Implementation of Fast Polynomial Arithmentic in Aldor

(Thesis format: Monograph)

by

Akpodigha **Filatei**

Graduate Program in Computer Science

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

Faculty of Gratuate Studies
University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Supervisor Dr. Marc Moreno Maza	Examiners Dr. Stephen Watt
Supervisory Committee	Dr. Lucian Ilie
	Dr. Graham Denham

The thesis by
Akpodigha Filatei
entitled:

Implementation of Fast Polynomial Arithmetic in Aldor

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

Date

Chair of Examining Board

Abstract

To solve large mathematical problems in science and engineering, there is an increasing need for both faster implementations of the well-studied classical algorithms, and, new faster algorithms in computer algebra. The theoretical asymptotic complexity of these algorithms are mostly known but achieving comparable practical complexity (efficiency) is a challenge. Our mission is to implement algorithms for fast polynomial computations over a generic coefficient ring. Implementations of fast algorithms in low-level languages have been widely investigated but the same is not true for implementations in high-level languages. Considering that it is easier for mathematicians with limited computer science knowledge to reuse and build on top of implementations in high-level languages than in low-level languages, we use a high-level programming environment for our implementation. We also write generic code that can be useful in a wider variety of applicaitons.

Our main focus is on fast algorithms for which implementation techniques remain to be investigated, such as the fast Extended Euclidean Algorithm (EEA). Our main application is fast algorithms for GCDs over fields and field extensions which are major tools for polynomial system solving.

In addition, we approach this by exploring in depth, the algorithmic ideas, and, use of a multitude of tricks and special relations to achieve practical efficiency that favourably compare the known theoretical complexity bounds.

In order to achieve high performance:

- we identify polynomial multiplication and power series inversion as the fundamental efficiency-critical operations that are often encountered in polynomial arithmetic and carefully implement the fastest known algorithms for these operations - FFT for polynomial multiplication and Newton iteration for power series inversion:
  - we take advantage of the optimizer of Aldor where necessary by adopting best coding practices,
  - we work in a lower level manner for manipulating arrays of coefficients rather than abstract data type for polynomials in the implementation of crucial operations such as the FFT-based multiplication and power series inversion,
  - we efficiently manage the memory, for instance, by pre-allocating needed memory before hand in order to avoid memory allocation during loop iterations.

- we compare the performance of our implementations with known related implementations in other computer algebra systems like NTL by V. Shoup and MAGMA.

- based on these efficiency-critical operations, we implement other algorithms from the book, and obtain competitive implementations,

- our FFT-based polynomial multiplication and power series inversion is only twice slower than highly optimized and non-generic code of NTL (written in C++); we explain why this is a very satisfactory result.

We choose ALDOR as our implementation and experimentation environment because ALDOR is a leading, highly optimized programming language.

Keywords: Aldor, Symbolic Computation, Computer Algebra System, Polynomial Arithmetic, FFT, FCRA, Polynomial Factorization.

Contents

1 Introduction **1**
1.1 Objectives . . . 1
1.2 Related Work . . . 2
1.3 Implementation Environment . . . 3
1.4 Achievements . . . 4

2 Background **8**
2.1 What are Fast Algorithms for Polynomial Arithmetic? . . . 8
2.2 What is ALDOR? . . . 10
2.3 Why are Fast Algorithms for Polynomial Arithmetic Challenging to Implement . . . 12

3 The FFT-based Multiplication and its Implementation in ALDOR **13**
3.1 Principle . . . 14
3.2 Primitive Root of Unity Finder . . . 16
3.3 Empirical Timings of FFT vs Karatsuba-based Multiplication . . . 20
3.4 Generic ALDOR Implementation vs Non-generic Optimized C++(NTL) and MAGMA Implementations . . . 21
3.5 Generic ALDOR Implementation vs its Translation in NTL . . . 21
3.6 Implementation Techniques and ALDOR Source Code of the FFT . . . 22

4 Power Series Inversion Using the Newton Iteration **27**
4.1 Principle . . . 28
4.2 What is Newton Iteration? . . . 28
4.3 The Middle Product Technique . . . 30
4.4 Implementation Techniques and Empirical Results . . . 33
4.5 Fast Monic Division with Remainder . . . 34

5 Subproduct Tree Techniques and Applications **37**
5.1 The Subproduct Tree and its Implementation . . . 37
5.2 Application: Simultaneous Reduction . . . 38
5.3 Application: Fast Chinese Remaindering Algorithm . . . 41

6 Half-GCD over a Field **44**
6.1 Standard Extended Euclidean Algorithm . . . 44

6.2 The Half-GCD Principle 48
6.3 Fast Extended Euclidean Algorithm - Benchmarks with NTL and MAGMA 52

7 Coprime Factorization **54**
7.1 Principle and Implementation 54

8 Conclusion **60**
Appendices 61

A FFT Multiplication Package **62**

B Fast Power Series Inversion **65**

C Naive Version of Power Series Inversion **67**

D Fast Chinese Remaindering Algorithm **68**

E Coprime Factorization **72**

F Fast Monic Division **76**

G Fast Extended Euclidean Algorithm **77**

List of Figures

1.1 FFT Multiplication vs Karatsuba Modulo a 27-bit Prime Field. . . . 6
1.2 Power Series Inversion: Naive vs Optimized vs Polynomial Multiplications Modulo a 27-bit Prime Field 6
1.3 Fast CRA vs Standard CRA in ALDOR Modulo a 27-bit Prime Field. 7

3.1 FFT-based Univariate Polynomial Multiplication for Two Polynomials with Degree less than n Over $\mathbb{Z}/p\mathbb{Z}$ 15
3.2 DFT-Reordering for an 8-element Array. 18
3.3 Generic GCD vs Specialized GCD in ALDOR Modulo a 27-bit Prime. 22
3.4 FFT: Our ALDOR Implementation vs. C++(NTL) Optimized Implementation Modulo a 27-bit Prime. 23
3.5 FFT: Our ALDOR Implementation vs. MAGMA implementation Modulo a 27-bit Prime. 24
3.6 FFT: Our ALDOR implementation vs. NTL's vs MAGMA's Modulo a 64-bit Prime. 25
3.7 FFT: ALDOR Implementation .vs. NTL Translation. 26

4.1 Power Series Inversion: The Middle Product Technique to Compute T Efficiently. 31
4.2 Power Series Inversion: Naive vs. Optimized Implementation vs. Multiplication Modulo a 27-bit Prime. 34
4.3 Power Series Inversion: Space Usage of Naive vs. Optimized Implementations Modulo a 27-bit Prime. 35
4.4 Fast Division vs Classical Division vs Multiplication Modulo a 27-bit Prime. 36

5.1 Subproduct Tree For 10 Points. 39
5.2 Subproduct Tree creation ($2d$ points) versus Multiplication(d*d) Modulo a 27-bit Prime. 40
5.3 CRA .vs. Fast CRA Modulo a 27-bit Prime. 41

6.1 Euclidean Algorithm in ALDOR: Standard .vs. Fast Modulo a Prime 27-bit 52
6.2 EEA: ALDOR vs. NTL vs. MAGMA Modulo a 27-bit Prime. 53

List of Algorithms

1 DFT: Iterative Algorithm for the DFT. 16
2 iDFT: Iterative Algorithm for the Inverse DFT. 17
3 FFTMult: Algorithm For The Convolution of Two Polynomials. . . . 17
4 Power Series Inversion of f to Precision ℓ 29
5 Middle Product Technique for the Newton Iteration in Power Series Inversion of f to Precision ℓ. 32
6 Fast Division with Remainder Algorithm 36
7 The Classical Euclidean Algorithm. 45
8 The Extended Euclidean Algorithm. 46
9 The Half-GCD Algorithm. 50
10 The MGCD Algorithm. 51
11 The Fast Extended Euclidean Algorithm. 51
12 multiGCD: Multiple GCD Computation Algorithma. 56
13 allPairGcd: All Pairs of GCD computation Algorithm 56
14 mergeCoprimeFactorization: Algorithm to merge coprime factorization of two given lists. 57
15 CPF: The Coprime Factorization Algorithm. 58

Chapter 1

Introduction

Today, it is common practice to assume that a new algorithm, say for GCD computations over algebraic number fields or field extensions, can rely on asymptotically fast polynomial multiplication. Therefore, it is desirable not only to offer implementations of asymptotically fast arithmetic, but also programming environments for developing new such algorithms. In addition, it is also a demand to achieve this goal in the context of high-level programming languages, where new ideas can be tested quickly and where algorithms can be easily made generic. In this chapter, we state our objectives, explore works of others that are related to ours and introduce the programming environment. We also give a summary of our achievements so far.

1.1 Objectives

A successful attack on a problem in science and engineering usually leads to some equations that have to be solved. "In principle, there are two ways of solving such equations: approximately or exactly" [47]. Computer Algebra is a recent area of computer science that serves as a computational tool for the exact solution of constructive problems in mathematics; numerical analysis on the other hand, provides highly successful mathematical methods and computer software to calculate approximate solutions [47, 45]. Questions of efficiency have hampered the development of exact methods. This is despite the fact that, exact solutions provide more interesting answers to problems than traditional numerical methods. We aim to contribute to efforts to overcome this challenge of efficiency of exact solutions to problems in science and engineering by implementing fast polynomial algorithms in a high-level programming language. Our code is generic - for instance, our implementation of the fast Extended Euclidean Algorithm [51] can work with polynomials for coefficients over the rationals $\mathbb{Q}$ or in $\mathbb{Z}/_p\mathbb{Z}$ (for some prime p).

1.2 Related Work

Asymptotically fast algorithms for exact polynomial and matrix arithmetic have been known for more than forty years. Among others, the work of Karatsuba [32], Cooley and Tukey [11], and Strassen [46] has initiated an intense activity in this area. Unfortunately, its impact on computer algebra systems has been limited until recently. One reason was, probably, the belief that these algorithms were of very limited practical interest. In [24] p. 132, referring to [37], the authors state that the FFT-based univariate polynomial multiplication is "better than the classical method approximately when $n + m \geq 600$", where n and m are the degrees of the input polynomials. In [33] p. 501, quoting [6], Knuth writes "He (R. P. Brent) estimated that Strassen's scheme would not begin to excel over Winograd's until $n \approx 250$ and such enormous matrices rarely occur in practice unless they are very sparse, when other techniques apply."

On the other hand, the implementation of asymptotically fast arithmetic was not the primary concern of the early computer algebra systems, which had many other challenges to face. For instance, one of the main motivations for the development of the AXIOM computer algebra system [30] was the design of a language where mathematical properties and algorithms could be expressed in a natural and efficient manner. Nevertheless, successful implementations of the FFT-based univariate polynomial multiplication [37] and Strassen's matrix multiplication [3] have been reported for several decades.

In the last decade, several software for performing symbolic computations have put a great deal of effort in providing outstanding performances, including successful implementation of asymptotically fast arithmetic. As a result, the general-purpose computer algebra system MAGMA [29] and the Number Theory Library NTL [43, 44] have set world records for polynomial factorization and determining orders of elliptic curves. The book, *Modern Computer Algebra* [47] (which serves a primary reference for most of our algorithms) has also contributed to increase the general interest of the computer algebra community for these algorithms.

As part of continuing efforts to improve on the practical efficiency of polynomial arithmetic operations in computer algebra systems, Xin Li [35] in his 2005 masters thesis work investigated techniques for efficient implementation of polynomials based on the FFT techniques. He used the AXIOM [30] computer algebra system as the platform. In Xin Li's work, codes written at high, middle and low level are combined in order to achieve high performance. We, on the other hand, remain at high-level only. Furthermore, his codes (unlike ours) work mainly for polynomials over finite fields, and so, are not generic. Another related work is BIPOLAR developed by the authors of [47]. BIPOLAR works only over F_2, the field with two elements.

1.3 Implementation Environment

ALDOR [2, 50] is the primary experimentation and implementation platform for our work. We have however, also implemented translations of some of our code in NTL.

The ALDOR language was designed at IBM by Prof. Stephen Watt in the early 90's, to be an extension language for Scratchpad (the programming language of AXIOM, in which computer algebra algorithms could be naturally and efficiently implemented. The current ALDOR libraries are split into dedicated libraries; the `libaldor` library gives access to machine types and the `libalgebra` library provides arithmetic for fundamental algebraic objects like polynomials and matrices. Today, the ALDOR language and libraries are developed at the `Ontario Research Center for Computer Algebra` [28], which supports this thesis. More details about ALDOR is in Section 2.2 of this report.

1.4 Achievements

We have realized a highly optimized implementation of the FFT-based univariate polynomial multiplication and power series inversion. Indeed, these operations are essential for other fast algorithms such as the Fast Chinese Remaindering Algorithm (CRA), Fast Extended Euclidean Algorithm (FEEA) and coprime factorization. In fact, it is based on these efficiency-critical operations that we have implemented the fast CRA, FEEA and the coprime factorization algorithms. We conducted experiments to determine crossover points at which known naive implementations of the algorithms that existed in the ALDOR `libalgebra` library become less efficient in time and space than our implementations. We have also conducted empirical comparisons between our implementations and similar implementations in other popular computer algebra systems like NTL and MAGMA. Furthermore, we have done comparisons of the time performance of our implementations with known theoretical complexities. These experiments and their analyses have served as benchmarks for our efforts to achieve practically efficient polynomial arithmetic and asymptotically faster (than classical counterparts) algorithms. These enrich the ALDOR programming language and further serve as a contribution to the computer algebra community on the achievement of optimal practical efficiency in a high-level programming environment.

A list of the algorithms we have implemented follows:

1. FFT-based multiplication over a generic coefficient ring that supports the FFT,
2. power series inversion over a field using Newton iteration and,
3. fast monic division,
4. subproduct tree techniques (these are used in several fast algorithms based on the divide-and-conquer strategy),
5. fast polynomial GCD with Bézout coefficients over a generic field,
6. fast Chinese remaindering algorithm and,

7. coprime factorization

Our ALDOR implementation of FFT-based multiplication over a generic coefficient ring that supports the FFT, for instance, out-performs the existing Karatsuba [32] multiplication (in ALDOR) as the input polynomial degree exceeds 150 as Figure 1.1 shows. Even though, Xin Li in [35] makes reference to some better cut-offs, the Karatsuba in ALDOR's `libalgebra` library is highly optimized which may explain this result. We have also implemented power series inversion using the Newton iteration method both naively and efficiently (by a combination of computation-saving techniques like the "middle-product" [25] and change of data representation).

Figure 1.2 showing the timings versus polynomial degrees show how the two versions compare. Multiplication and power series inversion are efficiency-critical operations and so we have expended special effort on them.

Another example is in Figure 1.3 which shows how the result of our implementation of the Chinese Remaindering Algorithm (CRA) compares with that of the already existing (in `libaldor`) naive/standard implementation. See Chapters 3 and 4 for details on our implementation of the FFT and power series inversion algorithms.

Furthermore, in Chapter 5 is a discussion of the subproduct tree implementation which forms the basis for the divide-and-conquer techniques employed in this work. Chapter 6 details our implementation of a faster extended euclidean algorithm for computing GCDs and their Bézout coefficients while Chapter 7 discusses a new coprime factorization algorithm and its implementation.

Source code of the algorithms we implemented is given in the appendix. All timings given are obtained on a bi-Pentium 4, 2.80GHz machine, with 1Gb of RAM.

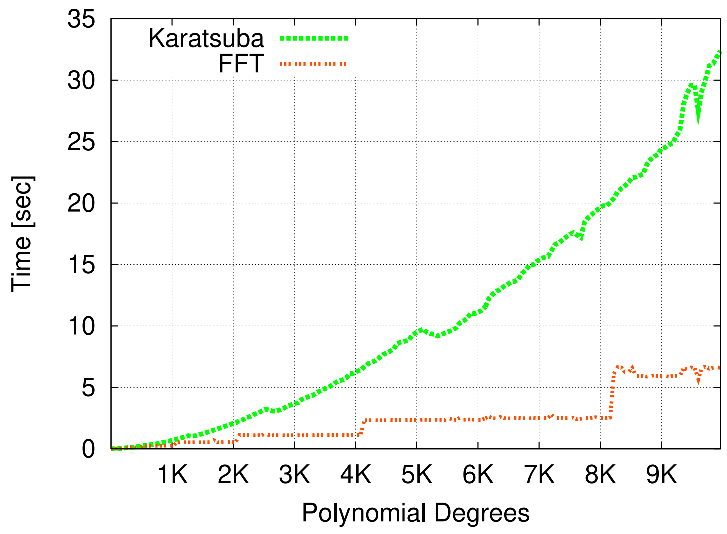

Figure 1.1: FFT Multiplication vs Karatsuba Modulo a 27-bit Prime Field.

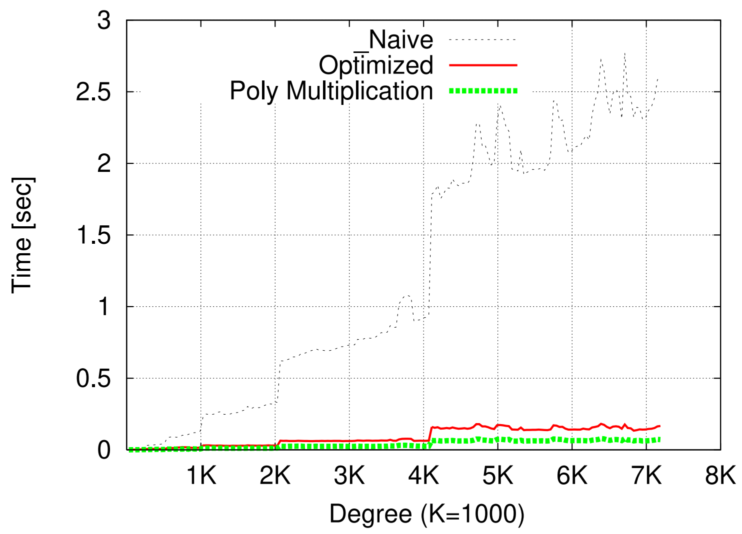

Figure 1.2: Power Series Inversion: Naive vs Optimized vs Polynomial Multiplications Modulo a 27-bit Prime Field

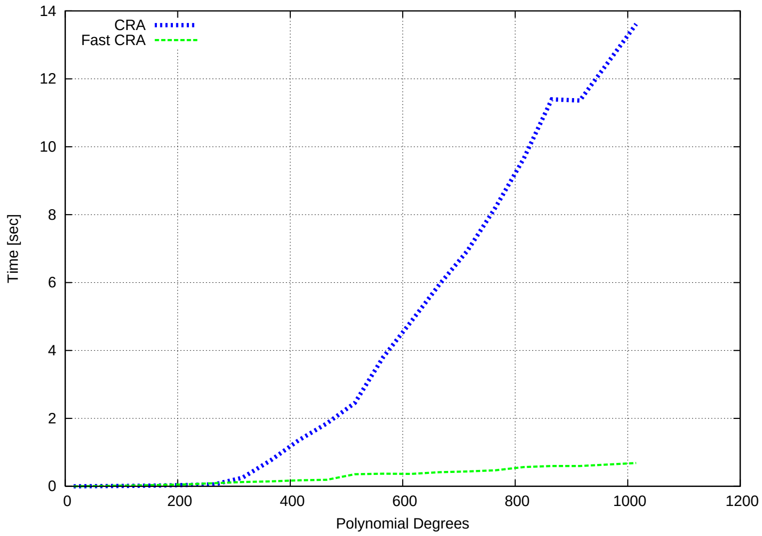

Figure 1.3: Fast CRA vs Standard CRA in ALDOR Modulo a 27-bit Prime Field.

Chapter 2

Background

In this chapter, we give background information about fast algorithms for polynomial arithmetic, the challenges with their implementation and the implementation environment. In the first section, we define fast algorithms as algorithms that spend almost linear time to do a computation despite the input size. In the second section, we present an overview of our implementation environment and in the third section, we give reasons why the asymptotically fast algorithms are challenging to implement.

2.1 What are Fast Algorithms for Polynomial Arithmetic?

Fast algorithms for polynomial arithmetic are algorithms for operations such as multiplication, division, GCD and resultants, interpolation and factorization, that have a running time complexity quasi-linear in the size of the input, counting only the number of operations in the coefficient ring. Recall that a $f(d)$ is quasi-linear if $f(d) \in O(d^{1+\epsilon})$ for every $\epsilon > 0$. That is, cost of the operation does not increase significantly with increasing size of the input. The FFT-based univariate polynomial multiplication [10] is an example of such fast algorithm. In addition, it is the basis for many other fast algorithms [15]. Simply put, if multiplication can be done quickly (in fewer steps), then many of the algorithms in polynomial arithmetic can be performed quickly. This leads to the following concept.

