Efficient Computation with Sparse and Dense Polynomials

by

Daniel Steven Roche

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Computations with polynomials are at the heart of any computer algebra system and also have many applications in engineering, coding theory, and cryptography. Generally speaking, the low-level polynomial computations of interest can be classified as arithmetic operations, algebraic computations, and inverse symbolic problems. New algorithms are presented in all these areas which improve on the state of the art in both theoretical and practical performance.

Traditionally, polynomials may be represented in a computer in one of two ways: as a "dense" array of all possible coefficients up to the polynomial's degree, or as a "sparse" list of coefficient-exponent tuples. In the latter case, zero terms are not explicitly written, giving a potentially more compact representation.

In the area of arithmetic operations, new algorithms are presented for the multiplication of dense polynomials. These have the same asymptotic time cost of the fastest existing approaches, but reduce the intermediate storage required from linear in the size of the input to a constant amount. Two different algorithms for so-called "adaptive" multiplication are also presented which effectively provide a gradient between existing sparse and dense algorithms, giving a large improvement in many cases while never performing significantly worse than the best existing approaches.

Algebraic computations on sparse polynomials are considered as well. The first known polynomial-time algorithm to detect when a sparse polynomial is a perfect power is presented, along with two different approaches to computing the perfect power factorization.

Inverse symbolic problems are those for which the challenge is to compute a symbolic mathematical representation of a program or "black box". First, new algorithms are presented which improve the complexity of interpolation for sparse polynomials with coefficients in finite fields or approximate complex numbers. Second, the first polynomial-time algorithm for the more general problem of sparsest-shift interpolation is presented.

The practical performance of all these algorithms is demonstrated with implementations in a high-performance library and compared to existing software and previous techniques.

Acknowledgements

There are many people to praise (or blame) for the quality of this work. First and foremost is my wife, Leah, who agreed to come with me to Canada and has tolerated countless long nights, meaningless scribble, and boring conversations for almost five years. Despite this document's completion, these habits of mine are likely to persist, and my hope is that her patience for them will as well.

I have always been very lucky to have the support of my family, and in particular my parents, Duane and Sheryl, deserve credit for always encouraging my academic pursuits. They managed to instill in me the importance of education without squelching creativity or the joy of learning, and I thank them for it.

A great debt is owed to my many excellent primary, secondary, and post-secondary school teachers and professors for encouraging me to ask interesting questions. I especially thank Dave Saunders at the University of Delaware for introducing me to the wonderful world of computer algebra in the summer of 2004.

In Waterloo, my academic pursuits have been moderated by healthy doses of musical expression. I thank the members of orchestra@uwaterloo, the Wellington Winds, Ebytown Brass, Brass Essentials, and the Guelph Symphony Orchestra for so many evenings and weekends of great playing.

Conversation with my fellow students in the Symbolic Computation Group lab ranges from polynomials to politics and is always enlightening. A particular thanks goes to my three office mates over the years, Scott Cowan, Curtis Bright, and Somit Gupta, as well as Reinhold Burger, Mustafa Elsheik, Myung Sub Kim, Scott MacLean, Jason Peasgood, Nam Pham, and Hrushikesh Tilak.

I have been fortunate to discuss the topics of this thesis with numerous top researchers in the world, whose valuable insights have been instrumental in various ways: Jacques Carette, James Davenport, Jean-Guillaume Dumas, Richard Fateman, Joachim von zur Gathen, Jürgen Gerhard, Pascal Giorgi, Jeremy Johnson, Pascal Koiran, George Labahn, Wen-Shin Lee, Marc Moreno Maza, John May, Michael Monagan, Roman Pearce, Clément Pernet, Erik Postma, Éric Schost, Igor Shparlinski, and Stephen Watt.

David Harvey of New York University was kind enough to email me with some ideas a few years ago which we developed together, yielding the main results presented here in Chapter 3. I thank and admire David for his great patience in working with me.

The members of my thesis examining committee all provided great feedback on this document, as well as some lively discussion. Thanks to Erich Kaltofen, Kevin Hare, Ian Munro, and Jeff Shallit.

The greatest thanks for this thesis goes to my two supervisors, Mark Giesbrecht and Arne Storjohann. Our countless conversations over the last five years have covered not only the content and direction of my research, but also teaching, publishing, and how to be a university faculty member. Their generosity in time, ideas, and funding have truly allowed me to flourish as a graduate student.

Finally, I am very grateful to the generous financial support during the course of my studies from the David R. Cheriton School of Computer Science, the Mathematics of Information Technology and Complex Systems (MITACS) research network, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Table of Contents

List of Algorithms

List of Tables

List of Figures

This little polynomial should keep the computer so busy it doesn't even know we're here.

—Chris Knight (Val Kilmer), *Real Genius* (1985)

Chapter 1

Introduction

This thesis presents new algorithms for computations with polynomials. Such computations form the basis of some of the most important problems in computational mathematics, from factorization to solving nonlinear systems. We seek algorithms that give improvements both in theoretical efficiency as well as practical performance. As the operations we consider are important subroutines in a range of applications, our low-level performance gains will produce improvements in a variety of problems.

A primary focus of this work is the issue of how polynomials are represented in memory. Polynomials have traditionally been stored in the *dense representation* as an array of coefficients, allowing efficient computations with polynomials such as

$$f = x^5 - 2x^4 + 8x^3 + 3x^2 - x + 9.$$

By contrast, the *sparse representation* of a polynomial is a list of nonzero coefficient-exponent tuples, a much more efficient representation when most of the coefficients are zero. This allows compact storage of a much larger set of polynomials, for instance

$$f = x^{5000} - 3x^{4483} - 10x^{2853} + 4x^{21}.$$

The types of polynomial computations we will examine fall into three general categories: basic arithmetic operations, algebraic problems, and inverse symbolic computations.

Basic arithmetic includes operations such as addition, subtraction, multiplication, and division. Optimal, linear-time algorithms for addition and subtraction of polynomials (in any representation) are easily derived. Division, modular reduction, and a great many other operations on densely-represented polynomials have been reduced to the cost of multiplication (sometimes with extra logarithmic factors). In fact, almost any nontrivial computation with polynomials in any representation uses multiplication as a subroutine. Hence multiplication emerges as the most crucial low-level arithmetic operation to consider.

At a slightly higher level, understanding the basic algebraic structure of polynomials is important for many applications in symbolic computation as well as cryptography. The fast algorithms that have been developed for polynomial factorization have been identified as some of the greatest successes of computer algebra. However, these algorithms are only efficient when the polynomials under consideration are represented densely. Many polynomials that arise in practice can only be feasibly stored using the sparse representation, but this more compact representation demands more sophisticated algorithms. Understanding the effects of sparsity on computational efficiency is fascinating theoretical work with important practical consequences.

The arithmetic and algebraic computations above involve manipulating and computing with symbolic representations of mathematical objects. Said mathematical objects are often not known explicitly, but rather given implicitly by a function or program that can be evaluated at any chosen point. Inverse symbolic problems involve computing a symbolic formula for a sampled function. Sparse polynomial interpolation in particular has a rich history as well as important applications in factorization and nonlinear system solving.

Kaltofen (2010a) has recently proposed a taxonomy of the most significant mid-level and high-performance computational tasks for exact mathematical computing, called the "seven dwarfs" of symbolic computation. In terms of this classification, the current work falls under the categories of the second and third dwarfs, exact polynomial algebra and inverse symbolic problems. We will also briefly touch on the fifth dwarf, hybrid symbolic-numeric computation.

1.1 Overview

The remaining sections of this chapter introduce the basic concepts and definitions that will be used for the remainder of the thesis. Our computational model has a carefully defined memory layout and counts machine word operations as well as arithmetic operations over an arbitrary algebraic domain. We show the implications of this model for the important basic operation of integer multiplication.

Chapter 2 follows by discussing the architecture and design of a high-performance C++ library containing all the algorithm implementations that we will present. This library is used to benchmark our algorithms against previous approaches and show how the theoretical improvements correspond to concrete practical gains.

The first algorithmic problem we turn to is also the most fundamental: multiplication of dense univariate polynomials. Chapter 3 examines the problem of computing the so-called Truncated Fourier Transform (TFT), which among other applications is used as a subroutine in fast polynomial multiplication. While the normal radix-2 Fast Fourier Transform (FFT) can be computed completely in-place, i.e., overwriting the input with the output, the TFT as originally developed by van der Hoeven (2004) requires linear extra space in the general case. Our new algorithm for an in-place TFT overcomes this shortcoming without sacrificing time cost.

The in-place TFT algorithm is used as a subroutine in one of two new algorithms for dense polynomial multiplication presented in Chapter 4. These new algorithms have the same time complexity as the two most commonly used sub-quadratic "fast" algorithms for multiplication, but improve on the linear amount of extra space previously required. Our first algorithm works over any ring, matches the $O(n^{1.59})$ time complexity of Karatsuba's algorithm, and only uses $O(\log n)$ extra space. The second algorithm works in any ring that admits radix-2 FFTs and matches the $O(n \log n)$ time cost of usual FFT-based multiplication over such rings, but uses only $O(1)$ extra storage space. Under our model of space cost, these are the first algorithms to achieve sub-quadratic time × space complexity for multiplication.

Next we turn to a broader view of polynomial multiplication, encompassing both dense and sparse polynomial multiplication. Two new approaches to this problem are presented in Chapter 5 which effectively provide a gradient between existing sparse and dense methods. Specifically, the algorithms adapt to the case of dense "chunks" of nonzero coefficients in an otherwise sparse polynomial, and to the opposite case of sparse polynomials in which the nonzero terms are mostly evenly spaced. These algorithms provide advantages over previous techniques by being more adaptive to the structure of the input polynomials, giving significant improvement in many cases while never being more than a constant factor more costly than any existing approach. We also show how to combine the two approaches into one algorithm that simultaneously achieves both benefits.

Chapter 6 examines an important algebraic problem for sparse polynomials. Given a sparse polynomial as input, we investigate how to determine whether it is a perfect power of another (unknown) polynomial, and if so, to compute this unknown polynomial root. Perfect powers are a special case of both polynomial factorization and decomposition, and fast algorithms for the problem are known when the input is given in the dense representation. We give the first polynomial-time algorithms for detecting *sparse* polynomial perfect powers, which are randomized of the Monte Carlo type, and work when the coefficients are rational numbers or elements of a finite field with sufficiently large characteristic. We then turn to the problem of actually computing the unknown polynomial root, and give two approaches to that problem. The first is output-sensitive and provably polynomial-time, while the second is much more efficient in practice but relies on a conjecture regarding the sparsity of intermediate results.

The third broad topic of this thesis is sparse polynomial interpolation. Chapter 7 gives a brief overview of existing algorithms for sparse interpolation over various domains. We show how to improve the polynomial-time complexity of the recent algorithm of Garg and Schost (2009) over sufficiently large finite fields. We also present a related algorithm for approximate sparse interpolation, where the coefficients are approximations to complex numbers, and show that our algorithm improves the numerical stability over existing approaches.

Building on these sparse interpolation algorithms, in Chapter 8 we extend their domain by presenting the first polynomial-time algorithm for the problem of sparsest-shift interpolation. This problem seeks a representation of the unknown sampled polynomial in the shifted power basis $1, (x-\alpha), (x-\alpha)^2, \ldots$, with α chosen to minimize the size of the representation, rather than the usual power basis $1, x, x^2, \ldots$. By first computing this sparsest shift α and then performing interpolation in that basis, we greatly expand the class of polynomials that can be efficiently interpolated.

1.2 Polynomials and representations

The issue of polynomial representations is of central importance to much of this thesis. We therefore pause to consider what a polynomial is and how one might be stored in a digital computer.

First, we define some standard notation that we will use throughout. Big-O notation, which has become standard in algorithmic analysis, is a convenient way to compare the asymptotic behavior of functions. Unfortunately, the notation itself can be somewhat misleading in certain situations, so we give a formal definition.

Given two k-variate functions $f, g : \mathbb{N}^k \to \mathbb{R}_{\geq 0}$, we say that $f(n_1, \ldots, n_k) \in O(g(n_1, \ldots, n_k))$ if there exist constants N and c such that, whenever $n_i \geq N$ for every $1 \leq i \leq k$, we have that $f(n_1, \ldots, n_k) \leq c\,g(n_1, \ldots, n_k)$. This is really defining a partial order on functions, but we can also think of $O(g(n_1, \ldots, n_k))$ as defining a set of all k-variate functions f which satisfy the above conditions, and our notation follows the latter convention.

In most cases, the k variables $n_1, \ldots, n_k$ will appear explicitly in $g(n_1, \ldots, n_k)$, but when they do not, or when other symbols appear that are *not* variables, these distinctions must unfortunately be inferred from the context. An important example is $O(1)$, which in the current context represents all k-variate functions bounded above by a constant.

The notations Ω, Θ, o, and ω can all be derived from the definition of O in the standard ways, and we will not list them here. Another common notation which we will use occasionally is *soft-O notation*: $f(n_1, \ldots, n_k) \in \tilde{O}(g(n_1, \ldots, n_k))$ if and only if

$$f(n_1, \ldots, n_k) \in O\left(g(n_1, \ldots, n_k) \cdot (\log g(n_1, \ldots, n_k))^c\right),$$

for some positive constant c. Informally, soft-O notation means ignoring logarithmic factors of polynomial terms. Note for example that the univariate function $\log n \cdot \operatorname{loglog} n$ is in $\tilde{O}(\log n)$ but *not* $\tilde{O}(1)$.

1.2.1 Functional representation

Consider a ring R (commutative, with identity) with binary operations $\times, +, -$, and a positive integer n. If only these three operations are allowed, say on a computer, then the set $\mathsf{R}[x_1, x_2, \ldots, x_n]$ of n-variate polynomials is exactly the set of all computable functions on n inputs. For instance, in a very simple view of modern digital computers operating directly (and exclusively) on bits, every possible computation corresponds to a multivariate polynomial over $\mathbb{F}_2$, the finite field with two elements.

With this fundamental connection between polynomials and computation in mind, we begin with a functional definition of the set $\mathsf{R}[x_1, x_2, \ldots, x_n]$ of n-variate polynomials, and will get to the more common algebraic definition later. Functionally, an n-variate polynomial f is simply a finite sequence of ring operations on a set of n indeterminates $x_1, x_2, \ldots, x_n$. This function can be represented by a so-called straight line program or, somewhat more expressively, by a division-free algebraic circuit.

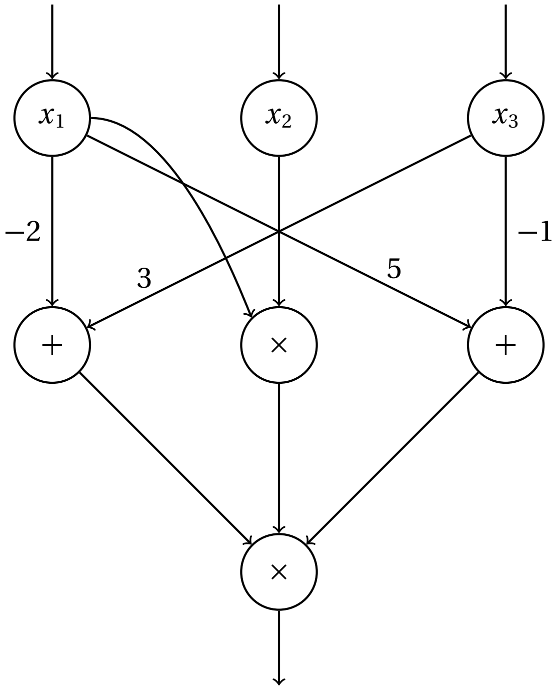


Figure 1.1: Algebraic circuit for $(-2x_1+3x_3)\cdot(x_1\cdot x_2)\cdot(5x_1-x_3)$

For our purposes, such a circuit will be a directed acyclic graph with n source nodes labeled x_1 through x_n and a single sink node for the output. Each node is labeled with $+$ or $\times$. Incoming edges to each $+$ node are also labeled with constant elements from R, and the value computed at that node is the R-linear combination of the values at its direct predecessors, times the values on the incoming edges. The value at each $\times$ node is simply the product of the values of its direct predecessor nodes. For instance, the algebraic circuit in Figure 1.1 represents the function

$$f(x_1,x_2,x_3)=(-2x_1+3x_3)\cdot(x_1\cdot x_2)\cdot(5x_1-x_3).$$

Computations on polynomials represented by algebraic circuits or straight-line programs have been studied extensively (Freeman, Imirzian, Kaltofen, and Yagati, 1988; Kaltofen, 1989; Kaltofen and Trager, 1990; Encarnación, 1997), and the main advantages of this representation are its potential for conciseness and connection to evaluation. Unfortunately, with the important exception of computing derivatives (Baur and Strassen, 1983), very few operations can be efficiently computed in polynomial-time in this representation. Even the most basic operation, determining whether two circuits represent the same polynomial, is notoriously difficult to solve, at least without randomization (Saxena, 2009). Because of these difficulties, many algorithms parameterize their cost based on some algebraic property of the polynomial(s) computed by the circuit. For instance, the best-known factorization algorithm by Kaltofen requires polynomial time in the degree, which could be exponentially larger than the circuit size.

1.2.2 Algebraic representations

A more typical way to represent polynomials is algebraically, as an R-linear combination of monomials in some chosen basis. The most common choice is the standard power basis, wherein each monomial is a product of the indeterminates, each raised to a nonnegative integer power. (Chapter 8 will consider an important alternate basis, namely the sparsest-shifted power basis.)

We now pause briefly to define some useful terminology. Each product of a monomial and a coefficient is called a *term*. We will typically focus on *nonzero terms*, i.e., terms with a nonzero coefficient, and we will implicitly assume that each monomial appears at most once. The highest exponent of a given indeterminate x_i in any nonzero term is called the *partial degree* in x_i. The greatest partial degree is called the *max degree*. The greatest sum of exponents in any nonzero term is called the *total degree*. The number of (distinct) nonzero terms is called the *sparsity* of the polynomial, and the monomials with nonzero coefficients make up its *support*.

For instance, the polynomial represented by the circuit in Figure 1.1 can be written algebraically, in the standard power basis, as $f = -3x_1x_2x_3^2 + 17x_1^2x_2x_3 - 10x_1^3x_2$. It has three nonzero terms, max degree 3, and total degree 4.

The question remains how to actually represent such expressions in a computer. We will briefly discuss the three most common representations, and refer to (Stoutemyer, 1984; Fateman, 2002) for a more in-depth comparison. The most straightforward choice is a multidimensional $d_1 \times d_2 \times \cdots \times d_n$ array of coefficients, where each d_i is greater than the partial degree in x_i. This we term the *dense* representation; its size is bounded by $O(d_1d_2\cdots d_n)$ ring elements, or more simply as $O(d^n)$, where d is the max degree. The dense representation typically admits very fast algorithms, but can be inefficient when the number of nonzero terms in the polynomial is very small compared to d^n.

More compact storage is afforded by the so-called *recursive dense* representation. The zero polynomial is represented as a null pointer; otherwise, terms are collected by the last variable and the representation is by a single length-d_n array of recursive dense polynomials in $n-1$ variables $x_1, x_2, \ldots, x_{n-1}$. This representation has proven to be practically useful in a number of situations, as it admits the use of dense univariate algorithms while being somewhat sensitive to sparsity. A major drawback is its high sensitivity to the variable ordering. Despite this, it is easy to see the size in the worst case is at most $O(dnt)$ ring elements, where t is the sparsity, i.e., number of nonzero terms. Since t could be at most $(d+1)^n$, this would seem to be potentially worse than the dense representation in cases where almost every term is nonzero, but this is not actually true; a better bound on the number of ring elements required for the recursive dense representation of $O(\min(d^n, dnt))$ is also easy to derive.

The most compact algebraic representation is the *sparse* representation (also called the *distributed sparse* representation). In this representation, a polynomial is stored as a list of coefficient-exponent tuples, wherein nonzero terms never need to be explicitly stored. This also corresponds exactly to the way we naturally write polynomials when doing mathematics, and for this reason it has become the standard and default representation in general-purpose computer algebra systems such as Maple, Mathematica, and Sage. This representation always uses exactly t ring elements for storage. However, in this case we also must take into account storage space for the potentially large exponents. This exponent storage is bounded above by $O(nt \log d)$ bits.

Observe that there is potentially an exponential-size gap between the size of each of these representations. For instance, assuming elements 0 and 1 in R are stored with a constant number of bits, the size of the polynomial $f = x_1^d x_2^d \cdots x_n^d$ in each of the dense, recursive dense, and sparse representations is $\Theta((d+1)^n)$, $\Theta((d+1)n)$, and $\Theta(n \log d)$ bits, respectively.

1.2.3 Algorithms

Most published algorithms for polynomial computation do not explicitly discuss what representation they have in mind, but nonetheless we can roughly categorize them based on those representations mentioned above.

Generally, algorithms in the literature that make no mention of sparsity and whose cost does not explicitly depend on t we will term *dense algorithms*. These algorithms are said to be polynomial-time if their cost is $(n^d)^{O(1)}$, i.e., polynomial in the size of the dense representation. Many important classical results in computer algebra fall into this class, including fast multiplication algorithms (Karatsuba and Ofman, 1963; Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991) and polynomial factorization (Berlekamp, 1967; Cantor and Zassenhaus, 1981).

Some algorithms for so-called sparse polynomial computation have complexity $(ndt)^{O(1)}$. Observe that this is polynomial-time in the size of the recursive dense representation. Richard Zippel's sparse interpolation algorithms (1979; 1990) are good examples of algorithms with this sort of cost. Algorithms for straight-line programs whose cost depends partially on the degree (e.g., Kaltofen and Trager, 1990) could also be considered of this type, as one can easily construct a small circuit for evaluating a sparse polynomial.

The most difficult algorithms to derive are those whose cost is polynomial in the size of the sparse representation defined above. These go by many names in the literature: "sparse" (e.g., Johnson, 1974), "lacunary" (e.g., Lenstra, 1999), and "supersparse" (e.g., Kaltofen and Koiran, 2005). To avoid any ambiguity, we will call any algorithm whose cost is polynomial in the size of the sparse representation — that is, polynomial in n, t, and $\log d$ — a *lacunary algorithm*. Observe that efficient lacunary algorithms automatically give efficient algorithms for all three algebraic representations mentioned above; the blow-up in complexity from a lacunary algorithm to a dense one is at most $O(n \log d)$. Lacunary algorithms also have advantages related to the univariate case, as we will discuss shortly. However, despite the ubiquity of the sparse representation in computer algebra systems, the development of lacunary algorithms has been much less rapid than dense algorithms; see Davenport and Carette (2009) for an overview of some of the specific algorithmic challenges that accompany the sparse representation.

Another class of algorithms we will occasionally consider are those whose cost depends even more strongly on one of the parameters, such as the sparsity t or the size of coefficients. These algorithms are polynomial-time in special cases, for instance when the number of terms or size of coefficients is constant, as in the recent examples by Filaseta, Granville, and Schinzel (2008) and Pébay, Rojas, and Thompson (2010) on computing gcds and extrema, respectively. However, observe that these are exponential-time algorithms no matter the choice of representation.

1.2.4 Univariate polynomials

Univariate polynomials are an important special case for consideration. Algorithms are of course easier to develop, describe, and implement in this case. But univariate polynomials also arise frequently in practice and have a strong relationship with multiple-precision integers. For instance, algorithms for fast multiplication of dense univariate polynomials have typically followed those for multiple-precision integers with the same complexity.

Observe that in the univariate case the recursive dense representation degenerates to the dense representation and hence becomes useless. So for univariate problems we need only consider two types of algebraic representations.

Fast algorithms for univariate polynomial computation can be applied to multivariate polynomials via the well-known Kronecker substitution:

Fact 1.1. *(Kronecker, 1882) Let* R *be a ring and* $n, d_1, d_2, \ldots, d_n \in \mathbb{N}$*, For any polynomial* $f \in \mathsf{R}[x]$ *with degree less than* $d_1 d_2 \cdots d_n$*, there exists a unique polynomial* $\hat{f} \in \mathsf{R}[x_1, x_2, \ldots, x_n]$*, with partial degrees less than* $d_1, d_2, \ldots, d_n$ *respectively, such that*

$$f(x) = \hat{f}(x, x^{d_1}, x^{d_1 d_2}, \ldots, x^{d_1 d_2 \cdots d_{n-1}}).$$

For many problems with multivariate polynomials, making this substitution (evaluating at high powers of a single variable) often allows the use of univariate algorithms. For instance, given the bivariate polynomial

$$\hat{f}(x, y) = 5 + 6x^2 y - 3xy^{10},$$

we could compute $\hat{f}^2$ by using the Kronecker substitution with bounds on the degrees in $\hat{f}^2$:

$$f(x) = \hat{f}(x, x^5) = 5 + 6x^7 - 3x^{51}.$$

We see that

$$f^2 = 25 + 60x^7 + 36x^{14} - 30x^{51} - 36x^{58} + 9x^{102},$$

and because we know that $\deg_x \hat{f}^2 < 5$, we can then apply Fact 1.1 to write down the coefficients of $\hat{f}^2$, by taking a quotient with remainder when each exponent in f^2 is divided by 5:

$$\hat{f}^2 = 25 + 60x^2 y + 36x^4 y^2 - 30xy^{10} - 36x^3 y^{11} + 9x^2 y^{20}.$$

More generally, the Kronecker substitution corresponds to writing each exponent of the univariate polynomial f in the mixed-radix representation with basis $(d_1, d_2, \ldots, d_n)$, and using the fact that every nonnegative integer less than the product $d_1 d_2 \cdots d_n$ has a unique representation as an n-tuple in the mixed-radix representation to that basis (see Knuth, 1981, §4.1).

A similar approach is used to convert a polynomial with coefficients in a finite field to an integer. For a polynomial with coefficients in $\mathbb{Z}/m\mathbb{Z}$, for example, we can write each coefficient as its unique integer residue in the range $0, 1, \ldots, m-1$, then convert this to an integer (in the binary representation) by evaluating the integer polynomial at a power of 2. If the power of 2 is large enough, then this is an invertible map, as used for example by Harvey (2009b) for the problem of multiplication.

This conversion between integers and univariate polynomials over finite fields can also be used in the other direction, as we will see in Section 1.4. However, observe that not every problem works as nicely as multiplication. For instance, to factor a bivariate polynomial $\hat{f} \in \mathsf{R}[x, y]$, we could again use the Kronecker substitution to write $f = \hat{f}(x, x^d)$ for $d > \deg_x \hat{f}$, and then factor the univariate polynomial $f \in \mathsf{R}[x]$. However, even though each factor of $\hat{f}$ will correspond to a factor of f, observe that the converse is not true, so that recovering the bivariate factors from the univariate ones is a non-trivial process. Reducing integer factorization to polynomial factorization is even more problematic because of carries.

The Kronecker map is especially useful when polynomials are stored in the sparse representation. If $f \in \mathsf{R}[x]$ and $\hat{f} \in \mathsf{R}[x_1, x_2, \ldots, x_n]$ corresponding to the Kronecker substitution as above, first notice that f and $\hat{f}$ will have the same number of nonzero terms, and in fact the same coefficients. Furthermore, if written in the sparse representation, these two polynomials have essentially the same size. For simplicity, assume the partial degrees are all less than d. Then each exponent in f is a single integer less than d^n, and each exponent in $\hat{f}$ is a vector of integers each less than d. Writing these in binary, their bit-length in both cases is $n \log_2 d$, and so the representation size in the sparse model is identical, at least asymptotically.

This relationship will be useful for our purposes, as it means that algorithms for univariate polynomials will remain polynomial-time under the Kronecker map. Roughly speaking, the exponential increase in the degree only corresponds to a polynomial increase in the logarithm of the degree, which corresponds to the size of the sparse representation. Hence algorithms for univariate polynomials with very few terms and very high degree have an important application. We will consider exactly this type of algorithm at various points throughout this thesis.

1.3 Computational model

It is always important to carefully define the computational framework that will be used to describe and analyse algorithms. In our case, this is more than an academic exercise, as the choice of model will be crucially important for understanding our results in the first few chapters.

1.3.1 Background

Countless descriptions of different abstract machines have been proposed over the last seventy years as general models of mathematical computation. These models have a wide variety of motivations and goals; some are defined according to the functions they can compute, some according to the structure of the computation, and some according to properties of a class of actual digital computers. In the present work, we will prefer this final category for its (hopeful) connection between theoretical results and practical performance. However, we will discuss some other models as well for comparison.

Turing (1937) developed the model of the first abstract computing machine, the so-called *Turing machine*, which we still use today as the basis for computability and basic complexity classes such as **P** and **NP**. The greatest strength of this model is its simplicity and computational equivalence to other fundamental models such as Church's lambda calculus and pushdown automata.

The slight refinement of *multi-tape Turing machines* are of particular interest here. The simplest such machine has three tapes, a read-only input tape, a write-only output tape, and a single working tape. More sophisticated models such as the seven-tape Turing machine of Schönhage, Grotefeld, and Vetter (1994) have shown some evidence of practicality for mathematical computations, despite bearing little resemblance to actual machines.

However, Turing machines are known to over-estimate the cost of certain operations on modern digital computers, the difference owing primarily to the way memory is accessed. In Turing machine models, each tape has a single head for reading and/or writing which is only allowed to move one position at each step of the computation. By contrast, the structure of internal memory in any modern computer allows random access to any location with essentially the same cost.

This motivated the definition of the *random access machine* (or RAM) by Cook and Reckhow (1973). Their model looks similar to a 3-tape Turing machine, except that the work memory is not a sequential-access tape but rather an array that allows random access and indirect addressing. Another important difference is that every memory location in input, output, or working space can hold an integer of unbounded size, as opposed to the finite alphabet restriction in Turing machines. In addition to the usual operations such as LOAD, STORE, and BRANCH, the instruction set of a RAM also contains operations for basic arithmetic such as addition and multiplication of integers.

While the RAM model has seen much success and seems to be very commonly used in algorithm design and analysis, it still has significant differences from actual digital computers. The most obvious is the RAM's ability to manipulate and store unbounded-length integers with unit cost. This is somewhat mitigated by the *log-cost RAM*, in which the cost of an operation on integers with values less than n is charged according to their bit-length: $O(\log n)$ for all operations except MUL and QUO, which cost $O(\log^2 n)$. A slightly more refined approach is presented by Bach and Shallit (1996, §3.6), where an interesting storage scheme is also mentioned which would allow for sub-linear time algorithms. However, as pointed out for instance by Grandjean and Robson (1991), these models still underestimate the cost of certain kinds of memory accesses and (at least for the log-cost RAM) define the cost of arithmetic too bluntly.

Motivated by these shortcomings is the primary model we will rely on in this thesis, the *Random Access Computer* (or RAC) of Angluin and Valiant (1979). This model bears some similarity to pointer machines (Kolmogorov and Uspenskiĭ, 1958; Schönhage, 1980; Schönhage et al., 1994) in that each memory location may hold a pointer to another memory location. Reconciling this with the RAM model, each "pointer" is actually an integer of bounded size, and this "word size" is dependent upon the size of the input. The model supports modular arithmetic on word-size integers as well as indirect addressing. This seems to be most closely aligned to machine instructions in modern (sequential) computing architectures, and the model we propose in the next subsection is essentially an elaboration on the RAC.

Having described the evolution and motivation of our model of choice, we briefly mention some other computational models that will be of interest. First, on the practical side, a shortcoming still of the RAC is that every random memory access is assumed to have the same cost. On any modern computer, the multi-level memory hierarchy of registers, cache, main memory, and disk grossly violates this assumption. Especially in algorithms with close to linear complexity, the dominant cost often becomes that of memory access (these are called *memory-bounded* operations). Models which seeks to address this are the I/O model or "DAM model" of Aggarwal and Vitter (1988) and its cache-oblivious extension by Frigo, Leiserson, Prokop, and Ramachandran (1999).

Backing away from practicality, most of the models mentioned so far are notoriously resistant to any lower bounds for interesting problems. Simpler models such as circuits (equivalently, straight-line programs), described in the previous section, are more useful for these purposes, while still being sufficiently general to describe many algorithms in practice. A special type of circuit of note here is the *bounded coefficients* model, in which every constant appearing on an edge in the circuit must be of constant size (Chazelle, 1998).

The most significant shortcoming of circuits is that they do not allow branching; this is overcome by *branching programs*, an extension of binary decision trees to general domains (Borodin and Cook, 1980). In such models, each node represents a "state", of the computation, which proceeds according to the values computed. The size complexity in such models is defined as the number of bits required to store this state, which is the logarithm of the number of nodes.

1.3.2 In-Memory Machine

As mentioned before, the model we will use in this thesis is the Random Access Computer (RAC) of Angluin and Valiant (1979). The primary motivation for this model was to get a closer connection between practical performance and theoretical complexity for *low-level* computations. While high-level computations or entire programs may often read a large input from a file (similar to the read-once input tape in a RAM), low-level computations are often subroutines within a program, and therefore their input, as well as space for the output, presumably resides in main (random-access) memory at the beginning of the computation.

In the RAC model, this in-memory property is handled by making a special additional instruction to those of a standard RAM that loads the entire input into working memory in a single step. Hence the memory structure of a RAC looks identical to that of a RAM, with the primary difference being that the integers stored in each memory location have bit-length — that is, the size of machine words — bounded (in some way) by the size of the input.

The model we define here gives a more detailed memory layout that will allow a richer discussion of space complexity in the algorithms we develop. Our variation will also give greater ease in composing algorithms, i.e., calling one algorithm as a subroutine of another. It is computationally equivalent to the RAC, but for clarity, we will use the distinct name of *In-Memory Machine*, or IMM for short.

Storage in the IMM is divided into four parts: constants (C), input (I), temporary working space (T), and output (O). Each of these is represented by a sequential array, which we will refer to respectively as M_C, M_I, M_T, and M_O. All four arrays allow random read access at unit cost, and the work space and output also allow for random write access at unit cost.

Instruction	Description
COPY(A, i, B, j)	Copy the value of $M_A[M_T[i]]$ to $M_B[M_T[j]]$. A must be one of C, I, T, or O. B must be one of T or O.
BRANCH(i, L)	Go to instruction labeled L iff $M_T[i] = 0$.
ADD(i, j, k)	$M_T[i] \leftarrow M_T[j] + M_T[k] \mod 2^w$
SUB(i, j, k)	$M_T[i] \leftarrow M_T[j] - M_T[k] \mod 2^w$
MUL(i, j, k)	$M_T[i] \leftarrow M_T[j] \cdot M_T[k] \mod 2^w$
QUO(i, j, k)	$M_T[i] \leftarrow \left\lfloor \frac{M_T[j]}{M_T[k]} \right\rfloor$
HALT	Stop execution

Table 1.1: Instruction set for IMM model

The instruction set for the IMM is given in Table 1.1. The parameter w gives the word size, so that $2^w - 1$ is the largest integer that can be stored in a single word of memory. Observe that only the COPY instruction involves indirect addressing. We must also specify that $M_T[i] = 0$ initially for any $i \geq 0$. For any instruction COPY(A, i, B, j) with $B = O$, i.e., a copy to the output space, we also require that $M_T[j]$ is less than the size of the output. More precisely, using the notation of the following paragraph, $0 \leq M_T[j] < \max(\#O, \#O')$.

A *problem* for IMM computation is described by a set $\mathscr{S} \subseteq (\mathbb{Z}^*)^3$ of tuples (I, O, O') indicating the initial configurations of the input and output space, and the corresponding final configuration of the output space. Most commonly, I will simply contain the input and O will be empty, but the model allows for the output space to be wholly or partly initialized, for reasons we will see. As O and O' will be stored in the same part of memory, the size of the output space is defined to be $\max(\#O, \#O')$, and this also bounds the size of this part of memory at any intermediate step in the computation, as detailed above. For the problem to be well-defined, we of course require that any initial input and output configuration appears at most once in $\mathscr{S}$. That is, for any $I, O, O_1', O_2' \in \mathbb{Z}^*$, if $(I, O, O_1') \in \mathscr{S}$ and $(I, O, O_2') \in \mathscr{S}$, then $O_1' = O_2'$.

The *size* n of a given instance $(I, O, O') \in \mathscr{S}$ is defined as the size of the input and output, $n = \#I + \max(\#O, \#O')$. Every integer stored in memory must fit in a single word, but as in the RAC model, this word size varies according to the size of the instance, as we will discuss below.

An *algorithm* for an IMM problem consists of three components: a labeled list of instructions from the set in Table 1.1, a function $\mathscr{W} : \mathbb{N} \to \mathbb{N}$ to define the word size, and a function $\mathscr{C} : \mathbb{N} \to \mathbb{Z}^*$ to define the values stored in the array C of constants.

The word-size function $\mathscr{W}(n)$ gives the number of bits w of a single word in memory for any instance of size n. Consider the restrictions on the behaviour of $\mathscr{W}(n)$. First, machine words must be large enough to hold every integer that appears in the input or output. For a given problem $\mathscr{S}$, define $\mathscr{M}_{\mathscr{S}}(n)$ to be the largest integer appearing in any valid size-n instance in $\mathscr{S}$. Then we must have $\mathscr{W}(n) \geq \log_2 \mathscr{M}_{\mathscr{S}}(n)$.

Furthermore, machine words must be large enough to address memory. To address the input and output, this immediately implies that we must have $\mathscr{W}(n) \geq \log_2 n$. But the algorithm may use more than n cells of memory in temporary storage. This is why the word-size function $\mathscr{W}(n)$ must be defined by the algorithm and not by the problem. However, we should be careful in allowing the algorithm to set the word size, as setting $\mathscr{W}(n)$ to be very large could make many computations trivial.

This “cheat” is avoided by requiring that algorithms may only increase the word-size by a constant factor. Specifically, we require

$$\mathscr{W}(n) \in O\left(\log_2 n + \log_2 \mathscr{M}_{\mathscr{S}}(n)\right).$$

A consequence is that the IMM model is sufficiently rich to describe the computation of any function in **PSPACE**.

There is one more comment to be made on the word size. The bound $\mathscr{M}_{\mathscr{S}}(n)$ comes from the problem itself, but for some problems this is also flexible in a certain sense. For instance, consider algorithms whose input or output contains arbitrary-precision integers. Encoding every integer into a single word clearly violates the principle of the model, and eliminates from discussion any algorithm for multiple-precision arithmetic. But how should a long integer be encoded into multiple words?

To address this issue, we say a problem is *scalable* if the size of integers in a length-n instance are proportional to $\log_2 n$. More precisely, a problem $\mathscr{S}$ is scalable if $\mathscr{M}_{\mathscr{S}}(n) \in O(\log n)$. Roughly speaking, this means that if the problem can be encoded into any actual computer with fixed machine precision, then the word-size and word operations in any algorithm for that problem correspond to a constant number of words and word operations on that real computer.

The third component of an algorithm solving a problem on an IMM is the constants computation function $\mathscr{C}(n)$. This gives the values stored in the array of constants C for any instance size n. To avoid gratuitous cheats such as computing all possible answers as constants, we also require that the number of constants, $\#C$, be fixed for the algorithm independently of the instance size. That is, $\#C$ does not vary according to the instance size, although the *values* in C may. The $\mathscr{C}(n)$ function must be computable, and in most practical situations will be trivial.

What it means to say that a given IMM algorithm correctly solves a given IMM problem should be clear. We now turn to the cost analysis of an algorithm A. The time complexity of A, written $T(n)$, is the maximum over all instances of size n in $\mathscr{S}$ of the number of steps taken by A to produce O' in the output space and halt. Importantly, this does *not* include the time to compute $\mathscr{W}(n)$ and $\mathscr{C}(n)$. In the case that $\mathscr{W}(n)$ and $\mathscr{C}(n)$ can be computed in $O(T(n))$ time by an IMM with word size exactly $\log_2 n$ and absolutely fixed constants, we say that the algorithm A is *universal*.

The *space complexity* of algorithm A, written $S(n)$, is based only on the size of M_T, the temporary working space. It is defined as the maximum i such that $M_T[i]$ is accessed in any instruction during the execution of any valid instance of size n. Note that this differs crucially from most notions of space complexity, in models where the input and output exist in read-once and write-once tapes, respectively. We will see how the ability to read from *and* write to the output space breaks at least some lower bounds.

Some portions of this thesis examine *in-place* algorithms. Intuitively, this means that the output is overwritten with the input. More precisely, we say a problem given by $\mathscr{S}$ is *in-output* if I is empty for every $(I, O, O') \in \mathscr{S}$. That is, the data used as input for the algorithm is stored in the initial configuration of the read-write output space. An algorithm for such a problem is said to be *in-place* if it uses only $O(1)$ temporary space in M_T. To our knowledge, ours is the first abstract computational model that allows a formal description of such problems and algorithms.

The reader is referred to Angluin and Valiant (1979) for connections between time complexity in the IMM (equivalently RAC) model and the more common RAM model. In particular, an algorithm with time complexity $T(n)$ on a log-cost RAM can be simulated by an IMM with the same cost, and an IMM algorithm with time complexity $T(n)$ can be simulated on a log-cost RAM with at most $O(T(n)\log^2 n + n\log n)$ operations.

Finally, some algorithms will make use of randomization to improve performance. To this end, define a *randomized IMM* to be equivalent to the normal IMM with an additional instruction RAND(i) that chooses an integer uniformly and randomly between 0 and $2^{\mathscr{W}(n)} - 1$ and stores its value in $M_T[i]$.

