrt{3}\sqrt{2}+\sqrt{2}+2\right)\sqrt{2}}/_{2+2\sqrt{3}}\right\}.$$

It remains to check whether $f_1 = f_2 = 0$ has a solution for $x_1 = 1$. Elementary calculations show that this is not the case, as suggested by Figure 1.1.

1.4 Subresultant Chains

While for the polynomial system in Equation (1.1), elementary calculations are sufficient, this is not the case for most polynomial systems that arise in practice or in theory. We illustrate that fact through a series of examples that also informally introduces the concept of a subresultant chain, which is at the core of our work.

1.4.1 The First Example

To understand the need for algebraic methods handling more complex polynomial systems, let us consider the following problem: computing the common roots of two (depressed) generic cubic equations:

$$\left\{\begin{array}{rcl} x^3 + ax + b & = & 0 \\ x^3 + cx + d & = & 0 \end{array}\right. . \tag{1.3}$$

In the above polynomial system, x is the unknown while a, b, c, d are parameters. Because a univariate algebraic equation of degree n has n complex solutions (counted with multiplicities) the above two equations are expected to have either 0, 1, 2 or 3 common roots, depending on the values of the parameters. Hence, we would like to obtain a case-discussion yielding conditions on a, b, c, d for the above two equations to have either 0, 1, 2 or 3 common roots. As counter-intuitive as it may sound, the case of exactly 2 common roots (counted with multiplicities) is impossible. But at this stage, we will pretend that we are not aware of that fact. And, in fact, we shall rediscover that fact with the algebraic method that we are going to present.

If a, b, c, d were actual numbers, then one could apply the well-known Euclidean algorithm. Naming r_3 and r_2 the polynomials $x^3 + ax + b$ and $x^3 + cx + d$, we would compute the remainder r_1 in the Euclidean division of r_3 by r_2, that is,

$$r_1 = (a - c)x + b - d. \tag{1.4}$$

Since r_1 has degree 1 in x (and not 2) we deduce that it is impossible for r_3 and r_2 to have exactly 2 common roots (counted with multiplicities). Indeed, if that case was possible, the Greatest Common Divisor (GCD) of r_3 and r_2 should have degree 2 in that particular case. Next, we observe that we need to distinguish two cases: $a = c$ and $a \neq c$.

In the case $a = c$, two sub-cases arise: $b = d$ in which case r_3 and r_2 are identical thus having 3 common roots or, $b \neq d$ in which case r_3 and r_2 have no common root (since the GCD is a non-zero constant).

In the case $a \neq c$, continuing with the Euclidean Algorithm, we compute the remainder r_0 in the Euclidean division of r_2 by r_1, that is,

$$r_0 = \frac{a^3d - a^2bc - 2\,a^2cd + 2\,abc^2 + ac^2d - bc^3 - b^3 + 3\,b^2d - 3\,bd^2 + d^3}{(a-c)^3}. \tag{1.5}$$

If the values of a, b, c, d are such that r_0 is zero then r_3 and r_2 have exactly one common root given by $r_1 = 0$, that is, $x = \frac{d-b}{a-c}$. Otherwise, if r_0 is not zero, then r_3 and r_2 have no common root (since the GCD is a non-zero constant).

To summarize what we have done with System (1.3), we have observed that we could solve our problem by regarding a, b, c, d as actual numbers and applying the Euclidean Algorithm. We could follow the same strategy with more complex problems, say, computing the common roots of two generic equations of degree 4, 5, etc. Various computational issues would then arise. First, the size of the coefficients of the successive remainders (computed by the Euclidean Algorithm) would grow very rapidly. Second, the number of cases to be considered would also grow rapidly.

It turns out that there exists an algebraic theory, the *theory of subresultants*, which deals with these two computational issues. This theory will be presented formally in

Section 2.2. In the mean time, we shall just introduce it informally in the sequel of this introductory chapter.

1.4.2 Theory of Subresultants (informally)

Consider two multivariate polynomials f and g, which are univariate in x and with coefficients that are polynomials in other variables $a, b, c, \ldots$. The Euclidean Algorithm produces a sequence of polynomials $r_0, r_{i_1}, r_{i_2}, \ldots, r_{i_{s-1}}, r_{i_s}$ (univariate in x and with coefficients that are rational functions in the other variables) where:

i) $r_{i_{s-1}}, r_{i_s}$ are f and g respectively, assuming $\deg(f, x) \leq \deg(g, x)$,

ii) the polynomial r_{i_j} is the remainder in the Euclidean division of $r_{i_{j+1}}$ by $r_{i_{j+2}}$, for $0 \leq j < s - 1$,

iii) the polynomial r_{i_j} has degree i_j in x, for $0 \leq j < s - 1$.

The theory of subresultants produces a sequence of polynomials $S_0, S_1, S_2, \ldots$ (univariate in x and with coefficients that are polynomials in the other variables) and called *subresultants*, such that:

1. the polynomial S_k is either null or has a degree in x less or equal to k,
2. if S_k and S_{k+1} are non-zero, then we have $\deg(S_k, x) \leq \deg(S_{k+1}, x)$,
3. each non-zero polynomial r_{i_j} (in the sequence of the Euclidean algorithm) is proportional to a non-zero polynomial S_k.

The above properties imply that the sequence $S_0, S_1, S_2, \ldots$ contains the same information as the sequence $r_0, r_{i_1}, r_{i_2}, \ldots, r_{i_{s-1}}, r_{i_s}$. On the example of System (1.3), the sequence computed by the theory of subresultants is:

$$\operatorname{numer}(r_0), r_1, 0, r_2, r_3. \tag{1.6}$$

where $\operatorname{numer}(r_0)$ denotes the numerator of r_0.

To illustrate the fact that subresultants have smaller coefficients than their remainder counterparts (in the Euclidean sequence), consider the polynomials $r_5 = x^5 + x^4 + x^3 + x^2 + 1$ and $r_4 = 5x^4 + x^3 + 1$. The remainders r_0, r_1, r_2, r_3 of the Euclidean sequence and their subresultant counterparts S_0, S_1, S_2, S_3 are shown below:

$$\begin{array}{rcl@{\qquad}rcl}
r_3 & = & {}^{21}\!/_{25}\, x^3 + x^2 - {}^{1}\!/_{5}\, x + {}^{21}\!/_{25}, & S_3 & = & 21x^3 + 25x^2 - 5x + 21, \\
r_2 & = & {}^{3125}\!/_{441}\, x^2 - {}^{2725}\!/_{441}\, x + {}^{125}\!/_{21}, & S_2 & = & 125x^2 - 109x + 105, \\
r_1 & = & {}^{236376}\!/_{390625}\, x - {}^{48069}\!/_{78125}, & S_1 & = & 536x - 545, \\
r_0 & = & {}^{886328125}\!/_{126697536}. & S_0 & = & 2269.
\end{array}$$

1.4.3 The Second Example

We use another example to illustrate the ability of subresultants to encode a case discussion. Consider the following polynomials f and g in the variables x and y:

$$\begin{array}{rcl} f & = & x^3 + y^2 + x + 1, \\ g & = & y^3 + x^2 + y + 1. \end{array} \tag{1.7}$$

Viewing f and g as polynomials in x, with coefficients that are polynomials in y, their subresultants S_0 and S_1 are:

$$\begin{array}{rcl} S_1 & = & (y^2+1)\,(yx-1)\,, \\ S_0 & = & (y+1)\,(y^2-y+1)\,(y^2+1)^3\,. \end{array}$$

As we shall see in Sections 2.1 and 2.2, the equation $S_0 = 0$ is a necessary condition for the system $f = g = 0$ to have solutions. Moreover, if the leading coefficients of f and g (regarded as polynomials in x) are constant (that is, do not depend on y) then $S_0 = 0$ is a necessary and sufficient condition for the system $f = g = 0$ to have solutions. This latter property applies here, which means that $f = g = 0$ has solutions if and only if one of the three following conditions hold:

i) $y + 1 = 0$,
ii) $y^2 - y + 1 = 0$, or
iii) $y^2 + 1 = 0$.

In the case $y + 1 = 0$, the subresultant S_1 becomes $-2(x + 1)$ and the theory of subresultants implies that $x + 1$ is the GCD of f and g, and thus $x = -1$ must hold. In the case $y^2 - y + 1 = 0$, the subresultant S_1 becomes $yx - 1$ and the theory of subresultants implies that $yx - 1$ is the GCD of f and g, thus $x = 1/y$ must hold.

In the case $y^2 + 1 = 0$, the subresultant S_1 is identically zero, while the polynomials f and g become respectively $x^3 + x$ and $x^2 + 1$; clearly g is the GCD of f and g, thus $x^2 + 1 = 0$ must hold. To summarize what we have done with this last example, we saw that the subresultants S_0 and S_1, together with the input polynomials f and g contain all the information to generate all cases when solving $f = g = 0$. The procedure `Intersect` discussed above in Section 1.3 uses that idea intensively.

1.4.4 The Third Example

We use this last example to illustrate an important idea in this thesis: in some examples, the whole sequence of subresultants may not be needed when solving a bivariate polynomial system $f = g = 0$. Consider the following polynomials f and g in the variables x and y:

$$\begin{aligned} f &= x^7 - 36x - 22y + 1, \\ g &= x^6 + 47x^3 - 60xy^2 - 6xy - 83y^2 - 10y + 50. \end{aligned} \tag{1.8}$$

Viewing f and g as polynomials in x with coefficients that are polynomials in y, their subresultants $S_4, \ldots, S_0$ from the list of all subresultants $\{S_4, \ldots, S_0\}$ are:

$$\begin{aligned} S_4 &= 46x^4 + 64x^2y^2 + 27x^2y + 13xy^2 + 45xy + 25x + 4y + 56, \\ S_3 &= 74x^2y^4 + 7x^3y^2 + 56x^2y^3 + 44xy^4 + 48x^3y + 44x^2y^2 + 3xy^3 + 14x^3 + 18x^2y \\ &\quad +41xy^2 + 69y^3 + 46x^2 + 62xy + 98y^2 + 86y + 53, \\ S_2 &= 25x^2y^8 + 10x^2y^7 + 26xy^8 + 62x^2y^6 + 36xy^7 + 32x^2y^5 + 99xy^6 + 8y^7 + 32x^2y^4 \\ &\quad +82xy^5 + 67y^6 + 6x^2y^3 + 53xy^4 + 101y^5 + 39x^2y^2 + 10xy^3 + 36y^4 + 87x^2y \\ &\quad +21xy^2 + 28y^3 + 46x^2 + 73xy + 69y^2 + 96x + 72y + 43, \\ S_1 &= 81xy^{12} + 28xy^{11} + 76y^{12} + 24xy^{10} + 5xy^9 + 26y^{10} + 18xy^8 + 28y^9 + 26xy^7 \\ &\quad +62y^8 + 88xy^6 + 87y^7 + 102xy^5 + 20y^6 + 72xy^4 + 2y^5 + 10xy^3 + 67y^4 + 66xy^2 \\ &\quad +75y^3 + 87xy + 8y^2 + 4x + 73y + 77, \\ S_0 &= 97y^{15} + 82y^{14} + 82y^{13} + 99y^{12} + 20y^{11} + 53y^{10} + 52y^9 + 20y^8 + 65y^7 + 83y^6 \\ &\quad +23y^5 + 89y^4 + 31y^3 + y^2 + 54y + 69. \end{aligned}$$

Meanwhile, the solutions of $f = g = 0$ are $(t_{0,0} = 0, t_{0,1} = 0)$ and $(t_{1,0} = 0, t_{1,1} = 0)$ where,

$$\begin{aligned} t_{0,0} &= (y^{10} + 67y^9 + 83y^8 + 23y^7 + 43y^6 + 45y^5 + 12y^4 + 33y^3 + 25y^2 + 33y + 100)x \\ &\quad +63y^{10} + 31y^9 + 36y^8 + 92y^7 + 86y^6 + 49y^5 + 79y^4 + 29y^3 + 52y^2 + 7y + 3, \\ t_{0,1} &= y^{11} + 64y^{10} + 36y^9 + 91y^8 + 82y^7 + 98y^6 + 30y^5 + 54y^4 + 56y^3 + 52y^2 \\ &\quad +73y + 86, \\ t_{1,0} &= (y^3 + 53y^2 + 100y + 68)x + 53y^3 + 12y^2 + 6y + 8, \\ t_{1,1} &= y^4 + 94y^3 + 80y^2 + 92y + 34. \end{aligned}$$

This set of polynomial equations, which are in the form of triangular sets, describe solutions of $f = g = 0$, and can be calculated from only the information in S_0 and S_1, and does not require the entire subresultant chain $\{S_4, \ldots, S_0\}$.

1.4.5 Subresultants in Theory and Practice

Subresultants were introduced by Sylvester in 1840 as determinants of submatrices of the *Sylvester Matrix* and later in [21]. The first algorithm to compute the entire subresultant chain did so by calling determinant procedures within a cubic cost. The first quadratic and recursive algorithm was introduced by Habicht [44] by taking advantage of a pseudo-division algorithm.

Collins in [29] showed the importance of coefficient swell by indicating lower bounds for the coefficient growth in the Euclidean algorithm. Ducos [36], Lickteig-Roy [68], and Lombardi-Roy-Safey El Din [70] are some of the most notable and pioneer researches that led to present techniques to compute subresultant chains with addressing both the coefficient swelling problem and the algorithmic complexity.

The `RegularChains` library in Maple is a collection of commands for solving systems of algebraic equations, inequations and inequalities symbolically follows the theory of regular chains [64]. The main objective of this thesis is algorithmic contributions, in particular, to high-performance computational schemes for subresultant chains and underlying routines, which extends that of the `RegularChains` library. Our contributions include designing and developing a multithreaded symbolic and incremental polynomial system solver based on the theory of regular chains.

Our code is publicly available at `www.bpaslib.org` as part of the Basic Polynomial Algebra Subprograms (BPAS) library [8] that is mainly written in C, with concurrency support and user interfaces written in C++. Most notably, This library provides univariate, bivariate, and multivariate polynomial arithmetic including a family of subresultant schemes that have been integrated, tested, and utilized in the multithreaded BPAS polynomial system solver.

1.5 Objectives and Contributions

In this thesis, we are particularly interested in algorithms for computing subresultants, in support of polynomial system solving by means on the `Triangularize` algorithm. We will proceed by the following three steps:

1. Designing *speculative strategies* that takes into account how subresultant chains are used, most of the time, by the `Intersect` core-routine of the `Triangularize` algorithm and to compute the so-called *Regular* GCDs. Indeed, as illustrated by the example in Section 1.4.4, when solving a bivariate system $f(x, y) = g(x, y) = 0$ only the subresultants S_0 and S_1 of the subresultant chain $S_0, S_1, S_2, S_3, \ldots$ may be needed. It is known that this pattern is the common case, while not the general one. Therefore, it is desirable to be able to,

   - compute S_0 and S_1 directly without computing $S_2, S_3, \ldots$, and
   - compute $S_2, S_3, \ldots$, if they happen to be needed recycling the *cached* data generated by the intermediate computations that have led to S_0 and S_1 in the first place.

   This speculative approach is a new technique, and for univariate and bivariate polynomials, we will rely on an existing technique, the so-called *Half-GCD algorithm* to achieve our goal; that is discussed in Section 3.4. We further design a speculative subresultant algorithm in order to compute subresultants via calculating the determinant of Bézout matrices; that is described in Section 5.3.

   The notion of speculative algorithm is inspired by the definition of *speculative execution* on computer hardware. We mean an algorithm that would normally compute a sequence S of items but assumes that only a prescribed sub-sequence S' of those may be needed. It manages to increase performance by taking this observation into account, while being able to recover S from the calculations of S' if the whole S turns out to be needed. Thus, the cost of that recovery along with computing S' is essentially that of computing S directly.

2. Deploying *faster algorithms* for computing subresultant chains, in particular for the fundamental cases of univariate and bivariate polynomials. Indeed, even when solving polynomial systems in more than 2 variables, the recursive nature of the `Intersect` core-routine makes the case of univariate and bivariate polynomials essential.

   Faster algorithms refer to core-routines based on asymptotically fast techniques for polynomials' basic arithmetic, e.g. multiplication, division, evaluation and interpolation; we review these algorithms and their complexities in Section 3.3. These ideas are not new but they remain to be verified experimentally that such methods can have a significant impact on the range of subresultant chain computations involved in polynomial system solving, in practice.

3. Optimizing *implementation techniques* for computing subresultant chains can be listed as,

   - In the case of algorithms based on evaluation-interpolation schemes, it is natural to consider a multithreaded implementation taking advantage of the BPAS interface for multithreading (Section 2.3). Yet again, it remains to measure the benefits of those ideas in practice and as part of the system solver; that is discussed in Section 3.5.
   - The use of subresultants by the `Intersect` algorithm (Section 2.1) relies on the so-called *specialization property of subresultants* (Section 2.2). One can reduce the use of this property taking advantage of normal form algorithms to reduce the subresultants w.r.t. the regular chains computed so far by the `Triangularize` algorithm (Section 7.1). We study optimizations of these routines for triangular sets as divisors in Section 4.3.
   - As for algorithms computing subresultant chains based on schemes that are not suitable for parallelization, it is desirable to minimize costs attributed to memory access patterns. This direction has not been explored in the literature. We introduce a cache-friendly version of the Ducos' subresultant chain algorithm in Section 4.4.
   - For matrices over multivariate polynomials, we make use of the Bareiss fraction-free LU decomposition and further optimize this algorithm with a so-called *smart-pivoting* technique to reduce the cost of the decomposition, and so the Bézout subresultant algorithms in Section 5.2 and Section 5.3, respectively.
   - For subresultant algorithms that rely on the computation of the matrix *determinant* via Bareiss FFLU algorithm, one can parallelize the *row-reduction* part of this algorithm. Yet again, tuning this algorithm to maximize the parallel-speed-up and working properly with the rest of the software are among implementation challenges that we address in Section 5.2.

Chapter 2

Primary Background

This chapter is intended to serve as a review of mathematical concepts and parallel patterns to be used throughout this thesis. Section 2.1 is a brief review on the triangular decomposition methods and the `Triangularize` algorithm. We discuss the theory of subresultant chains in Section 2.2. In Section 2.3, we review the parallel patterns supported by the BPAS multithreaded interface and discuss the *map* pattern and the implementation of the `parallel_for` loop.

2.1 `Triangularize` Algorithm

Let F be a finite set of polynomial equations. An informal definition of the `Triangularize` algorithm is in fact a polynomial system solver approach to decompose F incrementally, so that one equation is solved after another against each component produced by the previous iteration. This algorithm returns a set of components representing solutions of F in a particular form known as *regular chains* using a triangular decomposition method defined in the following.

Let $\mathbf{k}$ be a field and its closure be $\mathbf{K}$ and $\mathbf{k}[X] = \mathbf{k}[x_1, \ldots, x_v]$ be the ring of polynomials with variable ordering $x_1 < \cdots < x_v$ for $v \in \mathbb{N}$.

Definition 1 *A non-empty set $T = (T_1, \ldots, T_e) \subset \mathbf{k}[X]$ is a triangular set if T is a set of non-constant polynomials with distinct main variables.*

Definition 2 *Let $T = (T_1, \ldots, T_e) \subset \mathbf{k}[X]$ be a triangular set such that:*

$$\mathrm{mvar}(T_1) < \cdots < \mathrm{mvar}(T_e),$$

the set T is called a regular chain if,

$$\operatorname{res}(h,T) := \operatorname{res}(\cdots\operatorname{res}(\operatorname{res}(h,T_e),T_{e-1})\cdots) \neq 0,$$

where h is the product of $\operatorname{init}(T_i)$*, that is the leading coefficient of T_i w.r.t. its main variable, for $i = 1,\dots,e$ and* $\operatorname{res}(h,T_e)$ *is the resultant of h and T_e w.r.t. the main variable of T_e denoted by* $\operatorname{mvar}(T_e)$*. To compute* $\operatorname{res}(h,T)$*, each resultant is computed w.r.t. the main variable of T_i for $1 \le i \le e$.*

The properties and applications of regular chains have been studied for several decades; see [15, 66] for more details. In the following, we review the main specifications required to define the `Triangularize` algorithm and one of its main subroutines known as *Regular GCD* that the `Intersect` algorithm relies on.

Definition 3 *The zero-set or algebraic variety of a non-empty set $T \subset \mathbf{k}[X]$ is denoted by $V(T) \subset \mathbf{K}^v$ and is defined as:*

$$V(T) := \{t \in \mathbf{K}^v \mid \forall a \in T \;\, a(t) = 0\}.$$

Moreover, a refinement of this definition for $T = (T_1,\dots,T_e) \subset \mathbf{k}[X]$ that is so-called the quasi-component *of T is denoted by $W(T)$ and is defined as:*

$$W(T) := V(T) \backslash V(h),$$

where h is the product of $\operatorname{init}(T_i)$ *for $i = 1,\dots,e$.*

Definition 4 *Let $T \subset \mathbf{k}[X]$ be a non-empty set. The saturated ideal of T is denoted by* $\operatorname{sat}(T)$ *and defined as:*

$$\langle T\rangle : h^\infty := \{a \in \mathbf{k}[X] \mid \exists s \in \mathbb{N} \; s.t. \; h^s a \in \langle T\rangle\}.$$

From Definitions 3 and 4, we have the following equality that indicates the relation between the quasi-component of a regular chain and its saturated ideal:

$$\overline{W(T)} = V(\operatorname{sat}(T)),$$

that is also known as the *Zariski closure* of $W(T)$ [27].

Let F be a system of polynomials. From [13, 27], there are two well-studied triangular decomposition methods computable by the `Triangularize` algorithm:

i) The *Kalkbrener* method, that is decomposing the generic points of F to compute,

$$V(F) = \cup_{i=1}^{e} \overline{W(T_i)};\ \text{and}$$

ii) The *Lazard* method, that is decomposing all zeros of F to compute,

$$V(F) = \cup_{i=1}^{e} W(T_i).$$

Both methods are developed in the BPAS library [10, 13]; however the comprehensive results in [10] shows that the *Kalkbrener* method generally has a better performance. Thus, we consider this method to experiment computational schemes for subresultants in the BPAS solver throughout this thesis.

The `Triangularize` algorithm uses a variety of subroutines; see [27] for the list of all subroutines. One of the underlying procedures that plays an important role in the performance of the `Triangularize` algorithm is *Regular GCD*. Let $1 \leq i \leq v$ be an integer, $T \subset \mathbf{k}[X]$ be a regular chain, $a, b \in \mathbf{k}[X]$ be non-constant polynomials with the same main variable x_i, $g \in \mathbf{k}[X]$ be either constant or $\text{mvar}(g) \leq x_i$, and $\mathfrak{L} = \mathbf{k}[x_1, \ldots, x_{i-1}] / \sqrt{\text{sat}(T)}$.

Definition 5 *Assume that $x_i > x_j$ for all x_j in the set of main variables of T denoted by* mvar(T)*, and both* init(a), init(b) *are neither zero, nor a* zero-divisor[1] *(equivalently, they are* regular*) modulo* sat(T)*. The* regular GCD *of a, b modulo T is g, if the following conditions hold:*

i) $\text{lc}(g, x_i)$ *is a regular element of* $\mathfrak{L}$*;*

ii) $g \in \langle a, b \rangle$ *in* $\mathfrak{L}[x_i]$*; and*

iii) if $\deg(g, x_i) > 0$*, then* $\text{prem}(a, g, x_i) = \text{prem}(b, g, x_i) = 0$*, where* $\text{prem}(a, g, x_i)$ *and* $\text{prem}(b, g, x_i)$ *are, respectively, pseudo-remainders of a and b by g w.r.t. x_i in* $\mathfrak{L}[x_i]$.

For $0 \leq k \leq \text{mdeg}(b)$, S_k denotes the k-th subresultant of a, b in $\mathfrak{L}[x_i]$ and assume there exists $1 \leq d \leq \text{mdeg}(b)$ so that $S_j \in \text{sat}(T)$ for all $0 \leq j < d$. Then, S_d is the last non-zero and non-defective subresultant in $\mathfrak{L}$, and if $\text{lc}(S_d, x_i)$ is not regular modulo $\text{sat}(T)$ then S_d may be defective in $\mathfrak{L}$. In addition, S_d will vanish on all the components of $\text{sat}(T)$ up to sufficient splitting of $\text{sat}(T)$.

[1]A polynomial $a \in \mathbf{k}[X]$ is a zero-divisor modulo $\langle F \rangle$ if there exists a polynomial $b \in \mathbf{k}[X]$ such that $ab \in \langle F \rangle$ and $a, b \notin \langle F \rangle$.

Theorem 1 *Assume* $\mathrm{lc}(S_d, x_i)$ *is regular modulo* $\mathrm{sat}(T)$. *If* $\mathrm{sat}(T)$ *is radical or for all* $0 \leq k < d$, $\mathrm{coeff}(S_k, x_i, k)$ *is either* 0 *or regular modulo* $\mathrm{sat}(T)$. *Then,* S_d *is a regular GCD of* a, b *modulo* $\mathrm{sat}(T)$.

Proof. [27, Theorem 6]

Theorem 1 shows the way that `Triangularize` uses subresultants from a subresultant chain of input polynomials as well as the importance of subresultants in computing a regular GCD modulo $\mathrm{sat}(T)$.

In fact, one can compute the regular GCD of a, b modulo $\mathrm{sat}(T)$ by performing a bottom-up search in the subresultant chain. Besides, there are often only a few subresultants required, starting from S_0 and S_1, in practice.

2.2 Theory of Subresultant Chain

As seen in the previous section, the `Triangularize` algorithm to solve (or decompose) a system makes use implicitly or explicitly of a notion of GCD for univariate polynomials over coefficient rings that are not necessarily fields. For example, a polynomial in $\mathbb{Z}[x_1, \ldots, x_v][y]$ is in fact a univariate polynomial over ring of the multivariate polynomials.

A formal definition of these GCD variant is in fact the *Regular GCD* in Definition 5 that applies to residue class rings of the form $\mathbf{k}[X]/\mathrm{sat}(T)$ for a regular chain T. This weaker notion of a polynomial GCD is actually sufficient for solving polynomials systems via `Triangularize` when retaining the multiplicities of zeros is not required during the decomposition [27].

The Regular GCD algorithm is a simple procedure due to using a powerful algebraic theory namely *subresultants*. Here, we review a formal definition of subresultants over $\mathbb{B}$, a general commutative ring with identity, and study important specifications of subresultants that the Regular GCD, and so, the entire solver rely on. In this review, we follow the presentations in [36, 52].

Determinantal Polynomial

Definition 6 *Let* $\mathbb{B}$ *be a commutative ring with identity and let* $m \leq n$ *be positive integers. Let* M *be a* $m \times n$ *matrix with coefficients in* $\mathbb{B}$. *Let* M_i *be the square submatrix of* M *consisting of the first* $m-1$ *columns of* M *and the* i*-th column of* M, *for* $m \leq i \leq n$;

let $\det(M_i)$ *be the determinant of* M_i. *The determinantal polynomial of* M *denoted by* $\operatorname{dpol}(M)$ *is a polynomial in* $\mathbb{B}[y]$, *given by:*

$$\operatorname{dpol}(M) := \det(M_m)y^{n-m} + \det(M_{m+1})y^{n-m-1} + \cdots + \det(M_n).$$

Note that, if $\operatorname{dpol}(M)$ is not zero, then its degree is at most $n - m$.

Example 1 *Consider* $n = 4$, $m = 2$. *For polynomials* $a = a_3y^3 + a_2y^2 + a_1y + a_0$ *and* $b = b_2y^2 + b_1y + b_0$ *in* $\mathbb{B}[y]$. *We have:*

$$\operatorname{mat}(a, b) = \begin{bmatrix} a_3 & a_2 & a_1 & a_0 \\ 0 & b_2 & b_1 & b_0 \end{bmatrix},$$

with,

$$M_2 = \begin{bmatrix} a_3 & a_2 \\ 0 & b_2 \end{bmatrix}, M_3 = \begin{bmatrix} a_3 & a_1 \\ 0 & b_1 \end{bmatrix}, \text{ and } M_4 = \begin{bmatrix} a_3 & a_0 \\ 0 & b_0 \end{bmatrix}.$$

and consequently $\operatorname{dpol}(a, b) = a_3b_2y^2 + a_3b_1y + a_3b_0$.

Let $f_1, \ldots, f_m$ be polynomials of $\mathbb{B}[y]$ of degree less than n. We denote by $\operatorname{mat}(f_1, \ldots, f_m)$, the $m \times n$ matrix whose i-th row contains the coefficients of f_i, sorted in order of decreasing degree, and such that f_i is treated as a polynomial of degree $n - 1$. We denote by $\operatorname{dpol}(f_1, \ldots, f_m)$, the determinantal polynomial of $\operatorname{mat}(f_1, \ldots, f_m)$.

Definition 7 *Let* $a, b \in \mathbb{B}[y]$ *be non-constant polynomials of respective degrees* $m := \deg(a)$, $n := \deg(b)$ *with* $m \geq n$. *Let* k *be an integer with* $0 \leq k < n$. *Then, the* k*-th* subresultant *of* a *and* b *(also known as the* subresultant of index k *of* a *and* b*), denoted by* $S_k(a, b)$, *is:*

$$S_k(a, b) = \operatorname{dpol}(y^{n-k-1}a, y^{n-k-2}a, \ldots, a, y^{m-k-1}b, \ldots, b).$$

This is a polynomial which belongs to the ideal generated by a and b in $\mathbb{B}[y]$. In particular, $S_0(a, b)$ is the resultant of a and b denoted by $\operatorname{res}(a, b)$. Observe that if $S_k(a, b)$ is not zero then its degree is at most k. If $S_k(a, b)$ has degree k, then $S_k(a, b)$ is said to be *non-defective* or *regular*; if $S_k(a, b) \neq 0$ and $\deg(S_k(a, b)) < k$, then $S_k(a, b)$ is said to be *defective*.

We call k-th *nominal leading coefficient*, denoted by s_k, the coefficient of $S_k(a,b)$ in y^k. Observe that if $S_k(a,b)$ is defective, then we have $s_k = 0$. For convenience, we extend the definition to the n-th subresultant as follows:

$$S_n(a,b) = \begin{cases} \gamma(b)b, & \text{if } m > n \text{ or } \mathrm{lc}(b) \in \mathbb{B} \text{ is regular} \\ \text{undefined}, & \textit{otherwise} \end{cases}.$$

where $\gamma(b) = \mathrm{lc}(b)^{m-n-1}$. In the above, *regular* means *not a zero-divisor*. Note that when m equals n and $\mathrm{lc}(b)$ is a regular element in $\mathbb{B}$, then $S_n(a,b) = \mathrm{lc}(b)^{-1}b$ is in fact a polynomial over the total fraction ring of $\mathbb{B}$.

Theorem 2 *Assume $\mathbb{B}$ be a UFD and a, b are non-constant polynomials in $\mathbb{B}[y]$ with* $\deg(a) = m, \deg(b) = n$. *If for some $0 < k \leq \min(m,n)$, we have $S_k(a,b) \neq 0$ and $S_i(a,b) = 0$ for all $i < k$, then exist $\alpha, \beta \in \mathbb{B}$ such that:*

$$\alpha \gcd(a,b) = \beta\, S_k(a,b,y).$$

Proof. [27, Theorem 5]

According to the above proposition, S_k is a regular subresultant that is so-called the *last nonzero subresultant* of a, b.

Example 2 *Let two polynomials $a = y^3 - y^2$ and $b = y^2 - 3y$ in $\mathbb{Z}[y]$ with* $\gcd(a,b) = y$. *Then:*

$$S_0(a,b) = \mathrm{dpol}(y^2a, ya, a, yb, b) = \mathrm{dpol}(\begin{pmatrix} 1 & -3 & 0 & & \\ & 1 & -3 & 0 & \\ & & 1 & -3 & 1 \\ 1 & -1 & 0 & 0 & \\ & 1 & -1 & 0 & 0 \end{pmatrix}) = 0,$$

and,

$$S_1(a,b) = \mathrm{dpol}(ya, a, b) = \mathrm{dpol}(\begin{pmatrix} 1 & -3 & 0 & \\ & 1 & -3 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix}) = 6y.$$

We call *specialization property of subresultants* the following property. Let $\mathbb{A}$ be another commutative ring with identity and Ψ a ring homomorphism from $\mathbb{B}$ to $\mathbb{A}$ such that we have $\Psi(\mathrm{lc}(a)) \neq 0$ and $\Psi(\mathrm{lc}(b)) \neq 0$. Then, for $0 \leq k \leq n$, we have:

$$S_k(\Psi(a), \Psi(b)) = \Psi(S_k(a,b)).$$

Divisibility Relations of Subresultants

From now on, we assume that the ring $\mathbb{B}$ is an integral domain. The *subresultant chain* of a and b, defined as:

$$\mathrm{subres}(a, b) := (S_n(a, b), S_{n-1}(a, b), S_{n-2}(a, b), \dots, S_0(a, b)),$$

satisfies relations which induce a Euclidean-like algorithm for computing the entire subresultant chain: $\mathrm{subres}(a, b)$. This algorithm runs within $O(n^2)$ operations in $\mathbb{B}$, when $m = n$, see [36]. For convenience, we simply write S_k instead of $S_k(a, b)$ for each k. We write $a \sim b$, for $a, b \in \mathbb{B}[y]$, whenever a, b are associate elements in $\mathrm{frac}(\mathbb{B})[y]$, the field of fractions of $\mathbb{B}$. Then for $1 \leq k < n$, we have:

(i) $S_{n-1} = \mathrm{prem}(a, -b)$,

(ii) If S_{n-1} is non-zero, defining $e := \deg(S_{n-1})$, then we have:

$$S_{e-1} = \frac{\mathrm{prem}(b, -S_{n-1})}{\mathrm{lc}(b)^{(m-n)(n-e)+1}},$$

(iii) If $S_{k-1} \neq 0$, defining $e := \deg(S_{k-1})$ and assuming $e < k - 1$ (thus assuming S_{k-1} defective), then we have:

   (a) $\deg(S_k) = k$, thus S_k is non-defective,

   (b) $S_{k-1} \sim S_e$ and $\mathrm{lc}(S_{k-1})^{k-e-1} S_{k-1} = {s_k}^{k-e-1} S_e$, thus S_e is non-defective,

   (c) $S_{k-2} = S_{k-3} = \cdots = S_{e+1} = 0$, and

(iv) If both S_k and S_{k-1} are non-zero, with respective degrees k and e then we have:

$$S_{e-1} = \frac{\mathrm{prem}(S_k, -S_{k-1})}{\mathrm{lc}(S_k)^{k-e+1}}.$$

Algorithm 1 from [36] is a known version of this procedure that computes all non-zero subresultants $a, b \in \mathbb{B}[y]$. Note that the core of this algorithm is the while-loop in which the computation of the subresultants S_e and S_{e-1}, with the notations of the above points (ii), (iii), and (iv) are carried out.

Algorithm 1 Subresultant (a, b, y)

```
Require: $a, b \in \mathbb{B}[y]$ with $m = \deg(a) \geq n = \deg(b)$ and $\mathbb{B}$ is an integral domain
Ensure: the non-zero subresultants from $(S_n, S_{n-1}, S_{n-2}, \ldots, S_0)$
 1: if $m > n$ then
 2:     $S := (\mathrm{lc}(b)^{m-n-1} b)$
 3: else $S := ()$
 4: $s := \mathrm{lc}(b)^{m-n}$
 5: $A := b$;  $B := \mathrm{prem}(a, -b)$
 6: while true do
 7:     $d := \deg(A)$;  $e := \deg(B)$
 8:     if $B = 0$ then return $S$
 9:     $S := (B) \cup S$; $\delta := d - e$
10:     if $\delta > 1$ then
11:         $C := \mathrm{lc}(B)^{\delta-1} B / s^{\delta-1}$
12:         $S := (C) \cup S$
13:     else $C := B$
14:     if $e = 0$ then return $S$
15:     $B := \mathrm{prem}(A, -B) / s^{\delta} \mathrm{lc}(A)$
16:     $A := C$; $s := \mathrm{lc}(A)$
17: end while
```

2.3 BPAS Multithreaded Interface

In this section, we review the parallel *map* pattern and `parallel_for` loop from the multithreading interface in the BPAS library that is derived from the standard *Thread Support Library* of C++11 and implemented by Brandt [10, 13].

The *Thread Support Library* of C++11 provides basic parallel primitives including *threads*, *mutual exclusion*, *condition variables*, and *futures*; see

`https://en.cppreference.com/w/cpp/thread`

for more information. The BPAS multithreaded interface utilizes this C++11 library and supplies a *generic* implementation for the known parallel patterns including:

(i) **Map** pattern, that maps a function to each item in a collection and simultaneously executing the function on each independent data item. [73, Chapter 4]

(ii) **Workpile** pattern, that is a generalized version of the *map* pattern to handle *irregular* tasks and an unknown number of tasks. In this pattern, the collection of data is in a *queue* or *pile*. [73, Section 3.6.4]

(iii) **Fork-Join** pattern, that refers to the pair of control flow known as *forked* (or *spawned*) and *join*. A *fork* corresponds to spawning a thread and giving it some code segment to execute and a *join* corresponds to when the spawned thread is joined with the spawning thread. [73, Chapter 8]

(iv) **Producer-Consumer** pattern refers to when a producer of data items is *synced* to a consumer of data items from a data collection like *queue*. In this pattern, both producer and consumer execute concurrently. [73, Section 3.5.2]

(v) **Asynchronous Generators** that is derived from the producer-consumer pattern, where a *generator* also known as an *iterator* is a function that *yields* data elements one at the time rather than many together as a collection. Concurrency helps within generators when the generation of data items is expensive and can be performed concurrently to the processing of previously yielded data items. [73, Chapter 9]

(vi) **Pipeline** pattern, a sequence of function executions, where the data flows from one to the next. This pattern can also be seen as a sequence of producer-consumer pairs, with intermediary functions acting as both a producer and a consumer. [73, Chapter 9]

The use of threads in the *thread-level parallelism* raises particular overheads. These so-called *parallel overheads* can be listed as follows,

(i) **Spawning** a thread is not necessarily a cheap operation and can become problematic if many threads are spawned throughout a program's lifetime;

(ii) **Over-subscription** that refers to the case when the program spawns more threads than are supported concurrently by the hardware. This yields increasing the cost of repetitive *context switching* significantly; and

(iii) **Inter-thread communication** and **synchronization**, in particular serializing access to some shared data, should be minimized and, where absolutely necessary, implemented efficiently.

The BPAS multithreaded interface uses a so-called *thread pool* to combat the cost of spawning and the penalties of over-subscription. In a way that the program spawns a small number of threads only once at the beginning and keeping them active throughout the program's lifetime to minimizing spawning overheads. Moreover, by limiting the number of threads spawned by a thread pool to be less than or equal to the number of threads supported by the current hardware, the interface handles the over-subscription overheads.

However, to address the issues of inter-thread communication and synchronization, the programmer must effectively take care of them via re-designing the algorithms or the underlying data-structures. The goal, here, is to limit inter-thread dependencies and to balance the amount of work that is known as *load-balancing* between threads to maximize parallelism.

In Sections 3.5 and 5.2.3, we utilize the *map* pattern and tackle the load-balancing issues in the multithreaded subresultant algorithms. Moreover, we tune these parallel routines to work efficiently with the thread pool and the rest of the multithreaded BPAS solver.

```
Algorithm 2 MapPattern($A$, $n$, $F$)
Require: an array $A$ of size $n$, and a function $F$
Ensure: an array $B$ of size $n$ where $B[i] = F(A[i])$
1: parallel_for $i$ from $0$ to $n-1$ do
2:     $B[i] := F(A[i])$
3: return $B$
```

An important application of the *map* pattern is the `parallel_for` loop. In a **for** loop with independent iterations, each thread simply executes one or more iterations of the loop concurrently. We denote this pattern as `parallel_for` throughout the thesis. This abstraction not only makes the use of *map* pattern implicit, but also hides the number of threads and the division of work evenly across a certain number of threads in pseudo-codes. Algorithm 2 shows an example of the `parallel_for` loop.

In procedures like Algorithm 11, when several maps are executed in a row, threads operate in *lockstep*. To explain, all operations in a previous map step must finish before the next map step may begin. The overall performance of a map step is thus limited by the slowest operation in the group. Hence, the map pattern works well with regular parallelism where individual data items or tasks are similar in size or execution cost, and the number of tasks to execute should be known *a priori*.

Chapter 3

Dense Univariate and Bivariate Polynomials

3.1 Introduction

In this chapter, we examine how several optimization techniques for subresultant chain computations and underlying core routines for univariate and bivariate polynomials benefit polynomial system solving in practice. These optimizations rely on ideas which have appeared in previous works, but without the support of successful experimental studies.

For univariate polynomials over a prime field $\mathbb{Z}_p$ where $\mathbb{Z}_p[y] := \mathbb{Z}/p\mathbb{Z}[y]$ and $p \in \mathbb{N}$ is an odd prime of size one machine-word (64-bit), we first develop the cutting-edge quasi-linear basic arithmetic operations for such polynomials with coefficients represented in Montgomery representation [80].

For computing the subresultants, the first of these optimizations takes advantage of the *Half-GCD* algorithm for computing GCDs of univariate polynomials over a field $\mathbf{k} = \mathbb{Z}_p$. For input polynomials of degree (at most) n, this algorithm runs within $O(\mathrm{M}(n) \log n)$ operations in $\mathbf{k}$, where $\mathrm{M}(n)$ is a polynomial multiplication time, as defined in [100, Chapter 8]. The *Half-GCD* algorithm originated in the ideas of Knuth [56], Lehmer [63] and Schönhage [90], while a robust implementation was a challenge for many years. One of the earliest correct designs was introduced in [96].

The idea of speeding up subresultant chain computations by means of the *Half-GCD* algorithm takes various forms in the literature. In [87], Reischert proposes a fraction-free adaptation of the *Half-GCD* algorithm, which can be executed over an effective integral domain $\mathbb{B}$, within $O(\mathrm{M}(n) \log n)$ operations in $\mathbb{B}$. We are not aware of any implementation of Reischert's algorithm.

In [69], Lickteig and Roy propose a "divide and conquer" algorithm for computing subresultant chains, the objective of which is to control coefficient growth in defective cases. Lecerf in [61] introduces extensions and a complexity analysis of the algorithm of Lickteig and Roy, with a particular focus on bivariate polynomials. When run over an effective ring endowed with the partially defined division routine, the algorithm yields a running time estimate similar to that of Reischert's. Lecerf realized an implementation of that algorithm, but observed that computations of subresultant chains based on Ducos' algorithm [37], or on evaluation-interpolation strategies, were faster in practice.

In [100, Chapter 11], von zur Gathen and Gerhard show how the nominal leading coefficients (see Section 2.2 for this term) of the subresultant chain of two univariate polynomials a, b over a field can be computed within $O(\mathrm{M}(n) \log n)$ operations in $\mathbf{k}$, by means of an adaptation of the Half-GCD algorithm. Here, we introduce an extension of their approach to compute any pair of consecutive non-zero subresultants of a, b within the same time bound. The details are presented in Section 3.4.