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

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

Examples of multiplication times are:

- Classical: $2d^2$;
- Karatsuba: $C\, d^{\log_2(3)}$ with some C that can be taken equal to 9;
- FFT over an arbitrary ring: $C\, d \log(d) \log(\log(d))$ for some C that can taken equal to 64 [10].

Note that the FFT-based multiplication in degree d over a ring that supports the FFT (that is, possessing primitive n-th root of unity, where n is a power of 2 greater than $2d$) can run in $C\, d \log(d)$ operations in R, with some $C \geq 18$. The complexity estimates with respect to M of some of the fast algorithms of interest to us are recalled from [15]:

1. Dividing in $R[x]$ a polynomial of degree less than $2d$ by a monic polynomial of degree at most d requires at most $5\,\mathsf{M}(d) + O(d)$ operations $(+, \times, \div)$ in R.
2. Additions and multiplications in $R[x]/F$ requires at most $6\,\mathsf{M}(d) + O(d)$ operations $(+, \times, \div)$ in R. Where F is a monic polynomial.
3. With $F_1, .., F_s$ a non-constant monic polynomial in $R[x]$, with sum of degrees d, then one can compute the subproduct tree associated to $F_1, ..., F_s$ using at most $\mathsf{M}(d) \log(d)$ operations in R (see Chapter 5).
4. Assume that R is a field, then, given two polynomials in $R[x]$ of degree at most d, computing their monic GCD and their Bézout coefficients can be done in no more than $33\,\mathsf{M}(d) \log(d) + O(d \log(d))$ operations $(+, \times, \div)$ in R.

2.2 What is Aldor?

The Aldor language was designed at IBM by Prof. Stephen Watt in the early 90's to be an extension language for Scratchpad (the programming language for AXIOM) in which computer algebra algorithms could be naturally and efficiently implemented. The current Aldor libraries are split into dedicated libraries viz, `libaldor` and `libalgebra`. The `libaldor` library gives access to machine types and the `libalgebra` library provides arithmetic for fundamental alegbraic objects like polynomials and matrices. Today, the Aldor language and libraries are developed at the `Ontario Research Center for Computer Algebra` [28] which supports this thesis. Aldor [2] is a high-level computer programming language, like C++ or Java. Aldor is type-complete, strongly-typed and has a two-level object model of categories and domains that is similar to the concept of interfaces and classes respectively in Java. In Aldor, types and functions can be constructed and manipulated within its programs like the way values are manipulated. It allows programs to use a natural style, combining the more attractive and powerful properties of functional, object-oriented and aspect-oriented programming styles. The following quotation, taken from *http://www.aldor.org*, captures the important capabilities of the language.

> The formulation of the programming language has tried to balance the mathematical desire for generality and uniformity, on one hand, with the practical requirements of the most demanding symbolic and numeric computation, on the other. For example, types and functions are first class values, and dependent types are fully supported, but the precise formulation has been carefully arranged to allow effective optimizaton. Post facto extensions allow libraries to be built in layers by enriching existing objects, rather than introducing a proliferation of variations. Both object-oriented and functional programming styles are reconstructed naturally form these basic semantic elements.

An example of a generic aldor code extracted from a package in `libalgebra` is:

```
interpolate(p:List R, m:List R):R == {
		import from SI;
		assert(#p=#m);
		assert(#p>0);
		a := first p; p := rest p;
		M := first m; m := rest m;
		while ~empty? p repeat {
			np := first p; p := rest p;
			nm := first m; m := rest m;
			a  := combine(M, nm)(a, np);
			M  := nm * M;
		}
		a;
	}

combine(M:R, m:R):(R,R) -> R == {
		import from Integer, Partial R;
		TRACE("computing inverse of ", M rem m);
		TRACE("modulo ",m);
		s := retract diophantine(M rem m, 1, m);
		mover2:R := { mustBalance? => m quo (2::R); 0 };
		TRACE("result=",s);
		(A:R, a:R):R +-> {
			b := (s * (a - (A rem m))) rem m;
			-- the following is only for integer-like rings
			if mustBalance? and greater?(b, mover2) then b := b - m;
			A + b * M;
		}
	}
```

The preceding function *interpolate* takes as input two lists; m, a set of moduli, and, p, a set of remainders of type R where R is defined as an *Euclidean Domain*. It produces an element f of type R such that $f \bmod p_i = m_i$ for all i. Please see [2] for more details about ALDOR.

2.3 Why are Fast Algorithms for Polynomial Arithmetic Challenging to Implement

Algorithms for fast polynomial arithmetics are challenging to implement because:

- they usually require change of data representation,
- their complexity estimates generally have large constant factors, as recalled above,
- for most of them, such as the Half-GCD, management of recursive calls are needed.

The change of data representation and the recursive calls imply extra operations, especially at the memory management level. On a high-level language, this is even more challenging as we have to efficiently utilize the data structures and primitive types provided by the libraries and the compiler. In addition, experience shows that a variety of algorithms have to be coded and tested to determine the input sizes at which one algorithm beats the other (crossover points). Say, for multiplication, the classical method is best for small inputs, Karatsuba [32] for medium sizes and FFT-based methods excel for larger problems.

Chapter 3

The FFT-based Multiplication and its Implementation in ALDOR

The Discrete Fourier Transform (DFT) algorithm was known to Gauss but reinvented by Cooley and Turkey in 1965 [47]. The DFT is a fourier transform which operates on a discrete rather than continuous function. The Fast Fourier Transform (FFT) implements the DFT quickly. It converts a signal representation from the time-domain to the frequency domain. This chapter presents an application of the DFT to fast polynomial multiplication, and its implementation. The first section gives the principle behind the DFT algorithms, the second section discusses a probabilistic primitive root of unity finder, while the third provides empirical timings of performance of our implementation of the FFT-based multiplication versus ALDOR's Karatsuba-based [32] multiplication. The fourth section gives further empirical results of the comparisons between our FFT implementation in ALDOR and implementations in NTL and MAGMA computer algebra systems while the fifth section compares our implementation in ALDOR with a translation of our codes into C++ using the NTL library. The last section gives a summary of the techniques we adopted and the function signatures of the source code that implement the FFT-based multiplication. See Appendix A for the entire source code.

3.1 Principle

The Fast Fourier Transform (FFT) [1, ch. 7] provides an efficient method for evaluating a polynomial at definite roots of unity in a commutative ring. It is the basis of a well-developed algorithm for multiplication.

Let R be a ring (commutative with 1) and let's denote by R^n, the set of the "vectors" of length n with coordinates in R. A univariate polynomial $f(x) \in R[x]$ of degree less than n, with coefficients in R can be represented by a "vector" $A \in R^n$ such that elements of A are its coefficients. Evaluation of a polynomial induces a homomorphism from the polynomial ring to the ring of coefficients [37]. In fact, the FFT method uses the vector $A = [a_0, ..., a_{n-1}]$ of coefficients of the polynomial $f(x)$ and evalutes $f(x)$ at the special points $[1, \omega, ..., \omega^{n-1}]$ through a divide-and-conquer scheme, where ω is a primitive n-th root of unity of $f(x)$. To recover $f(x)$ from the evaluated points resulting from the FFT, an inverse FFT exists which produces output similar to interpolation of $f(x)$ given its values at those points.

The FFT-based multiplication which is reputed to be the fastest multiplication method for polynomials uses the aforementioned concept to produce the product of two polynomials in quasi-linear time in the size of the input. It does this by computing the FFT of the vector representation of each of the two polynomials followed by a point-wise multiplication of the resulting vectors and finally by computing the inverse FFT of the latter vector.

The name *Discrete Fourier Transform* (DFT) derives from this application of the FFT to evaluate a polynomial at discrete points. The principles of the FFT-based univariate polynomial multiplication are sketched in Figure 3.1.

In the chart of Figure 3.1, we consider two polynomials $f = \sum_{k=0}^{k=n-1} a_k x^k$ and $g = \sum_{k=0}^{k=n-1} b_k x^k$ over some field $\mathbb{K}$. We do not need to assume that they have the same degree; if they do not have the same degree, we add a "zero leading coefficient" to the one of smaller degree. We want to compute the product $fg = \sum_{k=0}^{k=2n-2} c_k$. The classical algorithm would compute the coefficient c_k of fg by

$$c_k = \sum_{i=0}^{i=k} a_i b_{k-i} \tag{3.1}$$

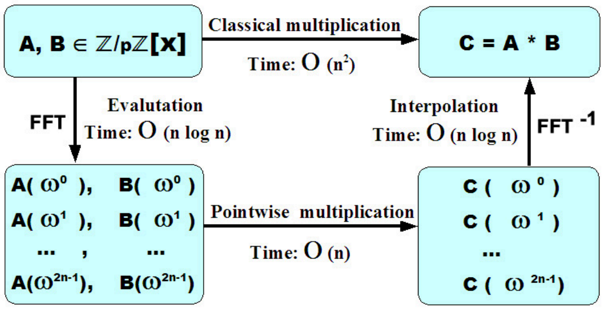

Figure 3.1: FFT-based Univariate Polynomial Multiplication for Two Polynomials with Degree less than n Over $\mathbb{Z}/p\mathbb{Z}$

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

The first idea in the FFT-based univariate polynomial multiplication is as follows: if the values of f and g are known at $2n-1$ different points of $\mathbb{K}$, say $x_0, \ldots, x_{2n-2}$ then we can obtain fg by computing $f(x_0)g(x_0), \ldots, f(x_{2n-2})g(x_{2n-2})$ amounting to $O(n)$ operations in $\mathbb{K}$.

The second idea is to use points $x_0, \ldots, x_{2n-1}$ in $\mathbb{K}$ such that:

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

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

Such points $x_0, \ldots, x_{2n-2}$ do not always exist in $\mathbb{K}$. However, there are techniques to overcome this limitation (essentially by considering a field extension of $\mathbb{K}$ where the desired points can be found). In the end, this leads to an algorithm for FFT-based univariate polynomial multiplication which runs in $O(n\log(n)\log(\log(n)))$ operations in $\mathbb{K}$ [10]. For this problem, this is the best known algorithm for arbitrary $\mathbb{K}$.

Algorithm 1 shows an iterative version of the DFT we implemented. Due to known practical inefficiencies of the recursive version, we implemented the iterative version of the DFT which is discussed in detail in [13, 37]. It relies on the reordering (or reshuffling) of the input array of coefficients. A reshuffled 8-element array is shown in Figure 3.2. The *shuffle* function in Algorithm 1 implements the shuffling mechanism depicted by Figure 3.2.

Algorithm 1 DFT: Iterative Algorithm for the DFT.

INPUT: $n = 2^r$, $A = [a_0, a_1, ..., a_{n/2-1}, ..., a_{n-1}]$ an array for the coefficient of the polynomial, $\Omega = [1, \omega, ...\omega^{n/2-1}, ..., \omega^{n-1}]$ an array of powers of a primitive n-th root of unity ω
OUTPUT: An array $y = [y_0, ..., y_{n-1}]$ such that y_i is $A(\omega^i)$ by calling $DFT(n, A, \Omega)$.

```
DFT(n, A, Ω) ==
  A := shuffle(A, n);
  for s = 1..r repeat
    m := 2^s
    for k = 0..n − 1 by m repeat
      for j = 0..m/2 − 1 repeat
        t := Ω_{j m} A[k + j + m/2]
        u := A[k + j]
        A[k + j] := u + t
        A[k + j + m/2] := u − t
  return A
```

Algorithm 2 shows *iDFT* which computes inverse DFT while Algorithm 3 shows the FFT-based product of two polynomials that utilizes the *DFT* and *iDFT* algorithms

We implement the FFT over a generic coefficient ring on which we are able to find primitive roots of unity. We also implement a primitive root of unity finder that uses probabilistic methods to determine unit generators from which a primitive root of unity is extracted. The next section describes the primitive root of unity finder.

3.2 Primitive Root of Unity Finder

The FFT-based multiplication for a product of size n relies on the finding of primitive n-th roots of unity, where n is a power of 2. We explain how to do it in a finite

Algorithm 2 iDFT: Iterative Algorithm for the Inverse DFT.

INPUT: $n = 2^r$, $A = [a_0, a_1, ..., a_{n/2-1}, ..., a_{n-1}]$ an array of evaluation of $f(x)$ at points in Ω where $\Omega = [1, \omega, ...\omega^{n/2-1}, ..., \omega^{n-1}]$ an array of powers of a primitive n-th root of unity ω
OUTPUT: An array $y = [y_0, ..., y_{n-1}]$ of coefficients of $f(x)$ by calling $iDFT(n, A, \Omega)$.

$iDFT(n, A, \Omega)$ ==
  $A := DFT(n, A, \Omega)$;
  $ninv := inv(n)$
  $n2 := n/2$
  for $i = 1..n2 - 1$ **repeat**
    $j = n - 1$
    $tmp := A[i]$
    $A[i] := ninv\, A[j]$
    $A[j] := ninv\, tmp$
  $A[0] := ninv\, A[0]$
  $A[n2] := ninv\, A[n2]$
  return A

Algorithm 3 FFTMult: Algorithm For The Convolution of Two Polynomials.

INPUT: $A = [a_0, a_1, ..., a_{n/2-1}, ..., a_{n-1}]$ and $B = [b_0, b_1, ..., b_{n/2-1}, ..., b_{n-1}]$, arrays of coefficients of two polynomials $A(x)$ and $B(x)$
OUTPUT: An array $y = [y_0, ..., y_{n-1}]$ of coefficients of $A(x) \times B(x)$

$FFTMult(A, B)$ ==
  $A :=$DFT(n, A, Ω_b);
  $B :=$DFT(n, B, Ω_b)
  for $i = 1..n$ **repeat**
    $AB := A.i\, B.i$
  return $iDFT(n, AB, \Omega_{ab})$

Figure 3.2: DFT-Reordering for an 8-element Array.

field $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$. As a requirement, n must divide $p-1$ for the correct p [47, 35, Ex. 14.14, Ch. 14]. Then, there exists an integer q such that $p = qn + 1$ holds. According to Little Fermat's theorem, for all $\alpha \in \mathbb{K}$, with $\alpha \neq 0$ and $\alpha \neq 1$, we have

$$\alpha^{p-1} = 1 \tag{3.2}$$

Therefore, we obtain

$$\alpha^{qn} = 1 \tag{3.3}$$

Which follows that α^q is a candidate for being a primitive n-th root of unity. If α^q is not a primitive n-th root of unity then:

- the sequence of the α^{qk} for $k = 0, \ldots, n-1$ is periodic, with a period dividing n, and
- in particular $\alpha^{qn/2} = 1$ holds.

Since $\alpha^{qn/2}$ equals either 1 or -1, we have the following proposition:

Proposition 1 *Let $p > 2$ be a prime number and let n, q be integers such that n is a power of 2 $p = nq + 1$. Let $\alpha \in \mathbb{Z}/p\mathbb{Z}$ different from 0 and 1. Then, α^q is a primitive n-th root of unity if and only if $\alpha^{qn/2} = -1$ holds.*

Based on this proposition, we implement a probabilistic primitive root of unity finder. The ALDOR source code below shows such function.

```
  macro {
    MI == MachineInteger;
    K == SmallPrimeField;
  }
  primitiveRootOfUnity(n:MI):Partial(K) == {
      local  omega, alpha:K;
      local pm, q, n2:MI;
      not isPowerOfTwo?(n) => return failed;
      import from Integer;
      pm := machine(characteristic$K);
      -- pm-1 needs to be divisible by n.
      not zero?((pm-1) rem n) => return failed;
      q := (pm-1) quo n;
      n2 := n quo 2;
      repeat {
        alpha := random()$K;
        zero?(alpha) => iterate;
        one?(alpha) => iterate;
        omega := alpha^q;
        one?(omega^n2) => iterate;
        return [omega];
      }
}
```

3.3 Empirical Timings of FFT vs Karatsuba-based Multiplication

General purpose computer algebra systems typically implement the Karatsuba multiplication which has a complexity of $O(n^{1.59})$ ring operations in R, for polynomials of degrees at most n, surpassing the classical multiplication which has a complexity of $O(n^2)$. The FFT with complexity $O(n \log(n))$ is the backbone of the fastest algorithms; this is why our implementation of it in ALDOR is a significant boost to ALDOR and to the other algorithms that rely on it. The well-known $\lim_{n\to\infty} \frac{nlog(n)}{n^\alpha} \to 0$, for $\alpha > 0$ implies that the FFT-based multiplication is asymptotically faster.

Karatsuba Algorithm: a brief recall of the Karatsuba algorithm may be necessary here. Let R be a ring and $a, b \in R[x]$, each of degree at most $n = 2^k, k \in \mathbb{N}$. Karatsuba's method splits a and b into two polynomials (each of size $n/2$):

$$a = a_2\, x^{n/2} + a_1 \tag{3.4}$$

$$b = b_2\, x^{n/2} + b_1 \tag{3.5}$$

where $a_1, a_2, b_1, b_2 \in R[x]$. Then the product ab can be written as:

$$ab = a_2\, b_2\, x^n + (a_1\, b_2 + a_2\, b_1)\, x^{n/2} + a_1\, b_1 = a_2\, b_2\, x^n + ((a_1 + a_2)(b_1 + b_2) - a_1\, b_1 - a_2\, b_2)\, x^{n/2} + a_1\, b_1 \tag{3.6}$$

The Karatsuba algorithm uses the Relation 3.6 to recursively compute $a_1\, b_1$, $a_2\, b_2$ and $(a_1 + a_2)(b_1 + b_2)$. It can be shown that computing the product $a\, b$ using this method requires three multiplications and two additions on polynomials of size $n/2$, two subtractions on polynomials of size $n - 1$ and one addition of size $n - 2$. Further analysis results in the earlier stated complexity estimate of $O(n^{1.59})$. FFT's estimate of $O(n\ \log(n))$ is an improvement over this.

Figure 1.1 shows the result of the comparisons of our ALDOR implementation of the FFT-based multiplication and ALDOR's Karatsuba-based.

3.4 Generic ALDOR Implementation vs Non-generic Optimized C++(NTL) and MAGMA Implementations

NTL and MAGMA computer algebra systems have "state of the art" implementations of fast algorithms and probably the fastest available. Hence, we compare our implementation with theirs. To compare with the non-generic but highly optimized polynomial implementation in NTL [44], we used the function `FFTMul` in the NTL classes `zz_p` and `ZZ_p`, respectively for small and big primes. For the small prime case (27-bit prime), Figure 3.4 gives the timings which show that NTL's outperforms ours by a factor of 2. Considering that we achieved our goal of an efficient implementation in a high-level programming environment without resort to schemes to exploit hardware resources (like cache-friendliness), which is the case in the NTL version, we believe that this factor is a good result. We say it is good because a similar comparison of ALDOR's specialized (for polynomials with coefficients in $\mathbb{Z}/_p\mathbb{Z}$) gcd implementation in function $gcd(a, b)$ and the generic version in function $euclid(a, b)$ shows a similar factor of 2 as Figure 3.3 shows. For MAGMA, we used the general multiplication function "*" over GF(p), the prime field of order p. Figure 3.4, for the 27-bit prime case, shows that our implementation is comparable to MAGMA's, though slightly slower.

Similar results were observed for the big prime case (64-bit prime) as Figure 3.4 shows for ALDOR, NTL and MAGMA.

3.5 Generic ALDOR Implementation vs its Translation in NTL

To explore further how the language affects the performance of the implementation, we have experimented with translation of our FFT using NTL's `ZZ_p` class for the coefficient field ($\mathbb{Z}/_p\mathbb{Z}$) and compared its performance with the ALDOR implementation. Figure 3.7 shows that the ALDOR implementation running at highest ALDOR optimization level outperforms the translation on NTL platform. This shows that

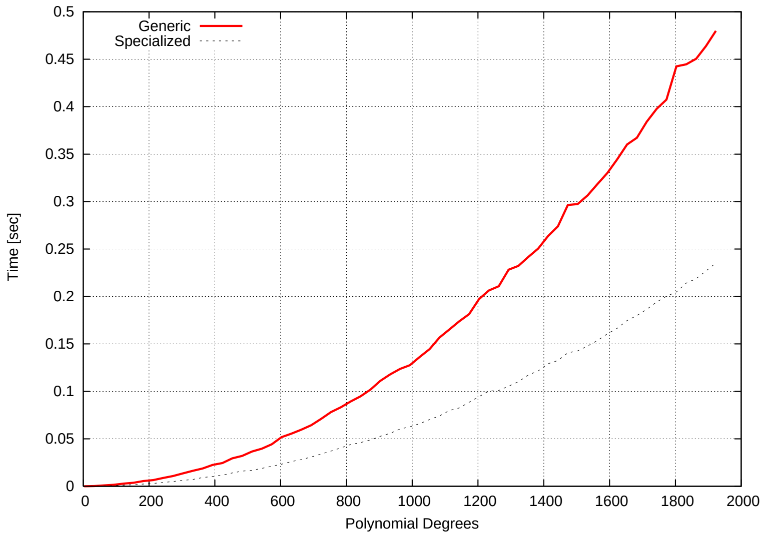

Figure 3.3: Generic GCD vs Specialized GCD in ALDOR Modulo a 27-bit Prime.

the ALDOR optimizer produces code that is comparable to hand-optimized C code.