1.3.3 Storage specification for elements in common rings

The subject of this thesis is computations with polynomials in $\mathsf{R}[x_1, \ldots, x_n]$, where the sorts of domains we have in mind for the ring R are the integers $\mathbb{Z}$, rationals $\mathbb{Q}$, finite fields $\mathbb{F}_q$ where q is a prime power, and floating-point approximations to real and complex numbers in $\mathbb{R}$ and $\mathbb{C}$. For completeness, we discuss briefly how elements in these rings can be stored in the memory of an IMM.

A natural number $a \in \mathbb{N}$ is stored in the 2^w-adic representation as a list $(a_0, a_1, \ldots, a_{n-1})$ with $0 \le a_i < 2^w$ for all i, and $a = a_0 + a_1 2^w + a_2 2^{2w} + \cdots + a_{n-1} 2^{(n-1)w}$. So each a_i fits in a single word of memory, and the entire representation of a may be stored either contiguously or (with twice as much space) in a linked list. To extend this to all integers, we add an additional word of storage for the sign. In any case, the size of the representation in memory is $O(\frac{1}{w}\log a)$ words of storage.

In the case of scalable problems as defined in the previous subsection, we conclude that $O(n)$ words in IMM (or RAC) memory can be used to represent any $(n\log_2 n)$-bit integer, and conversely a single n-bit integer can always be represented in $O(n/(\log n))$ words. (To be more pedantic, we might additionally specify that the representation of a is preceded by a single word indicating the length of the representation.)

A rational number in $\mathbb{Q}$ can always be written n/d with $n,d \in \mathbb{Z}$ relatively prime and $d > 0$. Such a number is simply represented as the pair (n,d).

Elements in a modular ring $\mathbb{Z}/m\mathbb{Z}$ are stored as integers in the range $\{0,1,\ldots,m-1\}$. This handles the case of finite fields $\mathbb{F}_p$ with prime order. For extension fields of size p^e, we will work in the isomorphic field $\mathbb{F}_p/\langle\Gamma\rangle$ for $\Gamma \in \mathbb{F}_p[x]$ irreducible of degree e. Then we represent elements in $\mathbb{F}_{p^e}$ by polynomials in $\mathbb{F}_p[x]$ with degree less than e using the dense representation, as an array of e elements from $\mathbb{F}_p$.

Consider a real number $\alpha \in \mathbb{R}$ in the range $(0,1)$, and any chosen small $\epsilon > 0$. To store an ϵ-approximate representation of α, we use an integer a stored in $k = \left\lceil \frac{1}{w}\log_2\frac{1}{\epsilon}\right\rceil$ words, satisfying $\left|\alpha - 2^{-kw}a\right| < \epsilon$. This can be extended to any real number β by adding an integer exponent b such that $\left|\beta - 2^{b-kw}a\right| < \epsilon \cdot 2^b$. An approximation to a complex number $\gamma \in \mathbb{C}$ is stored as a pair of approximate real numbers, representing the rectangular coordinates for γ.

For any of these rings, and for an element $u \in \mathsf{R}$ in the ring, we write size(u) for the number of machine words required to represent u (where the parameter w for the word-size is understood). So for instance if $u \in \mathbb{N}$ is a nonnegative integer, size$(u) = \lceil \log_w(u)\rceil$.

Observe that addition in any of these rings is easily accomplished in linear time in the size of their representation. We will discuss the cost of multiplication in the next subsection. An important subroutine is modular multiplication with word-sized operands: given $a,b,c \in \mathbb{N}$ with $0 \le a,b,c < 2^w$, computing $r \in \mathbb{N}$ with $0 \le r < c$ such that $a \cdot b \equiv r \bmod c$. First the multiplication $a \cdot b$ is performed to double-word precision, by splitting each a,b in half via a division with remainder by $2^{\lfloor w/2\rfloor}$, then performing four word-size multiplications and some additions. The division with remainder by c can then be accomplished using similar techniques. In summary, the modular multiplication subroutine can be performed in a constant number of word operations in the IMM.

1.3.4 Algebraic IMM

The results in the previous subsection handle storage and manipulation of algebraic objects of various types within the usual IMM model. However, it is often convenient to describe and analyse an algorithm independently of the particular domain of computation. For instance, the same algorithm might work over an arbitrary field, whether it be a prime field, an extension field, the rational numbers, or any number of other fields. To present the algorithm concisely, it would be useful to perform basic arithmetic in an arbitrary algebraic domain.

The concept of an algebraic RAM was developed to handle mathematical computations involving integers as well as elements from an arbitrary domain. The idea is a usual RAM with a single program but duplicated storage: two input tapes, two output tapes, and two memory banks. (Often the input and output are only on one side.) The *arithmetic* side of the RAM is in the usual model and computes with arbitrary-length integers, while the *algebraic* side computes with elements from the chosen algebraic domain. This seems to be the dominant model (whether or not explicitly stated) in algorithms for symbolic computation.

Along these lines, we extend our concept of IMM to an algebraic IMM. The four memory banks are duplicated on the arithmetic and algebraic sides, albeit not necessarily of the same size. All instructions can be performed on either side of the IMM, but the two may never be mixed. That is, it is impossible to (explicitly) copy values between the algebraic and arithmetic sides. Furthermore, the specific arithmetic operations for the algebraic side might differ from the ADD, SUB, MUL, and QUO of the arithmetic side, depending on the domain and the specific problem.

An algorithm for an algebraic IMM consists of a single list of instructions, the two functions $\mathscr{W}(n)$ and $\mathscr{C}(n)$, and a third function $\mathscr{A}(n)$ to generate the constants on the algebraic side. This function may produce different values for different domains *and* different instance sizes n, but the size of the constant array on the algebraic side must be fixed for any valid domain and any instance size.

A universal algorithm for an algebraic IMM is similar to before, with the additional requirement that $\mathscr{A}(n)$ can be computed in the same time and space complexity.

We could define an additional instruction for choosing random algebraic elements, but this is problematic to define in a sensible generic way. Instead, a randomized algebraic algorithm can define a subset of algebraic elements, stored in any part of the memory (input, constants, output, or temporary working space), and then use a randomly-chosen integer to index into the set. For all applications we are aware of, this will be sufficient to guarantee performance, and also sufficiently general to be applicable in any domain.

Now consider the problem of representing a polynomial $f \in \mathsf{R}[x_1, \ldots, x_n]$ in an algebraic IMM. In the dense representation, the entire polynomial can be stored in the algebraic side, perhaps with some word-sized integers for the size on the arithmetic side. The recursive dense representation is actually the most intricate of the three, and will be stored mostly on the arithmetic side, with pointers to ring elements on the algebraic side at the bottom level of recursion only. Finally, the sparse representation of a t-sparse polynomial will consist of a length-t array of nonzero coefficients on the algebraic side, coupled with a length-nt array of exponents on the arithmetic side.

1.4 Warm-up: $O(n \log n)$ multiplication

We now show the implications of the IMM model with an example: multiplication of multiple-precision integers in $\mathbb{Z}$. This is undoubtedly one of the most well-studied problems in mathematical computation, and dense multiplication algorithms also hold a central importance in this thesis. It is therefore not only instructive but prudent for us to examine the problem in our chosen model of computation. Furthermore, some of the number-theoretic tools we use here will come up again later in this thesis.

1.4.1 Summary of previous algorithms

The traditional model of study for integer multiplication algorithms is that of bit complexity. So consider the problem of multiplying two n-bit integers, that is, $a, b \in \mathbb{N}$ such that $0 \leq a, b < 2^n$.

The naïve algorithm for multiplication uses $O(n^2)$ word operations. Karatsuba's algorithm (Karatsuba and Ofman, 1963) uses a divide-and-conquer approach to improve the complexity to $O(n^{\log_2 3})$, or $O(n^{1.59})$. A family of divide-and-conquer algorithms due to Toom (1963) and Cook (1966) improves this to $O(n^{1+\epsilon})$, for any positive constant ϵ. Schönhage and Strassen (1971) make use of the Fast Fourier Transform algorithm to improve the bit complexity to $O(n \log n \operatorname{loglog} n)$. Recently, this technique has been carefully refined by Fürer (2007) to achieve $O(n \log n 2^{\log^* n})$ bit complexity for multiplication [1].

Knuth (1981, §4.3.3) discusses most of these algorithms from both a practical and a theoretical viewpoint. Of note here is an idea presented by Knuth but attributed to Schönhage to achieve $O(n \log n)$ complexity in a log-cost RAM for multiplication. The idea is to use complex number arithmetic and use the so-called Four Russians trick (Arlazarov, Dinic, Kronrod, and Faradžev, 1970) to precompute all products under a certain size. In the storage modification machine (SMM) model, Schönhage (1980, §6) shows how to achieve $O(n)$ time for n-bit integer multiplication, which is also similar to our result here. (By way of comparison, Schönhage explicitly states that the goal of his model is not to correspond to any "physical realization", but rather to develop a flexible and general model for its own sake. Our IMM model has the opposite motivation.)

1.4.2 Integer multiplication on an IMM

The algorithm presented here achieves a similar result to Schönhage's $O(n \log n)$ algorithm for a RAM, but without the need for extensive precomputation, and using only modular arithmetic. The idea is similar to the "Three primes FFT integer multiplication" algorithm of von zur Gathen and Gerhard (2003, §8.3), extended to arbitrary word sizes. We now proceed to describe the problem and our algorithm for the IMM model.

As is standard, for simplicity we restrict our attention to the case that both multiplication operands are of roughly the same size. A valid instance in $\mathscr{S}$ will consist of input I containing two integers a and b, each n words long, output O initially empty, and O' — the desired output — consisting of the product ab written in $2n$ words of memory. The total instance size is therefore $4n$, and this completes the formal description of the problem.

Write $m = \log_2 \max(4n, \mathscr{M}_{\mathscr{S}}(n))$, the lower bound on the word size implied by the problem definition. We assume the input integers a and b are written in the 2^m-adic representation as

$$a = a_0 + a_1 \cdot 2^m + a_2 \cdot 2^{2m} + \cdots + a_{n-1} \cdot 2^{(n-1)m}$$
$$b = b_0 + b_1 \cdot 2^m + b_2 \cdot 2^{2m} + \cdots + b_{n-1} \cdot 2^{(n-1)m}.$$

Now define the polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \in \mathbb{Z}[x]$ and $B(x) \in \mathbb{Z}[x]$ similarly. Our approach is to compute the product $A(x) \cdot B(x) = C(x) \in \mathbb{Z}[x]$, then evaluate $C(2^m)$ to compute the final result and write it again in the 2^m-adic representation.

[1] The iterated logarithm, denoted $\log^* n$, is the extremely slowly-growing function defined as the number of times logarithm must be taken to reduce n to 1. So, for instance, if $n > 1$, $\log^* n = \log^*(\log n) + 1$.

Let k be the least power of 2 greater than $2n$, i.e., $k = 2^{\lceil \log_2 n \rceil + 1}$. Our algorithm works by first choosing a set of primes $\mathscr{P}$ such that for each $p \in \mathscr{P}$, $(p-1)$ is divisible by k, and the product of the primes in $\mathscr{P}$ is at least 2^{3m}.

From the fact that $\deg A, \deg B < n$ and every coefficient of A and B is at less than 2^m, we can see that every coefficient in C is less than $n \cdot 2^{2m}$, which is less than 2^{3m} from the definition of m. Therefore computing the coefficients of $C \bmod p$ for each $p \in \mathscr{P}$, followed by Chinese remaindering, will give the actual integer coefficients of C.

Furthermore, since the multiplicative group of integers modulo each prime $p \in \mathscr{P}$ is divisible by a power of 2 greater than $\deg C$, the coefficients of $C \bmod p$ can be efficiently computed using the Fast Fourier Transform, using $O(n \log n)$ operations in $\mathbb{F}_p$, assuming a k'th primitive root of unity modulo p (which must exist) is known in advance. Denote such a root of unity ω_p for each $p \in \mathscr{P}$.

The crucial question that remains is how large each p must be to satisfy these conditions. For this, we turn to analytic number theory, and in particular Linnik's theorem (which we somewhat simplify for our particular purposes):

Fact 1.2. *(Linnik, 1944) There exist absolute constants q_L, c_L, L such that, for all $q \geq q_L$, the least prime p such that $p \equiv 1 \bmod q$ satisfies $p \leq c_L q^L$.*

Unfortunately, despite the large body of work devoted to finding concrete values for the exponent L, it appears that no proofs give explicit values for both q_L and c_L. Although the exponent L has been improved recently by Xylouris (2009), the most useful result for us is from Heath-Brown (1992), where it is proven that the inequality above holds for $q_L = 1$, $L = 5.5$, and c_L is an effectively computable absolute constant.

From this, we have the following.

Lemma 1.3. *There exists an absolute constant c such that, for any $k, m \in \mathbb{N}$ as above, there exists a set of at most three primes $\mathscr{P}$ with $\prod_{p \in \mathscr{P}} p \geq 2^{3m}$ and, for each $p \in \mathscr{P}$,*

1. *k divides $(p-1)$*
2. *$p \leq c 2^{16.5m}$.*

Proof. Let c_L be the (effectively computable) constant from the version of Linnik's theorem by Heath-Brown (1992).

Let p_1 be the least prime congruent to 1 modulo 2^m. We have that $p_1 < c_L(2^m)^{5.5} = c_L \cdot 2^{5.5m}$.

If $p_1 \geq 2^{3m}$, then we simply set $\mathscr{P} = \{p_1\}$. Otherwise, let p_2 be the least prime congruent to 1 modulo $2^{\lceil \log_2 p_1 \rceil}$. Since p_2 has a divisor greater than p_1, clearly $p_2 \neq p_1$, and furthermore we see that 2^m divides $(p_2 - 1)$. Finally, $2^{\lceil \log_2 p_1 \rceil} < 2^{3m+1}$, and therefore $p_2 \leq c_L(2^{3m+1})^{5.5} < 64 c_L \cdot 2^{16.5}$.

If $p_1 p_2 \geq 2^{3m}$, then set $\mathscr{P} = \{p_1, p_2\}$. Otherwise, let p_3 be the least prime congruent to 1 modulo $2^{\lceil \log_2 p_2 \rceil}$ as previously and observe that $p_3 < 64 c_L \cdot 2^{16.5}$ as well.

At this point, since each of p_1, p_2, p_3 is greater than 2^m, their product is at least 2^{3m}. So set $\mathscr{P} = \{p_1, p_2, p_3\}$ in this case.

From the definition of m, $4n < 2^m$. Recalling also that k is the least power of two greater than $2n$, it must be the case that $k \mid 2^m$, and therefore k divides each $p_i - 1$. Letting $c = 64c_L$, we get the stated result. □

For our IMM integer multiplication algorithm, then, we set $\mathscr{W}(n)$ — the word size for instance size n — to be $\mathscr{W}(n) = \lceil \log_2 c \rceil + 16.5m$. This means that each $p \in \mathscr{P}$ will fit into a single word in memory. Clearly $\mathscr{W}(n)$ is bounded by

$$\mathscr{W}(n) \in O(\log n + \log \mathscr{M}_{\mathscr{S}}(n)) = O(m),$$

so this is allowable in the IMM model.

The constants defined by $\mathscr{C}(n)$ for our algorithm will consist of the (at most three) primes in $\mathscr{P}$, along with the k'th primitive roots of unity modulo each prime in $\mathscr{P}$. For any n, this consists of at most six words of memory, so the constants are also valid for the IMM model.

This completes the description of our algorithm for multiplication in the IMM model. Because arithmetic modulo a word-sized integer can be performed in a constant number of steps, the cost of arithmetic operations in $\mathbb{F}_p$ is constant for each $p \in \mathscr{P}$. Therefore the total cost of the algorithm is simply $O(n \log n)$ word operations, the cost of the modular FFT computations.

This result is summarized in Theorem 1.4 below. Unfortunately, as the constant c_L is not known explicitly, we have only shown the *existence* of an algorithm. Some more careful steps and repeated doubling in the computation of $\mathscr{W}(n)$ and $\mathscr{C}(n)$ could avoid this issue, but we will not go into those details here.

Theorem 1.4. *There exists an IMM algorithm for multiplying n-word integers that runs in time $O(n \log n)$.*

Another unfortunate fact is that the algorithm is not universal, i.e., the word-size and constants cannot be computed in the same time as the algorithm itself. This is because of the cost of searching for each prime $p \in \mathscr{P}$, which could be avoided by either (1) using randomization to randomly choose primes in the progression, or (2) assuming a more realistic bound on the least prime in an arithmetic progression. The latter possibility will be discussed at length in Chapter 8.

1.4.3 Implications

Observe that the problem as defined for integer multiplication in the previous subsection is not *scalable*. That is, the size m of words in the input could be much larger than $\log_2 n$. This actually means that our $O(n \log n)$ algorithm is actually a stronger result, as it applies no matter what the word size is.

To compare the result of Theorem 1.4 against previous work in the bit complexity model, we should restrict the problem to be scalable. The word size for a size-n input is then restricted to $O(\log n)$, meaning that an n-bit integer requires exactly $\Theta(n/(\log n))$ words of storage. Therefore the algorithm described can be used to multiply n-bit integers using $O(n)$ word operations in the IMM model. This matches the algorithm of Schönhage (1980) in the SMM model but (like that result) can be misleading, as it counts the input in bits but the cost in word operations.

Simulating the algorithm described above on a log-cost RAM is a fairer comparison, and gives bit complexity $O(n\log^2 n)$ for n-bit integer multiplication. This could be improved at least to some extent with a more careful simulation, but we will not attempt that exercise.

The fast integer multiplication algorithm can be applied to other rings represented in words in the IMM, as well as to polynomials with coefficients in such rings and stored in the dense representation, using the Kronecker substitution.

For instance, consider the multiplication of two univariate integer polynomials $A, B \in \mathbb{Z}[x]$ with degrees less than d and all coefficients stored in at most k words. Observe that the instance size n is $\Theta(dk)$. To compute $A \cdot B \in \mathbb{Z}[x]$, simply write down the integers $a = A(2^{(2k+1)w})$ and $b = B(2^{(2k+1)w})$, and multiply them. Since each integer is stored in $O(dk) = O(n)$ words, the cost of this multiplication is $O(n\log n)$. Furthermore, each coefficient in $A \cdot B$ is at most $d \cdot (2^{wk})^2 \le 2^{(2k+1)w}$, so the coefficients of $A \cdot B$ can simply be read off from the integer product ab.

This same idea can be extended to any $\mathsf{R}[x_1,\ldots,x_m]$, with R any of the rings mentioned in subsection 1.3.3 to perform dense polynomial multiplication in time $O(n\log n)$, where n is the size of a single instance in the IMM. However, importantly, we do *not* have a $O(n\log n)$ algorithm for multiplication of dense polynomials over an arbitrary ring R in the algebraic IMM model. This is because there is no way to encode the elements of an arbitrary ring into integers. In this case, the best result is still that of Cantor and Kaltofen (1991), giving an algorithm for polynomial multiplication in an algebraic IMM using $O(n\log n\,\mathrm{loglog}\,n)$ ring operations.

In summary, we see that the IMM model can allow very slight improvements over the best known bit complexity results. However, these differences are quite minor, and furthermore we would argue that the IMM model gives a more accurate measure of the actual cost on actual physical machines. In the remainder, the IMM model will form the basis of our discussions and analysis, although the algorithms will not be presented so formally as they have been here. Furthermore, most of our results will apply equally well to more common models such as (algebraic) RAMs.

I must Create a System, or be enslav'd by another Man's.
I will not Reason & Compare; my business is to Create.

—William Blake, *Jerusalem* (1804)

Chapter 2

Algorithm implementations in a software library

Owing to their central importance in mathematical computing, software for polynomial computations exists in numerous programs and libraries that have been developed both in academia and in industry. We will examine this existing software and show that, surprisingly, no open-source high-performance library focusing on polynomial computation — including multivariate and sparse polynomials — is currently available.

We then discuss our proposed solution, a new software library for computations with polynomials, broadly defined, which we call the MVP library. We give an overview of the architecture and aims of the software, and discuss how some low-level operations have been implemented. Although the library is still in its infancy, we demonstrate that it is already competitive in speed with other software, while being at least general enough to hold all the algorithms that will be presented in this thesis.

The author extends his thanks to Clément Pernet for many fruitful discussions, and particular those which started the idea of this software project.

2.1 Existing software

Computations with polynomials are useful in a wide range of applications, from engineering to mathematical research. It would therefore be impossible to discuss every piece of software that includes polynomial computations at some level. Instead, we will attempt a broad overview of existing software that either takes a comprehensive view including such computations as a key component, or that which focuses specifically on some aspect of polynomial computation. Starting at the top level, we review existing and available software, and then highlight the gap in such software that we are attempting to fill.

MACSYMA was first developed starting in 1968 at MIT as the first comprehensive computer algebra system with the general goal of solving all manner of mathematical problems with the aid of a computer. Since then, numerous such systems have followed suit, albeit with varying aims. The most successful of these today are the commercial products Maple, Mathematica, and Magma, with open-source programs such as Sage, Maxima, Axiom, and Reduce also gaining popularity. Popular comprehensive numerical computation programs such as Matlab and Octave also have some symbolic computation capabilities, either built-in or available as an add-on package.

These programs have somewhat differing emphases, but they all include algorithms for a wide variety of mathematical problems, from differential equations to linear algebra. Of course also included are computations with polynomials, and these implementations are often highly optimized, as their speed affects the speed of many parts of the system. However, by their general nature as comprehensive systems, these programs all have extensive overhead costs incorporated, and are not well-suited for developing high-performance software for particular applications.

Large but somewhat less comprehensive systems have also been developed for a variety of mathematical problems, and some focus especially on polynomials. CoCoA (Abbott and Bigatti, 2011) and Singular (Decker, Greuel, Pfister, and Schönemann, 2010) are comprehensive systems for commutative and non-commutative algebra. They include multivariate polynomial arithmetic and focus specifically on Gröbner basis computations and algebraic geometry. Pari/GP (PAR, 2010) is a program with similar breadth, and also with support for polynomial arithmetic, but with a focus more on number theory. TRIP (Gastineau and Laskar, 2011) is a computer algebra system for celestial mechanics with very efficient sparse multivariate polynomial arithmetic over inexact domains. However, these programs are still quite general, featuring support for a wide range of operations and each with their own programming languages.

Finally, there are a few libraries for low-level computations with polynomials with a narrow focus on high performance for specific problems. A very popular and well-designed example is Victor Shoup's NTL, "a Library for doing Number Theory" (2009). This library contains very efficient implementations for univariate polynomials with integer, rational, and finite field coefficients, as well as basic algebraic operations such as factorization. NTL set the standard for the fastest implementations of these operations, using asymptotically fast algorithms along with simpler ones for small problem sizes, and making use of a variety of low-level tricks and tweaks. However, development has mostly been limited to bug fixes since 2005.

More recently, the FLINT library has emerged as a successor to NTL with similar goals but more active development (Hart, Johansson, and Pancratz, 2011). FLINT's core focus is on efficient arithmetic over $\mathbb{Z}[x]$, but it also has support for other kinds of univariate polynomials and some integer matrix computations as well, including lattice reduction algorithms. David Harvey, one of the original developers of FLINT, has also released an extremely efficient library for arithmetic in $\mathbb{Z}/m\mathbb{Z}$, where m is machine word-sized, called `zn_poly` (2008).

FLINT also contains implementations of multiple-precision integer arithmetic. Although the current work is focused specifically on computations with polynomials, there is an intimate connection with integer computations, as we have already seen. Besides FLINT, the Gnu Multiple Precision library (GMP) by Granlund et al. (2010) is certainly the most popular library for long integer arithmetic. It contains both `C` and `C++` bindings and has highly-tuned arithmetic routines, often written in assembly for a variety of different architectures.

The author is aware of only one software library focused on high-performance multivariate polynomial arithmetic, the SDMP package by Roman Pearce and Michael Monagan. This package features very efficient algorithms for sparse multiplication and division, and appears to be the fastest for some range of cases, even scaling to multi-cores (Monagan and Pearce, 2009, 2010a,b). However, a major drawback is that the domain is limited mostly to characteristic-zero rings and single-precision exponents. Furthermore, the software is closed-source and is only incorporated into the general-purpose commercial computer algebra system Maple.

Another high-performance library that is only available through Maple is `modpn` (Li, Maza, Rasheed, and Schost, 2009a). This library is mainly designed as the back-end for polynomial system solving via triangular decomposition, and it contains very efficient arithmetic for dense multivariate polynomials.

Finally, there have been a few academic software packages developed for computations on polynomials given in a functional representation. Dagwood is a Lisp-based system for computations with straight-line programs made to interface with Macsyma (Freeman et al., 1988). The FoxBox system allowed computations with polynomials represented by black boxes for their evaluation, including sparse interpolation (Díaz and Kaltofen, 1998). It made use of `C++` templates, relied on the use of NTL and GMP, and was built on top of the Saclib library. Unfortunately, Saclib appears to no longer be actively supported, and both Dagwood and FoxBox are no longer maintained or even available.

2.2 Goals of the MVP library

Considering the great success of low-level, open-source libraries for dense univariate polynomial arithmetic (namely NTL, FLINT, and `zn_poly`), it is surprising that no such library exists for multivariate and/or sparse polynomials. We are very thankful to the authors of efficient multivariate arithmetic libraries such as TRIP, SDMP, and `modpn` for publishing extensively on the design and implementation of their software. However, since their programs are closed-source, some details will always be hidden, and it will be difficult if not impossible to interface with their libraries, either by writing new algorithms to be inserted, or by including their libraries in a larger system. On the latter point, it should be noted that the open-source licenses and development of FLINT and Pari have led to their wide use as the back-end for polynomial computations in Sage.

Our goal therefore is a software library that is free and open-source and focuses on dense and sparse multivariate polynomial arithmetic and computations. In design, the software should be flexible enough to compare different algorithms, including interfacing with existing software, while also achieving high performance. In addition, the development should be open to allowing experimentation in new algorithms by researchers in symbolic computation. To our knowledge, no such library currently exists.

One consequence of this gap is that numerous algorithms developed over the past few decades currently have no readily-available implementation for experimentation and comparison. For instance, the celebrated sparse interpolation algorithm of Ben-Or and Tiwari (1988) and the sparse factorization algorithm of Lenstra (1999), while probably used as the back-end in at least a few existing software packages, have no currently-maintained implementations. This hinders both the development and widespread use of sparse polynomial algorithms.

In the remainder of this chapter, we describe the design and implementation of a new software library for polynomial arithmetic, which we call MVP (for MultiVariate Polynomials — also, we hope the library will be most valuable). The general goals are as described above, and the scope of algorithms is the same as those outlined in Chapter 1: arithmetic operations, algebraic computations, and inverse symbolic computation. The current state of the library is very immature and covers only basic arithmetic and the algorithms of this thesis. However, the architecture of the library is designed to allow significant additions in the coming years, including both celebrated past results that are currently without implementations, as well as (hopefully) future advances in algorithms for polynomial computation.

2.3 Library design and architecture

The design of the MVP library borrows heavily from FoxBox as well as the popular exact linear algebra package LinBox (Dumas, Gautier, Giesbrecht, Giorgi, Hovinen, Kaltofen, Saunders, Turner, and Villard, 2002). These packages organize data structures by mathematical significance and make extensive use of C++ templates to allow genericity while providing high performance. This genericity allows numerous external libraries to be "plugged in" and used for underlying arithmetic or other functionality.

We briefly describe some architectural features of the MVP library, followed by an overview of its contents and organization.

2.3.1 Library design

Template programming in C++ is essentially a built-in tool for compile-time code generation. Hence template libraries can achieve the same run-time performance as any C library, but have greater flexibility, composability, and maintainability.

For instance, an algorithm for multiplication of densely-represented univariate polynomials over an arbitrary ring can be written once, then used in any situation. At compile-time, the exact function calls for the underlying ring arithmetic are substituted directly, so that no branches or look-up tables are required at run-time.

Of course, template programming also has its drawbacks. For instance, consider a recursive polynomial representation using templates, so that an n-variate polynomial is defined as a univariate polynomial with $(n-1)$-variate polynomial coefficients. This could lead to a class definition such as `Poly< Poly< Poly< Poly< Ring > > > >`, which quickly becomes unwieldy and can result in tremendous code bloat. As compilers and hardware continue to improve, the cost of such expansive templates will decrease. However, the strain on users of the library will still be severe, especially considering that they may no be expert programmers but mathematicians requiring moderate but not optimal performance.

For these reasons, the MVP library also allows for polymorphism. This is achieved through abstract base classes for basic types such as polynomial rings, which then have a variety of implementing subclasses. Using polymorphism instead of templates means that types must be checked at run-time. This results in somewhat weaker performance, but potentially smaller code, faster compilation, and much easier debugging. By making full use of templates but also allowing polymorphism, the MVP library gives users more flexibility to trade off ease of programming with performance.

2.3.2 Main modules

The MVP library is divided into six modules: `misc`, `ring`, `poly`, `alg`, `test`, and `bench`. The coupling is also roughly in that order; for instance, almost every part of the library relies on something in the `misc` module, whereas nothing outside of the `bench` module relies on its contents. We now briefly describe the design and current functionality of each module.

`misc`: This module contains common utilities and other functionality that is useful in a variety of ways throughout the package. Functions for error handling and random number generation are two examples. Most functions in this module are not template functions and hence can be pre-compiled.

Handling random number generation globally allows for easily repeatable computations, by re-using the same seed. This is very important not only for benchmarking and debugging, but also for algorithm development and mathematical understanding. As we will see in later chapters of this thesis, the performance of some algorithms relies on unproven number-theoretic conjectures, and a failure may correspond to a counter-example to said conjectures.

`ring`: The abstract class `Ring` is the superclass of all coefficient types in the library. Here we follow the lead of LinBox in separating the ring class from ring elements. It is tempting to write a `C++` class for elements, say of the ring $\mathbb{Z}/m\mathbb{Z}$ for small integers m. Operator overloading would then allow easy use of arithmetic in this ring. However, this requires that the modulus m (and any other information on the ring) either be contained or pointed to by each field element, massively wasting storage, or be globally defined. NTL takes the latter approach, and as a result is inherently not able to be parallelized.

Rather, we have a `C++` class for the ring itself which contains functions for all kinds of arithmetic in the ring. Algorithms on ring elements must then be passed (or templatized on) the ring type. Elements of the ring can be any built-in or user-defined type, allowing more compact storage. The disadvantage to the user of not being able to use overloaded arithmetic operators is the only drawback.

Besides the abstract base class, the `ring` module also contains implementations for integer modular rings, arbitrary-precision integers and rational numbers, and approximate complex numbers. There are also adaptor classes to link with existing software such as NTL.

`poly`: The abstract class `Poly` is a subclass of `Ring` and the superclass of all polynomial ring types. The class is templatized over the coefficient ring, the storage type, and the number of variables. Using the number of variables as a template parameter will allow for highly efficient and easily maintainable recursive types, and specialisations for univariate and (perhaps in the future) bivariate polynomials are easily defined.

Separating the polynomial rings from their storage is not as important as in the low-level rings, but it will allow full composability, for instance defining a polynomial with polynomial coefficients. Of course the `Poly` type also contains extended functionality that are not applicable in general rings, e.g., for accessing coefficients,.

Implementations of polynomial rings include dense, sparse, and functional representations (i.e., black-box polynomials), as described in the previous chapter. It should be noted that the black-box representation has no problem with basic arithmetic functions such as multiplication and division, but other functions such as retrieving the coefficient of a given monomial are not supported and will throw an error.

The `poly` class contains extensive implementations of efficient polynomial arithmetic, and in particular multiplication. Most of these algorithms are specialized for certain polynomial representations and in some cases certain coefficient rings as well, for maximum performance.

`alg`: This module contains will contain algorithms for higher-level computations such as factorization, decomposition, and interpolation. Current functionality is limited to the problems discussed in this thesis, but extending this should be quite easy. The hope is that this module will become a testbed for new algorithmic ideas in polynomial computations.

`test`: Correctness tests are contained in this module. The goal of these tests is not to prove correctness but rather to catch programming bugs. As with the rest of the library, the correctness tests are generic and in general consist of constructing random examples and testing whether ring identities hold. For instance, in testing rings, for many triples of randomly-chosen elements (a, b, c) from the ring, we confirm distributivity: that $a \cdot (b + c) = a \cdot b + a \cdot c$. This kind of correctness test would be easy to fool with an intentionally incorrect and dishonest implementation, but is quite useful in catching bugs in programming. Generic tests also have the significant advantage of making it easy to add new rings without writing entirely new testing suites.

`bench`: One of the main purposes in writing the MVP library is to compare the practical performance of algorithms. Hence accurate and fair benchmarking is extremely important.

Benchmarking, like correctness testing, is also handled generically, but in a slightly different fashion as it is important to achieve the highest possible performance.

Rather than compiling a single program to benchmark every implementation of a given data structure or algorithm, we generate two programs for each test case. One “dry run” initializes all the variables and in most cases makes two passes through the computation that is to be tested. The “actual run” program does the same but makes one more pass through the computation. In examining the difference between the performance of these two programs, we see the true cost of the algorithm in question, subtracting any initialization and memory overhead for initialisation and (de)allocation. This also allows the use of excellent tools such as `valgrind` to evaluate performance which can only be run externally on entire programs. Furthermore, the code generation, compilation, and timing is all handled automatically by scripts, requiring only a single generic `C++` program, and a simple text file indicating the macro definitions to substitute in for each test case.

We have thus far used the benchmarking routines to manually tune performance, choosing optimal crossover points between algorithms for our target architecture. In the future, this process could be automated, allowing greater flexibility and performance on a wide variety of machines.

2.4 Algorithms for low-level operations

The MVP library is designed to allow other packages to be plugged in for low-level arithmetic. For instance, adaptor classes exist to use some of NTL’s types for modular and polynomial rings. However, the highest levels of performance for the most important operations are achieved with a bottom-up design and strong coupling between representations at all levels. Furthermore, at least including core functionality without requiring too many external packages will lead to more maintainable and (hopefully) longer-lasting code.

With this in mind, we now turn to the primitive operation of polynomial multiplication in $\mathbb{F}_p[x]$ for primes p that can be stored in a single machine word. This operation forms the basis for a variety of other algorithms, including long integer multiplication and representations of prime power fields. Furthermore, univariate polynomials of this kind will be used extensively as a test case for the algorithms developed in this thesis, as this domain bears true the often-useful algorithmic assumption that coefficients can be stored in a constant amount of space, and coefficient calculations take constant time.

We first examine algorithms and implementations for the coefficient field $\mathbb{F}_p$, and then turn to univariate multiplication over that field.

2.4.1 Small prime fields

Let p be a machine word-sized prime. Using the terminology of the IMM model, this means $p < 2^w$ for whatever problem size is under consideration. On modern hardware, this typically means p has at most 32 or 64 bits in the binary representation. For this discussion, we assume that arithmetic with powers of 2 is highly efficient. This is of course true on modern hardware; multiplications, divisions, additions, and subtractions modulo a power of two can be performed extremely quickly with shifts and bitwise logical instructions.

Most of the content here is well-known and used in many existing implementations. Further references and details can be found in the excellent books by Brent and Zimmermann (2010) and Hankerson, Menezes, and Vanstone (2004).

Addition and subtraction

The standard method for arithmetic modulo p is simply to perform computations over $\mathbb{Z}$, reducing each result modulo p via an integer division with remainder. This integer remainder operation, given by the `%` operator in `C++`, is very expensive compared to other basic instructions, and the main goal of the algorithms we now mention is to avoid its use.

We immediately see how this can be done for the operations of addition of subtraction. Let $a, b \in \mathbb{F}_p$, so that $0 \le a, b < p$ under the standard representation. Then $0 \le a + b < 2p - 1$, so after computing $a + b$ over $\mathbb{Z}$, we simply check if the sum is at least p, and if so subtract p, reducing into the desired range. We call this check-and-subtract a "correction". Subtraction reduces to addition as well since $-a \in \mathbb{F}_p$ is simply $p - a$, which always falls in the proper range when a is nonzero.

Some modern processors have very long pipelines and use branch prediction to "guess" which way an upcoming `if` statement will be evaluated. A wrong guess breaks the pipeline and is very expensive. It turns out that, on such processors, the idea above is quite inefficient since it continually requires checking whether the result is at least p. Furthermore, this branch will be impossible to reliably predict since it will be taken exactly one half of the time (when the operands are uniformly distributed).

Victor Shoup's NTL includes a very clever trick for avoiding this problem. First, the sum (or difference) is put into the range $-p + 1 \le s \le p - 1$, so that we need to check whether s is negative, and if so, add p to it. The key observation is that an arithmetic right shift of s to full width (either 32 or 64 bits) results in either a word with all `1` bits, if s is negative, or all `0` bits, if s is positive. By computing a bitwise `AND` operation with p, we get either p, if s is negative, or 0 otherwise. Finally, this is added to s to normalize into the range $[0, p - 1]$. Hence a branch is avoided at the cost of doing a single right shift and bitwise `AND` operation.

Despite these tricks, the modular reduction following additions is still costly. If the prime modulus p is much smaller than word-sized, further improvements can be gained by delaying modular reductions in certain algorithms. For instance, Monagan (1993) demonstrated improvements using this idea in polynomial multiplication. Say p has ℓ fewer bits than the width of machine words, i.e., $2^\ell p < 2^w$. Then 2^ℓ integers mod p can be summed in a single machine word, and the remainder of this sum modulo p can be computed with just ℓ "correction" steps, either using the branching or arithmetic right shift trick as discussed above.

Barrett's algorithm for multiplication

We now turn to the problem of multiplication of two elements $a, b \in \mathbb{F}_p$. The algorithm of Barrett (1987) works for any modulus p and takes advantage of the fact that arithmetic with powers of 2 is very efficient. The idea is to precompute an approximation for the inverse of p, which will enable a close approximation to the quotient when ab is divided by p.

Suppose p has k bits, i.e., $2^{k-1} \le p < 2^k$. Let $\mu = \lfloor 2^{2k}/p \rfloor$, stored as a precomputation. The algorithm to compute $ab \bmod p$ then proceeds in four steps:

1. $c_1 \leftarrow \lfloor ab/2^k \rfloor$
2. $q \leftarrow \lfloor c_1 \mu / 2^k \rfloor$
3. $r \leftarrow ab - qp$
4. Subtract p from r repeatedly until $0 \le r < p$

If the algorithm is implemented over the integers as stated, we know that the value of r computed on Step 3 is less than $4p$, and hence Step 4 iterates at most 3 times. Thus the reduced product is computed using three multiplications as well as some subtractions and bitwise shift operations.

If p can be represented exactly as a single- or double-precision floating point number, then we can precompute $\mu = 1/p$ as a floating-point approximation and perform the computation on Steps 1 and 2 in floating point as well, without any division by powers of 2. This will guarantee the initial value of r to satisfy $-p < r < 2p$, so only 2 "corrections" on Step 4 are required. This is in fact the approach used in NTL library on some modern processors that feature very fast double-precision floating point arithmetic. The general idea of using floating-point arithmetic for modular computations has been shown quite useful, especially when existing numerical libraries can be employed (Dumas, Giorgi, and Pernet, 2008, e.g.,).

Montgomery's algorithm

The algorithm of Montgomery (1985) is similar to Barrett's and again requires precomputation with p, but gains further efficiency by changing the representation. This time, let $k \in \mathbb{N}$ such that $p < 2^k$, but where 2^k can be much larger than p (unlike Barrett's algorithm). The finite field element $a \in \mathbb{F}_p$ is represented by the product $2^k a \bmod p$. Notice that the standard algorithms for addition and subtraction can still be used, since $2^k a + 2^k b = 2^k(a+b)$.

Now write $s = 2^k a$ and $t = 2^k b$, and suppose we want to compute the product of a and b, which will be stored as $2^k ab \bmod p$, or $st/2^k \bmod p$. The precomputation for this algorithm is $\mu = -1/p \pmod{2^k}$, which can be computed using the extended Euclidean algorithm. Notice this requires p to be a unit in $\mathbb{Z}/2^k\mathbb{Z}$, which means p must be odd for Montgomery's algorithm.

Given μ, s, t, the product $st/2^k \bmod p$ is computed in three steps:

1. $q \leftarrow (st) \cdot \mu \mod 2^k$

2. $r \leftarrow (st + qp)/2^k$

3. If $r < p$ return r, else return $r - p$

As with Barrett's algorithm, the product is computed using only three integer multiplications. Montgomery's form is more efficient, however, since the product on Step 1 is a "short product" where only the low-order bits are needed, and there are fewer additions, subtractions, and shifts required. Of course, there is a higher overhead associated with converting to/from the Montgomery representation, but this is negligible for most applications.

We have found a few more optimisations to be quite useful in the Montgomery representation, some of which are alluded to by Li, Maza, and Schost (2009b). First, the parameter k should be chosen to align with the machine word size, i.e., $k = w$, typically either 32 or 64. Using this, arithmetic modulo 2^k is of course just single-precision arithmetic; no additional steps are needed for example to perform the computation in Step 1 above.

We additionally observe that the high-order bits of a single-precision product are often computed "for free" by machine-level `MUL` instructions. These bits correspond to quotients modulo 2^k, as is needed on Step 2 above. The high-order bits are not directly available in high-level programming languages such as `C++`, but by writing very small inline assembly loops they may be extracted. For instance, in the `x86_64` architecture of popular contemporary 64-bit processors, any word-sized integer multiplication stores the high-order bits of the product into the `EDX` register. We use this fact in our implementation of the Montgomery representation in the MVP library, with positive results.