Our next optimization for subresultant chain computations relies on the observation that not all non-zero subresultants of a given subresultant chain may be needed. To illustrate this fact, consider two commutative rings $\mathbb{A}$ and $\mathbb{B}$, two non-constant univariate polynomials a, b in $\mathbb{A}[y]$ and a ring homomorphism Ψ from $\mathbb{A}$ to $\mathbb{B}$ so that $\Psi(\mathrm{lc}(a)) \neq 0$ and $\Psi(\mathrm{lc}(b)) \neq 0$ both hold. Then, the *specialization property of subresultants* tells us that the subresultant chain of $\Psi(a), \Psi(b)$ is the image of the subresultant chain of a, b via Ψ. This property has at least two important practical applications. When $\mathbb{B}$ is polynomial ring over a field, say $\mathbb{B}$ is $\mathbb{Z}_p[x]$ and $\mathbb{A}$ is $\mathbb{Z}_p$, then one can compute a GCD of $\Psi(a), \Psi(b)$ via evaluation and interpolation techniques. Similarly, say $\mathbb{B}$ is $\mathbb{Q}[x]/\langle m(x)\rangle$, where $m(x)$ is a square-free polynomial, then $\mathbb{B}$ is a product of fields then, letting $\mathbb{A}$ be $\mathbb{Q}[x]$, one can compute a GCD of $\Psi(a), \Psi(b)$ using the celebrated D5 Principle [33].

More generally, if $\mathbb{B}$ is $\mathbb{Q}[x_1, \ldots, x_n]/\langle T\rangle$, where $T = (t_1(x_1), \ldots, t_n(x_1, \ldots, x_n))$ is a zero-dimensional regular chain (generating a radical ideal), and $\mathbb{A}$ is $\mathbb{Q}[x_1, \ldots, x_n]$, then one can compute a so-called regular GCD of a and b modulo $\langle T\rangle$ [27]. The principle of that calculation generalizes the D5 Principle as follows:

(i) If the resultant of a, b is invertible modulo $\langle T\rangle$ then 1 is a regular GCD of a and b modulo $\langle T\rangle$;

(ii) If, for some k, the nominal leading coefficients $s_0, \ldots, s_{k-1}$ are all zero modulo $\langle T\rangle$, and s_k is invertible modulo $\langle T\rangle$, then the subresultant S_k of index k of a, b is a

Regular GCD of a and b modulo $\langle T \rangle$; and

(iii) One can always reduce to one of the above two cases by splitting T, when a zero-divisor of $\mathbb{B}$ is encountered.

In practice, in the above procedure, k is often zero, which can be seen as a consequence of the celebrated *Shape Lemma* [18]. This suggests to compute the subresultant chain of a, b in $\mathbb{A}[y]$ speculatively. To be precise, and taking advantage of the Half-GCD algorithm, it is desirable to compute the subresultants of index 0 and 1, delaying the computation of subresultants of higher index until proven necessary.

We discuss that idea of computing subresultants speculatively in Section 3.4. Making that approach successful, in comparison to non-speculative approaches, requires to overcome several obstacles:

1. computing efficiently the subresultants S_0 and S_1, via the Half-GCD; and

2. developing an effective "recovery" strategy in case of "misprediction", that is, when subresultants of index higher than 1 turn out to be needed.

To address the first obstacle, our implementation combines various schemes for the Half-GCD, inspired by the work done in NTL [93]. To address the second obstacle, when we compute the subresultants of index 0 and 1 via the Half-GCD, we record or *cache* the sequence of quotients (associated with the Euclidean remainders) so as to easily obtain subresultants of index higher than 1, if needed.

Moreover, we study parallel opportunities in order to compute subresultant chains of bivariate polynomials over the integers using evaluation-interpolation methods and and the Chinese Remainder Theorem (CRT). To implement these parallel schemes, we make use of the *map pattern* mechanism of the BPAS multithreaded interface (Section 2.3) to parallelize *evaluation*, *interpolation*, and *combination* parts of the code. Our experimentation for a test suite of over 3000 polynomial systems, which are collected from real-world applications, shows a parallel-speed-up of up to $4\times$ on a 12-core machine [10].

The extensive experimentation results in Section 3.6 indicate that the performance of our univariate polynomials over finite fields (based on FFT) are closely comparable with their counterparts in NTL. In addition, we have aggressively tuned our subresultant schemes based on evaluation-interpolation techniques. Our modular subresultant chain algorithms are up to $10\times$ and $400\times$ faster than non-modular counterparts (mainly Ducos' subresultant chain algorithm) in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively. Further, utilizing the Half-GCD algorithm to compute subresultants speculatively yields an additional speed-up factor of $7\times$ and $2\times$ for polynomials in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively.

Co-Authorship Statement

The data-structure and basic arithmetic are implemented by Asadi. The Montgomery operations over $\mathbb{Z}_p$ and FFT algorithm are borrowed from different research projects in BPAS [30, 31], while the rest of fast arithmetic operations are implemented by Asadi with following ideas in NTL [93] and `Aldor` [40]. The speculative and caching schemes are implemented by Asadi [14] follows pseudo-codes in [100], and the multithreaded bivariate subresultant chain schemes are implemented by Asadi and Brandt [10].

3.2 Basic Arithmetic

Let $\mathbb{B}$ be a commutative ring containing unity 1 where $0 \neq 1$. An ordered sequence $(a_0, \ldots, a_n) \in \mathbb{B}^{n+1}$ for a non-negative integer n is called a univariate polynomials over $\mathbb{B}$ and denoted as:

$$a = \sum_{i=0}^{n} a_i y^i = a_n y^n + \ldots + a_0 \in \mathbb{B}[y],$$

with $a_n \neq 0$. Then $\deg(a, y) := n$ is the *degree* of a w.r.t. y, $\mathrm{lc}(a, y) := a_n$ is the *leading coefficient* of a w.r.t. y, $a_n y^n$ is the *leading monomial* of a, and $\mathrm{coeff}(a, i, y) := a_i$ for $0 \leq i \leq n$.

We denote $\deg(a)$ and $\mathrm{lc}(a)$ for the degree and leading coefficient of a univariate polynomial $a \in \mathbb{B}[y]$ for simplicity. One can show that $\mathbb{B}[y]$ is a commutative ring of polynomials with addition and multiplication operations. In the following, we review the definitions of *zero-divisors* and *units* of $\mathbb{B}[y]$ from [100].

Definition 8 *Assume $a = \sum_{i=0}^{n} a_i y^i$ be a non-zero polynomial over the commutative ring $\mathbb{B}$. Then, a is a zero-divisor if and only if there exists some non-zero polynomial $b \in \mathbb{B}[y]$ with $ab = 0$.*

Euclidean Algorithm

One can define the division operation for a polynomial ring with unit leading coefficients in $\mathbb{B}[y]$ that is in fact a general version of *Euclidean* algorithm over a commutative ring.

Theorem 3 *Let $a, b \in \mathbb{B}[y]$, b is a non-constant polynomial and $\mathrm{lc}(b)$ is a unit. Then, there exist unique polynomials $q := \mathrm{quo}(a, b), r := \mathrm{rem}(a, b)$, are so-called quotient and remainder respectively, in $\mathbb{B}[y]$ such that the following property holds,*

$$a = qb + r \quad \textit{and} \quad (r = 0 \textit{ or } \deg(r) < \deg(b)).$$

PROOF. [100, Section 3.1]

In the division algorithm, if $\mathrm{lc}(b)$ is not a unit element of $\mathbb{B}$, then there is not any guarantee for the uniqueness of q, r in $\mathbb{B}[y]$.

Example 3 *Consider $a = y^2 + 3y + 2$ and $b = 3y^2 + 4y + 1$ in $\mathbb{Z}_6[y] := \mathbb{Z}/6\mathbb{Z}[y]$. Since the $\mathrm{lc}(b)$ is not a unit, the division of a by b using the Euclidean algorithm can produce several valid couples (q, r). For instance,*

$$(q_1, r_1) = (4y + 5, 3y + 3) \quad \textit{and} \quad (q_2, r_2) = (4y + 3, 5y + 5),$$

when $a = q_1 b + r_1 = q_2 b + r_2$ holds.

GCD and Resultants

From [96, 100], we review the definitions of GCD, Sylvester matrix and resultant of two univariate polynomials.

Definition 9 *Let $\mathbb{B}$ be a Euclidean domain. The greatest common divisor (GCD) of $a, b \in \mathbb{B}$ is such $g \in \mathbb{B}$ if g divides both a, b, and any common divisor of a, b divides g.*

The GCD of a, b is denoted $\gcd(a, b) \in \mathbb{B}$ and one can compute it using the *Euclidean* algorithm repeatedly to generate the sequence of remainders $(r_0 := a, r_1 := b, r_2 := \mathrm{rem}(r_0, r_1), \ldots, r_\ell := \mathrm{rem}(r_{\ell-2}, r_{\ell-1}))$ for $\ell \in \mathbb{N}$, $r_{\ell+1} = 0$ with $\deg(r_{\ell+1}) = -\infty$, $\gcd(a, b) := r_\ell$, and the corresponding quotients $(q_1 := \mathrm{quo}(r_0, r_1), q_2, \ldots, q_\ell := \mathrm{quo}(r_{\ell-1}, r_\ell))$ from

$$r_{i+1} = \mathrm{rem}(r_{i-1}, r_i) = r_{i-1} - q_i r_i,$$

for $1 \leq i < \ell$. This computation requires $O(n^2)$ operations in $\mathbb{B}$.

In order to analyze the algorithm, we represent r_i as linear combinations of a, b as $r_i = s_i a + t_i b$, and introduce the following 2×2 matrices where $(s_0, t_0, s_1, t_1) = (1, 0, 0, 1)$ and $q_i := \mathrm{quo}(r_{i-2}, r_{i-1})$ in $\mathbb{B}[x]$,

$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} \text{ and } Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \text{ for } 1 \leq i \leq k\ell, \tag{3.1}$$

then, we define $R_i = Q_i \cdots Q_1 R_0$ and the following theorem holds.

Theorem 4 *Assume $(r_0, r_1, \dots, r_k)$ is the sequence of remainders of a, b such that $r_i = s_i a + t_i b$ when $s_i, t_i \in \mathbb{B}[y]$ and $r_{k+1} = 0$. For every $0 \leq i \leq k$ we have:*

(i) $R_i \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$,

(ii) $R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$,

(iii) $\gcd(a, b) = \gcd(r_i, r_{i+1}) = r_k$,

(iv) $\gcd(s_i, t_i) = 1$.

Proof. [100, Lemma 3.8]

The polynomials $s_i, t_i \in \mathbb{B}[y]$ are called the Bézout coefficients of $\gcd(a, b)$ and the derived algorithm from Theorem 4 to compute them is a well-known algorithm named extended Euclidean algorithm (EEA). Generally, computing the GCD of two polynomials might fail, so Theorem 3 and thus the Euclidean algorithm do not work for an arbitrary polynomial ring.

Example 4 *Consider polynomials $a = y^2+3y+2$ and $b = 3y^2+4y+1$ in $\mathbb{Z}[y]$. Neither the quotient nor the remainder of this division is in $\mathbb{Z}[y]$, although the $\gcd(a, b) = y+1 \in \mathbb{Z}[y]$.*

A solution can be to work over the fraction field of the coefficient ring. However, computing over those domains generally lead to the severe coefficient growth. Besides, we would prefer to have an algorithm whose results remain in the same coefficient ring of $\mathbb{B}$.

Definition 10 *Let $a = \sum_{i=0}^{m} a_i y^i, b = \sum_{i=0}^{n} b_i y^i \in \mathbb{B}[y]$ be two non-zero and non-constant polynomials. The Sylvester matrix of a, b is the square matrix of order $n + m$ with coefficients in $\mathbb{B}$, denoted by $\operatorname{sylv}(a, b)$ and defined as follows:*

$$\operatorname{sylv}(a,b) = \begin{array}{c} n\left\{\vphantom{\begin{array}{c}1\\1\\1\\1\end{array}}\right. \\ m\left\{\vphantom{\begin{array}{c}1\\1\\1\\1\end{array}}\right. \end{array} \begin{bmatrix} a_m & a_{m-1} & \cdots & a_0 & & & \\ & a_m & a_{m-1} & \cdots & a_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & a_m & a_{m-1} & \cdots & a_0 \\ b_n & b_{n-1} & \cdots & b_0 & & & \\ & b_n & b_{n-1} & \cdots & b_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & b_n & b_{n-1} & \cdots & b_0 \end{bmatrix}, \tag{3.2}$$

Algorithm 3 Resultant (a, b, y)

```
Require: non-zero univariate polynomials $a, b \in \mathbb{B}[y]$ such that $\deg(a) + \deg(b) > 0$
Ensure: resultant of $a, b$ in $\mathbb{B}$
 1: $m := \deg(a)$; $n := \deg(b)$
 2: if $m < n$ then
 3:     return $(-1)^{nm}$Resultant$(b, a, y)$
 4: if $n = 0$ then
 5:     return $\mathrm{lc}(b)^m$
 6: $c := \frac{a}{b}$
 7: if $c = 0$ then
 8:     return $c$
 9: $p := \deg(c)$
10: return $(-1)^{nm}\mathrm{lc}(b)^{m-p}$resultant$(b, c, y)$
```

In addition, the determinant of $\mathrm{sylv}(a, b)$ is so-called the resultant a, b and denoted by $\mathrm{res}(a, b)$. Here, we present a Euclidean-like algorithm to compute the resultant of a, b based on the determinant of $\mathrm{sylv}(a, b)$ in Algorithm 3.

Example 5 *Consider two polynomials $a = a_2y^2 + a_1y + a_0$ and $b = 2a_2y + a_1$ in $\mathbb{B}[y]$. Then the Sylvester matrix of a and b is:*

$$S = \begin{bmatrix} a_2 & 2a_2 & 0 \\ a_1 & a_1 & 2a_2 \\ a_0 & 0 & a_1 \end{bmatrix},$$

whose determinant is $\det(S) = a_2(4a_2c_0 - b_1^2)$. Whenever $a_2 \neq 0$, polynomials a and b have a common solution (or equivalently, $a = 0$ has a solution of multiplicity 2) and it implies that $\mathrm{res}(a, b) = 0$ if and only if $\deg(\gcd(a, a')) > 0$.

Theorem 5 *Let $\mathbb{B}$ be a unique factorization domain (UFD). $\gcd(a, b)$ for polynomials $a, b \in \mathbb{B}[y]$ is a non-constant polynomial in $\mathbb{B}[y]$ if and only if $\mathrm{res}(a, b) = 0$.*

Proof. [100, Corollary 6.17]

The Algorithm 3 can be transformed to compute the GCD of $a, b \in \mathbb{B}[y]$ in case $\mathrm{res}(a, b) = 0$ using Theorem 5 and this leads to Algorithm 4.

Algorithm 4 GCD (a, b, y)

```
Require: non-zero univariate polynomials $a, b \in \mathbb{B}[y]$ such that $\deg(a) + \deg(b) > 0$
Ensure: $\gcd(a, b)$ if $\mathrm{res}(a, b) = 0$, else resultant $a, b$ in $\mathbb{B}$.
 1: $m := \deg(a)$; $n := \deg(b)$
 2: if $m < n$ then
 3:     $r := (-1)^{nm}$
 4:     $(a, b) := (b, a)$; $(m, n) := (n, m)$
 5: else
 6:     $r := 1$
 7: while true do
 8:     if $n = 0$ then
 9:         return $r\ \mathrm{lc}(b)^m$
10:     $c := a/b$
11:     if $c = 0$ then
12:         return $\frac{b}{\mathrm{lc}(b)}$
13:     $p := \deg(c)$
14:     $r := r\ (-1)^{nm} \mathrm{lc}(b)^{m-p}$
15:     $(a, b) := (b, c)$; $(m, n) := (n, p)$
16: end while
```

Subresultants

A refinement of Algorithm 3 would let us compute the subresultant chain of a, b:

$$\mathrm{subres}(a, b) = (S_n, \ldots, S_0 := \mathrm{res}(a, b)),$$

by keeping track of successive remainders. This algorithm is in fact an adapted version of the *fundamental theorem on subresultants* to compute the nominal leading coefficient of subresultants.

Theorem 6 *Assume the remainder sequence* $(r_0, r_1, \ldots, r_\ell)$ *with* $\deg(r_i) = n_i$ *for* $0 \leq i \leq \ell$*. For* $k = 0, \ldots, n_1$*, the nominal leading coefficient of the* k*-th subresultant of* (r_0, r_1) *is either* 0 *or* s_k *if there exists* $i \leq \ell$ *such that* $k = \deg(r_i)$*,*

$$s_k = (-1)^{\tau_i} \prod_{1 \leq j < i} \mathrm{lc}(r_j)^{n_{j-1} - n_{j+1}} \mathrm{lc}(r_i)^{n_{i-1} - n_i},$$

where $\tau_i = \sum_{1 \leq j < i} (n_{j-1} - n_i)(n_j - n_i)$.

Proof. [100, Theorem 11.16].

In the following, we review some important upper-bounds on subresultants. We take advantage of these bounds to perform subresultant algorithms in $\mathbb{Z}[x, y]$ from fast arithmetic over prime fields. Let $\mathbf{k}$ be a field, e.g. $\mathbb{Z}_p$.

Theorem 7 *Let $a, b \in \mathbf{k}[x, y]$. Then*

$$\deg(res(a, b, y), x) \leq tdeg(a)\ tdeg(b),$$

where $tdeg(a), tdeg(b)$ are total degrees of a, b, respectively.

Proof. [100, Theorem 6.22]

Theorem 8 *Let $a, b \in \mathbf{k}[x, y]$, $n = \deg(a, y)$, $m = \deg(b, y)$ and $d = \max\{deg(a, x), deg(b, x)\}$. For every $1 \leq k \leq \min\{(n, m)\}$,*

$$deg(S_k(a, b, y), x) \leq (n + m - 2k)d.$$

Proof. [100, Theorem 6.51]

3.3 Fast Arithmetic over Prime Fields

Arithmetic over prime fields plays a central role in computer algebra. The performance of these operations is important as they are core routines of other operations, e.g. resultant, subresultant, factorization, and evaluation-interpolation.

For instance, if we use a faster multiplication algorithm, then almost every polynomial arithmetic operation can be performed faster. In this thesis, the implementation of these prime fields are using a prime of size one machine word (64-bit). Hence, increasing the arithmetic to greater precision is achieved through the Chinese Remainder Theorem (CRT) algorithm. In this section, we review the definitions and algorithmic efficiencies of the following basic operations:

(i) Montgomery representation in $\mathbb{Z}_p$;

(ii) Classical, Karatsuba, and FFT-based multiplication operations in $\mathbb{Z}_p[y]$;

(iii) Division based on Power Series Inversion using Newton Iteration in $\mathbb{Z}_p[y]$;

(iv) GCD and Fast extended Euclidean algorithm (Half-GCD) operations in $\mathbb{Z}_p[y]$; and

(v) Lagrange and FFT-based evaluation-interpolation operations in $\mathbb{Z}_p[x, y]$.

3.3.1 Montgomery Representation

Montgomery multiplication was introduced by Montgomery in [80]. In this method, elements of a prime field convert to a so-called Montgomery Representation, where a modular multiplication is made faster by avoiding division by the modulus, thus also improving modular inverse. This representation has no effect on the performance of addition or subtraction.

Let $R > p$ be a number for a modulus p so that $\gcd(p, R) = 1$. Assume R is some power of 2; hence multiplication and division by R can be done cheaply by *bit shifting* operations. As $\gcd(p, R) = 1$, the extended Euclidean algorithm yields that there exists a unique pair (R', p') so that,

$$RR' - pp' = 1,$$

with $0 < R' < p$ and $0 < p' < R$, and so, $p' \equiv -p^{-1} \bmod R$.

For an arbitrary integer a where $0 \leq a < Rp$, the Montgomery *reduction* computes $c := aR^{-1} \bmod p$ by computing

$$\begin{aligned} m &\equiv ap' \bmod R, \\ c &= (a + mp)/R, \\ c &= c - p \textit{ if } c \geq p. \end{aligned}$$

For two elements of $a, b \in \mathbb{Z}_p$, the Montgomery *representation* of a, b are respectively $\bar{a} := aR \bmod p$ and $\bar{b} := bR \bmod p$. Addition and subtraction of $\bar{a}, \bar{b}$ conclude directly from $\bar{a} \pm \bar{b}$. Moreover, Multiplication derives using Montgomery *reduction* on $\bar{a}, \bar{b}$ as follows,

$$\bar{a}\bar{b}R^{-1} \equiv (aRbR)R^{-1} \equiv (ab)R \bmod p.$$

Throughout this thesis, we consider polynomials over prime fields where their coefficients are in *Montgomery representation*, and so, we use Montgomery arithmetic implemented in the BPAS library [30, 31].

3.3.2 Univariate Polynomial Multiplication

Polynomial multiplication is widely used within almost all routines in computer algebra. In the study of algorithms to compute multiplication, there are several procedures that must work together. This is due to the fact that as the degree of input polynomials increase, the complexity grows rapidly. Here we briefly review these algorithms and their complexities.

A multiplication time is considered as a map $\mathrm{M} : \mathbb{N} \to \mathbb{R}$, where $\mathbb{R}$ is the field of real numbers, such that:

(i) For any ring $\mathbb{B}$, polynomials of degrees less than n in $\mathbb{B}[y]$ can be multiplied in at most $\mathrm{M}(n)$ (addition and multiplication) operations in $\mathbb{B}$, and

(ii) For any $n \leq n'$, the inequality $\frac{\mathrm{M}(n)}{n} \leq \frac{\mathrm{M}(n')}{n'}$ holds.

Let polynomials $a = \sum_{i=0}^{n} a_i y^i$, $b = \sum_{j=0}^{m} b_j y^j$ in $\mathbb{Z}_p[y]$ with $\deg(a) = n$ and $\deg(b) = m$. The classical polynomial multiplication algorithm is calculated via the following generic formula,

$$ab = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j y^{i+j}.$$

The implementation of this algorithm requires two for-loops, and so, to multiply two polynomials with degree less than n, we have the multiplication time of

$$\mathrm{M}(n) = O(n^2).$$

Karatsuba algorithm is a divide-and-conquer multiplication algorithm discovered in 1962 [100]. The main trick with this algorithm is to reduce the number of intermediate multiplications. Let polynomials $a = \sum_{i=0}^{n-1} a_i y^i$, $b = \sum_{i=0}^{n-1} b_i y^i$ in $\mathbb{Z}_p[y]$ where $n = 2^k$ and $k \in \mathbb{N}$. The Karatsuba's trick computes the product ab as follows:

1. If $n = 1$ then compute the element a, b of $\mathbb{Z}_p$;
2. Define $a = A_1 y^{n/2} + A_0$ and $b = B_1 y^{n/2} + B_0$ where A_1, A_0, B_1, B_0 have degree less than $n/2$;
3. Compute A_0B_0, A_1B_1, and $(A_0 + A_1)(B_0 + B_1)$ recursively; and
4. Return $A_1B_1y^n + ((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1)y^{n/2} + A_0B_0$.

Since multiplication is more expensive than addition and subtraction, this trick reduces the total cost. One can shows that the complexity of this algorithm is:

$$\mathrm{M}(n) = O(n^{\log 3/\log 2}) = O(n^{1.59}).$$

There is further generalization of this algorithm in the literature known as Toom-Cook algorithm [100], that is faster for large enough n. This algorithm splits the input polynomials into $k \geq 3$ parts.

Let polynomials $a = \sum_{i=0}^{n-1} a_i y^i$, $b = \sum_{j=0}^{m-1} b_j y^j$ with $n \geq m$ in $\mathbb{Z}_p[y]$. The convolution of a, b is defined as follows.

Definition 11 *The convolution of polynomials $a, b \in \mathbb{Z}_p[y]$ w.r.t. n is denoted as $a *_n b$ and is defined:*

$$c = \sum_{k=0}^{n-1} \sum_{i+j \equiv k \ mod \ n} a_i b_j \ y^k.$$

From [100], the Discrete Fourier Transform (DFT) evaluates a univariate polynomial over $\mathbb{Z}_p$ with degree less than n at the successive powers of a so-called primitive root of unity denoted as $\omega \in \mathbb{Z}_p$. Note that using the inverse of this procedure over the evaluated images of a univariate polynomial yields a DFT-based interpolation algorithm.

An efficient implementation of the DFT algorithm known as Fast Fourier Transform (FFT) within the complexity $O(n \log n \log\log n)$ [31]. The FFT algorithm only works when appropriate roots of unity exist; hence, Schönhage and Strassen [91] showed how to create *virtual* roots that leads to a fast multiplication with a similar cost where there is not an appropriate ω; see [42] for a C implementation of this algorithm. From [100], we have $\text{FFT}_\omega(a *_n b) = \text{FFT}_\omega(a)\ \text{FFT}_\omega(b)$, hence the convolution of two polynomials can be computed using Algorithm 5 from [100].

Algorithm 5 FastConvolution(a, b, ω)

```
Require: $a, b \in \mathbb{Z}_p[y]$ with degree less than $n = 2^k \in \mathbb{N}$, and a $n$-th primitive root of unity
    $\omega \in \mathbb{Z}_p$
Ensure: $a *_n b \in \mathbb{Z}_p[y]$
 1: compute the first $n$ powers of $\omega$
 2: $\alpha := \text{FFT}_\omega(a)$
 3: $\beta := \text{FFT}_\omega(b)$
 4: $\gamma :=$ point-wise product of $\alpha$ and $\beta$
 5: return $\text{FFT}_\omega^{-1}(\gamma) := \frac{1}{n}\text{FFT}_{\omega^{-1}}(\gamma)$
```

We use a C implementation of FFT algorithm from [31] to achieve FFT-based multiplication for 64-bit primes in BPAS along with developing the classical and Karatsuba algorithms to multiply two univariate polynomials in $\mathbb{Z}_p[y]$ where FFT is not available.

3.3.3 Division Operation

Division is another important basic operation where the cost of the classical implementation is $O(n^2)$ using the famous Euclidean algorithm. Similar to multiplication, it has a direct impact on the performance of other algorithms. Therefore, an asymptotically fast division is required to obtain asymptotically fast polynomial arithmetic.

Power series inversion using Newton iteration method provides a fast method for computing multiplicative inverses [23]. This algorithm computes the inverse of polynomial $a \in \mathbb{Z}_p[y]$ with $a(0) = 1$ and $\deg(a) < \ell$ modulo y^ℓ for a given $\ell \in \mathbb{N}$. To achieve an efficient implementation of the inversion, we use a FFT-based approach known as the *Middle Product Technique* following the implementation in Aldor [24] to develop a fast division algorithm in the BPAS library.

3.3.4 Fast EEA (Half-GCD) Operation

The Fast extended Euclidean algorithm or Half-GCD technique originated with the ideas of Lehmer, Knuth and Schönhage [56, 63, 90]. This algorithm runs within $O(\mathrm{M}(n) \log n)$ operations in $\mathbb{Z}_p$ while the cost of the classical EEA is $O(n^2)$ [100]. The Half-GCD can be interpreted as an asymptotically fast technique to compute the GCD through calculating the quotient matrices and $R = Q_\ell \cdots Q_1 R_0$ in $\mathbb{Z}_p[y]$ so that we have:

$$\begin{pmatrix} \gcd(a, b) \\ 0 \end{pmatrix} = R \begin{pmatrix} a \\ b \end{pmatrix}.$$

The major difference between the classical EEA and Half-GCD is that while the EEA computes all the remainders $r_0, r_1, \ldots, r_\ell = \gcd(r_0, r_1)$, the Half-GCD computes only two consecutive ones with computing their associated $Q_j = Q_j \cdots Q_1 R_0$ for $1 \le j \le \ell$ and utilizing a sequence of truncated remainders.

Algorithm 6 presents a simplified version of this procedure. This algorithm computes one of the intermediate quotient matrices, namely Q_j, so that

$$\begin{pmatrix} a' \\ b' \end{pmatrix} = Q_j \begin{pmatrix} a \\ b \end{pmatrix},$$

where $deg(a') \ge deg(a)/2 > deg(b')$. One can use this algorithm recursively to compute the last quotient matrix and so the $\gcd(a, b)$ within $O(\mathrm{M}(n) \log n)$ operations in $\mathbb{Z}_p$.

Theorem 9 *Algorithm 6 is correct and terminates in $O(M(n) \log n)$ operations in $\mathbb{Z}_p$.*

Proof. [103, Lemma 6.10]

Here, the basic principle is the quotients of polynomials of degrees n_1 and n_2 with $n_1 > n_2$, depend only on the leading $2(n_1 + n_2) + 1$ terms of the dividend and the leading $n_1 - n_2 + 1$ terms of the divisor [100, Lemma 11.1]. And so, this fact of focusing on the quotient sequence, rather than the remainder sequence (like the classical EEA) yields better performance in realizing the GCD of two polynomials in $\mathbb{Z}_p[y]$.

Algorithm 6 HalfGCD(a, b)

```
Require: $a, b \in \mathbb{Z}_p[y]$ with degree less than $n := \deg(a) \geq \deg(b)$
Ensure: a $2 \times 2$ quotient matrix that reduces the computation of $gcd(a, b)$ to $gcd(a', b')$ where
    $deg(a') \geq deg(a)/2 > deg(b')$
 1: $m := \lceil n/2 \rceil$
 2: if $deg(b) < m$ or $m < 1$ then
 3:     return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
 4: $a_{trunc} := \mathrm{quo}(a, x^m)$; $b_{trunc} := \mathrm{quo}(b, x^m)$
 5: $R := \textsc{HalfGCD}(a_{trunc}, b_{trunc})$
 6: $\begin{pmatrix} a' \\ b' \end{pmatrix} := R \begin{pmatrix} a \\ b \end{pmatrix}$
 7: if $b' = 0$ then
 8:     return $R$.
 9: $r := \mathrm{rem}(a', b')$; $q := \mathrm{quo}(a', b')$
10: $Q := \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$
11: $l := 2m - deg(b')$
12: $a'_{trunc} := \mathrm{quo}(a', x^l)$; $b'_{trunc} := \mathrm{quo}(b', x^l)$
13: $S := \textsc{HalfGCD}(a'_{trunc}, b'_{trunc})$
14: return $SQR$
```

Example Consider $a = y^2 + 3y + 2,\ b = 3y^2 + 4y + 1$ in $\mathbb{Q}[y]$. In Algorithm 6, $(n, m) = (2, 1)$, $R = S = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and matrix Q builds with $a_{trunc} = y + 3, b_{trunc} = 3y + 4$. Then,

$$\textsc{HalfGCD}(a, b) = \begin{pmatrix} 0 & 1 \\ 1 & -1/3 \end{pmatrix}.$$

In addition, the multiplication of HalfGCD(a, b) and $\begin{pmatrix} a & b \end{pmatrix}^T$ produces $\frac{5}{3}(y + 1)$ that is a non-monic form of $\gcd(a, b)$.

3.3.5 Evaluation-Interpolation Operations

Let $\mathbb{B}$ be an integral domain and $\eta \in \mathbb{B}$. For a bivariate polynomial,

$$a(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_{i,j} x^i y^j,$$

in $\mathbb{B}[x, y]$ with variables $x < y$. An *evaluation* algorithm is in fact the *homomorphism* function:

$$\Psi_{x=\eta} : \ \mathbb{B}[x, y] \to \mathbb{B}[y],$$

where,

$$\Psi_{x=\eta}(a(x, y)) = a(\eta, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_{i,j} \eta^i y^j \in \mathbb{B}[y].$$

Throughout this chapter, we denote $\Psi_{x=\eta}(a(x, y))$ as $a(x, y)|_{x=\eta}$ or $a|_{x=\eta}$ where η is called the *evaluation point.* Moreover, The inverse of evaluation homomorphism is known as *interpolation* and is defined as follows.

Definition 12 *Let* $\mathbf{k}$ *be a field. Given distinct evaluation points* $\{\eta_0, \eta_1, \ldots, \eta_n\}$ *in* $\mathbf{k}$ *and the values* $\{\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_n\}$ *in* $\mathbf{k}[y]$*. The interpolation problem in* $\mathbf{k}[y]$ *is finding a polynomial* $a \in \mathbf{k}[x, y]$ *satisfying*

$$a(\eta_i, y) = \bar{a}_i, \quad \textit{for } 0 \le i \le n.$$

Theorem 10 shows the existence and uniqueness of such function.

Theorem 10 *For the distinct evaluation points* $\eta_0, \eta_1, \ldots, \eta_n$*, there exists a unique polynomial* $a \in \mathbf{k}[x, y]$ *of degree at most* n *in the variable* x *such that* $a(\eta_i, y) = \bar{a}_i$ *for* $0 \le i \le n$*.*

PROOF. [43, Theorem 5.8]
There are several so-called classical algorithms for interpolation [100, Section 5.3]. The algorithms known as *Newton interpolation* and *Lagrange interpolation* run within $O(n^2)$ operations in $\mathbf{k}[y]$ to interpolate $a \in \mathbf{k}[x, y]$ of degree at most n in the variable x.

For a univariate polynomial $b \in \mathbb{Z}_p[y]$, one can utilize the FFT algorithm for evaluation and interpolation within $O(\mathrm{M}(n) \log n)$ operations using FFT. In this approach, The FFT evaluates b at powers of an appropriate primitive root of unity ω. Also, the interpolation performs by executing FFT^{-1} from the images and evaluation points, i.e. powers of ω.

We extend this asymptotically fast approach to bivariate polynomials borrowing ideas of the *blocking strategy* developed in [45] to calculate the subresultant chain of multivariate polynomials on GPUs via FFT-based evaluation and interpolation algorithms.

3.4 Speculative Subresultant Algorithms

As discussed in the previous section, when the ring $\mathbb{B}$ is a prime field $\mathbb{Z}_p$ for an odd prime p, the computation of the subresultant chain of the polynomials $a, b \in \mathbb{Z}_p[y]$ could take advantage of asymptotically fast algorithms and the *fundamental theorem on subresultants* (Theorem 6) to compute the entire chain.

In this section, we introduce a speculative technique using the Half-GCD algorithm in order to compute two successive subresultants *speculatively* from subres(a, b). We further extend our speculative approach to *cache* the intermediate data computed in the Half-GCD algorithm to calculate subresultants with higher indices by request without calling the Half-GCD algorithm again.

Let $\mathbf{k}$ be a field, e.g. $\mathbf{k} = \mathbb{Z}_p$. Let non-zero polynomials $a, b \in \mathbf{k}[y]$ with $n_0 := \deg(a)$, $n_1 := \deg(b)$ with $n_0 \geq n_1$. The extended Euclidean algorithm (EEA) computes the successive remainders $(r_0 := a, r_1 := b, r_2, \ldots, r_\ell)$ with degree sequence $(n_0, n_1, n_2 \ldots, n_\ell)$ and the corresponding quotients $(q_1, q_2, \ldots, q_\ell)$; see Section 3.2 for more details.

We define $m_i := \deg(q_i)$, so that we have $m_i = n_{i-1} - n_i$ for $1 \leq i \leq \ell$. The degree sequence $(n_0, \ldots, n_l)$ is said to be *normal* if $n_{i+1} = n_i - 1$ holds, for $1 \leq i < \ell$, or, equivalently if $\deg(q_i) = 1$ holds, for $1 \leq i \leq \ell$. Throughout this section, we take advantage of the Half-GCD algorithm presented in [100, Chapter 11].

Speculative and Caching Schemes in $\mathbf{k}[y]$

For a non-negative $k \leq n_0$, the Algorithm 11.6 in [100] computes the quotients $q_1, \ldots, q_{h_k}$ where h_k is defined as:

$$h_k = \max\left\{0 \leq j \leq \ell \mid \sum_{i=1}^{j} m_i \leq k\right\}, \tag{3.3}$$

the maximum $j \in \mathbb{N}$ so that $\sum_{1 \leq i \leq j} \deg(q_i) \leq k$. This is done within $(22\mathrm{M}(k)+O(k)) \log k$ operations in $\mathbf{k}$. From Equation 3.3, $h_k \leq \min(k, \ell)$, and

$$\sum_{i=1}^{h_k} m_i = \sum_{i=1}^{h_k} (n_{i-1} - n_i) = n_0 - n_{h_k} \leq k < \sum_{i=1}^{h_k+1} m_i = n_0 - n_{h_k+1}. \tag{3.4}$$

Thus, $n_{h_k+1} < n_0 - k \leq n_{h_k}$, and so h_k can be uniquely determined.

Due to the deep relation between subresultants and the remainders of the EEA, the Half-GCD technique can support computing subresultants. This approach is studied in [100]. The Half-GCD algorithm is used to compute the nominal leading coefficient of subresultants up to s_ρ for $\rho = n_{h_k}$ by computing the quotients $q_1, \ldots, q_{h_k}$, calculating the

$\mathrm{lc}(r_i) = \mathrm{lc}(r_{i-1})/\mathrm{lc}(q_i)$ from $\mathrm{lc}(r_0)$ for $1 \leq i \leq h_k$, and applying Theorem 6. The resulting procedure runs within the same complexity as the Half-GCD algorithm.

However, for the purpose of computing two successive subresultants $S_{n_v}, S_{n_{v+1}}$ given $0 \leq \rho < n_1$, for $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$, we need to compute quotients $q_1, \ldots, q_{h_\rho}$ where h_ρ is defined as:

$$h_\rho = \max\Big\{0 \leq j < \ell \mid n_j > \rho\Big\}, \tag{3.5}$$

using Half-GCD. Let $k = n_0 - \rho$, Equations 3.4 and 3.5 deduce,

$$n_{h_\rho+1} \leq n_0 - k < n_{h_\rho}, \text{ and } h_\rho \leq h_k.$$

Thus, to compute the array of quotients $q_1, \ldots, q_{h_\rho}$, we can utilize an adaptation of the Half-GCD algorithm of [100]. Algorithm 7 is this adaptation and runs within the same complexity as Algorithm 11.6.

Algorithm 7 receives as input two polynomials $r_0 := a, r_1 := b$ in $\mathbf{k}[y]$, with $n_0 \geq n_1$, $0 \leq k \in \mathbb{N}$, $\rho \leq n_0$ where ρ, by default, is $n_0 - k$, and the array $\mathcal{A}$ of the leading coefficients of the remainders that have been computed so far. This array should be initialized to size $n_0 + 1$ with $\mathcal{A}[n_0] = \mathrm{lc}(r_0)$ and $\mathcal{A}[i] = 0$ for $0 \leq i < n_0$. $\mathcal{A}$ is updated in-place as necessary. The algorithm returns the array of quotients $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ and matrix $M := Q_{h_\rho} \cdots Q_1$.

A direct application of Algorithm 7 along with *the fundamental theorem on subresultants* is the *speculative* algorithm to compute non-zero subresultants $S_{n_v}, S_{n_{v+1}}$ via Algorithm 8. This algorithm is a *speculative* subresultant algorithm based on Half-GCD to calculate two successive subresultants without computing others in the chain. Moreover, this algorithm returns intermediate data that has been computed by the Half-GCD algorithm—the array $\mathcal{R}$ of the remainders, the array $\mathcal{Q}$ of the quotients and the array $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence—to later calculate higher subresultants in the chain without calling Half-GCD again. This *caching* scheme is shown in Algorithm 9.

The following example explains the caching technique utilizing the speculative approach.

Example 6 *For non-zero polynomials* $a, b \in \mathbf{k}[y]$ *with* $n_0 = \deg(a), n_1 = \deg(b)$*, so that we have* $n_0 \geq n_1$*. The subresultant call* $\textsc{Subresultant}(a, b, 0)$ *returns* $S_0(a,b), S_1(a,b)$ *speculatively without computing*

$$(S_{n_1}, S_{n_1-1}, S_{n_1-2}, \ldots, S_2),$$

Algorithm 7 AdaptedHGCD($r_0, r_1, k, \rho, \mathcal{A}$)

```
Require: $r_0, r_1 \in \mathbf{k}[y]$ with $n_0 = \deg(r_0) \geq n_1 = \deg(r_1)$, $0 \leq k \leq n_0$, $0 \leq \rho \leq n_0$ is an upper bound for the degree of the last computed remainder that, by default, is $n_0 - k$ and is fixed in recursive calls (See Algorithm 8), the array $\mathcal{A}$ of the leading coefficients of the remainders (in the Euclidean sequence) which have been computed so far
Ensure: $h_\rho \in \mathbb{N}$ so that $h_\rho = \max\{j \mid n_j > \rho\}$, the array $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ of the first $h_\rho$ quotients associated with remainders in the Euclidean sequence and the matrix $M := Q_{h_\rho} \cdots Q_1$; the array $\mathcal{A}$ of leading coefficients is updated in-place
 1: if $r_1 = 0$ or $\rho \geq n_1$ then
 2:     return $\left(0, (), \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$
 3: if $k = 0$ and $n_0 = n_1$ then
 4:     return $\left(1, (\mathrm{lc}(r_0)/\mathrm{lc}(r_1)), \begin{bmatrix} 0 & 1 \\ 1 & -\mathrm{lc}(r_0)/\mathrm{lc}(r_1) \end{bmatrix}\right)$
 5: $m_1 := \lceil \frac{k}{2} \rceil$; $\delta_1 := \max(\deg(r_0) - 2\ (m_1 - 1), 0)$; $\lambda := \max(\deg(r_0) - 2k, 0)$
 6: $\left(h', (q_1, \ldots, q_{h'}), R\right) :=$ AdaptedHGCD($\mathrm{quo}(r_0, y^{\delta_1}), \mathrm{quo}(r_1, y^{\delta_1}), m_1 - 1, \rho, \mathcal{A}$)
 7: $\begin{bmatrix} c \\ d \end{bmatrix} := R \begin{bmatrix} \mathrm{quo}(r_0, y^\lambda) \\ \mathrm{quo}(r_1, y^\lambda) \end{bmatrix}$ where $R := \begin{bmatrix} R_{00} & R_{01} \\ R_{10} & R_{11} \end{bmatrix}$
 8: $m_2 := \deg(c) + \deg(R_{11}) - k$
 9: if $d = 0$ or $m_2 > \deg(d)$ then
10:     return $\left(h', (q_1, \ldots, q_{h'}), R\right)$
11: $r := \mathrm{rem}(c, d)$; $q := \mathrm{quo}(c, d)$; $Q := \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix}$
12: $n_{h'+1} := n_{h'} - \deg(q)$
13: if $n_{h'+1} \leq \rho$ then
14:     return $\left(h', (q_1, \ldots, q_{h'}, q), R\right)$
15: $\mathcal{A}[n_{h'+1}] := \mathcal{A}[n_{h'}]/\mathrm{lc}(q)$
16: $\delta_2 := \max(2m_2 - \deg(d), 0)$
17: $\left(h^*, (q_{h'+2}, \ldots, q_{h'+h^*+1}), S\right) :=$
    AdaptedHGCD($\mathrm{quo}(d, y^{\delta_2}), \mathrm{quo}(r, y^{\delta_2}), \deg(d) - m_2, \rho, \mathcal{A}$)
18: return $\left(h_\rho := h' + h^* + 1, \mathcal{Q} := (q_1, \ldots, q_{h_\rho}), M := SQR\right)$
```

arrays $\mathcal{Q} = (q_1, \ldots, q_\ell)$, $\mathcal{R} = (r_\ell, r_{\ell-1})$, and $\mathcal{A}$. Therefore, any attempt to compute subresultants with higher indices can be addressed by utilizing the arrays $\mathcal{Q}, \mathcal{R}, \mathcal{A}$ instead of calling Half-GCD again

```
Algorithm 8 Subresultant$(a, b, \rho)$
Require: $a, b \in \mathbf{k}[x] \setminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$
Ensure: $S_{n_v}(a, b)$, $S_{n_{v+1}}(a, b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$, the array $\mathcal{Q}$ of the
    quotients, the array $\mathcal{R}$ of the remainders, and the array $\mathcal{A}$ of the leading coefficients of the
    remainders (in the Euclidean sequence) that have been computed so far
 1: $\mathcal{A} := (0, \ldots, 0, \mathrm{lc}(a))$ where $\mathcal{A}[n_0] = \mathrm{lc}(a)$ and $\mathcal{A}[i] = 0$ for $0 \leq i < n_0$
 2: if $\rho \geq n_1$ then
 3:     $\mathcal{A}[n_1] = \mathrm{lc}(b)$
 4:     return $\Big((a, \mathrm{lc}(b)^{m-n-1} b), (), (), \mathcal{A}\Big)$
 5: $(v, \mathcal{Q}, M) := \textsc{AdaptedHGCD}(a, b, n_0 - \rho, \rho, \mathcal{A})$
 6: deduce $\Big(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_v = \deg(r_v)\Big)$ from $a, b$ and $\mathcal{Q}$.
 7: $\begin{bmatrix} r_v \\ r_{v+1} \end{bmatrix} := M \begin{bmatrix} a \\ b \end{bmatrix}$; $\mathcal{R} := (r_v, r_{v+1})$; $n_{v+1} := \deg(r_{v+1})$
 8: $\tau_v := 0$; $\tau_{v+1} := 0$; $\alpha := 1$
 9: for $j$ from $1$ to $v - 1$ do
10:     $\tau_v := \tau_v + (n_{j-1} - n_v)(n_j - n_v)$
11:     $\tau_{v+1} := \tau_{v+1} + (n_{j-1} - n_{v+1})(n_j - n_{v+1})$
12:     $\alpha := \alpha\ \mathcal{A}[n_j]^{n_{j-1} - n_{j+1}}$
13: $\tau_{v+1} := \tau_{v+1} + (n_{v-1} - n_{v+1})(n_v - n_{v+1})$
14: $S_{n_v} := (-1)^{\tau_v} \alpha\ r_v$
15: $S_{n_{v+1}} := (-1)^{\tau_{v+1}} \alpha\ \mathcal{A}[n_v]^{n_{v-1} - n_{v+1}}\ r_{v+1}$
16: return $\Big((S_{n_v}, S_{n_{v+1}}), \mathcal{Q}, \mathcal{R}, \mathcal{A}\Big)$
```

In the `Triangularize` algorithm for solving systems of polynomial equations by triangular decomposition, the *Regular GCD* subroutine relies on this technique for improved performance; see Section 2.1 for more details.