3.6 Implementation Techniques and ALDOR Source Code of the FFT

Aside resorting to the iterative version rather than the recursive, we were careful with the handling of the storage. All the functions are written to work "in-place" such that enough storage is allocated ahead and values which are overwritten are those that are no longer required.

The signature of the ALDOR source code showing the functions that form the FFT multiplication package is shown below while the details of the fuctions are shown in the Appendix A.

```
macro {
    MI == MachineInteger;
    PAK == PrimitiveArray K;
    PRUF == PrimitiveRootOfUnityFinder;
    Z == Integer;
}
FFTPolMultiplicationPackage(K:Field): with {
    OmegaPowers: (PAK,K,MI) -> PAK;
```

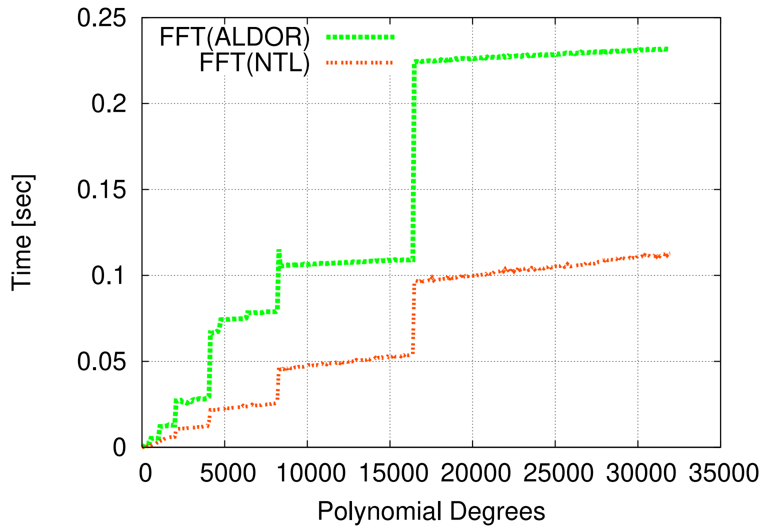

Figure 3.4: FFT: Our ALDOR Implementation vs. C++(NTL) Optimized Implementation Modulo a 27-bit Prime.

```
    ++ omegaPowers(arr,w,n) returns in arr the n first powers of
    ++a primitive nth root of unity, w
    dft!: (PAK,MI,PAK) -> PAK;
    ++ dft(pol,s,omega__powers) will return the FFT of pol at omega.
    fftProduct!: (PAK,MI,PAK,MI,MI,PAK) -> PAK;
    ++ fftProduct(pol1,dp,pol2,dq,s,wp).
    ++ returns the fft prod. Given pol1, pol2 and s,dp=size of pol1
    ++ and dq=size of pol2,  such that a w exists that
    ++ is 2^sth root of unity and wp which is omega-powers
    fftSize: (Z, Z) -> Z;
    ++ fftSize returns s given 2 polynomials such that
    ++ n=2^s >= deg(pol1)+deg(pol2)+2
    fftMult!: (PAK,PAK,PAK,MI) -> PAK;
    ++ fftMult(pol1,pol2,workspace,sz)
    ++ where sz is fftsize
    shuffler! : (PAK,PAK,MI) -> PAK;
    ++ shuffler!(pol1,wspace,s) shuffles pol1 using workspace wspace and
    ++ fftsize s
    fftTimesArr: (PAK,MI,PAK,MI,MI) -> Partial(PAK);
} == add {
```

... see Appendix A for details

```
}
```

The package *FFTPolMultiplicationPackage* accepts a generic field of the polynomial coefficients. Polynomials in the package are representated by an array of

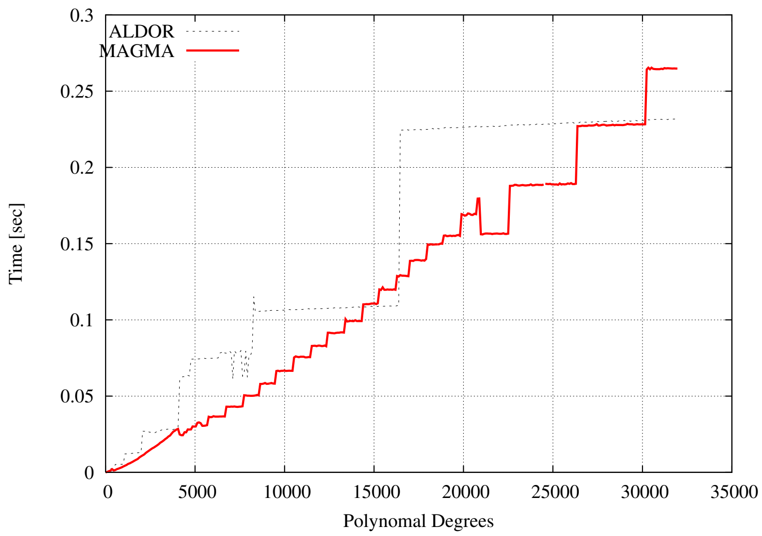

Figure 3.5: FFT: Our ALDOR Implementation vs. MAGMA implementation Modulo a 27-bit Prime.

coefficients. This package can be integrated into domain "SUP(K)" (Sparse univariate polynomial domain over field K) or "DUP(K)" (dense univariate polynomial domain over field K). An example code that integrates *FFTPolMultiplicationPackage* into the DUP(K) domain of ALDOR's `libalgebra` library follows.

```
fftTimes(p:%, q:%):% == {
```

import from FFTPolMultiplicationPackage(RFR);

```
  import from Partial(RFR);
  import from Partial(PrimitiveArray(R pretend Field));
  assert(~zero? p); assert(~zero? q);
  assert(~one? p); assert(~one? q);
  dp := degr p;
  dq := degr q;
  cp := coeffs p;
  cq := coeffs q;
```

ffs:I := fftSize(next dp,next dq);
result := fftTimesArr(cp, next dp, cq, next dq,ffs);

```
  failed?(result) => {
    nokara? => naiveProduct(p, q);
    karatsuba(p, q, karaCutoff);
  }
  d:I := dp + dq;
```

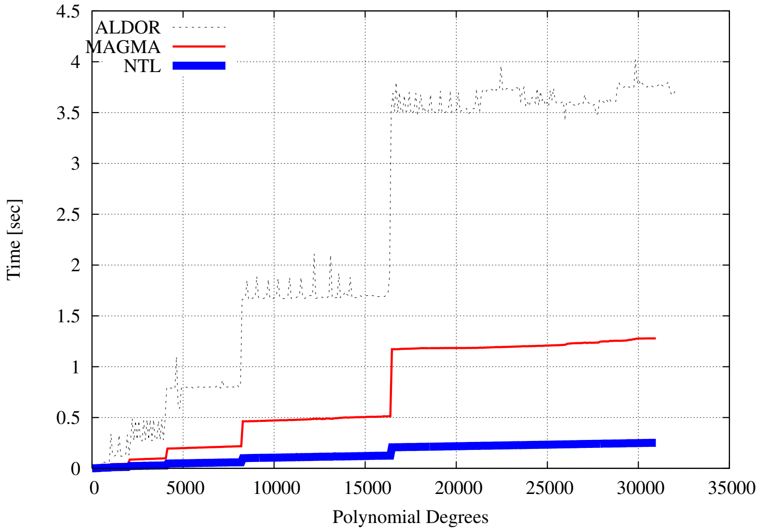

Figure 3.6: FFT: Our Aldor implementation vs. NTL's vs Magma's Modulo a 64-bit Prime.

```
  intdom? => poly(next d, d, retract(result));
  (dg := computeDegree(d, retract(result))) < 0 => 0;
  poly(next d, dg, retract(result));
}
```

The lines in boldface are functions from our package.

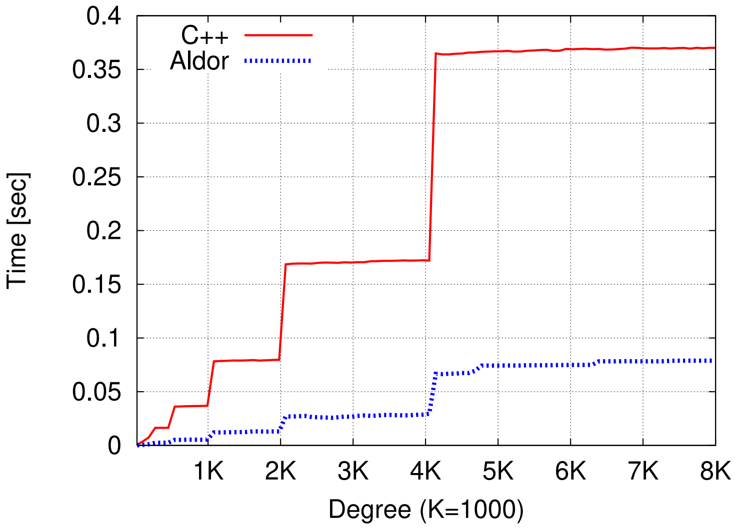

Figure 3.7: FFT: ALDOR Implementation .vs. NTL Translation.

Chapter 4

Power Series Inversion Using the Newton Iteration

Division with remainder is the basic operation for algorithms in euclidean domains such as the *Extended Euclidean Algorithm* (EEA) and the *Chinese Remaindering Algorithm (CRA)*.

Asymptotically fast EEA and CRA are obtained from an asymptotically fast algorithm for the division with remainder, which relies on fast polynomial multiplication via power series inversion. This latter operation is the second one that we view as efficiency-critical. In this chapter, we present the principle of the Newton power series inversion algorithm in the first section. In the second section, we describe a technique, the *Middle Product* technique, to achieve the inversion efficiently (that is, in fewer ring operations). The third section describes our implementation techniques. Also, empirical results that show the savings in memory and timings of our implementation of the middle product technique versus the implementation of the naive version are also presented in this section. The fourth section describes an application of the Newton inversion algorithm to achieve fast *polynomial division with remainder*. It also presents empirical results of the timings of the fast division versus the naive division algorithm.

As the results in the other chapters show, the work done in implementing the Newton power series inversion algorithm efficiently, pays off in its application to other algorithms.

4.1 Principle

Power series inversion using the Newton iteration method provides a fast method of computing multiplicative inverses. Given a commutative ring R with a 1 and $\ell \in \mathbb{N}$, it computes the inverse of the polynomial $f \in R[x]\, s.t\, f(0) = 1$ and $\deg f < \ell$, modulo x^{ℓ}.

Power series inversion also leads to:

- the logarithm and exponential of power series [8];
- fast algorithms for recovering polynomials from their Newton sums - Schönhage uses it in [42] for isolating the complex roots of univariate polynomials;
- fast algorithms for the power series development of solutions of Ordinary Differential Equations (ODE) [8].

4.2 What is Newton Iteration?

The Newton iteration is used in numerical analysis to compute successive approximations to solutions of $\phi(g) = 0$. From a suitable initial approximation g_0, subsequent approximations are computed using:

$$g_{i+1} = g_i - \frac{\phi(g_i)}{\phi'(g_i)} \tag{4.1}$$

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

$$g_{i+1} = g_i - \frac{1/g_i - f}{-1/g_i^2} = 2\, g_i - f\, g_i^2 \tag{4.2}$$

.

Proposition 2 that follows tells us a good initial approximation and shows that this method converges quickly to a solution, also in algebraic setting.

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

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

$$f\,g_0 \equiv f(0)g_0 \equiv 1.1 \equiv 1 \bmod x^{2^0} \tag{4.3}$$

For the induction step, we find

$$1 - f\,g_{i+1} \equiv 1 - f\,(2\,g_i - f\,g_i^2) \equiv 1 - 2\,f\,g_i + f^2\,g^2 \equiv (1 - f\,g_i)^2 \equiv 0 \bmod x^{2^{i+1}} \tag{4.4}$$

$\triangleleft$

Based on the foregoing, we obtain the following algorithm (Algorithm 4) to compute the inverse of $f \bmod x^\ell$.

Algorithm 4 Power Series Inversion of f to Precision ℓ

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

Output: $g \in R[x]$ such that $f\,g \equiv 1 \bmod x^\ell$. Runs in $3\mathsf{M}(\ell) + 0(\ell)$ operations in R. Recall from Definition 1 in Chapter 2 that M is multiplication time whose value is dependent on the multiplication algorithm used.

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

Proposition 3 *If ℓ is a power of 2, then Algorithm 4 uses at most $3\mathsf{M}(\ell) + O(\ell) \in O(\mathsf{M}(l))$ arithmetic operations in R [47, Ch. 9].*

Proof $\triangleright$ The correctness stems from Proposition 2 which concludes that

$$f\,g_i \equiv 1 \bmod x^{2^i} \tag{4.5}$$

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

$$g_i \equiv g_{i-1}\,(2 - f\,g_{i-1}) \equiv g_{i-1} \bmod x^{2^{i-1}} \tag{4.6}$$

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

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

$$\sum_{1 \leq i \leq r} \frac{3}{2}\,\mathsf{M}(2^i) + 2^{i-1} \leq (\frac{3}{2}\,\mathsf{M}(2^r) + 2^{r-1}) \sum_{1 \leq i \leq r} 2^{i-r} < 3\,\mathsf{M}(2^r) + 2^r = 3\,\mathsf{M}(\ell) + \ell \tag{4.7}$$

since $2\mathsf{M}(n) \leq \mathsf{M}(2n)$ for all $n \in \mathbb{N}$ (see Definition 1 in Section 2.1) $\lhd$

4.3 The Middle Product Technique

The middle product algorithm [25] is one of the best algorithms available to compute the inversion, $1/f$, where f is a polynomial or a series, using Newton's iteration. In the implementation of the FFT multiplication for the prime field case, this idea is used in the NTL library [43].

The middle product technique improves over the cost, $3\,\mathsf{M}(\ell) + O(\ell)$ in Section 4.1. It is possible to achieve a cost that is lower than $2\,\mathsf{M}(\ell) + O(\ell)$ ring operations [25, 43].

We now explain the principle behind the technique followed by the algorithm and an analysis. We assume that the FFT-based multiplication (see Chapter 3) is used in all cases.

The following relations follow from Algorithm 4:

$$g_i = \frac{1}{f} \bmod x^{2^i} \tag{4.8}$$

$$g_{i+1} = g_i\,(2 - g_i f) \bmod x^{2^{i+1}} \tag{4.9}$$

From Equations 4.8 and 4.9, we have:

$$f\,g_i = 1 + T\,x^{2^i} + S\,x^{2^i+1} \bmod x^{2^{i+1}}. \tag{4.10}$$

with the degrees of $f\,g_i$ at most 3.2^i and $\deg T$, $\deg S\ < 2^i$. We need to compute the T in the middle product of Equation 4.10 efficiently. Seeing that $f\,g_i \bmod x^{2^{i+1}} - 1$ yields $(1 + S) + T\,x^{2^i}$, to compute T, it is sufficient to compute $f\,g_i \bmod x^{2^{i+1}} - 1$. That is, an FFT product at a primitive root of order 2^{i+1} will do it. This reduces the cost of this middle product by a factor of 2, which in turn reduces the cost of the entire Newton iteration by a factor of about $2/3$.

Figure 4.1 is a graphical illustration of this technique.

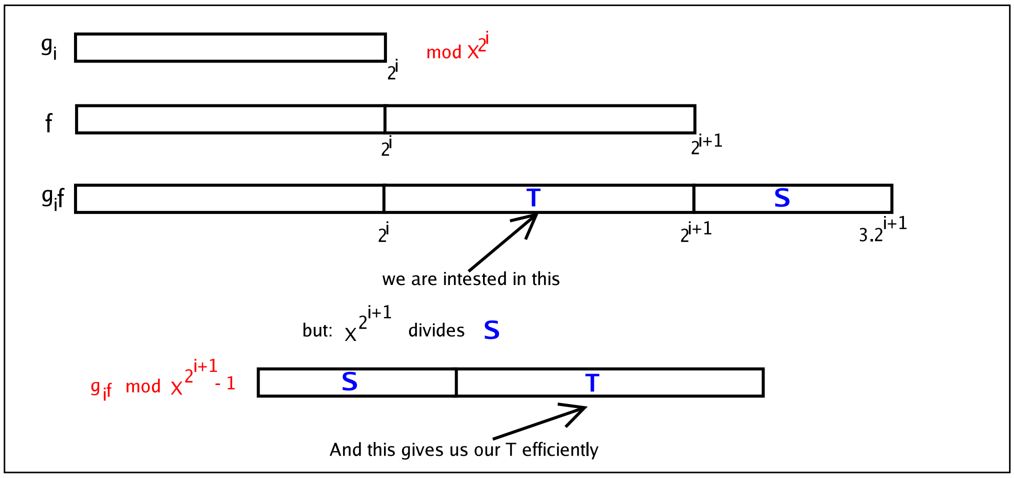

Figure 4.1: Power Series Inversion: The Middle Product Technique to Compute T Efficiently.

It is necessary to restate that the savings in operations is from the knowledge that a DFT and an iDFT on an input of size $2n - 1 \times n$ will require 4.2^i or 2^{i+2} points using the naive method. But with the middle product, we achieve $AB \to AB \bmod x^{2^i} - 1$ on 2^{i+1} points.

Further saving in operation is also achieved from reusing the DFT of g_i of Equation 4.8 at Equation 4.9

Algorithm 5 shows a pseudocode of our implementation of the middle product.

Algorithm 5 Middle Product Technique for the Newton Iteration in Power Series Inversion of f to Precision ℓ.

INPUT: Array $F = [F_1, \ldots, F_{\ell-1}]$ containing coefficients of $f \in R[x]$, with $f(0) = 1$, $\ell \in \mathbb{N}$, such that $\deg(f) < \ell$ and $R[x]$ a ring of power series.

OUTPUT: Array $G = [G_1, \ldots, G_n]$ containing coefficients of $g \in R[x]$ such that $f\,g \equiv 1 \bmod x^{\ell}$.

$MPInv(F, \ell)$ ==

1 $G_1 := 1$

2 $r := \lceil \log_2(\ell) \rceil$

3 **for** $i = 1 \ldots r$ **repeat**

comment: step i computes $1/F \bmod x^{2^i}$

3.1 $\omega :=$ primitiveRootOfUnity(2^i);

3.2 $\Omega := [\omega^j \, : \, j \in [1 \ldots 2^i]]$;

3.3 $DFTG :=$ DFT$(2^i, G, \Omega)$;

3.4 $DFTF :=$ DFT$(2^i, F \bmod x^{2^i}, \Omega)$;

3.5 $V := [DFTG[j] * DFTF[j] \, : \, j \in [1 \ldots 2^i]]$;

3.6 $iV :=$ iDFT$(2^i, V, \Omega)$;

3.7 $W := [iV[j] \, : j \in [2^{i-1}..2^i]$;

3.8 $DFTW :=$ DFT$(2^i, W, \Omega)$;

3.9 $X := [DFTG[j]\, DFTW[j] : j \in [1 \ldots 2^i]]$;

3.10 $iX :=$ iDFT$(2^i, X, \Omega)$;

3.11 $Y := [iX[j] \, : \, j \in [1 \ldots 2^{i-1}]$;

3.12 $G := [G[j] - \text{leftShift}(Y, 2^{i-1}) \, : \, j \in [1 \ldots 2^i]]$;

4 **return** G

The DFT, $iDFT$ and $primitiveRootOfUnity$ functions used in the algorithm (Algorithm 5) are specified in Chapter 3 of this report.

Analysis: Here we continue to use the concept of $\mathsf{M}(n)$ for the full FFT multiplication. But, we say that it costs $M(n)$ operations to compute both the DFT and iDFT of $2n - 1 \times n$ degree polynomials. The cost of one iteration of step 3 is $\mathsf{M}(2^{i-1})$ for the computation of iV in line 3.6 , $\mathsf{M}(2^{i-1})$ for the computation of iX in line 3.10 , 2^{i-1} operations in line 3.11 , and 2^i operations for the subtraction in line 3.12. The $leftShift$ in line 3.12 is free.