The Montgomery representation also lends itself well to delayed modular reduction in additions and subtractions, as discussed above. Again, say p has ℓ fewer bits than the machine word size, so that $2^\ell p < 2^w$. Now consider two un-normalized integers s, t in the Montgomery representation, satisfying $0 \le s, t \le 2^{\lfloor \ell/2 \rfloor} p$. The result r computed as their product in Step 2 of the Montgomery multiplication algorithm above is then bounded by

$$r = \frac{st + qp}{2^w} < \frac{2^\ell p^2 + 2^w p}{2^w} < 2p.$$

Hence $2^{\lfloor \ell/2 \rfloor}$ elements in $\mathbb{F}_p$ can be safely summed without performing any extra normalization.

In the MVP library, we embrace this by always storing finite field elements in the Montgomery reduction redundantly, as integers in the range $\{0, 1, \ldots, 2p\}$. When the modulus p is not too large, this allows direct remainder operations or "normalizations" to be almost entirely avoided in arithmetic calculations. The cost of comparisons is slightly increased, making this representation more useful for dense polynomial arithmetic, as opposed to sparse methods which frequently test for zero. Observe that zero testing is especially bad because there are three possible representations of zero: 0, p, or $2p$.

2.4.2 Univariate polynomial multiplication

The MVP library contains dense univariate polynomial multiplication routines incorporating the quadratic "school" method, Karatsuba's divide-and-conquer method, as well as FFT-based multiplication. The choice of algorithm for different ranges of input size is based on benchmarking on our test machine. We also implemented the well-known blocking algorithm for unbalanced polynomial multiplication, when the degrees of the two operands differ significantly. Roughly speaking, the idea is to break the larger polynomial into blocks equal in size to that of the smaller polynomial, compute the products separately, and then sum them. We will discuss all these algorithms for dense multiplication much more extensively in the next two chapters.

Multiplication of sparse univariate polynomials in the MVP library is performed using the algorithm of Johnson (1974), which has seen recent success in the work of Monagan and Pearce (2007, 2009). We briefly explain the idea of this algorithm. Given two polynomials $f, g \in \mathsf{R}[x]$ to be multiplied, write

$$\begin{aligned} f &= a_1 x^{d_1} + a_2 x^{d_2} + \cdots + a_s x^{d_s} \\ g &= b_1 x^{e_1} + b_2 x^{e_2} + \cdots + b_t x^{e_t}. \end{aligned}$$

As usual assume $d_1 < d_2 < \cdots < d_s = \deg f$, and similarly for the e_i's.

The terms in the product $f \cdot g \in \mathsf{R}[x]$ are computed one term at a time, in order. To accomplish this, we maintain a min-heap of size s which initially contains the pairs (d_i, e_1) for $1 \leq i \leq s$. The min-heap is ordered on the sum of the two exponents in each element, which corresponds to the degree of the single term in $f \cdot g$ that is the product of the corresponding terms in f and g. (For a refresher on heaps and other basic data structures, see any algorithms textbook such as Cormen, Leiserson, Rivest, and Stein (2001).)

From this ordering, the top of the min-heap always represents the next term (in increasing degree order) in the polynomial product we are computing. After adding the next term to a running total for $f \cdot g$, we update the corresponding pair in the heap by incrementing the second exponent along the terms in g. So, for example, the pair (d_3, e_5) is updated to (d_3, e_6), and then the heap is updated to maintain min-heap order, using $d_3 + e_6$ for this updated element's key. As the second exponent in each pair goes past b_t, it is removed from the heap, until the heap is empty and we are done.

For the cost analysis, notice that the number of times we must examine the top of the heap and update it is exactly $s \cdot t$, the total number of possible terms in the product. Because the heap has size s, the total time cost is $O(st \log s)$ exponent operations and $O(st)$ ring operations. In the IMM model, assuming each exponent requires at most k words of storage, this gives a total cost of $O(kst \log s)$ word operations. Observe that k is proportional to $\log \deg(f \cdot g)$.

This time cost matches other previously-known algorithms, for instance the "geobucket" data structure (Yan, 1998). However, the storage cost for this heap-based algorithm is better — only $O(s)$ words of intermediate storage are required, besides the input and output storage. In addition, since s and t are known, we can assume without loss of generality that $s \leq t$, improving the cost in unbalanced instances.

System component	Version / Capacity
Processor	AMD Phenom II
Processor speed	3.2 GHz
L1 Cache	512 KiB
L2 Cache	2 MiB
L3 Cache	8 MiB
Main memory (RAM)	8 GiB
Linux kernel	2.6.32-28-generic

Table 2.1: Properties of machine used for benchmarking

2.5 Benchmarking of low-level operations

We now compare our implementations of two low-level algorithms described above against the state of the art in existing software. The goal is not to claim that we have the best implementation of every algorithm, but merely that our algorithms are comparable to existing fast implementations. As these low-level routines are used in many of the higher-level algorithms discussed later in this thesis, the results here provide a justification for developing such algorithms in the MVP library.

All the timings given are on the 64-bit machine described in Table 2.1. We also used this machine to benchmark all the other implementations discussed in later chapters.

First we examine algorithms for finite field arithmetic. The MVP library currently contains a few implementations of finite field arithmetic for various purposes, but the fastest, at least for dense algorithms, is the Montgomery representation described above. We compared this representation against NTL's `zz_p` type for integers modulo a single-precision integer.

For the benchmarking, we used the so-called "axpy" operation that is a basic low-level operation for numerous algorithms in polynomial arithmetic as well as linear algebra. In this context, the axpy operation simply means adding a product ax to an accumulator y. Hence each axpy operation requires a single multiplication and addition in the ring.

NTL `zz_p`	MVP `ModMont`	Ratio
130.07 s	89.66 s	.689

Table 2.2: Benchmarking for ten billion axpy operations modulo a word-sized prime

Table 2.2 compares the cost, in CPU seconds, of 10^{10} (that is, ten billion) consecutive axpy operations using MVP's `ModMont` class (Montgomery representation), and using NTL's `zz_p`. These were compared directly in the `bench` module of MVP, using the "dry run" and actual run method described earlier to get accurate and meaningful results. We have also listed the ratio of time in MVP divided by time in NTL. So a lower number (and, in particular, a value less than one) means that our methods are outperforming NTL. This model will be repeated

Degree	Sparsity	SDMP time	MVP time	Ratio
1 000	100	9.25×10^{-4}	5.34×10^{-4}	.577
10 000	100	1.15×10^{-3}	6.47×10^{-4}	.563
1 000 000	100	1.21×10^{-3}	6.81×10^{-4}	.563
1 000	500	1.80×10^{-2}	1.03×10^{-2}	.572
10 000	500	2.58×10^{-2}	1.77×10^{-2}	.686
1 000 000	500	3.75×10^{-2}	2.08×10^{-2}	.555
1 000	1 000	6.23×10^{-2}	2.63×10^{-2}	.422
10 000	1 000	9.49×10^{-2}	6.99×10^{-2}	.737
1 000 000	1 000	1.62×10^{-1}	9.32×10^{-2}	.575
10 000	2 000	.337	.274	.813
1 000 000	2 000	.624	.427	.684
10 000	5 000	1.82	1.36	.747
1 000 000	5 000	3.53	3.20	.907

Table 2.3: Benchmarks for sparse univariate multiplication in MVP and SDMP

for most timing results we give, always with the time for the method under consideration in the numerator, and the time for the "standard" we are comparing against in the denominator.

The second problem we benchmarked is univariate sparse polynomial multiplication. Since multivariate sparse polynomials are represented in univariate using the Kronecker substitution, the univariate multiplication algorithm is of key importance. In these benchmarks, the coefficient ring is $\mathbb{F}_p$ for a single-precision integer p. Interestingly, we found that the cost was nearly the same using our Montgomery representation or NTL's `zz_p` representation. This is likely due in part to the fact that ring operations do not dominate the cost of sparse multiplication, as well as the increased cost of equality testing in our redundant Montgomery representation.

For comparison, we used the SDMP package for sparse polynomial arithmetic by Michael Monagan and Roman Pearce that has been integrated into MAPLE 14. Benchmarking our C++ program against a general-purpose computer algebra system with a run-time interpreted language such as MAPLE is inherently somewhat problematic. However, we tried to be as fair as possible to SDMP. We computed the timing difference between a "dry run" and an actual run, just as with the other benchmarking in MVP. We found that MAPLE spends a lot of time doing modular reductions, and so just computed the product over the integers in MAPLE, although our code took the extra time to do the modular reductions. Finally, we used the `SDMPolynom` routine in MAPLE to convert to the SDMP representation, and did not write out or echo any results. Since the SDMP library itself is (to our knowledge) written in C code linked with the MAPLE system, we feel this gives a very fair comparison to the MVP software.

The results of the sparse univariate multiplication benchmarks are shown in Table 2.3. As we can see, benchmarking sparse polynomial operations involves more parameters than just the number of iterations. The times shown are for a single multiplication with the specified degree and sparsity in each system, and in every test case both polynomials were randomly generated using the same parameters. In the actual benchmarking, we increased the number of iterations appropriately to get at least four significant figures of timing information for each example (on our machine, this meant at least ten seconds). As always, the ratio is the time of our implementation divided by the standard, which in this case is SDMP. Inverting these, we see that MVP is consistently between once and twice as fast as SDMP.

Conspicuously absent from this section is any discussion of dense polynomial arithmetic. While this is a primitive operation at least as important as those discussed here, we will explore this topic more thoroughly in Chapter 4. The new algorithms we present there have been implemented in the MVP library, and we show benchmarking comparisons against existing software in that chapter.

2.6 Further developments

The MVP library now contains fast implementations for basic arithmetic in a variety of polynomial rings, sparse interpolation over certain domains, and a few algebraic algorithms for sparse polynomials that will be presented later in this thesis. However, the software is still quite young and there is much work to be done before it will be useful to a wide audience. We mention now some of the most important tasks to improve the MVP library's usefulness.

First, a greater variety of algebraic algorithms should be included for basic problems such as factorization. One of our chief motivations in developing a new library was to give a good implementation of the sparse factorization algorithms by Lenstra (1999); Kaltofen and Koiran (2005, 2006), and this is certainly a key priority. Of course, dense factorization methods are also important and should be included. For this, it would be useful to tie in with existing fast dense multiplication algorithms implemented in NTL and FLINT.

Another major and important task for the future development of MVP is to build in parallelization. This will of course require not just careful implementations but in many cases new algorithmic ideas for a number of important, basic problems. Allowing for parallelization without affecting the sequential performance of algorithms will also be an important consideration and software engineering challenge. We have tried to design the library with thread safety in mind, but in some cases using global or static variables can drastically improve performance. Balancing this with the needs of parallel computation will present difficult and interesting challenges.

Finally, the success and longevity of any software package is heavily dependent on building and maintaining a community of users and developers. While the library has been solo-authored to begin with, it is hoped that more researchers and students will get involved with developing the library along the lines mentioned above. Ultimately, we would like MVP to be a popular venue to try and compare implementations of new algorithms for polynomial computations. In addition, if the library is fast and solves a variety of problems, it may be useful as an add-on package to larger systems such as SAGE or SINGULAR, further increasing the visibility and user base.

The great generality and simplicity of its rules makes arithmetic accessible to the dullest mind.

—Tobias Dantzig, *Numbers* (1930)

Chapter 3

In-place Truncated Fourier Transform

The fast Fourier transform (FFT) algorithm is a crucial operation in many areas of signal processing as well as computer science. Most relevant to our focus, all the asymptotically fastest methods for polynomial and long integer multiplication rely heavily on the FFT as a subroutine. Most commonly, particularly for multiplication algorithms, the radix-2 Cooley-Tukey FFT is used. In the typical case that the transform size is not exactly a power of two, the so-called truncated Fourier transform (TFT) can be used to avoid wasted computation. However, in using the TFT, a crucial property of the FFT is lost: namely, that it can be computed completely in-place, with the output overwriting the input. Here we present a new algorithm for the truncated Fourier transform which regains this property of the FFT and can be computed completely in-place and with the same asymptotic time cost as the original method.

The main results in this chapter represent joint work with David Harvey of New York University. We presented some of these results at ISSAC 2010 (Harvey and Roche, 2010). We also thank the helpful reviewer at that conference who pointed out the Devil's convolution algorithm of Crandall (1996).

3.1 Background

3.1.1 Primitive roots of unity and the discrete Fourier transform

Recall that an nth root of unity $\omega \in \mathsf{R}$ is any element such that $\omega^n = 1$; furthermore, ω is said to be an nth *primitive* root of unity, or n-PRU, if n is the least integer such that $\omega^n = 1$. The only roots of unity in the real numbers $\mathbb{R}$ are -1 and 1, whereas the complex numbers $\mathbb{C}$ contain an n-PRU for any n, for instance, $\mathbf{e}^{2\pi\mathbf{i}/n}$. Finite fields are more interesting: the finite field with q elements $\mathbb{F}_q$ contains an n-PRU if and only if $n \mid (q-1)$. If we have a generator α for the multiplicative group of the field, the ring element $\alpha^{(q-1)/n}$ gives one n-PRU.

The cyclic nature of PRUs is what makes them useful for computation. Mathematically, a n-PRU $\omega \in \mathsf{R}$ generates an order-n subgroup of R^*. In particular, $\omega^k = \omega^{k \operatorname{rem} n}$ for any integer exponent k.

The discrete Fourier transform (DFT) is a linear map that evaluates a given polynomial at powers of a root of unity. Given a polynomial $f \in \mathsf{R}[x]$ and an n-PRU $\omega \in \mathsf{R}$, the discrete Fourier transform of f at ω, written $\mathrm{DFT}_\omega(f)$, is defined as

$$\mathrm{DFT}_\omega(f) = \left(f\left(\omega^i\right)\right)_{0 \le i < n}.$$

That is, the DFT at ω of f is the polynomial f evaluated at every power of ω. By writing the polynomial $f = a_0 + a_1 x + a_2 x^2 + \cdots$ in the dense representation as a list $(a_0, a_1, a_2, \ldots) \in \mathsf{R}^*$ of its coefficients, the function DFT_ω is a map from R^* to R^n.

If we restrict to polynomials f with degree strictly less than n, then such polynomials can always be stored as an array of exactly n coefficients, and DFT_ω is a map from R^n to R^n. In these cases the DFT_ω map is also invertible, from the uniqueness of polynomial interpolation, using the fact that the values ω^i for $0 \le i < n$ are all distinct.

In fact, the inverse DFT is simply a DFT at ω^{-1}, scaled by n. To be precise,

$$\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(a_0, a_1, \ldots, a_{n-1})) = (na_0, na_1, \ldots, na_{n-1}).$$

This tight relationship between the forward and inverse DFT is crucial, especially for the application to multiplication.

3.1.2 Fast Fourier transform

Naïvely, evaluating $\mathrm{DFT}_\omega(f)$ requires evaluating f at n distinct points, for a total cost of $O(n^2)$ ring operations. The fast Fourier transform (FFT) is a much more efficient algorithm to compute $\mathrm{DFT}_\omega(f)$; for certain values of n, it improves the cost to $O(n \log n)$.

The FFT algorithm has been hugely useful in computer science, and in particular in the area of signal processing. For instance, it was named one of the top ten algorithms of the twentieth century (Dongarra and Sullivan, 2000), where the authors cite it as “the most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data”. The FFT has an interesting history: it was known to Gauss, but was later rediscovered in various forms by a number of authors in the twentieth century (Heideman, Johnson, and Burrus, 1984). The most famous and significant rediscovery was by Cooley and Tukey (1965), who were also the first to describe how to implement the algorithm in a digital computer.

We now briefly describe Cooley and Tukey’s algorithm. If n can be factored as $n = \ell m$, then the FFT is a divide-and-conquer strategy that computes a size-n DFT by m size-ℓ DFTs, followed by ℓ size-m DFTs. Given $f = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$, we define the polynomials $g_i \in \mathsf{R}[x]$, for $0 \le i < m$, by

$$g_i = \sum_{0 \le j < \ell} a_{i+mj} x^j = a_i + a_{i+m} x + a_{i+2m} x^2 + \cdots + a_{i+(\ell-1)m} x^{\ell-1}.$$

Observe that every coefficient in f appears in exactly one g_i, and each $\deg g_j < \ell$. In fact, we can write f as

$$f = \sum_{0 \le i < m} x^i g_i(x^m) = g_0(x^m) + x g_1(x^m) + \cdots x^{m-1} g_{m-1}(x^m).$$

Now let $i, j \in \mathbb{N}$ such that $0 \le i < \ell$ and $0 \le j < m$, and consider the $(i + \ell j)$th element of $\mathrm{DFT}_\omega(f)$, which is $f(\omega^{i+\ell j})$. From above, we have

$$f\left(\omega^{i+\ell j}\right) = \sum_{0\le k<m} \omega^{(i+\ell j)k} g_k\left(\omega^{(i+\ell j)m}\right) = \sum_{0\le k<m} \left(\omega^\ell\right)^{jk} \omega^{ik} g_k\left(\left(\omega^m\right)^i\right), \tag{3.1}$$

with the second equality following from the fact that $\ell m = n$ and $\omega^n = 1$. Now observe that ω^m is an ℓ-PRU in R, so each $g_k(\omega^{im})$ is simply the ith element of $\mathrm{DFT}_{\omega^m}(g_k)$. Once these DFTs are known, for each $i \in \{0, 1, \ldots, \ell - 1\}$, we can compute the polynomials

$$h_i = \sum_{0\le k<m} \omega^{ik} g_k\left(\left(\omega^m\right)^i\right) x^k.$$

Each h_i has degree less than m, and the coefficients of each h_i are computed by multiplying a power of ω with an element of $\mathrm{DFT}_{\omega^m}(g_k)$ for some k. (The powers of ω multiplied here, which have the form ω^{ik}, are called "twiddle factors".)

From (3.1), we can then write $f(\omega^{i+\ell j}) = h_i((\omega^\ell)^j)$. Since ω^ℓ is an m-PRU, these values are determined by a size-m DFT of h_i. Specifically, $(i + \ell j)$th element of $\mathrm{DFT}_\omega(f)$ is equal to the jth element of $\mathrm{DFT}_{\omega^\ell}(h_i)$.

This completes the description of the FFT algorithm. In summary, we first write down the g_is (which requires no computation) and compute m size-ℓ DFTs, $\mathrm{DFT}_{\omega^m}(g_i)$. These values are then multiplied by the twiddle factors to compute the coefficients of every h_i. This step only uses $O(n)$ ring operations, since we can first compute all powers of ω and then perform all the multiplications with elements of $\mathrm{DFT}_{\omega^m}(g_i)$. Finally, we compute ℓ size-m DFTs, $\mathrm{DFT}_{\omega^\ell}(h_i)$, which give the elements in $\mathrm{DFT}_\omega(f)$.

3.1.3 FFT variants

The radix-2 Cooley-Tukey FFT algorithm, which we will examine closely in this chapter, works when the size n of the DFT to be computed is a power of 2. This allows the FFT algorithm to be applied recursively. When $\ell = n/2$ and $m = 2$, the resulting algorithm is called the *decimation-in-time* variant. Since ℓ is also a power of 2, the 2 initial size-ℓ DFTs (on the g_is) can be computed with recursive calls. The base case is when the size $n = 2$, in which case the operation is computed directly with constant cost. This recursive algorithm has a total cost of $O(n \log n)$ ring operations.

The so-called decimation-in-frequency variant of the radix-2 FFT is just the opposite, setting $\ell = 2$ and $m = n/2$. (Both of these terms come from the application of FFT to signal processing.) Halfway between these two versions would be choosing ℓ to be the greatest power of two less than $\sqrt{n}$, and m the least power of two greater than $\sqrt{n}$. This type of FFT was carefully studied by Bailey (1990) and Harvey (2009a) and shown to improve the I/O complexity of FFT computation. However, all these variants still have time cost $O(n \log n)$.

The idea of the radix-2 FFT can of course be generalised to radix-3 FFT or, more generally, radix-k FFT for any $k \ge 2$. This will require that the input size n is a power of k, and the cost

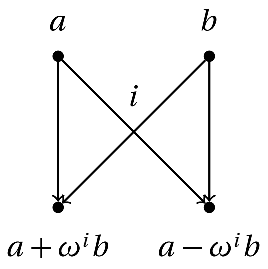


Figure 3.1: Butterfly circuit for decimation-in-time FFT

of the radix-k algorithm will be $O(kn \log_k n)$ — the same cost as the radix-2 version if k is constant.

Even more generally, any factorization of n can be used to generate an FFT algorithm. This is the idea of the mixed-radix FFT algorithm, whose cost depends on the largest factor of n used in the algorithm, but is at least $O(n \log n)$.

Though these methods are quite popular in signal processing algorithms, the radix-2 version (and sometimes radix 3) are used almost exclusively in symbolic computation. This is partly due to the fact that, at least for the application to integer multiplication, FFTs are most conveniently and commonly performed over fixed, chosen finite fields. Finding a prime p such that $p-1$ is divisible by a sufficiently high power of 2 is not difficult, and this then allows a radix-2 FFT of any smaller size to be performed in the field $\mathbb{F}_p$ of integers modulo p. This is a fundamentally different situation than working over the complex numbers, where no special construction or pre-computation is needed to find PRUs.

3.1.4 Computing the FFT in place

An important property of the FFT algorithm is that it can be computed in-place. Recall in the IMM model that this means the input space is empty, and the output space is initialised with the data to be transformed. An in-place algorithm is one which uses only a constant amount of temporary storage to solve this sort of problem. Any formulation of the Cooley-Tukey FFT algorithm described above can be computed in-place, as long as the base case of the recursion has constant size. In particular, the radix-2 and radix-3 versions can be computed in-place.

Consider the radix-2 decimation-in-time FFT. Since the algorithm is recursive and the base case has size 2, the entire circuit for this computation is composed of smaller circuits for size-2 DFTs, with varying twiddle factors (powers of ω). Such a circuit is called a “butterfly circuit”, and is shown in Figure 3.1. The circuit shown takes ring elements a, b as input and computes $a+\omega^i b$ and $a-\omega^i b$, where ω^i is the twiddle factor for this butterfly.

Figure 3.2 shows a circuit for the entire computation of a size-16 FFT (radix-2, decimation-in-time variant), with input polynomial $f = a_0 + a_1 x + \ldots + a_{15} x^{15}$. Though not shown, each of the butterflies making up this circuit has an associated twiddle factor as well. Because the size-2 butterfly can clearly be computed using at most one extra unit of storage, the entire computation can be as well. At each of the $\log_2 n$ steps through the algorithm, the elements

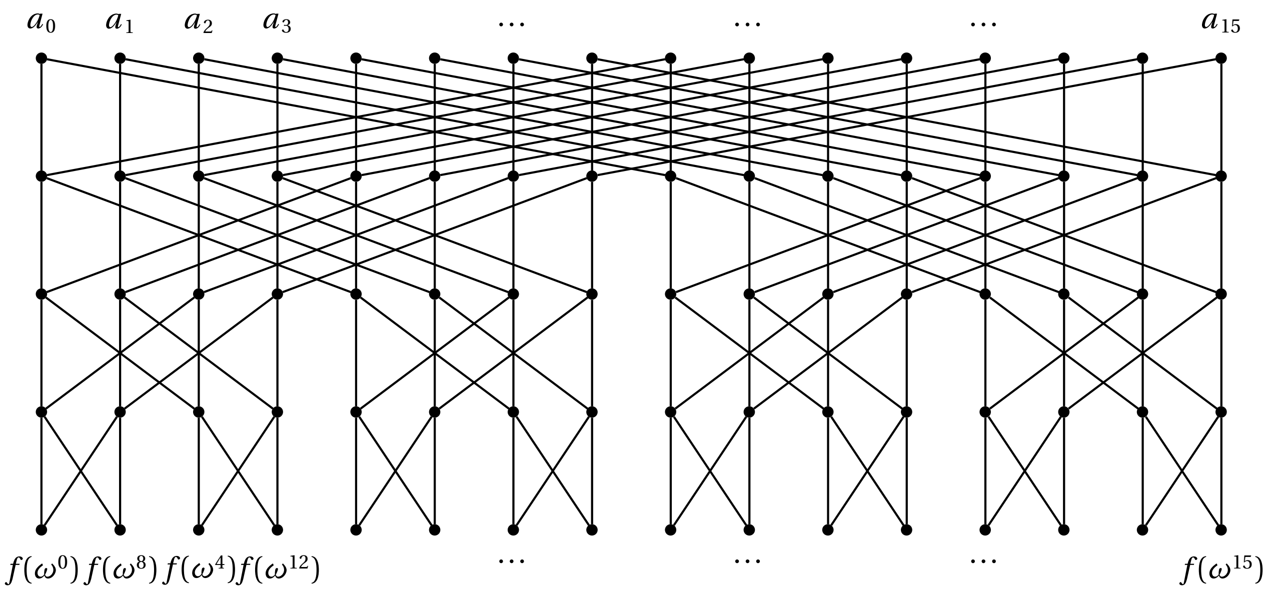


Figure 3.2: Circuit for radix-2 decimation-in-time FFT of size 16

stored in the size-n output space correspond to the nodes at one level of the circuit. Each step uses $O(n)$ time and $O(1)$ space to overwrite this array with the nodes at the next level of the circuit, until we reach the bottom and are done.

The only complication with this in-place computation is that the order of the elements is perturbed. In particular, if the ith position in the array holds the a_i at the beginning of the computation, the computed value in that position at the end of the algorithm will *not* be $f(\omega^i)$ but rather $f(\omega^{\text{rev}_k i})$. Here $k = \log_2 n$ and $\text{rev}_k\, i$ is the k-bit reversal of the binary representation of i. For instance, $\text{rev}_5(11) = 26$, since 11 is written `01011` in base 2 and 26 is `11010`. This permutation can easily be inverted in in-place to give the output in the correct order. However, for our application, this will actually not be necessary.

3.1.5 The truncated Fourier transform

As mentioned above, symbolic computation algorithms typically use radix-2 FFTs. However, the size of the input, for instance in the application to polynomial multiplication, will rarely be exactly a power of two. In this case, the traditional approach has been to pad the input with zeros so that the size is a power of two. However, this wastes not only memory space but also time in performing unnecessary computations.

Numerous approaches have been developed to address this problem and adapt radix-2 FFTs to arbitrary sized inputs and outputs. The "devil's convolution" algorithm of Crandall (1996) tackles this issue in the higher-level context of multiplication by breaking the problem into a large power-of-two size, solved with radix-2 FFTs, and a smaller arbitrary size, solved recursively. However, there are still significant jumps in the time and space cost (though not exactly at powers of two).

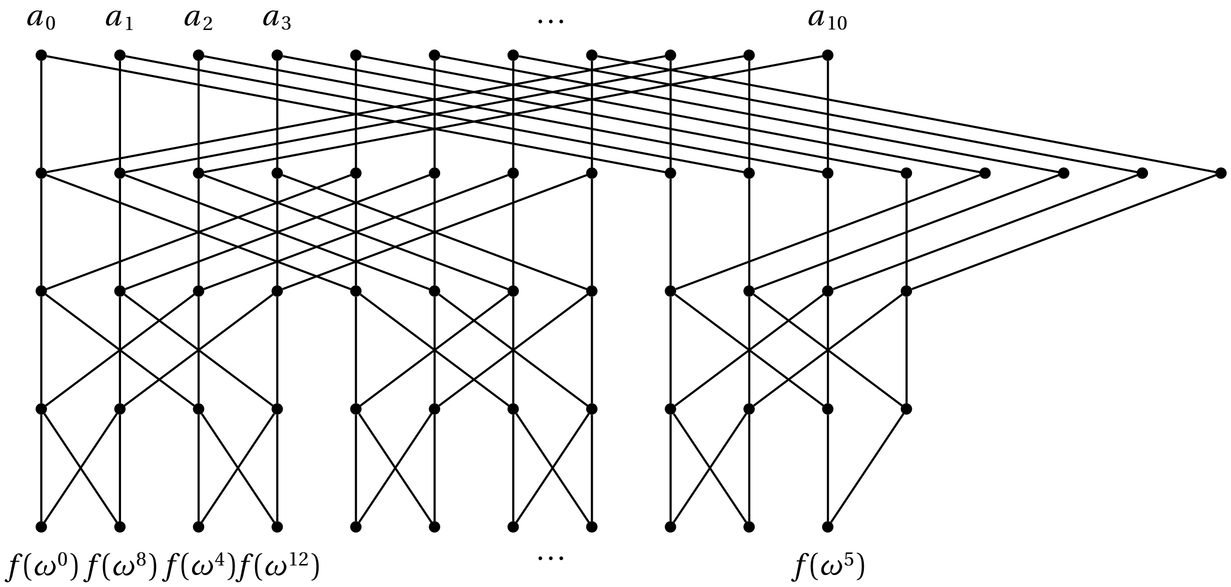


Figure 3.3: Circuit for forward TFT of size 11

At the lower-level context of DFTs, it has been known for some time that if only a subset of the output is needed, then the FFT can be truncated or "pruned" to reduce the complexity, essentially by disregarding those parts of the computation tree not contributing to the desired outputs (Markel, 1971; Sorensen and Burrus, 1993). More recently, van der Hoeven took the crucial step of showing how to invert this process by assuming a subset of input/output coefficients are zero, describing a truncated Fourier transform (TFT) and an *inverse* truncated Fourier transform (ITFT), and showing that this leads to a polynomial multiplication algorithm whose running time varies relatively smoothly in the input size (van der Hoeven, 2004, 2005).

Specifically, given an input vector of length $n \leq 2^k$, the TFT computes the first n coefficients of the ordinary Fourier transform of length 2^k, and the ITFT computes the inverse of this map. The running time of these algorithms smoothly interpolates the $O(n \log n)$ complexity of the standard radix-2 Cooley–Tukey FFT algorithm. One can therefore deduce an asymptotically fast polynomial multiplication algorithm that avoids the characteristic "jumps" in running time exhibited by traditional FFT-based polynomial multiplication algorithms when the output degree crosses a power-of-two boundary. This observation has been confirmed with practical implementations (van der Hoeven, 2005; Li et al., 2009b; Harvey, 2009a), with the most marked improvements in the multivariate case.

The structure of the TFT is shown in Figure 3.3, for an input of size 11. The structure is essentially the same as the size-16 decimation-in-time FFT, except that unnecessary computations — either those coming from inputs known to be zero, or those going to outputs that are not needed — are "pruned" away. Observe that, unlike with the FFT, inverting the TFT is completely non-trivial, and requires a more complicated algorithm, as shown by van der Hoeven (2004). We will not discuss the details of the inverse TFT algorithm presented there, but observe that it has roughly the same time cost as the forward TFT.

One drawback of van der Hoeven's algorithms is that while their time complexity varies smoothly with n, their space complexity does not. We can observe from the circuit for the size-11 TFT that 16 ring elements must be stored on the second step. In general, both the TFT and ITFT require $2^{\lceil \log_2 n \rceil} - n$ units of extra storage space, which in the worst case is $\Omega(n)$. Therefore the TFT and ITFT are not in-place algorithms.

3.1.6 DFT notation

For the discussion of our new algorithms below, it will be convenient to have a simpler notation for power-of-two roots of unity and bit reversals. First, denote by $\omega_{[k]}$ a primitive 2^k-th root unity, for some integer k. Our algorithm will require that such a root of unity exists for all $k \le \lceil \log_2 n \rceil$.

We assume that these PRUs are chosen compatibly, so that $\omega_{[k+1]}^2 = \omega_{[k]}$ for all $k \ge 0$. Define a sequence of roots $\omega_0, \omega_1, \ldots$ by $\omega_s = \omega_{[k]}^{\mathrm{rev}_k s}$, where $k \ge \lceil \lg(s+1) \rceil$ and $\mathrm{rev}_k s$ denotes the length-k bit-reversal of s. This definition is consistent because the PRUs were chosen compatibly. This is because, for any $i \in \mathbb{N}$, $\mathrm{rev}_{k+i} s = 2^i \cdot \mathrm{rev}_k s$, and therefore

$$\omega_{[k+i]}^{\mathrm{rev}_{k+i} s} = \omega_{[k+i]}^{2^i \, \mathrm{rev}_k s} = \omega_{[k]}^{\mathrm{rev}_k s}.$$

Thus we have

$$\begin{array}{llll} \omega_0 = \omega_{[0]} \ (=1) & \omega_2 = \omega_{[2]} & \omega_4 = \omega_{[3]} & \omega_6 = \omega_{[3]}^3 \\ \omega_1 = \omega_{[1]} \ (=-1) & \omega_3 = \omega_{[2]}^3 & \omega_5 = \omega_{[3]}^5 & \omega_7 = \omega_{[3]}^7 \end{array}$$

and so on. Note that

$$\omega_{2s+1} = -\omega_{2s} \qquad \text{and} \qquad \omega_{2s}^2 = \omega_{2s+1}^2 = \omega_s.$$

Let $f \in \mathsf{R}[x]$ be a polynomial with $\deg f < n$ as before. The notation just introduced has the convenient property that the ith element of $\mathrm{DFT}_\omega(f)$, for *any* power-of-two PRU ω, when written in the bit-reversed order as computed by the in-place FFT, is simply $f(\omega_i)$. If we again write a_i for the coefficient of x^i in f, then we will also write $\hat{a}_i = f(\omega_i)$ for the ith term in the DFT of f.

Algorithms 3.1 and 3.2 below follow the outline of the decimation-in-time FFT, and decompose f as

$$f(x) = g(x^2) + x \cdot h(x^2),$$

where $\deg g < \lceil n/2 \rceil$ and $\deg h < \lfloor n/2 \rfloor$. Write $b_0, b_1, \ldots, b_{\lceil n/2 \rceil - 1}$ for the coefficients of g and $c_0, c_1, \ldots, c_{\lfloor n/2 \rfloor}$ for the coefficients of h. Using the notation just introduced, the "butterfly relations" corresponding to the bottom level of the FFT circuit can be written

$$\begin{aligned} \hat{a}_{2s} &= \hat{b}_s + \omega_{2s} \hat{c}_s, \\ \hat{a}_{2s+1} &= \hat{b}_s - \omega_{2s} \hat{c}_s. \end{aligned} \tag{3.2}$$

3.2 Computing powers of ω

Both the TFT and ITFT algorithm require, at each recursive level, iterating through a set of index-root pairs such as $\{(i,\omega_i), 0 \le i < n\}$. A traditional, time-efficient approach would be to pre-compute all powers of $\omega_{[k]}$, store them in reverted-binary order, and then pass through this array with a single pointer. However, this is impossible under the restriction that no auxiliary storage space be used.

One solution, if using the ordinary FFT, is to use a decimation-in-frequency algorithm rather than decimation-in-time. In this version, at the kth level, the twiddle factors are accessed in normal order $\omega_{[k]}^0, \omega_{[k]}^1, \ldots$. Hence this approach can be used to implement the radix-2 FFT algorithm in-place.

However, our algorithm critically requires the decimation-in-time formulation, and thus accessing the twiddle factors in reverted binary order $\omega_0, \omega_1, \omega_2, \ldots$. To avoid this, we will compute the roots on-the-fly by iterating through the powers of $\omega_{[k]}$ in order, and through the indices i in bit-reversed order. That is, we will access the array elements in reverted binary order, but the powers of ω in the normal order.

Accessing array elements in reverted binary order simply requires incrementing an integer counter through $\mathrm{rev}_k\, 0, \mathrm{rev}_k\, 1, \mathrm{rev}_k\, 2, \ldots$. Of course such increments will no longer be single word operations, but fortunately the reverted binary increment operation can be performed in-place and in amortized constant time. The algorithm is simply the same as the usual bitwise binary increment operation, but performed in reverse.

3.3 Space-restricted TFT

In this section we describe an in-place TFT algorithm that uses only $O(1)$ temporary space (Algorithm 3.1). First we define the problem formally for the algebraic IMM model. There is no input or output in integers, so we denote an instance by the triple $(I, O, O') \in (\mathsf{R}^*)^3$ containing ring elements on the algebraic side. This is an *in-output* problem, so every valid instance has I empty, $O = (a_0, a_1, \ldots, a_{n-1})$, and $O' = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})$, representing the computation of a size-n DFT of the polynomial $f \in \mathsf{R}[x]$ with coefficients $a_0, \ldots, a_{n-1}$. To be completely precise, we should also specify that O contains a $2^{\lceil \log_2 n \rceil}$-PRU $\omega_{[\lceil \log_2 n \rceil]}$.

For the algorithm, we will use $M_O[i]$ to denote the ith element of the (algebraic) output space, at any point in the computation. The in-place algorithm we will present will use a constant number of extra memory locations $M_T[i]$, but we will not explicitly refer to them in this way.

The pattern of the algorithm is recursive, but we avoid recursion by explicitly moving through the recursion tree, avoiding unnecessary space usage. An example tree for $n = 6$ is shown in Figure 3.4. The node $S = (q, r)$ represents a sub-array of the output space with offset q and stride 2^r; the ith element in this sub-array is $S_i = M_O[q + i \cdot 2^r]$, and the length of the sub-array is given by

$$\mathrm{len}(S) = \left\lceil \frac{n-q}{2^r} \right\rceil.$$

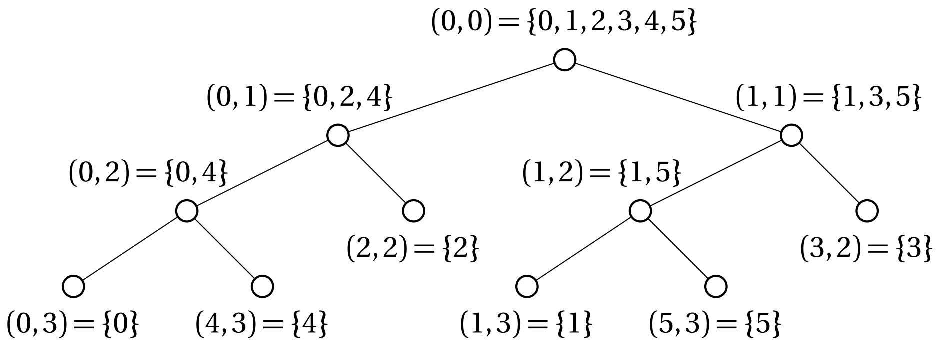


Figure 3.4: Recursion tree for in-place TFT of size $n = 6$

The root is $(0,0)$, corresponding to the entire input array of length n. Each sub-array of length 1 corresponds to a leaf node, and we define the predicate IsLeaf(S) to be true if and only if $\text{len}(S) = 1$. Each non-leaf node splits into even and odd child nodes.

To facilitate the path through the tree, we define a few functions on the nodes. First, for any non-leaf node (q,r), define

$$\text{EvenChild}(q,r) = (q, r+1),$$
$$\text{OddChild}(q,r) = (q+2^r, r+1).$$

If (q,r) is not the root node, define

$$\text{Parent}(q,r) = \begin{cases} (q, r-1), & \text{if } q < 2^{r-1}, \\ (q-2^{r-1}, r-1), & \text{if } q \geq 2^{r-1}. \end{cases}$$

Finally, for any node S we define

$$\text{LeftmostLeaf}(S) = \begin{cases} S, & \text{if IsLeaf}(S), \\ \text{LeftmostLeaf}(\text{EvenChild}(S)), & \text{otherwise.} \end{cases}$$

So for example, in the recursion tree shown in Figure 3.4, let the node $S = (1,1)$. Then $\text{OddChild}(S) = (3,2)$, the right child in the tree, $\text{Parent}(S) = (0,0)$, and $\text{LeftmostLeaf}(S) = (1,3)$. We also see that $\text{len}(S) = 3$ and $S_i = M_O[1+2i]$.

Algorithm 3.1 gives the details of our in-place algorithm to compute the TFT, using these functions to move around in the recursion tree.

The structure of the algorithm on our running example of size 11 is shown in the circuit in Figure 3.5. The unfilled nodes in the diagram represent temporary elements that are computed on-the-fly and immediately discarded in the algorithm, and the dashed edges indicate the extra computations necessary for these temporary computations. Observe that the structure of the circuit is more tangled than before, and in particular it is not possible to perform the computation level-by-level as one can for the normal FFT and forward TFT.

We begin with the following lemma.