Speculative and Caching Schemes in $\mathbb{Z}[y]$

For polynomials $a, b \in \mathbb{Z}[y]$ with integer coefficients, a modular algorithm can be achieved by utilizing the Chinese remainder theorem (CRT). In this approach, we use Algorithms 7 and 8 for a prime field $\mathbf{k}$. We define $\mathbb{Z}_p[y]$ as the ring of univariate polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, for some prime p.

Further, we use an iterative and probabilistic approach to CRT from [75]. We iteratively calculate subresultants modulo different primes $p_0, p_1, \ldots$, continuing to add modular images to the CRT direct product $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_i}$ for $i \in \mathbb{N}$ until the reconstruc-

Algorithm 9 SUBRESULTANT$(a, b, \rho, \mathcal{Q}, \mathcal{R}, \mathcal{A})$

Require: $a, b \in \mathbf{k}[x] \setminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$, the list $\mathcal{Q}$ of all the quotients in the Euclidean sequence, the list $\mathcal{R}$ of the remainders that have been computed so far; we assume that $\mathcal{R}$ contains at least $r_\mu, \ldots r_{\ell-1}, r_\ell$ with $0 \leq \mu \leq \ell - 1$, and the list $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence

Ensure: $S_{n_v}(a, b)$, $S_{n_{v+1}}(a, b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$; the list $\mathcal{R}$ of the remainders is updated in-place

```
1: deduce $\left(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_\ell = \deg(r_\ell)\right)$ from $a, b$ and $\mathcal{Q}$
2: if $n_\ell \leq \rho$ then  $v := \ell$
3: else find $0 \leq v < \ell$ such that $n_{v+1} \leq \rho < n_v$.
4: if $v = 0$ then
5:     return $\left(a, \mathrm{lc}(b)^{m-n-1} b\right)$
6: for $i$ from $\max(v, \mu + 1)$ down to $v$ do
7:     $r_i := r_{i+1} q_{i+1} + r_{i+2}$; $\mathcal{R} := \mathcal{R} \cup (r_i)$
8: compute $S_{n_v}, S_{n_{v+1}}$ using Proposition 6 from $r_v, r_{v+1}$
9: return $\left(S_{n_v}, S_{n_{v+1}}\right)$
```

tion *stabilizes*. That is to say, the reconstruction does not change from $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_{i-1}}$ to $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_i}$.

Let p_0, p_1 be two distinct primes. The CRT states that the residue class ring $\mathbb{Z}_{p_0 p_1} := \mathbb{Z}/(p_0 p_1)\mathbb{Z}$ is isomorphic to the direct product of the residue class rings $\mathbb{Z}_{p_0} := \mathbb{Z}/p_0\mathbb{Z}$ and $\mathbb{Z}_{p_1} := \mathbb{Z}/p_1\mathbb{Z}$, denoted as $\mathbb{Z}_{p_0} \otimes \mathbb{Z}_{p_1}$, and written $\mathbb{Z}_{p_0 p_1} \equiv \mathbb{Z}_{p_0} \otimes \mathbb{Z}_{p_1}$. A naïve generalization of CRT, applies for a set of distinct primes $\{p_0, \ldots, p_e\}$ and $e \in \mathbb{N}$, yielding the following:

$$\mathbb{Z}_{p_0 p_1 \cdots p_e} \equiv \mathbb{Z}_{p_0} \otimes \mathbb{Z}_{p_1} \otimes \cdots \otimes \mathbb{Z}_{p_e}.$$

In a naïve approach, $S_k(a \bmod p_i, b \bmod p_i)$ modulo $\mathbb{Z}_{p_i}[y]$ are computed for a set of distinct large primes $\{p_0, \ldots, p_e\}$ of size $e + 1$. Then, CRT algorithm yields $S_k(a, b)$ modulo $\mathbb{Z}_{p_0}[y] \otimes \cdots \otimes \mathbb{Z}_{p_e}[y]$. For a large enough e, results of the CRT algorithm get stabilized and for such e, we can deduce $S_k(a, b)$ in $\mathbb{Z}[y]$.

To determine the number of primes e which is required in CRT, we need to determine an upper-bound for the maximum size of coefficients of $\mathrm{res}(a, b)$. This estimation is usually challenging and not practical. Thus, we make use of an optimized algorithm from [75] to perform CRT iteratively and probabilistically. In this approach, we continue adding modular images to the CRT direct product until the reconstruction *stabilizes*.

Speculative and Caching Schemes in $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x, y]$

We further exploit this technique to compute subresultants of bivariate polynomials over prime fields and the integers. Let $a, b \in \mathbb{B}[y]$ be polynomials with coefficients in $\mathbb{B} = \mathbb{Z}_p[x]$, thus $\mathbb{B}[y] = \mathbb{Z}_p[x, y]$, where the main variable is y and $p \in \mathbb{N}$ is an odd prime.

A desirable subresultant algorithm then uses an evaluation-interpolation scheme and the aforementioned univariate routines to compute subresultants of univariate images of a, b in $\mathbb{Z}_p[y]$ and then interpolates back to obtain subresultants in $\mathbb{Z}_p[x, y]$. This approach is well-studied in [75] to compute the resultant of bivariate polynomials. We can use the same technique to compute the entire subresultant chain, or even particular subresultants speculatively through Algorithms 7 and 8.

We begin by choosing a set of evaluation points of size $N \in \mathbb{N}$ and evaluate each coefficient of $a, b \in \mathbb{Z}_p[x, y]$ with respect to the main variable (y). Then, we call the subresultant algorithm to compute subresultants images in $\mathbb{Z}_p[y]$. Finally, we can retrieve the bivariate subresultants by interpolating each coefficient of each subresultant from the images. The number of evaluation points is determined from an upper-bound on the degree of subresultants and resultants with respect to x. From Theorem 7, we have:

$$N \geq \deg(b, y) \deg(a, x) + \deg(a, y) \deg(b, x) + 1.$$

For bivariate polynomials with integer coefficients, we can use the CRT algorithm in a similar manner to that which has already been reviewed for univariate polynomials over $\mathbb{Z}$. Figure 3.1 demonstrates this procedure for two polynomials $a, b \in \mathbb{Z}[x, y]$. In this commutative diagram, $\bar{a}, \bar{b}$ represent the modular images of the polynomials a, b modulo prime p_i for $0 \leq i \leq e$.

$$\begin{tikzcd}
a, b \in \mathbb{Z}[x, y] \arrow[r, "\text{Algorithm 1}"] \arrow[d, "\text{modulo } p_0, p_1, \ldots, p_i"] & \mathrm{subres}(a, b, y) \in \mathbb{Z}[x, y] \\
\bar{a}, \bar{b} \in \mathbb{Z}_{p_i}[x, y] \arrow[d, "\text{Evaluate at } t_0, \ldots, t_N"] & \mathrm{subres}(\bar{a}, \bar{b}, y) \in \mathbb{Z}_{p_i}[x, y] \arrow[u, "CRT"'] \\
\bar{a}(x, y)|_{x=t_i}, \bar{b}(x, y)|_{x=t_i} \in \mathbb{Z}_{p_i}[y] \arrow[r, "\text{Algorithm 8}"] & \mathrm{subres}(\bar{a}(x, y)|_{x=t_i}, \bar{b}(x, y)|_{x=t_i}, y) \in \mathbb{Z}_{p_i}[y] \arrow[u, "\text{Interpolate at } x"']
\end{tikzcd}$$

Figure 3.1: Computing the subresultant chain of $a, b \in \mathbb{Z}[x, y]$ using modular arithmetic, evaluation-interpolation and CRT algorithms where $(t_0, \ldots, t_N)$ is the list of evaluation points, $(p_0, \ldots, p_i)$ is the list of distinct primes, $\bar{a} = a \bmod p_i$, and $\bar{b} = b \bmod p_i$.

In practice, as the number of variables increases, the use of dense evaluation-interpolation schemes become less effective, since degree bound estimates become less sharp. In fact, sparse evaluation-interpolation schemes become more attractive [104, 78], and we will consider them in future works.

3.5 Parallel Subresultant Algorithms

As we have seen in Section 2.1, the computation of subresultant chains is essential in order to the *Regular GCD* (Definition 5) from Theorem 1 and consequently to all the core subroutines of the `Triangularize` algorithm.

In practice, the computation of subresultant chains can become a bottleneck when the coefficient sizes and the degrees of the input polynomials become larger and larger. Parallelizing this computation is a way to use more computing resources (in particular cache memories and CPU), which can have a significantly positive impact on the performance of the client procedures.

This parallelization is more *fine-grained* than the tasks of `Triangularize` as the amount of work for each worker (task) is low, while the total number of tasks may be high. Nonetheless, it can sometimes be the most computationally expensive subroutine and thus should not be ignored as a candidate for parallelization.

To illustrate how this can be done, we consider the computation of $\mathrm{subres}(a, b)$ for $a, b \in \mathbb{B}[y]$, with $\mathbb{B} = \mathbb{Z}[x]$. Related strategies, for the case where $\mathbb{B}$ is a ring of multivariate polynomials, are presented in [83].

Parallel Scheme based on Classical Evaluation-Interpolation

Algorithm 10 presents the evaluation-interpolation approach to compute the entire subresultant chain that is illustrated in Figure 3.1. Here, we discuss the parallel opportunities and techniques in this algorithm.

From [75], there is an upper bound h for the coefficient size of $\mathrm{subres}(a, b)$ with $a, b \in \mathbb{Z}[x, y]$. Thus, Algorithm 10 terminates after finitely many iterations of its while-loop at Lines 2 to 17. However, the *stabilization* condition, that is, the test at Line 15, may break this while-loop earlier. The equality $C^* = C$ means that $C[k]$ equals $C^*[k]$ for every $0 \leq k < \deg(b, y)$, that is, the two subresultant chains (computed modulo M and modulo Mp, respectively) are equal.

At Line 3, the function NextGoodPrime generates a stream of distinct primes that

Algorithm 10 BivariateSRC(a, b)

```
Require: $a, b \in \mathbb{Z}[x, y]$ with $m = \deg(a, y)$, $n = \deg(b, y)$, $m \geq n$, $h \in \mathbb{N}$ be a coefficient
    bound for subres$(a, b)$, and $d = \max(\deg(a, x), \deg(b, x))$.
Ensure: subres$(a, b) = (S_{n-1}(a, b), \ldots, S_0(a, b))$ in $\mathbb{Z}[x, y]$.
 1: $N := n \deg(a, x) + m \deg(b, x)$; $M := 1$; $C^* := 0$
 2: while $M \leq 2h$ do
 3:     $p :=$ NextGoodPrime$(a, b)$;
 4:     $\bar{a} := a \bmod p$; $\bar{b} := b \bmod p$
 5:     parallel_for $j$ from $0$ to $N$ do
 6:         Compute distinct non-zero evaluation point $e_j \bmod p$
              such that $\text{init}(a)|_{x=e_j} \neq 0 \bmod p$ and $\text{init}(b)|_{x=e_j} \neq 0 \bmod p$
 7:         $A_j :=$ subres$(\bar{a}|_{x=e_j}, \bar{b}|_{x=e_j})$
 8:     parallel_for $k$ from $0$ to $n-1$ do
 9:         $r := min(N, (m+n-2k)d)$
10:         $B[k] :=$ Interpolate$([A_0[k], \ldots, A_r[k]], [e_0, \ldots, e_r])$
11:     if $M = 1$ then $C := B$;
12:     else
13:         parallel_for $k$ from $0$ to $n-1$ do
14:             $C[k] :=$ CRT$(C^*[k], M, B[k], p)$
15:     if $C = C^*$ then break
16:     $M := Mp$; $C^* := C$
17: return $C^*$
```

are *good* for a, b, that is, not cancelling their leading coefficients. For a given good prime p, the parallel for-loop at Lines 5 to 7 computes images of subres$(a \bmod p, b \bmod p)$ by evaluating x at appropriate values.

With the parallel for-loop at Lines 8 to 10, those images are interpolated yielding subres$(a \bmod p, b \bmod p)$. Note that, for every $0 \leq k < \deg(b)$, we interpolate the k-th subresultant of $\bar{a}, \bar{b} \in \mathbb{Z}_p[x, y]$ from the first $r+1$ images of $S_k(\bar{a}|_{x=e_j}, \bar{b}|_{x=e_j})$ for $0 \leq j \leq r$ where $r = \min(N, N')$, $N = n \deg(a, x) + m \deg(b, x)$, $N' = (m + n - 2k)d$, and $d = \max(\deg(a, x), \deg(b, x))$. Indeed, it is shown in Theorem 8 that $\deg(S_k(a, b), x) \leq N'$, and more precisely, $\deg(S_0(a, b), x) \leq N$ (Theorem 7).

Algorithm 11 FastBivariateSRC(a, b)

```
Require: $a, b \in \mathbb{Z}[x, y]$ with $m = \deg(a, y)$, $n = \deg(b, y)$, $m \geq n$, $h \in \mathbb{N}$ be a coefficient
    bound for subres$(a, b)$, and $d = \max(\deg(a, x), \deg(b, x))$.
Ensure: subres$(a, b) = (S_{n-1}(a, b), \ldots, S_0(a, b))$ in $\mathbb{Z}[x, y]$.
 1: $N$ := smallest power of $2 > n\deg(a, x) + m\deg(b, x)$; $M := 1$; $C^* := 0$
 2: while $M \leq 2h$ do
 3:     $p :=$ NextGoodPrime$(a, b)$;
 4:     $\bar{a} :=$ ZeroPadding$(a \bmod p, m + 1, N)$
 5:     $\bar{b} :=$ ZeroPadding$(b \bmod p, n + 1, N)$
 6:     Compute a $N$-th root of unity $\omega$ mod $p$ such that
          init$(a)|_{x=\omega^i}$ and init$(b)|_{x=\omega^i}$ are non-zero modulo $p$ for all $0 \leq i < N$.
 7:     parallel_for $j$ from $0$ to $m$ do
 8:         $\alpha_j :=$ FFT(coeff$(a, j, y), \omega, N, p)$
 9:     parallel_for $j$ from $0$ to $n$ do
10:         $\beta_j :=$ FFT(coeff$(b, j, y), \omega, N, p)$
11:     $\alpha :=$ Transpose$(\alpha)$; $\beta :=$ Transpose$(\beta)$
12:     parallel_for $j$ from $0$ to $N - 1$ do
13:         $A_j :=$ subres$(\alpha_j, \beta_j)$
14:     $A^t :=$ Transpose3D$(A)$
15:     parallel_for $k$ from $0$ to $n - 1$ do
16:         $B[k] := \frac{1}{N}$FFT$([A^t_0[k], \ldots, A^t_{N-1}[k]], \omega^{-1}, N, p)$
17:     if $M = 1$ then $C := B$
18:     else
19:         parallel_for $k$ from $0$ to $n - 1$ do
20:             $C[k] :=$ CRT$(C^*[k], M, B[k], p)$
21:     if $C = C^*$ then break
22:     $M := Mp$; $C^* := C$
23: return $C^*$
```

With the parallel for-loop at Lines 13-14, CRT is applied to deduce,

$$\text{subres}(a \bmod Mp, b \bmod Mp),$$

from

$$\text{subres}(a \bmod M, b \bmod M) \text{ and } \text{subres}(a \bmod p, b \bmod p).$$

For these three parallel-loops, the work can divided evenly between threads. This is, in fact, an instance of the *map* pattern discussed in Section 2.3. Therefore, we consider the following three parallel opportunities for Algorithm 10 where p is a good prime:

(i) **Evaluate** a, b at x and compute the univariate subresultant chains images in $\mathbb{Z}_p[y]$;

(ii) **Interpolate** subresultants in $\mathbb{Z}_p[x, y]$ from the univariate images in $\mathbb{Z}_p[y]$; and

(iii) **Combine** the interpolated subresultant chain in $\mathbb{Z}_p[x, y]$ and the CRT result of the previous iteration.

Parallel Scheme based on FFT-based Evaluation-Interpolation

Algorithm 11 is a variant of Algorithm 10 where the evaluation and interpolation steps are performed via FFT techniques. When the coefficient bound h and the degrees n, m, d are large enough, this FFT-based approach substantially reduce the amount of work (algebraic complexity) without reducing the opportunities for concurrency. However, it increases memory consumption (as zero-padding is needed, see Lines 4 and 5, in order to apply FFT) and require careful memory manipulation (like data transposition, see Lines 11 and 14) in order to reduce the number of *cache misses*.

Since a three-dimensional transposition could have different definitions depending on the context, we specify that used at Line 14. Given a 3-dim array A of format $N \times n \times N$, the Transpose3D(A) returns a 3-dim array A^t of format $n \times N \times N$ such that every element $A[j][k][i]$ is mapped to $A^t[k][i][j]$, for $0 \le j < N$, $0 \le k < n$, $0 \le i < N$. The two-dimensional transposition (at Line 11) and the three-dimensional transposition (at Line 13) can efficiently be done in parallel fashion. This is not necessary in Algorithm 11, however, since the contributions of those transpositions on the critical path are negligible.

Returning to the idea of speculative computation of subresultants, discussed at the previous section, one can easily modify Algorithm 10 and Algorithm 11 so that they compute a given pair of consecutive non-zero subresultants from subres(a, b) rather than computing the entire chain.

In the context of the *Regular GCD* algorithm, we use this speculative approach in order to compute $S_0(a, b), S_1(a, b)$. In the rare case where subresultants of index higher than 1 are needed, other pairs of consecutive non-zero subresultants from subres(a, b) are computed, one pair at a time.

3.6 Experimentation

In this section, we discuss the implementation and performance of our various subresultant algorithms and their underlying core routines. Our methods are implemented as part of the Basic Polynomial Algebra Subprograms (BPAS) library [8] and we compare their performance against the NTL library [93] and Maple 2020 [71]. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4, BPAS v1.791, GMP 6.1.2, and NTL 11.4.3, with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

The coefficients of univariate polynomials are stored in a C array containing all zero and non-zero terms in $\mathbb{Z}_p$ or $\mathbb{Z}$, while the bivariate polynomials are stored as a C array of univariate polynomials containing all zero and non-zero terms in $\mathbb{Z}_p[x]$ or $\mathbb{Z}[x]$. This polynomial view is known as a *dense* representation. We study a so-called *sparse* representation for multivariate polynomials in Chapter 4.

Moreover, we utilize the low-level routines of the GMP library for arbitrary-precision integers. That is broken into two distinct parts, which is called the *head* and the *tree* as described in [12]. The head contains a pointer to the tree and metadata about the tree, while the tree itself is what holds the numerical data. Thus, users only interact with the head.

Example 7 *Figure 3.2 demonstrates the polynomial $a = a_n y^n + \cdots + a_1 y + a_0 \in \mathbb{Z}[y]$ in* BPAS *where $a_0, a_1, \ldots, a_n$ are arbitrary-precision integers in GMP.*

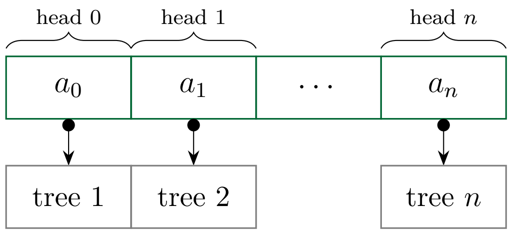


Figure 3.2: A one-dimensional C-array representing polynomial $a = \sum_{i=0}^{n} a_i y^i$ of $n + 1$ terms showing GMP trees and GMP heads as a_0, a_1, $\ldots$, a_n.

3.6.1 Routines in $\mathbb{Z}_p[y]$

We begin with foundational routines for arithmetic in finite fields and polynomials over finite fields. For basic arithmetic over a prime field $\mathbb{Z}_p$ where p is an odd prime, Montgomery multiplication, originally presented in [80], is used to speed up multiplication.

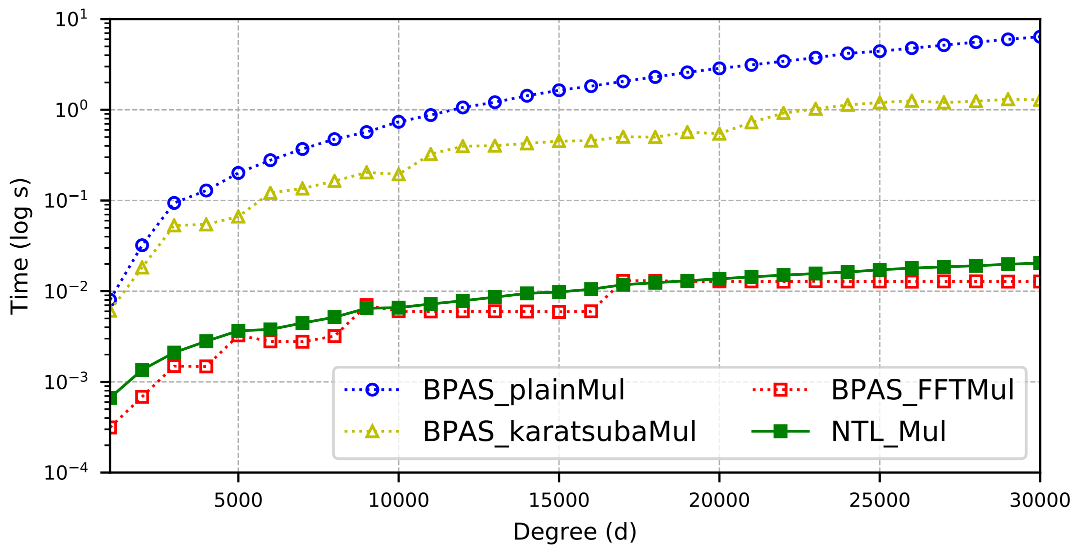


Figure 3.3: Comparing plain, Karatsuba, and FFT-based multiplication in BPAS with the wrapper `mul` method in NTL to compute ab for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$.

We have developed a dense representation of univariate polynomials which take advantage of Montgomery arithmetic (following the implementation in [30]) for prime fields with $p < 2^{64}$. Throughout this section we examine the performance of each operation for two randomly generated dense polynomials $a, b \in \mathbb{Z}_p$ with a 64-bit prime $p = 4179340454199820289$. Figures 3.3–3.6 examine, respectively, multiplication, division, GCD, and subresultant chain operations. These plots compare the various implementations within BPAS against NTL.

Our multiplication in $\mathbb{Z}_p[y]$ dynamically chooses the appropriate algorithm based on the input polynomials: plain or Karatsuba algorithms (following the routines in [100, Chapter 8]), or multiplication based on fast Fourier transform (FFT). The implementation of FFT itself follows that which was introduced in [31]. Figure 3.3 shows the performance of these routines in BPAS against a similar “wrapper” multiplication routine in NTL. From empirical data, our wrapper multiplication function calls the appropriate implementation of multiplication as follows. For polynomials a, b in $\mathbb{Z}_p[y]$, with $p < 2^{63}$, the plain algorithm is called when $s := \min(\deg(a), \deg(b)) < 200$ and the Karatsuba algorithm is called when $s \geq 200$. For 64-bit primes ($p > 2^{63}$), plain and Karatsuba algorithms are called when $s < 10$ and $s < 40$, respectively, otherwise FFT-based multi-

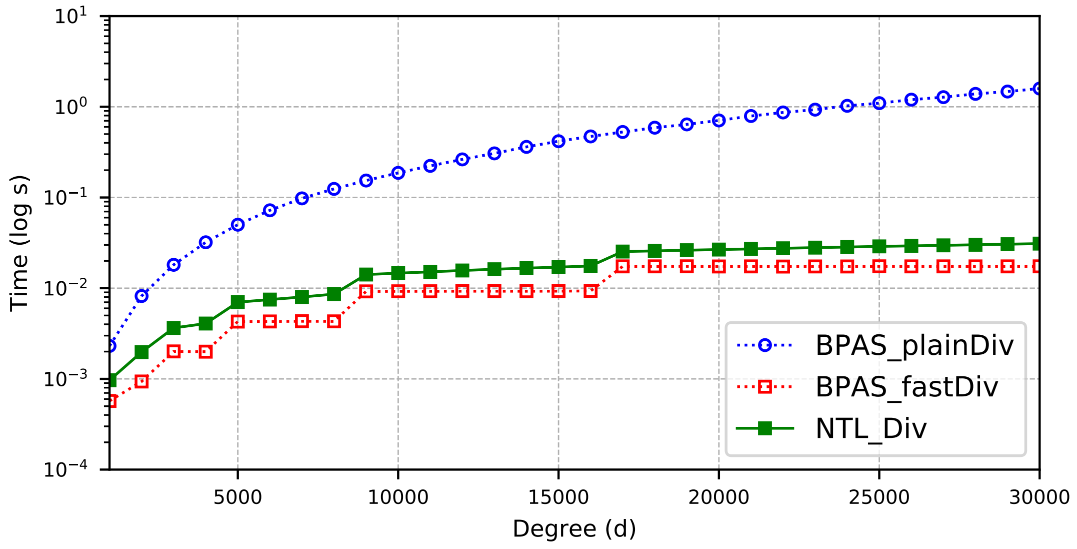


Figure 3.4: Comparing Euclidean and fast division algorithms in BPAS with the division method in NTL to compute $\mathrm{rem}(a, b)$ and $\mathrm{quo}(a, b)$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = 2(\deg(b) - 1) = d$.

plication is performed.

The division operation is again a wrapper function, dynamically choosing between Euclidean (plain) and fast division algorithms. The fast algorithm is an optimized power series inversion procedure that is firstly implemented in ALDOR [40] using the so-called middle-product trick. Figure 3.4 shows the performance of these two algorithms in comparison with the NTL division in $\mathbb{Z}_p[y]$. For polynomials a, b in $\mathbb{Z}_p[y]$, b the divisor, empirical data again guides the choice of appropriate implementation. Plain division is called for primes $p < 2^{63}$ and $\deg(b) < 1000$. However, for 64-bit primes, the plain algorithm is used when $\deg(b) < 100$, otherwise fast division supported by FFT is used.

Our GCD operation in $\mathbb{Z}_p[y]$ had two implementations: the classical extended Euclidean algorithm (EEA) and the Half-GCD (fast EEA) algorithm, respectively following the pseudo-codes in [100, Chapter 11] and the implementation in the NTL library [93]. Figure 3.5 shows the performance of these two approaches named `BPAS_plainGCD` and `BPAS_fastGCD`, respectively, in comparison with the NTL GCD algorithm for polynomials $a, b \in \mathbb{Z}_p[y]$ where $\gcd(a, b) = 1$.

To analyze the performance of our subresultant implementations, we compare the naïve EEA algorithm with the modular subresultant chain and the speculative subre-

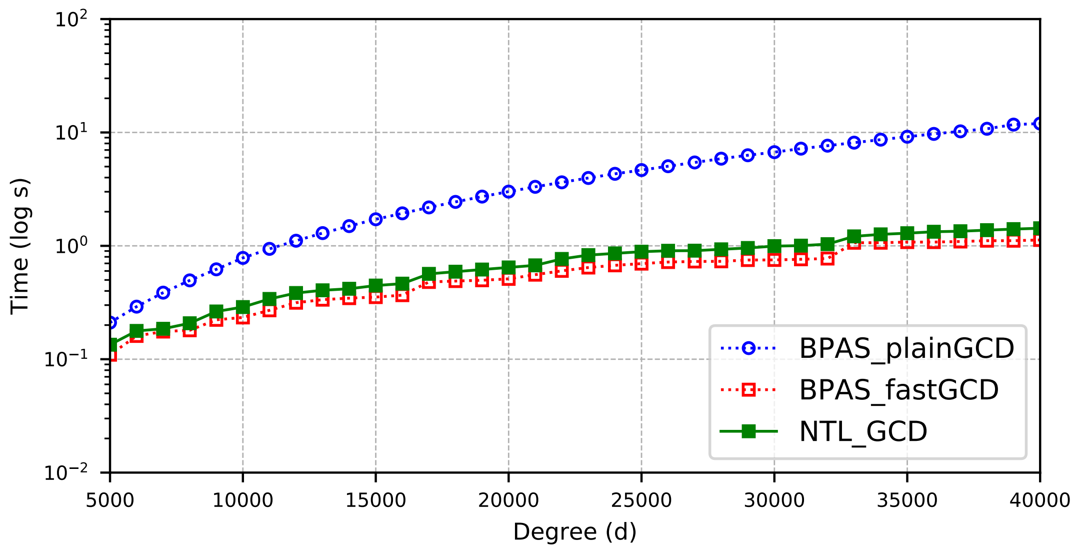


Figure 3.5: Comparing Euclidean-based GCD and Half-GCD-based GCD algorithms in BPAS with the GCD algorithm in NTL to compute $\gcd(a, b) = 1$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$.

sultant algorithm for $\rho = 0, 2$ in Figure 3.6. As this figure shows, using Half-GCD to compute two successive subresultants S_1, S_0 for $\rho = 0$ is approximately $5\times$ faster than computing the entire chain, while calculating other subresultants, e.g. S_3, S_2 for $\rho = 2$ taking advantage of the *cached* information from the first call (for $\rho = 0$), is nearly instantaneous.

3.6.2 Subresultants in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$

Following our previous discussion of various schemes for subresultants, we have implemented several subresultant algorithms in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$. We have four families of implementations:

1. `BPAS_modSRC`, that computes the entire subresultant chain using Proposition 6 and the CRT algorithm (and evaluation-interpolation in $\mathbb{Z}[x, y]$);

2. `BPAS_specSRC`, that refers to Algorithms 8 and 9 to compute two successive subresultants using Half-GCD and caching techniques;

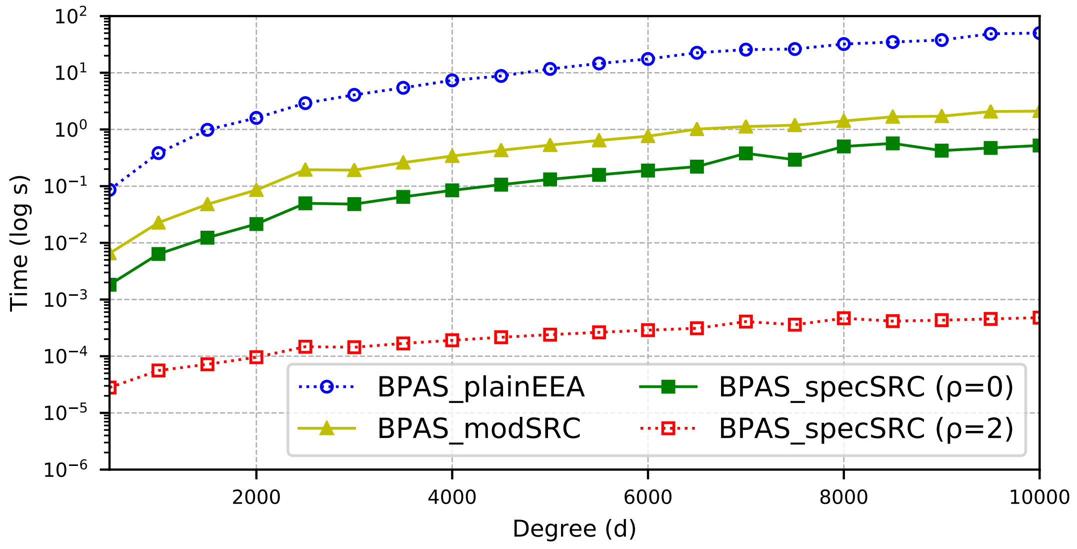


Figure 3.6: Comparing EEA, modular subresultant, and Half-GCD-based subresultant (`BPAS_specSRC`, $\rho = 0, 2$), in BPAS for dense polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$.

3. `BPAS_Ducos`, for Ducos' algorithm, based on Algorithm 18; and

4. `BPAS_OptDucos`, for Ducos' algorithm based on Algorithm 19.

Figure 3.7 compares the running time of those subresultant algorithms in $\mathbb{Z}[y]$ in the BPAS library and Maple. The modular approach is up to $5\times$ faster than the optimized Ducos' algorithm. Using speculative algorithms to compute only two successive subresultants yields a speed-up factor of 7 for $d = 2000$.

Figure 3.8 provides a favourable comparison between the family of subresultant algorithms in BPAS and the subresultant algorithm in Maple for dense bivariate polynomials $a, b \in \mathbb{Z}[x, y]$ where the main degree is fixed to 50, i.e. $\deg(a, y) = \deg(b, y) + 1 = 50$, and $\deg(a, x) = \deg(b, x) + 1 = d$ for $d \in \{10, 20, \ldots, 100\}$. Note that the `BPAS_specSRC` algorithm for $\rho = 0, 2, 4, 6$ is caching the information for the next call taking advantage of Algorithm 9.

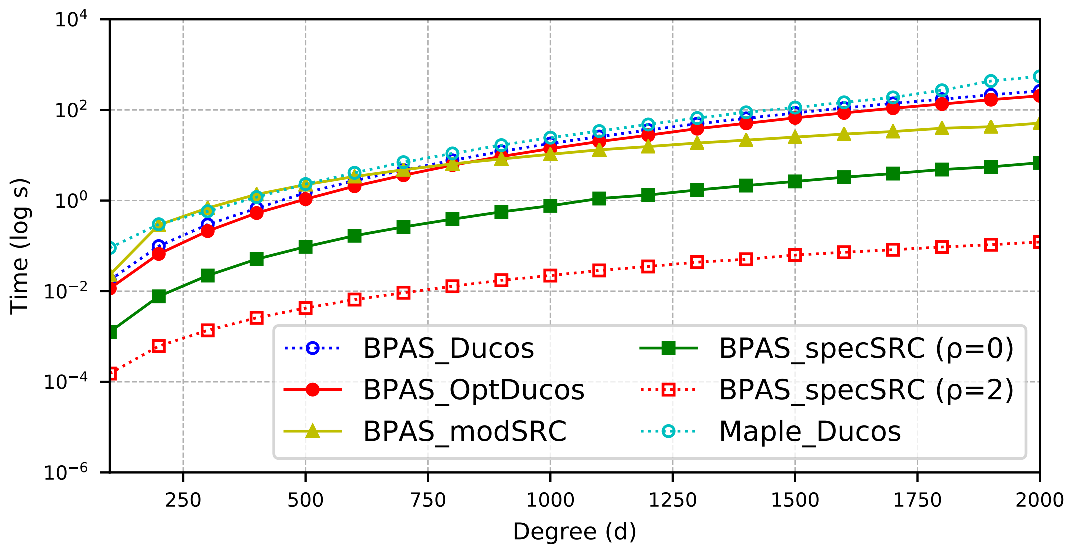
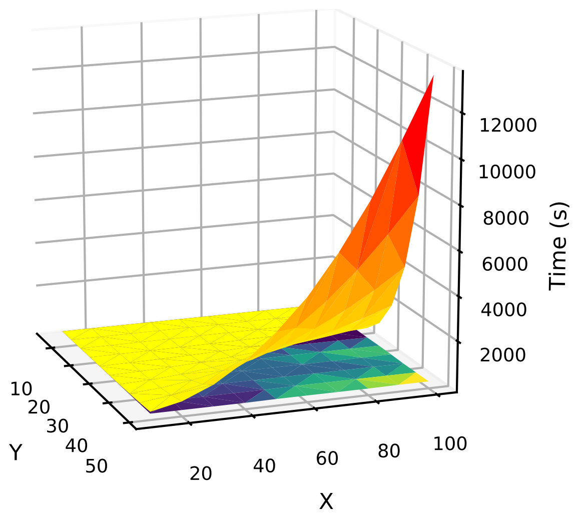


Figure 3.7: Comparing (optimized) Ducos' subresultant chain algorithm, modular subresultant chain, and speculative subresultant for $\rho = 0, 2$, algorithms in BPAS with Ducos' subresultant chain algorithm in MAPLE for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$.



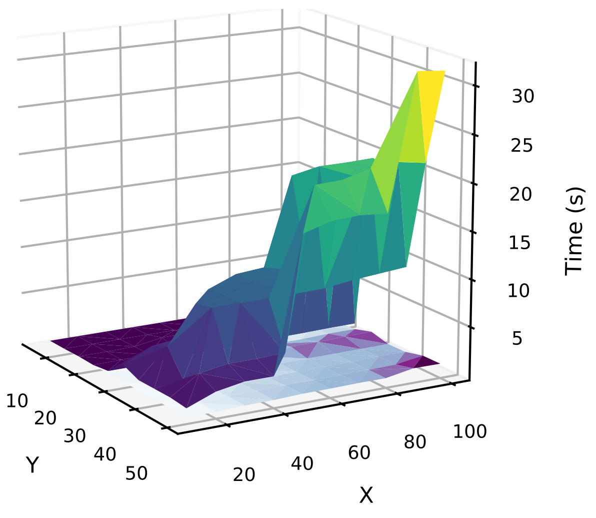


Figure 3.9: Comparing Opt. Ducos' algorithm (the top surface) and modular subresultant chain (the bottom surface) to compute the entire chain for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$.

Figure 3.10: Comparing modular subresultant chain using FFT (the top surface), and speculative subresultant ($\rho = 0$) (the bottom surface) for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$.

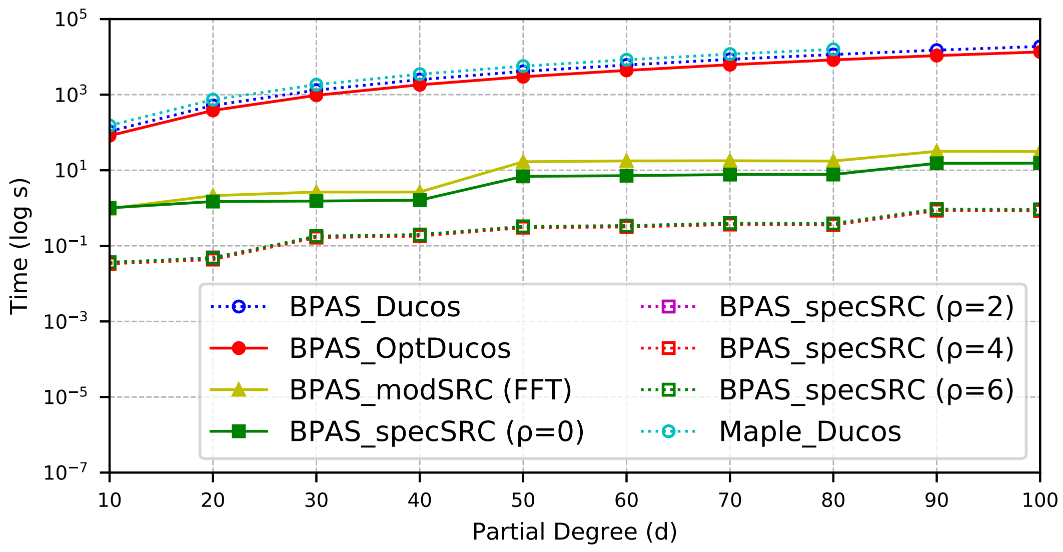


Figure 3.8: Comparing (optimized) Ducos' subresultant chain, modular subresultant chain, and speculative subresultant for $\rho = 0, 2, 4, 6$, in BPAS with Ducos' algorithm in Maple for dense polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = 50$ and $\deg(a, x) = \deg(b, x) + 1 = d$.

We next compare more closely the two main ways of computing an entire subresultant chain: the direct approach following Algorithm 1, and a modular approach using evaluation-interpolation and CRT (see Figure 3.1). Figure 3.9 shows the performance of the direct approach (the top surface), calling our memory-optimized Ducos' algorithm `BPAS_OptDucos`, in comparison with the modular approach (the bottom surface), calling `BPAS_modSRC`. Note that, in this figure, interpolation may be based on Lagrange interpolation or FFT algorithms depending on the degrees of the input polynomials.

Next, Figure 3.10 highlights the benefit of our speculative approach to compute the resultant and subresultant of index 1 compared to computing the entire chain. The FFT-based modular algorithm is presented as the top surface, while the speculative subresultant algorithm based on the Half-GCD is the bottom surface.

Lastly, we investigate the effects of different subresultant algorithms on the performance of the BPAS polynomial system solved based on triangular decomposition and regular chains; see [10, 27]. Tables 3.3, 3.1, and 3.2 investigate the performance of `BPAS_modSRC`, and `BPAS_specSRC` and the caching technique, for system solving.