Thus, we have $2\,\mathsf{M}(2^{i-1}) + 2^{i-1} + 2^i$. And total running time is:

$$\sum_{1\le i\le r} 2\,\mathsf{M}(2^{i-1}) + \frac{3}{2}\,2^i) \le (2\,\mathsf{M}(2^{r-1}) + \frac{3}{2}\,2^r) \sum_{1\le i\le r} 2^{i-r} < 2\,\mathsf{M}(2^r) + 3\,2^r = 2\,\mathsf{M}(\ell) + 3\,\ell \tag{4.11}$$

Here, we have used the relation $2\,\mathsf{M}(2^{i-1}) \le \mathsf{M}(2^i)$. It is note-worthy to point out the DFT saving in line 3.9.

Our ALDOR source code for the implementation of the power series inversion using the technique just described is shown in Appendix B.

4.4 Implementation Techniques and Empirical Results

We implemented two versions of the power series inversion algorithm given in Algorithm 4. The "naive" version implemented the Algorithm 4 as is, that is, without change of data representation and without any further improvement. Then, we implemented the middle product version (Algorithm 5 in a space-efficient manner (that is, Garbage Collector-friendly) and with a better practical efficiency, using the following ideas:

- we work with arrays of coefficients until result is obtained.
- even though we accept a generic polynomial type, we change the data representation to arrays of coefficients, work only with these arrays,
- we pre-determine all array sizes and pre-allocate all needed buffers, so that there is no memory allocation in the loop.
- we use wrapped convolution to compute the n middle coefficients of a $(2n-1)\times n$ full product (this is the middle-product operation of [25, 43] and explained in section 4.3).

Figure 4.2 shows the graphs of the runtimes of our two implementations, together with the time for a single polynomial multiplication over a prime field of the form

$\mathbb{Z}/_p\mathbb{Z}$. The results validate the proof and analysis results in equations 4.7 and 4.11 that the middle product technique improves on the Newton iteration.

To compare the relative space-efficiencies, we tested the two implementations on different degrees up to 8000, and measured the maximum resident set size (RSS) - that is, the maximum non-swapped physical memory that a task has used. Figure 4.3 shows that the naive version used a total of over 16000 kilobytes of memory to invert a polynomial of degree 8000 while the middle product version used less than 2500 kilobytes for the same polynomial (achieving a factor of 6). For higher degrees, the factor is even better.

For comparison's sake, we give the ALDOR source code of the naive version in Appendix C.

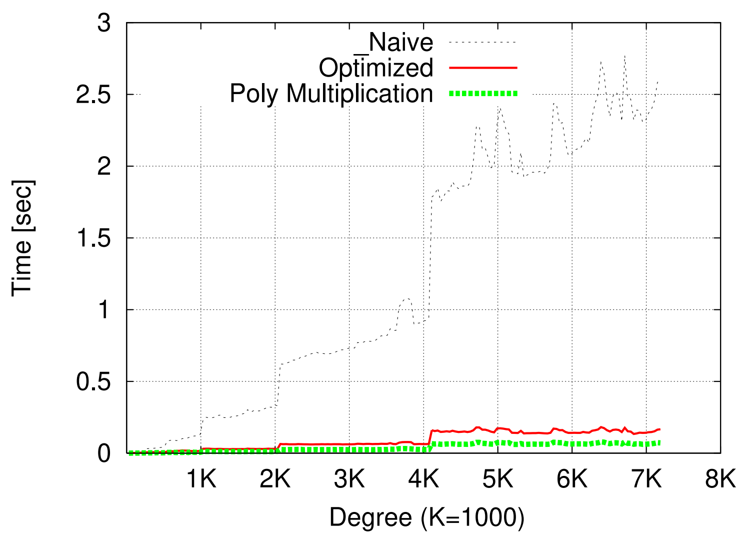

Figure 4.2: Power Series Inversion: Naive vs. Optimized Implementation vs. Multiplication Modulo a 27-bit Prime.

4.5 Fast Monic Division with Remainder

Given two polynomials a and b, both $\in R[x]$ and b monic, where R is a ring (commutative with 1); assuming they of are degrees m and n, with $m \leq n$, we can compute polynomials q and r in $R[x]$ satisfying $a = q\,b + r$ and degree of r is less than degree b. Using standard techniques (see [1, ch. 8]), this takes $O(n^2)$ operations

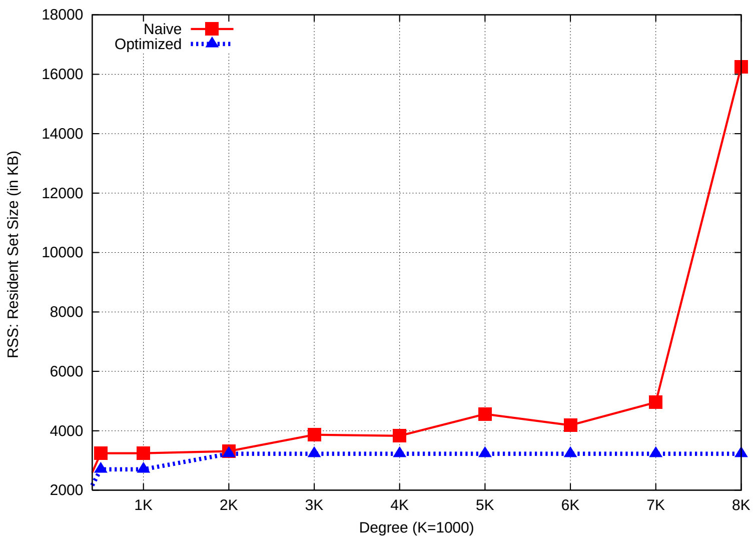

Figure 4.3: Power Series Inversion: Space Usage of Naive vs. Optimized Implementations Modulo a 27-bit Prime.

in R. Equipped with a fast power series inversion, it can be improved to $O(\mathsf{M}(n))$ operations in R [47].

We define A and B by the reversed form of a and b:

$$A(x) = x^{m+n}\, a(1/x) \tag{4.12}$$

$$B(x) = x^{m}\, b(1/x) \tag{4.13}$$

With the inverse $C \equiv 1/B(x) \bmod x^{n+1}$, we obtain q in its reversed form Q by subsequent multiplication:

$$Q(x) \equiv A(x)\, C(x) \bmod x^{n+1} \tag{4.14}$$

The full algorithm is shown in Algorithm 6.

In Algorthm 6, rev_a is the reverse representation of polynomial a. [47, Ch. 9] has a proof that shows Algorithm 6 can be applied to the case where a is of degree $m+n$ and the monic b is of degree n with $n \geq m \in \mathbb{N}$ at a cost of $4\mathsf{M}(m) + \mathsf{M}(n) + O(n)$ ring operations. However, the cost is $5\,\mathsf{M}(m) + \mathsf{M}(n) + O(n)$ if $m > n$. Figure 4.4 shows empirical timings of the fast versus classical divisions of two polynomials of degree n and $2n$ modulo a 27-bit Prime Field.

Algorithm 6 Fast Division with Remainder Algorithm

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

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

$FDiv(a, b)\ ==$

1 **if** $deg\, a < \deg\, b$ **then return** $q = 0$ and $r = a$
2 $m := \deg\ a - \deg\ b$;
3 $RBI = \mathrm{Inv}(rev_{\deg\ b}(b), m + 1)$
4 **return** $q = rev_m\,(q')$ and $r = a - b\,q$

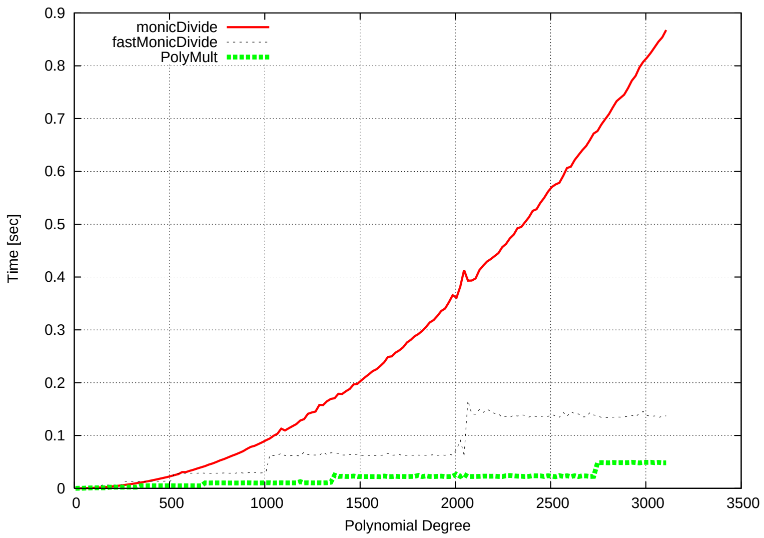

Figure 4.4: Fast Division vs Classical Division vs Multiplication Modulo a 27-bit Prime.

The source code of our implementation can be found in Appendix F.

Chapter 5

Subproduct Tree Techniques and Applications

In this chapter we describe our design, implementation and application of a subproduct binary tree stored as a single-dimensional array. Subproduct trees are used for implementing fast algorithms based on the divide-and-conquer strategy, such as simultaneous remainders, fast *Chinese Remaindering Algorithm* (CRA) and multipoint evaluation. The first section of this chapter defines the subproduct tree and how we implemented it, the second section discusses its application to realizing remainders simultaneously while the third section introduces our implementation of the fast CRA algorithms as described in the "Modern Computer Algebra" book [47, Ch. 10], using the subproduct tree.

5.1 The Subproduct Tree and its Implementation

Definition 2 *Let $a_0, \ldots, a_{r-1}$ be polynomials in $R[x]$ where R is a ring (commutative with 1). The subproduct tree $T(a_1, \ldots, a_r)$ is the tree defined recursively as follows:*

1. *if $r = 1$ then it consists of a_1, the single node;*
2. *if $r > 1$ then let q be the largest power of 2 which is less than r, then $T(a_1, \ldots, a_r)$ is a binary tree with $T(a_1, \ldots, a_q)$ as left subtree and $T(a_{q+1}, \ldots, a_r)$ as right subtree and with root labelled by the product $a_1 \cdots a_r$.*

Our design of the subproduct tree utilizes single-dimensional arrays and efficient supporting algorithms to traverse the array.

Figure 5.1 is an example of a subproduct tree with 10 points using our design while Figure 5.2 shows the complexity comparison between subproduct tree creation using our techniques and multiplication of two polynomials of degree d.

5.2 Application: Simultaneous Reduction

The subproduct tree is applied to compute fast simultaneous reduction of a family of polynomials with a given polynomial.

The algorithm for this takes the form below, for a $p \in \mathbb{K}[x]$ and family $[a_1, \ldots, a_e]$ of univariate polynomials over a field $\mathbb{K}$:

1. Build a subproduct tree $A(a_1, \ldots, a_e)$
2. Reduce p from the root to the leaves

Algorithms 10.14 and 10.16 in Chapter ten of [47] gives more detailed specification and proof of the principle behind this algorithm.

Complexity estimates

Naive approach: In the naive approach, the complexity estimate to compute the simultaneous remainders for four elements is as follows:

Knowing that $\deg a = 4\,d$ since $\deg a_i = \deg a_2 = \deg a_3 = \deg a_4 = d$,

it takes

$$2\,D(3d, d) \leq 2 \times 5\,\mathsf{M}(3\,d) + O(d) \leq 30\,\mathsf{M}(d) + O(d) \tag{5.1}$$

where $D(a, b)$ denotes an upper bound for the number of operations in R for dividing a polynomial of degree $a + b$ by a monic polynomial of degree b.

Subproduct Tree approach: on the other hand, with the subproduct tree approach, the complexity estimate is:

$$M(d)+D(2d, 2d)+2D(d, d) = M(d)+[4M(2d)+M(2d)]+2[5M(d)] \leq 1+8+2+10 = 21M(d) \tag{5.2}$$

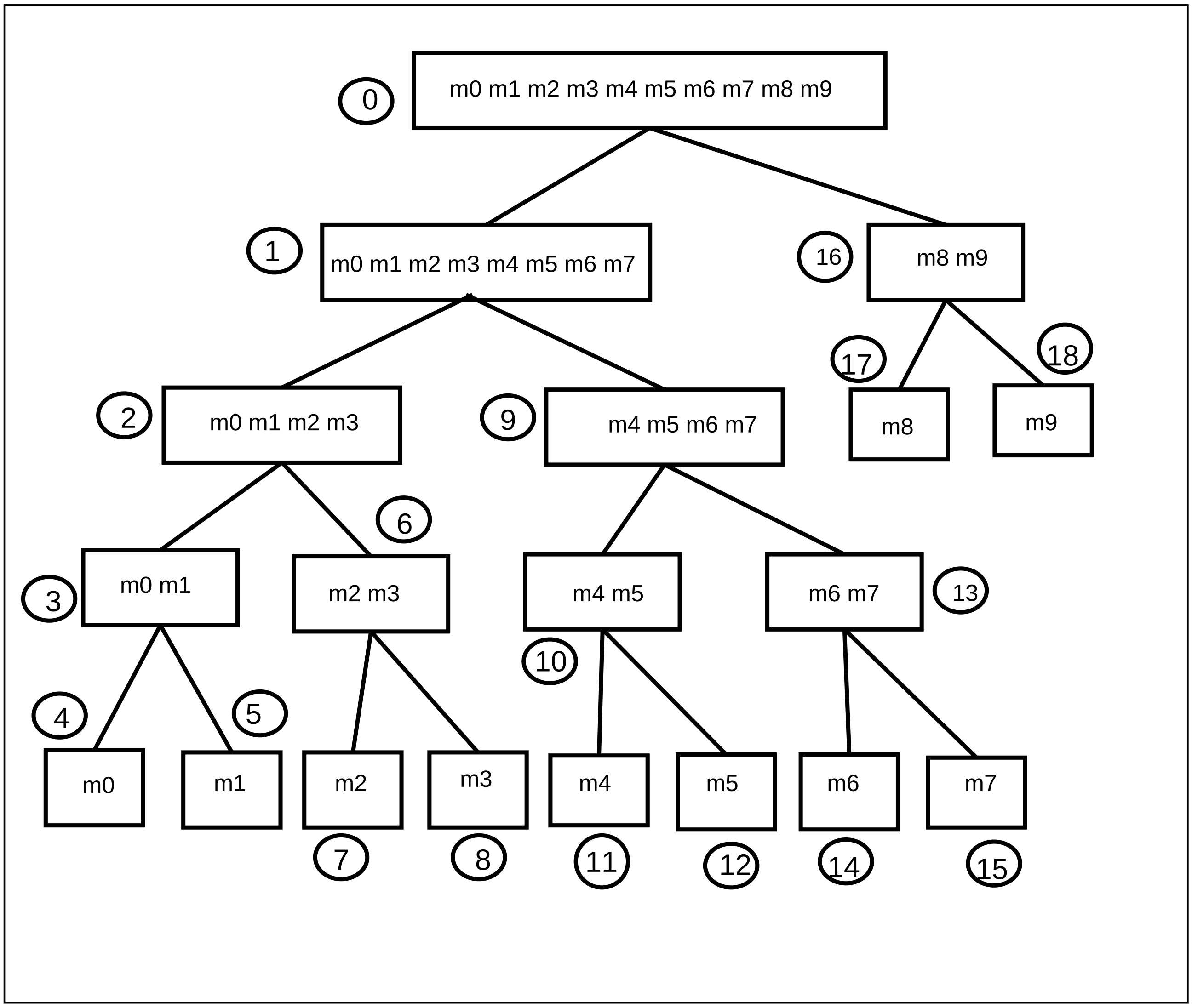

Figure 5.1: Subproduct Tree For 10 Points.

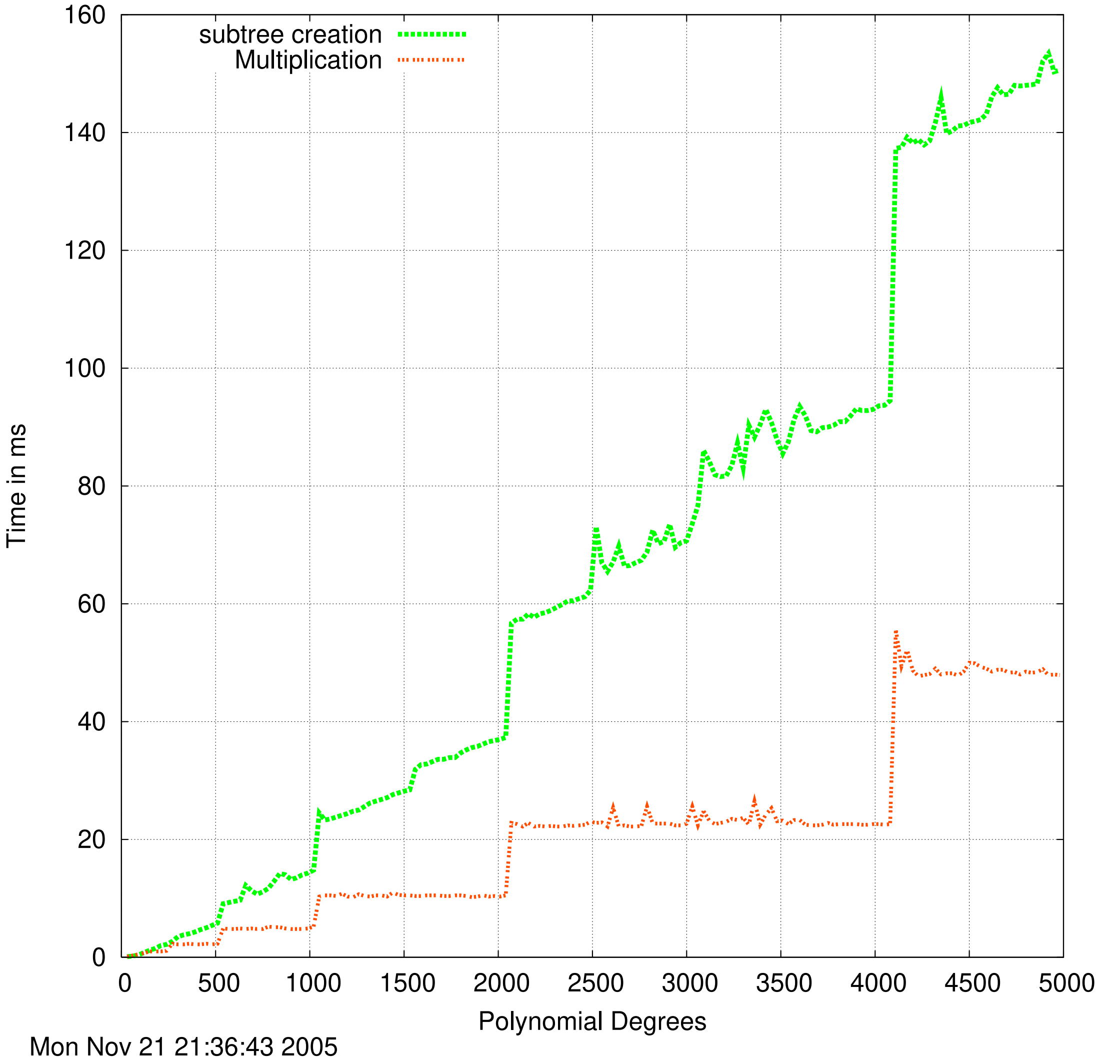

Figure 5.2: Subproduct Tree creation ($2d$ points) versus Multiplication(d*d) Modulo a 27-bit Prime.

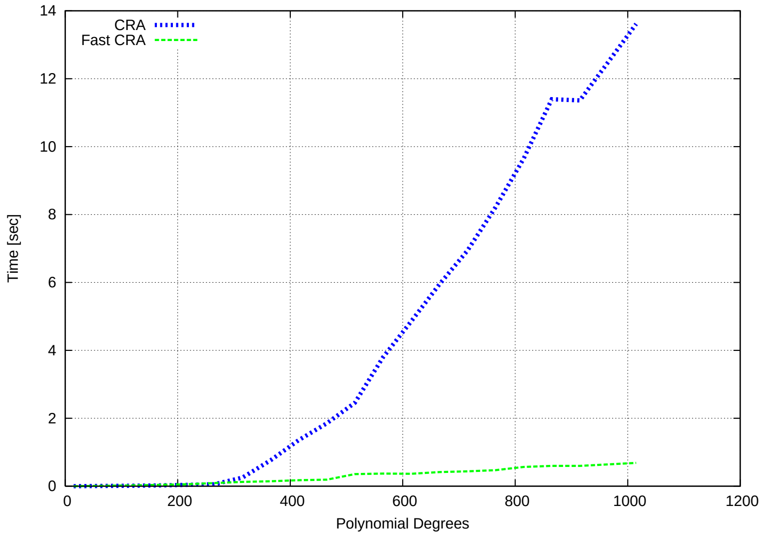

Figure 5.3: CRA .vs. Fast CRA Modulo a 27-bit Prime.