Algorithm 3.1: In-place truncated Fourier transform

```
Input: Coefficients $a_0, a_1, \dots, a_{n-1} \in \mathsf{R}$ and PRU $\omega \in \mathsf{R}$, stored in-order in the output
       space $M_O$
Output: $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1}$, stored in-order in the output space $M_O$
$S \leftarrow$ LeftmostLeaf$(0,0)$
$prev \leftarrow$ null
while $true$ do
    $m \leftarrow \text{len}(S)$
    if IsLeaf$(S)$ or $prev =$ OddChild$(S)$ then
        for $(i, \theta) \in \{(j, \omega_{2j}) : 0 \le j < \lfloor m/2 \rfloor\}$ do
            $\begin{bmatrix} S_{2i} \\ S_{2i+1} \end{bmatrix} \leftarrow \begin{bmatrix} S_{2i} + \theta S_{2i+1} \\ S_{2i} - \theta S_{2i+1} \end{bmatrix}$
        if $S = (0,0)$ then halt
        $prev \leftarrow S$
        $S \leftarrow$ Parent$(S)$
    else if $prev =$ EvenChild$(S)$ then
        if len$(S) \equiv 1 \bmod 2$ then
            $v \leftarrow \sum_{i=0}^{(m-3)/2} S_{2i+1} \cdot (\omega_{(m-1)/2})^i$
            $S_{m-1} \leftarrow S_{m-1} + v \cdot \omega_{m-1}$
        $prev \leftarrow S$
        $S \leftarrow$ LeftmostLeaf(OddChild$(S)$)
```

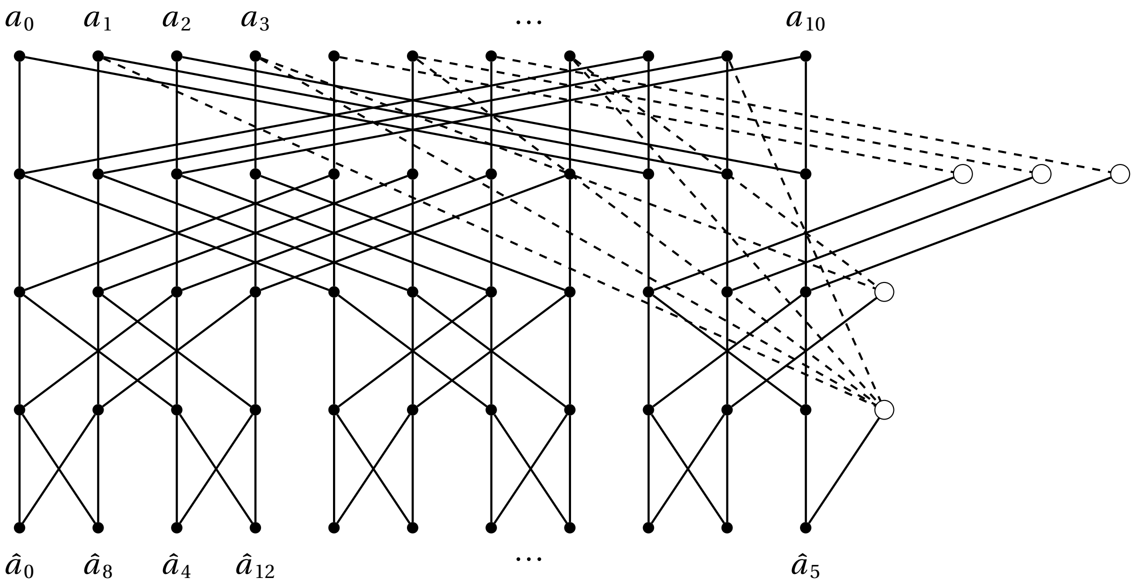


Figure 3.5: Circuit for Algorithm 3.1 with size $n = 11$

Lemma 3.1. *Let N be a node with* $\mathrm{len}(N) = \ell$, *and for* $0 \le i < \ell$, *define* α_i *to be the value stored in position* N_i *before some iteration of line 3 in Algorithm 3.1. If also* $S = \mathrm{LeftmostLeaf}(N)$ *at this point, then after a finite number of steps, we will have* $S = N$ *and* $N_i = \hat{\alpha}_i$ *for* $0 \le i < \ell$, *before the execution of line 8. No other array entries in* M_O *are affected.*

Proof. The proof is by induction on ℓ. If $\ell = 1$, then $\mathrm{IsLeaf}(N)$ is true and $\hat{\alpha}_0 = \alpha_0$ so we are done. Therefore assume $\ell > 1$ and that the lemma holds for all shorter lengths.

Consider the ring elements α_i as coefficients, and decompose the resulting polynomial into even and odd parts as

$$\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_{\ell-1} x^{\ell-1} = g(x^2) + x \cdot h(x^2).$$

So if we write $g = \sum_{0 \le i < \lceil \ell/2 \rceil} b_i x^i$ and $h = \sum_{0 \le i < \lfloor \ell/2 \rfloor} c_i x^i$ as before, then each $b_i = \alpha_{2i}$ and each $c_i = \alpha_{2i+1}$.

Since $S = \mathrm{LeftmostLeaf}(N)$ and N is not a leaf, $S = \mathrm{LeftmostLeaf}(\mathrm{EvenChild}(N))$ as well, and the induction hypothesis guarantees that the even-indexed elements of N, corresponding to the coefficients of g, will be transformed into

$$\mathrm{DFT}(g) = (\hat{b}_0, \hat{b}_1, \ldots, \hat{b}_{\lceil \ell/2 \rceil}),$$

and we will have $S = \mathrm{EvenChild}(N)$ before line 8. The following lines set $prev = \mathrm{EvenChild}(N)$ and $S = N$, so that lines 12–16 are executed on the next iteration.

If ℓ is odd, then $(\ell-1)/2 \ge \mathrm{len}(\mathrm{OddChild}(N))$, so $\hat{c}_{(\ell-1)/2}$ will not be computed in the odd subtree, and we will not be able to apply (3.2) to compute $\hat{\alpha}_{\ell-1} = \hat{b}_{(\ell-1)/2} + \omega_{\ell-1}\hat{c}_{(\ell-1)/2}$. This is why, in this case, we explicitly compute

$$v = h(\omega_{(\ell-1)/2}) = \hat{c}_{(\ell-1)/2}$$

on line 13, and then compute $\hat{\alpha}_{\ell-1}$ directly on line 14, before descending into the odd subtree.

Another application of the induction hypothesis guarantees that we will return to line 8 with $S = \mathrm{OddChild}(N)$ after computing $N_{2i+1} = \hat{c}_i$ for $0 \le i < \lfloor \ell/2 \rfloor$. The following lines set $prev = \mathrm{OddChild}(N)$ and $S = N$, and we arrive at line 6 on the next iteration. The **for** loop thus properly applies the butterfly relations (3.2) to compute $\hat{\alpha}_i$ for $0 \le i < 2\lfloor \ell/2 \rfloor$, which completes the proof. □

Now we are ready for the main result of this section.

Theorem 3.2. *Algorithm 3.1 correctly computes* $\mathrm{DFT}(f)$. *It is an in-place algorithm and uses* $O(n \log n)$ *algebraic and word operations.*

Proof. The correctness follows immediately from Lemma 3.1, since the algorithm begins by setting $S = \mathrm{LeftmostLeaf}(0,0)$, which is the first leaf of the whole tree. The space bound is immediate since each variable has constant size.

To verify the time bound, notice that the **while** loop visits each leaf node once and each non-leaf node twice (once with $prev = \mathrm{EvenChild}(S)$ and once with $prev = \mathrm{OddChild}(S)$). Since always $q < 2^r < 2n$, there are $O(n)$ iterations through the **while** loop, each of which has cost $O(\mathrm{len}(S) + \log n)$. This gives the total cost of $O(n \log n)$. □

3.4 Space-restricted ITFT

Next we describe an in-place inverse TFT algorithm (Algorithm 3.2). For some unknown polynomial $f = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \in R[x]$, the algorithm takes as input $\hat{a}_0, \ldots, \hat{a}_{n-1}$ and overwrites these values with $a_0, \ldots, a_{n-1}$.

The path of the algorithm is exactly the reverse of Algorithm 3.1, and we use the same notation as before to move through the tree. We only require one additional function:

$$\text{RightmostParent}(S) = \begin{cases} S, & \text{if } S = \text{OddChild}(\text{Parent}(S)), \\ \text{RightmostParent}(\text{Parent}(S)), & \text{otherwise.} \end{cases}$$

If LeftmostLeaf(OddChild(N_1)) = N_2, then Parent(RightmostParent(N_2)) = N_1. Therefore RightmostParent computes the inverse of the assignment on line 16 in Algorithm 3.1.

Algorithm 3.2: In-place inverse truncated Fourier transform

Input: Transformed coefficients $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1} \in \mathsf{R}$ and PRU $\omega \in \mathsf{R}$, stored in-order in the output space M_O

Output: The original coefficients $a_0, a_1, \ldots, a_{n-1}$, stored in-order in the output space M_O

```
1  $S \leftarrow (0,0)$
2  while $S \neq \text{LeftmostLeaf}(0,0)$ do
3      if IsLeaf($S$) then
4          $S \leftarrow \text{Parent}(\text{RightmostParent}(S))$
5          $m \leftarrow \text{len}(S)$
6          if $\text{len}(S) \equiv 1 \bmod 2$ then
7              $v \leftarrow \sum_{i=0}^{(m-3)/2} S_{2i+1} \cdot \omega_{(m-1)/2}^{i}$
8              $S_{m-1} \leftarrow S_{m-1} - v \cdot \omega_{m-1}$
9          $S \leftarrow \text{EvenChild}(S)$
10     else
11         $m \leftarrow \text{len}(S)$
12         for $(i,\theta) \in \{(j, \omega_{2j}^{-1}) : 0 \le j < \lfloor m/2 \rfloor\}$ do
13             $\begin{bmatrix} S_{2i} \\ S_{2i+1} \end{bmatrix} \leftarrow \begin{bmatrix} (S_{2i} + S_{2i+1})/2 \\ \theta \cdot (S_{2i} - S_{2i+1})/2 \end{bmatrix}$
14         $S \leftarrow \text{OddChild}(S)$
```

We leave it to the reader to confirm that the structure of the recursion in Algorithm 3.2 is identical to that of Algorithm 3.1, but in reverse. From this, the following analogues of Lemma 3.1 and Theorem 3.2 follow immediately:

Lemma 3.3. *Let N be a node with* len$(N) = \ell$, *and $a_0, a_1, \ldots, a_{n-1} \in \mathsf{R}$ be the coefficients of a polynomial in* $\mathsf{R}[x]$. *If $S = N$ and $N_i = \hat{a}_i$ for $0 \le i < \ell$ before some iteration of line 2 in*

Algorithm 3.2, then after a finite number of steps, we will have $S = \text{LeftmostLeaf}(N)$ *and* $N_i = a_i$ *for* $0 \le i < \ell$ *before some iteration of line 2. No other array entries in* M_O *are affected.*

Theorem 3.4. *Algorithm 3.2 correctly computes the inverse TFT. It is an in-place algorithm that uses* $O(n \log n)$ *algebraic and word operations.*

The fact that our in-place forward and inverse truncate Fourier transform algorithms are essentially reverses of each other is an interesting feature not shared by the original formulations in (van der Hoeven, 2004).

3.5 More detailed cost analysis

Our in-place TFT algorithm has the same asymptotic time complexity of previous approaches, but there will be a slight trade-off of extra computational cost against the savings in space. That is, our algorithms will perform more ring and integer operations than that of van der Hoeven (2004). Theorems 3.2 and 3.4 guarantee that this difference will only be a constant factor. By determining exactly what this constant factor is, we hope to gain insight into the potential practical utility of our in-place methods.

First we determine an upper bound on the number of multiplications in R performed by Algorithm 3.1, for an input of length n. In all the FFT variants, multiplications in the underlying ring dominate the cost, since the number of additions in R is at most twice the number of multiplications (and multiplications are more than twice as costly as additions), and the cost of integer (pointer) arithmetic is negligible.

Theorem 3.5. *Algorithm 3.1 uses at most*

$$\frac{5}{6} n \lceil \log_2 n \rceil + \frac{n-1}{3} \tag{3.3}$$

multiplications in R *to compute a size-*n *TFT.*

Proof. When $n > 1$, the number of multiplications performed depends on whether n is even or odd. If n is even, then the algorithm consists essentially of two recursive calls of size $n/2$, plus the cost at the top level (i.e., the root node in the recursion tree). Since n is even, the only multiplications performed at the top level are in lines 6–7, exactly $n/2$ multiplications.

For the second case, if n is odd, the number of multiplications are those used by two recursive calls of size $(n+1)/2$ and $(n-1)/2$, plus the number of multiplications performed at the top level. At the top level, there are again $\lfloor n/2 \rfloor$ multiplications from lines 6–7, but since n is odd, we must also count the $(n+1)/2$ multiplications performed on lines 13–14.

This leads to the following recurrence for $T(n)$, the number of multiplications performed by Algorithm 3.1 to compute a transform of length n:

$$T(n) = \begin{cases} 0, & n \le 1 \\ 2T\left(\frac{n}{2}\right) + \frac{n}{2}, & n \ge 2 \text{ is even} \\ T\left(\frac{n-1}{2}\right) + T\left(\frac{n+1}{2}\right) + n, & n \ge 3 \text{ is odd} \end{cases} \tag{3.4}$$

When n is even, we claim an even tighter bound than (3.3):

$$T(n) \leq (5n/6)\lceil \log_2 n \rceil - 1/3, \qquad n \text{ even.} \tag{3.5}$$

We now prove both bounds by induction on n.

For the base case, if $n = 1$, then the bound in (3.3) equals zero, and $T(1) = 0$ also.

For the inductive case, assume $n > 1$ and that (3.3) and (3.5) hold for all smaller values. We have two cases to consider: whether n is even or odd.

When n is even, from (3.4), the total number of multiplications is at most $2T(n/2) + n/2$. From the induction hypothesis, this is bounded above by

$$2\left(\frac{5n}{12}\left\lceil \log_2 \frac{n}{2} \right\rceil + \frac{n-2}{6}\right) + \frac{n}{2} = \frac{5n}{6}\lceil \log_2 n \rceil - \frac{2}{3} < \frac{5n}{6}\lceil \log_2 n \rceil - \frac{1}{3},$$

as claimed.

When n is odd, (3.4) tells us that exactly $T((n-1)/2) + T((n+1)/2) + n$ multiplications are performed. Now observe that at least one of the recursive-call sizes $(n+1)/2$ and $(n-1)/2$ must be even. Employing the induction hypothesis again, the total number of multiplications for odd n is bounded above by

$$\begin{aligned} &\frac{5(n-1)}{12}\left\lceil \log_2 \frac{n-1}{2} \right\rceil - \frac{1}{3} + \frac{5(n+1)}{12}\left\lceil \log_2 \frac{n+1}{2} \right\rceil + \frac{n-1}{6} + n \\ &= \frac{5n}{6}\left(\lceil \log_2 n \rceil - 1\right) + \frac{7n-3}{6} \\ &< \frac{5n}{6}\lceil \log_2 n \rceil + \frac{n-1}{3}. \end{aligned}$$

Therefore, by induction, the theorem holds for all $n \geq 1$.

□

Next we show that this upper bound is the best possible (up to lower order terms), by giving an exact analysis of the worst case.

Theorem 3.6. *Define the infinite sequence*

$$N_k = \frac{2^k - (-1)^k}{3}, \qquad k = 1, 2, 3, \ldots.$$

Then for all $k \geq 1$, we have

$$T(N_k) = \frac{5}{6}kN_k - \frac{23}{18}N_k + \frac{(-1)^k}{18}k - \frac{(-1)^k}{2}. \tag{3.6}$$

Proof. First observe that, since $4^i \equiv 1 \bmod 3$ for any $i \geq 0$, N_k is always an integer, and furthermore N_k is always odd. Additionally, for $k \geq 3$, we have $\lceil \log_2 N_k \rceil = k - 1$.

We now prove the theorem by induction on k. The two base cases are when $k = 1$ and $k = 2$, for which $T(N_1) = T(N_2) = T(1) = 0$, and the formula is easily verified in these cases.

Now let $k \geq 3$ and assume (3.6) holds for all smaller values of k. Because N_k is odd and greater than 1, from the recurrence in (3.4), we have $T(N_k) = T((N_k - 1)/2) + T((N_k + 1)/2) + n$. Rewriting -1 and 1 as $(-1)^k$ and $-(-1)^k$ (*not* necessarily respectively), we have

$$\begin{aligned} T(N_k) &= T\left(\frac{2^k + 2(-1)^k}{6}\right) + T\left(\frac{2^k - 4(-1)^k}{6}\right) + \frac{2^k - (-1)^k}{3} \\ &= T\left(\frac{2^{k-1} - (-1)^{k-1}}{3}\right) + T\left(\frac{2^{k-1} - 2(-1)^{k-2}}{3}\right) + \frac{2^k - (-1)^k}{3} \\ &= T(N_{k-1}) + 2T(N_{k-2}) + \frac{5 \cdot 2^{k-2} - 2(-1)^k}{3}. \end{aligned}$$

Now observe that $k-1$ and $k-2$ are both at least 1, so we can use the induction hypothesis to write

$$\begin{aligned} T(N_k) = & \frac{5}{6}(k-1)\frac{N_k + (-1)^k}{2} - \frac{23}{18} \cdot \frac{N_k + (-1)^k}{2} - \frac{(-1)^k}{18}(k-1) + \frac{(-1)^k}{2} \\ & + \frac{5}{3}(k-2)\frac{N_k - (-1)^k}{4} - \frac{23}{9} \cdot \frac{N_k - (-1)^k}{4} + \frac{(-1)^k}{9}(k-2) - (-1)^k \\ & + \frac{5}{4}N_k - \frac{(-1)^k}{4} \\ = & \frac{5}{6}kN_k - \frac{23}{18}N_k + \frac{(-1)^k}{18}k - \frac{(-1)^k}{2}. \end{aligned}$$

Therefore, by induction, (3.6) holds for all $k \geq 1$. □

Let us compare this result to existing algorithms. First, the usual FFT algorithm, with padding to the next power of 2, uses exactly $2^{\lceil \log_2 n \rceil - 1} \lceil \log_2 n \rceil$ multiplications for a size-n transform, which in the worst case is $n \lceil \log_2 n \rceil$. The original TFT algorithm improves this to $(n/2) \lceil \log_2 n \rceil + O(n)$ ring multiplications for any size-n transform.

In conclusion, we see that our in-place TFT algorithm gains some improvement over the normal FFT for sizes that are not a power of two, but not as much improvement as the original TFT algorithm.

3.6 Implementation

Currently implemented in the MVP library are our new algorithms for the forward and inverse truncated Fourier transform, as well as our own highly optimized implementations of the normal radix-2 FFT algorithm both forward and reverse. We ran a number of tests on these algorithms, alternating between sizes that were exactly a power of two, one more than a power

Size	Forward FFT (seconds)	Inverse FFT (ratio)	Algorithm 3.1 (ratio)	Algorithm 3.2 (ratio)
1024	7.50×10^{-5} s	1.17	1.62	2.36
1025	1.58×10^{-4} s	1.23	.949	1.23
1536	1.69×10^{-4} s	1.09	1.21	1.70
2048	1.62×10^{-4} s	1.17	1.60	2.83
2049	3.43×10^{-4} s	1.13	.930	1.24
3072	3.57×10^{-4} s	1.12	1.21	1.77
4096	3.61×10^{-4} s	1.14	1.52	2.34
4097	7.66×10^{-4} s	1.10	.877	1.17
6144	7.74×10^{-4} s	1.12	1.23	1.78

Table 3.1: Benchmarking results for discrete Fourier transforms

of two, and halfway between two powers of two. Of course we expect the normal radix-2 algorithm to perform best when the size of the transform is exactly a power of two. It would be easy to handle sizes of the form 2^k+1 specially, but we intentionally did not do this in order to see the maximum potential benefit of our new algorithms.

The results of our experiments are summarized in Table 3.1. The times for the forward FFT are reported in CPU seconds for a single operation, although as usual for the actual experiments there were several thousand iterations. The times for the other algorithms are reported only as ratios to the forward FFT time, and as usual a ratio less than one means that algorithm was faster than the ordinary forward FFT. Our new algorithms are competitive, but clearly not superior to the standard radix-2 algorithm, at least in the current implementation. While it might be possible to tweak this further, we submit a few justifications for why better run-time might not be feasible for these algorithms on modern systems.

First, observe that the structure of our in-place TFT algorithms is not level-by-level, as with the standard FFT and the *forward* TFT algorithm of van der Hoeven (2004). Both of our algorithms share more in common structurally with the original *inverse* TFT, which has an inherent recursive structure, similar to what we have seen here. In both these cases, the more complicated structure of the computation means in particular that memory will be accessed in a less regular way, and the consequence seems to be a slower algorithm.

Also regarding memory, it seems that a key potential benefit of savings in space is the ability — for certain sizes — to perform all the work at a lower level of the memory hierarchy. For instance, if two computations are identical, but one always uses twice as much space than the other, then for some input size the larger algorithm will require many accesses to main memory (or disk), whereas the smaller algorithm will stay entirely in cache (or main memory, respectively). Since the cost difference in accessing memory at different levels of the hierarchy is significant, we might expect some savings here.

In fact, we will see some evidence of such a phenomenon occurring with the multiplication algorithms of the next chapter. However, for the radix-2 DFT algorithms we have presented, this will *never* happen, precisely because the size of memory at any level in any modern architecture is always a power of two. This gives some justification for our findings here, but also lends motivation to studying the radix-3 case in the future, as we discuss below.

3.7 Future work

We have demonstrated that forward and inverse radix-2 truncated Fourier transforms can be computed in-place using $O(n \log n)$ time and $O(1)$ auxiliary storage. The next chapter will show the impact of this result on dense polynomial multiplication.

There is much work still to be done on truncated Fourier transforms. First, we see that our algorithm, while improving the memory usage, does not yet improve the time cost of other FFT algorithms. This is partly due to the cost of extra arithmetic operations, but as Theorem 3.5 demonstrates, our algorithm is actually superior to the usual FFT in this measure, at least in the worst case. A more likely cause for the inferior performance is the less regular structure of the algorithm. In particular, our in-place algorithms cannot be computed level-wise, as can the normal FFT and forward TFT algorithms. Developing improved in-place TFT algorithms whose actual complexity is closer to that of the original TFT would be a useful exercise.

To this end, it may be useful to consider an in-place truncated version of the radix-3 FFT algorithm. It should be relatively easy to adapt our algorithm to the radix-3 case, with the same asymptotic complexity. However, the radix-3 algorithm may perform better in practice, due to the nature of cache memory in modern computers.

Most of the cache in modern machines is least partly associative — for example, the L1 cache on our test machine is 8-way associative. This means that a given memory location can only be stored in a certain subset of the available cache lines. Since this associativity is always based on equivalence modulo powers of two, algorithms that access memory in power-of-two strides, such as in the radix-2 FFT, can suffer in cache performance. Radix-3 FFTs have been used in numerical libraries for many years to help mitigate this problem. Perhaps part of the reason they have not been used very much in symbolic computation software is that the blow-up in space when the size of the transform is not a power of 3 is more significant than in the radix-2 version. Developing an in-place variant for the radix-3 case would of course eliminate this concern, and so may give practical benefits.

Finally, it would be very useful to develop an in-place *multi-dimensional* TFT or ITFT algorithm. In one dimension, the ordinary TFT can hope to gain at most a factor of two over the FFT, but a d-dimensional TFT can be faster than the corresponding FFT by a factor of 2^d, as demonstrated in (Li et al., 2009b). An in-place variant along the lines of the algorithms presented in this paper could save a factor of 2^d in both time and memory, with practical consequences for multivariate polynomial arithmetic.

> The biggest difference between time and space is that you can't reuse time.
>
> —Merrick Furst

Chapter 4

Multiplication without extra space

The multiplication of dense univariate polynomials is one of the most fundamental problems in mathematical computing. In the last fifty years, numerous fast multiplication algorithms have been developed, and these algorithms are used in practice today in a variety of applications. However, all the fast multiplication algorithms require a linear amount of intermediate storage space to compute the result. For each of the two most popular fast multiplication algorithms, we develop variants with the same speed but which use much less temporary storage.

We first carefully define the problem under consideration, as well as the notation we will use throughout this chapter. Consider two polynomials $f, g \in \mathsf{R}[x]$, each with degree less than n, written

$$\begin{aligned} f &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ g &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}. \end{aligned}$$

We consider algorithms for dense univariate polynomial multiplication in an algebraic IMM. Formally, the input and output will all be on the algebraic side. The read-only input space will contain the $2n$ coefficients of f and g, and correct algorithms must write the $2n-1$ coefficients of their product into the output space and halt. In all algorithms discussed, the cost of ring operations dominates that of integer arithmetic with machine words, and so we state all costs in terms of ring operations.

Some of the results in this chapter were presented at ISSAC 2009 and 2010 (Roche, 2009; Harvey and Roche, 2010).

4.1 Previous results

Observe that the product $f \cdot g$ can be written

$$\sum_{i=0}^{2n-2} \left(\sum_{j=\max(0,i-n+1)}^{\min(i,n-1)} a_j b_{i-j} \right) x^i.$$

Using only a single temporary ring element as an accumulator, we can compute each inner summation, one at a time, to determine all the coefficients of the product. This is called the naïve or school method for multiplication. It works over any ring, uses $O(n^2)$ ring operations, and requires only a constant amount of temporary working space.

So-called fast multiplication algorithms are those whose cost is less than quadratic. We now briefly examine a few such algorithms.

4.1.1 Karatsuba's algorithm

Karatsuba was the first to develop a sub-quadratic multiplication algorithm (1963). This is a divide-and-conquer method and works by first splitting each of f and g into two blocks with roughly half the size. Writing $k = \lfloor n/2 \rfloor$, the four resulting polynomials are

$$\begin{aligned} f_0 &= a_0 + a_1 x + \cdots + a_{k-1} x^{k-1} & \qquad f_1 &= a_k + a_{k+1} x + \cdots a_{n-1} x^{n-k-1} \\ g_0 &= b_0 + b_1 x + \cdots + b_{k-1} x^{k-1} & \qquad g_1 &= b_k + b_{k+1} x + \cdots b_{n-1} x^{n-k-1}. \end{aligned}$$

We can therefore write $f = f_0 + f_1 x^k$ and $g = g_0 + g_1 x^k$, so their product is $f_0 g_0 + (f_0 g_1 + g_1 f_0) x^k + f_1 g_1 x^{2k}$. Gauss again was the first to notice (in the context of complex number multiplication) that this product can also be written

$$f \cdot g = f_0 g_0 + \left((f_0 + f_1)\,(g_0 + g_1) - f_0 g_0 - f_1 g_1 \right) x^k + f_1 g_1 x^{2k}.$$

Using this formulation, the product can be computed using exactly 3 recursive calls of approximately half the size, followed by $O(n)$ additions and subtractions. The total asymptotic cost is therefore $O(n^{\log_2 3})$, which is $O(n^{1.59})$.

To be specific, label the three intermediate products as follows:

$$\alpha = f_0 \cdot g_0, \qquad \beta = f_0 \cdot f_1, \qquad \gamma = (f_0 + f_1) \cdot (g_0 + g_1). \tag{4.1}$$

So the final product of f and g is computed as

$$f \cdot g = \alpha + (\gamma - \alpha - \beta) \cdot x^k + \beta \cdot x^{2k}. \tag{4.2}$$

A straightforward implementation might allocate n units of extra storage at each recursive step to store the intermediate product γ, resulting in an algorithm that uses a linear amount of extra space and performs approximately $4n$ additions of ring elements besides the three recursive calls.

There is of course significant overlap between the three terms of (4.2). To see this more clearly, split each polynomial α, β, γ into its low-order and high-order coefficients as with f and g. Then we have (with no overlap):

$$f \cdot g = \alpha_0 + (\gamma_0 + \alpha_1 - \alpha_0 - \beta_0) x^k + (\gamma_1 + \beta_0 - \alpha_1 - \beta_1) x^{2k} + \beta_1 x^{3k} \tag{4.3}$$

Examining this expansion, we see that the difference $\alpha_1 - \beta_0$ occurs twice, so the number of additions can be reduced to $7n/2$ at each recursive step. Popular implementations of Karatsuba's algorithm already make use of this optimisation.

A few authors have also examined the problem of minimising the extra storage space required for Karatsuba's algorithm. Maeder (1993) showed how to compute tight bounds on the amount of extra temporary storage required at the top level of the algorithm, allowing temporary space to be allocated once for all recursive calls. This bound is approximately $2n$ in Maeder's formulation, where the focus was specifically on long integer multiplication.

An improvement to this for polynomial multiplication was given by Thomé (2002), who showed how to structure Karatsuba's algorithm so that only about n extra ring elements need to be stored.

It should be mentioned that both these algorithms are already working in essentially the same model as an IMM, re-using the output space repeatedly in an attempt to minimise the amount of extra space required. These approaches are also discussed by Brent and Zimmermann (2010) in a broader context. Those authors, who have extensive experience in fast implementations of multiplication algorithms, claim that "The efficiency of an implementation of Karatsuba's algorithm depends heavily on memory usage". Our experiments, given at the end of this chapter, help support this claim.

The first algorithm that we will present has the same asymptotic time complexity as Karatsuba's algorithm and the variants above, but only requires $O(\log n)$ temporary storage.

4.1.2 FFT-based multiplication

The fastest practical algorithms for integer and polynomial multiplication are based on the fast Fourier transform algorithm discussed in the previous chapter. These algorithms use the FFT to evaluate the unknown product polynomial at a number of points, then apply the inverse FFT to compute the coefficients of the product from these evaluation points.

To compute the product of f and g, two univariate polynomials with degrees less than n as above, we use a 2^m-PRU ω, where $2^m \geq 2n-1$. We need a PRU of this order because the product polynomial has size at most $2n-1$, and we need at least this many points to uniquely recover it with polynomial interpolation.

The first step is to compute $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$. Observe that each DFT has size 2^m, which is at most $2(2n-1)-1 = 4n-3$. Since each input polynomial has degree at less than n, both coefficient vectors must be padded with zeroes to almost four times their width, at least if the usual radix-2 FFT algorithm is used. Even making use of the output space, and using in-place FFTs, this step requires extra temporary space for about $6n$ ring elements.

From the initial DFTs, we have $f(\omega^i)$ and $g(\omega^i)$ for $0 \leq i < 2^m$. Write $h = f \cdot g$ for their product in $\mathsf{R}[x]$. From the DFTs of f and g, we can easily write down $\mathrm{DFT}_\omega(h)$ using exactly 2^m multiplications in R, since $h(\omega^i) = f(\omega^i) \cdot g(\omega^i)$ for any i. Finally, we compute the inverse FFT of the evaluations of h to recover the coefficients of the product. Since 2^m could be about twice as large as $\deg h$, we may have to trim some zeros from the end of this output before copying the coefficients of h to the output space.

In summary, we compute

$$f \cdot g = \frac{1}{n} \mathrm{DFT}_{\omega^{-1}} \left(\mathrm{DFT}_{\omega}(f) * \mathrm{DFT}_{\omega}(g) \right),$$

where $*$ signifies pairwise multiplication of vector elements. The last two steps (writing down $\mathrm{DFT}_{\omega}(h)$ and computing the inverse FFT) can be computed in-place in the storage already allocated for the forward DFTs. Therefore the total amount of extra space required for the standard FFT-based multiplication algorithm is approximately $6n$ ring elements.

This basic formulation of FFT-based multiplication has been unchanged for many years, while the main focus of algorithmic development has been on the troublesome requirement that R contains an n-PRU ω. When it does not, we can construct a so-called virtual root of unity by working in a larger field and incurring a small multiplicative factor in the complexity.

For n-bit integer multiplication, Schönhage and Strassen (1971) showed how to achieve bit complexity $O(n \log n \operatorname{loglog} n)$ for this problem. This has recently been improved by Fürer (2007) and De, Kurur, Saha, and Saptharishi (2008) to $O(n \cdot \log n \cdot 2^{\log^* n})$, where $\log^* n$ indicates the *iterated logarithm*, defined as the number of times logarithm must be taken to reach a constant value. Recall from Section 1.4, however, that these subtleties of bit complexity are not meaningful in the IMM model.

Schönhage and Strassen's algorithm can also be applied to polynomial multiplication in the algebraic IMM model, giving a cost of $O(n \log n \operatorname{loglog} n)$ ring operations for degree-n polynomial multiplication, but only when the ring R admits division by 2. Schönhage (1977) used radix-3 FFTs to extend to rings of characteristic 2, and Cantor and Kaltofen (1991) further extended to arbitrary algebras, including non-commutative algebras, with the same asymptotic cost. It remains open whether the new improvements in the bit complexity of multiplication can be applied to the polynomial case.

In any case, this work explicitly dodges these issues by simply assuming that the ring R already contains an n-PRU ω. That is, we assume any of the above methods has been used to add virtual roots of unity already, and optimise from there on. The second algorithm we present reduces the amount of temporary storage required for FFT-based multiplication, over rings that already contain the proper 2^m-PRU, from $6n$ ring elements as in the standard algorithm above, to $O(1)$.

4.1.3 Lower bounds

There are a few lower bounds on the multiplication problem that merit mentioning here. First, in time complexity, Bürgisser and Lotz (2004) proved that at least $\Omega(n \log n)$ ring operations are required for degree-n polynomial multiplication over $\mathbb{C}[x]$ in the *bounded coefficients model*. This model encompasses algebraic circuits where the scalars (in our notation from Chapter 1, edge labels) are all at most 2 in absolute value. In particular, their result implies lower bounds for multiplication by *universal* IMM algorithms that contain no branch instructions.

More to the point of the current discussion, a few time-space tradeoffs for multiplication were given among the large number of such results that appeared a few decades ago. Savage and Swamy (1979) proved that $\Omega(n^2)$ time × space is required for boolean circuits computing binary integer multiplication. Here space is defined as the maximum number of nodes in the circuit that must reside in memory at any given point during the computation.

Algebraic circuits, or straight-line programs, are said to model "oblivious" algorithms, where the structure of the computation is fixed for a given input size. In fact, all the multiplication algorithms discussed above have this property. However, branching programs are more general in that they allow the computation path to change depending on the input. For binary integers, these are essentially an extension of binary decision trees. In this model, Abrahamson (1986) proved that at least $\Omega(n^2/\log^2 n)$ time × space is required for integer multiplication. In this model, space is measured as the logarithm of the number of nodes in the branching program, which corresponds to how much "state" information must be stored in an execution of the program to keep track of where it is in the computation.

The algorithms we present will break these lower bounds for time times temporary space in the algebraic IMM model. Our algorithms do not branch by examining ring elements, and therefore can be modelled by algebraic circuits or straight-line programs. They break these lower bounds by allowing both reads *and* writes to the output space. By re-using the output space, we show that these time-space tradeoffs can be broken. Observe that this follows similar results for more familiar problems such as sorting, which requires $\Omega(n^2)$ time × space in these models as well, but of course can be solved in-place in $O(n \log n)$ time, in the IMM model.

4.2 Space-efficient Karatsuba multiplication

We present an algorithm for polynomial multiplication which has the same mathematical structure as Karatsuba's, and the same time complexity, but which makes careful re-use of the output space so that only $O(\log n)$ temporary space is required for the computation. The description of the algorithm is cleanest when multiplying polynomials in $\mathsf{R}[x]$ with equal sizes that are both divisible by 2, so for simplicity we will present that case first. The corresponding methods for odd-sized operands, different-sized operands, and multiple-precision integers will follow fairly easily from this case.

4.2.1 Improved algorithm: general formulation

The key to obtaining $O(\log n)$ extra space for Karatsuba-like multiplication is by solving a slightly more general problem. In particular, two extra requirements are added to the algorithm at each recursive step.

Condition 4.1. *The low-order n coefficients of the output space are pre-initialized and must be added to. That is, half of the product space is initialized with a polynomial $h \in \mathsf{R}[x]$ of degree less than n.*

With Condition 4.1, the computed result should now equal $h + f \cdot g$.

Condition 4.2. *The first operand to the multiplication f is given as two polynomials which must first be summed before being multiplied by g. That is, rather than a single polynomial $f \in \mathsf{R}[x]$, we are given two polynomials $f^{(0)}, f^{(1)} \in \mathsf{R}[x]$, each with degree less than n.*

With both these conditions, the result of the computation should be $h + (f^{(0)} + f^{(1)}) \cdot g$.

Of course, these conditions should not be made on the very first call to the algorithm, and this will be discussed in the next subsection. We are now ready to present the algorithm in the easiest case that $f, g \in \mathsf{R}[x]$ with $\deg f = \deg g = 2k - 1$ for some $k \in \mathbb{N}$. If A is an array in memory, we use the notation of $A[i..j]$ for the sub-array from indices i to j (inclusive), with $0 \le i \le j < |A|$. If array A contains a polynomial f, then the array element $A[i]$ is the coefficient of x^i in f. The three read-only input operands $f^{(0)}, f^{(1)}, g$ are stored in arrays A, B, C, respectively, and the output is written to array D. From the first condition, $D[0..2k-1]$ is initialized with h.

In the notation of an algebraic IMM, we can think of the read-only arrays A, B, C as residing in the input memory M_I on the algebraic side, and D is the output memory space M_O on the algebraic side. In some recursive calls, however, A, B, C may actually reside in the output space, although this does not change the algorithm. Incidentally, this is the first IMM problem we have seen where valid instances (I, O, O') have all three parts non-empty.

Algorithm 4.1: Space-efficient Karatsuba multiplication

```
Input: $k \in \mathbb{N}$ and $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ with degrees less than $2k$ in arrays $A, B, C, D$,
        respectively
Output: $h + (f^{(0)} + f^{(1)}) \cdot g$ stored in array $D$
1  $D[k..2k-1] \leftarrow D[k..2k-1] + D[0..k-1]$
2  $D[3k-1..4k-2] \leftarrow A[0..k-1] + A[k..2k-1] + B[0..k-1] + B[k..2k-1]$
3  Recursive call with $A' \leftarrow C[0..k-1]$, $B' \leftarrow C[k..2k-1]$, $C' \leftarrow D[3k-1..4k-2]$, and
   $D' \leftarrow D[k..3k-2]$
4  $D[3k-1..4k-2] \leftarrow D[k..2k-1] + D[2k..3k-2]$
5  Recursive call with $A' \leftarrow A[0..k-1]$, $B' \leftarrow B[0..k-1]$, $C' \leftarrow C[0..k-1]$, and
   $D' \leftarrow D[0..2k-2]$
6  $D[2k..3k-2] \leftarrow D[2k..3k-2] - D[k..2k-2]$
7  $D[k..2k-1] \leftarrow D[3k-1..4k-2] - D[0..k-1]$
8  Recursive call with $A' \leftarrow A[k..2k-1]$, $B' \leftarrow B[k..2k-1]$, $C' \leftarrow C[k..2k-1]$, and
   $D' \leftarrow D[2k..4k-2]$
9  $D[k..2k-1] \leftarrow D[k..2k-1] - D[2k..3k-1]$
10 $D[2k..3k-2] \leftarrow D[2k..3k-2] - D[3k..4k-2]$
```

Table 4.1 summarizes the computation by showing the actual values (in terms of the input polynomials and intermediate products) stored in each part of the output array D after each step of the algorithm. Between the recursive calls on Steps 3, 5, and 8, we perform some additions and rearranging to prepare for the next multiplication. Notice that a few times a value is added somewhere only so that it can be cancelled off at a later point in the algorithm. An example of this is the low-order half of h, h_0, which is added to h_1 on Step 1 only to be cancelled when we subtract $(\alpha_0 + h_0)$ from this quantity later, on Step 7.

	$D[0..k-1]$	$D[k..2k-1]$	$D[2k..3k-2]^*$	$D[3k-1..4k-2]^*$
0	h_0	h_1	—	—
1	h_0	h_0+h_1	—	—
2	h_0	h_0+h_1	—	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
3	h_0	$h_0+h_1+\gamma_0$	γ_1	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
4	h_0	$h_0+h_1+\gamma_0$	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
5	$h_0+\alpha_0$	α_1	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
6	$h_0+\alpha_0$	α_1	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
7	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
8	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
9	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
10	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1-\beta_1$	β_1

*The last two sub-arrays shift by one to $D[2k..3k-1]$ and $D[3k..4k-2]$ at Step 8.

Table 4.1: Values stored in D through the steps of Algorithm 4.1

The final value stored in D after Step 10 is

$$\begin{aligned}&(h_0+\alpha_0)+(h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0)x^k\\&\quad+(\beta_0+\gamma_1-\alpha_1-\beta_1)x^{2k}+\beta_1\cdot x^{3k},\end{aligned} \tag{4.4}$$

which we notice is exactly the same as (4.3) with the addition of $h_0+h_1x^k$ as specified by Condition 4.1.

The base case of the algorithm will be to switch over to the classical algorithm for multiplication; the exact size at which the classical algorithm should be preferred will depend on the implementation. Even with the extra conditions of our problem, the classical algorithm can still be implemented with $O(1)$ space, using a single accumulator in temporary memory, as discussed earlier. This proves the correctness of the following theorem:

Theorem 4.3. *Let $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ be polynomials each with degree one less than $n = 2^b$ for some $b \in \mathbb{N}$, with all polynomials stored in read-only input memory except h, which is stored initially in the output space. Algorithm 4.1 correctly computes $h + (f^{(0)} + f^{(1)}) \cdot g$ using $O(1)$ temporary storage on the algebraic side and $O(\log n)$ temporary storage of word-sized integers. The time complexity is $O(n^{\log_2 3})$, or $O(n^{1.59})$.*

Proof. The size of the input polynomials n must be a power of 2 so that each recursive call is on even-sized arguments. To be more precise, the algorithm should be modified so that it initially checks whether the input size is below a certain (constant) threshold, in which case the space-efficient classical algorithm is called. Correctness follows from the discussion above. We can see that there are exactly three recursive calls on input of one-half the size of the original input, and this gives the stated time complexity bounds.

Finally, observe that the temporary storage required by the algorithm on a single recursive call consists of a constant number of pointers, stored on the integer side of the algebraic

IMM, and a single accumulator on the algebraic side. This accumulator can be re-used by the recursive calls, but the pointers must remain, so that the algorithm may proceed when the recursive calls terminate. Since the depth of recursion is $\log_2 n$, this means that $O(\log n)$ temporary space is required on the integer side, as stated. □

4.2.2 Initial calls

The initial call to compute the product of two polynomials f and g will not satisfy Conditions 4.1 and 4.2 above. So there must be top-level versions of the algorithm which do *not* solve the more general problem of $h + (f^{(0)} + f^{(1)}) \cdot g$.