Table 3.3 shows the running time of well-known and challenging bivariate systems,

Table 3.1: Comparing the execution time (in seconds) of subresultant implementations for constructed bivariate systems in Listing 3.1. Column headings follow Table 3.3, and `FFTBlockSize` is block size in the FFT-based evaluation and interpolation algorithms.

n	`ModSRC`	`SpecSRC`$_{\texttt{naïve}}$	`SpecSRC`$_{\texttt{cached}}$	deg(src[idx])	`Indexes`	`FFTBlockSize`
100	171.213	272.939	83.966	(0,50,100,150)	(0,51,101,150)	1024
110	280.952	370.628	117.106	(0,55,110,165)	(0,56,111,165)	1024
120	491.853	1035.810	331.601	(0,60,120,180)	(0,61,121,180)	2048
130	542.905	1119.720	362.631	(0,65,130,195)	(0,66,131,195)	2048
140	804.982	1445.000	470.649	(0,70,140,210)	(0,71,141,210)	2048
150	1250.700	1963.920	639.031	(0,75,150,225)	(0,76,151,225)	2048

where we have forced the solver to use only one particular subresultant algorithm. In `SpecSRC`$_{\texttt{naïve}}$, `BPAS_specSRC` does not cache data and thus does not reuse the sequence of quotients computed from previous calls. Among them, the caching ratio ($^{\texttt{SpecSRCnaïve}}/_{\texttt{SpecSRCcached}}$) of `vert_lines`, `L6_circles`, `ten_circles`, and `SA_2_4_eps` are 24.5, 21.6, 19.8, 9.2, respectively, while the speculative ratio ($^{\texttt{ModSRC}}/_{\texttt{SpecSRCcached}}$) of `tryme`, `mignotte_xy`, and `vert_lines` are 1.5, 1.2, and 1.2, respectively.

Tables 3.1 and 3.2 examine the performance of the polynomial system solver on constructed systems which aim to exploit the maximum speed-up of these new schemes. Listing 3.1 and 3.2 provide the MAPLE code to construct these input systems. For those systems created by Listing 3.1, we get 3× speed-up through caching the intermediate speculative data rather than repeatedly calling the Half-GCD algorithm for each subresultant call. Using `BPAS_specSRC` provides a 1.5× speed-up in using the `BPAS_modSRC` algorithm. Another family of constructed examples created by Listing 3.2 is evaluated in Table 3.2. Here, we get up to 3× speed-up with the use of cached data, and up to 2× speed-up in the modular method.

Figure 3.11 demonstrates the performance of parallel subresultant algorithms for bivariate polynomials over the integers using evaluation and interpolation operations. As discussed in Section 3.5, to implement these schemes, we took advantage of the *map pattern* mechanism in the BPAS multithreaded interface (Section 2.3) to parallelize *evaluation*, *interpolation*, and *combination* parts of the code. Our experimentation for a test suite of over 3000 polynomial systems, that are collected from user-data and bug reports of the *RegularChains* library and real-world applications, shows a parallel-speed-up factor of 4× on a 12-core machine [10].

Table 3.2: Comparing the execution time (in seconds) of subresultant algorithms for constructed bivariate systems in Listing 3.2. Column headings follow Table 3.1.

n	`ModSRC`	`SpecSRC`$_{\text{naïve}}$	`SpecSRC`$_{\text{cached}}$	deg(src[idx])	`Indexes`	`FFTBlockSize`
100	894.139	1467.510	474.241	(0,2,2)	(0,2,2)	512
110	1259.850	2076.920	675.806	(0,2,2)	(0,2,2)	512
120	1807.060	2757.390	963.547	(0,2,2)	(0,2,2)	512
130	2897.150	4311.990	1505.080	(0,2,2)	(0,2,2)	1024
140	4314.300	5881.640	2134.190	(0,2,2)	(0,2,2)	1024
150	5177.410	7869.700	2609.170	(0,2,2)	(0,2,2)	1024

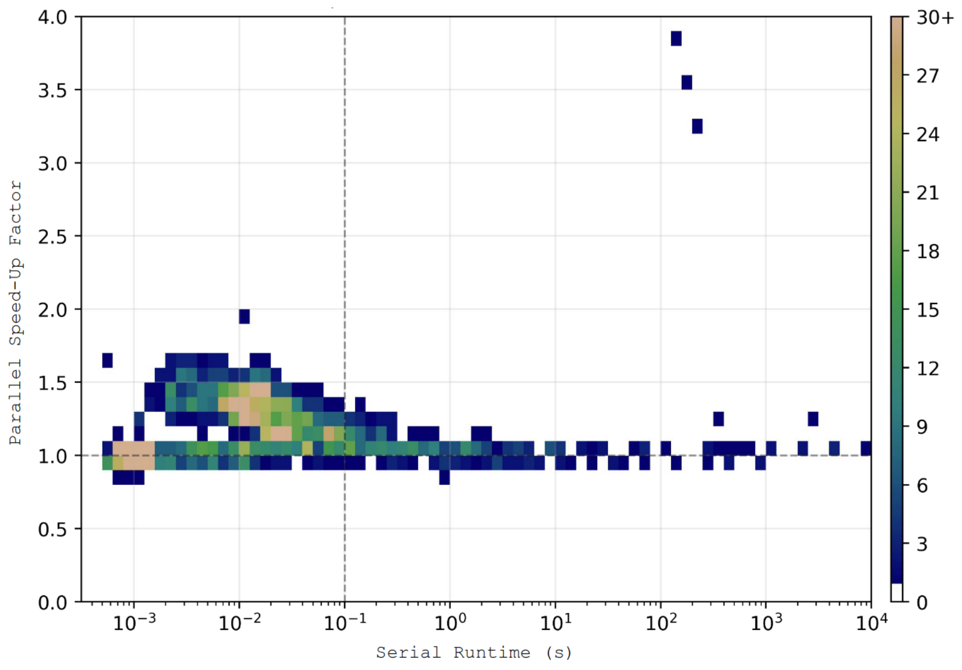


Figure 3.11: The parallel-speed-up of the `Triangularize` algorithm in BPAS when we use parallel subresultant algorithms (Section 3.5) for a test suite of more than 3000 polynomial systems.

Table 3.3: Comparing the execution time (in seconds) of subresultant algorithms on the BPAS `Triangularize` solver for well-known bivariate systems in the literature. deg(src[idx]) shows a list of minimum main degrees of the computed subresultants in each subresultant call and `Indexes` indicates a list of requested subresultant indexes.

SysName	`ModSRC`	`SpecSRC`$_{\text{naïve}}$	`SpecSRC`$_{\text{cached}}$	`OptDucos`	deg(src[idx])	`Indexes`
13_sings_9	3.416	3.465	3.408	3.417	(1)	(0)
compact_surf	11.257	26.702	10.26	10.258	(0,2,4,6)	(0,3,5,6)
curve24	4.992	4.924	4.911	4.912	(0,0,1)	(0,0,0)
curve_issac	2.554	2.541	2.531	2.528	(0,0,1)	(0,0,0)
cusps_and_flexes	4.656	8.374	4.656	4.488	(0,...,2)	(0,...,2)
degree_6_surf	81.887	224.215	79.394	344.564	(0,2,4,4)	(0,2,4,4)
hard_one	48.359	197.283	47.213	175.847	(0,...,2)	(0,...,2)
huge_cusp	23.406	33.501	23.41	23.406	(0,2,2)	(0,2,2)
L6_circles	32.906	721.49	33.422	32.347	(0,...,6)	(0,...,6)
large_curves	65.353	64.07	63.018	366.432	(0,0,1,1)	(0,0,0,0)
mignotte_xy	348.406	288.214	287.248	462.432	(1)	(0)
SA_2_4_eps	4.141	37.937	4.122	4.123	(0,...,6)	(0,...,6)
SA_4_4_eps	222.825	584.318	216.065	197.816	(0,...,3)	(0,...,6)
spider	293.701	294.121	295.198	293.543	(0,0,1,1)	(0,0,0,0)
spiral29_24	647.469	643.88	644.379	643.414	(1)	(0)
ten_circles	3.255	56.655	2.862	2.116	(0,...,4)	(0,...,4)
tryme	3728.085	4038.539	2415.28	4893.04	(0,2)	(0,2)
vert_lines	1.217	24.956	1.02	1.021	(0,...,6)	(0,...,6)

MAPLE Code for Polynomial Systems

```
SystemGenerator1 := proc(n)
  local R := PolynomialRing([x,y]);
  local J := PolynomialIdeals:-Intersect(<x^2+1,xy+2>,
  <x^2+3,xy^floor(n/2)+floor(n/2)+1>);
  J := PolynomialIdeals:-Intersect(J, <x^2+3,xy^n+n+1>);
  local dec := Triangularize(Generators(J),R);
  dec := map(NormalizeRegularChain,dec,R);
  dec := EquiprojectableDecomposition([%[1][1],%[2][1]],R);
  return map(expand, Equations(op(dec),R));
end proc:
```

Listing 3.1: MAPLE code of constructed polynomials in Table 3.1.

```
SystemGenerator2 := proc(n)
  local R := PolynomialRing([x,y]);
  local f := randpoly([x],dense,coeffs=rand(-1..1),degree=n);
  local J := <f,xy+2>;
  J :=PolynomialIdeals:-Intersect(J,<x^2+2,(x^2+3x+1)y^2+3>);
  local dec := Triangularize(Generators(J),R);
  dec := map(NormalizeRegularChain,dec,R);
  dec := EquiprojectableDecomposition([%[1][1],%[2][1]],R);
  return map(expand,Equations(op(dec),R));
end proc:
```

Listing 3.2: MAPLE code of constructed polynomials in Table 3.2.

Chapter 4

Sparse Multivariate Polynomials

4.1 Introduction

In this chapter we begin with reviewing basic arithmetic including the pseudo-division operation for multivariate polynomials. Then, we study a natural application of division, that is so-called multi-divisor division (or *normal form*), and pseudo-division, that is so-called multi-divisor pseudo-division, algorithms. We further introduce an optimized recursive algorithm in order to compute the multi-divisor pseudo-division where the set of divisors is a strongly *normalized* triangular set.

These multi-divisor (pseudo-) divisions could be utilized in `Intersect` to skip the use of the *specialization property of subresultants* via reducing (or normalizing) the subresultants w.r.t. the regular chains computed so far in the `Triangularize` algorithm. This technique is discussed in [72].

Subresultant algorithms for polynomials with more than 2 variables are crucial for developing a multivariate polynomial system solver via the `Triangularize` algorithm (Section 2.1). Therefore, there are subresultant algorithms for multivariate polynomials in almost all computer algebra software.

Most notably, the *RegularChains* library [64] in Maple provides three different algorithms to compute the entire chain based on Ducos' optimization [36], Bézout matrix [3], or evaluation-interpolation based on FFT. Each one is well-suited for a particular type of input polynomials w.r.t the number of variables and the coefficients ring; see the Maple help page for `SubresultantChain` command.

Similarly, the Algebramix library in Mathemagix [98] implements different subresultant algorithms, including routines based on evaluation-interpolation, Ducos' algorithm, and an enhanced version of Lickteig-Roy's algorithm [61].

In this chapter, we discuss a well-studied scheme known as Ducos' algorithm to compute the entire subresultant chain using a *recursive* and *sparse* representation of multivariate polynomials with taking advantage of a heap-based pseudo-division.

We further optimized this algorithm in the sense of performing efficient memory access patterns and utilizing in-place arithmetic that remarkably deducts the total memory usage. This so-called *optimized* Ducos' algorithm is discussed in Section 4.4.

In this approach we are no longer using fast modular arithmetic, as inputs are often substantially sparse and the complexity of utilizing a dense representation for multivariate polynomials is much more costly than the potential performance that modular routines could carry out. Moreover, the aforementioned algorithms for evaluation and interpolation of dense polynomials are not practical for sparse polynomials anymore.

Co-Authorship Statement

The data-structure and basic arithmetic are implemented by Brandt [11]. The normal form, multi-divisor pseudo-division, and (optimized) Ducos' subresultant chain algorithms are implemented by Asadi [11]. Theorem 12 is proved by Asadi and Moir [12] under the supervision of Prof. Marc Moreno Maza.

4.2 Basic Arithmetic

We have studied univariate and bivariate polynomials in the previous chapter. A common feature between those two categories of polynomials is that we represent them as *dense*, and so store them in memory with all their zero and non-zero terms. This representation has benefits including performing fast univariate and bivariate arithmetic such as dense evaluation-interpolation operations (Section 3.3). However, this is not practical neither for an increasing number of variable nor polynomials with many zero terms (i.e. *sparse* polynomials).

In this section, we study a *sparse* representation of polynomials with an arbitrary number of variables over the integers. We follow definitions and pseudo-codes from [12].

Example 8 *Let $a = x^{100} - 1 \in \mathbb{Z}[x]$. This is a sparse representation as zero terms of polynomial a are eliminated while a dense representation of a would be:*

$$a = \sum_{i=0}^{100} a_i x^i,$$

with $a_0 = -1,\ a_i = 0$ *for all* $0 < i < 100$*, and* $a_{100} = 1$*. In addition, this is considered as a sparse polynomial because of its many zero terms.*

Let $\mathbb{B}$ be a commutative ring with unity 1. We define $\mathbb{B}[X] = \mathbb{B}[x_1, \ldots x_v]$. A distributed polynomial $a \in \mathbb{B}[X]$ with variables $X = (x_1, \ldots, x_v)$ is indicated as:

$$a = \sum_{i=1}^{n_a} A_i = \sum_{i=1}^{n_a} a_i X^{\alpha_i},$$

where n_a is the number of non-zero terms, $A_i = a_i X^{\alpha_i}$ is a term, $a_i \in \mathbb{B}$, and α_i is an exponent vector for the variables X. In this section, we make use of variable ordering:

$$x_1 < x_2 < \cdots < x_v,$$

and a *lexicographical* term order so that the terms are ordered decreasingly for multivariate polynomials; see Section 4.3 for more details on the *lexicographical* ordering.

Further, We can represent polynomials in another practical way using a so-called *recursive* view. Let $a \in \mathbb{B}[x_1, \ldots, x_v]$, we can convert a to a univariate polynomial based on its main variable, $x_v = \text{mvar}(a)$ in $\tilde{\mathbb{B}}[x_v]$, where $\tilde{\mathbb{B}} = \mathbb{B}[x_1, \ldots, x_{v-1}]$. In this chapter, we say that a is in $\mathbb{B}[x_1, \ldots, x_{v-1}][x_v]$, if a is represented (and stored in the memory) in the *recursive* view.

In this section, we review basic arithmetic including *addition*, *multiplication*, *division*, and *pseudo-division* of multivariate polynomials. To study and implement these operations for polynomial in $\mathbb{B}[x_1, \ldots, x_v]$, we make use of both *distributed* and *recursive* representations in the BPAS library [12].

Addition and Subtraction Operations

Addition and subtraction of two polynomials simply performs by sorting the terms after combining those with identical exponents. To sort the terms of a polynomial, we use a merge-step based on merge sort [12, 51].

Multiplication and Division Operations

For multiplication and division, we make use of sparse polynomial arithmetic studied firstly in [51] and later extended in [76, 77]. An efficient multiplication algorithm for polynomials,

$$a = \sum_{i=0}^{n_a} a_i X^i, \text{ and } b = \sum_{i=0}^{n_b} b_i X^i,$$

in $\mathbb{B}[X]$ computes:

$$ab = \sum_{i=0}^{n_a} a_i b,$$

by merging all the partial products using a binary max-heap [92, Section 2.4]. We take advantage of the same idea to compute the division:

$$a/b = a - \sum_{i=0}^{k} q_i b,$$

where the heap elements multiply by q_i for $0 \leq i \leq k$ (which are terms of the quotient $q = \sum_{i=0}^{k} q_i X^i \in \mathbb{B}[X]$ and generate incrementally along with the divisor); see [12] for algorithms.

Pseudo-Division Operation

The *pseudo-division* algorithm is essentially a fraction-free division in $\mathbb{B}[x_1, \ldots, x_v][y]$. For two polynomials $a, b \in \mathbb{B}[X][y]$, this algorithm performs by multiplying a and $h := \mathrm{lc}(b) \in \mathbb{B}[X]$ rather than dividing a by h for each term of the quotient q. If $\deg(q) = k$, then this operation must satisfy:

$$h^{k+1} a = qb + r, \tag{4.1}$$

where $\mathrm{pquo}(a, b) := q \in \mathbb{B}[X][y]$ is known as the *pseudo-quotient* and $\mathrm{prem}(a, b) := r \in \mathbb{B}[X][y]$ is known as the *pseudo-remainder*. Algorithm 12 is adapted from the division algorithm and shows the naïve pseudo-division algorithm. Recall from Section 3.2 that $\mathrm{lt}(a)$ and $\mathrm{lc}(a)$ are denoted, respectively, as the leading term and leading coefficient of a.

Algorithm 12 NaïvePseudoDivide (a, b, y)

```
Require: $a, b \in \mathbb{B}[X][y]$, $\deg(b, y) > 0$
Ensure: $q, r \in \mathbb{B}[X][y]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r, y) < \deg(b, y)$.
1: $q := 0$; $r := 0$; $\ell := 0$; $h := \mathrm{lc}(b)$; $\gamma = \deg(b, y)$; $\tilde{r} := a$
2: while $\tilde{r} \neq 0$ do
3:     if $y^\gamma \mid \tilde{r}$ then
4:         $q := hq + \tilde{r}/y^\gamma$; $\ell := \ell + 1$
5:     else
6:         $r := r + \tilde{r}$
7:     $\tilde{r} := \mathrm{lt}(h^\ell a - qb - r)$
8: return $q, r, \ell$
```

In [11] and later in [12], a heap-optimization technique to compute the pseudo-division of two multivariate polynomials is introduced and developed in the BPAS library. The management of the heap in this algorithm is so that the computation of the products of ab requires:

- $heapInsert(a_i, b_j)$ that adds the product of a_i and b_j to the heap;
- $heapPeek()$ that gets the exponent vector ε of the top element in the heap; and
- $heapExtract()$ extracts from the heap a term of maximal degree from a product $a_i b_j$. The next term of the product $a_i b_j$ (if one exists) is then inserted back into the heap.

Algorithm 13 from [12] shows the pseudo-division algorithm based on a max-heap implemented in BPAS.

Algorithm 13 PSEUDODIVIDE (a, b, y)

```
Require: $a = \sum_{i=1}^{n_a} a_i y^{\alpha_i} = \sum_{i=1}^{n_a} A_i, b = \sum_{i=1}^{n_b} b_i y^{\alpha_i} = \sum_{i=1}^{n_b} B_i \in \mathbb{B}[X][y]$, with $b \neq 0$
Ensure: $q, r \in \mathbb{B}[X][y]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r, y) < \deg(b, y)$.
1: $q := 0$; $r := 0$; $\ell := 0$; $s := 0$; $k := 1$; $h := \mathrm{lc}(b)$; $\varepsilon := -1$; $\gamma := \deg(b, y)$
2: while $\varepsilon > -1$ or $k \leq n_a$ do
3:     if $\varepsilon < \deg(A_k, y)$ then
4:         $\tilde{r} := h^\ell A_k$
5:         $\eta := \deg(A_k, y)$; $k := k + 1$
6:     else if $\varepsilon = \deg(A_k, y)$ then
7:         $\tilde{r} := h^\ell A_k - heapExtract()$
8:         $\eta := \varepsilon$; $k := k + 1$
9:     else
10:        $\tilde{r} := -heapExtract()$
11:        $\eta := \varepsilon$
12:    if $\deg(b, y) \leq \eta$ then
13:        $\ell := \ell + 1$; $q := hq$
14:        $Q_\ell := \tilde{r}/y^\gamma$; $q := q + Q_\ell$
15:        $heapInsert(Q_\ell, B_2)$
16:    else
17:        $r := r + \tilde{r}$
18:    $\varepsilon := heapPeek()$
19: return $q, r, \ell$
```

Example 9 *Consider* $a = (x_1x_2 + x_2)y^3 + x_1,\ b = x_1x_2y + x_1x_2$ *in* $\mathbb{Z}[x_1, x_2][y]$*. The pseudo-division algorithm calculates:*

$$r := \text{prem}(a, b) = -x_1^4x_2^4 + x_1^4x_2^3 - x_1^3x_2^4,$$

and,

$$q := \text{pquo}(a, b) = x_1^3x_2^3y^2 - x_1^3x_2^3y + x_1^2x_2^3y^2 + x_1^3x_2^3 - x_1^2x_2^3y + x_1^2x_2^3,$$

so that, the equation $h^k a = qb + r$ *satisfies with* $h = x_1x_2$ *and* $k = 2$*.*

4.3 Multi-Divisor Division and Pseudo-Division

One natural application of multivariate division with remainder is the computation of normal forms with respect to a set of polynomials, a kind of multi-divisor division. In this section, we review the definition of the multi-divisor division algorithm, that we also call *normal form.* We study an efficient recursive algorithm to compute the normal form of triangular sets, and introduce a recursive multi-divisor pseudo-division for strongly *normalized* triangular sets.

For polynomials in $\mathbf{k}[X] := \mathbf{k}[x_1, \ldots, x_v]$, a *monomial order* is a relation $<$ on $\mathbb{N}^v$ such that $<$ is a total order, and if $\alpha < \beta$ then $\alpha + \gamma < \beta + \gamma$ for all $\alpha, \beta, \gamma \in \mathbb{N}^v$.

Definition 13 *A monomial order* $<$ *is a lexicographic order if for every* $\alpha, \beta \in \mathbb{N}^v$*,*

$$\alpha < \beta \iff \textit{the leftmost non-zero entry in } \alpha - \beta \textit{ is negative.}$$

Example 10 *For monomials in* $\mathbf{k}[x, y, z]$ *with Lexicographic order, we have:*

$$z^5 = x^0y^0z^5 \ < \ y^5 = x^0y^5z^0 \ < \ x^1y^1z^5 \ < \ x^1y^2z^1 \ < \ x^5 = x^5y^0z^0.$$

Throughout this thesis, we assume a *lexicographic* order for all polynomial ring. Our next goal is to define an algorithm for division with remainder in $\mathbf{k}[X]$ and for many divisors. Before presenting the pseudo-code, we review a couple of examples from [100] to illustrate important features of this algorithm.

Example 11 *Consider* $a = xy^2 + 1, b_1 = xy + 1, b_2 = y + 1$ *in* $\mathbb{Z}[x, y]$*, we have:*

	$xy+1$	$y+1$
xy^2+1	y	
$-(xy^2+y)$		
$-y+1$		*-1*
$-(-y-1)$		
2		

Here, the quotient in each step is recorded in the column below its divisor. In the last step, 2 is not divisible by the leading term of b_1 or b_2, and the process terminates. Hence,

$$a = (y)b_1 + (-1)b_2 + (2).$$

However, the multi-divisor division is not a determinism algorithm in the sense of the returned remainder. For instance, if we divide a by b_2 instead of b_1 at the first step, the final reduced result would be $x + 1$ instead of 2, and we have:

$$a = (0)b_1 + (xy - x)b_1 + (x + 1).$$

In the following example, we experience a situation that only occurs within multivariate cases. In the third step, the leading term x is not divisible by the leading term of b_1 or b_2 and is moved to the remainder column, and then, the division continues further.

Example 12 *Consider $a = x^2y + xy^2 + y^2, b_1 = xy - 1, b_2 = y^2 - 1$ in $\mathbb{Z}[x, y]$, we have:*

	$xy - 1$	$y^2 - 1$	*remainder*
$x^2y + xy^2 + y^2$	x		
$-(x^2y - x)$			
$xy^2 + x + y^2$	y		
$-(xy^2 - y)$			
$x + y^2 + y$			x
$-x$			
$y^2 + y$		1	
$-(y^2 - 1)$			
$y + 1$			$y + 1$

The result is $a = (x + y)b_1 + (1)b_2 + (x + y + 1)$.

4.3.1 Triangular Set Normal Form

Let $\mathbf{k}$ be a field. If $\mathcal{B} = (b_1, \ldots, b_k)$ is a set of polynomials where $b_j \in \mathbf{k}[x_1, \ldots, x_v]$ for $1 \leq i \leq k$, we can compute the normal form r of a polynomial $a \in \mathbf{k}[x_1, \ldots, x_v]$ (together with the quotients q_j) with respect to $\mathcal{B}$ by Algorithm 14, yielding,

$$a = q_1t_1 + \cdots + q_kt_k + r,$$

where r is reduced with respect to $\mathcal{B}$ so that no monomial in r is divisible by any of the leading terms of b_i for $1 \leq i \leq k$.

Algorithm 14 NormalForm (a, T)

```
Require: $a \in \mathbf{k}[x_1, \ldots, x_v]$, $\mathcal{B} = (b_1 \ldots, b_k) \subset \mathbf{k}[x_1, \ldots, x_v]$
Ensure: $\mathbf{q} = (q_1, \ldots, q_k) \subset \mathbf{k}[x_1, \ldots, x_v]$ and $r \in \mathbf{k}[x_1, \ldots, x_v]$ such that $a = q_1 b_1 + \cdots + q_k b_k + r$,
    where $r$ is reduced in the sense of no monomial in $r$ is divisible by any of the leading terms
    of $b_i$ for $1 \leq i \leq k$.
 1: $r := 0$; $p := a$
 2: $q_1 := 0$; $q_2 := 0$; $\cdots$; $q_k := 0$
 3: while $p \neq 0$ do
 4:     if $\mathrm{lt}(b_i) \mid \mathrm{lt}(p)$ for some $i \in \{1, \ldots, k\}$ then
 5:         $q_i := q_i + \frac{\mathrm{lt}(p)}{\mathrm{lt}(b_i)}$
 6:         $p := p - \frac{\mathrm{lt}(p)}{\mathrm{lt}(b_i)} b_i$
 7:     else
 8:         $r := r + \mathrm{lt}(p)$; $p := p - \mathrm{lt}(p)$
 9: end while
10: return $q_1, \ldots, q_k, r$
```

Theorem 11 *Algorithm 14 terminates and is correct.*

Proof. [100, Theorem 21.12]

This naïve normal form algorithm makes repeated calls to a multivariate division with remainder algorithm, thus we can take advantage of our optimized heap-based division to perform this procedure. Moreover, We can offer algorithmic improvements in some cases where the set of divisors forms a triangular set.

Definition 14 *A triangular set $T = \{t_1, \ldots, t_k\}$, with $t_j \in \mathbf{k}[x_1, \ldots, x_v]$ and,*

$$\mathrm{mvar}(t_1) < \cdots < \mathrm{mvar}(t_k),$$

is called normalized if, for every polynomial of T, every variable appearing in its initial is free, that is, is not the main variable of another polynomial of T.

In the case where a normalized triangular set is also *zero-dimensional*, i.e., $k = v$ so that being normalized implies that $\mathrm{init}(t_i) \in \mathbf{k}$ holds, the triangular set T is actually a so-called Gröbner basis for the ideal it generates.

For this case of zero-dimensional normalized (so-called Lazard) triangular sets one can use a recursive algorithm (Algorithm 15) which is taken from [65]. Since the algorithm is recursive we appropriately use the recursive representation of polynomials.

If $v = 1$, the result is obtained by simply applying normal division with remainder. Otherwise the coefficients of a with respect to $x_v = \mathrm{mvar}(t_v)$ are polynomials belonging to $\mathbf{k}[x_1, \ldots, x_{v-1}]$, because T is a triangular set. The coefficients of a are reduced with respect to the set $(t_1, t_2, \ldots, t_{v-1})$ by means of a recursive call, yielding a polynomial r. At this point, r is divided by t_v by applying the division algorithm. Since this operation can lead to an increase in degree of for the variables less than x_v, the coefficients of r are reduced with respect to $(t_1, \ldots, t_{v-1})$ by means of a second recursive call.

Algorithm 15 TriangularSetNormalForm (a, T)

```
Require: $a \in \mathbf{k}[x_1, \ldots, x_v]$, $T = (t_1, \ldots, t_v) \subset \mathbf{k}[x_1, \ldots, x_v]$, with $x_1 = \mathrm{mvar}(t_1) < \cdots < x_v = \mathrm{mvar}(t_v)$ and $\mathrm{init}(t_1), \ldots, \mathrm{init}(t_v) \in \mathbf{k}$
Ensure: $\mathbf{q} = (q_1, \ldots, q_v) \subset \mathbf{k}[x_1, \ldots, x_v]$ and $r \in \mathbf{k}[x_1, \ldots, x_v]$ such that $a = q_1 t_1 + \cdots + q_v t_v + r$, where $r$ is reduced (in the sense of Gröbner bases) with respect to the Lazard triangular set $T$
 1: if $v = 1$ then
 2:     $(q_1, r) :=$ Divide$(a, t_1)$
 3: else
 4:     for $i = 0$ to $\deg(a, x_v)$ do
 5:         $(\mathbf{q}^{(i)} := (q_1^{(i)}, \ldots, q_{v-1}^{(i)}), r^{(i)}) :=$
            TriangularSetNormalForm(coeff$(a, x_v, i), (t_1, \ldots, t_{v-1})$)
 6:     $\mathbf{q} := 0$
 7:     $r := \sum_i r^{(i)} {x_v}^i$
 8:     for $j = 1$ to $v - 1$ do
 9:         $q_j := q_j + \sum_i q_j^{(i)} {x_v}^i$
10:     $(\tilde{q}, r) :=$ Divide$(r, t_v)$; $q_v := q_v + \tilde{q}$
11:     for $i = 0$ to $\deg(r, x_v)$ do
12:         $(\mathbf{q}^{(i)} := (q_1^{(i)}, \ldots, q_{v-1}^{(i)}), r^{(i)}) :=$
            TriangularSetNormalForm(coeff$(r, x_v, i), (t_1, \ldots, t_{v-1})$)
13:     execute Lines 8-11
14: return $(\mathbf{q}, r)$
```

4.3.2 Triangular Set Pseudo-Division

This approach can be extended to pseudo-division of a polynomial by a triangular set, an operation that is important in triangular decomposition algorithms, in the case that the triangular set is normalized.

The pseudo-remainder r and pseudo-quotients q_j of a polynomial $a \in \mathbf{k}[x_1, \ldots, x_v]$ pseudo-divided by a triangular set $T = (t_1, \ldots, t_k)$ must satisfy:

$$ha = q_1 t_1 + \cdots + q_k t_k + r, \qquad \deg(r, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j)) \text{ for } 1 \le j \le k \quad (4.2)$$

where h is a product of powers of the initials (leading coefficients in the univariate sense) of the polynomials of T. If this condition is satisfied then r is said to be *reduced with respect to* T, again using the convention that $\deg(r) = -\infty$ if $r = 0$.

The pseudo-remainder r can be computed naïvely in k iterations where each iteration performs a single pseudo-division step with respect to each main variable in decreasing order $\mathrm{mvar}(t_k), \mathrm{mvar}(t_{k-1}), \ldots, \mathrm{mvar}(t_1)$. The remainder is initially set to a and is updated during each iteration.

This naïve algorithm is inefficient for two reasons:

(i) Since each pseudo-division step can increase the degree of lower variables in the order, if a is not already reduced with respect to T, the intermediate pseudo-remainders can experience significant coefficient swell; and

(ii) It is inefficient in terms of data locality because each pseudo-division step requires performing operations on data distributed throughout the polynomial.

A less naïve approach is a recursive algorithm that replaces each of the k pseudo-division steps in the naïve algorithm with a recursive call, amounting to k iterations where multiple pseudo-division operations are performed at each step. This algorithm deals with the first inefficiency issue of coefficient swell, but still runs into the issue with data locality. To perform this operation more efficiently we conceive a recursive algorithm based on the the recursive normal form algorithm (Algorithm 15). Using a recursive call for each coefficient of the input polynomial a ensures that we work only on data stored locally, handling the second inefficiency of the naïve algorithm.

Algorithm 16 TriangularSetPseudoDivide (a, T)

Require: $a, t_1, \ldots, t_k \in \mathbf{k}[x_1, \ldots, x_v]$, $T = (t_1, \ldots, t_k)$, with $\mathrm{mvar}(t_1) < \cdots < \mathrm{mvar}(t_k)$ and $\mathrm{init}(t_j) \notin \{\mathrm{mvar}(t_i) \mid t_i \in T\}$ for $1 \leq j \leq k$

Ensure: $\mathbf{q} = (q_1, \ldots, q_k) \subset \mathbf{k}[x_1, \ldots, x_v]$ and $r, h \in \mathbf{k}[x_1, \ldots, x_v]$ such that $ha = q_1 t_1 + \cdots + q_k t_k + r$, where r is reduced with respect to T

```
 1: if $k = 1$ then
 2:     $(q_1, r, e) :=$ PseudoDivide$(a, t_1)$; $h = \mathrm{init}(t_1)^e$
 3: else
 4:     $x_m := \mathrm{mvar}(t_k)$
 5:     for $i = 0$ to $\deg(a, x_m)$ do
 6:         $(\mathbf{q}^{(i)} := (q_1^{(i)}, \ldots, q_{k-1}^{(i)}), r^{(i)}, h^{(i)}) :=$
    TriangularSetPseudoDivide$(\mathrm{coeff}(a, x_m, i), (t_1, \ldots, t_{k-1}))$
 7:     $\mathbf{q} = 0$
 8:     $h_1 := \mathrm{lcm}(h^{(i)}), 0 \leq i \leq \deg(a, x_m)$
 9:     $r := \sum_i (h_1 / h^{(i)})\ r^{(i)} x_m^i$
10:     for $j = 1$ to $k - 1$ do
11:         $q_j := q_j + \sum_i (h_1 / h^{(i)})\ q_j^{(i)} x_m^i$
12:     if $\mathrm{mvar}(r) = x_m$ then
13:         $(\tilde{q}, r, \tilde{e}) :=$ PseudoDivide$(r, t_k)$
14:         $\tilde{h} = \mathrm{init}(t_k)^{\tilde{e}}$
15:         for $j = 1$ to $k - 1$ do
16:             $q_j := q_j \tilde{h}$
17:         $q_k := \tilde{q}$
18:         for $i = 0$ to $\deg(r, x_m)$ do
19:             $(\mathbf{q}^{(i)} := (q_1^{(i)}, \ldots, q_{k-1}^{(i)}), r^{(i)}, h^{(i)}) :=$
    TriangularSetPseudoDivide$(\mathrm{coeff}(r, x_m, i), (t_1, \ldots, t_{k-1}))$
20:         $h_2 := \mathrm{lcm}(h^{(i)}), 0 \leq i \leq \deg(r, x_m)$
21:         for $j = 1$ to $k$ do
22:             $q_j := q_j h_2$
23:         execute Lines 9-13 with $h_2$ replacing $h_1$
24:         $h := h_1 \tilde{h} h_2$
25:     else
26:         $h := h_1$; $q_k = 0$
27: return $(\mathbf{q}, r, h)$
```

Theorem 12 *Algorithm 16 terminates and is correct.*

Proof. The key difference between this algorithm and Algorithm 15 is the change from division to pseudo-division. By the correctness of Algorithm 13 the computed pseudo-remainders are reduced with respect to its divisor. The fact that the loops of recursive calls are all for a triangular set with one fewer variables ensures that the total number of recursive calls is finite and the algorithm terminates. If $k = 1$, then pseudo-division algorithm shows correctness of this algorithm, so assume that $k > 1$.

We must first show that lines 4 to 13 correctly reduce a with respect to the polynomials $(t_1, \ldots, t_{k-1})$. Let $c_i = \text{coeff}(a, x_m, i)$, so $a = \sum_{i=0}^{\deg(a,x_m)} c_i x_m^i$. Assuming the correctness of the algorithm, the result of these recursive calls are $q_j^{(i)}$, $r^{(i)}$ and $h^{(i)}$ such that:

$$h^{(i)} c_i = \sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)},$$

where $\deg(r^{(i)}, \text{mvar}(t_j)) < \deg(t_j, \text{mvar}(t_j))$ and $h^{(i)} = \prod_{j=1}^{k-1} \text{init}(t_j)^{e_j}$ for some non-negative integers e_j. It follows that $c_i = \left(\sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}\right) / h^{(i)}$. We seek a minimal h_1 such that:

$$h_1 a = \sum_i h_1 c_i x_m^i = \sum_i (h_1/h^{(i)}) \left(\sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}\right) x_m^i$$

is denominator-free, which is easily seen to be $\text{lcm}(h^{(i)})$. This then satisfies the required relation of the form (4.2), with h_1 in place of h, by taking $q_j = \sum_i (h_1/h^{(i)}) q_j^{(i)} t_j x_m^i$ and $r = \sum_i (h_1/h^{(i)}) r_j^{(i)} x_m^i$. This follows from the conditions $\deg(r^{(i)}, \text{mvar}(t_j)) < \deg(t_j, \text{mvar}(t_j))$ since h_1 contains none of the main variables of $(t_1, \ldots, t_{k-1})$ because T is normalized.

If at this point $\text{mvar}(r) \neq x_m$, then no further reduction needs to be done and the algorithm finishes with the correct result by returning $(q_1, \ldots, q_{k-1}, 0, r, h_1)$. This is handled by the else clause on lines 30-31 of the conditional on lines 14 to 32. If, on the other hand, $\text{mvar}(r) = x_m$, we must reduce r with respect to t_k. The pseudo-division algorithm indicates that after executing line 15,

$$\deg(r, \text{mvar}(t_k)) < \deg(t_k, \text{mvar}(t_k)),$$

and together with lines 16-20 implies that with the updated pseudo-quotients:

$$\tilde{h} h_1 a = \sum_{j=1}^{k} q_j t_j + r. \tag{4.3}$$

Since the pseudo-division step at line 15 may increase the degrees of the variables of r less than x_m in the variable ordering, we must issue a second set of recursive calls to ensure that (4.2) is satisfied. Again given the correctness of the algorithm, it follows that the result of the recursive calls on lines 21-23 taking as input $r = \sum_{i=0}^{\deg(r,x_m)} c_i x_m^i$, with $c_i = \mathrm{coeff}(r, x_m, i)$, are $q_j^{(i)}$, $r^{(i)}$ and $h^{(i)}$ such that $h^{(i)} c_i = \sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}$, where $\deg(r^{(i)}, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j))$. Combining these results as before and taking $h_2 = \mathrm{lcm}(h^{(i)})$ it follows that:

$$h_2 r = \sum_{j=1}^{k-1} \tilde{q}_j t_j + \tilde{r}, \tag{4.4}$$

satisfies a reduction condition of the form (4.2) with $\tilde{q} = \sum_i (h_2/h^{(i)}) q_j^{(i)} t_j x_m^i$ and $\tilde{r} = \sum_i (h_2/h^{(i)}) r_j^{(i)} x_m^i$, again because T is normalized. Multiplying (4.3) by h_2 and using equation (4.4) yields:

$$h_2 \tilde{h} h_1 a = \sum_{j=1}^{k} h_2 q_j t_j + h_2 r = \sum_{j=1}^{k} h_2 q_j t_j + \sum_{j=1}^{k-1} \tilde{q}_j t_j + \tilde{r} = \sum_{j=1}^{k-1} (h_2 q_j + \tilde{q}_j) t_j + h_2 q_k t_k + \tilde{r},$$

which gives the correct conditions for updating the pseudo-quotients on lines 25-27, with the $\tilde{q}_j$ and $\tilde{r}$ computed at line 28. Now $\tilde{r}$ is reduced with respect to x_m because r is and with respect to $\mathrm{mvar}(t_1), \ldots, \mathrm{mvar}(t_{k-1})$ because of the above argument, so that the correct overall multiplier is $h = h_2 \tilde{h} h_1$, set on line 29. The algorithm is therefore correct.

4.3.3 Experimentation

For comparing multi-divisor division (normal form) and pseudo-division with respect to a triangular set, we require more structure to our operands. For these experiments we use a zero-dimensional normalized (Lazard) triangular set. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4, and GMP 6.1.2, with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

For our benchmarks we use polynomials with 5 variables, say x_1, x_2, x_3, x_4, x_5, and thus a triangular set of size 5: $\mathbf{T} = (t_1, t_2, t_3, t_4, t_5)$. The polynomials in the divisor set and dividend (a) are always fully dense and have the following degree pattern. For some positive integer δ we let $\deg(a, x_1) = 2\delta$, $\deg(a, x_i) = \lg(\delta)$, $\deg(a, x_1) - \deg(t_1, x_1) = \delta$ and $\deg(a, x_i) - \deg(t_i, x_i) = 1$ for $1 < i \leq 5$.

There is a large gap in the lowest variable, but a small gap in the remaining variables, a common structure of which the recursive algorithms can take advantage. For both polynomials over $\mathbb{Q}$ (Figures 4.1 and 4.3) and over $\mathbb{Z}$ (Figures 4.2 and 4.4) we compare the naïve and recursive algorithms for both normal form and pseudo-division by a triangular set against MAPLE.

For normal form we call Maple's `Groebner:-NormalForm` with respect to the `rem` while for triangular set pseudo-division we implement Algorithm 16 in Maple using `prem`. Since `prem` is a kind of pseudo-division that we implement in the BPAS library.

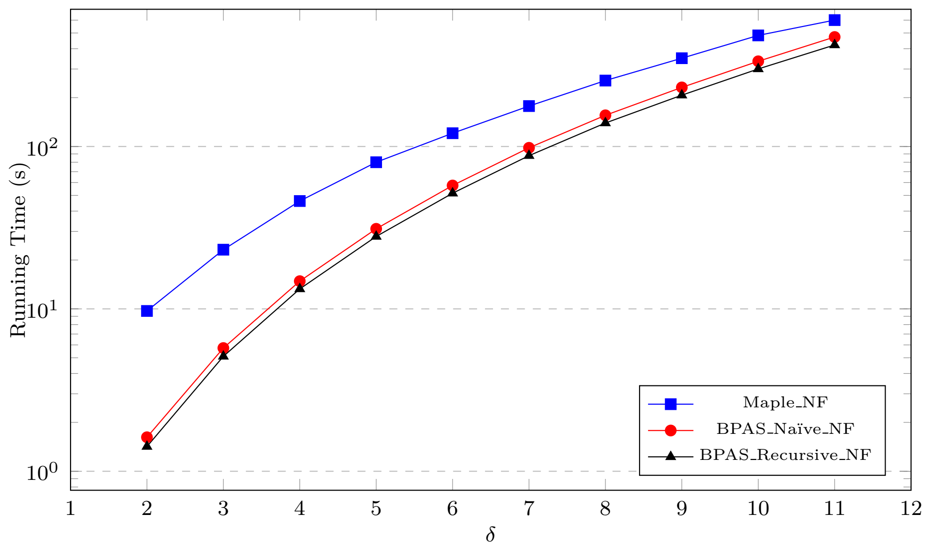


Figure 4.1: Comparing normal form algorithms over $\mathbb{Q}$ where polynomials are in $\mathbb{Q}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit.

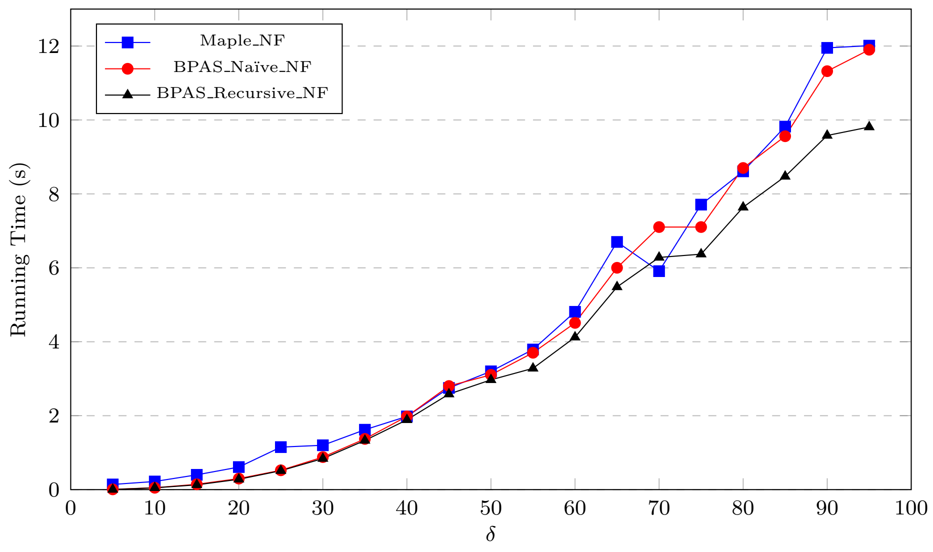


Figure 4.2: Comparing normal form algorithms over $\mathbb{Z}$ where polynomials are in $\mathbb{Z}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit.