5.3 Application: Fast Chinese Remaindering Algorithm

The CRA is a well known algorithm which has many applications (modular methods, interpolation, etc) [47]. We consider the case of univariate polynomials over a field, $\mathbb{K}$. In this context, the specification of the CRA is as follows

CRA Principle

INPUT: $m_0, ..., m_{r-1} \in \mathbb{K}[x]$ such that m_i and m_j are relatively prime, $i \neq j$, where $\mathbb{K}$ is a Field, $r = 2^k$ for some $k \in \mathbb{N}$, and $v_0, ..., v_{r-1} \in R[x]$ with deg $v_i <$ deg m_i for all i.

OUTPUT: The unique polynomial $f \in \mathbb{K}[x]$ of degree less than $n = \sum_{0 \leq i < r} deg\ m_i$ such that $\boxed{f \equiv v_i \bmod \ m_i}$ for $0 \leq i < n$ in $O(M(n) log(n))$ operations in $\mathbb{K}$

The classical CRA Algorithm 5.4 in Chapter five of [47] runs in $O(n^2)$ operations in $\mathbb{K}$. The *Fast Chinese Remaining Algorithm* (FCRA) has the same specification but runs in $O(M(n) \log(n)$ operations in $\mathbb{K}$. It is based on the subproduct tree techniques. See Chapter ten in [47] for more details of the FCRA.

Among other uses, the FCRA has application in helping to divide big rings to smaller rings which in turn aid in speeding up algorithms. It also aids in the implementation of new algorithms. Our work on it is a direct implementation of the algorithms specified in [47, Ch.10]; with an only contribution of converting the subproduct tree from a 2-dimensional array of input elements that are of size, a power of two, to, a 1-dimensional array that supports input elements of arbitrary size. The subproduct tree helps to optimize the computation of simultaneous remainders during the CRT computation. Furthermore, we contribute to replace the "interpolate" function in the ALDOR library with an asymptotically faster Chinese Remainder package. Our work on this algorithm also buttresses the fact that efforts to really improve efficiency-crucial operations like multiplication and modular inversion enhances the performance of other algorithms like this that utilize them.

Figure 5.3 shows the empirical timings of our CRT implementation and the 'interpolate' function of ALDOR's `libalgebra`. The signature of our ALDOR source codes that implements the algorithms in [47, Ch. 10.3] is given below while the detailed codes are given in Appendix D.

```
macro {
    MI == MachineInteger;
    AU == Array U;
    FEEA == FastExtendedEuclideanAlgorithmPackage(K,U);
    FFTMP == FFTMiultiplicationPackage(K,U);
}

FCRMPackage(K:Field,U:UnivariatePolynomialAlgebra(K)): with {
    subProductTree:  (AU,MI) -> AU;
    ++  subProductTree(m,r) builds the subProduct tree and will
    ++  return the one-dimensional array with the tree.

    fSimulReduction!: (AU,MI,MI,U,AU,MI,MI) -> Generator U;
```

```
    ++ Fast Simultaneous Reduction.
    ++ fSimulReduction!: (mr,i,j,f,a,b,e)
    ++ input: mr is workspace and will return the final reduced array
    ++ i,j are the begin and end indices of mr, a is subproduct tree,
    ++ f in R[x], b and e are begin and end indices of a
    ++ output: f rem m0,...,f rem m(r-1) where mi is from M.

    linearCombine: (AU,AU,MI,AU,MI) -> U;
    ++ Linear Combination
    ++ linearCombine(m,i,j,c,ic,jc,a,b,e)
    ++ input: m in R[x] of any length,i,j for input begin/end indeces
    ++ c in R[x] of same length, ic, jc for cu's indices,a for
    ++ workspace, b and e for workspace begin/end indices
    ++ output: f = sum(cu.i*m/mu.i) where m := mu.0...mu.(r-1)

    fCRA: (AU,MI,AU) -> U;
    ++  fChRem(m,k,v) (Fast Chinese Remaindering Algorithm - 10.22) returns a
    ++ polynomial f = vi mod mi of deg less than n=sum(deg mi). Alg 10.22 in
    ++ MCA (Modern Computer Algebra Book by J. Gathen and J. Gerhard)
} == add {
```

see Appendix D for details of the fuctions in this package.

```
}
```

Chapter 6

Half-GCD over a Field

The Half-GCD is a fundamental notion to obtain the *Fast Extended Euclidean Algorithm* (FEEA). More precisely, for input polynomials a, b of degree $\leq d \in \mathbb{K}[x]$, where $\mathbb{K}$ is a field, it computes their GCD in at most $33M(d)\, \log(d) + O(d\, log(d))$ operations in $\mathbb{K}$.

The Half-GCD originated in the ideas of Lehmer, Knuth and Schönhage. Though, it has appeared in many books including [1, 51, 47] and in many other publications, it is frequently specified incorrectly. This creates an implementation challenge. We adapted Yap's [51] version of the algorithm in our implementation. In this chapter, we describe the standard Extended Euclidean Algorithm in the first section. In the second section, we give the Fast Extended Euclidean Algorithm which uses the Half-GCD technique. We also report on our ALDOR implementation of the FEEA and compare it with the standard algorithm and other implementations in NTL and MAGMA.

6.1 Standard Extended Euclidean Algorithm

Definition 3 *An integral domain R together with a function $d : R \to \mathbb{N} \cup \{-\infty\}$ is a* **Euclidean Domain** *if:*

1. *for all $a, b \in R^+$, $d(a\, b) > d(a)$,*

2. *for all $a, b \in R$ with $b \neq 0$, we can divide a by b to produce $q, r \in R$ such that $a = b\, q + r$ and $d(r) < d(b)$*

$q = a$ quo b and $r = a$ rem b are called the *quotient* and *remainder* respectively and the function d is called a *Euclidean Function* on R. For example, if R is a Field, then $R[x]$, the ring of univariate polynomials over R, is a Euclidean domain with $d(a) = \deg a$.

Definition 4 *Let R be a ring and a, b, $c \in R$. Then c is a* greatest common divisor *(GCD) of a and b if:*

1. *$c|a$ and $c|b$,*
2. *if $d|a$ and $d|b$, then $d|c$, for all $d \in R$*

Generally, the GCD of a and b need not be unique but all their GCDs need be associates. Algorithm 7 shows how the classical Euclidean algorithm (EA) computes the GCD of two elements in an arbitrary Euclidean domain. It can be deduced from the algorithm that the GCD is basically the last nonzero element of the remainder sequence.

Algorithm 7 The Classical Euclidean Algorithm.

INPUT: a, $b \in R$, where R is a Euclidean Domain

OUTPUT: GCD of a and b

```
gcd(a, b) ==
1     r_0 := a, r_1 := b
2     i = 1
3     while r_i ≠ 0 do
3.1      r_{i+1} := r_{i-1} rem r_i
3.2      i := i + 1
4     return r_{i-1}
```

The *Extended Euclidean Algorithm* (EEA) extends the classical algorithm to compute not only the GCD of a and b but also the elements s and t that satisfy the relation $s\,a + t\,b = g$ where g is the GCD of a and b. Algorithm 8 describes an EEA that produces monic remainders.

The algorithm terminates because the $d(r_i)$ are strictly decreasing nonnegative integers. The elements r_i, s_i, t_i are called the the ith row on the EEA, for $0 \leq$

Algorithm 8 The Extended Euclidean Algorithm.

INPUT: $f,\, g \in R$, where R is a Euclidean Domain

OUTPUT: $\ell \in \mathbb{N}, \rho_i, r_i, s_i, t_i \in R$ for $0 \leq i \leq \ell + 1$, and $q_i \in R$ for $1 \leq i \leq \ell$

$EEA(a, b)\ ==$

$\rho_0 := lc(f)$, $r_0 := f/rho_0$, $s_0 := 1/rho_0$, $t_0 := 0$
$\rho_1 := lc(g)$, $r_1 := g/rho_1$, $s_1 := 0$, $t_1 := 1/\rho_1$
$i = 1$
while $r_i \neq 0$ **do**
  $q_i := r_{i-1}$ quo r_i
  $\rho_{i+1} := lc(r_{i-1}$ rem $r_i)$
  $r_{i+1} :=(r_{i-1}$ rem $r_i)/\rho_{i+1}$
  $s_{i+1} := (s_{i-1} - q_i\, s_i)/\rho_{i+1}$
  $t_{i+1} := (t_{i-1} - q_i\, t_i)/\rho_{i+1}$
  $i := i + 1$
$\ell := i - 1$
return $\ell, \rho_i,\, r_i,\, s_i,\, t_i$ for $0 \leq i \leq \ell + 1,$ and q_i for $1 \leq i \leq \ell$

$i \leq \ell + 1$. ℓ is the *Euclidean length* of the pair (f, g). The central property is that $s_\ell\, f + t_\ell\, g = r_\ell\ = gcd(f, g)$; where the elements s_ℓ and t_ℓ are the **Bézout coefficients** of f and g. Though we are interested in the GCD and the Bézout coefficients, most of the other intermediate results computed by the algorithm have applications for various tasks in computer algebra.

To undestand the algorithm better, consider the matrices:

$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} \tag{6.1}$$

$$Q_i = \begin{pmatrix} 0 & 1 \\ 1/\rho_{i+1} & -q_i/\rho_{i+1} \end{pmatrix} \tag{6.2}$$

in $R[x]^{2\times2}$ and $R_i = Q_i \dots Q_1\, R_0$ for $1 \leq i \leq \ell$. From Algorithm 8 we have:

$$Q_i \begin{pmatrix} s_{i-1} & t_{i-1} \\ s_i & t_i \end{pmatrix} = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}, \tag{6.3}$$

$$Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}, \tag{6.4}$$

for $1 \leq i \leq \ell$.

The Proposition that follows defines some properties of the EEA.

Proposition 4 *Let $n_i = \deg r_i$ in the EEA of f and g. Similarly, let $r_0 = f/lc(f)$, $r_1 = g/lc(g)$ and $r_{l+1} = 0$. Then for $0 \leq i \leq \ell$:*

1. $n_i > n_{i+1}$ *where* $i \neq 0$,

2. $gcd(f, g) = gcd(r_i, r_{i+1}) = r_\ell$,

3. $R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$,

4. $R_i \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$,

5. $s_i\, f + t_i\, g = r_i$,

6. $gcd(s_i, t_i) = 1$,

7. $gcd(r_i, t_i) = gcd(f, t_i)$

The proof of the above proposition is provided in [47, Ch. 3].

Cost analysis for the EEA where $R[x]$, R a Field:

Let $f, g \in R[x]$, with R a field and deg $f >$ deg $g \geq 0$. Furthermore, let $m = \deg f$ and $n = \deg g$. For $0 \leq i \leq \ell + 1$ with $r_{l+1} = 0$, where r_i's are monic remainders generated by the EEA algorithm (Algorithm 8) for the EEA of f and g. We need $n_i(n_{i-1} - n_i)$ multiplications to compute the quotient and remainder of a monic polynomial of degree n_{i-1} divided by a second monic polynomial of degree $n_i < n_{i-1}$. We also need $n_i(n_{i-1} - n_i + 1)$ subtractions. These operations are all in R. To further obtain a monic remainder, we compute one inverse plus n_{i+1} multiplications in R. The total cost of computing all quotients and monic remainders using the EEA algorithm then becomes:

$$\sum_{i=1}^{\ell} (2n_i(n_{i-1} - n_i) + n_i) + \sum_{i=1}^{\ell-1} n_{i+1} \tag{6.5}$$

subtractions and multiplications plus $\ell - 1$ inversions in R. Further simplification will show that the total cost of the EEA is at most $m+2$ inverses and $6mn + O(n)$ additions and multiplications in F.

6.2 The Half-GCD Principle

The HGCD algorithm takes two polynomials f and g, with $n = \deg f > \deg g$, and produces the matrix R_{0j}, where $j = \ell(n/2)$. That is, r_j is the last term of the remainder sequence whose degree exceeds half that of f.

The basic principle behind the HGCD is that, quotients of polynomials of degrees d_1 and d_2 with $d_1 > d_2$, depend only on the leading $2(d_1 - d_2) + 1$ terms of the dividend and the leading $d_1 - d_2 + 1$ terms of the divisor [1]. It can be further shown that relying on the quotient sequence rather than the remainder sequence (as is the case with EEA) gives a better cost in realising the GCD and the Bézout coefficients of two polynomials. For example, given $P_0,\ P_1 \in R[X]$ with $n = \deg P_0 > degP_1 \geq 0$, consider the Euclidean remainder sequence:

$$P_0,\ P_1,\ P_2, ..., P_h \quad (h \geq 1) \tag{6.6}$$

Call the sequence normal if $\deg P_{i-1} = 1 + \deg P_i$ for $i = 2, \ldots, h$. A random choice for P_0, P_1 gives rise to a normal sequence with high probability . The complexity of the algorithm that produces this will be $O(M(n)n)$. The reason for this result is that, if the sequence in 6.6 is normal with $n = h$ and $\deg P_i = n - i(i = 0, ..., n)$ then

$$\sum_{i=0}^{h} \deg P_i = n\,(n-1)/2 \tag{6.7}$$

On the other hand if the focus is changed to the quotient sequence $Q_1,\ Q_2,\ \ldots,\ Q_h$ associated with the remainder sequence, we can derive them in:

$$\sum_{i=0}^{h} \deg Q_i = n \tag{6.8}$$

It can be shown that any member of the remainder sequence can be derived from the Quotient sequence in $O(n\ \log^2 n)$ time. It is this fact that motivated the development of the HGCD algorithm.

Many versions of the algorithm exist but we have produced Algoritm 9 which is an adaptation of the corrected version by Yap [51]. Yap and various others have analysed it to be asymptotically fast when compared to the direct computation of the full Euclidean sequence, in the computation of the EEA. The resulting algorithm that computes the EEA with the use of the HGCD is the Fast EEA (FEEA) for polynomials over a Field, with a complexity of $M(n)log(n)$, counting ring operations in the Field, where $M(n)$ is the FFT multiplication time. Our Algorithm 11 describes it. It uses Algorithm 10 which calls the HGCD to recursively compute the remaining rows of euclidean sequence to produce the matrix that yields the Bézout coefficients.

The source code of our ALDOR implementation of these algorithms is in Appendix G. This implementation outperforms the standard Euclidean algorithm by a factor of 8 at degree 3000 as Figure 6.1 shows.

Algorithm 9 The Half-GCD Algorithm.

INPUT: $a,\ b \in R[x]$, where R is Field, and $n = \deg a \geq \deg b$

OUTPUT: A regular 2×2 matrix M which reduces $gcd(a,b)$ to $gcd(c,d)$ where $\deg c \geq (\deg a)/2 > d$. i.e $M\ \left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right) \rightarrow \left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right)$ meands $\deg c \leq n/2 > \deg d$

M_{hgcd}(a,b) $==$

$d := \deg(a)$; $m := \lceil \frac{d}{2} \rceil$;
if $\deg(b) < m$ **then return** $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$
$a^{\uparrow} := a$ quo x^m
$b^{\uparrow} := b$ quo x^m
$M_1 := \mathrm{M}_{\text{hgcd}}(a^{\uparrow}, b^{\uparrow})$
$\left(\begin{smallmatrix} t \\ s \end{smallmatrix}\right) := M_1 \left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right)$
if $s = 0$ **then return** M_1
$(q, r) :=$ QuotientRemainder(t, s)
if $r = 0$ **then**
    $M_2 := \left(\begin{smallmatrix} 0 & 1 \\ 1 & -q \end{smallmatrix}\right)$
    return $M_2 M_1$
$v :=$ LeadingCoefficient$(r)^{-1}$
$\overline{r} := rv$
$M_2 := \left(\begin{smallmatrix} 0 & 1 \\ v & -vq \end{smallmatrix}\right)$
$\ell := 2m - \deg(s)$
$s^{\uparrow} := s$ quo x^{ℓ}
$r^{\uparrow} := \overline{r}$ quo x^{ℓ}
$M_3 := \mathrm{M}_{\text{hgcd}}(s^{\uparrow}, r^{\uparrow})$
return $M_3 M_2 M_1$

Algorithm 10 The MGCD Algorithm.

INPUT: $a,\, b \in R[x]$, where R is Field, and $\deg a \geq \deg b$

OUTPUT: A regular 2×2 matrix M such that $M \left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right) \rightarrow \left(\begin{smallmatrix} gcd(a,b) \\ 0 \end{smallmatrix}\right)$ and M_{11}, M_{12} produce the Bézout coefficients s and t needed to satisfy $sa + tb = gcd(a, b)$.

$\mathrm{M}_{\mathrm{gcd}}$(a,b) ==

1 $M_1 := \mathrm{M}_{\mathrm{hgcd}}(a, b)$;
2 $\left(\begin{smallmatrix} t \\ s \end{smallmatrix}\right) := M_1 \left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right)$
3 **if** $s = 0$ **then return** M_1
4 $(q, r) :=$ QuotientRemainder(t, s)
5 **if** $r = 0$ **then**
5.1 $M_2 := \left(\begin{smallmatrix} 0 & 1 \\ 1 & -q \end{smallmatrix}\right)$
5.2 **return** $M_2 \times M_1$
6 $v :=$ LeadingCoefficient$(r)^{-1}$
7 $M_2 := \left(\begin{smallmatrix} 0 & 1 \\ v & -vq \end{smallmatrix}\right)$
8 $\overline{r} := rv$
9 $M_3 := \mathrm{M}_{\mathrm{gcd}}(s, \overline{r})$
10 **return** $M_3\, M_2\, M_1$

Algorithm 11 The Fast Extended Euclidean Algorithm.

INPUT: $a,\, b \in R[x]$, where R is Field, and $\deg a \geq \deg b$

OUTPUT: $gcd(a, b)$ and the Bézout coefficients s and t satisfying $sa + tb = gcd(a, b)$.

FEEA(a, b) ==

1 **if** $\deg a > \deg b$ **then**
1.1 $M := M_{gcd}(a, b)$;
1.2 $g := M_{11}\, a + M_{12}\, b$;
1.3 **return** (g, M_{12}, M_{11});
2 **else if** $\deg a = \deg b$ **then**
2.1 **return** FEEA$(b, a \bmod b)$
3 **else return** FEEA(b, a)

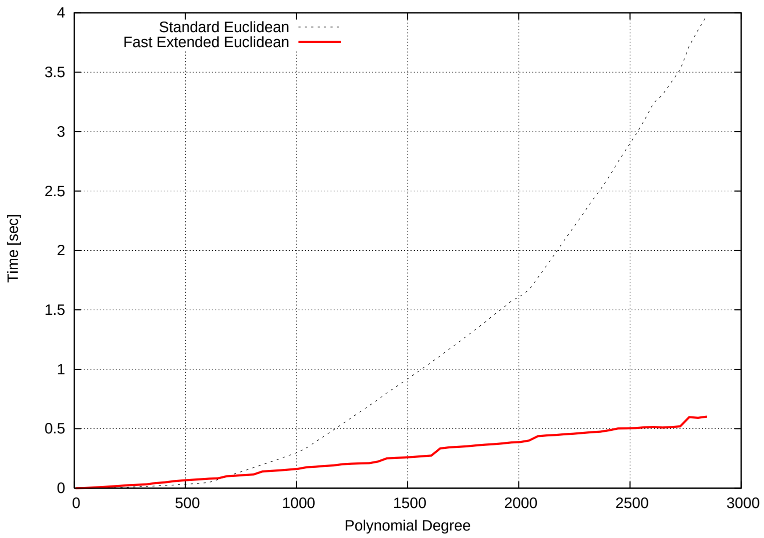

Figure 6.1: Euclidean Algorithm in ALDOR: Standard .vs. Fast Modulo a Prime 27-bit

6.3 Fast Extended Euclidean Algorithm - Benchmarks with NTL and MAGMA

We also compare our generic fast algorithm with the existing implementations in NTL and MAGMA. Unlike ours, the NTL implementation is not over a generic field but over a finite field, and uses improvements like FFT-based polynomial matrix multiplication. MAGMA's performance differ, according to whether we use the `GCD` or `XGCD` commands: we report on both.

Figure 6.2 shows the relative performances; our inputs were degree d polynomials, with a GCD of degree $d/2$. As shown, NTL is the fastest, and ALDOR comes second, within a factor of 2.