First, denote by SE_KarMult_1+2 Algorithm 4.1, indicating that both conditions 1 and 2 are satisfied by this version.

Working backwards, we then denote by SE_KarMult_1 an algorithm that is similar to Algorithm 4.1, but which does not satisfy Condition 4.2. Namely, the input will be just three polynomials $f, g, h \in \mathsf{R}[x]$, with h stored in the output space, and the algorithm computes $h + f \cdot g$. In this version, two of the additions on Step 2 are eliminated. The function call on Step 3 is still to SE_KarMult_1+2, but the other two on Steps 5 and 8 are recursive calls to SE_KarMult_1.

Similarly, SE_KarMult will be the name of the top-level call which does not satisfy either Condition 4.1 or 4.2, and therefore simply computes a single product $f \cdot g$ into an uninitialized output space. Here again we save two array additions on Step 2, and in addition Step 1 is replaced with the instruction:

$$D[0..k-1] \leftarrow C[0..k-1] + C[k..2k-1].$$

This allows the first two function calls on Steps 3 and 5 to be recursive calls to SE_KarMult, and only the last one on Step 8 is a call to SE_KarMult_1.

The number of additions (and subtractions) at each recursive step determine the hidden constant in the $O(n^{1.59})$ time complexity measure. We mentioned that a naïve implementation of Karatsuba's algorithm uses $4n$ additions at each recursive step, and it is easy to improve this to $7n/2$. By inspection, Algorithm 4.1 uses $9n/2 + O(1)$ additions at each recursive step, and therefore both SE_KarMult_1 and SE_KarMult use only $7n/2 + O(1)$ additions at each recursive step. So SE_KarMult_1 and SE_KarMult match the best known existing algorithms in the number of additions required, and SE_KarMult_1+2 uses only $n + O(1)$ more additions and subtractions than this.

Asymptotically, the cost of calls to SE_KarMult_1+2 will eventually dominate, incurring a slight penalty in extra arithmetic operations. However, most of the initial calls, particularly for smaller values of n (where Karatsuba's algorithm is actually used), will be to SE_KarMult_1 or SE_KarMult, and should not be any more costly in time than a good existing implementation. This gives us hope that our space-efficient algorithm might be useful in practice; this hope is confirmed in Section 4.4.

4.2.3 Unequal and odd-sized operands

So far our algorithm only works when both input polynomials have the same degree, and that degree is exactly one less than a power of two. This is necessary because the size of both operands at each recursive call to Algorithm 4.1, as stated, must be even. Special cases to handle the cases when the input polynomials have even degree (and therefore an odd size) or different degrees will resolve these issues and give a general-purpose multiplication algorithm.

First consider the case that $\deg f^{(0)}$, $\deg f^{(1)}$, $\deg g$, and $\deg h$ are all equal to $2k$ for some $k \in \mathbb{N}$, so that each polynomial has an odd number of coefficients. In order to use Algorithm 4.1 in this case, we first pull off the low-order coefficients of each polynomial, writing

$$\begin{aligned}
f^{(0)} &= a_0 + x\hat{f}^{(0)} \\
f^{(1)} &= b_0 + x\hat{f}^{(1)} \\
g &= c_0 + x\hat{g} \\
h &= d_0 + d_1 x + x^2\hat{h}
\end{aligned}$$

Now we can rewrite the desired result as follows:

$$\begin{aligned}
h + (f^{(0)} + f^{(1)}) \cdot g &= d_0 + d_1 x + x^2\hat{h} + (a_0 + b_0 + x\hat{f}^{(0)} + x\hat{f}^{(1)}) \cdot (c_0 + x\hat{g}) \\
&= d_0 + d_1 x + x(a_0 + b_0)\hat{g} + x c_0 \hat{f}^{(0)} + x c_0 \hat{f}^{(1)} + x^2(\hat{h} + (\hat{f}^{(0)} + \hat{f}^{(1)}) \cdot \hat{g}).
\end{aligned}$$

Therefore this result can be computed with a single call to Algorithm 4.1 with even-length arguments, $\hat{h} + (\widehat{f^{(0)}} + \widehat{f^{(1)}}) \cdot \hat{g}$, followed by three additions of a scalar times a polynomial, and a few more scalar products and additions. Since we can multiply a polynomial by a scalar and add to a pre-initialized result without using any extra space, this still achieves the same asymptotic time and space complexity as before, albeit with a few extra arithmetic operations.

In fact, our implementation in the MVP library uses only $n/2 + O(1)$ more operations at each recursive step than the power-of-two algorithm analysed earlier, by making careful use of additional versions of these routines when the size of the operands differ by exactly one. These "one-different" versions have the same structure and asymptotic cost as the routines already discussed, so we will not give further details here.

If the degrees of f and g differ by more than one, we use the well-known trick of blocking the larger polynomial into sub-arrays whose length equal the degree of the smaller one. That is, given $f, g \in \mathsf{R}[x]$ with $n = \deg f$, $m = \deg g$ and $n > m$, we write $n = qm + r$ and reduce the problem to computing q products of a degree-m by a degree-$(m-1)$ polynomial and one product of a degree-m by a degree-$(r-1)$ polynomial. The m by $m-1$ products are handled with the "one-different" versions of the algorithms mentioned above.

Each of the q initial products overlap in exactly m coefficients (half the size of the output), so Condition 4.1 actually works quite naturally here, and the q m-by-$(m-1)$ products are calls to a version of `SE_KarMult_1`. The single m-by-$(r-1)$ product is performed *first* (so that the entire output is uninitialized), and this is done by a recursive call to this same procedure multiplying arbitrary unequal-length polynomials.

All these special cases give the following result, which is the first algorithm for dense univariate multiplication to achieve sub-quadratic time times space.

Theorem 4.4. *For any $f, g \in \mathsf{R}[x]$ with degrees less than n, the product $f \cdot g$ can be computed using $O(n^{\log_2 3})$ ring operations, $O(1)$ temporary storage of ring elements, and $O(\log n)$ temporary storage of word-sized integers.*

4.2.4 Rolling a log from space to time

In fact, the $O(\log n)$ temporary storage of integers can be avoided entirely, at the cost of an extra logarithmic factor in the time cost. The idea is to store the current position of the execution in the recursion tree, by a list of integers $(r_0, r_1, \ldots)$. Each r_i is 0, 1, or 2, indicating which of the three recursive calls to the algorithm has been taken on level i of the recursion. Since the recursion depth is at most $\log_2 n$, this list can be encoded into a single integer R with at most $(\log_2 3)\log_2 n$ bits. This is bounded by $O(\log n)$ bits, and therefore the IMM algorithm can choose the word size w appropriately to store R in a single word. Each time the ith recursive call is made, R is updated to $3R + i$, and each time a call returns, R is reset to $\lfloor R/3 \rfloor$. With the position in the recursion tree thus stored in a single machine word, each recursive call overwrites the pointers from the parent call, using only $O(1)$ temporary space throughout the computation.

But when each recursive call returns, the algorithm will not know what parts of memory to operate on, since the parent call's pointers have been overwritten! This is solved by using the encoded list of r_i's, simulating the same series of calls from the top level to recover the pointers used by the parent call. Such a simulation can always be performed, even though the values in the memory locations are different than at the start of the algorithm, since the algorithm is "oblivious". That is, its recursive structure does not depend on the values in memory, but only on their size. The cost of this simulation is $O(\log n)$ at each stage, increasing the time complexity to $O(n^{\log_2 3} \log n)$. Somewhat dishonestly, we point out that this is still actually $O(n^{1.59})$, since 1.59 is strictly greater than $\log_2 3$.

4.3 Space-efficient FFT-based multiplication

We now show how to perform FFT-based multiplication in $O(n \log n)$ time and using only $O(1)$ temporary space, when the coefficient ring R already contains the necessary primitive root of unity. A crucial subroutine will be our in-place truncated Fourier transform algorithm from Chapter 3.

Recall the notation for PRUs introduced in subsection 3.1.6 of the previous chapter: $\omega_{[k]}$ is a 2_k-PRU in R for any k small enough that R actually contains such a PRU, and ω_i is defined as $\omega_{[k]}^{\mathrm{rev}_k i}$.

For polynomials $f, g \in \mathsf{R}[x]$, write $n = \deg f + \deg g - 1$ for the number of coefficients in their product. The multiplication problem we consider is as follows. Given f, g, and a 2^k-PRU $\omega_{[k]}$, with $k \geq \log_2 n$, compute the coefficients of the product $h = f \cdot g$. As usual, the coefficients of f and g are stored in read-only input space along with the PRU $\omega_{[k]}$, and the coefficients of h must be written to the size-n output space.

A straightforward implementation of the standard FFT-based multiplication algorithm outlined at the start of this chapter would require $O(n)$ temporary memory to compute this product. Our approach avoids the need for this temporary memory by computing entirely in the output space. In summary, we first compute at least one-third of $\mathrm{DFT}(f \cdot g)$, then at least one-third of what remains, and so forth until the operation is complete. A single in-place inverse TFT then completes the algorithm.

4.3.1 Folded polynomials

The initial stages of the algorithm compute partial DFTs of each of f and g, then multiply them to obtain a partial DFT of h. A naïve approach to computing these partial transforms would be to compute the entire DFT and then discard the unneeded parts, but this would violate both the time and space requirements of our algorithm.

Instead, we define the *folded polynomials* as smaller polynomials determined from each of f, g, h whose ordinary, full-length DFTs correspond to contiguous subsequences of the DFTs of f, g, h.

Definition 4.5. *For any $u \in \mathsf{R}[x]$, and any $i, j \in \mathbb{N}$, the* folded polynomial $u_{a,b} \in \mathsf{R}[x]$ *is the polynomial with degree less than 2^b given by*

$$u_{a,b}(x) = u\,(\omega_a \cdot x) \operatorname{rem} x^{2^b} - 1.$$

To see how to compute the folded polynomials, first write $u = \sum_{i \ge 0} c_i x^i$. Then

$$u_{a,b} = u(\omega_a x) \operatorname{rem} x^{2^b} - 1 = \sum_{i \ge 0} c_i \omega_a^i x^{i \operatorname{rem} 2^b}.$$

Using this formulation, we can compute $u_{a,b}$ using $O(\deg u)$ ring operations, and using only the storage space for the result, plus a single temporary ring element, an accumulator for the powers of ω_a.

The usefulness of the folded polynomials is illustrated in the following lemma, which shows the relationship between $\mathrm{DFT}(u)$ and $\mathrm{DFT}(u_{a,b})$.

Lemma 4.6. *For any $u \in \mathsf{R}[x]$ and $a, b \in \mathbb{N}$ such that $2^b \mid a$, the elements of the discrete Fourier transform of $u_{a,b}$ at $\omega_{[b]}$ are exactly*

$$\mathrm{DFT}_{\omega_{[b]}}(u_{a,b}) = u(\omega_a), u(\omega_{a+1}), \ldots, u(\omega_{a+2^b-1}).$$

Proof. Let $s \in \{0, 1, \ldots, 2^b - 1\}$. Then ω_s can be written as $\omega_{[b]}^{\mathrm{rev}_b s}$, from the definitions. Therefore, we see that ω_s is a 2^b-PRU, and furthermore, $\omega_s^{2^b} - 1 = 0$ in R. This implies that $u_{a,b}(\omega_s) = u(\omega_a \omega_s)$.

Now since $2^b \mid a$, a and s have no bits in common, so for any $k > \log_2 a$, we have that $\operatorname{rev}_k(a+s) = \operatorname{rev}_k a + \operatorname{rev}_k s$, and therefore, from the definitions,

$$\omega_a \omega_s = \omega_{[k]}^{\operatorname{rev}_k a + \operatorname{rev}_k s} = \omega_{a+s}.$$

Putting this together, we see that

$$\begin{aligned}\mathrm{DFT}_{\omega_{[b]}}(u_{a,b}) &= u_{a,b}(\omega_0), u_{a,b}(\omega_1), \ldots, u_{a,b}(\omega_{2^b-1})\\ &= u(\omega_a), u(\omega_{a+1}), \ldots, u(\omega_{a+2^b-1}).\end{aligned}$$ □

4.3.2 Constant-space algorithm

Our strategy for FFT-based multiplication with $O(1)$ extra space is then to compute

$$\mathrm{DFT}_{\omega_{[b_i]}}(h_{a_i,b_i})$$

for a sequence of indices a_i and b_i satisfying $a_0 = 0$ and $a_i = a_{i-1} + 2^{b_{i-1}}$ for $i \geq 1$. From Lemma 4.6 above, this sequence will give the DFT of h itself after enough iterations. The b_i's at each step will be chosen so that the computation of that step can be performed entirely in the output space.

This approach is presented in Algorithm 4.2. The input consists of two arrays $A, B \in \mathsf{R}^*$, where $A[i]$ is the coefficient of x^i in f and $B[i]$ is the coefficient of x^i in g. The algorithm uses the output space of size $\deg f + \deg g + 1$ and ultimately writes the coefficients of the product $h = fg$ in that space. The subroutine `FFT` indicates the usual radix-2 in-place FFT algorithm on a power-of-two input size, and the subroutine `InplaceITFT` is Algorithm 3.2 from the previous chapter.

Theorem 4.7. *Algorithm 4.2 correctly computes the product $h = f \cdot g$, using $O(n \log n)$ ring operations and $O(1)$ temporary storage of ring elements and word-sized integers.*

Proof. The main loop terminates since q is strictly increasing. Let m be the number of iterations, and let $q_0 > q_1 > \cdots > q_{m-1}$ and $L_0 \geq L_1 \geq \cdots \geq L_{m-1}$ be the values of q and L on each iteration. By construction, the intervals $[q_i, q_i + L_i)$ form a partition of $[0, r-1)$, and L_i is the largest power of two such that $q_i + 2L_i \leq r$. Therefore each L can appear at most twice (i.e., if $L_i = L_{i-1}$ then $L_{i+1} < L_i$) Furthermore, $m \leq 2 \lg r$, and we have $L_i \mid q_i$ for each i.

At each iteration, lines 7–8 compute the coefficients of the folded polynomial $f_{q,\ell}$, placing the result in $M_O[q, \ldots, q+L-1]$. From Lemma 4.6, we know that the FFT on Line 9 then computes $M_O[q+i] = f(\omega_{q+i})$ for $0 \leq i < L$. The next two lines similarly compute $M_O[q+L+i] = g(\omega_{q+i})$ for $0 \leq i < L$. (The condition $q + 2L \leq r$ ensures that both of these transforms fit into the output space.) Lines 13–14 then compute $M_O[q+i] = f(\omega_{q+i}) \cdot g(\omega_{q+i}) = h(\omega_{q+i})$ for $0 \leq i < L$. These are the corresponding elements of the discrete Fourier transform of h, written in reverted binary order.

After line 16 we finally have $M_O[i] = h(\omega_i)$ for all $0 \leq i < n$. Observe that the last element is handled specially since the output space does not have room for both evaluations. The call to Algorithm 3.2 on Line 17 then recovers the coefficients of h, in the normal order.

Algorithm 4.2: Space-efficient FFT-based multiplication

```
Input: $f, g \in \mathsf{R}[x]$, stored in arrays $A$ and $B$, respectively
Output: The coefficients of $h = f \cdot g$, stored in output space $M_O$
$n \leftarrow \deg f + \deg g - 1$
$q \leftarrow 0$
while $q < n - 1$ do
    $\ell \leftarrow \lfloor \lg(n-q) \rfloor - 1$
    $L \leftarrow 2^{\ell}$
    $M_O[q, q+1, \dots, q+2L-1] \leftarrow (0, 0, \dots, 0)$
    for $0 \le i \le \deg f$ do
        $M_O[q + (i \operatorname{rem} L)] \leftarrow M_O[q + (i \operatorname{rem} L)] + \omega_q^i A[i]$
    FFT($M_O[q, q+1, \dots, q+L-1]$)
    for $0 \le i \le \deg g$ do
        $M_O[q + L + (i \operatorname{rem} L)] \leftarrow M_O[q + L + (i \operatorname{rem} L)] + \omega_q^i B[i]$
    FFT($M_O[q+L, q+L+1, \dots, q+2L-1]$)
    for $0 \le i < L$ do
        $M_O[q+i] \leftarrow M_O[q+i] \cdot M_O[q+L+i]$
    $q \leftarrow q + L$
$M_O[n-1] \leftarrow f(\omega_{n-1}) \cdot g(\omega_{n-1})$
InplaceITFT($M_O[0, 1, \dots, n-1]$)
```

We now analyse the time and space complexity. Computing the folded polynomials in the loops on lines 6, 7, 10 and 13 takes $O(n)$ operations per iteration, or $O(n \log n)$ in total, since $m = O(\log n)$. The FFT calls contribute $O(L_i \log L_i)$ per iteration, for a total of

$$O(\sum_i L_i \log L_i) = O(\sum_i L_i \log L) = O(n \log n).$$

Line 16 requires $O(n)$ ring operations, and line 17 requires $O(n \log n)$. The space requirements follow directly from the fact that the FFT and InplaceTFT calls are all performed in-place. □

4.4 Implementation

These algorithms have been implemented in the MVP library for dense polynomial multiplication over prime fields $\mathbb{F}_p[x]$ with p a single machine word-sized prime. For the FFT-based multiplication method, we further require that $\mathbb{F}_p$ contains 2^k-PRU of sufficiently high order, as mentioned above.

For benchmarking, we compared our algorithms with the C++ library NTL (Shoup, 2009). This is a useful comparison, as NTL also relies on a blend of classical, Karatsuba, and FFT-based algorithms for different ranges of input sizes, using the standard Karatsuba and FFT algorithms we presented earlier. There are some faster implementations of modular multiplication, most notably zn_poly (Harvey, 2008), but these use somewhat different algorithms and methods. We also used the Mod<long> class in MVP for the coefficient arithmetic, as opposed to the (faster) Montgomery representation mentioned earlier, because this is essentially the same as NTL's implementation for coefficient arithmetic. Our aim is not to claim the "fastest" implementation but merely to demonstrate that the space-efficient algorithms presented can compete in time cost with other methods that use more space. By comparing against a similar library with similar coefficient arithmetic, we hope to gain insight about the algorithms themselves.

Table 4.2 shows the results of some early benchmarking tests. For all cases, we multiplied two randomly-chosen dense polynomials of the given degree, and the time per iteration, in CPU seconds, is reported. NTL uses classical multiplication for degree less than 16 (and as a base case for Karatsuba), Karatsuba's algorithm for size up to 512, and FFT-based multiplication for the largest sizes. Our crossover points, determined experimentally on the target machine, are a little higher: 32 and 768 respectively. This is probably due to the slightly higher complication of our new algorithms, but there could also be other factors at play.

We should also point out that we used the radix-2 not-in-place inverse FFT for the final step of Algorithm 4.2, rather than the in-place truncated Fourier transform of the previous chapter. This choice was made because of the especially bad performance of the in-place inverse TFT in our experiments before. Observe that when the size of the product is a power of two, our algorithm is actually in-place, and even this is better than previously-known approaches in terms of space efficiency.

Size	NTL	Low-space in MVP	Ratio
100	4.00×10^{-5}	3.00×10^{-5}	.750
150	7.94×10^{-5}	6.24×10^{-5}	.786
200	1.26×10^{-4}	8.70×10^{-5}	.690
250	1.70×10^{-4}	1.17×10^{-4}	.688
300	2.35×10^{-4}	1.88×10^{-4}	.800
350	2.97×10^{-4}	2.24×10^{-4}	.754
400	3.61×10^{-4}	2.60×10^{-4}	.720
450	4.34×10^{-4}	3.10×10^{-4}	.714
500	5.02×10^{-4}	3.50×10^{-4}	.697
600	7.04×10^{-4}	5.68×10^{-4}	.807
700	8.32×10^{-4}	6.96×10^{-4}	.837
800	8.42×10^{-4}	7.78×10^{-4}	.924
900	8.48×10^{-4}	8.24×10^{-4}	.972
1000	8.74×10^{-4}	8.34×10^{-4}	.954
1200	1.70×10^{-3}	1.54×10^{-3}	.906
1400	1.77×10^{-3}	1.59×10^{-3}	.898
1600	1.80×10^{-3}	1.70×10^{-3}	.944
1800	1.90×10^{-3}	1.76×10^{-3}	.926
2000	1.95×10^{-3}	1.88×10^{-3}	.964
3000	4.07×10^{-3}	3.58×10^{-3}	.879
4000	4.22×10^{-3}	4.04×10^{-3}	.957
5000	1.05×10^{-2}	7.38×10^{-3}	.703
6000	1.06×10^{-2}	8.00×10^{-3}	.755
7000	1.09×10^{-2}	8.44×10^{-3}	.774
8000	1.10×10^{-2}	8.80×10^{-3}	.800
9000	2.28×10^{-2}	1.59×10^{-2}	.697

Table 4.2: Benchmarks versus NTL

4.5 Conclusions

The two algorithms presented in this chapter match existing “fast” algorithms in asymptotic time complexity, but need considerably less auxiliary storage space to compute the result. Our algorithms are novel theoretically as the first methods to achieve less than quadratic time × space for multiplication. This is achieved in part by working in the more permissive (and realistic) IMM model.

Much work remains to be done on this topic. The most obvious question is how these algorithms can be adapted to multi-precision integer multiplication. The FFT-based algorithm can already be used directly as a subroutine, since integer multiplication algorithms that rely on the FFT generally work over modular rings that contain the required PRUs. A straightforward adaptation of the space-efficient Karatsuba multiplication to the multiplication of multi-precision integers is not difficult to imagine, but the extra challenges introduced by the presence of carries, combined with the extreme efficiency of existing libraries such as GMP (Granlund et al., 2010), mean that an even more careful implementation would be needed to gain an advantage in this case. For instance, it would probably be better to use a subtractive version of Karatsuba’s algorithm to avoid some carries. This is also likely the area of greatest potential utility of our new algorithms, as routines for long integer multiplication are used in many different areas of computing.

There are also some more theoretical questions left open here. One direction for further research would be to see if a scheme similar to the one presented here for Karatsuba-like multiplication with low space requirements could also be adapted to the Toom-Cook 3-way divide-and-conquer method, or even their arbitrary k-way scheme (Toom, 1963; Cook, 1966). Some of these algorithms are actually used in practice for a range of input sizes between Karatsuba and FFT-based methods, so there is a potential practical application here. Another task for future research would be to improve the Karatsuba-like algorithm even further, either by reducing the amount of extra storage below $O(\log n)$ or (more usefully) reducing the implied constant in the $O(n^{1.59})$ time complexity.

Finally, a natural question is to ask whether polynomial or long integer multiplication can be done completely in-place. We observe that the size of the input (counting the coefficients of both multiplicands) is roughly the same as the size of the output, so asking for an in-place transformation overwriting the input with the output seems plausible. Upon further reflection, it seems that the data dependencies will make this task impossible, but we have no proof of this at the moment.

> Everybody who has analyzed the logical theory of computers has come to the conclusion that the possibilities of computers are very interesting — if they could be made to be more complicated by several orders of magnitude. If they had millions of times as many elements, they could make judgments. They would have time to calculate what is the best way to make the calculation that they are about to make.
>
> —Richard Feynman, "There's plenty of room at the bottom" (1959)

Chapter 5

Adaptive multiplication

As we have seen, the multiplication of univariate polynomials is one of the most important operations in mathematical computing, and in computer algebra systems in particular. However, existing algorithms are closely tied to the standard dense and sparse representations. In this chapter, we present new algorithms for multiplication which allow the operands to be stored in either the dense or sparse representation, and automatically adapt the algorithm to the sparse structure of the input polynomials (in some sense). The result is a method that is never significantly slower than any existing algorithm, but is significantly faster for a large class of inputs.

Preliminary progress on some of these problems was presented at the Milestones in Computer Algebra (MICA) conference in 2008 (Roche, 2008). The algorithms presented here have also been accepted to appear in the Journal of Symbolic Computation (Roche, 2010).

5.1 Background

We start by reviewing the dense and sparse representations, as well as corresponding algorithms for multiplication that have been developed. Let $f \in \mathsf{R}[x]$ with degree less than n written as

$$f = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}, \tag{5.1}$$

for $c_0, \ldots, c_{n-1} \in \mathsf{R}$. Recall that the dense representation of f is simply an array $[c_0, c_1, \ldots, c_{n-1}]$ of length n containing every coefficient in f.

Now suppose that at most t of the coefficients are nonzero, so that we can write

$$f = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_t x^{e_t}, \tag{5.2}$$

for $a_1,\ldots,a_t \in \mathsf{R}$ and $0 \le e_1 < \cdots < e_t$. Hence $a_i = c_{e_i}$ for $1 \le i \le t$, and in particular $e_t = \deg f$. The sparse representation of f is a list of coefficient-exponent tuples $(a_1,e_1),\ldots,(a_t,e_t)$. This always requires $O(t)$ ring elements of storage (on the algebraic side of an algebraic IMM). However, the exponents in this case could be multi-precision integers, and the total size of their storage, on the integer side, is proportional to $\sum_i (1+\log_2 e_i)$ bits. Assuming every exponent is stored to the same precision, this is $O(t \log n)$ bits in total. Since the integer exponents are not stored in bits but in machine words, this can be reduced by a logarithmic factor for scalable IMM problems, but we will ignore that technicality for the sake of clarity in the present discussion.

The previous chapter discussed algorithms for dense polynomial multiplication in depth. Recall that the $O(n^2)$ school method was first improved by Karatsuba and Ofman (1963) to $O(n^{1.59})$ with a two-way divide-and-conquer scheme, later generalized to k-way by Toom (1963) and Cook (1966). Building on results of Schönhage and Strassen (1971) and Schönhage (1977), Cantor and Kaltofen (1991) developed an algorithm to multiply polynomials with $O(n \log n \operatorname{loglog} n)$ operations in any algebra. This stands as the best asymptotic complexity for polynomial multiplication.

In practice, all of these algorithms will be used in certain ranges, and so we employ the usual notation of a *multiplication time* function $\mathsf{M}(n)$, the cost of multiplying two dense polynomials with degrees both less than n. Also define $\delta(n) = \mathsf{M}(n)/n$. If $f, g \in \mathsf{R}[x]$ with different degrees $\deg f < n$, $\deg g < m$, and $n > m$, by splitting f into $\lceil n/m \rceil$ size-m blocks we can compute the product $f \cdot g$ with cost $O(\frac{n}{m}\mathsf{M}(m))$, or $O(n \cdot \delta(m))$.

Algorithms for sparse polynomial multiplication were briefly discussed in Chapter 2. In this case, the school method also uses $O(t^2)$ ring operations, but for sparse polynomials this cannot be improved in the worst case. However, since the degrees could be very large, the cost of exponent arithmetic becomes significant. The school method uses $O(t^3 \log n)$ word operations and $O(t^2)$ space. Yan (1998) reduces the number of word operations to $O(t^2 \log t \log n)$ with the “geobuckets” data structure. Finally, recent work by Monagan and Pearce (2007), following Johnson (1974), gets this same time complexity but reduces the space requirement to $O(t+r)$, where r is the number of nonzero terms in the product.

The algorithms we present are for univariate polynomials. They can also be used for multivariate polynomial multiplication by using *Kronecker substitution*, as described previously in section 1.2.4: Given two n-variate polynomials $f, g \in \mathsf{R}[x_1,\ldots,x_n]$ with max degrees less than d, substitute $x_i = y^{(2d)^{i-1}}$ for $1 \le i \le n$, multiply the univariate polynomials over $\mathsf{R}[y]$, then convert back. Of course there are many other ways to represent multivariate polynomials, but we will not consider them further here.

5.2 Overview of Approach

The performance of an *adaptive algorithm* depends not only on the size of the input but also on some inherent difficulty measure. Such algorithms match standard approaches in their worst-case performance, but perform far better on many instances. This idea was first applied to sorting algorithms and has proved to be useful both in theory and in practice (see Petersson and Moffat, 1995). Such techniques have also proven useful in symbolic computation, for example the early termination strategy of Kaltofen and Lee (2003).

Hybrid algorithms combine multiple different approaches to the same problem to effectively handle more cases (e.g., Duran, Saunders, and Wan, 2003). Our algorithms are also hybrid in the sense that they provide a smooth gradient between existing sparse and dense multiplication algorithms. The adaptive nature of the algorithms means that in fact they will be faster than existing algorithms in many cases, while never being (asymptotically) slower.

The algorithms we present will always proceed in three stages. First, the polynomials are read in and converted to a different representation which effectively captures the relevant measure of difficulty. Second, we multiply the two polynomials in the alternate representation. Finally, the product is converted back to the original representation.

The second step is where the multiplication is actually performed, and it is this step alone that depends on the difficulty of the particular instance. Therefore this step should be the dominating cost of the entire algorithm, and in particular the cost of the conversion steps must be linear in the size of the input polynomials.

In Section 5.3, we give the first idea for adaptive multiplication, which is to write a polynomial as a list of dense "chunks". The second idea, presented in Section 5.4, is to write a polynomial with "equal spacing" between coefficients as a dense polynomial composed with a power of the indeterminate. Section 5.5 shows how to combine these two ideas to make one algorithm which effectively captures both difficulty measures. We examine how our algorithms perform in practice with some benchmarking against existing algorithms in Section 5.6. Finally, a few conclusions and ideas for future directions are discussed in Section 5.7.

5.3 Chunky Polynomials

The basic idea of chunky multiplication is a straightforward combination of the standard sparse and dense representations, providing a natural gradient between the two approaches for multiplication. We note that a similar idea was noticed (independently) around the same time by Fateman (2008, page 11), although the treatment here is much more extensive.

For $f \in \mathsf{R}[x]$ of degree n, the *chunky representation* of f is a sparse polynomial with dense polynomial "chunks" as coefficients:

$$f = f_1 x^{e_1} + f_2 x^{e_2} + \cdots + f_t x^{e_t}, \tag{5.3}$$

with $f_i \in \mathsf{R}[x]$ and $e_i \in \mathbb{N}$ for each $1 \le i \le t$. We require only that $e_{i+1} > e_i + \deg f_i$ for $1 \le i \le t-1$, and each f_i has nonzero constant coefficient.

Recall the notation introduced above of $\delta(n) = \mathsf{M}(n)/n$. A unique feature of our approach is that we will actually use this function to tune the algorithm. That is, we assume a subroutine is given to evaluate $\delta(n)$ for any chosen value n.

If n is a word-sized integer, then the computation of $\delta(n)$ must use a constant number of word operations. If n is more than word-sized, then we are asking about the cost of multiplying two dense polynomials that cannot fit in memory, so the subroutine should return ∞ in such cases. Practically speaking, the $\delta(n)$ evaluation will usually be an approximation of the actual value, but for what follows we assume the computed value is always exactly correct.

Furthermore, we require $\delta(n)$ to be an increasing function which grows more slowly than linearly, meaning that for any $a, b, d \in \mathbb{N}$ with $a < b$,

$$\delta(a+d) - \delta(a) \geq \delta(b+d) - \delta(b). \tag{5.4}$$

These conditions are clearly satisfied for all the dense multiplication algorithms and corresponding $\mathsf{M}(n)$ functions discussed above, including the piecewise function used in practice.

The conversion of a sparse or dense polynomial to the chunky representation proceeds in two stages: first, we compute an "optimal chunk size" k, and then we use this computed value as a parameter in the actual conversion algorithm. The product of the two polynomials is then computed in the chunky representation, and finally the result is converted back to the original representation. The steps are presented in reverse order in the hope that the goals at each stage are more clear.

5.3.1 Multiplication in the chunky representation

Multiplying polynomials in the chunky representation uses sparse multiplication on the outer loop, treating each dense polynomial chunk as a coefficient, and dense multiplication to find each product of two chunks.

For $f, g \in \mathsf{R}[x]$ to be multiplied, write f as in (5.3) and g as

$$g = g_1 x^{d_1} + g_2 x^{d_2} + \cdots + g_s x^{d_s}, \tag{5.5}$$

with $s \in \mathbb{N}$ and similar conditions on each $g_i \in \mathsf{R}[x]$ and $d_i \in \mathbb{N}$ as in (5.3). Without loss of generality, assume also that $t \geq s$, that is, f has more chunks than g. To multiply f and g, we need to compute each product $f_i g_j$ for $1 \leq i \leq t$ and $1 \leq j \leq s$ and put the resulting chunks into sorted order. It is likely that some of the chunk products will overlap, and hence some coefficients will also need to be summed.

By using heaps of pointers as in (Monagan and Pearce, 2007), the chunks of the result are computed in order, eliminating unnecessary additions and using little extra space. A min-heap of size s is filled with pairs (i, j), for $i, j \in \mathbb{N}$, and ordered by the corresponding sum of exponents $e_i + d_j$. Each time we compute a new chunk product $f_i \cdot g_j$, we check the new exponent against the degree of the previous chunk, in order to determine whether to make a new chunk in the product or add to the previous one. The details of this approach are given in Algorithm 5.1.

After using this algorithm to multiply f and g, we can easily convert the result back to the dense or sparse representation in linear time. In fact, if the output is dense, we can preallocate space for the result and store the computed product directly in the dense array, requiring only some extra space for the heap H and a single intermediate product h_{new}.

Algorithm 5.1: Chunky Multiplication

```
   Input: $f, g$ as in (5.3) and (5.5)
   Output: The product $f \cdot g = h$ in the chunky representation
1  $\alpha \leftarrow f_1 \cdot g_1$ using dense multiplication
2  $b \leftarrow e_1 + d_1$
3  $H \leftarrow$ min-heap with pairs $(1, j)$ for $j = 2, 3, \ldots, s$, ordered by exponent sums
4  if $i \geq 2$ then insert $(2, 1)$ into $H$
5  while $H$ is not empty do
6      $(i, j) \leftarrow$ extract-min($H$)
7      $\beta \leftarrow f_i \cdot g_j$ using dense multiplication
8      if $b + \deg \alpha < e_i + d_j$ then
9          write $\alpha x^b$ as next term of $h$
10         $\alpha \leftarrow \beta$;   $b \leftarrow e_i + d_j$
11     else $\alpha \leftarrow \alpha + \beta x^{e_i + d_j - b}$ stored as a dense polynomial
12     if $i < t$ then insert $(i + 1, j)$ into $H$
13 write $\alpha x^b$ as final term of $h$
```

Theorem 5.1. *Algorithm 5.1 correctly computes the product of f and g using*

$$O\left(\sum_{\substack{\deg f_i \geq \deg g_j \\ 1 \leq i \leq t,\ 1 \leq j \leq s}} (\deg f_i) \cdot \delta(\deg g_j) \quad + \sum_{\substack{\deg f_i < \deg g_j \\ 1 \leq i \leq t,\ 1 \leq j \leq s}} (\deg g_j) \cdot \delta(\deg f_i) \right)$$

ring operations and $O(ts \cdot \log s \cdot \log(\deg fg))$ word operations.

Proof. Correctness is clear from the definitions. The bound on ring operations comes from Step 7 using the fact that $\delta(n) = \mathsf{M}(n)/n$. The cost of additions on Step 11 is linear and hence also within the stated bound.

The cost of word operations is incurred in removing from and inserting to the heap on Steps 6 and 12. Because these steps are executed no more than $t_f t_g$ times, the size of the heap is never more than t_g, and each exponent sum is bounded by the degree of the product, the stated bound is correct. □

Notice that the cost of word operations is always less than the cost would be if we had multiplied f and g in the standard sparse representation. We therefore focus only on minimizing the number of ring operations in the conversion steps that follow.

5.3.2 Conversion given optimal chunk size

The general chunky conversion problem is, given $f, g \in \mathsf{R}[x]$, both either in the sparse or dense representation, to determine chunky representations for f and g which minimize the cost of Algorithm 5.1. Here we consider a simpler problem, namely determining an optimal chunky representation for f given that g has only one chunk of size k.

The following corollary comes directly from Theorem 5.1 and will guide our conversion algorithm on this step.

Corollary 5.2. *Given $f \in \mathsf{R}[x]$ as in (5.3), the number of ring operations required to multiply f by a single dense polynomial with degree less than k is*

$$O\left(\delta(k)\sum_{\deg f_i \geq k} \deg f_i \quad + \quad k\sum_{\deg f_i < k} \delta(\deg f_i)\right)$$

For any high-degree chunk (i.e., $\deg f_i \geq k$), we see that there is no benefit to making the chunk any larger, as the cost is proportional to the sum of the degrees of these chunks. In order to minimize the cost of multiplication, then, we should not have any chunks with degree greater than k (except possibly in the case that every coefficient of the chunk is nonzero), and we should minimize $\sum \delta(\deg f_i)$ for all chunks with size less than k.

These observations form the basis of our approach in Algorithm 5.2 below. For an input polynomial $f \in \mathsf{R}[x]$, each "gap" of consecutive zero coefficients in f is examined, in order. We determine the optimal chunky conversion if the polynomial were truncated at that gap. This is accomplished by finding the previous gap of highest degree that should be included in the optimal chunky representation. We already have the conversion for the polynomial up to that gap (from a previous step), so we simply add on the last chunk and we are done. At the end, after all gaps have been examined, we have the optimal conversion for the entire polynomial.

Let $a_i, b_i \in \mathbb{Z}$ for $0 \leq i \leq m$ be the sizes of each consecutive "gap" of zero coefficients and "block" of nonzero coefficients, in order. Each a_i and b_i will be nonzero except possibly for a_0 (if f has a nonzero constant coefficient), and $\sum_{0 \leq i \leq m}(a_i + b_i) = \deg f + 1$. For example, the polynomial

$$f = 5x^{10} + 3x^{11} + 9x^{13} + 20x^{19} + 4x^{20} + 8x^{21}$$

has $a_0 = 10, b_0 = 2, a_1 = 1, b_1 = 1, a_2 = 5$, and $b_2 = 3$.

Finally, reusing notation from the previous subsection, define d_i to be the size of the polynomial up to (not including) gap i, i.e., $d_i = \sum_{0 \leq j < i}(a_j + b_j)$. And define e_i to be the size *including* gap i, i.e., $e_i = d_i + a_i$.

For the gap at index ℓ, for $1 \leq \ell \leq m$, we store the optimal chunky conversion of $f \bmod x^{d_\ell}$ by a linked list of indices of all gaps in f that should also be gaps between chunks in the optimal chunky representation. We also store, in c_ℓ, the cost in ring operations of multiplying $f \bmod x^{d_\ell}$ (in this optimal representation) by a single chunk of size k, scaled by $1/k$. In particular, since from the discussion above we can conclude that every chunk in this optimal representation has size at most k, and writing s_i for the size of the i'th chunk in the optimal chunky representation of $f \bmod x^{d_\ell}$, c_ℓ is equal to $\sum \delta(s_i)$.

The conversion algorithm works by considering, for each ℓ, the gap of highest degree that should be included in the optimal chunky representation of $f \bmod x^{d_\ell}$. If this gap has index

i, then from above we have $c_\ell = c_i + \delta(d_\ell - e_i)$. This is because the optimal chunky representation will consist exactly of the chunks in $f \bmod x^{d_i}$, followed by the last chunk, which goes from gap i to gap ℓ and has size $d_\ell - e_i$.

Hence to determine the optimal chunky conversion of $f \bmod x_{d_\ell}$ we must find an index $i < \ell$ that minimizes $c_i + \delta(d_\ell - e_i)$. To see how this search is performed efficiently, first observe that, from the discussion above, we need not consider chunks of size greater than k, and therefore at this stage indices i where $d_\ell - e_i > k$ may be excluded. Furthermore, from (5.4), we know that, if $1 \le i < j < \ell$ and $c_i + \delta(d_\ell - e_i) < c_j + \delta(d_\ell - e_j)$, then this same inequality continues to hold as ℓ increases. That is, as soon as an earlier gap results in a smaller cost than a later one, that earlier gap will continue to beat the later one.

Thus we can essentially precompute the values of $\min_{i<\ell}(c_i + \delta(d_\ell - e_i))$ by maintaining a stack of index-index pairs. A pair (i, j) of indices on the top of the stack indicates that $c_i + \delta(d_\ell - e_i)$ is minimal as long as $\ell \le j$. The second pair of indices indicates the minimal value from gap j to the gap of the second index of the second pair, and so forth up to the bottom of the stack and the last gap. Observe that the first index decreases and the second index increases as we move from the top of the stack to the bottom. The initial pair at the bottom of the stack is $(0, m+1)$, indicating that gap 0, corresponding to the power of x that divides f (if any), is always beneficial to take in the chunky representation.

The details of this rather complicated algorithm are given in Algorithm 5.2.

For an informal justification of correctness, consider a single iteration through the main **for** loop. At this point, we have computed all optimal costs $c_1, c_2, \ldots, c_{\ell-1}$, and the lists of gaps to achieve those costs $L_1, L_2, \ldots, L_{\ell-1}$. We also have computed the stack S, indicating which of the gaps up to index $\ell - 2$ is optimal and when.

The **while** loop on Step 5 removes all gaps from the stack which are no longer relevant, either because their cost is now beaten by a previous gap (when $j < \ell$), or because the size of the resulting chunk would be greater than k and therefore unnecessary to consider.

If the condition of Step 7 is true, then there is no index at which gap $(\ell - 1)$ should be used, so we discard it.