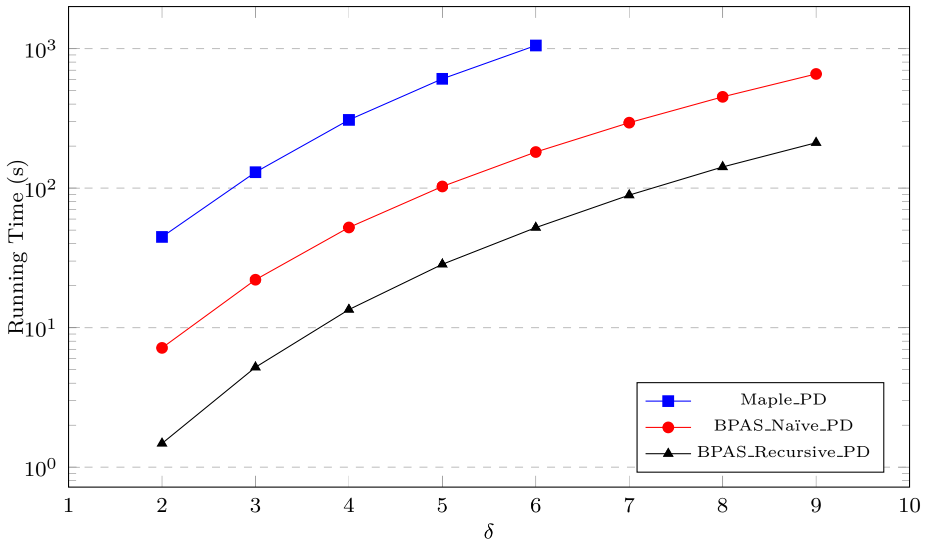


Figure 4.3: Comparing triangular set pseudo-division algorithms over $\mathbb{Q}$ where polynomials are in $\mathbb{Q}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit.

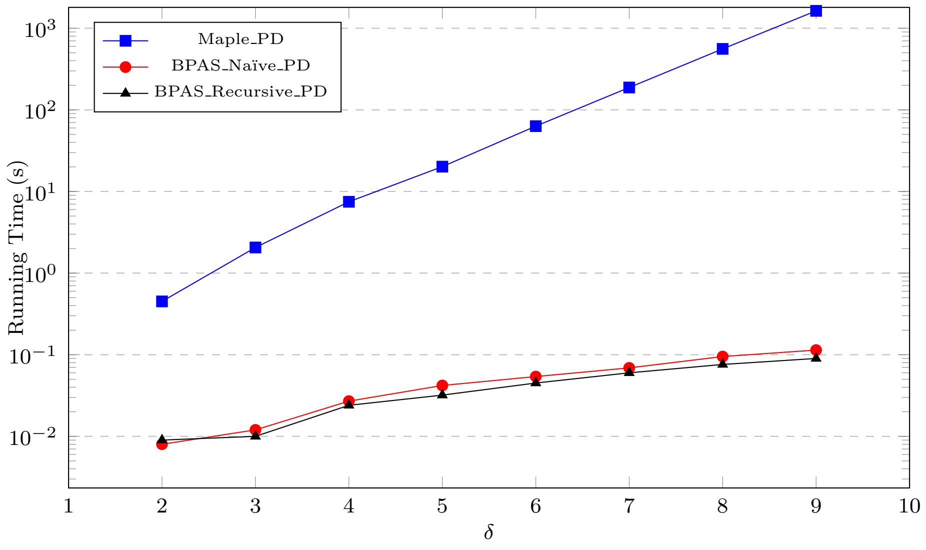


Figure 4.4: Comparing triangular set pseudo-division algorithms over $\mathbb{Z}$ where polynomials are in $\mathbb{Z}[x_1, x_2, x_3, x_4, x_5]$ and the coefficient bound is 128-bit.

4.4 Ducos' Subresultant Chain Algorithm

In [37], Ducos proposes two optimizations for Algorithm 1. The first one, attributed to Lazard, deals with the potentially expensive exponentiations and division at Line 11 of Algorithm 1. The second optimization considers the potentially expensive exact division (of a pseudo-remainder by an element from the coefficient ring) at Line 15 of this algorithm.

Applying both improvements to Algorithm 1 yields an efficient subresultant chain procedure that is known as Ducos' algorithm. In this section, we review both optimizations that are so-called Lazard optimization and Ducos optimization algorithms. In addition, we discuss a memory-efficient version of the latter optimization. The performance of these routines have been examined within the BPAS system solver; see Sections 3.6.2 and 4.4.3 for the experimentation details.

4.4.1 Original Scheme

In Algorithm 1 that computes the list of entire non-zero subresultants of two polynomials in $\mathbb{B}[y]$, two main calculations are carried out in the main while-loop. The Theorem 13 borrowed from [37] shows both formulas in this algorithm.

Theorem 13 *Let $\mathbb{B}$ be an integral ring. S_d be a regular (i.e. of degree d) subresultant polynomial of $a, b \in \mathbb{B}[y]$ with $d \leq min(\deg(a), \deg(b))$ and $S_{d-1} \neq 0$ of degree $e \in \{0, \ldots, d-1\}$. Then:*

$$S_e := \frac{\mathrm{lc}(S_{d-1})^{d-e-1} S_{d-1}}{\mathrm{lc}(S_d)^{d-e-1}},$$

$$S_{e-1} := \frac{\mathrm{prem}(S_d, -S_{d-1})}{\mathrm{lc}(S_d)^{d-e+1}}.$$

Proof. [37, Theorem 1]

Lazard Optimization

For the first equation in Theorem 13, Lazard proved that it is possible to avoid the expensive exponentiations $\mathrm{lc}(S_{d-1})^{d-e-1}$ and $\mathrm{lc}(S_d)^{d-e-1}$, and their division as follows:

$$S_e := \frac{\dfrac{\dfrac{\dfrac{\dfrac{\frac{\mathrm{lc}(S_{d-1})^2}{s_d} \times \cdots \times \mathrm{lc}(S_{d-1})}{s_d} \times \mathrm{lc}(S_{d-1})}{s_d} \times \mathrm{lc}(S_{d-1})}{s_d} \times S_{d-1}}{s_d},$$

where $s_d := \mathrm{lc}(S_d)$ and every division is *exact division*; see Algorithm 17. In this equation, we have:

$$\frac{\mathrm{lc}(S_{d-1})^{\delta+1}}{\mathrm{lc}(S_d)^{\delta}} \in \mathbb{B},$$

for all $0 \leq \delta \leq d-e$. Algorithm 17 shows this so-called Lazard optimization and used in the Ducos subresultant chain algorithm to compute S_e (Algorithm 20). This optimization is indeed efficient in the case of large degree gaps in the subresultant chain; see [37] for the details.

Algorithm 17 LAZARDOPTIMIZATION (S_d, S_{d-1})

```
Require: $S_d, S_{d-1} \in \mathbb{B}[y]$
Ensure: $S_e$, the next subresultant in the subresultant chain of $a, b$
 1: $n := \deg(S_d) - \deg(S_{d-1}) - 1$
 2: if $n = 0$ then
 3:     return $S_{d-1}$
 4: $(x, z) := (\mathrm{lc}(S_{d-1}), \mathrm{lc}(S_d))$
 5: $a := 2^{\lfloor \log_2(n) \rfloor}$
 6: $c := x$
 7: $n := n - a$
 8: while $a \neq 1$ do
 9:     $a := a/2$
10:     $c := c^2/z$
11:     if $n \geq a$ then
12:         $c := cx/z$
13:         $n := n - a$
14: return $cS_{d-1}/z$
```

Ducos Optimization

Ducos also proved a further optimization in order to compute the second equation. The calculation of $\mathrm{prem}(S_d, -S_{d-1})$, the exponentiation and the quotient are usually expensive. So, Ducos presented a so-called Ducos optimization in order to compute S_{e-1} while limiting the size of intermediate coefficients.

In Ducos optimization, he extended a so-called *Euclidean divisibility* relation between subresultant polynomials from [68, Section 6] to compute S_{e-1} similar to the computation of S_e with Lazard optimization. This method is presented in Algorithm 18. Ducos in [37] took advantage of both improvements to present Algorithm 20.

The Ducos' Subresultant Chain presented in Algorithm 20 is one of the fastest and optimized algorithm to compute the entire subresultants of two polynomials $a, b \in \mathbb{B}[y]$.

This algorithm is implemented in different computer algebra software including, but not limited to, Maple as part of the *RegularChains* library [64].

```
Algorithm 18 DucosOptimization ($S_d, S_{d-1}, S_e, s_d$)
Require: Given $S_d, S_{d-1}, S_e \in \mathbb{B}[y]$ and $s_d \in \mathbb{B}$
Ensure: $S_{e-1}$, the next subresultant in the subresultant chain of $a, b$
1: $(d, e) := (\deg(S_d), \deg(S_{d-1}))$
2: $(c_{d-1}, s_e) := (\mathrm{lc}(S_{d-1}), \mathrm{lc}(S_e))$
3: for $j = 0, \ldots, e-1$ do
4:     $H_j := s_e y^j$
5: $H_e := s_e y^e - S_e$
6: for $j = e+1, \ldots, d-1$ do
7:     $H_j := yH_{j-1} - \frac{\mathrm{coeff}(yH_{j-1}, e) S_{d-1}}{c_{d-1}}$
8: $D := \frac{\sum_{j=0}^{d-1} \mathrm{coeff}(S_d, j) H_j}{\mathrm{lc}(S_d)}$
9: return $(-1)^{d-e+1} \frac{c_{d-1}(yH_{d-1}+D) - \mathrm{coeff}(yH_{d-1}, e) S_{d-1}}{s_d}$
```

4.4.2 Memory-Efficient Scheme

The Ducos optimization that is presented in Algorithm 18 from [37, Section 3], is a well-known improvement of Algorithm 1 to compute the subresultant S_{e-1}. This optimization provides a faster procedure to compute the pseudo-division of two successive subresultants, namely $S_d, S_{d-1} \in \mathbb{B}[y]$, and a division by a power of $\mathrm{lc}(S_d)$. The main part of this algorithm is for-loops to compute:

$$D := \frac{\sum_{j=0}^{d-1} \mathrm{coeff}(S_d, j) H_j}{\mathrm{lc}(S_d)}.$$

Recall that $\mathrm{coeff}(S_d, j)$ is the coefficient of S_d in y^j. We now introduce a new optimization for this algorithm to make better use of memory resources through in-place arithmetic. This is shown in Algorithm 19.

In this algorithm we use a procedure named InplaceTail to compute the tail (the reductum of a polynomial with respect to its main variable) of a polynomial, and its leading coefficient, in-place. This operation is essentially a coefficient shift. In this way, we reuse existing memory allocations for the tails of polynomials S_d, S_{d-1}, and S_e.

Algorithm 19 *cache-friendly* DucosOptimization (S_d, S_{d-1}, S_e, s_d)

```
Require: $S_d, S_{d-1}, S_e \in \mathbb{B}[y]$ and $s_d \in \mathbb{B}$
Ensure: $S_{e-1}$, the next subresultant in the subresultant chain of subres$(a, b)$
 1: $(p, c_d) :=$ InplaceTail$(S_d)$
 2: $(q, c_{d-1}) :=$ InplaceTail$(S_{d-1})$
 3: $(h, s_e) :=$ InplaceTail$(S_e)$
 4: Convert $p$ to a recursive representation format in-place
 5: $h := -h$; $a := \mathrm{coeff}(p, e)\ h$
 6: for $i = e+1, \ldots, d-1$ do
 7:     if $\deg(h) = e - 1$ then
 8:         $h := y\ \mathrm{tail}(h) -$ ExactQuotient$(\mathrm{lc}(h)\ q, c_{d-1})$
 9:     else $h := y\ \mathrm{tail}(h)$
10:     $a := a + \mathrm{lc}(\mathrm{coeff}(p, i))\ h$
11: $a := a + s_e\ \sum_{i=0}^{e-1} \mathrm{coeff}(p, i) y^i$
12: $a :=$ ExactQuotient$(a, c_d)$
13: if $\deg(h) = e - 1$ then
14:     $a := c_{d-1}\ (y\ \mathrm{tail}(h) + a) - \mathrm{lc}(h)\ q$
15: else $a := c_{d-1}\ (y\ h + a)$
16: return $(-1)^{d-e+1}$ ExactQuotient$(a, s_d)$
```

Furthermore, we reduce the cost of calculating $\sum_{j=e}^{d-1} \mathrm{coeff}(S_d, j) H_j$ computing the summation iteratively and in-place in the same for-loop that is used to update polynomial h (lines 6-10 in Algorithm 19). This greatly improves data locality.

We also update the value of h depending on its degree with respect to y as $\deg(h) \leq e - 1$ for all $e + 1 \leq i < d$. We utilize an optimized exact division algorithm denoted by ExactQuotient to compute quotients rather a classical Euclidean algorithm.

Algorithm 20 Subresultant (a, b, y)

```
Require: $a, b \in \mathbb{B}[y]$ with $m = \deg(a) \geq n = \deg(b)$ and $\mathbb{B}$ is an integral domain
Ensure: the non-zero subresultants from $(S_n, S_{n-1}, S_{n-2}, \ldots, S_0)$
 1: if $m > n$ then
 2:     $S := (\mathrm{lc}(b)^{m-n-1} b)$
 3: else $S := ()$
 4: $s := \mathrm{lc}(b)^{m-n}$
 5: $A := b$;  $B :=$ PseudoDivide$(a, -b)$
 6: while true do
 7:     $d := \deg(A)$;  $e := \deg(B)$
 8:     if $B = 0$ then return $S$
 9:     $S := (B) \cup S$; $\delta := d - e$
10:     if $\delta > 1$ then
11:         $C :=$ LazardOptimization of $S_e$
12:         $S := (C) \cup S$
13:     else $C := B$
14:     if $e = 0$ then return $S$
15:     $B :=$ (cache-friendly) DucosOptimization of $S_{e-1}$
16:     $A := C$; $s := \mathrm{lc}(A)$
17: end while
```

4.4.3 Experimentation

Table 4.1: Ducos' test examples in [37, Section 5].

Test	$a \in \mathbb{Z}[X]$	$b \in \mathbb{Z}[X]$
1	$ax^6 + bx^5 + cx^4 + dx^3 + ex^2 + fx + g$	derivative of a
2	$x^5 + ax^4 + bx^3 + cx^2 + dx + e$	$x^5 + fx^4 + gx^3 + hx^2 + ix + j$
3	$x^7 + ax^3 + bx^2 + cx + d$	$x^7 + ex^3 + fx^2 + gx + h$
4	$x^{20} + ax^{15} + b$	$x^{20} + cx^5 + d$
5	$(x + a)^{15}$	$(x + z)^{15}$
6	$x^{30} + ax^{20} + 2ax^{10} + 3a$	$x^{25} + 4bx^{15} + 5bx^5$
7	$(a + x)^{90}$	$(a - x)^{60}$
8	$\sum_{j=0}^{75} a^{75-j} x^j$	$\sum_{j=0}^{n} j a^{n-j} x^j$
9	$\sum_{j=0}^{200} x^j$	$1 + \sum_{j=0}^{100} j x^j$
10	$1 + \sum_{j=1}^{900} j x^j$	$1 + \sum_{j=1}^{900} j^2 x^j$

Table 4.2: Comparing memory usage (GB) of Ducos' subresultant chain algorithms for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$ in Figure 3.7 in $\mathbb{Z}[y]$.

Degree	`BPAS_Ducos`	`BPAS_OptDucos`	`Maple_Ducos`
1000	1.088	0.320	3.762
1100	1.450	0.430	5.080
1200	1.888	0.563	6.597
1300	2.398	0.717	8.541
1400	2.968	0.902	10.645
1500	3.655	1.121	12.997
1600	4.443	1.364	15.924
1700	5.341	1.645	19.188
1800	6.325	1.958	23.041
1900	7.474	2.332	27.353
2000	8.752	2.721	31.793

We compare both the original Ducos' subresultant chain and the optimized one with the Ducos subresultant chain algorithm in MAPLE that is implemented as part of the *RegularChains* library [64]. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4, GMP 6.1.2, and MAPLE 2020, with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

Table 4.2 shows the memory usage for computing the entire subresultant chain of polynomials $a, b \in \mathbb{Z}[y]$, with $\deg(a) = \deg(b) + 1 = d$. The table presents `BPAS_Ducos`, `BPAS_OptDucos`, and `Maple_Ducos`. For $d = 2000$, Table 4.2 shows that the optimized algorithm uses approximately $3\times$ and $11\times$ less memory than our original implementation and the Ducos' algorithm in MAPLE, respectively.

In [37, Section 5], Ducos compared Algorithm 20 over 10 pairs of multivariate polynomials presented in Table 4.1. We compare the implementation of these methods in BPAS for these pairs of polynomials. The results are demonstrated in Table 4.3.

Table 4.3: Comparing the execution time (in seconds) of Ducos' subresultant chain algorithms for polynomials in Table 4.1.

Test	BPAS_Ducos	BPAS_OptDucos	BPAS_Ducos/BPAS_OptDucos
1	5.38	5.044	1.067
2	95.445	79.978	1.194
3	63.582	62.239	1.021
4	0.65	0.622	1.045
5	86.615	90.416	0.958
6	95.412	53.385	1.787
7	16.352	10.305	1.586
8	21.371	20.372	1.049
9	9.064	10.39	0.873
10	13.894	12.818	1.083

Table 4.4: Comparing memory usage (MB) of Ducos' subresultant chain algorithms for extended Ducos' Tests in Table 4.1 where deg$(a, x) = deg(b, x) + 1 = d$.

Test	d	BPAS_Ducos	BPAS_OptDucos	Maple_Ducos
1	9	421.8	427.7	466.1
2	90	30.4	29.9	21.9
3	14	154.6	153.6	130.1
4	55	36.2	34.8	53.4
5	30	84.4	77.5	320.5
6	120	123.4	47.128	46.4
7	540	5068.7	4434.3	14425.6
8	500	406.2	245.4	2279.4
9	2000	297.156	193.612	3007.7
10	5000	3955.5	300.2	22192.3

We further consider an extension of these test examples in Table 4.1 increasing the degree of x in both $a, b \in \mathbb{Z}[X]$ and collected the memory usage of them in Table 4.4. The result indicates the remarkable impact of our cache-friendly algorithm in the performance of Ducos' subresultant chain (Algorithm 20).

Chapter 5

Bézout Matrix over Multivariate Polynomials

5.1 Introduction

In the previous chapter, we reviewed the multivariate polynomial arithmetic over the *sparse* representation implemented in the BPAS library. In addition, we introduced a *cache-friendly* version of the well-known Ducos' subresultant chain algorithm. In this chapter, we study an alternative way to compute subresultants using linear-algebra methods, and introduce *speculative* subresultant algorithms for multivariate polynomials based on Hybrid Bézout Matrices.

As reviewed in Section 3.2, the standard definition of subresultants is based on the so-called *Sylvester matrix* (Definition 10); for two non-constant polynomials $a = \sum_{i=0}^{m} a_i y^i, b = \sum_{i=0}^{n} b_i y^i$ in $\mathbb{B}[y]$ with $\mathbb{B} = \mathbb{Z}[x]$, their *Sylvester* matrix is:

$$\mathrm{sylv}(a,b) = \begin{array}{c} n\left\{\begin{array}{c} \\ \\ \\ \\ \end{array}\right. \\ m\left\{\begin{array}{c} \\ \\ \\ \\ \end{array}\right. \end{array} \begin{bmatrix} a_m & a_{m-1} & \cdots & a_0 & & & \\ & a_m & a_{m-1} & \cdots & a_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & a_m & a_{m-1} & \cdots & a_0 \\ b_n & b_{n-1} & \cdots & b_0 & & & \\ & b_n & b_{n-1} & \cdots & b_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & b_n & b_{n-1} & \cdots & b_0 \end{bmatrix}.$$

For two polynomials $a, b \in \mathbb{Z}[x, y]$ with $\deg(a, x) = \deg(b, x) = d$, and $\deg(a, y) = \deg(b, y) = n$. The determinant of the Sylvester matrix of a, b is the subresultant of index 0 that is in fact the resultant of a, b, is a polynomial of degree bound d^{2n}. However, this degree bound is pessimistic in practice and as this matrix formulation is highly structured, its determinant can be computed in *quadratic* time instead of the usual *cubic* time; see Section 2.2 for more details.

There is another matrix formulation known as *Bézout matrix* for computing the resultant of a, b given as the determinant of matrix of size $n \times n$ where coefficients have degree $2d$. This determinant returns a polynomial of degree bound $(2d)^n$. Unlike the former approach, the Bézout matrix approaches involve a less structured matrix requiring a *cubic* algorithm to perform determinants.

There have been several studies to reduce the complexity of computing subresultants based on Bézout matrices. Most notably, Abdeljaoued et al. in [2] introduced an algorithm to compute the nominal coefficients of subresultants from calculating the determinants of sub-matrices of a modified version of the *Bézout matrix*. Later, Kerber in [55] generalized this approach to compute the entire subresultants instead of only the nominal coefficients.

Although this approach is still theoretically slower than Ducos' subresultant chain algorithm, early experimental results in Maple, that is collected during the development of the `SubresultantChain` method in the *RegularChains* library [64], indicates that Bézout approaches are well-suited for super sparse polynomials with many variables.

Section 5.2 tackles the problem of computing determinants of sub-matrices of the Bézout matrix. In this section, we study a so-called *fraction-free* LU decomposition algorithm (FFLU) that is one of the most practical approaches to compute the *exact* matrix factorization, and so determinant, over the polynomial ring $\mathbb{Z}[x_1, \ldots, x_v]$ for $v > 2$ [50, 62]. We further make use of different optimization techniques including the fraction-free Bareiss algorithm (Section 5.2.1), smart-pivoting (Section 5.2.2), and BPAS multithreaded interface to parallelize the row elimination step (Section 5.2.3). The performance of these routines in BPAS are demonstrated in Section 5.2.4.

Section 5.3 is focused on the computation of subresultants using Bézout matrix. In section 5.3.1, we review definitions of Bézout matrix and a modified version of it that is known as Hybrid Bézout matrix. We also discuss different approaches to calculate subresultants from these matrices. Furthermore, we introduce speculative and caching schemes to compute two successive subresultants, e.g. (S_0, S_1), (S_2, S_3), with modifying the fraction-free LU factorization and utilizing the Hybrid Bézout matrices in Section 5.3.2. We implement these schemes in the BPAS library and the results are presented in Section 5.3.3.

Co-Authorship Statement

The data-structure and basic matrix arithmetic along with the determinant and subresultants schemes are implemented by Asadi follows [50] under the supervision of Prof. Marc Moreno Maza.

5.2 Fraction-Free LU Decomposition

The fraction-free methods for *exact* matrix computations is emphasized by Bareiss [17] to find the solutions of systems with integer equations. He introduced fraction-free Gaussian elimination of the augmented matrix $[Ab]$ for the system,

$$AX = b.$$

Because of the close relation between linear system solving and the LU matrix decomposition, this fraction-free idea is extended to compute the *exact* matrix factorization [62, 50]. In this section, we study the Bareiss algorithm to compute the fraction-free LU (FFLU) decomposition with following definitions in [50] and the determinant algorithm based on FFLU. Given a matrix A over $\mathbb{Z}[x_1, \ldots, x_v]$:

$$A = P_r L D U P_c,$$

where L and U are respectively lower and upper triangular matrices, and D is a diagonal matrix. Note that the entries of these matrices are multivariate polynomials from $\mathbb{Z}[x_1, \ldots, x_v]$. The main specification of Bareiss' algorithm is that it creates common factors to every entry in a row, but these factors can be removed by doing *exact divisions* afterwards.

To further optimize the FFLU algorithm, we introduce a new technique named *smart-pivoting* to find the best pivot by searching into the sub-matrices to pick the polynomial with the minimum number of terms in each iteration. The goal of this technique is to reduce the cost of the exact divisions in the Bareiss' algorithm; see Section 5.2.2 for the details.

In addition, we discuss the parallel opportunities of this algorithm in Section 5.2.3 using the BPAS multithreaded interface. Finally, Section 5.2.4 demonstrates the performance of these algorithms in the BPAS library utilizing the *sparse* multivariate polynomials arithmetic (Chapter 4).

5.2.1 Bareiss FFLU Algorithm

Let $\mathbb{B}$ be an integral domain and $A \in \mathbb{B}^{m\times n}$ be a matrix. We are interested in computing the determinant of A over $\mathbb{B}$. A standard LU decomposition for A in the context of linear algebra is performed as follow. Suppose that

$$A = \begin{pmatrix} a & \dots \\ b & \dots \\ \vdots & \ddots \end{pmatrix},$$

where a, b are non-zero elements of $\mathbb{B}$. We first choose a pivot in the usual Gaussian elimination. Let $a \in \mathbb{B}$ be our pivot. We next perform a *row operation* by subtracting the $\operatorname{quo}(b, a)$ multiplied by the first row from the second row. This division forces the computation in the quotient field of $\mathbb{B}$, that may be computationally expensive. From Chapter 9 in [43], a straightforward way to avoid this division is to multiply the second row by a and then subtract b multiplied by the first row. This division-free technique avoids fractions, but, this is computationally expensive and infeasible.

Bareiss [17] introduced another technique to deal with fractions in the Gaussian elimination. He noticed that in the division-free technique after carrying out cross-multiplication for the first two rows every entry of the updated matrix in the third row and below is divisible by a, and so, one can utilize exact divisions to remove these factors. In [50] a fraction-free LU factorization based on the idea of Bareiss' algorithm is introduced. We review the main result of that paper in Theorem 14.

Theorem 14 *A rectangular matrix A with elements from an integral domain $\mathbb{B}$, having dimensions $m \times n$ and rank r, may be factored into matrices containing only elements from $\mathbb{B}$ in the form,*

$$A = P_r LDUP_c = P_r \begin{pmatrix} \mathcal{L} \\ \mathcal{M} \end{pmatrix} D \begin{pmatrix} \mathcal{U} & \mathcal{V} \end{pmatrix} P_c,$$

where the permutation matrix P_r is $m \times m$; the permutation matrix P_c is $n \times n$; $\mathcal{L}$ is $r \times r$, lower triangular and has full rank:

$$\mathcal{L} = \begin{pmatrix} p_1 & 0 & \dots & 0 \\ l_{21} & p_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{r1} & l_{r2} & \dots & p_r \end{pmatrix},$$

where the $p_i \neq 0$ are the pivots in a Gaussian elimination; $\mathcal{M}$ is $(m-r) \times r$ and could be null; D is $r \times r$ and diagonal:

$$D = diag(p_1, p_1p_2, p_2p_3, \cdots, p_{r-2}p_{r-1}, p_{r-1}p_r),$$

$\mathcal{U}$ is $r \times r$ and upper triangular, while $\mathcal{V}$ is $r \times (n-r)$ and could be null:

$$\mathcal{U} = \begin{pmatrix} p_1 & u_{12} & \dots & u_{1r} \\ 0 & p_2 & \dots & u_{2r} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & p_r \end{pmatrix}.$$

PROOF. [50, Theorem 2].

Algorithm 21 is the Bareiss' algorithm and is in fact the core routine of the fraction-free LU decomposition algorithm in [50]. This algorithm updates the input matrix A in-place, to become the upper triangular matrix U, the denominator d, rank and the permutation patterns of the input matrix. These are sufficient information to calculate the determinant of a matrix by Algorithm 22 and the L and U matrices by Algorithm 23.

In Algorithm 22, the CHECK-PARITY routine calculates the parity of the given permutation modulo 2. Note that in both Algorithms 21 and 23, we only consider row-operations to find the pivot and store the row permutation patterns in list P_r of size m. We take into account the column-operations and the list P_c of column permutation patterns in Section 5.2.2.

Example 13 *Consider matrix $A \in \mathbb{B}^{4\times 4}$ where $\mathbb{B} = \mathbb{Z}[x]$,*

$$A^{(0)} = \begin{pmatrix} 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ -2x+3 & 0 & 0 & -x \end{pmatrix}.$$

To compute the determinant of this matrix, Algorithm 21 starts with $d=1$, $k=0$, $c=0$, $P_r = [0,1,2,3]$, $A_{0,0} = 11x^2-11x+3 \neq 0$, and $r=1$. The nested for-loops updates the entire sub-matrix from the second row and column:

$$A^{(1)} = \begin{pmatrix} 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & (11x^2-11x+3)^2 & A^{(1)}_{1,2} & 0 \\ 0 & 0 & (11x^2-11x+3)^2 & A^{(1)}_{2,3} \\ -2x+3 & 3(x-1)(2x-3)^2 & 0 & (11x^2-11x+3)x \end{pmatrix},$$

Algorithm 21 FFLU-HELPER(A)

Require: a $m \times n$ matrix $A = (a_{i,j})_{0 \le i < m,\ 0 \le j < n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).

Ensure: r, d, P_r where r is the rank, d is the denominator, so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular) and $s \in (-1, 1)$ is decided by the parity of permutations, and P_r is permutation patterns of A.

```
1: $k := 0$; $d := 1$; $k := 0$; $c := 0$; $P_r := [0, 1, \ldots, m-1]$
2: while $k < m$ and $c < n$ do
3:     if $a_{k,c} = 0$ then
4:         $i := k + 1$
5:         while $i < m$ do
6:             if $a_{i,c} \neq 0$ then
7:                 SWAP $i$-th and $k$-th rows of A
8:                 $P_r[i], P_r[k] := P_r[k], P_r[i]$
9:                 break
10:            $i := i + 1$
11:        if $m \le i$ then
12:            $c := c + 1$
13:            continue
14:    $r := r + 1$
15:    for $i = k + 1, \ldots, m - 1$ do
16:        for $j = c + 1, \ldots, n - 1$ do
17:            $a_{i,j} := a_{i,c}\ a_{k,j} - a_{i,j}\ a_{k,c}$
18:            if $k = 0$ then $a_{i,j} := -a_{i,j}$
19:            else $a_{i,j} :=$ EXACTQUOTIENT($a_{i,j}, d$)
20:    $d := -a_{k,c}$; $k := k + 1$; $c := c + 1$
21: return $r, -d, P_r$
```

Algorithm 22 DET(A)

Require: a $n \times n$ matrix A over $\mathbb{B}$

Ensure: $\det(A)$, the determinant of A

```
1: $r, d, P_r :=$ FFLU-HELPER($A$)
2: if $r < n$ then return 0
3: $p :=$ CHECK-PARITY($P_r$)
4: if $p \neq 0$ then $d := -d$
5: return $d$
```

Algorithm 23 FFLU(A)

Require: a $m \times n$ matrix $A = (a_{i,j})_{0\le i<m,\ 0\le j<n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).

Ensure: r, d, P, L, U where r is the rank, d is the denominator, so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular) and $s \in (-1, 1)$ is decided by the parity of permutations, P is permutation patterns of A, L is the lower triangular matrix, and U is the upper triangular matrix s.t. $PA = LDU$.

```
1:  $U := A$; $i = 0$; $j = 0$; $k = 0$
2:  $r, d, P :=$ FFLU-HELPER($U$)
3:  while $i < m$ and $j < n$ do
4:      if $U[i, j] \neq 0$ then
5:          for $l = 0, \dots, i-1$ do $L_{l,k} := 0$
6:          $L_{i,k} := U_{i,j}$
7:          for $l = 0, \dots, m-1$ do $L_{l,k} := U_{l,j}$; $U_{l,j} := 0$
8:          $i := i + 1$; $k := k + 1$
9:      $j := j + 1$
10: while $k < m$ do
11:     for $l = 0, \dots, k-1$ do $L_{l,k} := 0$
12:     $L_{k,k} := 1$
13:     for $l = k+1, \dots, m$ do $L_{l,k} := 0$
14:     $k := k + 1$
15: return $r, d, P, L, U$
```

where $A^{(1)}_{1,2} = A^{(1)}_{2,3} = 3(x-1)(2x-3)(11x^2-11x+3)$. *In the second iteration of the while-loop, we have* $d = -11x^2 + 11x - 3$, $k = 1$, $c = 1$, $A_{1,1} = (11x^2 - 11x + 3)^2 \neq 0$, *and* $r = 2$. *Then,*

$$A^{(2)} = \begin{pmatrix} 11x^2 - 11x + 3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & (11x^2-11x+3)^2 & A^{(1)}_{1,2} & 0 \\ 0 & 0 & -(11x^2-11x+3)^3 & -(11x^2-11x+3)A^{(1)}_{23} \\ -2x+3 & 3(x-1)(2x-3)^2 & 0 & -(11x^2-11x+3)^2 x \end{pmatrix}.$$

In the third iteration of the while-loop, we have $d = -(11x^2 - 11x + 3)^2$, $k = 2$, $c = 2$,

$A_{2,2} = -(11x^2 - 11x + 3)^3 \neq 0$, *and* $r = 3$. *And so,*

$$A^{(3)} = \begin{pmatrix} 11x^2 - 11x + 3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & (11x^2 - 11x + 3)^2 & A^{(1)}_{1,2} & 0 \\ 0 & 0 & -(11x^2 - 11x + 3)^3 & -(11x^2 - 11x + 3)A^{(1)}_{2,3} \\ -2x + 3 & 3(x-1)(2x-3)^2 & 0 & A^{(3)}_{3,3} \end{pmatrix},$$

where $A^{(3)}_{3,3} = -1763x^7 + 7881x^6 - 19986x^5 + 35045x^4 - 41157x^3 + 30186x^2 - 12420x + 2187$. *In fact, one can check that* $A^{(3)}_{3,3}$ *is the determinant of the full-rank* ($r = 4$) *matrix* $A \in \mathbb{Z}[x]^{4\times 4}$.

Moreover, Bareiss in [17] introduced an updated version of this algorithm as *multi-step* Bareiss' algorithm to compute fraction-free LU decomposition. This method reduces the computation of row eliminations by adding three cheaper divisions to compute each row in the while-loop and removing one multiplication in each iteration of the nested for-loops; see the results in Table 5.1 and [43, Chapter 9] for more details.

In the next sections, we investigate optimizations of Algorithm 21 to compute the determinant of matrices over multivariate polynomials. These optimizations are achieved by reducing the cost of exact divisions by finding better pivots and utilizing the BPAS multithreaded interface to parallelize this algorithm.

5.2.2 Smart-Pivoting in FFLU Algorithm

Returning to Example 13, we performed exact divisions for the following divisors in the second and third iterations,

$$\begin{aligned} d^{(1)} &= -11x^2 + 11x - 3, \\ d^{(2)} &= -121x^4 + 242x^3 - 187x^2 + 66x - 9. \end{aligned}$$

However we could pick a polynomial with the fewer terms as our pivot in every iteration to reduce the cost of these exact divisions. Such a method, which finds a polynomial with the minimum number of terms in each column as the pivot of each iteration, is refereed to as column-wise smart-pivoting. For matrix $A^{(0)} \in \mathbb{Z}^{4\times 4}$, one can pick $A_{3,0} = -2x + 3$ as the first pivot. Applying this method in Example 13, we have:

$$A^{(1)} = \begin{pmatrix} -2x + 3 & 0 & 0 & -x \\ 0 & -22x^3 + 55x^2 - 39x + 9 & 12x^3 - 48x^2 + 63x - 27 & 0 \\ 0 & 0 & -22x^3 + 55x^2 - 39x + 9 & 12x^3 - 48x^2 + 63x - 27 \\ 11x^2 - 11x + 3 & 12x^3 - 48x^2 + 63x - 27 & 0 & 11x^3 - 11x^2 + 3x \end{pmatrix},$$

where $d = -2x+3$. Continuing this method from Algorithm 21, we get the following matrix for $r = 4$:

$$A^{(4)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & -22x^3+55x^2-39x+9 & 12x^3-48x^2+63x-27 & 0 \\ 0 & 0 & A^{(4)}_{2,2} & A^{(4)}_{2,3} \\ 11x^2-11x+3 & 12x^3-48x^2+63x-27 & A^{(4)}_{3,2} & A^{(4)}_{3,3} \end{pmatrix},$$

where

$$A^{(4)}_{2,2} = -242x^5+847x^4-1100x^3+693x^2-216x+27,$$

$$A^{(4)}_{2,3} = 132x^5-660x^4+1257x^3-1134x^2+486x-81,$$

$$A^{(4)}_{3,2} = 72x^5-468x^4+1206x^3-1539x^2+972x-243,$$

$A^{(4)}_{3,3} = 1763x^7-7881x^6+19986x^5-35045x^4+41157x^3-30186x^2+12420x-2187$ and $P_r = [3,1,2,0]$. And we have $\text{DET}(A) = -A^{(4)}_{3,3}$ from Algorithm 22.

In the *column-wise smart-pivoting*, we limited our search to find the best pivot to the corresponding column of the current row. To extend this method, one can try searching the best pivot in the sub-matrix starting from the next current row and column. To perform this method refereed to as *(fully) smart pivoting*, we need to use column-operations and column-wise permutation matrix P_c. The column operations along with row operations are not cache-friendly. This is certainly an issue for matrices with (large) multivariate polynomial entries while this may not be an issue with (relatively small) matrices with numerical entries. Therefore, we avoid column swapping within the decomposition, and instead we keep track of column permutations in the list of column-wise permutation patterns P_c to calculate the parity check later in Algorithm 22.

Algorithm 24 presents the pseudo-code of the *smart pivoting* fraction-free LU decomposition utilizing both row-wise and column-wise permutation patterns P_r, P_c. This algorithm updates A in-place, to become the upper triangular matrix U, and returns the rank and denominator of the given matrix $A \in \mathbb{B}^{m\times n}$.

5.2.3 Parallel FFLU Algorithm

We further investigate the parallel opportunities of the fraction-free LU decomposition with using the `parallel_for` loop from the BPAS multithreaded interface (Section 2.3) to do the row reduction step in parallel. Algorithm 25 shows a naïve implementation of this algorithm in parallel.

Algorithm 24 SPFFLU-HELPER(A)

Require: a $m \times n$ matrix $A = (a_{i,j})_{0 \le i < m,\ 0 \le j < n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).

Ensure: r, d, P_r, P_c where r is the rank, d is the denominator, so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular) and $s \in (-1, 1)$ is decided by the parity of permutations, and P_r, P_c are permutation patterns of A.

```
1:  $k := 0$; $d := 1$; $k := 0$; $c := 0$
2:  $P_r := [0, 1, \ldots, m-1]$
3:  $P_c := [0, 1, \ldots, n-1]$
4:  while $k < m$ and $c < n$ do
5:      if $a_{k,c} = 0$ then
6:          $i := k + 1$
7:          while $i < m$ do
8:              if $a_{i,c} \neq 0$ then
9:                  $(i, j) :=$ FindBestPivot($A, i, c$)
10:                 swap $i$-th and $k$-th rows of A
11:                 $P_r[i], P_r[k] := P_r[k], P_r[i]$
12:                 $P_c[j], P_c[c] := P_c[c], P_c[j]$
13:                 break
14:             $i := i + 1$
15:         if $m \le i$ then
16:             $c := c + 1$
17:             continue
18:     else
19:         $(i, j) :=$ FindBestPivot($A, k, c$)
20:         swap $i$-th and $k$-th rows of A
21:         $P_r[i], P_r[k] := P_r[k], P_r[i]$
22:         $P_c[j], P_c[c] := P_c[c], P_c[j]$
23:     $r := r + 1$
24:     for $i = k+1, \ldots, m-1$ do
25:         for $j = c+1, \ldots, n-1$ do
26:             $a_{i,P_c[j]} := a_{i,P_c[c]}\ a_{k,P_c[j]} - a_{i,P_c[j]}\ a_{k,P_c[c]}$
27:             if $k = 0$ then $a_{i,P_c[j]} := -a_{i,P_c[j]}$
28:             else $a_{i,P_c[j]} :=$ ExactQuotient($a_{i,P_c[j]}, d$)
29:     $d := -a_{k,P_c[c]}$; $k := k + 1$; $c := c + 1$
30: return $r, -d, P_r, P_c$
```

In practice, as the size of the sub-matrices decreases in each iteration and to address the *load-balancing* between threads to maximize parallelism, we only use the parallel loop within the first for-loop (line 1) of this nested for-loops in Algorithm 25.

Algorithm 25 PARALLEL-SPFFLU-HELPER(A)

```
   // --snip--
1: parallel_for $i = k+1, \ldots, m-1$
2:     parallel_for $j = c+1, \ldots, n-1$
3:         $a_{i,j} := a_{i,c}\ a_{k,j} - a_{i,j}\ a_{k,c}$
4:         if $k = 0$ then  $a_{i,j} := -a_{i,j}$
5:         else $a_{i,j} :=$ ExactQuotient($a_{i,j}, d$)
6:     end for
7: end for
   // --snip--
8: return $r, -d, P_r, P_c$
```

5.2.4 Experimentation

In this section, we compare the fraction-free LU decomposition algorithms for Bézout matrix (Definition 16) of randomly generated, non-zero and sparse polynomials in $\mathbb{Z}[x_1, \ldots, x_v]$ for $v \geq 5$ in the BPAS library. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4 and GMP 6.1.2, with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

Table 5.1 shows the comparison between the standard implementation of the fraction-free LU decomposition (Algorithm 21) denoted as `plain`, the *column-wise* smart pivoting denoted as `col-wise smart-pivoting`, the *fully* smart-pivoting method (Algorithm 24), and the multi-step Bareiss technique in Algorithm 24 denoted as `multi-step` for sparse polynomials with $v = 5$ and sparsity ratio 0.98 where sparsity is defined as the maximum degree difference between any two successive non-zero terms.

This table indicates that using smart-pivoting yields speed-up up to factor 3. Comparing `col-wise smart-pivoting` and `smart-pivoting` also shows that calculating P_c (column-wise permutation patterns) along with P_r (row-wise permutation patterns) does not cause any slow-down in the calculation of d.

Moreover, using both multi-step technique and smart-pivoting does not bring any additional speed-up as the smart-pivoting technique is already minimized the cost of exact divisions in each iteration. Table 5.2 shows $^{\text{plain}}/_{\text{smart-pivoting}}$, $^{\text{plain}}/_{\text{multi-step in Algo. 24}}$, and $^{\text{smart-pivoting}}/_{\text{multi-step}}$ ratios from Table 5.1.