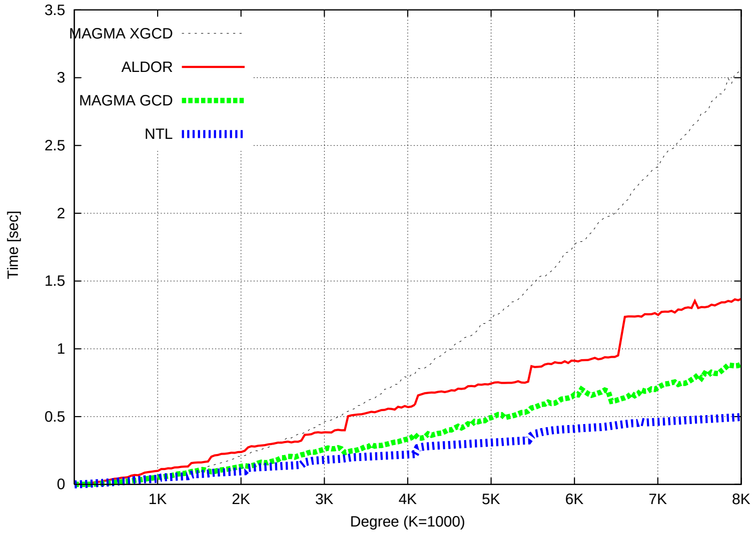

Figure 6.2: EEA: ALDOR vs. NTL vs. MAGMA Modulo a 27-bit Prime.

Chapter 7

Coprime Factorization

Let $a_1, \ldots, a_e$ be multivariate polynomials in $K[x_1, ..., x_n]$ (where K is a field) that encode a set of inequalities in some geometrical problem. More precisely, we have $a_1 \neq 0, ..., a_e \neq 0$. A common factor for some a_i and a_j with $i \neq j$ is a redundant information which coprime factorization removes. The algorithm presented here is discussed in [15] and by Yuzhen Xie in a 2005 talk titled "Fast Simultaneous Reduction and Coprime Factorization." The univariate case is nearly optimal; with a running time complexity that is quasi-linear in $\sum_{1 \leq i \leq e} \deg a_i$ counting operations in K. Other fast algorithms for this problem are given by Gautier and Roch in [23] with a concern for parrallel efficiency, and by Bernstein in [5]. Bernstein's covers a wider setting but with a slightly less efficient complexity. We present the principle, algorithms, and their complexities and our Aldor implementation in the section that follows.

7.1 Principle and Implementation

In our implementation, we consider a GCD domain otherwise known as a UFD (Unique Factorization Domain). The polynomial $K[x_1, ..., x_n]$ is a particular case of GCD domain.

In the algorithms below, we consider $\mathbb{R}$ a Unique Factorization Domain (UFD). We say that an element $a \in R$ is *square-free* if for every prime element $p \in R$ the square of p does not divide a. Furthermore, we say that two elements $a, b \in R$ are

GCD-free if every common factor of a and b is a unit; or that a and b are *copime* or *relatively prime* if there exsist $u, v \in R$ such that $au + bv = 1$ When $\mathbb{R}$ is a Principal Ideal Domain (PID), it is well known that a, b are GCD-free if and only if they are coprime. The definition that follows defines a GCD-free basis and a comprime factorization.

Definition 5 *Let each of* $\mathbf{a} = (a_1, \ldots, a_e)$ *be in* $\mathbb{R}$. *We assume that each of them is square free. A* GCD-free basis *of* $\mathbf{a}$ *is a sequence* $(b_1, ..., b_f)$ *of elements of* $\mathbb{R}$ *such that;*

- *each* b_j *is square free,*
- $\gcd(b_i, b_j) = 1$ *for* $1 \leq i \leq j \leq e$,
- *each* a_i *is a product of some of the* b_j*'s up to a unit factor.*

A coprime factorization *of* $\mathbf{a}$ *is the data of a gcd-free basis* $(b_1, ..., b_f)$ *of* $\mathbf{a}$ *together with a map* g *from* $\{1, \ldots, e\}$ *from* $\{1, \ldots, f\}$ *such that for all* $1 \leq i \leq e$ *the element* a_i *equals the product of the* b_k *for all* $k \in g(i)$, *up to a unit factor.*

Example: Consider the polynomials

$$\begin{aligned} & a_1 = y^2 - 3y + 2 = (y-1)(y-2) \\ \text{and} \quad & a_2 = y^2 - 4y + 3 = (y-1)(y-3) \end{aligned}$$

in $\mathbb{Q}[y]$. Let $b_1 = y-1$, $b_2 = y-2$, $b_3 = y-3$ and $\mathbf{b} = (b_1, b_2, b_3)$. Define $g(1) = \{1, 2\}$ and $g(2) = \{1, 3\}$. Then $\mathbf{b}$ forms a GCD-free basis of (a_1, a_2); and $(\mathbf{b}, g)$ is a coprime factorization of (a_1, a_2).

Our algorithms compute both the GCD-free basis coprime factors. First, we describe algorithms for computing GCD-free basis. Then, we deduce an algorithm for computing a coprime factorization.

The first algorithm (Algorithm 12) computes the gcd of a given polynomial and each polynomial in a given list of polynomials. Recall from Definition 1 that $\mathsf{M}(d)$ is the multiplication time for two $dth-degree$ polynomials. We write $d = \sum_{i=1}^{e} \deg a_i$

which is used in the complexity estimates for all the algorithms in this chapter. Alogorithms 12, 13, and 14 and their complexity estimates are given in the case of $K[x]$,

---

Algorithm 12 multiGCD: Multiple GCD Computation Algorithma.

INPUT: $p \in K[x]$ and $a = [a_1, \ldots, a_e]$, a family of e square-free elements of $\mathbb{K}$.
OUTPUT: $[gcd(p, a_1), \ldots, gcd(p, a_r)]$

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

1 **if** $\deg(p) \geq d$ **then** $p := p \bmod a_1 \ldots a_e$ $[\mathsf{M}(\deg p)]$
2 **for** $i = 1 \ldots e$, compute $p \bmod a_i$ $[\mathsf{M}(d) \log(d)]$
3 $\mathtt{L} := \{\}$; **for** i **in** $1 \ldots e$, do $\mathtt{L} := \mathtt{L} \cup \{\gcd(p_i, a_i)\}$ $\left[\sum_i \mathsf{M}(\deg a_i) \log(\deg a_i)\right]$
4 **return** $\mathtt{L}$

The function $d \mapsto \mathsf{M}(d)\log(d)$ is super-additive, so the complexity at line 3 fits in $O(\mathsf{M}(d)\log(d))$. Hence, the total cost of this algorithm is in $O(\mathsf{M}(d)\log(d))$.

Next, we present the algorithm to compute several pairs of GCDs. This algorithm (see Algorithm 13) uses the subproduct tree algorithms of Chapter 5.

Algorithm 13 allPairGcd: All Pairs of GCD computation Algorithm

INPUT: Two families of polynomials $\{a_1, \ldots, a_e\}, \{b_1, \ldots, b_s\}$, where all a_i (resp all b_i) $\in K[x]$, are squarefree and pairwise coprime.
OUTPUT: all polynomials $\gcd(a_i, b_j)$.

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

1 Build a subproduct tree $\mathsf{Sub}(a_1, \ldots, a_e)$ and let $f = \mathrm{RootOf}(\mathsf{Sub})$ $[\mathsf{M}(d)\log(d)]$
2 Label the root of Sub by multiGcd($f, \{b_1, \ldots, b_s\}$) $[\mathsf{M}(d)\log(d)]$
3 **for** every node $N \in \mathsf{Sub}$, going top-down **repeat**
3.1 **if** N is not a leaf and has label $\mathbf{g}$ **then**
3.1.1 $f_1 :=$ leftChild(N); $f_2 :=$ rightChild(N);
3.1.2 $\{h_1, \ldots, h_s\} :=$multiGcd($f_1, \mathbf{g}$) $[\mathsf{M}(d)\log(d)^2]$
3.1.3 $\{w_1, \ldots, w_s\} :=$ multiGcd($f_2, \mathbf{g}$) $[\mathsf{M}(d)\log(d)^2]$
3.1.4 f_1 is labeled by $\{h_1, \ldots, h_s\}$
3.1.5 f_2 is labeled by $\{w_1, \ldots, w_s\}$
4 **else** yield $\mathbf{g}$

To give the complexity of this algorithm, it is enough to prove that the total number of operations along each row is in $O(\mathsf{M}(d)\log(d))$ resulting in a total cost $O(\mathsf{M}(d)\log(d)^2)$.

The third algorithm (Algorithm 14) computes a merge of the coprime factorizations of two given lists.

Algorithm 14 mergeCoprimeFactorization: Algorithm to merge coprime factorization of two given lists.

INPUT: Two families of polynomials $\{a_1, \ldots, a_e\}, \{b_1, \ldots, b_s\}$, where all a_i (resp all b_i) $\in K[x]$, are squarefree and pairwise coprime.
OUTPUT: Coprime Factorization of the family $\{a_1, \ldots, a_e, b_1, \ldots, b_s\}$.

mergeCoprimeFactorization($\{a_1, \ldots, a_e\}, \{b_1, \ldots, b_s\}$) ==
1 $\{g_{i,j}\}_{1 \le i \le e, 1 \le j \le s}$:= allPairGcd($\{a_1, \ldots, a_e\}, \{b_1, \ldots, b_s\}$) $[\mathsf{M}(d)\log(d)^2]$
2 **for** j **in** $1 \ldots s$ **repeat**
2.1 $\alpha_j := \prod_{1 \le i \le e} g_{i,j}$; $\gamma_j := b_j$ quo α_j $[\mathsf{M}(d)\log(d)]$
2.2 $newbasis$:= basisFind($newbasis, g_{i,j}, gamma_j$)
2.3 $factors$:= newFactors($sizeof(newbasis)$)
3 **for** i **in** $1 \ldots e$ **repeat**
3.1 $\beta_i := \prod_{1 \le j \le s} g_{i,j}$; $\delta_i := a_i$ quo β_i $[\mathsf{M}(d)\log(d)]$
3.2 $newbasis$:= basisFind($newbasis, g_{i,j}, delta_i$)
3.3 $factors$:= newFactors($sizeof(newbasis)$)
4 **return** $\{[a1, \ldots, a_e], [b_1, \ldots, b_s], newbasis, factors\}$

The validity of this algorithm can be easily checked. The estimates for the cost of lines 2.1 and 3.1 include the cost necessary to build a subproduct tree, together with degree estimates on the polynomials α_j and β_i. Hence, the total cost is in $O(\mathsf{M}(d)\log(d)^2)$ operations.

We finally give the algorithm for coprime factorization (in Algorithm 15) followed by the signature of the source code for the entire algorithm implemented as an Aldor domain.

Algorithm 15 CPF: The Coprime Factorization Algorithm.

INPUT: $a = [a_1, \ldots, a_e]$, a family of e square-free elements of K. OUTPUT: $basis == [m_1, \ldots, m_t]$, a family of pairwise coprime and square-free UFDs such that every a_i can be expressed as a product of m_k's, $factors == [\{f_1\}, \ldots, \{f_s\}]$, a list of lists of indexes of m such that $\prod f_i = m.i$, and $generators == \{a\}$

CPF(a) ==

1 $n :=$ sizeof(a)
2 **if** $n = 1$ **then**
2.1 **return** $generators == \{a\}$, $basis == \{a\}$, $factors == \{1\}$
3 **If** $n = 2$ **then**
3.1 $g :=$ gcd($a[1], a[2]$);
3.2 **If** unit?(g) **then**
3.2.1 **return** $[generators == [a[1],\ a[2]],\ basis == [a[1],\ a[2]], factors == [[1],\ [2]]]$;
3.3 **else**
3.3.1 $xx :=$ quotient($a[1], g$);
3.3.2 $yy :=$ quotient($a[2], g$);
3.3.3 **return** $[generators == [a[1],\ a[2]], basis == [xx,\ g,\ yy], factors == [[1,\ 2], [2,\ 3]]]$;
4 $a1 := [a[i]$ **for** $i \in 0 \ldots (m-1)]$;
5 $a2 := [a[i]$ **for** $i \in m \ldots (n-1)]$;
6 **return** mergeCoprimeFactorization($a1, a2$);

```
macro {
    MI == MachineInteger;
    Z==Integer;
}
CoprimeFactorization(R: GcdDomain): with {
  bracket: R -> %;
  bracket: (Array(R), squarefree?: Boolean == true)  ->  %;
    ++ ‘bracket(a)’ returns a coprime factorization of ‘a’
    ++ That is  a pair consisting of an ‘Array(R)’ b
    ++ and an ‘Array(List(MI))’ ‘f’ such that the following holds:
    ++ (1) For any two distinst elements ‘b_i’ and ‘b_j’
    ++ of ‘b’ we have ‘unit?(gcd(b_i, b_j))’
    ++ (2) For all ‘i = 1, .., #a’, the product of the
    ++ ‘b_j’ for ‘j=f.i.1, ... f.i.(# f.i)’ is equal to ‘a_i’
```

```
    ++ Moreover, the elements of 'a' are assumed to be
    ++ squarefree iff 'squarefree?' is 'true'.

    bracket: (R,R,squarefree?: Boolean == true) -> %;
    mbasis: % -> Array(R);
    generators: % -> Array(R);
    factorizations: % ->  List(List(MI));
    ++  subProductTree: Array(R) -> Array(R);
    ++  multiGcd: (R, Array(R)) -> Array(R);

    pairsOfGcd: (Array R,Array R) -> Array (Array R);
    allPairsOfGcd: (Array(R), Array(R)) -> List(List(R)); -- naive
    ++ merge: (%, %) -> %;
    ++ 'merge(a, b)' returns a coprime factorization
    ++ of '[c.0, ..., c.(#c -1),d.0,...,d.(#d -1)]'
    ++ where 'c := generators(a)' and 'd := generators(b)'.

} == add {
    Rep == Record(generators: Array(R), basis: Array(R),
      factorizations: List(List(MI)));
```

see Appendix E for the implementations of the functions.

```
}
```

Chapter 8

Conclusion

In the course of this work, we studied and acquired the necessary background knowledge in abstract algebra, computer algebra systems and algorithmic design concepts. We have used best techniques to design and implement fast polynomial multiplication; fast GCD computation; fast Chinese Remaindering; and fast Coprime Factorization. Moreover, we could implement algorithms that are asymtotically fast, in a high-level language and over generic coefficient ring. Our implementation is competitive with the fastest specialized software comparable. At the end of our work, the ALDOR programming language is enriched with faster algorithms with state-of-the-art techniques in polynomial arithmetic.

With ALDOR gaining wide acceptance in the computer algebra research community as a platform for experimentation, we believe that, this is a worthy and significant contribution.

It is worth recapping some lessons of this work:

- A large variety of algorithms have to be coded and tested to determine crossover points.
- Just casting algorithms from the books into software does not result in optimal implementations.
- Need to customize algorithms for specific data types and sizes.

We believe that our methods will motivate further interest in fast arithmetic research and find application in other computer algebra systems. More work requires to be done in extending our work cover multivariate polynomial arithmetic. This will provide a complete tool-kit for a polynomial solving system.

Appendix A

FFT Multiplication Package

```
------------------------------------
-- Local Functions
------------------------------------
local smallestGreaterOrEqualPowerOfTwo(d:MI):MI == {
  s:MI := 2;
  n:MI := 4;
  while n < d repeat {
    n := shift(n,1); -- n * 2^1
    s := s + 1;
  }
  s;
}

------------------------------------------------------------
-- Exported Functions
------------------------------------------------------------
+++ fftSize returns s given 2 polynomials such that n=2^s >=
+++ deg(pol1)+deg(pol2)+2.  dp and dq are sizes of coeff arrays
fftSize(dp:MI,dq:MI):MI == {
  local d:MI := dp + dq;
  s := smallestGreaterOrEqualPowerOfTwo(d);
  return s;
}

+++ fftSize returns s given 2 polynomials such that n=2^s >=
+++ deg(pol1)+deg(pol2)+2.
fftSize(dp:Z,dq:Z):Z == {
  local d:Z := dp + dq;
  s := smallestGreaterOrEqualPowerOfTwo(d);
  return s;
}

shuffler!(pol:PAK,wspace:PAK,s:MI):PAK == {
  local kkk:MI := 0;
  local ijj,n0,j0:MI;
  local nn:MI := shift(1,s);
  for k in 0..prev(s) repeat { -- for each level starting from original array level 0
    n0 := shift(1,k); -- 2^k
    j0 := nn quo n0;
    -- n / 2^2 is the number of sub-arrays in each level
    --  i.e at level 1 of an 8-element array, we have it split to j0 := 2 sub-arrays
    for ii in 1..n0 repeat { -- for each subarray, reorder
       for i in 0..prev(j0) by 2 repeat { -- left part - even indexes
         jj := j0 * (prev ii);
         ijj := i + jj;
         wspace.kkk := pol.(ijj);
         kkk := next kkk;
       }
       for i in 1..prev(j0) by 2 repeat { -- right part, odd indexes
```

```
        jj := j0 * (prev ii);
        ijj := i + jj;
        wspace.kkk := pol.(ijj);
        kkk := next kkk;
      }
    }
    for i in 0..prev(nn) repeat pol.i := wspace.i;
    kkk := 0
  }
  -- zero-out workspace
  for i in 0..prev(nn) repeat set!(wspace,i,0$K);
  return pol;
} -- end of shuffler!

dft!(pol:PAK,r:MI,wp:PAK):PAK == {
  local np,n,m,mm,m2,kj,kjm,jrs:MI;
  local t,u:K;
  n := shift(1,r);
  np == prev(n);
  mm := shift(n,-1); -- n quo 2
  for s in 1..r repeat {
    -- for each floor of the tree, bottom-up.
    m := shift(1,s); -- 2^s
    m2 := shift(m,-1);
    k:MI := 0;
    while (k <= np) repeat { -- the wings
      -- for each internal node from left to right
      for j in 0..(m2-1) repeat {
        -- combine its two children
--  depending on the level in the butterfly modify wp.j
        kj := k+j;
        kjm := k+j+m2;
        if (s = 1) or (s = r) then {
          -- note that we are using only wp.0 here since when s = 1, m = 2,m2=1,
          -- j can only assume 0 when s=1
          t := wp.j * pol.(kjm);
          u := pol.(kj);
        }
        else {
          -- for the middle levels we want to multiply the respective wing with
          -- wp.jrs because depending on the level, we like to wp's indexes (0,2,4...)
          -- or (0,4,8..)
          jrs := mm*j; -- 2^rs
          t := (wp.jrs) * (pol.kjm);
          u := pol.kj;
        }
        pol.kj := u + t;
        pol.kjm := u-t;
      } -- end of j loop
      k := k + m;
    } -- end of k loop
    mm := shift(mm,-1);
  } -- end of s loop
  return pol;
}

-- this version provides the workspace. coeff1 and coeff2 are
-- assumed to be of equal size
fftProduct!(coeff1:PAK,coeff2:PAK,s:MI,wp:PAK,workspace:PAK):PAK == {
  local n2,n:MI;
  -- new and more efficient shuffle function
  --- shuffler!(pol,ws,n) will return dft-ordered pol array of size n
  --- using workspace ws. ws should be a prim array of size n, each element init zero
  n := shift(1,s); -- 2^s
  n2 := n quo 2; -- n/2
   coeff1 := shuffler!(coeff1,workspace,s); -- this reorders in DFT ordering
  coeff2 := shuffler!(coeff2,workspace,s); -- this reorders in DFT ordering
  coeff1 := dft!(coeff1,s,wp);
```

```
  coeff2 := dft!(coeff2,s,wp);

  -- this is for debugging and could be commented out later
  coeff1 := pakTimes!(coeff1,coeff2,n);
  nk:K := n::Z::K;
  nv := inv(nk);
  coeff1 := shuffler!(coeff1,workspace,s); -- this reorders in DFT ordering
  -- inverse dft starts
  coeff1 := dft!(coeff1,s,wp);
  for i in 1..n2-1 repeat {
    j := n - i;
    tmp := coeff1.i;
    coeff1.i := nv * coeff1.j;
    coeff1.j := nv * tmp;
  }
  coeff1.0 := nv * coeff1.0;
  coeff1.n2 := nv * coeff1.n2;
  return coeff1;
}

OmegaPowers!(op:AK,w:K,n:MI):AK == {
  op.0 := 1;
  for k in 1..prev(n) repeat op.k := op.(k-1) * w;
  return op;
}

+++ omegaPowers(w,n) returns the n first powers of a
+++ primitive n-th root of unity, where w is omega - the prim nth root of unity
OmegaPowers(op:PAK,w:K,n:MI):PAK == {
  op.0 := 1;
  for k in 1..prev(n) repeat op.k := op.(k-1) * w;
  return op;
}

-- times. non-destructive (PAK)
fftMult!(pol1:PAK,pol2:PAK,buf3:PAK,mii:MI):PAK == {
  s := fftSize(mii,mii);
  n:MI := shift(1,s); -- 2^s
  ww:Partial(K) := primitiveRootOfUnity(n);
  assert(not failed?(ww));
  res := fftProduct!(pol1,pol2,s,OmegaPowers(op,retract(ww),n),buf3);
  return res;
}

fftTimesArr(pol1:PAK,dp:MI,pol2:PAK,dq:MI,s:MI):Partial(PAK) == {
  n:MI := shift(1,s); -- 2^s
  ww:Partial(K) := primitiveRootOfUnity(n);
  if failed?(ww) then return failed;
  op:PAK := new(n,0$K);
  res:PAK := fftProduct(pol1,dp,pol2,dq,s,OmegaPowers(op,retract(ww),n));
  free! op;
  return [res];
}
```