Otherwise, the gap at index $\ell - 1$ is good at least some of the time, so we proceed to the task of determining the largest gap index v at which gap $(\ell - 1)$ might still be useful. First, in Steps 14–16, we repeatedly check whether gap $(\ell - 1)$ always beats the gap at the top of the stack S, and if so remove it. After this process, either no gaps remain on the stack, or we have a range $r \le v \le j$ in which binary search can be performed to determine v.

From the definitions, $d_{m+1} = \deg f + 1$, and so the list of gaps L_{m+1} returned on the final step gives the optimal list of gaps to include in $f \bmod x^{\deg f + 1}$, which is of course just f itself.

Theorem 5.3. *Algorithm 5.2 returns the optimal chunky representation for multiplying f by a dense size-k chunk. The running time of the algorithm is linear in the size of the input representation of f.*

Proof. Correctness follows from the discussions above.

Algorithm 5.2: Chunky Conversion Algorithm

```
Input: $k \in \mathbb{N}$, $f \in \mathsf{R}[x]$, and integers $a_i, b_i, d_i$ for $i = 0, 1, 2, \ldots, m$ as above
Output: A list $L$ of the indices of gaps to include in the optimal chunky representation
        of $f$ when multiplying by a single chunk of size $k$
$L_0 \leftarrow$ (empty);   $L_1 \leftarrow 0$
$c_0 \leftarrow 0$;   $c_1 \leftarrow \delta(b_0)$
$S \leftarrow (0, m+1)$
for $\ell = 2, 3, \ldots, m+1$ do
    while top pair $(i, j)$ from $S$ satisfies $j < \ell$ or $d_\ell - e_i > k$ do
        Remove $(i, j)$ from $S$
    if $S$ is not empty and the top pair $(i, j)$ from $S$ satisfies
    $c_i + \delta(d_\ell - e_i) \leq c_{\ell-1} + \delta(d_\ell - e_{\ell-1})$ then
        $L_\ell \leftarrow L_i,\ i$
        $c_\ell \leftarrow c_i + \delta(d_\ell - e_i)$
    else
        $L_\ell \leftarrow L_{\ell-1},\ (\ell - 1)$
        $c_\ell \leftarrow c_{\ell-1} + \delta(d_\ell - e_{\ell-1})$
        $r \leftarrow \ell$
        while top pair $(i, j)$ from $S$ satisfies $c_i + \delta(d_j - e_i) > c_{\ell-1} + \delta(d_j - e_{\ell-1})$ do
            $r \leftarrow j$
            Remove $(i, j)$ from $S$
        if $S$ is empty then $(i, j) \leftarrow (0, m+1)$
        else $(i, j) \leftarrow$ top pair from $S$
        $v \leftarrow$ least index with $r \leq v < j$ s.t. $c_{\ell-1} + \delta(d_v - e_{\ell-1}) > c_i + \delta(d_v - e_i)$
        Push $(\ell - 1, v)$ onto the top of $S$
return $L_{m+1}$
```

For the complexity analysis, first note that the maximal size of S, as well as the number of saved values a_i, b_i, d_i, s_i, L_i, is m, the number of gaps in f. Clearly m is less than the number of nonzero terms in f, so this is bounded above by the sparse or dense representation size. If the lists L_i are implemented as singly-linked lists, sharing nodes, then the total extra storage for the algorithm is $O(m)$.

The total number of iterations of the two **while** loops corresponds to the number of gaps that are removed from the stack S at any step. Since at most one gap is pushed onto S at each step, the total number of removals, and hence the total cost of these **while** loops over all iterations, is $O(m)$.

Now consider the cost of Step 19 at each iteration. If the input is given in the sparse representation, we just perform a binary search on the interval from r to j, for a total cost of $O(m \log m)$ over all iterations. Because m is at most the number of nonzero terms in f, $m \log m$ is bounded above by the sparse representation size, so the theorem is satisfied for sparse input.

When the input is given in the dense representation, a binary search is again used on Step 19, but we start with a one-sided binary search, also called “galloping” search. This search starts from either r or j, depending on which interval endpoint v is closest to. The cost of this search is at a single iteration is $O(\log \min\{v - r, i_2 - v\})$. Notice that the interval (r, j) in the stack is then effectively split at the index v, so intuitively whenever more work is required through one iteration of this step, the size of intervals is reduced, so future iterations should have lower cost.

More precisely, a loose upper bound in the worst case of the total cost over all iterations is $O(\sum_{i=1}^{u} 2^i \cdot (u - i + 1))$, where $u = \lceil \log_2 m \rceil$. This is less than 2^{u+2}, which is $O(m)$, giving linear cost in the size of the dense representation. □

5.3.3 Determining the optimal chunk size

All that remains is to compute the optimal chunk size k that will be used in the conversion algorithm from the previous section. This is accomplished by finding the value of k that minimizes the cost of multiplying two polynomials $f, g \in \mathsf{R}[x]$, under the restriction that every chunk of f and of g has size k.

If f is written in the chunky representation as in (5.3), there are many possible choices for the number of chunks t, depending on how large the chunks are. So define $t(k)$ to be the least number of chunks if each chunk has size at most k, i.e., $\deg f_i < k$ for $1 \le i \le t(k)$. Similarly define $s(k)$ for $g \in \mathsf{R}[x]$ written as in (5.5).

Therefore, from the cost of multiplication in Theorem 5.1, in this part we want to compute the value of k that minimizes

$$t(k) \cdot s(k) \cdot k \cdot \delta(k). \tag{5.6}$$

Say $\deg f = n$. After $O(n)$ preprocessing work (making pointers to the beginning and end of each “gap”), $t(k)$ could be computed using $O(n/k)$ word operations, for any value k. This leads to one possible approach to computing the value of k that minimizes (5.6) above: simply compute (5.6) for each possible $k = 1, 2, \ldots, \max\{\deg f, \deg g\}$. This naïve approach is too costly for our purposes, but underlies the basic idea of our algorithm.

Rather than explicitly computing each $t(k)$ and $s(k)$, we essentially maintain chunky representations of f and g with all chunks having size less than k, starting with $k = 1$. As k increases, we count the number of chunks in each representation, which gives a tight approximation to the actual values of $t(k)$ and $f(k)$, while achieving linear complexity in the size of either the sparse or dense representation.

To facilitate the "update" step, a minimum priority queue Q (whose specific implementation depends on the input polynomial representation) is maintained containing all gaps in the current chunky representations of f and g. For each gap, the key value (on which the priority queue is ordered) is the size of the chunk that would result from merging the two chunks adjacent to the gap into a single chunk.

So for example, if we write f in the chunky representation as

$$f = (4 + 0x + 5x^2) \cdot x^{12} + (7 + 6x + 0x^2 + 0x^3 + 8x^4) \cdot x^{50},$$

then the single gap in f will have key value $3 + 35 + 5 = 43$, More precisely, if f is written as in (5.3), then the i^{th} gap has key value

$$\deg f_{i+1} + e_{i+1} - e_i + 1. \tag{5.7}$$

Each gap in the priority queue also contains pointers to the two (or fewer) neighboring gaps in the current chunky representation. Removing a gap from the queue corresponds to merging the two chunks adjacent to that gap, so we will need to update (by increasing) the key values of any neighboring gaps accordingly.

At each iteration through the main loop in the algorithm, the smallest key value in the priority queue is examined, and k is increased to this value. Then gaps with key value k are repeatedly removed from the queue until no more remain. This means that each remaining gap, if removed, would result in a chunk of size strictly greater than k. Finally, we compute $\delta(k)$ and an approximation of (5.6).

Since the purpose here is only to compute an optimal chunk size k, and not actually to compute chunky representations of f and g, we do not have to maintain chunky representations of the polynomials as the algorithm proceeds, but merely counters for the number of chunks in each one. Algorithm 5.3 gives the details of this computation. Note in particular that Q_f and Q_g initially contain *every* gap — even empty ones — so that $\left|Q_f\right| = t_f - 1$ and $\left|Q_g\right| = t_g - 1$ initially.

All that remains is the specification of the data structures used to implement the priority queues Q_f and Q_g. If the input polynomials are in the sparse representation, we simply use standard binary heaps, which give logarithmic cost for each removal and update. Because the exponents in this case are multi-precision integers, we might imagine encountering chunk sizes that are larger than the largest word-sized integer. But as discussed previously, such a chunk size would be meaningless since a dense polynomial with that size cannot be represented in memory. So our priority queues may discard any gaps whose key value is larger than

Algorithm 5.3: Optimal Chunk Size Computation

```
   Input: $f, g \in \mathsf{R}[x]$
   Output: $k \in \mathbb{N}$ that minimizes $t(k) \cdot s(k) \cdot k \cdot \delta(k)$
1  $Q_f, Q_g \leftarrow$ minimum priority queues initialized with all gaps in $f$ and $g$, respectively
2  $k, k_{\min} \leftarrow 1$;   $c_{\min} \leftarrow t_f t_g$
3  while $Q_f$ and $Q_g$ are not both empty do
4      $k \leftarrow$ smallest key value from $Q_f$ or $Q_g$
5      while $Q_f$ has an element with key value $\leq k$ do
6          Remove a $k$-valued gap from $Q_f$ and update neighbors
7      while $Q_g$ has an element with key value $\leq k$ do
8          Remove a $k$-valued gap from $Q_g$ and update neighbors
9      $c_{\text{current}} \leftarrow (|Q_f| + 1) \cdot (|Q_g| + 1) \cdot k \cdot \delta(k)$
10     if $c_{\text{current}} < c_{\min}$ then
11         $k_{\min} \leftarrow k$;   $c_{\min} \leftarrow c_{\text{current}}$
12 return $k_{\min}$
```

word-sized. This guarantees all keys in the queues are word-size integers, which is necessary for the complexity analysis later.

If the input polynomials are dense, we need a structure which can perform removals and updates in constant time, using $O(\deg f + \deg g)$ time and space. For Q_f, we use an array with length $\deg f$ of (possibly empty) linked lists, where the list at index i in the array contains all elements in the queue with key i. (An array of this length is sufficient because each key value in Q_f is at least 2 and at most $1 + \deg f$.) We use the same data structure for Q_g, and this clearly gives constant time for each remove and update operation.

To find the smallest key value in either queue at each iteration through Step 4, we simply start at the beginning of the array and search forward in each position until a non-empty list is found. Because each queue element update only results in the key values *increasing*, we can start the search at each iteration at the point where the previous search ended. Hence the total cost of Step 4 for all iterations is $O(\deg f + \deg g)$.

The following lemma proves that our approximations of $t(k)$ and $s(k)$ are reasonably tight, and will be crucial in proving the correctness of the algorithm.

Lemma 5.4. *At any iteration through Step 10 in Algorithm 5.3,* $|Q_f| < 2t(k)$ *and* $|Q_g| < 2s(k)$.

Proof. First consider f. There are two chunky representations with each chunk of degree less than k to consider: the optimal having $t(k)$ chunks and the one implicitly computed by Algorithm 5.3 with $|Q_f| + 1$ chunks. Call these $\bar{f}$ and $\hat{f}$, respectively.

We claim that any single chunk of the optimal $\bar{f}$ contains at most three constant terms of chunks in the implicitly-computed $\hat{f}$. If this were not so, then two chunks in $\hat{f}$ could be combined to result in a single chunk with degree less than k. But this is impossible, since all such pairs of chunks would already have been merged after the completion of Step 5.

Therefore every chunk in $\bar{f}$ contains at most two constant terms of distinct chunks in $\hat{f}$. Since each constant term of a chunk is required to be nonzero, the number of chunks in $\hat{f}$ is at most twice the number in $\bar{f}$. Hence $|Q_f|+1 \leq 2t(k)$. An identical argument for g gives the stated result. □

Now we are ready for the main result of this subsection.

Theorem 5.5. *Algorithm 5.3 computes a chunk size k such that $t(k)\cdot s(k)\cdot k\cdot\delta(k)$ is at most 4 times the minimum value. The worst-case cost of the algorithm is linear in the size of the input representations.*

Proof. If k is the value returned from the algorithm and k^* is the value which actually minimizes (5.6), the worst that can happen is that the algorithm computes the actual value of $c_f(k)\,c_g(k)\,k\,\delta(k)$, but overestimates the value of $c_f(k^*)\,c_g(k^*)\,k^*\,\delta(k^*)$. This overestimation can only occur in $c_f(k^*)$ and $c_g(k^*)$, and each of those by only a factor of 2 from Lemma 5.4. So the first statement of the theorem holds.

Write c for the total number of nonzero terms in f and g. The initial sizes of the queues Q_f and Q_g is $O(c)$. Since gaps are only removed from the queues (after they are initialized), the total cost of all queue operations is bounded above by $O(c)$, which in turn is bounded above by the sparse and dense sizes of the input polynomials.

If the input is sparse and we use a binary heap, the cost of each queue operation is $O(\log c)$, for a total cost of $O(c\log c)$, which is a lower bound on the size of the sparse representations. If the input is in the dense representation, then each queue operation has constant cost. Since $c\in O(\deg f+\deg g)$, the total cost linear in the size of the dense representation. □

5.3.4 Chunky Multiplication Overview

Now we are ready to examine the whole process of chunky polynomial conversion and multiplication. First we need the following easy corollary of Theorem 5.3.

Corollary 5.6. *Let $f\in\mathsf{R}[x]$, $k\in\mathbb{N}$, and $\hat{f}$ be any chunky representation of f where all chunks have degree at least k, and $\bar{f}$ be the representation returned by Algorithm 5.2 on input k. The cost of multiplying $\bar{f}$ by a single chunk of size $\ell<k$ is then less than the cost of multiplying $\hat{f}$ by the same chunk.*

Proof. Consider the result of Algorithm 5.2 on input ℓ. We know from Theorem 5.3 that this gives the optimal chunky representation for multiplication of f with a size-ℓ chunk. But the only difference in the algorithm on input ℓ and input k is that more pairs are removed at each iteration on Step 5 on input ℓ.

This means that every gap included in the representation $\bar{f}$ is also included in the optimal representation. We also know that all chunks in $\bar{f}$ have degree less than k, so that $\hat{f}$ must have fewer gaps that are in the optimal representation than $\bar{f}$. It follows that multiplication of a size-ℓ chunk by $\bar{f}$ is more efficient than multiplication by $\hat{f}$. □

To review, the entire process to multiply $f, g \in \mathsf{R}[x]$ using the chunky representation is as follows:

1. Compute k from Algorithm 5.3.
2. Compute chunky representations of f and g using Algorithm 5.2 with input k.
3. Multiply the two chunky representations using Algorithm 5.1.
4. Convert the chunky result back to the original representation.

Because each step is optimal (or within a constant bound of the optimal), we expect this approach to yield the most efficient chunky multiplication of f and g. In any case, we know it will be at least as efficient as the standard sparse or dense algorithm.

Theorem 5.7. *Computing the product of $f, g \in \mathsf{R}[x]$ never uses more ring operations than either the standard sparse or dense polynomial multiplication algorithms.*

Proof. In Algorithm 5.3, the values of $t(k) \cdot s(k) \cdot k \cdot \delta(k)$ for $k = 1$ and $k = \min\{\deg f, \deg g\}$ correspond to the costs of the standard sparse and dense algorithms, respectively. Furthermore, it is easy to see that these values are never overestimated, meaning that the k returned from the algorithm which minimizes this formula gives a cost which is not greater than the cost of either standard algorithm.

Now call $\hat{f}$ and $\hat{g}$ the implicit representations from Algorithm 5.3, and $\bar{f}$ and $\bar{g}$ the representations returned from Algorithm 5.2 on input k. We know that the multiplication of $\hat{f}$ by $\hat{g}$ is more efficient than either standard algorithm from above. Since every chunk in $\hat{g}$ has size k, multiplying $\hat{f}$ by $\bar{g}$ will have an even lower cost, from Theorem 5.3. Finally, since every chunk in $\hat{f}$ has size at most k, Corollary 5.6 tells us that the cost is further reduced by multiplying $\bar{f}$ by $\bar{g}$.

The proof is complete from the fact that conversion back to either original representation takes linear time in the size of the output. □

5.4 Equal-Spaced Polynomials

Next we consider an adaptive representation which is in some sense orthogonal to the chunky representation. This representation will be useful when the coefficients of the polynomial are not grouped together into dense chunks, but rather when they are spaced evenly apart.

Let $f \in \mathsf{R}[x]$ with degree n, and suppose the exponents of f are all divisible by some integer k. Then we can write $f = a_0 + a_1 x^k + a_2 x^{2k} + \cdots$. So by letting $f_D = a_0 + a_1 x + a_2 x^2 + \cdots$, we have $f = f_D \circ x^k$ (where the symbol $\circ$ indicates functional composition).

One motivating example suggested by Michael Monagan is that of homogeneous polynomials. Recall that a multivariate polynomial $h \in \mathsf{R}[x_1, \ldots, x_n]$ is *homogeneous of degree d* if every nonzero term of h has total degree d. It is well-known that the number of variables in a homogeneous polynomial can be effectively reduced by one by writing $y_i = x_i/x_n$ for $1 \le i < n$ and $h = x_n^d \cdot \hat{h}$, for $\hat{h} \in \mathsf{R}[y_1,\ldots,y_{n-1}]$ an $(n-1)$-variate polynomial with max degree d. This leads to efficient schemes for homogeneous polynomial arithmetic.

But this is only possible if (1) the user realizes this structure in their polynomials, and (2) every polynomial used is homogeneous. Otherwise, a more generic approach will be used, such as the Kronecker substitution mentioned in the introduction. Choosing some integer $\ell > d$, we evaluate $h(y, y^{\ell}, y^{\ell^2}, \ldots, y^{\ell^{n-1}})$, and then perform univariate arithmetic over $\mathsf{R}[y]$. But if h is homogeneous, a special structure arises: every exponent of y is of the form $d + i(\ell-1)$ for some integer $i \ge 0$. Therefore we can write $h(y,\ldots,y^{\ell^{n-1}}) = (\bar{h} \circ y^{\ell-1}) \cdot y^d$, for some $\bar{h} \in \mathsf{R}[y]$ with much smaller degree. The algorithms presented in this section will automatically recognize this structure and perform the corresponding optimization to arithmetic.

The key idea is *equal-spaced representation*, which corresponds to writing a polynomial $f \in \mathsf{R}[x]$ as

$$f = (f_D \circ x^k) \cdot x^d + f_S, \tag{5.8}$$

with $k, d \in \mathbb{N}$, $f_D \in \mathsf{R}[x]$ dense with degree less than $n/k - d$, and $f_S \in \mathsf{R}[x]$ sparse with degree less than n. The polynomial f_S is a “noise” polynomial which contains the comparatively few terms in f whose exponents are not of the form $ik + d$ for some $i \ge 0$.

Unfortunately, converting a sparse polynomial to the best equal-spaced representation seems to be difficult. To see why this is the case, consider the much simpler problem of verifying that a sparse polynomial f can be written as $(f_D \circ x^k) \cdot x^d$. For each exponent e_i of a nonzero term in f, this means confirming that $e_i \equiv d \bmod k$. But the cost of computing each $e_i \bmod k$ is roughly $O(\sum (\log e_i)\delta(\log k))$, which is a factor of $\delta(\log k)$ greater than the size of the input. Since k could be as large as the exponents, we see that even verifying a proposed k and d takes too much time for the conversion step. Surely computing such a k and d would be even more costly!

Therefore, for this subsection, we will always assume that the input polynomials are given in the dense representation. In Section 5.5, we will see how by combining with the chunky representation, we effectively handle equal-spaced sparse polynomials without ever having to convert a sparse polynomial directly to the equal-spaced representation.

5.4.1 Multiplication in the equal-spaced representation

Let $g \in \mathsf{R}[x]$ with degree less than m and write $g = (g_D \circ x^{\ell}) \cdot x^e + g_S$ as in (5.8). To compute $f \cdot g$, simply sum up the four pairwise products of terms. All these except for the product $(f_D \circ x^k) \cdot (g_D \circ x^{\ell})$ are performed using standard sparse multiplication methods.

Notice that if $k = \ell$, then $(f_D \circ x^k) \cdot (g_D \circ x^{\ell})$ is simply $(f_D \cdot g_D) \circ x^k$, and hence is efficiently computed using dense multiplication. However, if k and ℓ are relatively prime, then almost any term in the product can be nonzero.

This indicates that the gcd of k and ℓ is very significant. Write r and s for the greatest common divisor and least common multiple of k and ℓ, respectively. To multiply $(f_D \circ x^k)$ by

$(g_D \circ x^\ell)$, we perform a transformation similar to the process of finding common denominators in the addition of fractions. First split $f_D \circ x^k$ into s/k (or ℓ/r) polynomials, each with degree less than n/s and right composition factor x^s, as follows:

$$f_D \circ x^k = (f_0 \circ x^s) + (f_1 \circ x^s) \cdot x^k + (f_2 \circ x^s) \cdot x^{2k} \cdots + (f_{s/k-1} \circ x^s) \cdot x^{s-k}$$

Similarly split $g_D \circ x^\ell$ into s/ℓ polynomials $g_0, g_1, \ldots, g_{s/\ell-1}$ with degrees less than m/s and right composition factor x^s. Then compute all pairwise products $f_i \cdot g_j$, and combine them appropriately to compute the total sum (which will be equal-spaced with right composition factor x^r).

Algorithm 5.4 gives the details of this method.

Algorithm 5.4: Equal Spaced Multiplication

```
   Input: $f = (f_D \circ x^k) \cdot x^d + f_S$,   $g = (g_D \circ x^\ell) \cdot x^e + g_S$,
   with $f_D = a_0 + a_1 x + a_2 x^2 + \cdots$,   $g_D = b_0 + b_1 x + b_2 x^2 + \cdots$
   Output: The product $f \cdot g$
 1 $r \leftarrow \gcd(k, \ell)$,   $s \leftarrow \operatorname{lcm}(k, \ell)$
 2 for $i = 0, 1, \ldots, s/k - 1$ do
 3     $f_i \leftarrow a_i + a_{s+i} x + a_{2s+i} x^2 + \cdots$
 4 for $i = 0, 1, \ldots, s/\ell - 1$ do
 5     $g_i \leftarrow b_i + b_{s+i} x + b_{2s+i} x^2 + \cdots$
 6 $h_D \leftarrow 0$
 7 for $i = 0, 1, \ldots, s/k - 1$ do
 8     for $j = 0, 1, \ldots, s/\ell - 1$ do
 9         Compute $f_i \cdot g_j$ by dense multiplication
10         $h_D \leftarrow h_D + ((f_i \cdot g_j) \circ x^s) \cdot x^{ik+j\ell}$
11 Compute $(f_D \circ x^k) \cdot g_S$, $(g_D \circ x^\ell) \cdot f_S$, and $f_S \cdot g_S$ by sparse multiplication
12 return $h_D \cdot x^{e+d} + (f_D \circ x^k) \cdot g_S \cdot x^d + (g_D \circ x^\ell) \cdot f_S \cdot x^e + f_S \cdot g_S$
```

As with chunky multiplication, this final product is easily converted to the standard dense representation in linear time. The following theorem gives the complexity analysis for equal-spaced multiplication.

Theorem 5.8. *Let f, g be as above such that $n > m$, and write t_f, t_g for the number of nonzero terms in f_S and g_S, respectively. Then Algorithm 5.4 correctly computes the product $f \cdot g$ using*

$$O\Big((n/r) \cdot \delta(m/s) + n t_g/k + m t_f/\ell + t_f t_g\Big)$$

ring operations.

Proof. Correctness follows from the preceding discussion.

The polynomials f_D and g_D have at most n/k and m/ℓ nonzero terms, respectively. So the cost of computing the three products in Step 11 by using standard sparse multiplication is $O(n t_g/k + m t_f/\ell + t_f t_g)$ ring operations, giving the last three terms in the complexity measure.

The initialization in Steps 2–5 and the additions in Steps 10 and 12 all have cost bounded by $O(n/r)$, and hence do not dominate the complexity.

All that remains is the cost of computing each product $f_i \cdot g_j$ by dense multiplication on Step 9. From the discussion above, $\deg f_i < n/s$ and $\deg g_j < m/s$, for each i and j. Since $n > m$, $(n/s) > (m/s)$, and therefore this product can be computed using $O((n/s)\delta(m/s))$ ring operations. The number of iterations through Step 9 is exactly $(s/k)(s/\ell)$. But $s/\ell = k/r$, so the number of iterations is just s/r. Hence the total cost for this step is $O((n/r)\delta(m/s))$, which gives the first term in the complexity measure. □

It is worth noting that no additions of ring elements are actually performed through each iteration of Step 10. The proof is as follows. If any additions were performed, we would have

$$i_1 k + j_1 \ell \equiv i_2 k + j_2 \ell \pmod{s}$$

for distinct pairs (i_1, j_1) and (i_2, j_2). Without loss of generality, assume $i_1 \neq i_2$, and write

$$(i_1 k + j_1 \ell) - (i_2 k + j_2 \ell) = qs$$

for some $q \in \mathbb{Z}$. Rearranging gives

$$(i_1 - i_2)k = (j_2 - j_1)\ell + qs.$$

Because $\ell | s$, the left hand side is a multiple of both k and ℓ, and therefore by definition must be a multiple of s, their lcm. Since $0 \leq i_1, i_2 < s/k$, $|i_1 - i_2| < s/k$, and therefore $|(i_1 - i_2)k| < s$. The only multiple of s with this property is of course 0, and since $k \neq 0$ this means that $i_1 = i_2$, a contradiction.

The following theorem compares the cost of the equal-spaced multiplication algorithm to standard dense multiplication, and will be used to guide the approach to conversion below.

Theorem 5.9. *Let f, g, m, n, t_f, t_g be as before. Algorithm 5.4 does not use asymptotically more ring operations than standard dense multiplication to compute the product of f and g as long as $t_f \in O(\delta(n))$ and $t_g \in O(\delta(m))$.*

Proof. Assuming again that $n > m$, the cost of standard dense multiplication is $O(n\delta(m))$ ring operations, which is the same as $O(n\delta(m) + m\delta(n))$.

Using the previous theorem, the number of ring operations used by Algorithm 5.4 is

$$O((n/r)\delta(m/s) + n\delta(m)/k + m\delta(n)/\ell + \delta(n)\delta(m)).$$

Because all of k, ℓ, r, s are at least 1, and since $\delta(n) < n$, every term in this complexity measure is bounded by $n\delta(m) + m\delta(n)$. The stated result follows. □

5.4.2 Converting to equal-spaced

The only question when converting a polynomial f to the equal-spaced representation is how large we should allow t_S (the number of nonzero terms in of f_S) to be. From Theorem 5.9 above, clearly we need $t_S \in \delta(\deg f)$, but we can see from the proof of the theorem that having this bound be tight will often give performance that is equal to the standard dense method (not worse, but not better either).

Let t be the number of nonzero terms in f. Since the goal of any adaptive method is to in fact be faster than the standard algorithms, we use the lower bound of $\delta(n) \in \Omega(\log n)$ and $t \leq \deg f + 1$ and require that $t_S < \log_2 t$.

As usual, let $f \in \mathsf{R}[x]$ with degree less than n and write

$$f = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_t x^{e_t},$$

with each $a_i \in \mathsf{R} \setminus \{0\}$. The reader will recall that this corresponds to the sparse representation of f, but keep in mind that we are assuming f is given in the dense representation; f is written this way only for notational convenience.

The conversion problem is then to find the largest possible value of k such that all but at most $\log_2 t$ of the exponents e_j can be written as $ki + d$, for any nonnegative integer i and a fixed integer d. Our approach to computing k and d will be simply to check each possible value of k, in decreasing order. To make this efficient, we need a bound on the size of k.

Lemma 5.10. *Let $n \in \mathbb{N}$ and $e_1, \ldots, e_t$ be distinct integers in the range $[0, n]$. If at least $t - \log_2 t$ of the integers e_i are congruent to the same value modulo k, for some $k \in \mathbb{N}$, then*

$$k \leq \frac{n}{t - 2\log_2 t - 1}.$$

Proof. Without loss of generality, order the e_i's so that $0 \leq e_1 < e_2 < \cdots < e_t \leq n$. Now consider the telescoping sum $(e_2 - e_1) + (e_3 - e_2) + \cdots + (e_t - e_{t-1})$. Every term in the sum is at least 1, and the total is $e_t - e_1$, which is at most n.

Let $S \subseteq \{e_1, \ldots, e_t\}$ be the set of at most $\log_2 t$ integers not congruent to the others modulo k. Then for any $e_i, e_j \notin S$, $e_i \equiv e_j \bmod k$. Therefore $k | (e_j - e_i)$. If $j > i$, this means that $e_j - e_i \geq k$.

Returning to the telescoping sum above, each $e_j \in S$ is in at most two of the sum terms $e_i - e_{i-1}$. So all but at most $2\log_2 t$ of the terms are at least k. Since there are exactly $t - 1$ terms, and the total sum is at most n, we conclude that $(t - 2\log_2 t - 1) \cdot k \leq n$. The stated result follows. □

We now employ this lemma to develop an algorithm to determine the best values of k and d, given a dense polynomial f. Starting from the largest possible value from the bound, for each candidate value k, we compute each $e_i \bmod k$, and find the majority element — that is, a common modular image of more than half of the exponents.

To compute the majority element, we use a now well-known approach first credited to Boyer and Moore (1991) and Fischer and Salzberg (1982). Intuitively, pairs of different elements are repeatedly removed until only one element remains. If there is a majority element, this remaining element is it; only one extra pass through the elements is required to check whether this is the case. In practice, this is accomplished without actually modifying the list.

Algorithm 5.5: Equal Spaced Conversion

```
   Input: Exponents $e_1, e_2, \dots, e_t \in \mathbb{N}$ and $n \in \mathbb{N}$ such that $0 \le e_1 < e_2 < \cdots < e_t = n$
   Output: $k, d \in \mathbb{N}$ and $S \subseteq \{e_1, \dots, e_t\}$ such that $e_i \equiv d \bmod k$ for all exponents $e_i$ not in $S$,
           and $|S| \le \log_2 t$.
1  if $t < 32$ then $k \leftarrow n$
2  else $k \leftarrow \lfloor n/(t - 1 - 2\log_2 t) \rfloor$
3  while $k \ge 2$ do
4      $d \leftarrow e_1 \bmod k$;   $j \leftarrow 1$
5      for $i = 2, 3, \dots, t$ do
6          if $e_i \equiv d \bmod k$ then $j \leftarrow j + 1$
7          else if $j > 0$ then $j \leftarrow j - 1$
8          else $d \leftarrow e_i \bmod k$;   $j \leftarrow 1$
9      $S \leftarrow \{e_i : e_i \not\equiv d \bmod k\}$
10     if $|S| \le \log_2 t$ then return $k$, $d$, $S$
11     $k \leftarrow k - 1$
12 return 1, 0, $\emptyset$
```

Given k, d, S from the algorithm, in one more pass through the input polynomial, f_D and f_S are constructed such that $f = (f_D \circ x^k) \cdot x^d + f_S$. After performing separate conversions for two polynomials $f, g \in \mathsf{R}[x]$, they are multiplied using Algorithm 5.4.

The following theorem proves correctness when $t > 4$. If $t \le 4$, we can always trivially set $k = e_t - e_1$ and $d = e_1 \bmod k$ to satisfy the stated conditions.

Theorem 5.11. *Given integers $e_1, \dots, e_t$ and n, with $t > 4$, Algorithm 5.5 computes the largest integer k such that at least $t - \log_2 t$ of the integers e_i are congruent modulo k, and uses $O(n)$ word operations.*

Proof. In a single iteration through the **while** loop, we compute the majority element of the set $\{e_i \bmod k : i = 1, 2, \dots, t\}$, if there is one. Because $t > 4$, $\log_2 t < t/2$. Therefore any element which occurs at least $t - \log_2 t$ times in a t-element set is a majority element, which proves that any k returned by the algorithm is such that at least $t - \log_2 t$ of the integers e_i are congruent modulo k.

From Lemma 5.10, we know that the initial value of k on Step 1 or 2 is greater than the optimal k value. Since we start at this value and decrement to 1, the largest k satisfying the stated conditions is returned.

For the complexity analysis, first consider the cost of a single iteration through the main **while** loop. Since each integer e_i is word-sized, computing each $e_i \bmod k$ has constant cost, and this happens $O(t)$ times in each iteration.

If $t < 32$, each of the $O(n)$ iterations has constant cost, for total cost $O(n)$.

Otherwise, we start with $k = \lfloor n/(t-1-2\log_2 t)\rfloor$ and decrement. Because $t \geq 32$, $t/2 > 1+2\log_2 t$. Therefore $(t-1-2\log_2 t) > t/2$, so the initial value of k is less than $2n/t$. This gives an upper bound on the number of iterations through the **while** loop, and so the total cost is $O(n)$ word operations, as required. □

Algorithm 5.5 can be implemented using only $O(t)$ space for the storage of the exponents $e_1, \ldots, e_t$, which is linear in the size of the output, plus the space required for the returned set S.

5.5 Chunks with Equal Spacing

The next question is whether the ideas of chunky and equal-spaced polynomial multiplication can be effectively combined into a single algorithm. As before, we seek an *adaptive* combination of previous algorithms, so that the combination is never asymptotically worse than either original idea.

An obvious approach would be to first perform chunky polynomial conversion, and then equal-spaced conversion on each of the dense chunks. Unfortunately, this would be asymptotically less efficient than equal-spaced multiplication alone in a family of instances, and therefore is not acceptable as a proper adaptive algorithm.

The algorithm presented here does in fact perform chunky conversion first, but instead of performing equal-spaced conversion on each dense chunk independently, Algorithm 5.5 is run simultaneously on all chunks in order to determine, for each polynomial, a single spacing parameter k that will be used for every chunk.

Let $f = f_1x^{e_1}+f_2x^{e_2}+\cdots+f_tx^{e_t}$ in the optimal chunky representation for multiplication by another polynomial g. We first compute the smallest bound on the spacing parameter k for any of the chunks f_i, using Lemma 5.10. Starting with this value, we execute the **while** loop of Algorithm 5.5 for each polynomial f_i, stopping at the largest value of k such that the total size of all sets S on Step 9 for all chunks f_i is at most $\log_2 t_f$, where t_f is the total number of nonzero terms in f.

The polynomial f can then be rewritten (recycling the variables f_i and e_i) as

$$f = (f_1 \circ x^k)\cdot x^{e_1} + (f_2 \circ x^k)\cdot x^{e_2} + \cdots + (f_t \circ x^k)\cdot x^{e_t} + f_S,$$

where f_S is in the sparse representation and has $O(\log t_f)$ nonzero terms.

Let k^* be the value returned from Algorithm 5.5 on input of the entire polynomial f. Using k^* instead of k, f could still be written as above with f_S having at most $\log_2 t_f$ terms. Therefore the value of k computed in this way is always greater than or equal to k^* if the initial bounds are correct. This will be the case except when every chunk f_i has few nonzero terms (and therefore t is close to t_f). However, this reduces to the problem of converting a sparse polynomial to the equal-spaced representation, which seems to be intractable, as discussed above. So our cost analysis will be predicated on the assumption that the computed value of k is never smaller than k^*.

We perform the same equal-spaced conversion for g, and then use Algorithm 5.1 to compute the product $f \cdot g$, with the difference that each product $f_i \cdot g_j$ is computed by Algorithm 5.4 rather than standard dense multiplication. As with equal-spaced multiplication, the products involving f_S or g_S are performed using standard sparse multiplication.

Theorem 5.12. *The algorithm described above to multiply polynomials with equal-spaced chunks never uses more ring operations than either chunky or equal-spaced multiplication, provided that the computed "spacing parameters" k and ℓ are not smaller than the values returned from Algorithm 5.5.*

Proof. Let n, m be the degrees of f, g respectively and write t_f, t_g for the number of nonzero terms in f, g respectively. The sparse multiplications involving f_S and g_S use a total of

$$t_g \log t_f + t_f \log t_g + (\log t_f)(\log t_g)$$

ring operations. Both the chunky or equal-spaced multiplication algorithms always require $O(t_g \delta(t_f) + t_f \delta(t_g))$ ring operations in the best case, and since $\delta(n) \in \Omega(\log n)$, the cost of these sparse multiplications is never more than the cost of the standard chunky or equal-spaced method.

The remaining computation is that to compute each product $f_i \cdot g_j$ using equal-spaced multiplication. Write k and ℓ for the powers of x in the right composition factors of f and g respectively. Theorem 5.8 tells us that the cost of computing each of these products by equal-spaced multiplication is never more than computing them by standard dense multiplication, since k and ℓ are both at least 1. Therefore the combined approach is never more costly than just performing chunky multiplication.

To compare with the cost of equal-spaced multiplication, assume that k and ℓ are the actual values returned by Algorithm 5.5 on input f and g. This is the worst case, since we have assumed that k and ℓ are never smaller than the values from Algorithm 5.5.

Now consider the cost of multiplication by a single equal-spaced chunk of g. This is the same as assuming g consists of only one equal-spaced chunk. Write $d_i = \deg f_i$ for each equal-spaced chunk of f, and r, s for the gcd and lcm of k and ℓ, respectively. If $m > n$, then of course m is larger than each d_i, so multiplication using the combined method will use $O((m/r)\sum \delta(d_i/s))$ ring operations, compared to $O((m/r)\delta(n/s))$ for the standard equal-spaced algorithm, by Theorem 5.8.

Now recall the cost equation (5.6) used for Algorithm 5.3:

$$c_f(b) \cdot c_g(b) \cdot b \cdot \delta(b),$$

where b is the size of all dense chunks in f and g. By definition, $c_f(n) = 1$, and $c_g(n) \le m/n$, so we know that $c_f(n)\, c_g(n)\, n\, \delta(n) \le m\, \delta(n)$. Because the chunk sizes d_i were originally chosen by Algorithm 5.3, we must therefore have $m \sum_{i=1}^{t} \delta(d_i) \le m\delta(n)$. The restriction that the δ function grows more slowly than linear then implies that $(m/r)\sum \delta(d_i/s) \in O((m/r)\delta(n/s))$, and so the standard equal-spaced algorithm is never more efficient in this case.

When $m \le n$, the number of ring operations to compute the product using the combined method, again by Theorem 5.8, is

$$O\left(\delta(m/s)\sum_{d_i \ge m}(d_i/r)+(m/r)\sum_{d_i<m}\delta(d_i/s)\right), \tag{5.9}$$

compared with $O((n/r)\delta(m/s))$ for the standard equal-spaced algorithm. Because we always have $\sum_{i=1}^{t} d_i \le n$, the first term of (5.9) is $O((n/r)\delta(m/s))$. Using again the inequality $m\sum_{i=1}^{t}\delta(d_i) \le m\delta(n)$, along with the fact that $m\delta(n) \in O(n\delta(m))$ because $m \le n$, we see that the second term of (5.9) is also $O((n/r)\delta(m/s))$. Therefore the cost of the combined method is never more than the cost of equal-spaced multiplication alone. □

5.6 Implementation and benchmarking

We implemented the full chunky conversion and multiplication algorithm for dense polynomials in MVP. We compare the results against the standard dense and sparse multiplication algorithms in MVP. Previously, in sections 2.5 and 4.4, we saw that the multiplication routines in MVP are competitive with existing software. Therefore our comparison against these routines is fair and will provide useful information on the utility of the new algorithms.

Recall that the chunky conversion algorithm requires a way to approximate $\delta(n)$ for any integer n. For this, we actually used the timing results reported in Section 4.4 for dense polynomial multiplication in MVP on our testing machine. Using this data and the known crossover points, we did a least-squares fit to the three curves

$$\begin{aligned} n\delta_1(n) &= a_1 n^2 + b_1 n + c_1 \\ n\delta_2(n) &= a_2 n^{\log_2 3} + b_2 n + c_2 \\ n\delta_3(n) &= a_3 2^{\lceil \log_2 n \rceil} \lceil \log_2 n \rceil + b_3 n + c_3, \end{aligned}$$

for the ranges of classical, Karatsuba, and FFT-based multiplication, respectively. Clearly this could be automated as part of a tuning phase, but this has not yet been implemented. However, observe that as only the *relative* times are important, it is possible that some default timing data could still be useful on different machines.

Choosing benchmarks for assessing the performance of these algorithms is a challenging task. Our algorithms adapts to chunkiness in the input, but this measure is difficult to quantify. Furthermore, while we can (and do) generate examples that are more or less chunky, it is difficult to justify whether such polynomials actually appear in practice.

With these reservations in mind, we proceed to describe the benchmarking trials we performed to compare chunky multiplication against the standard dense and sparse algorithms. First, to generate random polynomials with some varying "chunkiness" properties, we used the following approach. The parameters to the random polynomial generation are the usual degree n and desired sparsity t, as well as a third parameter c, which should be a small positive integer. The polynomials were then generated by randomly choosing t exponents, successively, and for each exponent choosing a random nonzero coefficient. To avoid trivially easy cases, we always set the constant coefficient and the coefficient of x^d to be nonzero.

Chunkiness c	Standard sparse	Chunky multiplication
0	.981	1.093
10	.969	1.059
20	.984	1.075
25	.899	.971
30	.848	1.262
32	.778	.771
34	.749	.313
36	.729	.138
38	.646	.084
40	.688	.055

Table 5.1: Benchmarks for adaptive multiplication with varying chunkiness

However, to introduce “chunkiness” as c grows larger, the exponents of successive terms were not always chosen independently. With probability proportional to $1/c$, the next exponent was chosen independently and randomly from $\{1, 2, \ldots, d-1\}$. However, with high probability proportional to $(c-1)/c$, the next successive exponent was chosen to be “close” to the previous one. Specifically, in this case the next exponent was chosen randomly from a set of size proportional to $n/2^c$ surrounding the previously-chosen exponent. This deviation from independence has the effect of creating clusters of nonzero coefficients, when the parameter c is sufficiently large.