Table 5.1: Compare the execution time (in seconds) of fraction-free LU decomposition algorithms for Bézout matrix of randomly generated, non-zero and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_5]$ with $x_5 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, $\deg(a, x_2) = \deg(b, x_2) + 1 = 5$, $\deg(a, x_3) = \deg(b, x_3) = 1$, $\deg(a, x_4) = \deg(b, x_4) = 1$, $\deg(a, x_5) = \deg(b, x_5) = 1$.

d	`plain`	`col-wise smart-pivoting`	`smart-pivoting` (Algo. 24)	`multi-step` in Algo. 24
6	0.048346	0.018623	0.021154	0.021257
7	2.379480	0.941655	0.954981	0.953532
8	3.997310	0.444759	0.426654	0.475043
9	73.860600	32.531600	31.764200	30.882500
10	2726.690000	1431.430000	1408.140000	1398.370000
11	9059.290000	5113.530000	4768.950000	5348.520000
12	5953.150000	3937.250000	3521.140000	3711.790000
13	81411.900000	42858.500000	42043.600000	41850.800000

Table 5.2: Ratios of FFLU algorithms for polynomials in Table 5.1.

d	$^{\text{plain}}/_{\text{smart-pivoting}}$	$^{\text{plain}}/_{\text{multi-step in Algo. 24}}$	$^{\text{smart-pivoting}}/_{\text{multi-step}}$
6	2.285431	2.274357	0.995155
7	2.491652	2.495438	1.001520
8	9.368973	8.414628	0.898138
9	2.325278	2.391665	1.028550
10	1.936377	1.949906	1.006987
11	1.899640	1.693794	0.891639
12	1.690688	1.603849	0.948637
13	1.936368	1.945289	1.004607

To analyze the performance of parallel FFLU algorithm, we compare Algorithm 25 and Algorithm 24 for $n \times n$ matrices of randomly generated non-zero univariate polynomials with integer coefficients and degree 1. Table 5.3 and Figure 5.1 show a $2.14\times$ speed-up for $n = 75$, where with increasing $n > 150$, we can achieve speed-up of up to factor 3.

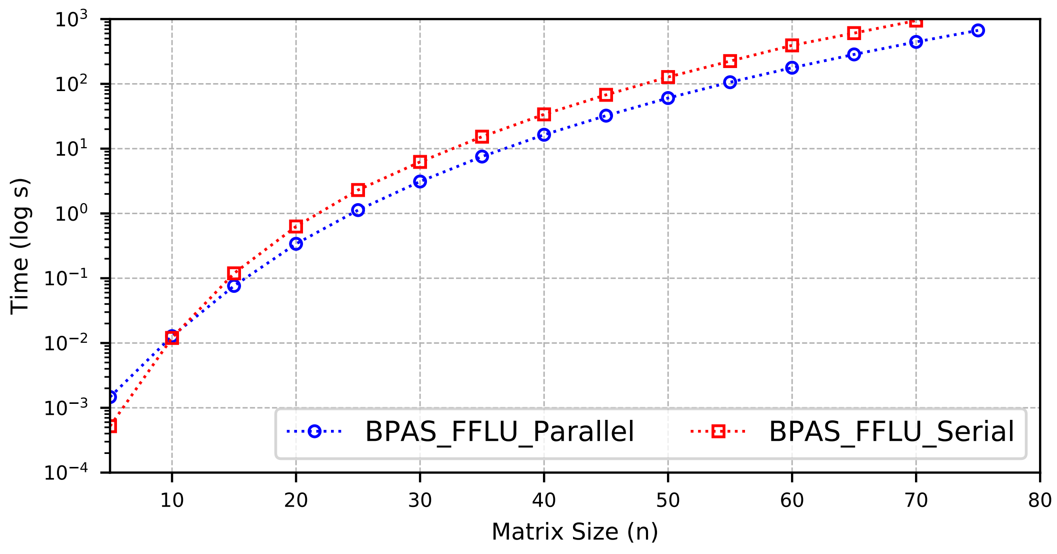


Figure 5.1: Comparing Algorithm 24 (`BPAS_FFLU_Serial`) and Algorithm 25 (`BPAS_FFLU_Parallel`) for $n \times n$ matrices filled with random non-zero univariate polynomials with integer coefficients and degree 1.

Table 5.3: Comparing the execution time (in seconds) of Algorithm 24 and Algorithm 25 for $n \times n$ matrices filled with random non-zero univariate polynomials with integer coefficients and degree 1.

n	*serial* FFLU	*parallel* FFLU	serial/parallel
10	0.011976	0.012765	0.938190
15	0.118972	0.076118	1.562994
20	0.628613	0.339738	1.850288
25	2.299270	1.126620	2.040857
30	6.241600	3.109840	2.007049
35	15.305100	7.552200	2.026575
40	33.831800	16.387200	2.064526
45	67.702600	32.307100	2.095595
50	127.438000	60.420000	2.109202
55	224.681000	106.043000	2.118773
60	392.795000	177.456000	2.213478
65	607.089000	284.659000	2.132689
70	947.805000	444.181000	2.133826
75	1432.180000	668.991000	2.140806

5.3 Bézout Subresultant Algorithms

In Section 4.4, we studied subresultants of two multivariate polynomials with integer coefficients using pseudo-division and introduced a cache-friendly Ducos' subresultant chain algorithm to compute the entire subresultants in a top-down procedure. In this section, we continue exploring the subresultant algorithms for multivariate polynomials based on calculating the determinant of (Hybrid) Bézout matrices.

5.3.1 Bézout Matrix and Subresultants

A traditional way to define subresultants is via computing determinants of submatrices of the Sylvester matrix (Section 2.2). Li [67] presented an elegant way to calculate subresultants directly from the following matrices. This method follows the same idea as subresultants based on Sylvester matrix.

Theorem 5.3.1 *The k-th subresultant of $a = \sum_{i=0}^{m} a_i y^i, b = \sum_{i=0}^{n} b_i y^i \in \mathbb{B}[y]$ is calculated by the determinant of the following $(m+n-k) \times (m+n-k)$ matrix:*

$$E_k := \begin{array}{r|ccccccccc|}
 & a_m & a_{m-1} & \cdots & a_2 & a_1 & a_0 & & & \\
n-k & & \ddots & & & & \ddots & & & \\
 & & & a_m & a_{m-1} & \cdots & a_2 & a_1 & a_0 & \\
 & & & & & & 1 & -y & & \\
k & & & & & & & \ddots & \ddots & \\
 & & & & & & & & 1 & -y \\
 & b_n & b_{n-1} & \cdots & b_2 & b_1 & b_0 & & & \\
m-k & & \ddots & & & & \ddots & & & \\
 & & & b_n & b_{n-1} & \cdots & b_2 & b_1 & b_0 &
\end{array}, \qquad (5.1)$$

so that,

$$S_k(a,b) = (-1)^{k(m-k+1)} det(E_k) \,.$$

Proof. [67, Section 2]

Another practical division-free approach is through utilizing the Bézout matrix to compute the subresultant chain of multivariate polynomials by calculating determinant of the Bézout matrix of input polynomials [48]. From [20], we define the *symmetric* Bézout matrix as follows.

Definition 15 *The Bézout matrix associated to $a, b \in \mathbb{B}[y]$ with $m := \deg(a) \geq n :=$*

deg(b) *is the symmetric matrix:*

$$Bez(a,b) := \begin{pmatrix} c_{0,0} & \cdots & c_{0,m-1} \\ \vdots & \ddots & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,m-1} \end{pmatrix},$$

where the $c_{i,j}$ for $0 \leq i,j < m$ are defined by the so-called Cayley expression as follows,

$$\frac{a(x)b(y) - a(y)b(x)}{x - y} = \sum_{i,j=0}^{m-1} c_{i,j} y^i x^j.$$

The relations between the *Sylvester* and Bézout matrices have been studied for decades yielding an efficient algorithm to construct the Bézout matrix [3] using a so-called Hybrid Bézout matrix.

Definition 16 *The Hybrid Bézout matrix of $a = \sum_{i=0}^{m} a_i y^i$ and $b = \sum_{i=0}^{n} b_i y^i$ is defined as the $m \times m$ matrix*

$$HBez(a,b) := \begin{pmatrix} h_{0,0} & \cdots & h_{0,m-1} \\ \vdots & \ddots & \vdots \\ h_{m-1,0} & \cdots & h_{m-1,m-1} \end{pmatrix},$$

where the $h_{i,j}$ for $0 \leq i,j < m$ are defined as:

$$\begin{aligned} h_{i,j} &= \mathrm{coeff}(H_{m-i+1}, m-j) \text{ for } 1 \leq i \leq n, \\ h_{i,j} &= \mathrm{coeff}(x^{m-i}b, m-j) \text{ for } m+1 \leq i \leq n, \end{aligned}$$

with,

$$\begin{aligned} H_i = {} & (a_m y^{i-1} + \cdots + a_{m-i+1})(b_{n-i} y^{m-i} + \cdots + b_0 y^{m-n}) \\ & - (a_{m-i} y^{m-i} + \cdots + a_0)(b_n y^{i-1} + \cdots + b_{n-i+1}). \end{aligned}$$

Example 14 *Consider polynomials $a = 5y^5 + y^3 + 2y + 1$ and $b = 3y^3 + y + 3$ in $\mathbb{Z}[y]$. The Sylvester matrix of a, b (Definition 10) is:*

$$Sylv(a,b) = \begin{pmatrix} 5 & 0 & 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 5 & 0 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 5 & 0 & 1 & 0 & 2 & 1 \\ 3 & 0 & 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 1 & 3 \end{pmatrix},$$

and the Bézout matrix of a, b is:

$$Bez(a,b) = \begin{pmatrix} 0 & -15 & 0 & -5 & -15 \\ -15 & 0 & -5 & -15 & 0 \\ 0 & -5 & -15 & 5 & 0 \\ -5 & -15 & 5 & 0 & 0 \\ -15 & 0 & 0 & 0 & -5 \end{pmatrix},$$

while the Hybrid Bézout matrix of a, b is:

$$HBez(a,b) = \begin{pmatrix} 15 & -6 & 0 & -2 & -1 \\ 2 & 15 & -6 & -3 & 0 \\ 0 & 2 & 15 & -6 & -3 \\ 3 & 0 & 1 & 3 & 0 \\ 0 & 3 & 0 & 1 & 3 \end{pmatrix}.$$

Diaz-Toca and Gonzalez-Vega indicated the relation of the Bézout matrices and the subresultants [34], and Hou and Wang presented another way to prove the application of the Hybrid Bézout matrix for the calculation of subresultants [48].

Definition 17 *Let J_m denote the backward identity matrix of order m and let B and H be defined as follows:*

$$B := J_m \; Bez(a,b) \; J_m = \begin{pmatrix} c_{m-1,m-1} & \cdots & c_{m-1,0} \\ \vdots & \ddots & \vdots \\ c_{0,m-1} & \cdots & c_{0,0} \end{pmatrix},$$

$$H := J_m \; HBez(a,b) \quad = \begin{pmatrix} h_{m-1,0} & \cdots & h_{m-1,m-1} \\ \vdots & \ddots & \vdots \\ h_{0,0} & \cdots & h_{0,m-1} \end{pmatrix}.$$

Now, we can define algorithms to compute the subresultants from Bézout matrices as follows.

Theorem 15 *For polynomials $a = \sum_{i=0}^{m} a_i y^i$ and $b = \sum_{i=0}^{n} b_i y^i$ in $\mathbb{B}[y]$, the k-th subresultant of a, b, i.e. $S_k(a,b)$, can be obtained from:*

$$(-1)^{(m-1)(m-k-1)/2} a_m^{m-n} S_k(a,b) = \sum_{i=0}^{k} B_{m-k,k-i} \; y^i,$$

where $B_{m-k,i}$ for $0 \leq i \leq k$ denotes the $(m-k) \times (m-k)$ minor extracted from the first $m-k$ rows, the first $m-k-1$ columns and the $(m-k+i)$-th column of B.

PROOF. [3, Theorem 2.3]

Theorem 16 *For those polynomials $a, b \in \mathbb{B}[y]$, the k-th subresultant of a, b, i.e. $S_k(a,b)$, can be obtained from:*

$$(-1)^{(m-1)(m-k-1)/2} S_k(a,b) = \sum_{i=0}^{k} H_{m-k,k-i}\, y^i,$$

where $H_{m-k,i}$ for $0 \le i \le k$ denotes the $(m-k)\times(m-k)$ minor extracted from the first $m-k$ rows, the first $m-k-1$ columns and the $(m-k+i)$-th column of H.

PROOF. [3, Theorem 2.3]
Abdeljaoued et al. in [3] studies further this relation between subresultants and Bézout matrices. Theorem 17 is the main result of this paper.

Theorem 17 *For those polynomials $a, b \in \mathbb{B}[y]$, the k-th subresultant of a, b can be obtained from the following $m \times m$ matrices:*

$$(-1)^{(m-k)(m-k-1)/2} a_m^{m-n} S_k(a,b) = (-1)^k \begin{vmatrix} c_{m-1,m-1} & c_{m-1,m-2} & \cdots & \cdots & \cdots & c_{m-1,0} \\ \vdots & \vdots & \cdots & \cdots & \cdots & \vdots \\ c_{k,m-1} & c_{k,m-2} & \cdots & \cdots & \cdots & c_{k,0} \\ & & 1 & -y & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -y \end{vmatrix},$$

$$(-1)^{(m-k)(m-k-1)/2} S_k(a,b) = (-1)^k \begin{vmatrix} h_{m-1,0} & h_{m-1,1} & \cdots & \cdots & \cdots & h_{m-1,m-1} \\ \vdots & \vdots & \cdots & \cdots & \cdots & \vdots \\ h_{k,0} & h_{k,1} & \cdots & \cdots & \cdots & h_{k,m-1} \\ & & 1 & -y & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -y \end{vmatrix}.$$

PROOF. [3, Theorem 2.4]

The advantage of this aforementioned method is that one can compute the entire subresultant chain in a bottom-up fashion. This process starts from computing the determinant of matrix H (or B) in Definition 17 to calculate $S_0(a,b)$, the resultant of a, b, and update the last k-th row of H (or B) to calculate $S_k(a,b)$ for $1 \le k \le n$.

Example 15 *Consider polynomials* $a = -5y^4x+3yx-y-3x+3$ *and* $b = -2y^3x+3y^3-x$ *in* $\mathbb{Z}[x, y]$ *where* $x < y$*. From Definition 16, the Hybrid Bézout matrix of* a, b *is:*

$$HBez(a, b) = \begin{pmatrix} 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ -2x+3 & 0 & 0 & -x \end{pmatrix}.$$

The determinant of this matrix is calculated using the fraction-free LU decomposition schemes, similar to the Example 13. Theorem 17 for $k = 0$ *yields that,*

$$S_0(a, b) = -1763x^7 + 7881x^6 - 19986x^5 + 35045x^4 - 41157x^3 + 30186x^2 - 12420x + 2187.$$

For $k = 1$*, one can calculate* $S_1(a, b)$ *from the determinant of:*

$$H^{(1)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

that is,

$$S_1(a, b) = -242x^5y + 132x^5 + 847x^4y - 660x^4 - 1100x^3y + 1257x^3 + 693x^2y - 1134x^2 - 216xy + 486x + 27y - 81.$$

We can continue calculating subresultants of higher indices with updating matrix $H^{(1)}$*. For instance, the 2nd and 3rd subresultants achieve, respectively, from the determinant of:*

$$H^{(2)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

and,

$$H^{(3)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 1 & -y & 0 & 0 \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

that are,

$$S_2(a,b) = 22yx^3 - 12x^3 - 55yx^2 + 48x^2 + 39yx - 63x - 9y + 27,$$
$$S_3(a,b) = -2y^3x + 3y^3 - x.$$

We further studied the performance of computing subresultants from Theorem 17 in comparison to the Hybrid Bézout matrix in Definition 16 for multivariate polynomials with integer coefficients. In our implementation, we took advantage of *FFLU* schemes reviewed in Section 5.2 to compute the determinant of these matrices using *smart-pivoting* technique in *parallel*; see Section 5.3.3 for implementation details and results.

5.3.2 Speculative Bézout Subresultant Algorithms

In Example 15, we made use of the Hybrid Bézout matrix to compute subresultants of two polynomials in $\mathbb{Z}[x,y]$. In this approach, we constructed the square matrix H from Definition 17 and updated the last $k \geq 0$ rows following Theorem 17. Thus, the kth subresultant could be directly computed from the determinant of this matrix.

Consider the `Triangularize` algorithm (Section 2.1) to solve a polynomial system. The *Regular GCD* subroutine of this algorithm requires the computation of subresultants in a bottom-up fashion, starting from $S_0(a,b), S_1(a,b)$ for multivariate polynomials a,b with respect to their main variable.

In the aforementioned approach, we have to call the determinant algorithm twice for $H^{(0)} := H$ and $H^{(1)}$ to compute S_0, S_1 respectively. Here, we study a speculative approach to compute both S_0, S_1 within the complexity of computing only one of them. This approach also can be extended to compute any two successive subresultants S_k, S_{k+1} for $2 \leq k < \deg(b, x_n)$ speculatively.

To compute S_0, S_1 of polynomials $a = -5y^4x+3yx-y-3x+3$ and $b = -2y^3x+3y^3-x$ in $\mathbb{Z}[x,y]$ from Example 15, consider the $(m+1)\times m$ matrix $H^{(0,1)}$ with $m = 4$ derived from the Hybrid Bézout matrix of a,b,

$$H^{(0,1)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & 0 & 1 & -y \end{pmatrix}.$$

In this matrix, the first three rows are identical to the first three rows of $H^{(0)}$ and $H^{(1)}$, while the 4th row is the 4th row of $H^{(0)}$ and the 5th row is the 4th row of $H^{(1)}$. A deeper look into the determinant algorithm reveals that the *Gaussian (row) elimination* for the first three rows in each iteration of the fraction-free LU decomposition is similar in both $H^{(0)}$ and $H^{(1)}$ and the only difference is within the 4th row.

Hence, managing these row eliminations in the fraction-free LU decomposition, we can compute determinants of $H^{(0)}$ and $H^{(1)}$ by utilizing $H^{(0,1)}$ instead where we only need to call the FFLU algorithm once. Indeed, when this algorithm tries to eliminate the last rows of $H^{(0)}$ and $H^{(1)}$, we should use the last two rows of $H^{(0,1)}$ separately and return two denominators corresponding to S_0, S_1.

We further can extend this speculative approach to compute S_2, S_3 updating matrix $H^{(0,1)}$ to get the $(m+3)\times m$ matrix $H^{(2,3)}$:

$$H^{(2,3)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 1 & -y & 0 & 0 \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix}.$$

Therefore, To calculate subresultants of index 2 and 3, we should respectively consider the 2nd and 5th rows of $H^{(2,3)}$ in the fraction-free LU decomposition while the grey rows are ignored. Therefore, an adaptation of the FFLU algorithm updates $H^{(2,3)}$ as follows to return $d_{(2)}$ ignoring the 5th and grey rows,

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & -2x+3 & -y(-2x+3) & 0 \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 1 & -y & 0 & 0 \\ 0 & 0 & -22x^3+55x^2-39x+9 & 12x^3-48x^2+63x-27 \\ 0 & 0 & -2x+3 & d_{(2)} \end{pmatrix},$$

where $d_{(2)} = -22x^3y + 12x^3 + 55x^2y - 48x^2 - 39xy + 63x + 9y - 27$ and $S_2 = -d_{(2)}$. Note that the 2nd and 6th rows are swapped to find a proper pivot. The adapted FFLU algorithm also updates $H^{(2,3)}$ to return $d_{(3)}$ ignoring the 2nd and grey rows,

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ 11x^2-11x+3 & -3(x-1)(2x-3) & 0 & 0 \\ 1 & -y(-2x+3) & 0 & x \\ 0 & -2x+3 & -y^2(-2x+3) & x \\ 0 & 0 & y(-2x+3) & d_{(3)} \end{pmatrix},$$

where $d_{(3)} = -2xy^3 + 3y^3 - x$ and $S_3 = d_{(3)}$.

Moreover, to compute subresultants of index k and $k+1$, one can construct the matrix $H^{(k,k+1)}$ from the previously constructed $H^{(k-2,k-1)}$ for $k > 1$, and calculate the adapted FFLU algorithm over:

- the first $m-k-1$ rows,
- the blue row for computing S_k, or the red row for computing S_{k+1}, and
- the last k rows,

of matrix $H^{(k,k+1)} \in \mathbb{B}^{(m+k)\times k}$ with $\mathbb{B} = \mathbb{Z}[x_1, \ldots, x_v]$,

$$H^{(k,k+1)} = \begin{pmatrix} h_{m-1,0} & h_{m-1,1} & \cdots & \cdots & \cdots & h_{m-1,m-1} \\ \vdots & \vdots & \cdots & \cdots & \cdots & \vdots \\ h_{k,0} & h_{k,1} & \cdots & \cdots & \cdots & h_{k,m-1} \\ h_{k-1,0} & h_{k-1,1} & \cdots & \cdots & \cdots & h_{k-1,m-1} \\ \vdots & \vdots & \cdots & \cdots & \cdots & \vdots \\ h_{0,0} & h_{0,1} & \cdots & \cdots & \cdots & h_{0,m-1} \\ & & 1 & -y & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -y \end{pmatrix}.$$

As seen in the last example, the FFLU algorithm depending on the input polynomials may create two completely different submatrices to calculate $d_{(2)}$ and $d_{(3)}$. Thus, the cost of computing S_k, S_{k+1} from $H^{(k,k+1)}$ speculatively may not necessarily be less than computing them successively from $H^{(k)}, H^{(k+1)}$ for some $k > 1$.

We further optimize the computation of S_k, S_{k+1} speculatively through *caching* the intermediate data calculated to compute S_{k-2}, S_{k-1} from $H^{(k-2,k-1)}$. In this approach, the adapted FFLU algorithm returns $d_{(k-2)}, d_{(k-1)}$ along with $H^{(k-2,k-1)}$, the reduced matrix $H^{(k-2,k-1)}$ to compute $d_{(k-1)}$, the list of permutation patterns and pivots.

Therefore, we can utilize $H^{(k-2,k-1)}$ to construct $H^{(k,k+1)}$. In addition, if the first $\delta := m - k - 1$ pivots are picked from the first δ rows of $H^{(k-2,k-1)}$, then one can use the first δ rows of the reduced matrix $H^{(k-2,k-1)}$ along with the list of permutation patterns and pivots to perform the first δ row eliminations of $H^{(k,k+1)}$ via recycling the first δ rows of the reduced matrix cached *a priori*.

5.3.3 Experimentation

In this section, we compare the subresultant algorithms based on (Hybrid) Bézout matrix against the Ducos' subresultant chain algorithm in BPAS and Maple. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4, GMP 6.1.2, and Maple 2020, with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

Table 5.4 and Table 5.5 show the running time of plain, speculative subresultant algorithms for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_6]$ with $x_6 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \le i \le 6$. Table 5.6 and Table 5.7 show the running time of plain, speculative and caching subresultant schemes for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_7]$ with $x_7 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \le i \le 7$.

Table 5.4: Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_6, x_5, \ldots, x_1]$ with $x_6 < x_5 < \cdots < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \le i \le 6$.

	Maple		BPAS			
d	Bézout ($\rho = 0$)	Ducos	Bézout ($\rho = 0$)	Bézout ($\rho = 1$)	SpecBézout ($\rho = 0$)	OptDucos
10	0.05128	0.03000	0.024299	0.026762	0.032166	0.045270
11	0.06001	0.04574	0.057312	0.068722	0.058843	0.049532
12	0.02515	0.05100	0.007223	0.019530	0.012792	0.061419
13	0.81209	16.81200	0.421278	0.739842	0.594225	9.527660
14	3.14360	112.280	2.414530	3.829530	3.250710	69.957100
15	518.380	7163.30	151.656	779.9240	512.260	3655.820

In these tables, the Bézout algorithm in Maple computes the resultant of a, b ($S_0(a, b)$) while the Maple's Ducos' algorithm computes the entire subresultants. In BPAS, *Bézout* ($\rho = 0$) calculates the resultant ($S_0(a, b)$) via the determinant of Hybrid Bézout matrix of a, b; *Bézout* ($\rho = 2$) calculates $S_1(a, b)$ following Theorem 17 from the Hybrid Bézout matrix of a, b; *SpecBézout* ($\rho = 0$) calculates $S_0(a, b), S_1(a, b)$ speculatively from utilizing $H^{(0,1)}$ in Section 5.3.2; *SpecBézout* ($\rho = 2$) calculates $S_2(a, b), S_3(a, b)$ speculatively via $H^{(2,3)}$; *SpecBézout*$_{\texttt{cached}}$ ($\rho = 2$) calculates $S_2(a, b), S_3(a, b)$ speculatively via $H^{(2,3)}$ using *cached* information calculated in *SpecBézout* ($\rho = 0$) from $H^{(0,1)}$; *SpecBézout*$_{\texttt{cached}}$ ($\rho = \texttt{all}$) calculates the entire subresultant chain using speculative and caching subresultant algorithm in Section 5.3.2; and *OptDucos* calculates the entire subresultant chain using the optimized Ducos' algorithm in Section 4.4 (Algorithm 20).

To Compute subresultants from Bézout matrices in MAPLE, we use the `Subresultant-Chain(..., 'representation'='BezoutMatrix')` command in `RegularChains` that is a naïve implementation of subresultant algorithm based on Bézout matrices. Our Bézout algorithm is up to 3× faster than the MAPLE implementation to calculate only S_0. Besides, Our results indicate the performance of Bézout algorithms in comparison with the Ducos' algorithm in both BPAS and MAPLE for sparse polynomials with many variables.

Tables 5.4 and 5.6 show that the cost of computing subresultants S_0, S_1 speculatively is comparable to the running time of computing only one of them. Tables 5.5 and 5.7 indicate the importance of recycling cached data to compute higher subresultants speculatively. Our Bézout algorithms can calculate the entire subresultants speculatively in a comparable running time to the Ducos' algorithm. This allows one to integrate these implementations on the BPAS solver for super sparse polynomials with many variables (`nvar` ≥ 5) without losing performance.

We further investigate the performance of these subresultant implementations within the BPAS system solver. Table 5.8 and Table 5.9 compare the execution time of systems with the number of variables greater than 4 (`nvar` ≥ 5) from the pool of more than 3000 polynomial systems. This test suite is collected from user-data and bug reports of the *RegularChains* library as well as real-world problems [10, Chapter 6].

To integrate subresultant algorithms based on the determinant of (Hybrid) Bézout matrix in the BPAS system solver, we define thresholds with respect to our comprehensive experimentation over polynomials with different number of variables, degrees, and sparsity. The solver utilizes these Bézout-based schemes for polynomials of minimum number of variables 5, sparsity ratio 0.8, and main degree 3 (that is the size of Bézout matrix). From tables 5.8 and 5.9, we deduce a speed-up factor of 1.6× for solving a few hard polynomials systems in the pool.

Table 5.5: Comparing the execution time (in seconds) of speculative subresultant algorithms for polynomials in Table 5.4.

	BPAS			
d	SpecBézout ($\rho = 0$)	SpecBézout ($\rho = 2$)	SpecBézout$_{\texttt{cached}}$ ($\rho = 2$)	SpecBézout$_{\texttt{cached}}$ ($\rho = \texttt{all}$)
10	0.032166	0.022125	0.016432	0.076283
11	0.058843	0.079425	0.043512	0.193512
12	0.012792	0.010566	0.004148	0.071435
13	0.594225	2.106280	1.535510	7.891180
14	3.250710	8.735510	4.133760	73.59940
15	512.260	953.1170	579.8580	4877.130

Table 5.6: Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_7]$ with $x_7 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 7$.

	Maple		BPAS			
d	Bézout ($\rho = 0$)	Ducos	Bézout ($\rho = 0$)	Bézout ($\rho = 1$)	SpecBézout ($\rho = 0$)	OptDucos
5	0.00032	0.00041	0.000023	0.000277	0.000262	0.000258
6	0.00098	0.00372	0.001303	0.001427	0.001553	0.002444
7	0.01148	0.43145	0.080210	0.174460	0.095569	0.279023
8	15.1850	34.8540	7.057270	10.834100	8.380050	22.440500
9	74.1390	327.570	36.8450	66.8430	44.7160	194.4860
10	9941.20	inf	4130.980	6278.240	5686.060	14145.30

Table 5.7: Comparing the execution time (in seconds) of speculative and caching subresultant algorithms for polynomials in Table 5.6.

	BPAS			
d	SpecBézout ($\rho = 0$)	SpecBézout ($\rho = 2$)	SpecBézout$_{\texttt{cached}}$ ($\rho = 2$)	SpecBézout$_{\texttt{cached}}$ ($\rho = \texttt{all}$)
5	0.000262	0.000351	0.000217	0.000479
6	0.001553	0.001812	0.001350	0.003519
7	0.095569	0.103801	0.053730	0.213630
8	8.380050	13.10210	5.7240	25.83050
9	44.7160	67.86560	31.12090	136.8930
10	5686.060	8853.10	3856.550	17569.20

Table 5.8: Comparing the execution time (in seconds) of systems with `nvar` ≥ 5 from the pool of more than 3000 polynomial systems [10] so that their running-times ($t \in \mathbb{R}$) using *OptDucos* take $1 \leq t \leq 50$ seconds.

SysName	OptDucos	Bézout	SpecBézout	SpecBézout$_{\texttt{cached}}$	OptDucos/SpecBézout	Bézout/SpecBézout
Sys2922	7.91041	7.93589	7.95695	7.95698	0.994151	0.997353
Sys2880	5.55801	5.70138	5.46921	5.41538	1.016236	1.042450
Sys2433	8.75830	8.77473	8.75625	8.76812	1.000234	1.002110
Sys2161	1.08153	0.89279	0.56666	0.63128	1.908605	1.575530
Sys2642	8.06066	6.97177	4.89233	3.21021	1.647612	1.425041
Sys2695	3.18267	3.05706	2.98045	2.12872	1.067849	1.025704
Sys2238	8.75708	8.75923	8.75251	8.75813	1.000522	1.000768
Sys2943	6.70348	6.12246	4.54511	4.69512	1.474877	1.347043
Sys1935	4.14390	5.01831	3.01449	1.98466	1.374660	1.664729
Sys2882	2.42182	2.37203	2.38065	2.35716	1.017294	0.996379
Sys2588	4.49268	4.51135	4.49201	4.49792	1.000149	1.004305
Sys2449	1.23251	1.28321	1.24507	1.26588	0.989912	1.030633
Sys2874	6.99887	7.22326	6.99438	7.11027	1.000642	1.032723
Sys2932	6.27556	6.25798	6.31953	6.29113	0.993042	0.990260
Sys2269	1.03128	1.03253	1.03961	1.04012	0.991987	0.993190

Table 5.9: Comparing the execution time (in seconds) of systems with `nvar` ≥ 5 from the pool of more than 3000 polynomial systems [10] so that their running-times using *OptDucos* take > 50 seconds.

SysName	OptDucos	Bézout	SpecBézout	SpecBézout$_{\texttt{cached}}$	OptDucos/SpecBézout	Bézout/SpecBézout
Sys2797	466.4250	425.8670	386.3810	325.1170	1.207163	1.102194
Sys2539	55.8694	55.8531	55.5113	55.4933	1.006451	1.006157
Sys2681	458.6800	458.5810	458.5360	458.5780	1.000314	1.000098
Sys2745	599.8020	599.3290	599.0610	599.2150	1.001237	1.000447
Sys3335	6406.7400	5843.7300	4799.9700	4801.1200	1.334746	1.217451
Sys2703	322.2940	487.0120	485.8170	491.1520	0.663406	1.002460
Sys2000	55.7026	56.1724	56.3106	57.0079	0.989203	0.997546
Sys2877	2127.4900	1914.5200	1253.8200	1247.4400	1.696807	1.526950

Chapter 6

Multivariate Power Series in Maple

6.1 Introduction

In elementary courses on univariate calculus, power series are often introduced as limits of sequences of the form "the first n terms of a given sequence". This leads students to the study of analytic functions and the use of power series in computing function limits. While the extension of those notions to the multivariate case is a standard topic in advanced calculus courses, the availability of multivariate power series and multivariate analytic functions in computer algebra systems is somehow limited.

In Maple [71], SageMath [95], and Mathematica [49], power series are restricted to being either only univariate or truncated, that is, reduced modulo a fixed power of the ideal $\langle X_1, \ldots, X_n \rangle$ generated by the variables of those power series. A truncated implementation, while simple, may be insufficient for, or computationally more expensive in, some particular circumstances. For instance, modern algorithms for polynomial system solving require the intensive use of modular methods based on Hensel lifting. In those lifting procedures, degrees of truncation may not be known a priori, thus leading to truncated power series being ineffective.

Considering that a power series has potentially an infinite number of terms naturally suggests to represent it as a procedure which, given a particular (total) degree, produces the terms of that degree. This leads to a so-called lazy evaluation scheme, where the terms of any power series are produced only as needed, via such a *generator* function.

The usefulness of lazy evaluation in computer algebra has been studied for a few decades. In particular, see the work of Karczmarczuk [54], discussing different mathematical objects with an infinite length; Burge and Watt [26], and van der Hoeven [97], discussing lazy univariate power series; and Monagan and Vrbik [79], discussing lazy arithmetic for polynomials.

In this chapter, we present `MultivariatePowerSeries`, which is among the new features released in Maple 2021 and publicly available in [1]. This library, written in the Maple language, provides the ability to create and manipulate multivariate power series with rational or algebraic number coefficients, as well as univariate polynomials whose coefficients are multivariate power series. Through lazy evaluation techniques and a careful implementation, our library achieves very high performance. These power series and univariate polynomials over power series (UPoPS) are employed in optimized implementations of Weierstrass Preparation Theorem and factorization of UPoPS via Hensel's lemma.

Our implementation follows the lazy evaluation scheme of multivariate power series in the BPAS library [8]. The multivariate power series of BPAS, written in the C language, is discussed in [22] and extends upon the work of the `PowerSeries` subpackage of the `RegularChains` Maple library [6, 82]. The `PowerSeries` package is the only preexisting implementation of multivariate power series integrated in Maple. In [22], it is shown that the BPAS implementation provides exceptional performance, surpassing that of the `PowerSeries` package, the basic Maple function `mtaylor`, and the multivariate power series available in SageMath [95] by multiple orders of magnitude.

A key design element of our library, in addition to lazy evaluation techniques, is the use of Maple *objects* and object-oriented programming. An object in Maple is a special kind of module which encapsulates together data and procedures manipulating that data, just like objects in any other object-oriented language; see [19, Chapters 8, 9]. To the best of our knowledge, few Maple libraries make use of those objects, which, as our report suggests, are worth considering for improving performance. In particular, objects allow for the overloading of existing builtin Maple functions in order to integrate these new custom objects with existing Maple library code. Our results show that `Multivariate-PowerSeries` is comparable in performance to the implementation of BPAS, is thus similarly several orders of magnitude faster than other existing implementations. These experimental results are discussed in Section 6.6.

We begin in Section 6.2 with reviewing definitions of formal power series, and univariate polynomial over power series, followed by a brief discussion about the basic arithmetic, Weierstrass preparation theorem and factorization via Hensel's lemma. Section 6.3 presents an overview of the `MultivariatePowerSeries` package, while Section 6.4 explores its underlying design principles. Implementation details are discussed in Section 6.5, followed by our experimentation in Section 6.6. Finally, we conclude and present future works in Section 6.6.

Co-Authorship Statement

This library is designed and developed by Asadi under the supervision of Prof. Marc Moreno Maza and Dr. Erik J. Postma [9], follows the C implementation of multivariate power series in BPAS [22].

6.2 Preliminary

In this section we review the basic properties of formal power series and univariate polynomials over those series, following G. Fischer in [41]. While various proofs of Theorems 18 of 6.2.1 can be found in the literature, the proofs given in [22] are constructive and support our implementation. Throughout this chapter, $\mathbf{k}$ is an algebraic number field.

6.2.1 Power Series

Given a positive integer n, we denote by $\mathbf{k}[\![X_1, \ldots, X_n]\!]$ the set of multivariate formal power series with coefficients in $\mathbf{k}$ and variables $X_1, \ldots, X_n$.

Definition 18 *Let $f = \sum_{e \in \mathbb{N}^n} a_e X^e \in \mathbf{k}[\![X_1, \ldots, X_n]\!]$ and $d \in \mathbb{N}$ where $X^e = X_1^{e_1} \cdots X_n^{e_n}$ and $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$. The homogeneous part and polynomial part of f in degree d are respectively defined by $f_{(d)} := \sum_{|e|=d} a_e X^e$, and $f^{(d)} := \sum_{k \leq d} f_{(k)}$, where $|e| = e_1 + \cdots + e_n$.*

The sum (resp. difference) of two formal power series $f, g \in \mathbf{k}[\![X_1, \ldots, X_n]\!]$ is defined by the sum (and resp. difference) of their homogeneous parts of the same degree; thus we have:

$$f \pm g := \sum_{d \in \mathbb{N}} f_{(d)} \pm g_{(d)}.$$

The product $h = f \cdot g$ can be defined as $h = \sum_{d \in \mathbb{N}} h_{(d)}$ with $h_{(d)} := \sum_{k+l=d} f_{(k)}\ g_{(l)}$. With the above addition and multiplication, the set $\mathbf{k}[\![X_1, \ldots, X_n]\!]$ is a local ring with $\mathcal{M} := \langle X_1, \ldots, X_n \rangle$ as maximal ideal; $\mathbf{k}[\![X_1, \ldots, X_n]\!]$ is also a unique factorization domain (UFD) [41].

Definition 19 *The order of the power series f, denoted by $\operatorname{ord}(f)$, is defined as:*

$$\operatorname{ord}(f) := min\{d \in \mathbb{N} \mid f_{(d)} \neq 0\},$$

if $f \neq 0$, and as ∞ otherwise.

We also observe that the following equality holds for every $k \geq 1$:

$$\mathcal{M}^k = \{f \in \mathbf{k}[\![X_1, \ldots, X_n]\!] \mid \mathrm{ord}(f) \geq k\}.$$

If f is a unit, that is, if $f \notin \mathcal{M}$ (or equivalently, if $\mathrm{ord}(f) = 0$) then the sequence $(h_m)_{m \in \mathbb{N}}$, where

$$h_m = c^{-1}(1 + g + \cdots + g^m), \ c = f_{(0)}, \text{ and } g = 1 - c^{-1}f,$$

converges to the *inverse* of f. This convergence is the sense of Krull topology, see [41] for details.

6.2.2 Univariate Polynomials over Power Series

We denote by $\mathcal{A}$ and $\mathcal{M}$ the power series ring $\mathbf{k}[\![X_1, \ldots, X_n]\!]$ and its maximal ideal. We allow $n = 0$, in which case we have $\mathcal{M} = \langle 0 \rangle$. Let $f \in \mathcal{A}[\![X_{n+1}]\!]$, written as $f = \sum_{i=0}^{\infty} a_i X_{n+1}^i$ with $a_i \in \mathcal{A}$ for all $i \in \mathbb{N}$. Then, Weierstrass Preparation Theorem (WPT) states the following.

Theorem 18 *Assume $f \not\equiv 0 \bmod \mathcal{M}[\![X_{n+1}]\!]$. Let $d \geq 0$ be the smallest integer such that $a_d \notin \mathcal{M}$. Then, there exists a unique pair (α, p) satisfying the following:*

(i) α is an invertible power series of $\mathcal{A}[\![X_{n+1}]\!]$,

(ii) $p \in \mathcal{A}[X_{n+1}]$ is a monic polynomial of degree d,

(iii) writing $p = X_{n+1}^d + b_{d-1}X_{n+1}^{d-1} + \cdots + b_1 X_{n+1} + b_0$, we have $b_{d-1}, \ldots, b_0 \in \mathcal{M}$,

(iv) $f = \alpha p$ holds.

Moreover, if f is a polynomial of $\mathcal{A}[X_{n+1}]$ of degree $d + m$, for some m, then α is a polynomial of $\mathcal{A}[X_{n+1}]$ of degree m.

Proof [22, Theorem 1]

Since $\mathcal{A}$ is a UFD, then Gauss' lemma implies that the polynomial ring $\mathcal{A}[X_{n+1}]$ is also a UFD. Hensel's lemma shows how factorizing a polynomial in $\mathcal{A}[X_{n+1}]$ can be reduced to factorizing a polynomial in $\mathbf{k}[X_{n+1}]$.

Theorem 6.2.1 (Hensel's Lemma) *Assume that f is a polynomial of degree k in $\mathcal{A}[X_{n+1}]$. We define $\overline{f} = f(0, \ldots, 0, X_{n+1}) \in \mathbf{k}[X_{n+1}]$. We assume that f is monic in X_{n+1}, that is, $a_k = 1$. We further assume that $\mathbf{k}$ is algebraically closed. Thus, there exists positive integers $k_1, \ldots, k_r$ and pairwise distinct elements $c_1, \ldots, c_r \in \mathbf{k}$ such that we have:*

$$\overline{f} = (X_{n+1} - c_1)^{k_1}(X_{n+1} - c_2)^{k_2} \cdots (X_{n+1} - c_r)^{k_r}.$$

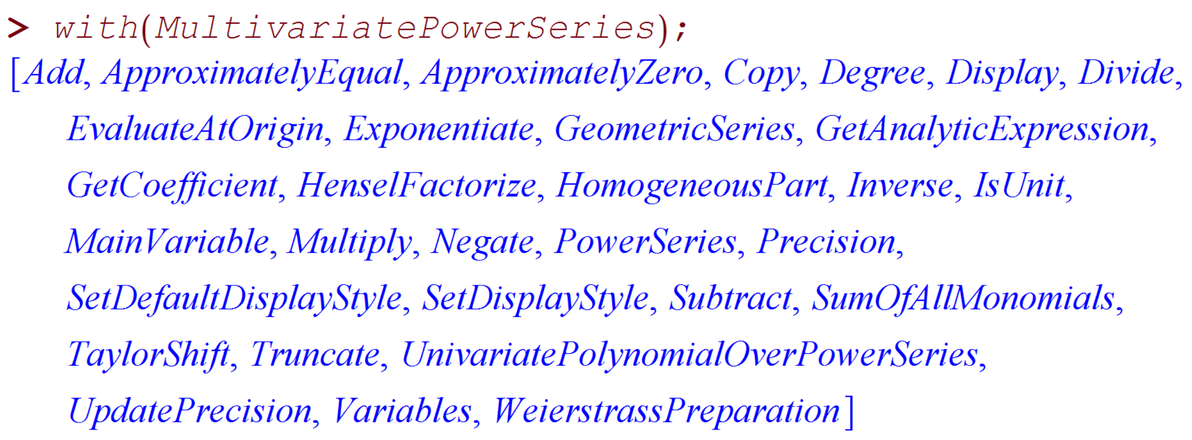
Then, there exists $f_1, \ldots, f_r \in \mathcal{A}[X_{n+1}]$, all monic in X_{n+1}, such that we have:

i. $f = f_1 \cdots f_r$,

ii. the degree of f_j is k_j, for all $j = 1, \ldots, r$,

iii. $\overline{f_j} = (X_{n+1} - c_j)^{k_j}$, *for all $j = 1, \ldots, r$.*

Proof [22, Theorem 2]



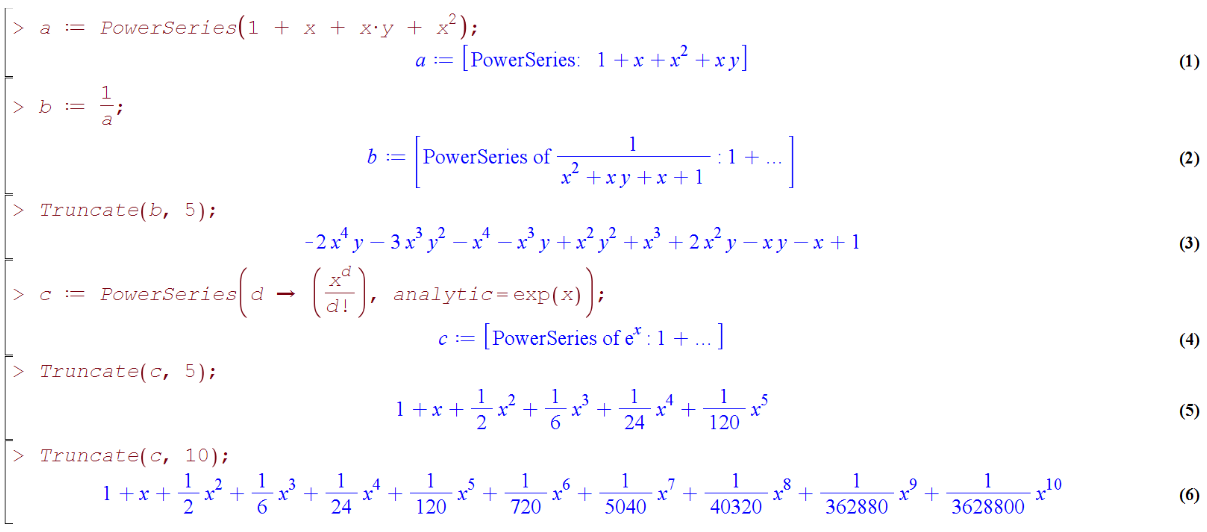
```
> with(MultivariatePowerSeries);
[Add, ApproximatelyEqual, ApproximatelyZero, Copy, Degree, Display, Divide,
    EvaluateAtOrigin, Exponentiate, GeometricSeries, GetAnalyticExpression,
    GetCoefficient, HenselFactorize, HomogeneousPart, Inverse, IsUnit,
    MainVariable, Multiply, Negate, PowerSeries, Precision,
    SetDefaultDisplayStyle, SetDisplayStyle, Subtract, SumOfAllMonomials,
    TaylorShift, Truncate, UnivariatePolynomialOverPowerSeries,
    UpdatePrecision, Variables, WeierstrassPreparation]
```

Figure 6.1: List of the commands of `MultivariatePowerSeries`.