Appendix B

Fast Power Series Inversion

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

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

– allocate storage

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

– stores g_{i-1}

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

– stores truncated f

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

– degree of g_i

```
   di := mii - 1;
```

– $X^{\mathtt{ndi-1}} * Y$: the middle product starts

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

– **op stores OmegaPowers up to mii**

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

– f mod X^{2^i}**: truncates f**

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

– **dftf*dftg pointwise**

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

– **takes the end part**

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

– $X^{\mathtt{ndi--1}} * Y$**: the middle product ends**

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

– **to allow dft! in-place of G, save G**

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

– **convert to polynomial**

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

Appendix C

Naive Version of Power Series Inversion

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

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

Appendix D

Fast Chinese Remaindering Algorithm

```
------------------------------------
-- Local Functions
------------------------------------
import from U,AU,MI,FFTMP(K,U),FEEA(K,U);
++  spt_internal(m,i,j,a,b,e) builds the subProduct tree
++  from an array m = m0..mr-1 where r doesn't have to
++  be 2^k for some k m is input array , a is workspace and out array,
++  b and e  are begin and end of resulting array
++  for recursion, i and j and begin and end of input array
++  Designed by Prof Marc Moreno-Maza.
local sptInternal!(m:AU,i:MI,j:MI,a:AU,b:MI,e:MI): Boolean == {
  local r,k,d:MI;
  i = j => { -- if input array has only one item
    assert(b = e);
    a.b := m.i;
    true;
  }
  j = i+1 => {  -- if input array contains 2 items
    assert(e = b + 2);  -- ensure that e is b+2 since tree will have 3 nodes
    a.b := m.i * m.j; -- root of resulting tree
    a.(b+1) := m.i; -- left of resulting tree
    a.e := m.j; -- right of resulting tree
    true;
  }
  r := j - i + 1;  -- size of input
  k := myLog2(r) - 1;
  q := shift(1,k);  -- 2^k size of subtree

  -- Pls note that the shift(q,1) is same as 2*q
  d := b + shift(q,1) - 1; -- size of resulting tree

  sptInternal!(m,i,q+i-1,a,b+1,d);
  sptInternal!(m,q+i,j,a,d+1,e);
  a.b := a.(b+1)*a.(d+1);
  true;
}

-- returns upper bound of log2(r)
local myLog2(r:MI):MI == {
  assert(r ~= 0);
  q:MI := 1;
  kk:MI := 0;

  repeat  {
    q >= r => return kk;
    q := shift(q,1);
```

```
    kk := next kk;
  }
}

+++ sptSize returns the size of the subproduct tree
+++ of an array with r items
local sptSize(r:MI):MI == {
  local k,n,q:MI;
  r = 0 => 0;
  r = 1 =>1;
  r = 2 => 3;
  k := myLog2(r) - 1;
  n := shift(1,k+1) -  1; --2^k+1 -1, size of nodes
  q := shift(1,k);  -- # leaves
  return (1 + n + sptSize(r-q));
}

-- Algorithm 10.16
--  Fast Simultenous modular Reduction
local fSimulMReduction(mr:AU,r:MI,f:U,a:AU,s:MI):AU == {
  k:MI := 0;
  for kk in fSimulReduction!(mr,0,r-1,f,a,0,s-1) repeat {
    mr.k := kk;
    k := next k;
  }
  mr;
}

+++ Algorithm 10.18.  Simultaneous Inverse Computation
+++ m = m0.m1....m(r-1) and m0,m1 in R[x] and mu := [m0..mr-1]
+++ outputs s0,s1,...,sr-1 with si(m/mi) equiv 1 mod mi and deg si < deg mi
local  simulInverse(mu:AU,i:MI,j:MI,m:U):AU == {
  local q,r,s:MI;
  local a,musq,mrr,ss:AU;
  local MM:PAU;
  import from Integer;
  r := j-i+1;
  k := length(r) - 1;
  q := shift(1,k);
  s := sptSize(r); -- size of subproduct tree
  musq := new(r,0$U);
  a := new(s,0$U); -- will store subproduct tree
  for n in 0..prev(r) repeat {
    musq.n := mu.n * mu.n;  -- compute mu^2
  }
  a := subProductTree(musq,r); -- compute subproduct tree for musq

  -- call Algorithm 10.16 with musq to returm m rem (mi)^2
  wspace:AU := new(r,0$U);
  mrr := fSimulMReduction(wspace,r,m,a,s);

  -- note that m rem mi^2 / mi is equiv to m/mi rem mi
  for ii in 0..prev(r) repeat {
    assert(~zero? mu.ii);
    mrr.ii := mrr.ii quo mu.ii;
  }
  ss := new(r,0$U);
  for ii in 0..prev(r) repeat {
    if (degree(mrr.ii)>degree(mu.ii)) then {
MM := MGcd(mrr.ii,mu.ii);
g := MM.0*mrr.ii + MM.1*mu.ii;
si := MM.0;
    }
    else if (degree(mrr.ii) < degree(mu.ii)) then {
      MM := MGcd(mu.ii,mrr.ii);
      g := MM.1*mrr.ii + MM.0*mu.ii;
      si := MM.1;
    }
```

```
    else {
      MM := MGcd(mrr.ii,mrr.ii rem mu.ii);
      g := MM.0*mrr.ii + MM.1*mu.ii;
      si := MM.0;
    }
    assert(g = 1);
    assert(machine(degree(si))<machine(degree(mu.ii)));
    ss.ii := si;
  }
  ss;
}

+++ Linear Combination
+++ input: m in R[x] of any length,i,j for input begin/end indeces
+++ c in R[x] of same length, ic, jc for cu's indices,a for
+++ workspace, b and e for workspace begin/end indices
+++ output: f = sum(cu.i*m/mu.i) where m := mu.0...mu.(r-1)
local linearCombineInternal(m:AU,i:MI,j:MI,c:AU,ic:MI,jc:MI,a:AU,b:MI,e:MI):U == {
  local r0,r1:U;
  local r,k,q,d:MI;
  r := j - i + 1;
  import from Integer;

  for ii in 0..prev(r) repeat assert(machine(degree(c.ii)) < machine(degree(m.ii)));
  r = 1 =>  return c.i;
  k := myLog2(r) - 1;
  q := shift(1,k); -- 2^q
  -- shift(q,1) is 2*q
  d := b + shift(q,1) - 1;
  r0 := linearCombineInternal(m,i,q+i-1,c,ic,q+ic-1,a,b+1,d);
  r1 := linearCombineInternal(m,q+i,j,c,q+ic,jc,a,d+1,e);

  a.(d+1)*r0 + a.(b+1)*r1;
}

------------------------------------------
-- Exported Functions
------------------------------------------
+++ SubProductTree Wrapper
+++ m is array, r is size of array
+++ returns the subproduct tree
subProductTree(m:AU,r:MI):AU == {
  import from U,AU,LU,MI;
  s:MI := sptSize(r);
  a:AU := new(s,0$U);
  sptInternal!(m,0,r-1,a,0,s-1);
  return a;
}

+++ Fast Simultaneous Reduction.
+++ input: mr is workspace and will return the final reduced array
+++ i,j are the begin and end indices of mr, a is subproduct tree, f in R[x], b and e are
+++ begin and end indices of a
+++ output: f rem m0,...,f rem m(r-1) where mi is from M.
fSimulReduction!(mr:AU,i:MI,j:MI,f:U,a:AU,b:MI,e:MI):Generator U == generate {
  local k,q,d:MI;
  local f0,f1:U;
  r := j-i+1; -- size of input array
  r = 1 => {
    yield f;
  }
  k := myLog2(r) - 1;
  q := shift(1,k);
  d := b + shift(q,1) - 1; -- size of subproduct tree
  f0 := f rem a.(b+1);
  f1 := f rem a.(d+1);
  for kk in fSimulReduction!(mr,i,q+i-1,f0,a,b+1,d) repeat yield kk;
  for kk in fSimulReduction!(mr,q+i,j,f1,a,d+1,e) repeat yield kk;
```

```
}

+++ Linear Combine Wrapper
+++ input: m in R[x] of any length,i,j for input begin/end indeces
+++ c in R[x] of same length, ic, jc for cu's indices,a for
+++ workspace, b and e for workspace begin/end indices
+++ output: f = sum(cu.i*m/mu.i) where m := mu.0...mu.(r-1)
linearCombine(m:AU,c:AU,r:MI,a:AU,s:MI):U == {
  ret:U := linearCombineInternal(m,0,r-1,c,0,r-1,a,0,s-1);
  ret;
}

+++ Algorithm 10.22 : Fast Chinese Remainder Algorithm
+++ input m0,...,m(r-1) in R[x], r := 2^k, k in N and v0,...,v(r-1) in R[x]
+++ with deg vi < deg mi for all i.
+++ Output:  The unique polynomial f in R[x] of degree less than n = sum(deg mi)
+++ for all i such that f eq vi mod mi for 0<=i<n
fCRA(mu:AU,r:MI,vu:AU):U == {
  local s,nu,cu,M:AU;
  nu := copy mu;
  cu := copy vu;
  -- 1. Call algorithm 10.3
  M := subProductTree(nu,r);
  -- 2. call algorithm 10.18
  s := simulInverse(nu,0,r-1,M.0);
  -- 3. call algorithm 10.20
  for i in 0..prev(r) repeat {
    assert(~zero? nu.i);
    cu.i := (cu.i*s.i) rem nu.i;
  }
  ssize := sptSize(r);
  linearCombine(nu,cu,r,M,ssize);
}
```

Appendix E

Coprime Factorization

```
macro {
  AR == Array R; LR== List R;
  AI==Array MI;
  M==Array(LR);
  LMI==List MI;
  ALMI == Array (List MI);
}

import from Z,R,MI,AR,LR,Array(Array(MI)),Array(MI),Rep,Character,String,LMI,ALMI;
generators(x: %): Array(R) == rep(x).generators;
mbasis(x: %): Array(R) == rep(x).basis;
factorizations(x: %): List(List(MI)) == rep(x).factorizations;
bracket(x: R): % == per [generators == [x], basis == [x],
factorizations == [[1$MI]] ];
bracket(x: R, y: R, squarefree?: Boolean == true): % == {
  not squarefree? => error "not implemented yet";
  local g: R := gcd(x, y);
  local cf: %;

  if unit?(g) then {
    cf := per [generators == [x, y],
    basis == [x, y],
    factorizations == [[1], [2]] ];
  }
  else {
    local xx: R := quotient(x,g);
    local yy: R := quotient(y,g);

    if unit?(xx) and not unit?(yy) then {
      cf := per [generators == [x, y],
      basis == [g, yy],
      factorizations == [[1], [1, 2]] ];
    }
    if not unit?(xx) and unit?(yy) then {
      cf := per [generators == [x, y],
      basis == [xx, g],
      factorizations == [[1,2], [2]] ];
    }
    if  unit?(xx) and  unit?(yy) then {
      cf := per [generators == [x, y],
      basis == [g],
      factorizations == [[1], [1]] ];
    }
    if not unit?(xx) and not unit?(yy) then {
      cf := per [generators == [x, y],
      basis == [xx, g, yy],
      factorizations == [[1, 2], [2, 3]] ];
    }
  }
```

```
  return cf;
}

bracket(a: Array(R), squarefree?: Boolean == true): % == {
  import from Character, Symbol, String, TextWriter;
  local n: MI := #a;
  assert(n > 0);
  n = 1 => [a.0]@%;
  n = 2 => [a.0,a.1]@%;
  local m: MI := n quo 2;
  local a1: Array(R) := [a.i for i in 0..(m-1)];
  local a2: Array(R) := [a.i for i in m..(n-1)];
  x: % := merge([a1]@%, [a2]@%);
  free!(a1);
  free!(a2);
  x;
}

----------------- --------------
-- INPUT: a,b, two families of polys whose elements are sq free
--        and coprime
-- OUTPUT: coprime factorization of the family {a,b}
merge(aa:%,bb:%):% == {
  a:Array(R) := mbasis(aa);
  b:Array(R) := mbasis(bb);
  oldfactora:List(List MI) := factorizations(aa);
  oldfactorb:List(List MI) := factorizations(bb);
  al:LR := [i for i in a];
  bl:LR := [i for i in b];
  r:MI := #al;
  s:MI := #bl;
  g:Array(AR) := pairsOfGcd(a,b);
  gena:Array(R) := generators(aa);
  genb:Array(R) := generators(bb);
  gen:AR := new(#gena+#genb,0$R);

  -- new generator
  for i in 0..prev(#gena) repeat gen.i := gena.i;
  for i in 0..prev(#genb) repeat gen.(#gena+i) := genb.i;

  --new basis and new factorizations
  newbasis:LR := empty;
  newfactorizations:List (List MI) := empty;

  -- initialize lists alpha,beta, gamma
  alpha:LR := new(#bl,0$R);
  gamma:LR := new(#bl,0$R);
  beta:LR := new(#al,0$R);
  delta:LR := new(#al,0$R);

  for i in 1..r repeat {
    beta.i := 1$R;
    mapi:List MI := empty;
    for j in 1..s repeat {
      if not unit?((g.(i-1)).(j-1)) then {
        beta.i := beta.i * (g.(i-1)).(j-1);
        newbasis := append!((g.(i-1)).(j-1),newbasis);
        mapi := append!(mapi,#newbasis);
      }
    }
    delta.i := quotient(al.i,beta.i);
    if not unit?(delta.i) then {
      newbasis := append!(newbasis,delta.i);
      mapi := append!(mapi,#newbasis);
    }
    newfactorizations := append!(newfactorizations,mapi);
  }
  for j in 1..s repeat {
```

```
    alpha.j := 1$R;
    mapj:List MI := empty;
    for i in 1..r repeat {
      if not unit?((g.(i-1)).(j-1)) then {
        alpha.j := alpha.j * (g.(i-1)).(j-1);
(test:LR, p:MI) := find(newbasis,(g.(i-1)).(j-1));
mapj := append!(mapj,p);
      }
    }
    gamma.j := quotient(bl.j,alpha.j);
    if not unit?(gamma.j) then {
      newbasis := append!(newbasis,gamma.j);
      mapj := append!(mapj,#newbasis);
    }
    newfactorizations := append!(newfactorizations,mapj);
  }
  tt:List MI:= empty;
  finalfact:List(List MI) := new(#oldfactora+#oldfactorb,tt);

  -- construct final factorizations
  for i in 1..#oldfactora repeat
    for j in 1..#(oldfactora.i) repeat
      finalfact.i := append!(finalfact.i,oldfactora.i.j);
    for i in 1..#oldfactorb repeat
      for j in 1..#(oldfactorb.i) repeat
        finalfact.(i+#oldfactora) := append!(finalfact.(i+#oldfactora), newfactorizations.(oldfactorb.i.j)+#a));
  newbasis2:AR := [i for i in newbasis];
  per  [gen,newbasis2,finalfact];
}
```

– INPUT: a,b, two array of polynomials where all a_i and b_i – are squarefree and pairwise coprime. – OUTPUT: all polynomials $gcd(a_i, b_i)$

```
pairsOfGcd(a:AR,b:AR):Array AR ==  {
  s:MI := sptSize(#a);
  (tr:AR,pci:AI) := subProductTree(a,#a);
  L:LR := new(#b,0$R);
  label:M := new(s,L);
  wspace:LR := new(1,0$R);
  label.0 := multiGcd(wspace,tr.0,b);
  wspace:LR := new(1,0$R);
  kk:MI :=0;
  bb:AR := new(#b,0$R);
  ret:Array (Array R) := new(#a,bb);
  -- for each node N
  for k in 0..prev(s) repeat {
    if not leaf?(k,pci) then {
      ilc:MI := lchild(k,pci);
      irc:MI := rchild(k,pci);
      lc:R := tr.ilc;
      rc:R := tr.irc;
      labelk:AR := [nn for nn in label.k];
      wspace:LR := new(1,0$R);
      label.ilc := multiGcd(wspace,lc,labelk);
      wspace:LR := new(1,0$R);
      label.irc := multiGcd(wspace,rc,labelk);
    }
    else {
      ret.kk := [i for i in label.k]$AR;
      kk := kk + 1;
    }
  }
  ret;
}
```

– input: $f, a1, ..., a_n$ in $K[x]$, $d = sum(degai)$ – wspace is a 1-element list, n is size of a – output: unique $gcd(p, a1), ..., gcd(p,$

```
local multiGcd(wspace:LR,f:R,a:AR):(LR) == {
```

```
  local buf2,tr:AR;
  local buf1:LR;
  local n:MI;
  n := #a;
  (tr:AR,pci:AI) := subProductTree(a,n);  -- O(M(d)log(d))
  f := f - quotient(f,tr.0)*tr.0; --f rem tr.0; -- O~(deg f)
  buf1 := new(1,0$R);
  buf2 := new(n,0$R);  -- stores result from reduction
```

– **computes** $f \bmod a_i$

```
  fSimulMReduce!(buf2,n,f,tr,sptSize(n)); -- M(d)log(d)
  wspace.1 := gcd(buf2.0,a.0); -- result
  for i in 1..prev(n) repeat {
    tmp1:R := gcd(buf2.i,a.i);
    append!(wspace,tmp1);
  }
  free! buf1;
  free! buf2;
  return wspace;
}
```

Appendix F

Fast Monic Division

```
--fastMonicDivide to compute f quo g and f rem g  which are polynomials
-- input: a,b in D[x], b is monic and ~= 0, D is a ring (comm with 1)
-- output: q,r in D[x] such that a := qb+r, deg r < deg b
fastMonicDivide(a:U, b:U): (U,U) == {
  assert(monic?(b));
  local m: U == monom;
  local nn: Z := degree(a) - degree(b);
  local  q, r,fff,result: U;
  if nn<0 then {
    q := 0;
    r := a;
  }
  else{
    fff := revert(b);
    result := fastModInverse(fff,nn+1);
    q := revert(a)*result;
    q := truncate!(q,nn+1)$U;
    if (degree(q) = nn) then
      q:= revert(q);
    else {
      q := m^(nn - degree(q)) * revert(q);
    }
    r := a-b *q;
   }
   return (q,r);
}
```