We make no claims that this randomization has any basis in a particular application, but point out that it does provide more of a challenge to our algorithms than the simplistic approach of clustering nonzero terms into completely dense chunks with no intervening gaps. Also observe that when $c = 0$, we simply have a sparse polynomial with randomly-chosen support and no particular chunky structure. Furthermore, when c is on the order of $\log_2 n$, the generated polynomial is likely to consist of only a few very dense chunks. So by varying c in this range, we see the performance of the chunky multiplication algorithm on a range of cases.

Specifically, for our benchmarks we fixed the degree d at one million and the sparsity t at 3000. With these parameters, the usual dense and sparse algorithms in MVP have roughly the same cost. We then ran a number of tests varying the chunkiness parameter c between 0 and 40. In every case, the dense algorithm used close to 1.91 CPU seconds for a single operation. The timings reported in Table 5.1 are relative to this time, the cost of the dense multiplication algorithm. As usual, a number less than one means that algorithm was faster than the dense algorithm for the specified choice of c. Observe that although the sparsity t was invariant for all test cases, the normal sparse algorithm still derives some benefit from polynomials with high chunkiness, simply because of the reduced number of terms in the output.

With the slight anomaly at $c = 30$ (probably due to some algorithmic “confusion” and inaccuracy in $\delta(n)$), we see that the algorithm performs quite well, and as expected according to our theoretical results. In particular, the chunky multiplication algorithm is always either very nearly the same as existing methods, or much better than them. We also report, perhaps surprisingly, that the overhead associated with the chunky conversion is negligible in nearly every case. In fact, we can see this in the first few benchmarks with small c, where the chunky representation produced is simply the dense representation; the performance hit for the unnecessary conversion, at least in these cases, is less than 10 percent.

5.7 Conclusions and Future Work

Two methods for adaptive polynomial multiplication have been given where we can compute optimal representations (under some set of restrictions) in linear time in the size of the input. Combining these two ideas into one algorithm inherently captures both measures of difficulty, and will in fact have significantly better performance than either the chunky or equal-spaced algorithm in many cases.

However, converting a sparse polynomial to the equal-spaced representation in linear time is still out of reach, and this problem is the source of the restriction of Theorem 5.12. Some justification for the impossibility of such a conversion algorithm was given, due to the fact that the exponents could be long integers. However, we still do not have an algorithm for sparse polynomial to equal-spaced conversion under the (probably reasonable) restriction that all exponents be word-sized integers. A linear-time algorithm for this problem would be useful and would make our adaptive approach more complete, though slightly more restricted in scope.

On the subject of word size and exponents, we assumed at the beginning that $O(t \log n)$ words are required to store the sparse representation of a polynomial. This fact is used to justify that some of the conversion algorithms with $O(t \log t)$ complexity were in fact linear-time in the size of the input. However, the original assumption is not very good for many practical cases, e.g., where all exponents fit into single machine words. In such cases, the input size is $O(t)$, but the conversion algorithms still use $O(t \log t)$ operations, and hence are not linear-time. We would argue that they are still useful, especially since the sparse multiplication algorithms are quadratic, but there is certainly room for improvement in the sparse polynomial conversion algorithms.

Yet another area for further development is multivariate polynomials. We have mentioned the usefulness of Kronecker substitution, but developing an adaptive algorithm to choose the optimal variable ordering would give significant improvements. An interesting observation is that using the chunky conversion under a dense Kronecker substitution seems similar in many ways to the recursive dense representation of a multivariate polynomial. Examining the connections here more carefully would be interesting, and could give more justification to the practical utility of the algorithms we have presented.

Finally, even though we have proven that our algorithms produce optimal adaptive representations, it is always under some restriction of the way that choice is made (for example, requiring to choose an "optimal chunk size" k first, and then compute optimal conversions given k). These results would be significantly strengthened by proving lower bounds over all available adaptive representations of a certain type, but such results have thus far been elusive.

The essence of being human is that one does not seek perfection,. . . that one is prepared in the end to be defeated and broken up by life, which is the inevitable price of fastening one's love upon other human individuals. No doubt alcohol, tobacco, and so forth, are things that a saint must avoid, but sainthood is also a thing that human beings must avoid.

—George Orwell, "Reflections on Gandhi" (1949)

Chapter 6

Sparse perfect powers

We examine the problem of detecting when a given sparse polynomial f is equal to h^r for some other polynomial h and integer $r \geq 2$. In this case we say f is a *perfect power*, and h is its rth root. We give randomized algorithms to detect when f is a perfect power, by repeatedly evaluating the polynomial in specially-chosen finite fields and checking whether the evaluations are themselves perfect powers. These detection algorithms are randomized of the Monte Carlo type, meaning that they are always fast but may return an incorrect answer (with small probability). In fact, the only kind of wrong answer ever returned will be a false positive, so we have shown that the perfect power detection problem is in the complexity class **coRP**. As a by-product, these decision problems also compute an integer $r > 2$ such that f is an rth perfect power, if such an r exists.

Once the power r is known, we turn to the problem of actually computing the perfect rth root of f, and give two different algorithms. In the case of rational number coefficients, we exhibit a deterministic algorithm that is *output-sensitive*; i.e., the cost of the algorithm depends on the size of the perfect root h in the sparse representation. The algorithm relies on polynomial factorization over algebraic extension fields to achieve output-sensitive polynomial time.

The second algorithm we consider for computing the perfect root is inspired by the Newton iteration algorithm that is most efficient for dense polynomials. Our modified Newton iteration is more sensitive to the sparsity of the problem, and carefully avoids blow-up of intermediate results, but only if a certain conjecture regarding the sparsity of intermediate powers is true. We have evidence to believe this conjecture, and under that assumption our algorithm is very efficient and practical.

A preliminary version of this work appeared at ISSAC 2008 (Giesbrecht and Roche, 2008), and further progress has been presented in an article accepted to appear in the Journal of Symbolic Computation (Giesbrecht and Roche, 2011). We are very grateful to the helpful discussions of these topics with Erich Kaltofen and Igor Shparlinski, and particularly to Pascal Koiran for pointing out the perfect power certification idea using logarithmic derivatives that we will discuss in Section 6.3.1.

6.1 Background

This chapter and its sequel focus on computational problems where the input polynomial is given in the sparse representation. Recall that this means we write

$$f = \sum_{1 \le i \le t} c_i \overline{x}^{\overline{e}_i} \in \mathsf{R}[x_1, \ldots, x_\ell], \tag{6.1}$$

where $c_1, \ldots, c_t \in \mathsf{R}^*$, $\overline{e}_1, \ldots, \overline{e}_t \in \mathbb{N}^\ell$ are distinct exponent tuples with $0 \le \|\overline{e}_1\|_1 \le \cdots \le \|\overline{e}_t\|_1 = \deg f$, and $\overline{x}^{\overline{e}_i}$ is the monomial $x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_\ell^{e_{i\ell}}$ of total degree $\|\overline{e}_i\|_1 = \sum_{1 \le j \le \ell} e_{ij}$. We say f is t*-sparse* and write $\tau(f) = t$. We present algorithms which require time polynomial in $\tau(f)$ and $\log \deg f$.

6.1.1 Related work

Computational work on sparse polynomials has proceeded steadily for the past three decades. We briefly pause to examine some of the most significant results in this area.

David Plaisted (1977; 1984) initiated the study of computational complexity for sparse polynomial computations. His early work showed, surprisingly, that some basic problems that admit fast algorithms for densely-represented polynomials are **NP**-hard when the input is sparse. In particular, Plaisted gives a reduction from 3-SAT to *relative primality testing*, the problem of determining whether a pair of sparse polynomials $f, g \in \mathbb{Z}[x]$ has any non-trivial common factor. This implies for instance that a polynomial-time greatest common divisor algorithm for sparse polynomials is unlikely to exist.

A number of other researchers have investigated complexity-theoretic problems relating to sparse polynomials as well. Recall that #**P** is the class of problems that counts accepting inputs for problems in **NP**, and important examples of a #**P**-complete problems are counting the number of satisfying assignments for a boolean formula or computing the permanent of a matrix. Quick (1986) and von zur Gathen, Karpinski, and Shparlinski (1996) followed Plaisted by proving (among other results) that the sparse gcd problem is actually #**P**-complete, when the problem is defined to determine the degree of the greatest common divisor of two given sparse polynomials.

Observe that relative primality of two polynomials corresponds to their gcd having minimal degree, while *divisibility* of sparse polynomials corresponds to the degree of their gcd being maximal. It is not yet known whether sparse divisibility testing can be performed in polynomial time, but Grigoriev, Karpinski, and Odlyzko (1996) showed the existence of short proofs of non-divisibility, meaning the problem is in **coNP**. Later, Karpinski and Shparlinski (1999) showed that testing *square-freeness* of sparse polynomials is also **NP**-hard, via a reduction from relative primality testing.

But not every computational problem with sparse polynomials is hard. In fact, there has been significant progress in developing polynomial-time algorithms for sparse polynomial computation over the years. While some operations, such as computing derivatives, are trivially polynomial-time in the sparse representation, many problems require non-trivial algorithms. One major area of research in this direction regards sparse polynomial interpolation, the topic of the next two chapters.

Computing low-degree factors and in particular roots of sparse polynomials is another area of rapid progress in algorithmic development (Cucker, Koiran, and Smale, 1999; Lenstra, 1999; Kaltofen and Koiran, 2005, 2006). These algorithms generally take a degree bound d and a sparse polynomial f and compute all factors of f with degree at most d, in polynomial-time in d and the sparse representation size of f. Since the number of terms in high-degree factors may be exponentially larger than the size of the input, this is in some sense the best that can be hoped for without considering output-sensitive algorithms.

We now turn to the topic of this chapter, namely detecting and computing perfect power factorizations of sparse polynomials. The algorithm we present for perfect powers is interesting in that, unlike the factorization methods mentioned above, the sparse factors it produces may have exponentially high degree in the size of the input.

Two well-known techniques can be applied to the problem of testing for perfect powers, and both are very efficient when $f = h^r$ is given in the dense representation. We can compute the squarefree decomposition of f as in (Yun, 1976), and determine whether f is a perfect power by checking whether the greatest (integer) common divisor of the exponents of all nontrivial factors in the squarefree decomposition is at least 2. An even faster method (in theory and practice) to find h given $f = h^r$ is by a Newton iteration. This technique has also proven to be efficient in computing perfect roots of (dense) multi-precision integers (Bach and Sorenson, 1993; Bernstein, 1998). In summary however, we note that both these methods require at least linear time in the *degree* of f, which may be exponential in the sparse representation size.

Newton iteration has also been applied to finding perfect polynomial roots of lacunary (or other) polynomials given by straight-line programs. Kaltofen (1987) shows how to compute a straight-line program for h, given a straight-line program for $f = h^r$ and the value of r. This method has complexity polynomial in the size of the straight-line program for f, and in the degree of h, and in particular is effective for large r. Our algorithms, which require input in the sparse representation, are not as general, but they do avoid the dependence on the degree of h. Interestingly, we will show that the case of large r, which is important for straight-line programs, cannot happen for sparse polynomials, at least over the rings that we will consider.

Closest to this current work, (Shparlinski, 2000) shows how to recognize whether $f = h^2$ for a lacunary polynomial $f \in \mathbb{F}_q[x]$. Shparlinski uses random evaluations and tests for quadratic residues. How to determine whether a lacunary polynomial is *any* perfect power is posed as an open question; our Algorithm 6.4 demonstrates that this problem can be solved in randomized polynomial-time.

6.1.2 Overview of results

We will always assume that the sparsity $\tau(f) \geq 2$. Otherwise $f = x^n$, and determining whether f is a perfect power is equivalent to determining whether $n \in \mathbb{N}$ is composite. This is of course tractable, but producing an $r \geq 2$ such that f is a perfect rth power is then equivalent to long integer factorization, a notoriously difficult problem in number theory that we do not solve here. Surprisingly, the intractability of computing such an r is avoided when $\tau(f) \geq 2$.

We first consider the problem of detecting perfect powers and computing the power r for the univariate case, where we write

$$f = \sum_{1 \leq i \leq t} c_i x^{e_i} \in \mathsf{R}[x], \tag{6.2}$$

with integer exponents $0 \leq e_1 < e_2 < \cdots < e_t = \deg f$.

Two cases for the ring R are handled: the integers and finite fields of characteristic p greater than the degree of f. When f has integer coefficients, our algorithms also require time polynomial in the size of those integers in the IMM model.

To be precise about this definition of size, first recall from Chapter 1 that size(a) for $a \in \mathbb{Z}$ is the number of machine words needed to represent a in an IMM. So, if w is the size of machine words, size$(a) = \lceil \log_2(|a|+1)/w \rceil + 1$. For $f \in \mathbb{Z}[x]$ written as in (6.2), define:

$$\text{size}(f) = \sum_{i=1}^{t} \text{size}(c_i) + \sum_{i=1}^{t} \text{size}(e_i). \tag{6.3}$$

We will often employ the following upper bound for simplicity:

$$\text{size}(f) \leq t\left(H(f) + \text{size}(\deg f)\right), \tag{6.4}$$

where $H(f)$ is defined as $\max_{1 \leq i \leq t} \text{size}(c_i)$.

For the analysis, as in the previous chapter, we will write $\mathsf{M}(r)$ for the number of ring operations required for degree-r dense polynomial multiplication. For $a \in \mathbb{N}$, We also define $\mathsf{N}(a)$ to be the number of IMM operations required to multiply two integers at most a in absolute value. From Section 1.4, we know that $\mathsf{N}(a) \in O(\text{size}(a) \log \text{size}(a))$. Our motivation for the $\mathsf{N}(a)$ notation is purely for clarity and brevity, not to reflect the possibility of improved algorithms for this problem in the future. One consequence is that, if the finite field elements are represented in machine words on a normal IMM, multiplication of degree-r dense polynomials in $\mathbb{F}_p[x]$ is performed with

$$O(N(p^r)) \in O(r\,\text{size}(p) \cdot \log(r + \text{size}(p))) \in O(r\mathsf{N}(p)\log r)$$

IMM operations, by encoding the whole polynomial into an integer, as discussed in Chapter 1.

Notice that the algorithm for integer polynomials will also cover those with rational coefficients, since if $f \in \mathbb{Q}[x]$, we can simply work with $\overline{f} = cf \in \mathbb{Z}[x]$ for the smallest possible $c \in \mathbb{Z}^*$.

We present polynomial-time algorithms for polynomials over finite fields and over the integers. Efficient techniques will also be presented for reducing the multivariate case to the univariate one, and for computing a root h such that $f = h^r$.

6.2 Testing for perfect powers

In this section we describe a method to determine if a sparse polynomial $f \in \mathsf{R}[x]$ is a perfect power. That is, do there exist $h \in \mathsf{R}[x]$ and $r > 1$ such that $f = h^r$? Importantly, the cost does not depend on the sparsity of h. That is, the root h could have exponentially more nonzero coefficients than the input polynomial f. However, some widely-believed conjectures indicate that this will never happen, as we will discuss later. In any case, our algorithms in this section will only compute the power $r \in \mathbb{N}$, not the polynomial root h.

We first describe algorithms to test if an $f \in \mathsf{R}[x]$ is an rth power of some polynomial $h \in \mathsf{R}[x]$, where f and r *are both given* and r is assumed to be prime. We present and analyse variants that work over finite fields $\mathbb{F}_q$ and over $\mathbb{Z}$. In fact, these algorithms for given r are for *black-box* polynomials: they only need to evaluate f at a small number of points. That this evaluation can be done quickly is a very useful property of sparse polynomials over finite fields.

For a sparse polynomial f we then show that, in fact, if h exists at all then r must be small unless $f = x^n$. (By "small", we mean bounded in value by the size of the sparse representation of f.) If f is a non-trivial perfect power, then there certainly exists a prime r such that f is an rth power. So in fact the restrictions that r is small and prime are sufficient to cover all interesting cases, and our method is complete.

6.2.1 Detecting given rth powers

Our main tool in this work is the following theorem which says that, with reasonable probability, a polynomial is an rth power if and only if the modular image of an evaluation in a specially constructed finite field is an rth power. Put another way, if f is not an rth power, then the finite field will have sufficiently many witnesses to this fact.

Theorem 6.1. *Let $\varrho \in \mathbb{Z}$ be a prime power and $r \in \mathbb{N}$ a prime dividing $\varrho - 1$. Suppose that $f \in \mathbb{F}_\varrho[x]$ has degree $n \le 1 + \sqrt{\varrho}/2$ and is* not *a perfect rth power in $\mathbb{F}_\varrho[x]$. Then*

$$R_f^{(r)} = \#\left\{c \in \mathbb{F}_\varrho : f(c) \in \mathbb{F}_\varrho \text{ is an } r\text{th power}\right\} \le \frac{3\varrho}{4}.$$

Proof. The rth powers in $\mathbb{F}_\varrho$ form a subgroup H of $\mathbb{F}_\varrho^*$ of index r and size $(\varrho-1)/r$ in $\mathbb{F}_\varrho^*$. Also, $a \in \mathbb{F}_\varrho^*$ is an rth power if and only if $a^{(\varrho-1)/r} = 1$. We use the method of "completing the sum" from the theory of character sums. We refer to (Lidl and Niederreiter, 1983), Chapter 5, for an excellent discussion of character sums. By a multiplicative character we mean a homomorphism $\chi : \mathbb{F}_\varrho^* \to \mathbb{C}$ which necessarily maps $\mathbb{F}_\varrho$ onto the unit circle. As usual we extend our multiplicative characters χ so that $\chi(0) = 0$, and define the trivial character $\chi_0(a)$ to be 0 when $a = 0$ and 1 otherwise.

For any $a \in \mathbb{F}_\varrho^*$,

$$\frac{1}{r} \sum_{\chi^r = \chi_0} \chi(a) = \begin{cases} 1, & \text{if } a \in H, \\ 0, & \text{if } a \notin H, \end{cases}$$

where χ ranges over all the multiplicative characters of order r on $\mathbb{F}_\varrho^*$ — that is, all characters that are isomorphic to the trivial character on the subgroup H. Thus

$$R_f^{(r)} = \sum_{a\in\mathbb{F}_\varrho^*}\left(\frac{1}{r}\sum_{\chi^r=\chi_0}\chi(f(a))\right) = \frac{1}{r}\sum_{\chi^r=\chi_0}\sum_{a\in\mathbb{F}_\varrho^*}\chi(f(a))$$

$$\le \frac{\varrho}{r} + \frac{1}{r}\sum_{\substack{\chi^r=\chi_0\\ \chi\neq\chi_0}}\left|\sum_{a\in\mathbb{F}_\varrho}\chi(f(a))\right|.$$

Here we use the obvious fact that

$$\sum_{a\in\mathbb{F}_\varrho^*}\chi_0(f(a)) \le \sum_{a\in\mathbb{F}_\varrho}\chi_0(f(a)) = \varrho - d \le \varrho,$$

where d is the number of distinct roots of f in $\mathbb{F}_\varrho$. We next employ the powerful theorem of (Weil, 1948) on character sums with polynomial arguments (see Theorem 5.41 of (Lidl and Niederreiter, 1983)), which shows that if f is *not* a perfect rth power of another polynomial, and χ has order $r > 1$, then

$$\left|\sum_{a\in\mathbb{F}_\varrho}\chi(f(a))\right| \le (n-1)\varrho^{1/2} \le \frac{\varrho}{2},$$

using the fact that we insisted $n \le 1+\sqrt{\varrho}/2$. Summing over the $r-1$ non-trivial characters of order r, we deduce that

$$R_f^{(r)} \le \frac{\varrho}{r} + \frac{r-1}{r}\cdot\frac{\varrho}{2} \le \frac{3\varrho}{4},$$

since $r \ge 2$. □

6.2.2 Certifying specified powers over $\mathbb{F}_q[x]$

Theorem 6.1 allows us to detect when a polynomial $f \in \mathbb{F}_\varrho[x]$ is a perfect rth power, for known r dividing $\varrho - 1$: choose random $\alpha \in \mathbb{F}_\varrho$ and evaluate $\xi = f(\alpha)^{(\varrho-1)/r} \in \mathbb{F}_\varrho$. Recall that $\xi = 1$ if and only if $f(\alpha)$ is an rth power.

Then we have two cases to consider. If f is an rth power, then clearly $f(\alpha)$ is an rth power as well, and for any $\alpha \in \mathbb{F}_\varrho$, we always have $\xi = 1$.

Otherwise, if f is not an rth power, Theorem 6.1 demonstrates that for at least $1/4$ of the elements of $\mathbb{F}_\varrho$, $f(\alpha)$ is not an rth power. Thus, for α chosen randomly from $\mathbb{F}_\varrho$ we would expect $\xi \neq 1$ with probability at least $1/4$.

This idea works whenever the size of the multiplicative group is a multiple of the power r. For coefficients in an arbitrary finite field $\mathbb{F}_q$, where $q-1$ is not divisible by r, we work in a suitably chosen extension finite extension field. First, the requirement that the characteristic of $\mathbb{F}_q$ is strictly greater than $\deg f$ means that $q = p^e$ for some prime p greater than $\deg f$.

Since r must be less than $\deg f$, this implies that $r \nmid p$, and therefore that $r \nmid q$. Then from Fermat's Little Theorem, we know that $r \mid (q^{r-1} - 1)$ and so we construct an extension field $\mathbb{F}_{q^{r-1}}$ over $\mathbb{F}_q$ and proceed as above. Algorithm 6.1 gives a more formal presentation of this idea.

Algorithm 6.1: Perfect rth power over $\mathbb{F}_q$

```
  Input: A prime power $q$, $f \in \mathbb{F}_q[x]$ of degree $n$ such that $\mathrm{char}(\mathbb{F}_q) < n \le 1 + \sqrt{q}/2$, $r \in \mathbb{N}$
         a prime dividing $n$, and $\epsilon \in \mathbb{R}_{>0}$
  Output: true if $f$ is the $r$th power of a polynomial in $\mathbb{F}_q[x]$; otherwise false with
          probability at least $1 - \epsilon$.
1 Find an irreducible $\Gamma \in \mathbb{F}_q[z]$ of degree $r - 1$, successful with probability at least $\epsilon/2$
2 $\varrho \leftarrow q^{r-1}$
3 Define $\mathbb{F}_\varrho = \mathbb{F}_q[z]/(\Gamma)$
4 $m \leftarrow 2.5(1 + \lceil \log_2(1/\epsilon) \rceil)$
5 for $i$ from $1$ to $m$ do
6     Choose random $\alpha \in \mathbb{F}_\varrho$
7     $\xi \leftarrow f(\alpha)^{(\varrho-1)/r} \in \mathbb{F}_\varrho$
8     if $\xi \ne 1$ then return false
9 return true
```

To accomplish Step 1, a number of fast probabilistic methods are available to find irreducible polynomials. We employ the algorithm of (Shoup, 1994). This algorithm requires $O((r^2 \log r + r \log q) \log r \log\log r)$ operations in $\mathbb{F}_q$. It is probabilistic of the Las Vegas type, meaning always correct and probably fast. We modify this in the trivial way to a Monte Carlo algorithm that is always fast and probably correct. That is, we allow the algorithm to execute the specified number of operations, and if no answer has been returned after this time, the algorithm is halted and returns "fail". From the run-time analysis of the Las Vegas algorithm, the probability of failure is at most 1/2, and the algorithm never returns an incorrect answer. This modification allows us to use Shoup's algorithm in our Monte Carlo algorithm. To obtain an irreducible Γ with failure probability at most $\epsilon/2$ we run (our modified) Shoup's algorithm $1 + \lceil \log_2(1/\epsilon) \rceil$ times.

The restriction that $n \le 1 + \sqrt{q}/2$ (or equivalently that $q \ge 4(n-1)^2$) is not at all limiting. If this condition is not met, simply extend $\mathbb{F}_q$ with an extension of degree $\nu = \left\lceil \log_q(4(n-1)^2) \right\rceil$ and perform the algorithm over $\mathbb{F}_{q^\nu}$. At worst, each operation in $\mathbb{F}_{q^\nu}$ requires $O(\mathsf{M}(\log n))$ operations in $\mathbb{F}_q$, which will not be significant in the overall complexity.

Theorem 6.2. *Let q, f, n, r, ϵ be as in the input to the Algorithm 6.1. If f is a perfect rth power the algorithm always reports this. If f is not a perfect rth power then, on any invocation, this is reported correctly with probability at least $1 - \epsilon$.*

Proof. It is clear from the above discussion that the algorithm always works when f is perfect power. When f is not a perfect power, each iteration of the loop will obtain $\xi \ne 1$ (and hence a correct output) with probability at least 1/4. By iterating the loop m times we ensure that the probability of failure is at most $\epsilon/2$. Adding this to the probability that Shoup's algorithm (for Step 1) fails yields a total probability of failure of at most ϵ. □

Theorem 6.3. *On inputs as specified, Algorithm 6.1 requires*

$$O((r\mathsf{M}(r)\log r\log q)\cdot\log(1/\epsilon))$$

operations in $\mathbb{F}_q$, plus the cost to evaluate $f(\alpha)$ at $O(\log(1/\epsilon))$ points $\alpha\in\mathbb{F}_{q^{r-1}}$.

Proof. As noted above, Shoup's 1994 algorithm requires $O((r^2\log r+r\log q)\log r\log\log r)$ field operations per iteration, which is within the time specified. The main cost of the loop in Steps 5–8 is computing $f(\alpha)^{(\varrho-1)/r}$, which requires $O(\log\varrho)$ or $O(r\log q)$ operations in $\mathbb{F}_\varrho$ using repeated squaring, plus one evaluation of f at a point in $\mathbb{F}_\varrho$. Each operation in $\mathbb{F}_\varrho$ requires $O(\mathsf{M}(r))$ operations in $\mathbb{F}_q$, and we repeat the loop $O(\log(1/\epsilon))$ times. □

Combining these facts, we have that our Monte Carlo algorithm for perfect power testing is correct and works over any finite field of sufficiently large characteristic.

Corollary 6.4. *Given $f\in\mathbb{F}_q[x]$ of degree n with $\tau(f)=t$, and $r\in\mathbb{N}$ a prime dividing n, we can determine if f is an rth power with*

$$O\left((r\mathsf{M}(r)\log r\log q+t\mathsf{M}(r)\log n)\cdot\log(1/\epsilon)\right)$$

operations in $\mathbb{F}_q$, provided $n>\operatorname{char}(\mathbb{F}_q)$. When f is an rth power, the output is always correct, while if f is not an rth power, the output is correct with probability at least $1-\epsilon$.

The results here are counting field operations on an algebraic IMM. However, since the field is specified carefully as $\mathbb{F}_q$, and we know how to represent such elements on the integer side, we could demand all computation be performed on a normal IMM and count word operations. The consequences for the complexity would be that the cost as stated is increased by a factor of $O(\mathsf{N}(q))$, but every $\mathsf{M}(r)$ can be written simply as $O(r\log r)$, as discussed above.

6.2.3 Certifying specified powers over $\mathbb{Z}[x]$

For an integer polynomial $f\in\mathbb{Z}[x]$, we proceed by working in the homomorphic image of $\mathbb{Z}$ in $\mathbb{F}_p$ (and then in an extension of that field). We must ensure that the homomorphism preserves the perfect power property we are interested in, at least with high probability. For any polynomial $g\in\mathsf{F}[x]$, let $\operatorname{disc}(g)=\operatorname{res}(g,g')$ be the discriminant of g (the resultant of g and its first derivative). It is well known that g is squarefree if and only if $\operatorname{disc}(g)\neq 0$ (see e.g., von zur Gathen and Gerhard, 2003, §15.2). Also define $\operatorname{lcoeff}(g)$ as the leading coefficient of g, the coefficient of the highest power of x in g. Finally, for $g\in\mathbb{Z}[x]$ and p a prime, denote by $g\operatorname{rem}p$ the unique polynomial in $\mathbb{F}_p[x]$ with all coefficients in g reduced modulo p.

Lemma 6.5. *Let $f\in\mathbb{Z}[x]$ and $\tilde{f}=f/\gcd(f,f')$ its squarefree part. Let p be a prime such that $p\nmid\operatorname{disc}(\tilde{f})$ and $p\nmid\operatorname{lcoeff}(f)$. Then f is a perfect power in $\mathbb{Z}[x]$ if and only if $f\operatorname{rem}p$ is a perfect power in $\mathbb{F}_p[x]$.*

Proof. Clearly if f is a perfect power, then $f \operatorname{rem} p$ is a perfect power in $\mathbb{Z}[x]$. To show the converse, assume that $f = f_1^{s_1} \cdots f_m^{s_m}$ for distinct irreducible $f_1, \ldots, f_m \in \mathbb{Z}[x]$, so $\tilde{f} = f_1 \cdots f_m$. Clearly $f \equiv f_1^{s_1} \cdots f_m^{s_m} \bmod p$ as well, and because $p \nmid \operatorname{lcoeff}(f)$ we know $\deg(f_i \operatorname{rem} p) = \deg f_i$ for $1 \le i \le m$. Since $p \nmid \operatorname{disc}(\tilde{f})$, $\tilde{f} \operatorname{rem} p$ is squarefree (see (von zur Gathen and Gerhard, 2003), Lemma 14.1), and each of the $f_i \operatorname{rem} p$ must be pairwise relatively prime and squarefree for $1 \le i \le m$. Now suppose $f \operatorname{rem} p$ is a perfect rth power modulo p. Then we must have $r \mid s_i$ for $1 \le i \le m$. But this immediately implies that f is a perfect power in $\mathbb{Z}[x]$ as well. □

Given any polynomial $g = g_0 + g_1 x + \cdots + g_m x^m \in \mathbb{Z}[x]$, we define the height or coefficient ∞-norm of g as $\left\|g\right\|_\infty = \max_i |g_i|$. Similarly, we define the coefficient 1-norm of g as $\left\|g\right\|_1 = \sum_i |g_i|$, and 2-norm as $\left\|g\right\|_2 = \left(\sum_i |g_i|^2\right)^{1/2}$. With $f, \tilde{f}$ as in Lemma 6.5, $\tilde{f}$ divides f, so we can employ the factor bound of (Mignotte, 1974) to obtain

$$\left\|\tilde{f}\right\|_\infty \le 2^n \left\|f\right\|_2 \le 2^n \sqrt{n+1} \cdot \left\|f\right\|_\infty .$$

Since $\operatorname{disc}(\tilde{f}) = \operatorname{res}(\tilde{f}, \tilde{f}')$ is the determinant of matrix of size at most $(2n-1) \times (2n-1)$, Hadamard's inequality implies

$$|\operatorname{disc}(\tilde{f})| \le \left(2^n (n+1)^{1/2} \left\|f\right\|_\infty\right)^{n-1} \left(2^n (n+1)^{3/2} \left\|f\right\|_\infty\right)^n < 2^{2n^2} (n+1)^{2n} \cdot \left\|f\right\|_\infty^{2n} .$$

Also observe that $|\operatorname{lcoeff}(f)| \le \left\|f\right\|_\infty$. Thus, the product $\operatorname{disc}(\tilde{f}) \cdot \operatorname{lcoeff}(f)$ has at most

$$\mu = \left\lceil \frac{\left\lceil \log_2 \left(2^{2n^2} (n+1)^{2n} \left\|f\right\|_\infty^{2n+1}\right) \right\rceil}{\left\lfloor \log_2 \left(4(n-1)^2\right) \right\rfloor} \right\rceil$$

prime factors greater than $4(n-1)^2$ (we require the lower bound $4(n-1)^2$ to employ Theorem 6.1 without resorting to field extensions). Choose an integer $\gamma \ge 4(n-1)^2$ such that the number of primes between γ and 2γ is at least $4\mu + 1$. By (Rosser and Schoenfeld, 1962), Corollary 3, the number of primes in this range is at least $3\gamma/(5 \ln \gamma)$ for $\gamma \ge 21$.

Now let $\gamma \ge \max\{21 \mu \ln \mu, 226\}$. It is easily confirmed that if $\mu \le 6$ and $\gamma \ge 226$, then $3\gamma/(5 \ln \gamma) > 4\mu + 1$. Otherwise, if $\mu \ge 7$, then $\ln(21 \ln \mu) < 2 \ln \mu$, so

$$\frac{\gamma}{\ln \gamma} \ge \frac{21 \mu \ln \mu}{\ln \mu + \ln(21 \ln \mu)} > 7\mu,$$

and therefore $3\gamma/(5 \ln \gamma) > 21\mu/5 > 4\mu + 1$.

Thus, if $\gamma \ge \max\{21 \mu \ln \mu, 226\}$, then a random prime not equal to r in the range $\gamma \ldots 2\gamma$ divides $\operatorname{lcoeff}(f) \cdot \operatorname{disc}(f)$ with probability at most $1/4$. Primes p of this size have only $\log_2 p \in O(\log n + \log\log \left\|f\right\|_\infty)$ bits.

Theorem 6.6. *Let $f \in \mathbb{Z}[x]$ of degree n, $r \in \mathbb{N}$ dividing n and $\epsilon \in \mathbb{R}_{>0}$. If f is a perfect rth power, then Algorithm 6.2 always reports this. If f is not a perfect rth power, on any invocation of the algorithm, this is reported correctly with probability at least $1 - \epsilon$.*

Algorithm 6.2: Perfect rth power over $\mathbb{Z}$

```
Input: $f \in \mathbb{Z}[x]$ of degree $n$; $r \in \mathbb{N}$ a prime dividing $n$; $\epsilon \in \mathbb{R}_{>0}$;
Output: true if $f$ is the $r$th power of a polynomial in $\mathbb{Z}[x]$; false otherwise
1 $\mu \leftarrow \left\lceil \left\lceil \log_2 \left( 2^{2n^2} (n+1)^{2n} \|f\|_\infty^{2n+1} \right) \right\rceil / \left\lfloor \log_2 \left( 4(n-1)^2 \right) \right\rfloor \right\rceil$
2 $\gamma \leftarrow \max\{\lceil 21\mu \ln \mu \rceil, 4(n-1)^2, 226\}$
3 for $i$ from 1 to $\lceil \log_2(1/\epsilon) \rceil$ do
4     $p \leftarrow$ random prime in the range $\gamma \ldots 2\gamma$
5     if Algorithm 6.1 returns false on input $(p, f \operatorname{rem} p, r, 1/4)$ then return false
6 return true
```

Proof. If f is an rth power then so is $f \operatorname{rem} p$ for any prime p, and so is any $f(\alpha) \in \mathbb{F}_p$. Thus, the algorithm always reports that f is an rth power. Now suppose f is not an rth power. If $p \mid \operatorname{disc}(f)$ or $p \mid \operatorname{lcoeff}(f)$ it may happen that $f \operatorname{rem} p$ *is* an rth power. This happens with probability at most $1/4$ and we will assume that the worst happens in this case. When $p \nmid \operatorname{disc}(f)$ and $p \nmid \operatorname{lcoeff}(f)$, the probability that Algorithm 6.1 incorrectly reports that f is an rth power is also at most $1/4$, by our choice of parameter ϵ in the call on step 5. Thus, on any iteration of steps 3–5, the probability of finding that f is an rth power is at most $1/2$. The probability of this happening $\lceil \log_2(1/\epsilon) \rceil$ times is at most ϵ. □

Theorem 6.7. *On inputs as specified, Algorithm 6.2 requires*

$$O\left(r^2 \log^2 r \cdot (\log n + \log\log \|f\|_\infty)^2 \cdot (\log\log n + \log\log\log \|f\|_\infty) \cdot \log(1/\epsilon) \right)$$

or $\tilde{O}(r^2 \cdot (\operatorname{size}(f))^2 \cdot \log(1/\epsilon))$ *word operations in the IMM model, plus the cost to evaluate* $f(\alpha) \bmod p$ *at* $O(\log(1/\epsilon))$ *points* $\alpha \in \mathbb{F}_p$ *for primes* p *with* $\log p \in O(\log n + \log\log \|f\|_\infty)$.

Proof. The number of operations required by each iteration is dominated by Step 5, for which $O(r\mathsf{M}(r) \log r \log p)$ operations in $\mathbb{F}_p$ are sufficient by Theorem 6.3. Since $\log p \in O(\log n + \log\log \|f\|_\infty)$, and using the multiplication algorithm of Section 1.4, we obtain the final complexity as stated. □

The cost of evaluating a t-sparse polynomial $f \in \mathbb{Z}[x]$ modulo a prime p is

$$O(t \cdot \operatorname{size}(\|f\|_\infty) \cdot \operatorname{size}(p) + t \log n \cdot \mathsf{N}(p))$$

word operations, which is $\tilde{O}(\operatorname{size}(f) \cdot \operatorname{size}(p))$. Furthermore, from the theorem, we see that each prime has size bounded by $\operatorname{size}(f)$. We then obtain the following corollary for t-sparse polynomials in $\mathbb{Z}[x]$.

Corollary 6.8. *Given* $f \in \mathbb{Z}[x]$ *of degree* n *and* $r \in \mathbb{N}$ *a prime dividing* n, *we can determine if* f *is an* r*th power with*

$$\tilde{O}\left(r^2 \cdot (\operatorname{size} f)^2 \cdot \log(1/\epsilon) \right)$$

machine word operations. When f *is an* r*th power, the output is always correct, while if* f *is not an* r*th power, the output is correct with probability at least* $1-\epsilon$.

6.2.4 An upper bound on r

The algorithms we have seen so far require the power r to be known in advance. To adapt these to the general situation that r is not known, we show that r must be small compared to the sparse representation size of f, and therefore there are not too many "guesses" of r that we must make in order to solve the problem.

Specifically, we show in this subsection that if $f = h^r$ and $f \neq x^n$ then the value of r is polynomially bounded by $\tau(f)$. Over $\mathbb{Z}[x]$ we show that $\|h\|_2$ is small as well. A sufficiently strong result over many fields is demonstrated by (Schinzel, 1987), Theorem 1, where it is shown that if f has sparsity $t \geq 2$ then $t \geq r+1$ (in fact a stronger result is shown involving the sparsity of h as well). This holds when either the characteristic of the ground field of f is zero or greater than $\deg f$.

Here we give a (much) simpler result for polynomials in $\mathbb{Z}[x]$, which bounds $\|h\|_2$ and is stronger at least in its dependency on t though it also depends upon the size of coefficients in f.

Theorem 6.9. *Suppose $f \in \mathbb{Z}[x]$ with $\deg f = n$ and $\tau(f) = t$, and $f = h^r$ for some $h \in \mathbb{Z}[x]$ of degree s and $r \geq 2$. Then $\|h\|_2 \leq \left\|f\right\|_1^{1/r}$.*

Proof. Let $p > n$ be prime and $\zeta \in \mathbb{C}$ a pth primitive root of unity. Then

$$\|h\|_2^2 = \sum_{0\leq i\leq s} |h_i|^2 = \frac{1}{p}\sum_{0\leq i<p} |h(\zeta^i)|^2.$$

(this follows from the fact that the Discrete Fourier Transform (DFT) matrix is orthogonal). In other words, the average value of $|h(\zeta^i)|^2$ for $i = 0\ldots p-1$ is $\|h\|_2^2$, and so there exists a $k \in \{0,\ldots,p-1\}$ with $|h(\zeta^k)|^2 \geq \|h\|_2^2$. Let $\theta = \zeta^k$. Then clearly $|h(\theta)| \geq \|h\|_2$. We also note that $f(\theta) = h(\theta)^r$ and $|f(\theta)| \leq \left\|f\right\|_1$, since $|\theta| = 1$. Thus,

$$\|h\|_2 \leq |h(\theta)| = |f(\theta)|^{1/r} \leq \left\|f\right\|_1^{1/r}. \qquad \square$$

The following corollary is particularly useful.

Corollary 6.10. *If $f \in \mathbb{Z}[x]$ is not of the form x^n, and $f = h^r$ for some $h \in \mathbb{Z}[x]$, then*

(i) $r \leq 2\log_2 \left\|f\right\|_1$,

(ii) $\tau(h) \leq \left\|f\right\|_1^{2/r}$.

Proof. Part (i) follows since $\|h\|_2 \geq \sqrt{2}$. Part (ii) follows because $\|h\|_2 \geq \sqrt{\tau(h)}$. □

These bounds relate to the sparsity of f since $\left\|f\right\|_1 \leq \tau(f)\left\|f\right\|_\infty$.

Algorithm 6.3: Perfect power detection over $\mathbb{Z}$

```
Input: $f \in \mathbb{Z}[x]$ of degree $n$ and sparsity $t \geq 2$, $\epsilon \in \mathbb{R}_{>0}$
Output: true and $r$ if $f = h^r$ for some $h \in \mathbb{Z}[x]$; false otherwise.
$\mathscr{P} \leftarrow \{\text{primes } r \mid n \text{ and } r \leq 2\log_2(t\,\|f\|_\infty)\}$
for $r \in \mathscr{P}$ do
    if Algorithm 6.2 returns true on input $(f, r, \epsilon/\#\mathscr{P})$ then return true and $r$
return false
```

6.2.5 Perfect Power Detection Algorithm

We can now complete the perfect power detection algorithm, when we are given only the t-sparse polynomial f (and not r).