6.3 An Overview of the User-Interface

From the point of view of the end-user, the `MultivariatePowerSeries` package is a collection of commands for manipulating multivariate power series and univariate polynomials over multivariate power series. The field of coefficients of all power series created by the command `PowerSeries` consists of all complex numbers that are constructible in Maple, thus including rational numbers and algebraic numbers. The main algebraic functionalities of this package deal with arithmetic operations (addition, multiplication, inversion, evaluation), for both multivariate power series and univariate polynomials over multivariate power series (UPoPS), as well as factorization of such polynomials. The list of the exposed commands is given in Figure 6.1.

The commands `PowerSeries` and `UnivariatePolynomialOverPowerSeries` create power series and univariate polynomials over multivariate power series, respectively, from objects like polynomials, sequences, and functions which produce homogeneous parts of a power series, as illustrated in Figures 6.2 and 6.3. The commands `GeometricSeries` and `SumOfAllMonomials` respectively create the geometric series and sum of all monomials for an input list of variables.



> $a := PowerSeries(1 + x + x \cdot y + x^2);$

$$a := [\text{PowerSeries: } 1 + x + x^2 + x\,y] \qquad \textbf{(1)}$$

> $b := \frac{1}{a};$

$$b := \left[\text{PowerSeries of } \frac{1}{x^2 + x\,y + x + 1} : 1 + \ldots\right] \qquad \textbf{(2)}$$

> $Truncate(b, 5);$

$$-2\,x^4\,y - 3\,x^3\,y^2 - x^4 - x^3\,y + x^2\,y^2 + x^3 + 2\,x^2\,y - x\,y - x + 1 \qquad \textbf{(3)}$$

> $c := PowerSeries\left(d \rightarrow \left(\frac{x^d}{d!}\right), analytic = \exp(x)\right);$

$$c := [\text{PowerSeries of } e^x : 1 + \ldots] \qquad \textbf{(4)}$$

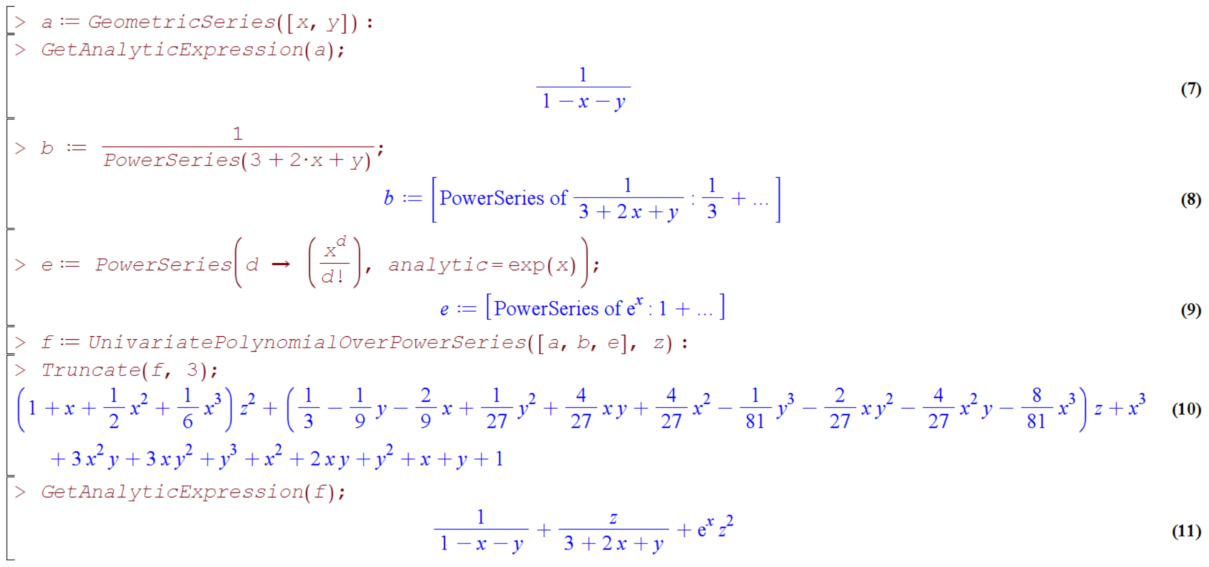
> $Truncate(c, 5);$

$$1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4 + \frac{1}{120}\,x^5 \qquad \textbf{(5)}$$

> $Truncate(c, 10);$

$$1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4 + \frac{1}{120}\,x^5 + \frac{1}{720}\,x^6 + \frac{1}{5040}\,x^7 + \frac{1}{40320}\,x^8 + \frac{1}{362880}\,x^9 + \frac{1}{3628800}\,x^{10} \qquad \textbf{(6)}$$

Figure 6.2: Creating power series from a polynomial or an anonymous function.



```
> a := GeometricSeries([x, y]):
> GetAnalyticExpression(a);
```

$$\frac{1}{1-x-y} \tag{7}$$

```
> b := 1/PowerSeries(3 + 2·x + y);
```

$$b := \left[\text{PowerSeries of } \frac{1}{3+2x+y} : \frac{1}{3} + \ldots\right] \tag{8}$$

```
> e := PowerSeries(d → (x^d/d!), analytic = exp(x));
```

$$e := \left[\text{PowerSeries of } e^x : 1 + \ldots\right] \tag{9}$$

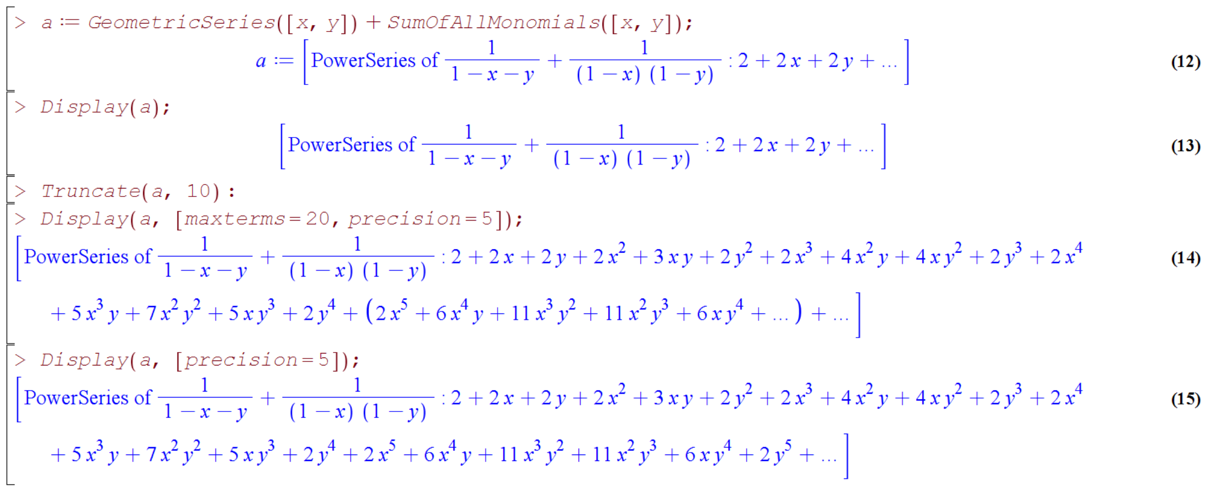
```
> f := UnivariatePolynomialOverPowerSeries([a, b, e], z):
> Truncate(f, 3);
```

$$\left(1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3\right)z^2 + \left(\frac{1}{3} - \frac{1}{9}y - \frac{2}{9}x + \frac{1}{27}y^2 + \frac{4}{27}xy + \frac{4}{27}x^2 - \frac{1}{81}y^3 - \frac{2}{27}xy^2 - \frac{4}{27}x^2y - \frac{8}{81}x^3\right)z + x^3 + 3x^2y + 3xy^2 + y^3 + x^2 + 2xy + y^2 + x + y + 1 \tag{10}$$

```
> GetAnalyticExpression(f);
```

$$\frac{1}{1-x-y} + \frac{z}{3+2x+y} + e^x z^2 \tag{11}$$

Figure 6.3: Creating a univariate polynomial over power series



```
> a := GeometricSeries([x, y]) + SumOfAllMonomials([x, y]);
```

$$a := \left[\text{PowerSeries of } \frac{1}{1-x-y} + \frac{1}{(1-x)(1-y)} : 2 + 2x + 2y + \ldots\right] \tag{12}$$

```
> Display(a);
```

$$\left[\text{PowerSeries of } \frac{1}{1-x-y} + \frac{1}{(1-x)(1-y)} : 2 + 2x + 2y + \ldots\right] \tag{13}$$

```
> Truncate(a, 10):
> Display(a, [maxterms = 20, precision = 5]);
```

$$\left[\text{PowerSeries of } \frac{1}{1-x-y} + \frac{1}{(1-x)(1-y)} : 2 + 2x + 2y + 2x^2 + 3xy + 2y^2 + 2x^3 + 4x^2y + 4xy^2 + 2y^3 + 2x^4 + 5x^3y + 7x^2y^2 + 5xy^3 + 2y^4 + \left(2x^5 + 6x^4y + 11x^3y^2 + 11x^2y^3 + 6xy^4 + \ldots\right) + \ldots\right] \tag{14}$$

```
> Display(a, [precision = 5]);
```

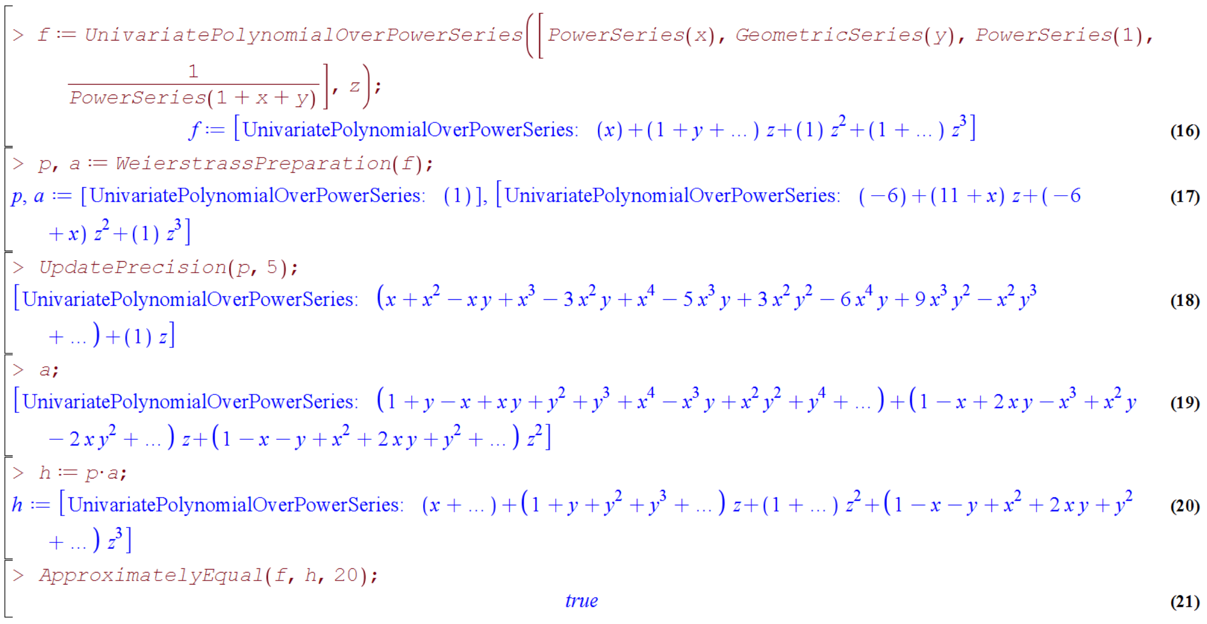
$$\left[\text{PowerSeries of } \frac{1}{1-x-y} + \frac{1}{(1-x)(1-y)} : 2 + 2x + 2y + 2x^2 + 3xy + 2y^2 + 2x^3 + 4x^2y + 4xy^2 + 2y^3 + 2x^4 + 5x^3y + 7x^2y^2 + 5xy^3 + 2y^4 + 2x^5 + 6x^4y + 11x^3y^2 + 11x^2y^3 + 6xy^4 + 2y^5 + \ldots\right] \tag{15}$$

Figure 6.4: Controlling the output format of a multivariate power series.

Whenever possible, the package associates every power series with its so-called *analytic expression*. For each power series `s`, created by the command `PowerSeries` as the image of a polynomial `p` (under the natural embedding from $\mathbb{C}[X_1, \ldots, X_n]$ to $\mathbb{C}[[X_1, \ldots, X_n]]$) the polynomial `p` is the analytic expression of `s`. If a power series is defined by the sequence of its homogeneous parts, as illustrated on Figure 6.3, the user can optionally specify the *sum* of that series which is then set to its analytic expression. Power series that have an analytic expression are closed under addition, multiplication and inversion. Propagating that information provides the opportunity to speed up some computations and make decisions that could not be made otherwise. For instance, the command `HenselFactorize` needs to decide whether its input polynomial has an invertible leading coefficient; to do it starts by checking whether the analytic expression of that leading coefficient is known and equal to one.

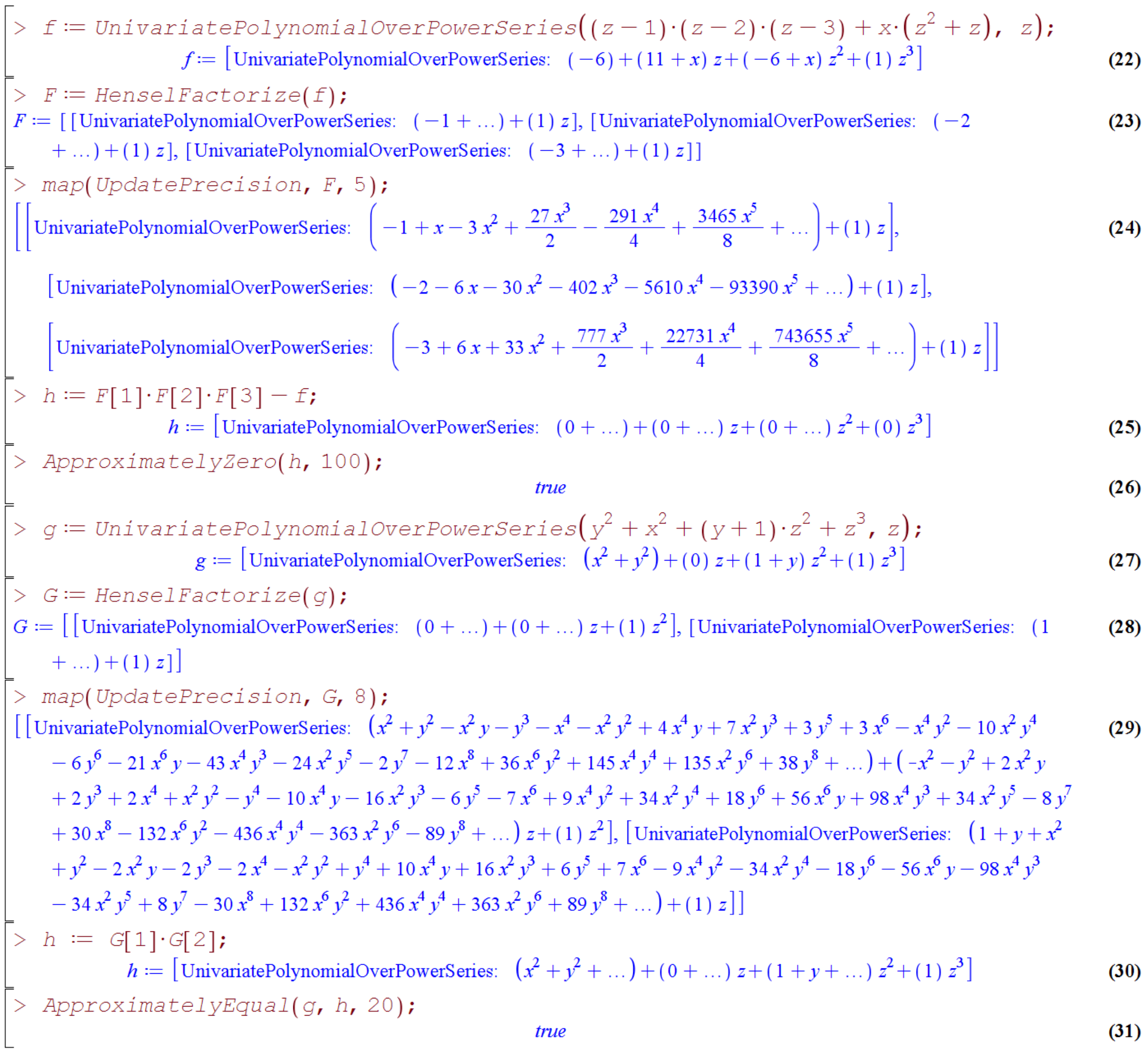
The commands `Display`, `SetDefaultDisplayStyle` and `SetDisplayStyle` control the output format of multivariate power series and UPoPS. Meanwhile, the commands `HomogeneousPart`, `Truncate`, `GetCoefficient`, `Precision`, `Degree`, `MainVariable` access data from a power series or a univariate polynomial over power series, as illustrated by Figure 6.4.

The commands `TaylorShift`, `Add`, `Negate`, `Multiply`, `Exponentiate`, `Inverse`, `Divide`, and `EvaluateAtOrigin` perform arithmetic operations on multivariate power series and univariate polynomials over multivariate power series. The functionality of the first six commands can also be accessed using the standard arithmetic operators. As will be discussed in Sections 6.4 and 6.5, the implementation of every arithmetic operation, such as addition, multiplication, inversion builds the resulting power series (sum, product or inverse) "lazily", by creating its generator from the generators of the operands, which are called *ancestors* of the resulting power series.



```
> f := UnivariatePolynomialOverPowerSeries([PowerSeries(x), GeometricSeries(y), PowerSeries(1),
        1/PowerSeries(1 + x + y)], z);
              f := [UnivariatePolynomialOverPowerSeries:  (x) + (1 + y + ... ) z + (1) z^2 + (1 + ... ) z^3]       (16)
> p, a := WeierstrassPreparation(f);
p, a := [UnivariatePolynomialOverPowerSeries:  (1)], [UnivariatePolynomialOverPowerSeries:  (-6) + (11 + x) z + (-6    (17)
    + x) z^2 + (1) z^3]
> UpdatePrecision(p, 5);
[UnivariatePolynomialOverPowerSeries:  (x + x^2 - x y + x^3 - 3 x^2 y + x^4 - 5 x^3 y + 3 x^2 y^2 - 6 x^4 y + 9 x^3 y^2 - x^2 y^3    (18)
    + ... ) + (1) z]
> a;
[UnivariatePolynomialOverPowerSeries:  (1 + y - x + x y + y^2 + y^3 + x^4 - x^3 y + x^2 y^2 + y^4 + ... ) + (1 - x + 2 x y - x^3 + x^2 y    (19)
    - 2 x y^2 + ... ) z + (1 - x - y + x^2 + 2 x y + y^2 + ... ) z^2]
> h := p·a;
h := [UnivariatePolynomialOverPowerSeries:  (x + ... ) + (1 + y + y^2 + y^3 + ... ) z + (1 + ... ) z^2 + (1 - x - y + x^2 + 2 x y + y^2    (20)
    + ... ) z^3]
> ApproximatelyEqual(f, h, 20);
                                          true                                                              (21)
```

Figure 6.5: Factoring univariate polynomials using `WeierstrassPreparation`.



```
> f := UnivariatePolynomialOverPowerSeries((z − 1)·(z − 2)·(z − 3) + x·(z^2 + z), z);
```

$$f := [\text{UnivariatePolynomialOverPowerSeries:}\ \ (-6) + (11 + x)\, z + (-6 + x)\, z^2 + (1)\, z^3] \tag{22}$$

```
> F := HenselFactorize(f);
```

$$F := [[\text{UnivariatePolynomialOverPowerSeries:}\ \ (-1 + \ldots) + (1)\, z], [\text{UnivariatePolynomialOverPowerSeries:}\ \ (-2 + \ldots) + (1)\, z], [\text{UnivariatePolynomialOverPowerSeries:}\ \ (-3 + \ldots) + (1)\, z]] \tag{23}$$

```
> map(UpdatePrecision, F, 5);
```

$$\left[\left[\text{UnivariatePolynomialOverPowerSeries:}\ \ \left(-1 + x - 3\,x^2 + \frac{27\,x^3}{2} - \frac{291\,x^4}{4} + \frac{3465\,x^5}{8} + \ldots\right) + (1)\, z\right], \left[\text{UnivariatePolynomialOverPowerSeries:}\ \ \left(-2 - 6\,x - 30\,x^2 - 402\,x^3 - 5610\,x^4 - 93390\,x^5 + \ldots\right) + (1)\, z\right], \left[\text{UnivariatePolynomialOverPowerSeries:}\ \ \left(-3 + 6\,x + 33\,x^2 + \frac{777\,x^3}{2} + \frac{22731\,x^4}{4} + \frac{743655\,x^5}{8} + \ldots\right) + (1)\, z\right]\right] \tag{24}$$

```
> h := F[1]·F[2]·F[3] − f;
```

$$h := [\text{UnivariatePolynomialOverPowerSeries:}\ \ (0 + \ldots) + (0 + \ldots)\, z + (0 + \ldots)\, z^2 + (0)\, z^3] \tag{25}$$

```
> ApproximatelyZero(h, 100);
```

$$\mathit{true} \tag{26}$$

```
> g := UnivariatePolynomialOverPowerSeries(y^2 + x^2 + (y + 1)·z^2 + z^3, z);
```

$$g := [\text{UnivariatePolynomialOverPowerSeries:}\ \ (x^2 + y^2) + (0)\, z + (1 + y)\, z^2 + (1)\, z^3] \tag{27}$$

```
> G := HenselFactorize(g);
```

$$G := [[\text{UnivariatePolynomialOverPowerSeries:}\ \ (0 + \ldots) + (0 + \ldots)\, z + (1)\, z^2], [\text{UnivariatePolynomialOverPowerSeries:}\ \ (1 + \ldots) + (1)\, z]] \tag{28}$$

```
> map(UpdatePrecision, G, 8);
```

$$\begin{aligned}&[[\text{UnivariatePolynomialOverPowerSeries:}\ \ (x^2 + y^2 - x^2 y - y^3 - x^4 - x^2 y^2 + 4\,x^4 y + 7\,x^2 y^3 + 3\,y^5 + 3\,x^6 - x^4 y^2 - 10\,x^2 y^4\\&\quad - 6\,y^6 - 21\,x^6 y - 43\,x^4 y^3 - 24\,x^2 y^5 - 2\,y^7 - 12\,x^8 + 36\,x^6 y^2 + 145\,x^4 y^4 + 135\,x^2 y^6 + 38\,y^8 + \ldots) + (-x^2 - y^2 + 2\,x^2 y\\&\quad + 2\,y^3 + 2\,x^4 + x^2 y^2 - y^4 - 10\,x^4 y - 16\,x^2 y^3 - 6\,y^5 - 7\,x^6 + 9\,x^4 y^2 + 34\,x^2 y^4 + 18\,y^6 + 56\,x^6 y + 98\,x^4 y^3 + 34\,x^2 y^5 - 8\,y^7\\&\quad + 30\,x^8 - 132\,x^6 y^2 - 436\,x^4 y^4 - 363\,x^2 y^6 - 89\,y^8 + \ldots)\, z + (1)\, z^2], [\text{UnivariatePolynomialOverPowerSeries:}\ \ (1 + y + x^2\\&\quad + y^2 - 2\,x^2 y - 2\,y^3 - 2\,x^4 - x^2 y^2 + y^4 + 10\,x^4 y + 16\,x^2 y^3 + 6\,y^5 + 7\,x^6 - 9\,x^4 y^2 - 34\,x^2 y^4 - 18\,y^6 - 56\,x^6 y - 98\,x^4 y^3\\&\quad - 34\,x^2 y^5 + 8\,y^7 - 30\,x^8 + 132\,x^6 y^2 + 436\,x^4 y^4 + 363\,x^2 y^6 + 89\,y^8 + \ldots) + (1)\, z]]\end{aligned} \tag{29}$$

```
> h := G[1]·G[2];
```

$$h := [\text{UnivariatePolynomialOverPowerSeries:}\ \ (x^2 + y^2 + \ldots) + (0 + \ldots)\, z + (1 + y + \ldots)\, z^2 + (1)\, z^3] \tag{30}$$

```
> ApproximatelyEqual(g, h, 20);
```

$$\mathit{true} \tag{31}$$

Figure 6.6: Factoring univariate polynomials using `HenselFactorize`.

The commands `WeierstrassPreparation` and `HenselFactorize` factorize univariate polynomials over multivariate power series. Thanks to their implementation based on lazy evaluation, each of these factorization commands returns the factors as soon as enough information is discovered for initializing the data structures of the factors; see Figures 6.5 and 6.6.

The precision of each returned factor, that is, the common precision of its coefficients (which are power series) is zero. However the generator (see Section 6.4 for this term) of each coefficient is known and, thus, the computation of more coefficients can be resumed when a higher precision is requested. Such a request can be explicit by calling `UpdatePrecision`, or implicit, when requesting data of a higher precision than has been previously requested through, e.g., `Truncate` or `HomogeneousPart`.

6.4 Design Principles

In this section we examine several design principles underpinning the implementation of the `MultivariatePowerSeries` library. Foremost is lazy evaluation: an algorithmic technique where the computation of data is postponed until explicitly required (Section 6.4.1). The eventual implementations of these lazy-evaluation algorithms make deliberate efforts to use appropriate Maple data structures and built-in functions to optimize performance (Section 6.4.2). Lastly, in support of software quality and integration with existing Maple library code, we employ Maple's object-oriented mechanisms (Section 6.4.3).

6.4.1 Lazy Evaluation

Lazy evaluation is an optimization technique most commonly appearing in the study of functional programming languages [46]. The lazy evaluation or "call-by-need" refers to delaying the call to a function until its result is genuinely needed. This is often complemented by storing the result for later look-up.

In the case of power series, consider a bivariate geometric series $f = \sum_{d=0}^{\infty} f_{(d)}$ where $f_{(0)} = 1$, $f_{(1)} = x + y$, $f_{(2)} = x^2 + 2xy + y^2$, $\ldots$, $f_{(d)} = (x + y)^d$. One can prove that f converges to $\frac{1}{1-x-y}$. Of course, in practice, it is impossible to store an infinite number of terms on a computer with finite memory. A naïve implementation then suggests storing $f^{(d)}$ for some large and predetermined d. Thus, one can approximate power series as multivariate polynomials. Such an implementation could be called *truncated power series*.

While this representation of power series is easy to implement, it leads to notable restrictions for the study of formal power series. First, one must a priori determine the *precision*, i.e. the particular value of d. Second, in a most naïve implementation, previously-computed homogeneous parts must be recomputed whenever a new, greater precision is required. For example, the polynomial $f^{(d+1)}$ is likely to be constructed "from scratch" despite the polynomial $f^{(d)}$ possibly being already computed. Third, storing and manipulating the polynomial part of a power series up to a degree d needs a large portion of memory. This latter problem is exacerbated when the predetermined precision is not a tight upper bound on the required precision.

To combat the challenges of a truncated power series implementation, we take advantage of lazy evaluation. Every power series is represented by a unique procedure to compute a homogeneous part for a given degree. For example, Listing 6.1 shows such a procedure for the bivariate geometric series which converges to $\frac{1}{1-x-y}$. As we will see, this lazy evaluation design can be paired with an array of polynomials storing the previously computed homogeneous parts.

```
generator := proc(d :: nonnegint)
  return expand((x+y)^d);
end proc;
```

Listing 6.1: A Maple implementation of $f_{(d)}$ in $\frac{1}{1-x-y} = \sum_{d=0}^{\infty} f_{(d)}$.

6.4.2 Maple Data Structures and Built-in Functions

Using an appropriate data structure for encoding and manipulating data is critical for performance, particularly in high-level and interpreted programming languages like Maple. In Maple, modifying an existing list or set—such as by appending, replacing, or deleting an element—may lead to the creation of a new list or set, rather than modifying the original one in-place. In contrast, an `Array` is a low-level and mutable data-structure which allows for in-place modification of its elements. These functionalities provide much better performance than lists or sets when the collection is frequently changed or when the elements being modified are themselves large in size.

Looking more closely at the `Array` data structure, an n-dimensional `Array` is stored as a n-dimensional rectangular block named `RTABLE`. The length of the associated `RTABLE` is $2n+d$ where d is maximum number of elements that may be stored, i.e., the allocation size of the `Array`; see [19, Appendix 1]. For the storage of homogeneous parts of a power series, and the power series coefficients of a UPoPS, we utilize 1-dimensional `Array`s. Listing 6.2 in the next section shows this as the variables `hpoly` and `upoly`, respectively.

To further improve performance, we make use of low-level built-in functions. Such functions are provided as compiled code within the Maple kernel, and therefore not written in the Maple language. Most notably, instead of using Maple for-loops and the typical `+` and `*` syntaxes for addition and multiplication, respectively, we reduce the cost of summations and multiplications remarkably by taking advantage of built-in

MAPLE functions, `add` and `mul`. These built-in functions, respectively, add or multiply the terms of an entire sequence of expressions together to return a single sum or product. These functions avoid a large number of high-level function calls and reduce memory usage by avoiding copying and re-allocation of data.

6.4.3 MAPLE Objects

An often overlooked aspect of MAPLE is its object-oriented capability. An object allows for variables and procedures operating on that data to be encapsulated together in a single entity. In MAPLE, a class—the definition of a particular type of object—can be declared by including the option `object` in a module declaration. Evaluating this declaration returns an object of that class. This new object is often a so-called “prototype” object which, when passed to the `Object` routine, returns a new object of the same class. See [19, Chapter 9] for further details on object-oriented programming in MAPLE.

Our power series and UPoPS types are implemented using these object-oriented features of MAPLE. The classes for each are named, respectively, `PowerSeriesObject` and `UnivariatePolynomialOverPowerSeriesObject`.

The use of object-oriented programming in MAPLE has two key benefits: (*i*) the organization object-oriented code provides better software quality through modularity and maintainability; and (*ii*) allows for the overloading of built-in functions, thus allowing objects to be integrated with, and used natively by, existing MAPLE library functions.

```
MultivariatePowerSeries := module()
option package;
  local
    PowerSeriesObject,
    UnivariatePolynomialOverPowerSeriesObject;

  # create a power series:
  export PowerSeries := proc(...)

  # create a UPoPS:
  export UnivariatePolynomialOverPowerSeries := proc(...)

  #Additional procedures to interface these two classes

  module PowerSeriesObject()
  option object;
    local
```

```
      hpoly :: Array,
      precision :: nonnegint,
      generator :: procedure;
  # other members and methods
  end module;

  module UnivariatePolynomialOverPowerSeriesObject()
  option object;
  local
    upoly :: Array,
    vname :: name;
  # other members and methods
  end module;

end module;
```

Listing 6.2: An overview of the `MultivariatePowerSeries` package.

The `MultivariatePowerSeries` library contains a package of the same name which groups together those two aforementioned classes along with additional procedures to construct and manipulate objects of those classes. These additional procedures are used to "hide" the object-oriented nature of the library behind simple procedure calls. This keeps the package syntactically and semantically consistent with the general paradigm of Maple which does not use object-oriented programming. As an example of such a procedure, `PowerSeries`, as seen in Fig. 6.2 (Section 6.3), handles various different types of input parameters to correctly construct a `PowerSeriesObject` object through delegation to the correct class method.

Listing 6.2 shows the declaration of our two classes and the `MultivariatePower-Series` package. The latter is created by using `option package` in a `module` declaration; see [19, Chapter 8]. The implementation of these two classes is further discussed in Section 6.5.

6.5 Implementation of `MultivariatePowerSeries`

The `MultivariatePowerSeries` package provides a collection of procedures which form simple wrappers for the methods of the aforementioned classes, `PowerSeriesObject` and `UnivariatePolynomialOverPowerSeriesObject`.

These classes, respectively, define the data structures and algebraic functionalities for creating and manipulating multivariate power series and univariate polynomials over power series. This section discusses those data structures as well as the implementation of basic arithmetic, Weierstrass Preparation Theorem, and factorization via Hensel's lemma, all following a lazy evaluation scheme.

6.5.1 `PowerSeriesObject`

The `PowerSeriesObject` class provides basic arithmetic operations, like addition, multiplication, inversion, and evaluation, for multivariate power series, all utilizing lazy evaluation techniques. Let $f \in \mathbf{k}[[X_1, \ldots, X_n]]$ be a non-zero multivariate power series defined as $f = \sum_{d=0}^{\infty} f_{(d)}$. f is encoded as an object of type `PowerSeriesObject`, containing the following attributes.

First, the power series *generator* is the procedure to compute $f_{(d)}$, the d-th homogeneous part of f, for $d \in \mathbb{N}$. Second, the *precision* is a non-negative integer encoding the maximum degree of the homogeneous parts which have so far been computed. Third, the 1-dimensional array storing the previously computed homogeneous parts of f, denoted as `hpoly` in Listing 6.2.

To create a power series object this class provides a variety of constructors. Power series objects may be created from polynomials, algebraic numbers, UPoPS objects, or procedures defining the generator of the power series.

Every arithmetic operation returns a lazily-constructed power series object by creating its generator from the generators of the operands, but without explicitly computing any homogeneous parts of the result. Thus, this is a lazy power series, so that, the homogeneous parts of the result are computed when truly needed. Once homogeneous parts are eventually computed, they are stored in the array `hpoly`. An important aspect of this organization is that the generator of the resulting power series becomes implicitly connected to the generators of the operands; the latter are thus called the *ancestors* of the former.

Moreover, the addition and multiplication operations are not only binary operations (operations taking two parameters), but are m-ary operations. For multiplication, a sequence of power series $f_1, \ldots, f_m \in \mathbf{k}[[X_1, \ldots, X_n]]$ may be passed to the multiplication algorithm to produce the product $f_1 \cdot f_2 \cdots f_m$ via lazy evaluation. Similarly, addition may take the sequence $f_1, \ldots, f_m$ to return the sum $f_1 + f_2 + \cdots + f_m$. Further, addition may also take as a parameter an optional sequence of polynomial coefficients $c_1, \ldots, c_m \in \mathbf{k}[X_1, \ldots, X_n]$ to return the sum $c_1 f_1 + \cdots + c_m f_m$ constructed lazily.

A key part to the efficiency of lazy evaluation is to not re-compute any data. We have already seen that the `hpoly` array stores previously computed homogeneous parts for a `PowerSeriesObject` object. What is missing is to ensure that the array is accessed where possible rather than calling the generator function. Moreover, one must avoid directly accessing that array for homogeneous parts which are not yet computed. We thus provide the function `HomogeneousPart`(f, d), demonstrated in Listing 6.3, to handle both of these cases. This function returns the d-th homogeneous part of the power series f; if d is greater than the *precision* (`f:-precision`), then this method iteratively calls the *generator* to update `hpoly` and `precision`, otherwise it simply returns the previously computed homogeneous part. From here on we use `hpart` as shorthand for the `HomogeneousPart` function.

```
export HomogeneousPart ::static := proc(f, d :: nonnegint)
  if d > f:-precision then
    # resize the hpoly array:
    f:-hpoly(d+1) := 0;
    for local i from f:-precision + 1 to d do
      f:-hpoly[i] := f:-generator[i];
    end do;
    f:-precision := d;
  end if;
  return f:-hpoly[d];
end proc;
```

Listing 6.3: A simplified version of the `HomogeneousPart` function in `PowerSeries-Object`.

Listing 6.4 shows a simplified implementation of `Divide` that computes the quotient of two power series objects $f, g \in \mathbf{k}[[X_1, \ldots, X_n]]$. In particular, notice the creation of the local procedure `gen` for the generator of the quotient. Note that `EXPAND` is a local macro defined in `MultivariatePowerSeries` to efficiently perform expansion and normalization supporting *algebraic* inputs.

```
export Divide ::static := proc(f, g)
  if hpart(g,0)=0 then
    error "invalid input: not invertible";
  end if;

  local h := Array(0..0,EXPAND(hpart(f,0)/hpart(g,0)));

  local gen := proc(d :: nonnegint)
```

```
    local s := hpart(f,d);
    s -= add(EXPAND(hpart(g,i)*hpart(f,d-i)),i=1..d);
    return EXPAND(s/hpart(g,0));
  end proc;

  return Object(PowerSeriesObject,h,0,gen);
end proc;
```

Listing 6.4: A simplified version of the division method in `PowerSeriesObject`.

6.5.2 UnivariatePolynomialOverPowerSeriesObject

The `UnivariatePolynomialOverPowerSeriesObject` class is implemented as a simple dense univariate polynomial with the simple and obvious implementations of associated arithmetic (see, e.g., [100, Chapter 2]). The arithmetic operations are achieved directly from coefficient arithmetic, that is, `PowerSeriesObject` arithmetic. Since the latter is implemented using lazy evaluation techniques, UPoPS arithmetic is inherently and automatically lazy.

For example, the addition of two UPoPS objects $f = \sum_{i=0}^{k} a_i X_{n+1}^i$ and $g = \sum_{i=0}^{k} b_i X_{n+1}^i$ in $\mathbf{k}[[X_1, \ldots, X_n]][X_{n+1}]$ is the summation $(a_i + b_i)X_{n+1}^i$ for all $0 \leq i \leq k$, where a_i, b_i are `PowerSeriesObject` objects. Other basic arithmetic operations behave similarly. However, there are important operations on UPoPS which are not as straightforward. In the following we explain our implementation of Weierstrass Preparation Theorem, Taylor shift, and factorization via Hensel's lemma for UPoPS, all of which follow lazy evaluation techniques.

Weierstrass Preparation

Let $f, p, \alpha \in \mathbf{k}[[X_1, \ldots, X_n]][X_{n+1}]$ be such that they satisfy the conditions of Theorem 18 and such that $f = \sum_{i=0}^{d+m} a_i X_{n+1}^i$, $p = {X_{n+1}}^d + \sum_{i=0}^{d-1} b_i X_{n+1}^i$, and $\alpha = \sum_{i=0}^{m} c_i X_{n+1}^i$. Equating coefficients in $f = p\alpha$ we derive the two following systems of equations:

$$\begin{cases} a_0 & = & b_0 c_0 \\ a_1 & = & b_0 c_1 + b_1 c_0 \\ & \vdots & \\ a_{d-1} & = & b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \end{cases} \tag{6.1}$$

$$\begin{cases} a_d & = \quad b_0c_d + b_1c_{d-1} + \cdots + b_{d-1}c_1 + c_0 \\ & \vdots \\ a_{d+m-1} & = \quad b_{d-1}c_m + c_{m-1} \\ a_{d+m} & = \quad c_m \end{cases} \tag{6.2}$$

To solve these systems we proceed by solving them modulo successive powers of $\mathcal{M}$, following the proof of Theorem 18 in [22]. Notice that solving modulo successive powers of $\mathcal{M}$ is precisely the same as computing homogeneous parts of increasing degree. Thus, this follows our lazy evaluation scheme perfectly. The power series $b_0, \ldots, b_{d-1}$ are generated by Equations (6.1) and $c_0, \ldots, c_m$ by Equations (6.2).

Consider that $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ are known modulo $\mathcal{M}^r$ while $a_0, \ldots, a_{d-1}$ are known modulo $\mathcal{M}^{r+1}$; this latter fact is simple since f is the input to Weierstrass Preparation and is fully known. From the first equation in (6.1), b_0 can be computed modulo $\mathcal{M}^{r+1}$ since $b_0 \in \mathcal{M}$, c_0 is known modulo $\mathcal{M}^r$, and a_0 is known $\mathcal{M}^{r+1}$. Then, the equation $a_1 = b_0c_1 + b_1c_0$, that is, $a_1 - b_0c_1 = b_1c_0$ can be solved for b_1 modulo $\mathcal{M}^{r+1}$ since, again, $b_1 \in \mathcal{M}$ and the other terms are sufficiently known. We compute all $b_2, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$ with the same argument. After determining $b_0, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$, we can compute $c_m, c_{m-1}, \ldots, c_0$ modulo $\mathcal{M}^{r+1}$ from Equations (6.2) with simple power series multiplication and subtraction, working iteratively, in a bottom up fashion. For example, $c_{m-1} = a_{d+m-1} - b_{d-1}c_m$.

As yet, we have not explicitly seen how the coefficients of p and α will be updated. The key idea is that to update a single power series coefficient of p or α requires simultaneously updating all coefficients of p and α. Thus, all the generators of $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ simply call a single "Weierstrass update" function to update all power series simultaneously using Equations (1) and (2). Algorithm 26 shows this Weierstrass update function.

In order to update the coefficients of p, we frequently need to compute $a_i - \sum_{j=0}^{i-1} b_jc_{i-j}$ for $0 \leq i < d$. To optimize this operation, we a priori create helper power series as the set $\mathcal{F} = \{F_i \mid F_i = a_i - \sum_{j=0}^{i-1} b_jc_{i-j}, i = 0, \ldots, d-1\}$. The power series F_i, following power series arithmetic with lazy evaluation, allows for the efficient computation of homogeneous parts of increasing degree of $a_i - \sum_{j=0}^{i-1} b_jc_{i-j}$. This set $\mathcal{F}$ is passed to the Weierstrass update function to optimize the overall computation.