Appendix G

Fast Extended Euclidean Algorithm

```
macro {
  Z == Integer;
  ARR == PrimitiveArray;
  AK == Array K;
  MI == MachineInteger;
  FFTPKG == FFTPolMultiplicationPackage;
  PRIMRT == PrimitiveRootOfUnityFinder;
}
FastExtendedEuclideanAlgorithmPackage(K: Field, U: UnivariatePolynomialAlgebra(K)): with {
  times!: (ARR U, ARR U) -> ARR U;
  ++ times(m1,m2) returns the product of m1 and m2 viewed as 2 by 2 matrices over U
  ++ this operation overwrites its first argument
  times:  (ARR U, U, U) -> (U,U);
  ++ times(m, p, q) returns the product of m by v where v is the column-vector
  ++ with p as first coordinate and q as second coordinate
  fastModInverse: (U,Z) -> U;
  fastModInverse: (U,MI) -> U;
  modularInversion: (U,Z) -> U;
  modularInversion: (U,MI) -> U;
  -- Less Efficient
  -- Using newton iteration for Computing
  -- the number of iteration
  fastMonicDivide: (AK, AK) -> (U,U);
  fastMonicDivide: (U,U) -> (U,U);
  ++ it does one division by hand which is computing quotient and reminder
  halfGcd: (U,U) -> (ARR U);
  ++ fastExtendedEuclideanAlgorithm which computes the gcd in the middle of rows
  MGcd: (U,U)-> (ARR U);
  ++ it computes the gcd of the row that we are intrested
  Gcd: (U,U)-> U;
  ++ it computes the gcd of the 2 polynomials

} == add {
  local zero__K : K == 0;
  local two__K: K == 1 + 1;

  import from K,FFTPKG(K),PRIMRT(K);
  import from (U pretend EuclideanDomain);
  import from TextWriter,String,Character;
  M == ARR U;
  import from Z,AK,M,U,MI;

 --------------------------------
 --local functions
 --------------------------------
```

```
--- MatrixMult!(RR:MM,a,b,c,d) returns RR.0 := a*RR.0 etc
local MatrixMult!(RR:M,a:U,b:U,c:U,d:U):M == {
 RR.0 := a * RR.0;
 RR.1 := b * RR.1;
 RR.2 := c * RR.2;
 RR.3 := d * RR.3;
 RR;
}

--- MatrixMan!(RR:MM,a,b,c,d) returns RR.0 := a,RR.1:=b etc
local MatrixMan!(RR:M,a:U,b:U,c:U,d:U):M == {
 RR.0 := a;
 RR.1 := b;
 RR.2 := c;
 RR.3 := d;
 RR;
}

--------------------
-- Exported Functions
--------------------
times!(m1: M, m2: M): M == {
 local a, b, c, d: U;
 a: := m1.0 * m2.0 + m1.1 * m2.2;
 b: := m1.0 * m2.1 + m1.1 * m2.3;
 c: := m1.2 * m2.0 + m1.3 * m2.2;
 d: := m1.2 * m2.1 + m1.3 * m2.3;
 m1.0 := a;
 m1.1 := b;
 m1.2 := c;
 m1.3 := d;
 return  m1;
}

times(m1: M , v1: U, v2: U): (U,U) == {
 import from MI, M;
 (v1 * m1.0 + v2 * m1.1, v1 * m1.2 + v2 * m1.3);
}

---------------------------------------------------------------------
--%%% Naive version - the algorithm 9.10 P-adic inversion using newton iteration %%%--
---------------------------------------------------------------------
modularInversion(f:U,n:Z):U == {
 local m: U == monom;
 assert(one?(trailingCoefficient(f)));

 local g0:U:=1;
 local g__old:U:=1;
 local g__new:U:=1;
 local r,mii:MI;
 local mi:U;

 if PowerOfTwo?(n) then r := length(n)-1;
 else  r := length(n);

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

modularInversion(u: U, mi: MI): U == {
 import from Z;
 z: Integer := mi::Z;
 modularInversion(u, z);
```

```
}

modInverseSp(f:U,n:MI):U == {
 nz:Z := n::Z;
 modInverseSp(f,nz);
}

+++ unvectorize(pak,n) will return a polynomial from
+++ coefficients vector ak of size n
local unvectorize(pak:AK,n:MI):U == {
 import from K,U,MI,Z,AK;
 local m:MI;
 m := prev(n);
 local ret:U := term(pak.m, m::Z );
 for k in prev(m)..0 by -1 repeat {
   if zero?(pak.k) then iterate;
   ret := add!(ret,pak.k,k::Z);
 }
 return ret;
}

fastModInverse(f:U, mi:MI): U == {
 z:Z  := mi::Z;
 fastModInverse(f, z);
}

-- More space-efficient version
-- Uses coefficient vector and the fft trick
-- input f in R[x] with f(0) = 1, and l in N where l is degree gap+1
-- output: g in R[x] satisfying fg ~ 1 mod x^l
fastModInverse(f:U,n:Z):U == {
 import from Z,MI;
 local dftf,dftg,Y,G,workspace,dftw,op,coeff:AK;
 local di__1,di,r,mii:MI;
 local res:U;
 local wi:K;
 assert(one?(trailingCoefficient(f)));

 if PowerOfTwo?(n) then r := length(n)-1;
 else  r := length(n);
 nn:MI := shift(1,r); -- 2^r
 dftg := new(nn,0$K);
 Y := new(nn,0$K);
 G := new(nn,0$K);
 workspace := new(nn,0$K);
 op := new(nn,0$K);
 G.0 := 1$K;
 dftg.0 := 1$K;
 coeff := new(nn,0$K);
 dftf := new(nn,0$K);  -- holds truncated f
 dftw := new(nn,0$K);
 kk:MI := 0;
 for k in coefficients(f) repeat {
   kk = nn => break;
   coeff.kk := k;
   kk := next(kk);
 }
 for i in 1..r repeat {
   mii := shift(1,i); -- 2^i
   di := mii - 1; --  degree of g_i
   w:Partial K := primitiveRootOfUnity(mii);
   wi := retract(w);
   OmegaPowers!(op,wi,mii);
   dftg := dft!(dftg,mii,i,op,workspace);
   -- buf4 = f mod x^2^i
   for j in 0..di repeat dftf.j := coeff.j; -- truncates f
   dftf := dft!(dftf,mii,i,op,workspace);
   for j in 0..di repeat dftf.j := dftf.j*dftg.j;  -- dftf*dftg pointwise
```

```
    dftf := dft!(dftf,mii,i,op,workspace);
    dftf := idft!(dftf,mii); -- invert dft
    di__1 := shift(1,i-1) - 1; --  degree of g_i_1
    ndi__1 := next di__1;
    kk := 0;
    for j in next di__1..di repeat {
      dftw.kk := dftf.j;
      kk:=next kk;
    }
    dftw := dft!(dftw,mii,i,op,workspace);
    for j in 0..di repeat dftg.j := dftg.j * dftw.j;  -- dftw*dftg pointwise
    dftg := dft!(dftg,mii,i,op,workspace);
    dftg := idft!(dftg,mii);
    -- to do G - x^(next di__1)*Y
    for j in 0..di__1 repeat { -- x^(next di__1)*Y
Y.(j+(ndi__1)) := dftg.j;
    }
    for j in ndi__1..di repeat G.j := G.j - Y.j;
    for j in 0..di repeat dftg.j := G.j;  -- to allow dft! of G
  }
  res := unvectorize(dftg,nn); -- convert to polynomial;
  free!(dftg);
  free!(dftf);
  free!(dftw);
  free!(workspace);
  free!(op);
  free!(coeff);
  return res;
}

 +++ unvectorize(pak) will return a polynomial from
 +++ coefficients vector pak
 local unvectorize(pak:AK):U == {
 import from K,U,MI,Z,AK;
  local n,m:MI;
  n:MI := #pak;
  m := prev(n);
  local ret:U := term(pak.m, m::Z );
  for k in prev(m)..0 by -1 repeat {
    if zero?(pak.k) then iterate;
    ret := add!(ret,pak.k,k::Z);
  }
  return ret;
 }

 +++ multiplies a vector by an integer, in-place.
 +++ destructive
 local times!(f:AK,n:MI):AK == {
  import from MI,AK,K,Z;
  sz:MI := #f;
  for k in 0..prev(sz) repeat {
    set!(f,k, (n::Z::K)*f.k);
  }
  return f;
 }

 -- destructively truncates an array  to n.
 -- ie return f.0..f.(n-1)
 local truncate!(f:AK,n:MI):AK == {
  import from AK,MI,K;
  if #f <= n then return f;
  resize!(f,n);
  return f;
 }


 ---------------------------------------------
 -----------%%  Power Of Two  %%---------------
```

```
 -------------------------------------------
 local PowerOfTwo?(n:Z):Boolean=={
    n=0 => return (false);
    n=1 => return (true);
    aaa:MI := length(n)-1;
    bbb := prev(aaa);
    for i in 0.. bbb repeat{
if bit?(n,i) then return (false);
    }
    if bit?(n,bbb) then return (true);
    return(false);
 }

 ---fastMonicDivide to compute f quo g and f rem g  which are polynomials-------
 fastMonicDivide(a: U, b: U): (U,U) == {
  local m: U == monom;
  nn: Z := degree(a) - degree(b);
  local  q, r: U;
  if nn<0 then {
     q := zero__K::U;
     r := a;
  }
  else {
    fff: U := revert(b);
    result:U := fastModInverse(fff,nn+1);
    q := revert(a)*result;
    q := truncate!(q,nn+1);
    if (degree(q) = nn) then q:= revert(q);
    else q := m^(nn - degree(q)) * revert(q);
    r := a - b * q;
  }
  return (q,r);
 }

 -- makes x^n a vector
 local vectorize(wspace:AK,n:MI):AK == {
    import from AK;
    resize!(wspace,n);
    wspace.(n+1) := 1$K;
    return wspace;
 }

 --- the HalfGCD Alghorithm ------------
 -- d := next(degree(r0)) / 2
 halfGcd(r0:U,r1:U):M  == {
  assert(degree(r0) > degree(r1));
  d0:Z := degree(r0);
  d1:Z := degree(r1);
  s:MI := 4;
  m:Z := (d0+1) quo 2;
  if degree(r1) <= m then return [1,0,0,1]$M;

  r00:U := copy r0;
  r11:U := copy r1;
  shift!(r00,-m)$U; -- r00 quo x^m
  shift!(r11,-m)$U; -- r00 quo x^m
  M1:M := halfGcd(r00, r11);

  (r__j__minus__1:U,r__j:U) := times(M1,r0,r1);
  if zero?(r__j) then return M1;

  (q__j:U, rem__j:U) := fastMonicDivide(r__j__minus__1, r__j);
  rho__j__plus__1: K := leadingCoefficient(rem__j);
  (r__j__plus__1 : U, lc__f : U , inv__lc__f: U) := unitNormal(rem__j);
  invrho__j__plus__1: U := inv(rho__j__plus__1)::U;
  M2: M := new(s);
  M2.0 := 0;
  M2.1 := 1;
```

```
 M2.2 := (invrho__j__plus__1);
 M2.3 := -invrho__j__plus__1 * q__j;
 M2 := times!(M2,M1);
 if zero?(r__j__plus__1) then return M2;
 dstar: Z := 2*m-(degree(r__j));
 r__j1:U := copy r__j;
 r__j__plus__11:U := copy r__j__plus__1;
 shift!(r__j1,-dstar);
 shift!(r__j__plus__11,-dstar);
 M3:M := halfGcd(r__j1,r__j__plus__11);
 return times!(M3,M2);
}


-- The GCD Algorithm -------------------------
MGcd(r0:U,r1:U):(M)  == {
 local s: MI == 4;
 M1:M := halfGcd(r0,r1);
 (r__j__minus__1:U,r__j:U) := times(M1,r0,r1);

  if zero?(r__j) then return M1;

  (q__j:U, rem__j:U) := fastMonicDivide(r__j__minus__1, r__j);
  rho__j__plus__1:K := leadingCoefficient(rem__j);
  (r__j__plus__1 : U, lc__f : U , inv__lc__f: U) := unitNormal(rem__j);
  invrho__j__plus__1: U := inv(rho__j__plus__1)::U;

  result:M := new(s);

  MatrixMan!(result,0,1,invrho__j__plus__1,(-invrho__j__plus__1*q__j));
  M1 := times!(result,M1);
  if zero?(r__j__plus__1) then return M1;
  SS:M := MGcd(r__j,r__j__plus__1);
  ret:M := times!(SS,M1);
  return ret;
}

+++ Gcd(a,b) returns the gcd of the two polynomials a and b.
Gcd(a:U,b:U):U == {
 import from U,ARR U,MI,M;
 local MM:M;
 local d1:MI := machine(degree(a))$Integer;
 local d2:MI := machine(degree(b))$Integer;
 if d1 > d2 then {
   MM := MGcd(a,b);
   g := MM.0 * a + MM.1 * b;
   return g;
 }
 else if d1 = d2 then
 return Gcd(b, a rem b);
 else return Gcd(b,a);
}

local shuffle!(pol:AK,wspace:AK,s:MI):AK == {
 local kkk:MI := 0;
 local ijj,n0,j0:MI;
 local nn:MI := shift(1,s);
 for k in 0..prev(s) repeat { -- for each level starting from original array level 0
   n0 := shift(1,k); -- 2^k
   j0 := nn quo n0;
   -- n / 2^2 is the number of sub-arrays in each level
   --  i.e at level 1 of an 8-element array, we have it split to j0 := 2 sub-arrays
   for ii in 1..n0 repeat { -- for each subarray, reorder
     for i in 0..prev(j0) by 2 repeat { -- left part - even indexes
       jj := j0 * (prev ii);
       ijj := i+jj;
       wspace.kkk := pol.(ijj);
       kkk := next kkk;
```

```
      }
      for i in 1..prev(j0) by 2 repeat { -- right part, odd indexes
        jj := j0 * (prev ii);
        ijj := i + jj;
        wspace.kkk := pol.(ijj);
        kkk := next kkk;
      }
    }
    for i in 0..prev(nn) repeat
      pol.i := wspace.i; -- copy wspace to pol
    kkk:=0
  }

  -- zero-out workspace
  for i in 0..prev(nn) repeat wspace.i := 0$K;
  return pol;
} -- end of shuffle!

+++ dft(pol,n,r, omega__powers,workspace) will return the DFT
+++ of pol at omega, provided that degree(pol) < size(omega__powers)
+++ pol is a polynomial represented as array of its coefficients and n | n is 2^r
+++ pol is not in DFT ordering
+++workspace has to be of length 2^r
local dft!(pol:AK,n:MI,r:MI,wp:AK,workspace:AK):AK == {
  local np,m,mm,m2,kj,kjm,jrs:MI;
  local t,u:K;
  np == prev(n);
  mm := shift(n,-1); -- n quo 2
  pol := shuffle!(pol,workspace,r); -- this reorders in DFT ordering
  for s in 1..r repeat {
  -- for each floor of the tree, bottom-up.
    m := shift(1,s); -- 2^s
    m2 := shift(m,-1);
    k:MI := 0;
    while (k <= np) repeat { -- the wings
      -- for each internal node from left to right
      for j in 0..(m2-1) repeat {
        -- combine its two children
        --  depending on the level in the butterfly modify wp.j
        kj := k+j;
        kjm := k+j+m2;
        if (s = 1) or (s = r) then {
          -- note that we are using only wp.0 here since when s = 1, m = 2,m2=1,
          -- j can only assume 0 when s=1
          t := wp.j * pol.(kjm);
          u := pol.(kj);
        }
        else {
          -- some explanation here
          -- for the middle levels we want to multiply the respective wing with
          -- wp.jrs because depending on the level, we like to wp's indexes (0,2,4...)
          -- or (0,4,8..)
          jrs := mm*j; -- 2^rs
          t := (wp.jrs) * (pol.kjm);
          u := pol.kj;
        }
        pol.kj := u + t;
        pol.kjm := u-t;
      } -- end of j loop
      k := k + m;
    } -- end of k loop
    mm := shift(mm,-1);
  } -- end of s loop
  return pol;
}

-- idft will produce inverted DFT from the dft array
local idft!(coeff1:AK,n:MI):AK == {
```

```
    nv:K := inv(n::Z::K);
    n2:MI := shift(n,-1); -- n quo 2
    for i in 1..n2-1 repeat {
      j := n - i;
      tmp := coeff1.i;
      coeff1.i := nv * coeff1.j;
      coeff1.j := nv * tmp;
    }
    coeff1.0 := nv * coeff1.0;
    coeff1.n2 := nv * coeff1.n2;
    return coeff1;
  }

  +++ Multiplies two arrays by pointwise multiplication
  +++ We assume that they are of equal size Destructive
  local pakTimes!(a:AK,b:AK,n:MI):AK == {
    import from AK;
    for k in 0..prev(n) repeat a.k := a.k * b.k;
    return a;
  }

  -- for lower degrees of A less the crossover
  local iterHGCD(A:U,B:U):M == {
   Mout:M := [1,0,0,1]$M;
   goal:Z := degree(A) quo 2;
   if (degree(B) <= goal) then return Mout;

   while (degree(B) > goal) repeat {
     Q:U := A quo B;
     A:U := A mod B;
     swap(A,B);
     t:U := Q*Mout.2;
     t := Mout.0 -t;
     Mout.0 := Mout.2;
     Mout.2 := copy t;
     t := Q * Mout.3;
     t := Mout.1 - t;
     Mout.3 := copy t;
   }
   return Mout;
  }
}
```

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1974.

[2] aldor.org. *The* Aldor *compiler web site*. University of Western Ontario, Canada, 2002.

[3] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[4] D. J. Bernstein. Removing redundancy in high-precision newton iteration. `http://cr.yp.to/papers.html#fastnewton`, 2004.

[5] D. J. Bernstein. Factoring into coprimes in essentially linear time. *J. Algorithms*, 54(1):1–30, 2005.

[6] R. P. Brent. Algorithms for matrix multiplication. Master's thesis, Stanford University, 1970. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/.

[7] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun. Fast solution of Toeplitz systems of equations and computations of Padé approximants. *Journal of Algorithms*, 1:259–295, 1980.

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

[9] R. E. Bryant and D. O'Hallaron. *Computer Systems*. Pearson Education, Inc., 2003.

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

[11] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier +series. *Math. Comp.*, 19:297–301, 1965.

[12] K. D. Cooper and L. Torczon. *Engineering A Compiler*. Morgan Kaufmann, 2004.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2002.

[14] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Equiprojectable decompositions of zero-dimensional varieties. In *ICPSS*, pages 69–71. University of Paris 6, France, 2004.

[15] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the d5 principle. In *ISSAC'05*, 2005. to appear.

[16] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *submitted to TC2006*, 2006.

[17] X. Dahan and É. Schost. Sharp estimates for triangular sets. In *ISSAC 04*, pages 103–110. ACM Press, 2004.

[18] J. Della Dora, C. Discrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL 85 Vol. 2*, volume 204 of *Lect. Notes in Comp. Sci.*, pages 289–290. Springer-Verlag, 1985.

[19] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC 02*, pages 63–74. ACM, 2002.

[20] Stephen Watt et al. A first report on the $A^{\#}$ compiler. In *ISSAC 94*. ACM Press, 1994.

[21] R. J. Fateman. Vector-based polynomial recursive representation arithmetic. 1990. `http://www.norvig.com/ltd/test/poly.dylan`.

[22] Free Software Foundation. *GNU Multiple Precision Arithmetic Library.* `http://swox.com/gmp/`, 2000-2005.

[23] T. Gautier and J.-L. Roch. NC2 computation of gcd-free basis and application to parallel algebraic numbers computation. In *PASCO '97: Proceedings of the second international symposium on Parallel symbolic computation*, pages 31–37. ACM Press, 1997.

[24] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluver Acadenic Publishers, 1992.

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

[26] J. van der Hoeven. Truncated fourier transform. In *Proc. ISSAC'04*. ACM Press, 2004.

[27] http://www.linalg.org/. *LinBox.* The LinBox group, 2005.

[28] http://www.orcca.on.ca. *The Ontario Research Centre for Computer Algebra.* webmasterorcca.on.ca, 2005.

[29] The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry.* `http://magma.maths.usyd.edu.au/magma/`, 2004-2006.

[30] R. D. Jenks and R. S. Sutor. *AXIOM the Scientific Computation System.* Springer-Verlag, 1992.

[31] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *Proc. ISSAC'05*. ACM Press, 2005.

[32] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[33] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1999.

[34] Grégoire Lecerf and Éric Schost. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research*, 5(1), September 2003.

[35] X. Li. Efficient management of symbolic computations with polynomials. Master's thesis, University of Western Ontario, August 2005.

[36] D. Lichtblau. Half-gcd and fast rational recovery. In *ISSAC 05*, pages 231–236. ACM Press, 2005.

[37] R. T. Moenck. Practical fast polynomial multiplication. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148, New York, NY, USA, 1976. ACM Press.

[38] M. Moreno Maza. Algorithmic properties of polynomial rings, 2005. preprint.

[39] M. Moreno Maza and C. Onacea. On polynomial gcds over products of fields presented by towers of simple extensions. Technical report, University of Western Ontario, 2004.

[40] Bill Page. *The* AXIOM *developers web site.* `http://page.axiom-developer.org`, 2005.

[41] Victor Y. Pan. Simple multivariate polynomial multiplication. *J. Symb. Comp.*, 18(3):183–186, 1994.

[42] Arnold Schönhage. The fundamental theorem of algebra in terms of computional complexity - prelim report. *Math. Inst. Universität Tübingen*, 1982.

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

[44] Victor Shoup. *The Number Theory Library.* Dr. Victor Shoup, 1990.

[45] Hans J. Stetter. *Numerical Polynomial Algebra.* Society for Industrial and Applied Mathematics, Philadephia, 2004.

[46] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik.*, 13:354–356, 1969.

[47] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[48] S. M. Watt. FOAM: A first order abstract machine. Technical report, IBM Research, 1994.

[49] Stephen M. Watt. The $A^{\#}$ programming language and its compiler. Technical report, IBM Research, 1993.

[50] Stephen M. Watt. *A# Language Reference V0.35.* IBM Research Report RC 19530, 1994.

[51] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1993.