Finally, the Weierstrass preparation must be initialized before continuing with Weierstrass updates. Namely, the degree of p and the initial values of p and α modulo $\mathcal{M}$ must first be computed. The degree of p, namely d, is set to be the smallest integer i such that a_i is a unit. If $d = 0$, then $p = 1$ and $\alpha = f$, otherwise, m equals the difference

Algorithm 26 WEIERSTRASSUPDATE$(p, \alpha, \mathcal{F}, r)$

```
Require: $p = X_{n+1}^d + \sum_{i=0}^{d-1} b_i X_{n+1}^i$, $\alpha = \sum_{i=0}^{m} c_i X_{n+1}^i$, $r \in \mathbb{N}$, and $\mathcal{F} = \{F_i \mid F_i = a_i - \sum_{j=0}^{i-1} b_j c_{i-j}, 0 \le i < d\}$ are all known modulo $\mathcal{M}^r$
Ensure: $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ modulo $\mathcal{M}^{r+1}$
    # update $b_0, ..., b_{d-1}$ modulo $\mathcal{M}^{r+1}$
1: for $i$ from $0$ to $d-1$ do
2:     $s :=$ add(seq(hpart($b_i, r-k$) $\cdot$ hpart($c_0, k$), $k = 1 .. r-1$));
3:     hpart($b_i, r$) := (hpart($F_i, r$) $-$ $s$)/hpart($c_0, 0$);
    # ensure $c_0, ..., c_m$ are updated modulo $\mathcal{M}^{r+1}$
4: for $i$ from $0$ to $m$ do
5:     hpart($c_i, r$);
```

between the degree of f and d, and we initialize $b_i = 0$ for $0 \le i < d$. Then, $c_m, \ldots, c_0$ are initialized using power series arithmetic following Equations (6.2). Lastly, the set $\mathcal{F}$ is initialized.

Taylor Shift

This operation takes a UPoPS object $f \in \mathbf{k}[\![X_1, \ldots, X_n]\!][X_{n+1}]$ and performs the translation $X_{n+1} \to X_{n+1} + c$, i.e. $f(X_{n+1} + c)$, for some $c \in \mathbf{k}$. In our implementation, c can be a `numeric` or `algebraic` MAPLE type with the purpose of being used efficiently in factorization via Hensel's Lemma.

Assume $f = \sum_{i=0}^{k} a_i X_{n+1}^i$ is a UPoPS in $\mathbf{k}[\![X_1, \ldots, X_n]\!][X_{n+1}]$ and $c \in \mathbf{k}$. As the `PowerSeriesObject` objects $a_0, \ldots, a_k$ are lazily evaluated power series, we want to also make Taylor shift a lazy operation. Thus, we need to create a generator for the power series coefficients of $f(X_{n+1} + c)$. Let $\mathfrak{T} = (t_{i,j})$ be the lower triangular matrix of the coefficients of ${X_{n+1}}^j$ in the binomial expansion $(X_{n+1} + c)^i$, for $0 \le i \le k$, and $0 \le j \le i$. Let $(b_0, \ldots, b_k)$ be the list of coefficients of $f(X_{n+1} + c)$ in $\mathbf{k}[\![X_1, \ldots, X_n]\!]$. Then, it is easy to prove that for every $0 \le i \le k$, b_i is the inner product of the i-th sub-diagonal of $\mathfrak{T}$ with the lower $k + 1 - i$ elements of the vector $(a_0, \ldots, a_k)$. This inner product can be computed efficiently by taking advantage of the m-ary addition operation described for the `PowerSeriesObject` (see Section 6.5.1). Since this operation returns a lazily-constructed power series, this precisely defines the lazy construction of the power series $b_0, \ldots, b_k$, thus making Taylor shift a lazy operation.

Factorization via Hensel's Lemma

Hensel's lemma for factorizing univariate polynomials over power series was reviewed in Theorem 6.2.1, where $\mathbf{k}$ is algebraically closed and $f \in \mathbf{k}[[X_1, \ldots, X_n]][X_{n+1}]$ is a UPoPS object. Following the ideas of [22], we compute the factors of f in a lazy fashion. Algorithm 27 proceeds through iterative applications of Taylor shift and Weierstrass Preparation Theorem in order to create one factor of f at a time. Those factors are actually computed through lazy evaluation thanks to the lazy behavior of the procedures WEIERSTRASSPREPARATION and TAYLORSHIFT. This Algorithms thus computes and updates the factors modulo the successive powers $\mathcal{M}, \mathcal{M}^2, \mathcal{M}^3, \ldots$ of the maximal ideal $\mathcal{M}$.

Algorithm 27 HENSELFACTORIZE(f)

```
Require: $f = \sum_{i=0}^{k} a_i {X_{n+1}}^i \in \mathbf{k}[[X_1, \ldots, X_n]][X_{n+1}]$
Ensure: A list of factors $\{f_1, \ldots, f_r\}$ so that $f = a_k f_1 \cdots f_r$ and Theorem 6.2.1
 1: if $a_k \notin \mathcal{M}$ then
 2:     $f^* := \frac{1}{a_k} f$;
 3: else
 4:     error "$a_k$ must be a unit."
 5: $\bar{f}$ := EVALUATEATORIGIN($f^*$);
 6: $c_1, \ldots, c_r$ := ROOTS($\bar{f}, X_{n+1}$);
 7: for $i$ from 1 to $r$ do
 8:     $g$ := TAYLORSHIFT($f^*, c_i$);
 9:     $p, \alpha$ := WEIERSTRASSPREPARATION($g$);
10:     $f_i$ := TAYLORSHIFT($p, -c_i$);
11:     $f^*$ := TAYLORSHIFT($\alpha, -c_i$);
12: return $\{f_1, \ldots, f_r\}$;
```

Note that the generation of the factors $f_1, \ldots, f_r$ takes place after factorizing $\bar{f} \in \mathbf{k}[X_{n+1}]$. Recall that $\bar{f}$ is obtained by evaluating each X_i to 0 for $1 \leq i \leq n$. This is called EVALUATEATORIGIN in our implementation. To efficiently factor $\bar{f}$, we take advantage of the package `SolveTools` [94], which allows us to compute the splitting field of $\bar{f}$ (which, in practice, is a polynomial with coefficients in some algebraic extension of $\mathbb{Q}$) and factorize $\bar{f}$ into linear factors.

Let $c_1, \ldots, c_r$ be the distinct roots of $\bar{f}$ and $k_1, \ldots, k_r$ their respective multiplicities. To describe one iteration of Algorithm 27, let f^* be the current polynomial to factorize.

For a root c_i of $\bar{f}$, and thus f^*, we perform a Taylor shift to obtain $g = f^*(X_{n+1} + c_i)$. Then, we apply Weierstrass preparation on g to obtain p and α where p is monic and of degree k_i. Again, by using Taylor Shift, we apply the reverse shift to p to obtain $f_i = p(X_{n+1} - c_i)$, a factor of f, and $f^* = \alpha(X_{n+1} - c_i)$, for the next iteration. As mentioned above, since both Taylor shift and Weierstrass preparation are implemented using lazy evaluation, our factorization via Hensel's lemma is inherently lazy.

6.6 Experimentation

We compare the performance of the `MultivariatePowerSeries` package, denoted `MPS`, with the previous Maple implementation of multivariate power series, the `PowerSeries` package, denoted `RCPS`, and the recent implementation of power series via lazy evaluation in the BPAS library. This latter implementation is written in the C language on top of efficient sparse multivariate arithmetic; see [12, 22]. It has already been shown in [22] that the implementation in BPAS is orders of magnitude faster than the `PowerSeries` package, Maple's `mtaylor` command, and the multivariate power series available in SageMath. As we will see, our implementation performs comparably to that of BPAS.

Throughout this section, we collect our benchmarks on a machine running Ubuntu 18.04.4, Maple 2020, and BPAS (ver. 1.652), with an Intel Xeon X5650 processor running at 2.67GHz, with 12x4GB DDR3 memory at 1.33 GHz.

Figures 6.7, 6.8, and 6.9, respectively, show the performance of division and multiplication algorithms to compute $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for power series $f_1 = 1 + X_1 + X_2$, $f_2 = 1 + X_1 + X_2 + X_3$, and $f_3 = 2 + \frac{1}{3}(X_1 + X_2)$. It can be seen that `MPS` power series division is 9×, 2100×, and 3× faster than the previous Maple implementation for f_1, f_2, and f_3 respectively. The speed-ups for multiplication are significantly higher. Moreover, `MPS` results are comparable with the C implementation of similar algorithms in BPAS. Figure 6.10 then highlights the efficiency of m-ary addition (see Section 6.5.1), compared to iterative applications of binary addition. Recall that m-ary addition is exploited in the Weierstrass preparation algorithm.

Next, we compare the performance of Weierstrass preparation (Section 6.5.2). Figures 6.11 and 6.12 demonstrate the running time of this algorithm for two different UPoPS. Looking at these results, we can see a 2200× speed-up in comparison with the similar algorithm in `RCPS` and timings comparable to BPAS.

We also compare the factorization via Hensel's lemma and Taylor shift algorithms for a set of UPoPS $f = \prod_{i=1}^{k}(X_2 - i) + X_1(X_2^{k-1} + X_2)$ in $\mathbf{k}[\![X_1]\!][X_2]$ with $k = 3, 4$ in Figures 6.13 and 6.14. Our factorization implementation is orders of magnitude faster than that of `RCPS`. However, factorization performs worse than expected compared to BPAS, having already seen comparable performance of Weierstrass preparation in Figures 6.11 and 6.12. This difference can be attributed to Taylor shift, the other core operation of HenselFactorize, as seen in Figure 6.14. The implementation in `MPS` is slower than the same procedure in BPAS by several order of magnitude. This, in turn, can be attributed to using Maple matrix arithmetic, rather than the direct manipulation of C-arrays as in BPAS, within the Taylor shift algorithm.

Conclusion and Future Work

Throughout this work we have discussed the object-oriented design and implementation of power series and univariate polynomials over power series following lazy evaluation techniques. Basic arithmetic operations for both are examined as well as Weierstrass Preparation Theorem, Taylor shift, and factorization via Hensel's lemma for univariate polynomials over power series. Our implementation in Maple is orders of magnitude faster than the existing multivariate power series implementation in the `PowerSeries`

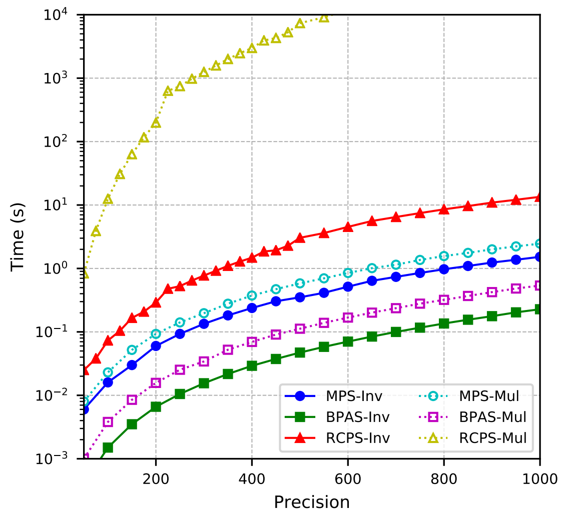


Figure 6.7: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_1 = 1 + X_1 + X_2$.

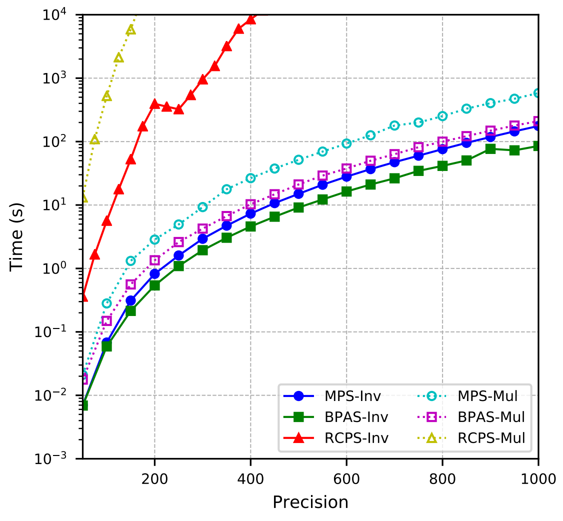


Figure 6.8: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_2 = 1 + X_1 + X_2 + X_3$.

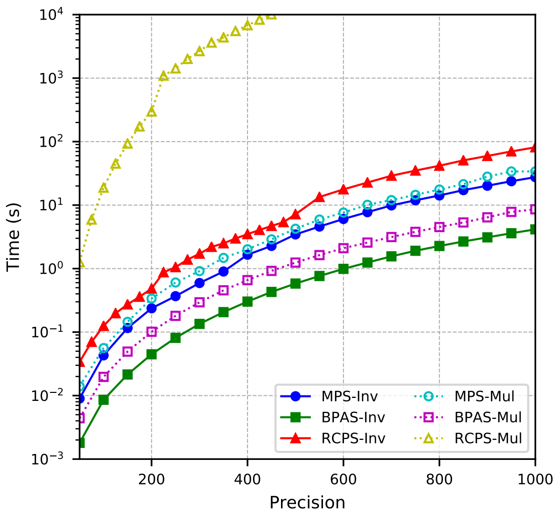


Figure 6.9: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_3 = 2 + \frac{1}{3}(X_1 + X_2)$.

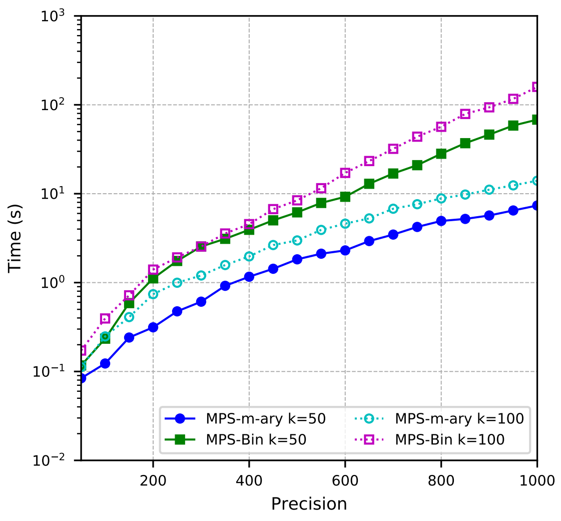


Figure 6.10: Computing $f = \sum_{i=1}^{k} \frac{1}{1-x-y}$ using m-ary and binary addition.

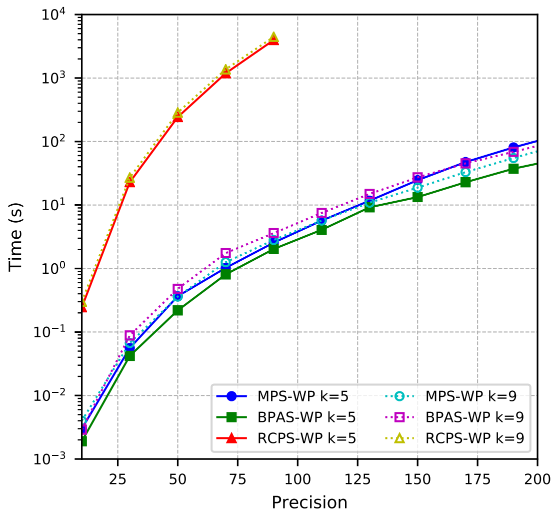


Figure 6.11: Computing Weierstrass preparation of $f_1 = \frac{1}{1+X_1+X_2} {X_3}^k + {X_3}^{k-1} + \cdots + X_2X_3 + X_1 \in \mathbf{k}[\![X_1, X_2]\!][X_3]$.

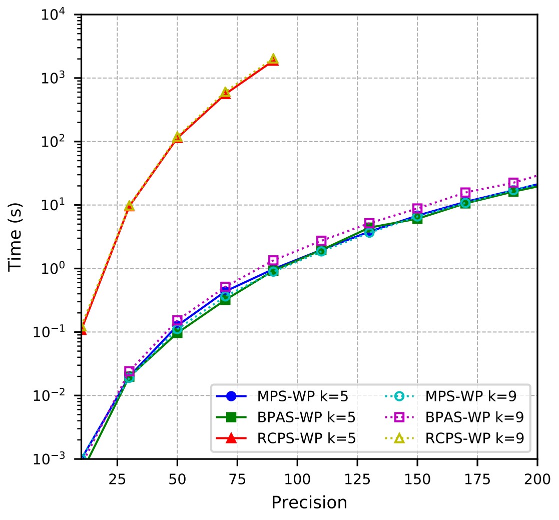


Figure 6.12: Computing Weierstrass preparation of $f_2 = \frac{1}{1+X_1+X_2} {X_3}^k + X_2{X_3}^{k-1} + \cdots + X_3 + X_1 \in \mathbf{k}[\![X_1, X_2]\!][X_3]$.

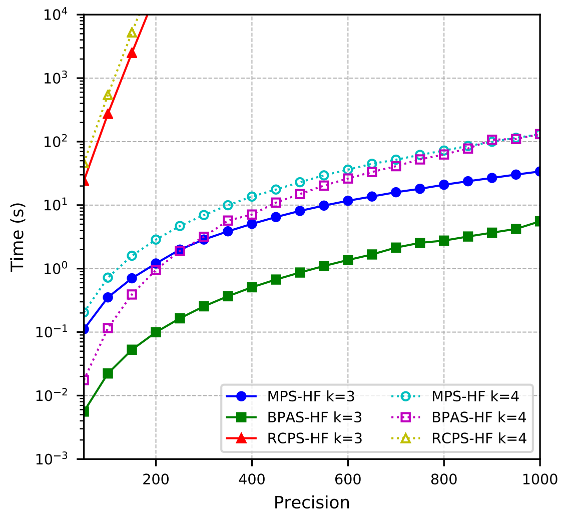


Figure 6.13: For $f = \prod_{i=1}^{k}(X_2 - i) + X_1({X_2}^{k-1} + X_2)$, computing HenselFactorize(f).

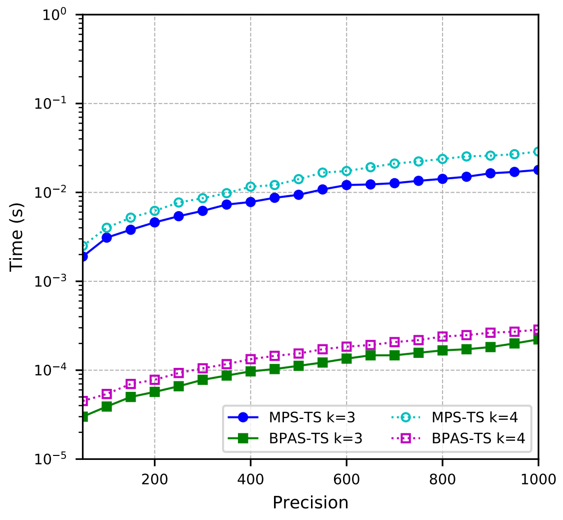


Figure 6.14: For $f = \prod_{i=1}^{k}(X_2 - i) + X_1({X_2}^{k-1} + X_2)$, computing TaylorShift$(f, 1)$.

package of the `RegularChains` library. Moreover, our implementation is comparable with the C implementation of power series and univariate polynomials over power series in BPAS.

Further work is needed to extend lazy evaluation techniques to more sophisticated algorithms. For example, a division algorithm based on Newton's method, a general Extended Hensel Construction (EHC) [82], and the Abhyankar-Jung Theorem [85]. As a consequence, it is possible to re-implement the EHC algorithm found in `RegularChains` using this library. Further, as Maple supports multithreading, it is possible to apply parallel processing to our algorithms. In particular, the computation of UPoPS coefficients in Weierstrass preparation is embarrassingly parallel. Meanwhile, the successive application of Weierstrass preparation and Taylor shift in HenselFactorize present an opportunity for *pipelining*. Both should be exploited in to achieve even further performance improvements.

Chapter 7

Conclusion and Future Work

In this research, we have optimized, designed and developed algorithms to compute subresultants in support of developing a high-performance polynomial system solving. We have also examined the performance of these schemes in the C/C++ BPAS library and integrated them to work with the multithreaded BPAS solver.

In Chapter 3, we designed speculative algorithms taking advantage of fast arithmetic over prime fields for univariate and bivariate polynomials over prime fields and arbitrary-precision integers. These schemes address how subresultants are usually used by the `Triangularize` algorithm. These speculative schemes yield speed-up factor of $7\times$ and $2\times$ for polynomials in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively.

To develop this speculative strategies for univariate and bivariate polynomials, we extended a well-known divide-and-conquer algorithm naming *Half-GCD*. In addition, we made use of asymptotically fast algorithms to implement basic arithmetic on polynomials over prime fields with following best-practices in NTL and ALDOR. These subresultant algorithms based on fast arithmetic are up to $10\times$ and $400\times$ faster than non-modular schemes, e.g. Ducos' algorithm, for polynomials in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively.

In the case of subresultant chain algorithms based on evaluation and interpolation schemes, we utilized the BPAS multithreaded interface to implement multithreaded modular and speculative subresultant algorithms for bivariate polynomials over prime fields and integers. The integration of these schemes in the multithreaded BPAS solver yields up to $4\times$ speed-up on a 12-core machine.

In Chapter 4, we were interested on optimizing algorithms for a *recursive* and *sparse* representation of multivariate polynomials over arbitrary-precision integers. We deployed a recursive normal form algorithm by making use of an efficient heap-based division operation, and designed a multi-divisor pseudo-division algorithm where the set of divisors is a strongly *normalized* triangular set. The latter routine is respectively up to 20× and 3.5× faster than a naïve implementation in Maple and BPAS.

We further optimized a well-studied scheme known as Ducos' subresultant chain algorithm. This optimization relies intensively on in-place basic arithmetic and efficient memory access patterns. This optimized Ducos' algorithm is 2× faster than its counterpart in Maple. Moreover, for univariate polynomials of degree as large as 2000, the optimized Ducos' subresultant chain algorithm uses 3.2× and 11.7× less memory, respectively, than our implementation of the original Ducos' algorithm in BPAS and the implementation of Ducos' algorithm in Maple.

In Chapter 5, we studied subresultant algorithms based on computing the determinant of (Hybrid) Bézout matrices. We made use of the Bareiss fraction-free LU decomposition to compute determinant and further optimized this decomposition algorithm via smart-pivoting technique and using BPAS multithreaded interface. To compute the resultant of two sparse polynomials with 6 variables, our subresultant algorithm based on Hybrid Bézout matrix is 24× faster than the optimized Ducos' algorithm.

Furthermore, we designed speculative algorithms in order to compute subresultants based on Hybrid Bézout matrix. For two sparse polynomials with 6 variables, computing S_0, S_1 speculatively is around 2× faster than computing them successively. We also showed that the integration of these schemes in the BPAS solver yields up to 1.6× speed-up to solve several real-world multivariate polynomial systems.

In Chapter 6, we discussed our implementation of power series which is available as the `MultivariatePowerSeries` library in Maple 2021. This library provides arithmetic for formal power series and univariate polynomials over such series (UPoPS) by taking advantage of object-oriented mechanisms in Maple and lazy evaluation techniques. The comprehensive results indicated that `MultivariatePowerSeries` is comparable in performance to the C implementation of multivariate power series in BPAS that is thus similarly several orders of magnitude faster than other existing implementations including the previous Maple package for multivariate power series.

7.1 Future Work

Research on subresultant chain algorithms in order to further optimizing the polynomial system solver via the `Triangularize` algorithm can be continued in the following four directions in the near future,

(i) For *dense* univariate and bivariate polynomials, one could use the big prime field FFT approach. In [28], a FFT-based multiplication of two integers modulo Generalized Fermat Prime Fields (GFPF) of size 8 or 16 machine words on GPUs has introduced. One of the primary results in this paper is that computing over a single *big* prime field can outperform computing over several small (of size one machine word) prime fields. Thus, subresultant algorithms based on modular methods can take advantage of primes larger than one machine-word size following ideas in [28, 30].

(ii) For polynomials with more than 2 variables, but sparse, one should consider sparse evaluation and interpolation algorithms. This problem was originally studied by Zippel in [104] to compute GCDs and later by Monagan [78] to implement the Hensel lifting step of a multivariate polynomial factorization. Furthermore, one can implement speculative subresultant algorithms for such polynomials with taking advantage of sparse evaluation-interpolation and fast modular arithmetic.

(iii) For subresultant algorithms based on the (Hybrid) Bézout matrix, one should consider optimizations such as,

- computing determinants of such matrices modulo several *small* primes, or one *big* prime to take advantage of fast modular arithmetic; and
- specializing one or more variables in order to optimize smart pivoting technique through reducing the complexity of finding proper pivots.
- using different pivoting strategies like "Rook's pivoting" [86] in the FFLU decomposition algorithm and different formulations for computing Bézout matrices such as "polynomial by values" [7].

(iv) For computing a subresultant chain in $\mathcal{A}[y]$ where,

$$\mathcal{A} := \mathrm{Frac}(\mathbb{B}[x_1, \ldots, x_v]/\mathrm{sat}(T)),$$

and T is a regular chain of $\mathbb{B}[x_1, \ldots, x_v]$, practically, there are two approaches:

(a) computing the corresponding subresultant chain in $\mathbb{B}[x_1, \ldots, x_v][y]$ and apply the *specialization property of subresultants* (as in this thesis); or

(b) computing the corresponding subresultant chain directly in $\mathcal{A}[y]$ (as discussed in [72]).

If T is a zero-dimensional regular chain, it is worth revisiting (b) since better implementations of normal form algorithms are available today. In addition, where T is positive dimensional, one can keep expression swelling under control and use the second approach with taking advantage of more advanced algorithmic techniques such as methods in order to reduce the input positive dimensional regular chain to zero-dimensional regular chains.

Bibliography

[1] https://github.com/orcca-uwo/MultivariatePowerSeries, 2021.

[2] Jounaidi Abdeljaoued, Gema M. Diaz-Toca, and Laureano Gonzalez-Vega. Minors of Bézout matrices, subresultants and the parameterization of the degree of the polynomial greatest common divisor. *International Journal of Computer Mathematics*, 81(10):1223–1238, 2004.

[3] Jounaidi Abdeljaoued, Gema M. Diaz-Toca, and Laureano González-Vega. Bézout matrices, subresultant polynomials and parameters. *Applied Mathematics and Computation*, 214(2):588–594, 2009.

[4] Rafał Abłamowicz. Some applications of Gröbner bases in robotics and engineering. In *Geometric Algebra Computing*, pages 495–517. Springer, 2010.

[5] Martin Albrecht and Carlos Cid. Algebraic techniques in differential cryptanalysis. In *International Workshop on Fast Software Encryption*, pages 193–208. Springer, 2009.

[6] Parisa Alvandi, Mahsa Kazemi, and Marc Moreno Maza. Computing limits with the regularchains and powerseries libraries: From rational functions to zariski closure. *ACM Communications in Computer Algebra*, 50(3):93–96, 2016.

[7] Dhavide A. Aruliah, Robert M. Corless, Laureano Gonzalez-Vega, and Azar Shakoori. Geometric applications of the bezout matrix in the lagrange basis. In *Proceedings of the 2007 International Workshop on Symbolic-Numeric Computation*, SNC '07, page 55–64, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Mohammadali Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, Lin-Xiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), 2021. http://www.bpaslib.org.

[9] Mohammadali Asadi, Alexander Brandt, Mahsa Kazemi, Marc Moreno Maza, and Erik J. Postma. Multivariate power series in maple. In Robert M. Corless, Jürgen Gerhard, and Ilias S. Kotsireas, editors, *Maple in Mathematics Education and Research*, pages 48–66, Cham, 2021. Springer International Publishing.

[10] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. Parallelization of triangular decompositions: Techniques and implementation. *Journal of Symbolic Computation*, 2021 (to appear).

[11] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. Sparse polynomial arithmetic with the BPAS library. In *Computer Algebra in Scientific Computing, CASC 2018, Lille, France, Proceedings*, pages 32–50, 2018.

[12] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. Algorithms and data structures for sparse polynomial arithmetic. *Mathematics*, 7(5):441, May 2019.

[13] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. On the parallelization of triangular decompositions. In Ioannis Z. Emiris and Lihong Zhi, editors, *ISSAC '20: International Symposium on Symbolic and Algebraic Computation, Kalamata, Greece, July 20-23, 2020*, pages 22–29. ACM, 2020.

[14] Mohammadali Asadi, Alexander Brandt, and Marc Moreno Maza. Computational schemes for subresultant chains. In *Computer Algebra in Scientific Computing, CASC 2021, Sochi, Russia (to appear)*, 2021.

[15] Philippe Aubry, Daniel Lazard, and Marc Moreno Maza. On the theories of triangular sets. *Journal of Symbolic Computation*, 28(1-2):105–124, 1999.

[16] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of the F5 Gröbner basis algorithm. *Journal of Symbolic Computation*, 70:49–70, 2015.

[17] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of computation*, 22(103):565–578, 1968.

[18] Eberhard Becker, Teo Mora, Maria Grazia Marinari, and Carlo Traverso. The shape of the shape lemma. In *Proc. of ISSAC 1994*, pages 129–133. ACM, 1994.

[19] Laurent Bernardin, Paulina Chin, Paul DeMarco, Keith O. Geddes, K. Michael Heal, George Labahn, John P. May, James McCarron, Michael B. Monagan, Darin Ohashi, and Stefan M. Vorkoetter. *Maple Programming Guide*. Maplesoft, a division of Waterloo Maple Inc., 1996-2020.

[20] Dario Bini and Victor Y. Pan. *Polynomial and matrix computations: fundamental algorithms*. Springer Science & Business Media, 2012.

[21] Maxime Bôcher and Edmund Pendleton Randolph Duval. *Introduction to higher algebra*. Macmillan, 1922.

[22] Alexander Brandt, Mahsa Kazemi, and Marc Moreno Maza. Power series arithmetic with the BPAS library. In *Computer Algebra in Scientific Computing (CASC)*, pages 108–128. Springer, 2020.

[23] Richard P. Brent and Hsiang T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM (JACM)*, 25(4):581–595, 1978.

[24] Manuel Bronstein, Marc Moreno Maza, and Stephen M Watt. Generic programming techniques in ALDOR. In *Proceedings of AWFS 2007*, pages 72–77, 2007.

[25] Bruno Buchberger. Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal. *PhD thesis, Universitat Insbruck*, 1965.

[26] William H. Burge and Stephen M. Watt. Infinite structures in scratchpad II. In *European Conference on Computer Algebra*, pages 138–148. Springer, 1987.

[27] Changbo Chen and Marc Moreno Maza. Algorithms for computing triangular decomposition of polynomial systems. *Journal of Symbolic Computation*, 47(6):610–642, 2012.

[28] Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. Big prime field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 85–92, New York, NY, USA, 2017. Association for Computing Machinery.

[29] George E. Collins. Subresultants and reduced polynomial remainder sequences. *Journal of the ACM (JACM)*, 14(1):128–142, 1967.

[30] Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza, and Lin-Xiao Wang. Big prime field FFT on multi-core processors. In *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019*, pages 106–113, 2019.

[31] Svyatoslav Covanov and Marc Moreno Maza. Putting Fürer algorithm into practice. Technical report, 2014.

[32] Xavier Dahan and Éric Schost. Sharp estimates for triangular sets. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 103–110, New York, NY, USA, 2004. Association for Computing Machinery.

[33] Jean Della Dora, Claire Dicrescenzo, and Dominique Duval. About a new method for computing in algebraic number fields. In *Proc. of EUROCAL 1985, vol. 2*, volume 204 of *LNCS*, pages 289–290, 1985.

[34] Gema M. Diaz-Toca and Laureano Gonzalez-Vega. Various new expressions for subresultants and their applications. *Applicable Algebra in Engineering, Communication and Computing*, 15(3-4):233–266, 2004.

[35] Jack J. Dongarra, James W. Demmel, and Susan Ostrouchov. LAPACK: A linear algebra library for high-performance computers. In Yadolah Dodge and Joe Whittaker, editors, *Computational Statistics*, pages 23–28, Heidelberg, 1992. Physica-Verlag HD.

[36] Lionel Ducos. Algorithme de bareiss, algorithme des sous-résultants. *ITA*, 30(4):319–347, 1996.

[37] Lionel Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145(2):149–163, 2000.

[38] Jean-Charles Faugère. Parallelization of Gröbner basis. In *Parallel Symbolic Computation PASCO 1994 Proceedings*, volume 5, page 124. World Scientific, 1994.

[39] Jean-Charles Faugère. FGb: A library for computing Gröbner bases. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 84–87. Springer, 2010.

[40] Akpodigha Filatei, Xin Li, Marc Moreno Maza, and Eric Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 93–100, 2006.

[41] Gerd Fischer. *Plane algebraic curves*. American Mathematical Society, 2001.

[42] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *ISSAC*, pages 167–174, 2007.

[43] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for computer algebra*. Springer Science & Business Media, 1992.

[44] Walter Habicht. Eine verallgemeinerung des sturmschen wurzelzählverfahrens. *Commentarii Mathematici Helvetici*, 21(1):99–116, 1948.

[45] Sardar Anisul Haque, Xin Li, Farnam Mansouri, Marc Moreno Maza, Wei Pan, and Ning Xie. Dense arithmetic over finite fields with the CUMODP library. In *International Congress on Mathematical Software*, pages 725–732. Springer, 2014.

[46] Robert Harper. *Practical foundations for programming languages: Lazy Evaluation*, pages 323–332. Cambridge University Press, 2nd edition, 2016.

[47] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, and Nathaniel J. Smith. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[48] Xiaorong Hou and Dongming Wang. Subresultants with the Bézout matrix. In *Computer Mathematics*, pages 19–28. World Scientific, 2000.

[49] Wolfram Research Inc. Mathematica, 2020. `www.wolfram.com/mathematica`.

[50] David John Jeffrey. LU factoring of non-invertible matrices. *ACM Communications in Computer Algebra*, 44(1/2):1–8, 2010.

[51] Stephen C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, 8(3):63–71, 1974.

[52] M'hammed El Kahoui. An elementary approach to subresultants theory. *Journal of Symbolic Computation*, 35(3):281–292, 2003.

[53] Michael Kalkbrener. A generalized euclidean algorithm for computing triangular representations of algebraic varieties. *Journal of Symbolic Computation*, 15(2):143–167, 1993.

[54] Jerzy Karczmarczuk. Generating power of lazy semantics. *Theoretical Computer Science*, 187(1-2):203–219, 1997.

[55] Michael Kerber. Division-free computation of subresultants using Bézout matrices. *International Journal of Computer Mathematics*, 86(12):2186–2200, 2009.

[56] Donald E. Knuth. The analysis of algorithms. *The Actes du Congrés International des Mathématiciens*, 3:269–274, 1970.

[57] Emanuel Lasker. Zur theorie der moduln und ideale. *Mathematische Annalen*, 60(1):20–116, 1905.

[58] Daniel Lazard. A new method for solving algebraic systems of positive dimension. *Discrete Applied Mathematics*, 33(1-3):147–160, 1991.

[59] Daniel Lazard. Thirty years of polynomial system solving, and now? *Journal of Symbolic Computation*, 44(3):222–231, 2009. Polynomial System Solving in honor of Daniel Lazard.

[60] Grégoire Lecerf. Computing the equidimensional decomposition of an algebraic closed set by means of lifting fibers. *Journal of Complexity*, 19(4):564–596, 2003.

[61] Grégoire Lecerf. On the complexity of the Lickteig-Roy subresultant algorithm. *Journal of Symbolic Computation*, 92:243–268, 2019.

[62] Hong R. Lee and David Saunders. Fraction free Gaussian elimination for sparse matrices. *Journal of symbolic computation*, 19(5):393–402, 1995.

[63] Derrick H. Lehmer. Euclid's algorithm for large numbers. *The American Mathematical Monthly*, 45(4):227–233, 1938.

[64] Francois Lemaire, M Moreno Maza, and Yuzhen Xie. The RegularChains library in Maple. *ACM SIGSAM Bulletin*, 39(3):96–97, 2005.

[65] Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: From theory to practice. *Journal of Symbolic Computation*, 44(7):891–907, 2009.

[66] Xin Li, Marc Moreno Maza, and Wei Pan. Computations modulo regular chains. In *ISSAC 2009, Seoul, Republic of Korea, Proceedings*, pages 239–246, 2009.

[67] Yong-Bin Li. A new approach for constructing subresultants. *Applied Mathematics and Computation*, 183(1):471–476, 2006.

[68] Thomas Lickteig and Marie-Françoise Roy. Cauchy index computation. *Calcolo*, 33(3-4):337–351, 1996.

[69] Thomas Lickteig and Marie-Françoise Roy. Semi-algebraic complexity of quotients and sign determination of remainders. *Journal of Complexity*, 12(4):545–571, 1996.

[70] Henri Lombardi, Marie-Françoise Roy, and Mohab Safey El Din. New structure theorem for subresultants. *Journal of symbolic computation*, 29(4-5):663–690, 2000.

[71] Maplesoft, a division of Waterloo Maple Inc. Maple 2020. `www.maplesoft.com/`.

[72] Marc Moreno Maza and Renaud Rioboo. Polynomial gcd computations over towers of algebraic extensions. In Gérard D. Cohen, Marc Giusti, and Teo Mora, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 11th International Symposium, AAECC-11, Paris, France, July 17-22, 1995, Proceedings*, volume 948 of *Lecture Notes in Computer Science*, pages 365–382. Springer, 1995.

[73] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.

[74] Carl D. Meyer. *Matrix analysis and applied linear algebra*, volume 71. Siam, 2000.

[75] Michael Monagan. Probabilistic algorithms for computing resultants. In *Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 245–252, 2005.

[76] Michael Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In *CASC 2007*, pages 295–315. Springer, 2007.

[77] Michael Monagan and Roman Pearce. Sparse polynomial division using a heap. *Journal of Symbolic Computation*, 46(7):807–822, 2011.

[78] Michael Monagan and Baris Tuncer. Factoring multivariate polynomials with many factors and huge coefficients. In *International Workshop on Computer Algebra in Scientific Computing*, pages 319–334. Springer, 2018.

[79] Michael Monagan and Paul Vrbik. Lazy and forgetful polynomial arithmetic and applications. In *Computer Algebra in Scientific Computing, 11th International Workshop, CASC 2009, Proceedings*, pages 226–239, 2009.

[80] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[81] Marc 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.

[82] Marc Moreno Maza. Polynomials over power series and their applications to limit computations, Tutorial at Computer Algebra in Scientific Computing (CASC), 2018. `www.csd.uwo.ca/~mmorenom/Publications/Polynomials_over_power_series_and_their_applications_lecture.PDF`.

[83] Marc Moreno Maza and Wei Pan. Solving bivariate polynomial systems on a GPU. *Journal of Physics: Conference Series*, 341, 2011.

[84] Emmy Noether. Idealtheorie in ringbereichen. *Mathematische Annalen*, 83(1):24–66, 1921.

[85] Adam Parusiński and Guillaume Rond. The Abhyankar–Jung theorem. *Journal of Algebra*, 365:29–41, 2012.

[86] George Poole and Larry Neal. The Rook's pivoting strategy. *Journal of Computational and Applied Mathematics*, 123(1):353–369, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[87] Daniel Reischert. Asymptotically fast computation of subresultants. In *Proc. of ISSAC 1997*, pages 233–240. ACM, 1997.

[88] Joseph Fels Ritt. *Differential equations from the algebraic standpoint*, volume 14. American Mathematical Soc., 1932.

[89] Massimiliano Sala. Gröbner bases, coding, and cryptography: a guide to the state-of-art. In *Gröbner Bases, Coding, and Cryptography*, pages 1–8. Springer, 2009.

[90] Arnold Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica 1*, pages 139–144, 1971.

[91] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[92] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[93] Victor Shoup. NTL: A library for doing number theory, 2021. `www.shoup.net/ntl/`.

[94] The Maple Developers. `SolveTools` package in Maple 2020. Maplesoft, a division of Waterloo Maple Inc. `www.maplesoft.com/support/help/Maple/view.aspx?path=SolveTools`.

[95] The Sage Developers. *SageMath, the Sage Mathematics Software System*, 2020. `www.sagemath.org`.

[96] Klaus Thull and Chee-Keng Yap. A unified approach to hgcd algorithms for polynomials and integers. 1990.

[97] Joris van der Hoeven. Relax, but don't be too lazy. *Journal of Symbolic Computation*, 34(6):479–542, 2002.

[98] Joris van der Hoeven, Grégoire Lecerf, and Bernar Mourrain. Mathemagix, from 2002. `http://www.mathemagix.org`.

[99] Bartel Leendert van der Waerden. *Algebra*. Springer, 1960.

[100] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, NY, USA, 3 edition, 2013.

[101] Wu Wen-Tsún. A zero structure theorem for polynomial-equations-solving and its applications. In *European Conference on Computer Algebra*, pages 44–44. Springer, 1987.

[102] Lu Yang and Jingzhong Zhang. Searching dependency between algebraic equations: an algorithm applied to automated reasoning. 1990.

[103] Chee-Keng Yap. *Fundamental problems of algorithmic algebra*, volume 49. Oxford University Press Oxford, 2000.

[104] Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pages 216–226, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

Curriculum Vitae

Name:	**Mohammadali Asadi**
Post-Secondary Education and Degrees:	University of Western Ontario London, Ontario, Canada 2017 - 2021 Ph.D.
	Isfahan University of Technology Isfahan, Iran (2010 - 2014 B.Sc.) (2014 - 2017 M.Sc.)
Selected Awards and Honours:	Western Graduate Research Scholarship for Ph.D. in Computer Science 2017 - 2021
	Mitacs Research and Development Scholarship for Internship at Maplesoft Summer 2020
	Ranked first (based on GPA) among B.Sc. (2014) and M.Sc. (2016) students of Department of Mathematical Science
Related Work Experience:	Teaching Assistant University of Western Ontario 2017 - 2021

Research Assistant
University of Western Ontario
2017 - 2021

R&D Software Developer Intern
Maplesoft, Waterloo Maple Inc.
Summer 2020

Refereed Software: M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, Lin-Xiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), `http://www.bpaslib.org`

M. Asadi, A. Brandt, M. Kazemi, M. Moreno Maza, and E. J. Postma. `MultivariatePowerSeries` in Maple 2021, `https://github.com/orcca-uwo/MultivariatePowerSeries`

Related Publications:	M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza; *Sparse polynomial arithmetic with the BPAS library.* CASC 2018, pp. 32-50, Springer, 2018.
	M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. *Algorithms and data structures for sparse polynomial arithmetic.* MDPI Journal: Mathematics, 7(5):441, May 2019.
	M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. *On the parallelization of triangular decompositions.* ISSAC 2020, pp. 22-29, ACM, 2020.
	M. Asadi, A. Brandt, M. Kazemi, M. Moreno Maza, and E. J. Postma. *Multivariate power series in maple.* Maple in Mathematics Education and Research, pp. 48-66, Springer 2021.
	M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. *Parallelization of triangular decompositions: Techniques and implementation.* Journal of Symbolic Computation, 2021. (to appear)
	M. Asadi, A. Brandt, and M. Moreno Maza. *Computational schemes for subresultant chains.* CASC 2021, pp. 21-41, Springer, 2021.