Theorem 6.11. *If $f \in \mathbb{Z}[x] = h^r$ for some $h \in \mathbb{Z}[x]$, then Algorithm 6.3 always returns "True" and returns r correctly with probability at least $1-\epsilon$. Otherwise, it returns "False" with probability at least $1-\epsilon$. The algorithm requires $\tilde{O}((\text{size}(f))^4 \cdot \log(1/\epsilon))$ word operations.*

Proof. From the preceding discussions, we can see that if f is a perfect power, then it must be a perfect rth power for some $r \in \mathscr{P}$. So the algorithm must return true on some iteration of the loop. However, it may incorrectly return true *too early* for an r such that f is not actually an rth power; the probability of this occurring is the probability of error when f is not a perfect power, and is less than $\epsilon/\#\mathscr{P}$ at each iteration. So the probability of error on any iteration is at most ϵ, which is what we wanted.

The complexity result follows from the fact that each $r \in \tilde{O}(\text{size}(f))$ and using Corollary 6.8. □

We now turn to the case of finite fields. Here we rely on Schinzel's bound that $r \leq t-1$, and obtain the following algorithm.

Algorithm 6.4: Perfect power detection over $\mathbb{F}_q$

```
Input: A prime power $q$, $f \in \mathbb{F}_q[x]$ of degree $n$ and sparsity $t$ such that $n < \text{char}(\mathbb{F}_q)$,
       and $\epsilon \in \mathbb{R}_{>0}$
Output: true and $r$ if $f = h^r$ for some $h \in \mathbb{F}_q[x]$; false otherwise.
$\mathscr{P} \leftarrow \{\text{primes } r \mid n \text{ and } r \leq t\}$
for $p \in \mathscr{P}$ do
    if Algorithm 6.1 returns true on input $(f, r, \epsilon/\#\mathscr{P})$ then
        return true and $r$

return false
```

Theorem 6.12. *If $f = h^r$ for $h \in \mathbb{F}_q[x]$, then Algorithm 6.4 always returns "True" and returns r correctly with probability at least $1-\epsilon$. Otherwise, it returns "False" with probability at least $1-\epsilon$. The algorithm requires $\tilde{O}(t^3(\log q + \log n))$ operations in $\mathbb{F}_q$.*

Proof. The proof is equivalent to that of Theorem 6.11, using the complexity bounds in Corollary 6.4. □

6.2.6 Detecting multivariate perfect powers

In this subsection we examine the problem of detecting multivariate perfect powers. That is, given a lacunary $f \in \mathsf{F}[x_1,\ldots,x_\ell]$ of total degree n as in (6.1), we want to determine if $f = h^r$ for some $h \in \mathsf{F}[x_1,\ldots,x_\ell]$ and $r \in \mathbb{N}$. This is done simply as a reduction to the univariate case.

First, given $f \in \mathsf{F}[x_1,\ldots,x_\ell]$, define the squarefree part $\tilde{f} \in \mathsf{F}[x_1,\ldots,x_\ell]$ as the squarefree polynomial of highest total degree which divides f.

Lemma 6.13. *Let $f \in \mathsf{F}[x_1,\ldots,x_\ell]$ be of total degree $n > 0$ and let $\tilde{f} \in \mathsf{F}[x_1,\ldots,x_\ell]$ be the squarefree part of f. Define*

$$\Delta = \operatorname{disc}_x(\tilde{f}(y_1x,\ldots,y_\ell x)) = \operatorname{res}_x(\tilde{f}(y_1x,\ldots,y_\ell x), \tilde{f}'(y_1x,\ldots,y_\ell x)) \in \mathsf{F}[y_1,\ldots,y_\ell]$$

and

$$\Lambda = \operatorname{lcoeff}_x(f(y_1x,\ldots,y_\ell x)) \in \mathsf{F}[y_1,\ldots,y_\ell]$$

for independent indeterminates $x, y_1,\ldots,y_\ell$. Assume that $a_1,\ldots,a_\ell \in \mathsf{F}$ with $\Delta(a_1,\ldots,a_\ell) \neq 0$ and $\Lambda(a_1,\ldots,a_n) \neq 0$. Then $f(x_1,\ldots,x_\ell)$ is a perfect power if and only if $f(a_1x,\ldots,a_\ell x) \in \mathsf{F}[x]$ is a perfect power.

Proof. Clearly if f is a perfect power, then $f(a_1x,\ldots,a_\ell x)$ is a perfect power. To prove the converse, assume that

$$f = f_1^{s_1} f_2^{s_2} \cdots f_m^{s_m}$$

for irreducible $f_1,\ldots,f_m \in \mathsf{F}[x_1,\ldots,x_\ell]$. Then

$$f(y_1x,\ldots,y_m x) = f_1(y_1x,\ldots,y_m x)^{s_1} \cdots f_m(y_1x,\ldots,y_m x)^{s_m}$$

and each of the $f_i(y_1x,\ldots,y_m x)$ are irreducible. Now, since $\Lambda(a_1,\ldots,a_m) \neq 0$, we know the $\deg(f(a_1x,\ldots,a_\ell x)) = \deg f$ (the total degree of f). Thus, $\deg f_i(a_1x,\ldots,a_\ell x) = \deg f_i$ for $1 \leq i \leq \ell$ as well. Also, by our assumption, $\operatorname{disc}(f(a_1x,\ldots,a_\ell x)) \neq 0$, so all of the $f_i(a_1x,\ldots,a_\ell x)$ are squarefree and pairwise relatively prime for $1 \leq i \leq k$, and

$$f(a_1x,\ldots,a_\ell x) = f_1(a_1x,\ldots,a_\ell x)^{s_1} \cdots f_m(a_1x,\ldots,a_\ell x)^{s_m}.$$

Assume now that $f(a_1x,\ldots,a_\ell x)$ is an rth perfect power. Then r divides s_i for $1 \leq i \leq m$. This immediately implies that f itself is an rth perfect power. □

It is easy to see that the total degree of Δ is less than $2n^2$ and the total degree of Λ is less than n, and that both Δ and Λ are non-zero. Thus, for randomly chosen $a_1,\ldots,a_\ell$ from a set $\mathscr{S} \subseteq \mathsf{F}$ of size at least $8n^2 + 4n$ we have $\Delta(a_1,\ldots,a_\ell) = 0$ or $\Lambda(a_1,\ldots,a_\ell) = 0$ with probability less than 1/4, by the famous Schwartz-Zippel lemma (Demillo and Lipton, 1978; Zippel, 1979;

Schwartz, 1980). This can be made arbitrarily small by increasing the set size and/or by repetition. We then run the appropriate univariate algorithm over $\mathsf{R}[x]$ (depending upon the ring) to identify whether or not f is a perfect power, and if so, to find r.

For integer polynomials, $f(a_1x,\ldots,a_\ell x)$ will *not* be explicitly computed over $\mathbb{Z}[x]$, as the size of the coefficients in this univariate polynomial could be exponentially larger than the sparse representation size of f itself. Instead, we pass $f(a_1x,\ldots,a_\ell x)$ unevaluated to Algorithm 6.2, which only performs the computation once a small finite field has been chosen. This preserves polynomial-time for sparse multivariate integer polynomials.

6.3 Computing perfect roots

Once we have determined that $f \in \mathsf{F}[x]$ is equal to h^r for some $h \in \mathsf{F}[x]$, the next task is to actually compute h. Unfortunately, as noted in the introduction, there are no known bounds on $\tau(h)$ which are polynomial in $\tau(f)$.

The question of how sparse the polynomial root of a sparse polynomial must be (or equivalently, how dense any power of a dense polynomial must be) relates to some questions first raised by Erdös (1949) on the number of terms in the square of a polynomial. Schinzel extended this work to the case of perfect powers and proved that $\tau(h^r)$ tends to infinity as $\tau(h)$ tends to infinity (Schinzel, 1987). Some conjectures of Schinzel suggest that $\tau(h)$ should be $O(\tau(f))$. A recent breakthrough of Zannier (2007) shows that $\tau(h)$ is bounded by a function which does not depend on $\deg f$, but this bound is unfortunately not polynomial in $\tau(f)$.

However, our own (limited) investigations, along with more extensive ones by Coppersmith and Davenport (1991), and later Abbott (2002), suggest that, for any $h \in \mathsf{F}[x]$, where the characteristic of F is not too small, $\tau(h) \in O(\tau(h^r)+r)$. The first algorithm presented here avoids this problem by being output-sensitive. That is, the cost of the algorithm is proportional to $\tau(h)$, whatever that might be. We then present different algorithm for this problem that will be much more efficient in practice, but whose analysis relies on a modest conjecture.

6.3.1 Computing rth roots in output-sensitive polynomial time

In this subsection we present an algorithm for computing an h such that $f = h^r$ given $f \in \mathbb{Z}[x]$ and $r \in \mathbb{Z}$ or showing that no such h exists. The algorithm is deterministic and requires time polynomial in size(f) as well as a given upper bound μ on $m = \tau(h)$. Neither its correctness nor complexity is conditional on any conjectures. We will only demonstrate that this algorithm requires polynomial time, as the (conjecturally fast) algorithm of the next chapter will be the obvious choice in practical situations.

The basic idea of the algorithm here is that we can recover all the coefficients in $\mathbb{Q}$ as well as modular information about the exponents of h from a homomorphism into a small cyclotomic field over $\mathbb{Q}$. Doing this for a relatively small number of cyclotomic fields yields h.

Assume that (the unknown) $h \in \mathbb{Z}[x]$ is written as

$$h = b_1x^{d_1} + b_2x^{d_2} + \cdots + b_mx^{d_m},$$

for $b_1,\ldots,b_m \in \mathbb{Z}^*$, and $0 \leq d_1 < d_2 < \cdots < d_m$,. Also assume that $p > 2$ is a prime distinct from r such that

$$p \nmid \prod_{1\leq i<j\leq m} (d_j - d_i), \text{ and } p \nmid \prod_{1\leq i\leq m} (d_i + 1). \tag{6.5}$$

Let $\zeta_p \in \mathbb{C}$ be a pth primitive root of unity, and $\Phi_p = 1 + z + \cdots + z^{p-1} \in \mathbb{Z}[z]$ its minimal polynomial, the pth cyclotomic polynomial (which is irreducible in $\mathbb{Q}[z]$). Computationally we represent $\mathbb{Q}(\zeta_p)$ as $\mathbb{Q}[z]/(\Phi_p)$, with $\zeta_p \equiv z \operatorname{rem} \Phi_p$. Observe that $\zeta_p^k = \zeta_p^{k \operatorname{rem} p}$ for any $k \in \mathbb{Z}$, where $k \operatorname{rem} p$ is the least non-negative residue of k modulo p. Thus, if we define

$$h_p = \sum_{1\leq i\leq m} b_i x^{d_i \operatorname{rem} p} \in \mathbb{Z}[x],$$

then $h(\zeta_p) = h_p(\zeta_p)$, and h_p is the unique representation of $h(\zeta_p)$ as a polynomial of degree less than $p - 1$. This follows from the conditions (6.5) on our choice of prime p because

- No pair of distinct exponents d_i and d_j of h is equivalent modulo p (since $p \nmid (d_i - d_j)$), and
- All the exponents reduced modulo p are strictly less than $p - 1$ (since our conditions imply $d_i \not\equiv (p - 1) \bmod p$ for $1 \leq i \leq m$).

This also implies that the coefficients of h_p are exactly the same as those of h, albeit in a different order.

Now observe that we can determine h_p quite easily from the roots of

$$\Gamma_p(y) = y^r - f(\zeta_p) \in \mathbb{Q}(\zeta_p)[y].$$

These roots can be found by factoring the polynomial $\Gamma_p(y)$ in the algebraic extension field $\mathbb{Q}(\zeta_p)[y]$, and the roots in $\mathbb{C}$ must be $\omega^i h(\zeta_p) \in \mathbb{C}$ for $0 \leq i < r$, where ω is a primitive rth root of unity. When $r > 2h_p = \sum_{1\leq i\leq m} b_i x^{d_i \operatorname{rem} p} \in \mathbb{Z}[x]$,, and since we chose p distinct from r, the only rth root of unity in $\mathbb{Q}(\zeta_p)$ is 1. Thus $\Gamma_p(y)$ has exactly one linear factor, and this must be equal to $y - h(\zeta_p) = y - h_p(\zeta_p)$, precisely determining h_p. When $r = 2$, we have

$$\Gamma_p(y) = (y - h(\zeta_p))(y + h(\zeta_p)) = (y - h_p(\zeta_p))(y + h_p(\zeta_p)),$$

and we can only determine $h_p(\zeta_p)$ (and h_p and, for that matter, h) up to a factor of ± 1. However, the exponents of h_p and $-h_p$ are the same, and the ambiguity is only in the coefficients (which we resolve later).

Finally, we need to repeatedly perform the above operations for a sequence of cyclotomic fields $\mathbb{Q}(\zeta_{p_1}), \mathbb{Q}(\zeta_{p_2}), \ldots, \mathbb{Q}(\zeta_{p_k})$ such that the primes in $\mathscr{P} = \{p_1,\ldots,p_k\}$ allow us to recover all the exponents in h. Each prime $p \in \mathscr{P}$ gives the set of exponents of h reduced modulo that prime, as well as *all* the coefficients of h in $\mathbb{Z}$. That is, from each of the computations with $p \in \mathscr{P}$ we obtain

$$\mathscr{C} = \{b_1,\ldots,b_m\} \quad \text{and} \quad \mathscr{E}_p = \left\{d_1 \operatorname{rem} p, d_2 \operatorname{rem} p, \ldots, d \operatorname{rem} p\right\},$$

but with no clear information about the order of these sets. In particular, it is not obvious how to correlate the exponents modulo the different primes directly. To do this we employ the clever sparse interpolation technique of (Garg and Schost, 2009) (based on a method of Grigoriev and Karpinski (1987) for a different problem), which interpolates the symmetric polynomial in the exponents:

$$g = (x - d_1)(x - d_2) \cdots (x - d_m) \in \mathbb{Z}[x].$$

For each $p \in \mathscr{P}$ we compute the symmetric polynomial modulo p,

$$g_p = (x - (d_1 \operatorname{rem} p))(x - (d_2 \operatorname{rem} p)) \cdots (x - (d_m \operatorname{rem} p)) \equiv g \pmod{p},$$

for which we do not need to know the order of the exponent residues. We then determine $g \in \mathbb{Z}[x]$ by the Chinese remainder theorem and factor g over $\mathbb{Z}[x]$ to find the $d_1, \ldots, d_m \in \mathbb{Z}$. Thus the product of all primes in $p \in \mathscr{P}$ must be at least $2\left\|g\right\|_\infty$ to recover the coefficients of g uniquely. It is easily seen that $2\left\|g\right\|_\infty \leq 2n^m$.

As noted above, the computation with each $p \in \mathscr{P}$ recovers all the exponents of h in $\mathbb{Z}$, so using only one prime $p \in \mathscr{P}$, we determine the jth exponent of h as the coefficient of $x^{d_j \operatorname{rem} p}$ in h_p for $1 \leq j \leq m$. If $r = 2$ we can choose either of the roots of $\Gamma_p(y)$ (they differ by only a sign) to recover the coefficients of h.

Finally, once we have a candidate root h, we certify that $f = h^r$ by taking logarithmic derivatives to obtain

$$\frac{f'}{f} = \frac{r h' h^{r-1}}{h^r},$$

which simplifies to $f'h = rh'f$. This equation only involves two sparse multiplications and is therefore confirmed in polynomial time, and along with checking leading coefficients implies that in fact $f = h^r$.

The above discussion is summarized in the following algorithm.

Theorem 6.14. *Algorithm 6.5 works as stated. It requires a number of word operations polynomial in* size(f) *and* μ.

Proof. We assume throughout the proof that there *does* exist an $h \in \mathbb{Z}[x]$ such that $f = h^r$ and $\tau(h) \leq \mu$. If it does not, this will be caught in the test in Steps 22–24 by the above discussion, if not before.

In Steps 1–2 we construct a set of primes $\mathscr{P}$ which is guaranteed to contain sufficiently many *good* primes to recover g, where primes are good in the sense that for all $p \in \mathscr{P}$

$$\beta = r \cdot \prod_{1 \leq i < j \leq m} (d_j - d_i) \cdot \prod_{1 \leq i \leq m} (d_i + 1) \not\equiv 0 \bmod p.$$

It is easily derived that $\beta < n^{\mu^2}$, which has fewer than $\log_2 \beta \leq \mu^2 \log_2 n$ prime factors, so there are at most $\mu^2 \log_2 n$ *bad* primes. We also need to recover g in Step 16, and $\left\|g\right\|_\infty \leq n^\mu$, for which we need at least $1 + \log_2 \left\|g\right\| \leq 2\mu \log_2 n$ good primes. Thus if $\mathscr{P}$ has at least $(\mu^2 + 2\mu) \log_2 n$ primes, there are a sufficient number of good primes to reconstruct g in Step 16.

Algorithm 6.5: Algebraic algorithm for computing perfect roots

```
   Input: $f \in \mathbb{Z}[x]$ as in (6.2) with $\deg f = n$, and $r, \mu \in \mathbb{N}$
   Output: $h \in \mathbb{Z}[x]$ such that $f = h^r$ and $\tau(h) \le \mu$, provided such an $h$ exists
1  $\gamma \leftarrow$ smallest integer $\ge 21$ such that $3\gamma/(5\ln\gamma) \ge (\mu^2 + 2\mu)\log_2 n$
2  $\mathscr{P} \leftarrow \{p \in \{\gamma, \ldots, 2\gamma\}$ and $p$ prime$\}$
3  for $p \in \mathscr{P}$ do
4      Represent $\mathbb{Q}(\zeta_p)$ by $\mathbb{Q}[x]/(\Phi_p)$, where $\Phi_p \leftarrow 1 + z + \cdots + z^{p-1}$ and $\zeta_p \equiv z \bmod \Phi_p$
5      Compute $f(\zeta_p) = \sum_{1 \le i \le t} c_i \zeta_p^{e_i \operatorname{rem} p} \in \mathbb{Z}[\zeta_p]$
6      Factor $\Gamma_p(y) \leftarrow y^r - f(\zeta_p) \in \mathbb{Q}(\zeta_p)[y]$ over $\mathbb{Q}(\zeta_p)[y]$
7      if $\Gamma_p(y)$ has no roots in $\mathbb{Z}[\zeta_p]$ then
8          return "$f$ is not an $r$th power of a $\mu$-sparse polynomial"
9      Let $h_p(\zeta_p) \in \mathbb{Z}[\zeta_p]$ be a root of $\Gamma_p(y)$
10     Write $h_p(x) = \sum_{1 \le i \le m_p} b_{ip} x^{d_{ip}}$, for $b_{ip} \in \mathbb{Z}$ and distinct $d_{ip} \in \mathbb{N}$ for $1 \le i \le m_p$
11     if $\deg h_p = p - 1$ then
12         $m_p \leftarrow 0$; Continue with next prime $p \in \mathscr{P}$ at Step 3
13     $g_p \leftarrow (x - d_{1p})(x - d_{2p}) \cdots (x - d_{m_p p}) \in \mathbb{Z}_p[x]$
14 $m \leftarrow \max\{m_p : p \in \mathscr{P}\}$
15 $\mathscr{P}_0 \leftarrow \{p \in \mathscr{P} : m_p = m\}$
16 Reconstruct $g \in \mathbb{Z}[x]$ from $\{g_p\}_{p \in \mathscr{P}_0}$ by the Chinese Remainder Algorithm
17 $\{d_1, d_2, \ldots, d_k\} \leftarrow$ distinct integer roots of $g$
18 if $k < m$ then
19     return "$f$ is not an $r$th power of a $\mu$-sparse polynomial"
20 Choose any $p \in \mathscr{P}_0$. For $1 \le j \le m$, let $b_j \in \mathbb{Z}$ be the coefficient of $x^{d_j \operatorname{rem} p}$ in $h_p$
21 $h \leftarrow \sum_{1 \le j \le m} b_j x^{d_j}$
22 if $f' h = r h' f$ and $\operatorname{lcoeff}(f) = \operatorname{lcoeff}(h)^r$ then
23     return $h$
24 else
25     return "$f$ is not an $r$th power of a $\mu$-sparse polynomial"
```

By (Rosser and Schoenfeld, 1962), Corollary 3, for $\gamma \geq 21$ we have that the number of primes in $\{\gamma, \ldots, 2\gamma\}$ is at least $3\gamma/(5\ln\gamma)$, which is at least $(\mu^2+2\mu)\log_2 n$ by our choice of γ in Step 1, and $\gamma \in \tilde{O}(\mu^2 \log(n))$. Numbers of this size can easily be tested for primality.

Since we assume that a root h exists, $\Gamma_p(y)$ will always have exactly one root $h_p \in \mathbb{Z}[\zeta_p]$ when $r > 2$, and exactly two roots in $\mathbb{Z}[\zeta_p]$ when $r = 2$ (differing only by sign).

Two conditions cause the primes to be identified as bad. If the map $h \mapsto h(\zeta_p)$ causes some exponents of h to collide modulo p, this can only reduce the number of non-zero exponents m_p in h_p, and so such primes will not show up in the list of good primes $\mathscr{P}_0$, as selected in Step 15. Also, if any of the exponents of h are equivalent to $p-1$ modulo p we will not be able to reconstruct the exponents of h from h_p, and we identify these as bad in Step 12 (by artificially marking $m_p = 0$, which ensures they will not be added to $\mathscr{P}_0$).

Correctness of the remainder of the algorithm follows from the previous discussion.

The complexity is clearly polynomial for all steps except for factoring in $\mathbb{Q}(\zeta_p)[y]$ (Step 6). It is well-established that factoring dense polynomials over algebraic extensions can be done in polynomial time, for example using the algorithm of Landau (1985). □

As stated, Algorithm 6.5 is not actually output-sensitive, as it requires an *a priori* bound μ on $\tau(h)$. To avoid this, we could start with any small value for μ, say $\tau(f)$, and after each failure double this bound. Provided that the input polynomial f is in fact an rth perfect power, this process with terminate after a number of steps polynomial in the sparse size of the output polynomial h. There are also a number of other small improvements that could be made to increase the algorithm's efficiency, which we have omitted here for clarity.

6.3.2 Faster root computation subject to conjecture

Algorithm 6.5 is output sensitive as the cost depends on the sparsity of the root h. As discussed above, there is considerable evidence that, roughly speaking, the root of a sparse polynomial must always be sparse, and so the preceding algorithm may be unconditionally polynomial-time.

In fact, with suitable sparsity bounds we can derive a more efficient algorithm based on Newton iteration. This approach is simpler as it does not rely on advanced techniques such as factoring over algebraic extension fields. It is also more general as it applies to fields other than $\mathbb{Z}$ and to powers r which are not prime.

Unfortunately, this algorithm is not purely output-sensitive, as it relies on a conjecture regarding the sparsity of powers of h. We first present the algorithm and prove its correctness. Then we give our modest conjecture and use it to prove the algorithm's efficiency.

Our algorithm is essentially a Newton iteration, with special care taken to preserve sparsity. We start with the image of h modulo x, using the fact that $f(0) = h(0)^r$, and at Step $i = 1, 2, \ldots, \lceil \log_2(\deg h + 1) \rceil$, we compute the image of h modulo x^i.

Here, and for the remainder of this section, we will assume that $f, h \in \mathsf{F}[x]$ with degrees n and s respectively such that $f = h^r$ for $r \in \mathbb{N}$ at least 2, and that the characteristic of F is either zero or greater than n. As usual, we define $t = \tau(f)$.

Algorithm 6.6: Sparsity-sensitive Newton iteration to compute perfect roots

```
   Input: $f \in \mathsf{F}[x]$, $r \in \mathbb{N}$ such that $f$ is a perfect $r$th power
   Output: $h \in \mathsf{F}[x]$ such that $f = h^r$
1  $u \leftarrow$ highest power of $x$ dividing $f$
2  $f_u \leftarrow$ coefficient of $x^u$ in $f$
3  $g \leftarrow f/(f_u x^u)$
4  $h \leftarrow 1, \quad k \leftarrow 1$
5  while $kr \le \deg g$ do
6      $\ell \leftarrow \min\{k, (\deg g)/r + 1 - k\}$
7      $a \leftarrow \dfrac{(hg - h^{r+1}) \operatorname{rem} x^{k+\ell}}{rx^k}$
8      $h \leftarrow h + (a/g \bmod x^\ell) \cdot x^k$
9      $k \leftarrow k + \ell$
10 $b \leftarrow$ any $r$th root of $f_u$ in $\mathsf{F}$
11 return $bhx^{u/r}$
```

Theorem 6.15. *If $f \in \mathsf{F}[x]$ is a perfect rth power, then 6.6 returns an $h \in \mathsf{F}[x]$ such that $h^r = f$.*

Proof. Let u, f_u, g be as defined in Steps 1–3. Thus $f = f_u g x^u$. Now let $\hat{h}$ be some rth root of f, which we assume exists. If we similarly write $\hat{h} = \hat{h}_v \hat{g} x^v$, with $\hat{h}_v \in \mathsf{F}$ and $\hat{g} \in \mathsf{F}[x]$ such that $\hat{g}(0) = 1$, then $\hat{h}^r = {\hat{h}_v}^r \hat{g}^r x^{vr}$. Therefore f_u must be a perfect rth power in F, $r|u$, and g is a perfect rth power in $\mathsf{F}[x]$ of some polynomial with constant coefficient equal to 1.

Denote by h_i the value of h at the beginning of the ith iteration of the while loop. So $h_1 = 1$. We claim that at each iteration through Step 6, $h_i^r \equiv g \bmod x^k$. From the discussion above, this holds for $i = 1$. Assuming the claim holds for all $i = 1, 2, \ldots, j$, we prove it also holds for $i = j + 1$.

From Step 8, $h_{j+1} = h_j + (a/g \bmod x^l)x^k$, where a is as defined on the jth iteration of Step 7. We observe that

$$h_j h_j^r \equiv h_j^{r+1} + r h_j^r (a/g \bmod x^l) x^k \pmod{x^{k+\ell}}.$$

From our assumption, $h_j^r \equiv f \bmod x^k$, and $l \le k$, so we have

$$h_j h_{j+1}^r \equiv h_j^{r+1} + rax^k \equiv h_j^{r+1} + h_j f - h_j^{r+1} \equiv h_j f \pmod{x^{k+\ell}}.$$

Therefore $h_{j+1}^r \equiv f \bmod x^{k+\ell}$, and so by induction the claim holds at each step. Since the algorithm terminates when $kr > \deg g$, we can see that the final value of h is an rth root of g. Finally, $(bhx^{u/r})^r = f_u g x^u = f$. □

Algorithm 6.6 will only be efficient if the low-order terms of the polynomial power h^{r-1} can be efficiently computed on Step 7. Since we know that h and the low-order terms of h^{r-1} are sparse, we need only a guarantee that the *intermediate powers* will be sparse as well. This is stated in the following modest conjecture.

Conjecture 6.16. *For $r, s \in \mathbb{N}$, if the characteristic of* F *is zero or greater than rs, and $h \in \mathsf{F}[x]$ with $\deg h = s$, then*

$$\tau(h^i \bmod x^{2s}) < \tau(h^r \bmod x^{2s}) + r, \qquad i = 1, 2, \ldots, r-1.$$

This corresponds to intuition and experience, as the system is still overly constrained with only s degrees of freedom. Computationally, the conjecture has also been confirmed for all of the numerous examples we have tested, and for the special case of $r = 2$ and $s \leq 12$ by Abbott (2002), although a more thorough investigation of its truth would be interesting. A weaker inequality would suffice to prove polynomial time, but we use the stated bounds as we believe these give more accurate complexity measures.

The application of Conjecture 6.16 to Algorithm 6.6 is given by the following lemma, which essentially tells us that the "error" introduced by examining higher-order terms of h_1^r is not too dense.

Lemma 6.17.[2] *Let $k, \ell \in \mathbb{N}$ such that $\ell \leq k$ and $k + \ell \leq s$, and suppose $h_1 \in \mathsf{F}[x]$ is the unique polynomial with degree less than k satisfying $h_1^r \equiv f \bmod x^k$. Then*

$$\tau(h_l^{r+1} \bmod x^{k+\ell}) \leq 2t(t+r).$$

Proof. Let $h_2 \in \mathsf{F}[x]$ be the unique polynomial of degree less than ℓ satisfying $h_1 + h_2 x^k \equiv h \bmod x^{k+\ell}$. Since $h^r = f$,

$$f \equiv h_1^r + r h_1^{r-1} h_2 x^k \mod x^{k+\ell}.$$

Multiplying by h_1 and rearranging gives

$$h_1^{r+1} \equiv h_1 f - r f h_2 x^k \mod x^{k+\ell}.$$

Because $h_1 \bmod x^k$ and $h_2 \bmod x^\ell$ each have at most $\tau(h)$ terms, which by Conjecture 6.16 is less than $t - r$, the total number of terms in $h_1^{r-1} \bmod x^{k+\ell}$ is less than $2t(t-r)$. □

We are now ready to prove the efficiency of the algorithm, assuming the conjecture.

Theorem 6.18.[2] *If $f \in \mathsf{F}[x]$ has degree n and t nonzero terms, then Algorithm 6.6 uses*

$$O\left((t+r)^4 \log r \log n\right)$$

operations in F *and an additional $O\left((t+r)^4 \cdot \operatorname{size}(r) \cdot \log^2 n\right)$ word operations, not counting the cost of root-finding in the base field* F *on Step 10.*

Proof. First consider the cost of computing h^{r+1} in Step 7. This will be accomplished by repeatedly squaring and multiplying by h, for a total of at most $2\lfloor \log_2(r+1) \rfloor$ multiplications. As well, each intermediate product will have at most $\tau(f) + r < (t+r)^2$ terms, by Conjecture 6.16. The number of field operations required, at each iteration, is $O\left((t+r)^4 \log r\right)$, for a total cost of $O\left((t+r)^4 \log r \log n\right)$.

[2] Subject to the validity of Conjecture 6.16.

Furthermore, since $k+\ell \leq 2^i$ at the i'th step, for $1 \leq i < \log_2 n$, the total cost in word operations is less than

$$\sum_{1\leq i<\log_2 n} (t+r)^4 \cdot \operatorname{size}(ri) \in O\left((t+r)^4 \operatorname{size}(r) \log^2 n\right).$$

In fact, this is the most costly step. The initialization in Steps 1–3 uses only $O(t)$ operations in F and on integers at most n. And the cost of computing the quotient on Step 8 is proportional to the cost of multiplying the quotient and dividend, which is at most $O(t(t+r))$. □

When $\mathsf{F}=\mathbb{Q}$, we must account for coefficient growth. Observe that the difference in the number of word operations on an IMM and the number of bit operations for the same algorithm is at most a factor of the word size w. From the definition of an IMM, we know that this is logarithmic in the size of the input, and therefore does not affect the $\tilde{O}(\cdots)$ complexity.

So for ease of presentation, and in particular in order to make use of Theorem 6.9, we will count the cost of the computation in bit complexity. First we define the height of a polynomial in terms of bit length: For $\alpha \in \mathbb{Q}$, write $\alpha = a/b$ for a,b relatively prime integers. Then define $\mathscr{H}(\alpha) = \max\{|a|,|b|\}$. And for $f \in \mathbb{Q}[x]$ with coefficients $c_1,\ldots,c_t \in \mathbb{Q}$, write $\mathscr{H}(f) = \max \mathscr{H}(c_i)$.

Thus, the size of the lacunary representation of $f \in \mathbb{Q}[x]$ is proportional to $\tau(f)$, $\deg f$, and $\log \mathscr{H}(f)$. Now we prove the complexity of our algorithm is polynomial in these values, when $\mathsf{F}=\mathbb{Q}$.

Theorem 6.19.[2] *Suppose $f \in \mathbb{Q}[x]$ has degree n and t nonzero terms, and is a perfect rth power. Algorithm 6.6 computes an rth root of f using $\tilde{O}\left(t(t+r)^4 \cdot \log n \cdot \log \mathscr{H}(f)\right)$ bit operations.*

Proof. Let $h \in \mathbb{Q}[x]$ such that $h^r = f$, and let $c \in \mathbb{Z}_{>0}$ be minimal such that $ch \in \mathbb{Z}[x]$. Gauß's Lemma tells us that c^r must be the least positive integer such that $c^r f \in \mathbb{Z}[x]$ as well. Then, using Theorem 6.9, we have:

$$\mathscr{H}(h) \leq \|ch\|_\infty \leq \|ch\|_2 \leq (t \left\|c^r f\right\|_\infty)^{1/r} \leq t^{1/r} \mathscr{H}(f)^{(t+1)/r}.$$

(The last inequality comes from the fact that the lcm of the denominators of f is at most $\mathscr{H}(f)^t$.)

Hence $\log \mathscr{H}(h) \in O\left((t \log \mathscr{H}(f))/r\right)$. Clearly the most costly step in the algorithm will still be the computation of h_i^{r+1} at each iteration through Step 7. For simplicity in our analysis, we can just treat h_i (the value of h at the ith iteration of the while loop in our algorithm) as equal to h (the *actual* root of f), since we know $\tau(h_i) \leq \tau(h)$ and $\mathscr{H}(h_i) \leq \mathscr{H}(h)$.

Lemma 6.17 and Conjecture 6.16 tell us that $\tau(h^i) \leq 2(t+r)^2$ for $i = 1,2,\ldots,r$. To compute h^{r+1}, we will actually compute $(ch)^{r+1} \in \mathbb{Z}[x]$ by repeatedly squaring and multiplying by ch, and then divide out c^{r+1}. This requires at most $\lfloor \log_2 r+1 \rfloor$ squares and products.

[2] Subject to the validity of Conjecture 6.16.

Note that $\left\|(ch)^{2i}\right\|_\infty \le (t+r)^2 \left\|(ch)^i\right\|_\infty^2$ and $\left\|(ch)^{i+1}\right\|_\infty \le (t+r)^2 \left\|(ch)^i\right\|_\infty \|ch\|_\infty$. Therefore

$$\left\|(ch)^i\right\|_\infty \le (t+r)^{2r} \|ch\|_\infty^r, \quad i = 1,2,\ldots,r,$$

and thus $\log\left\|(ch)^i\right\|_\infty \in O\left(r(t+r) + t\log\mathscr{H}(f)\right)$, for each intermediate power $(ch)^i$.

Thus each of the $O\left((t+r)^4 \log r\right)$ field operations at each iteration costs $O(\mathsf{M}(t\log\mathscr{H}(f) + \log r(t+r)))$ bit operations, which then gives the stated result. □

Again, observe that the bit complexity in general is an upper bound on the number of IMM operations, and when we use the blunt measure of $\tilde{O}(\cdots)$ notation, the costs are exactly the same.

The method used for Step 10 depends on the field F. For $\mathsf{F} = \mathbb{Q}$, we just need to find two integer perfect roots, which can be done in "nearly linear" time by the algorithm of (Bernstein, 1998). Otherwise, we can use any of the well-known fast root-finding methods over $\mathsf{F}[x]$ to compute a root of $x^r - f_u$.

6.3.3 Computing multivariate roots

For the problem of computing perfect polynomial roots of multivariate polynomials, we again reduce the problem to a univariate one, this time employing the well-known Kronecker substitution method.

Suppose $f, h \in \mathsf{F}[x_1,\ldots,x_\ell]$ and $r \in \mathbb{N}$ such that $f = h^r$. It is easily seen that each partial degree of f is exactly r times the corresponding partial degree in h, that is, $\deg_{x_i} f = r\deg_{x_i} h$, for all $r \in \{1,\ldots,\ell\}$.

Now suppose f and r are given and we wish to compute h. First use the relations above to compute $d_i = \deg_{x_i} h + 1$ for each $i \in \{1,\ldots,\ell\}$. (If any $\deg_{x_i} f_i$ is not a multiple of r, then f must not be an rth power.)

Now use the Kronecker substitution and define

$$\hat{f} = f\left(y, y^{d_1}, y^{d_1 d_2}, \ldots, y^{d_1\cdots d_{\ell-1}}\right) \quad \text{and} \quad \hat{h} = h\left(y, y^{d_1}, y^{d_1 d_2}, \ldots, y^{d_1\cdots d_{\ell-1}}\right),$$

where y is a new variable. Clearly $\hat{f} = \hat{h}^r$, and since each $d_i > \deg_{x_i} h$, h is easily recovered from the sparse representation of $\hat{h}$ in the standard way: For each non-zero term cy^e in $\hat{h}$, compute the digits of e in the mixed radix representation corresponding to the sequence $d_1, d_2, \ldots, d_{\ell-1}$. That is, decompose e (uniquely) as $e = e_1 + e_2 d_1 + e_3 d_1 d_2 + \cdots + e_\ell d_1 \cdots d_{\ell-1}$ with each $e_i \in \mathbb{N}$ such that $e_i < d_i$. Then the corresponding term in h is $c x_1^{e_1} \cdots x_\ell^{e_\ell}$.

Therefore we simply use either algorithm above to compute $\hat{h}$ as the rth root of $\hat{f}$ over $\mathsf{F}[y]$, then invert the Kronecker map to obtain $h \in \mathsf{F}[x_1,\ldots,x_\ell]$. The conversion steps are clearly polynomial-time, and notice that $\log\deg\hat{f}$ is at most ℓ times larger than $\log\deg f$. Therefore the lacunary sizes of $\hat{f}$ and $\hat{h}$ are polynomial in the lacunary sizes of f and h, and the algorithms in this section yield polynomial-time algorithms to compute perfect rth roots of multivariate lacunary polynomials.

6.4 Implementation

To investigate the practicality of our algorithms, we implemented Algorithm 6.3 using NTL (Shoup, 2009), for dense univariate polynomials with arbitrary-precision integers as their coefficients. The only significant difference between our implementation and the algorithm specified above is our choice of the ground field. Rather than working in a degree-$(r-1)$ extension of $\mathbb{F}_p$, we simply find a random p in the same range such that $(r-1) \mid p$. Proving good complexity bounds is more difficult in this situation, as it requires bounds on primes in arithmetic progressions. That issue in particular will be discussed much more thoroughly in Chapter 8, but for now we simply point out that this is extremely efficient in practice, as it avoids working in extension fields.

To our knowledge, all existing algorithms which could be used to test for perfect powers actually compute the root h. We implemented the fastest method for densely-represented polynomials, both in theory and in practice, which is a standard Newton iteration. This algorithm uses dense polynomial arithmetic in its implementation, and also requires the power r to be given explicitly. This Newton iteration method is adapted to the case when r is not known by simply trying all possible powers r, and then checking with a single evaluation whether the computed candidate h is actually an rth root of f. This technique is not provably correct, but in all of our trials it has never failed. Furthermore, since this is the algorithm we are comparing *against*, we graciously give it every possible advantage.

The results of these experiments were reported in (Giesbrecht and Roche, 2008). We found that, when the given polynomial f was *not* a perfect power, our algorithm detected this extremely quickly, in fact always with just one random evaluation. Even when the inputs had mostly nonzero coefficients, we found that our algorithm performed competitively with the dense Newton iteration approach. This is due to the black-box nature of the algorithm — it requires only a way to quickly evaluate a polynomial at a given point, which is of course possible whether the polynomial is sparsely or densely represented. Much more on the black box representation and what other kinds of information can be learned from it will be discussed in the next two chapters.

6.5 Conclusions

Given a sparse polynomial over the integers, rational numbers, or a finite field, we have shown polynomial-time Monte Carlo algorithms to determine whether the polynomial is a perfect rth power for any $r \geq 2$. In every case, the error is one-sided, i.e., the algorithm may produce false positives but never true negatives. Therefore we have shown that these problems are in the complexity class **coRP**.

Actually computing h such that $f = h^r$ is a somewhat trickier problem, at least insofar as bounds on the sparsity of h have not been completely resolved. We presented an output-sensitive algorithm that makes use of factorization over algebraic extension fields, and also a conjecturally-fast sparse Newton iteration.

There are many interesting potential connections from our algorithms to other related problems. First, polynomial perfect powers are a special case of the functional decomposition problem. Given a polynomial $f \in \mathsf{R}[x]$, the univariate version of this problem asks for a pair of polynomials $g, h \in \mathsf{R}[x]$ such that $f(x) = g(h(x))$. This connection is made more explicit in our discussion of Algorithm 6.6 in the next chapter. For now, it suffices to say that it is not known whether decomposition can be solved in polynomial time for sparse polynomials.

Perfect powers are also a special case of factorization. As mentioned in the beginning of this chapter, existing algorithms for factorization of sparse polynomials, for example those by Lenstra (1999); Kaltofen and Koiran (2006), require polynomial-time in the degree of the computed factors. Because the degree of a sparse polynomial can be exponential in the sparse representation size, this precludes the computation of high-degree factors which have few nonzero terms. The combination of Algorithms 6.3 and 6.5 gives the first polynomial-time algorithm to compute *any* sparse, high-degree factors of a sparse polynomial. A rich topic of further investigation would be the development of further algorithms of this type, with the ultimate goal being an either an algorithm to compute all t-sparse factors in $t^{O(1)}$ time, or a reduction showing the general problem to be intractable.

> I do not pretend to start with precise questions. I do not think you can start with anything precise. You have to achieve such precision as you can, as you go along.
>
> —Bertrand Russell, *The Philosophy of Logical Atomism* (1918)

Chapter 7

Sparse interpolation

The next two chapters examine methods to determine the sparse representation of an unknown polynomial, given only a way to evaluate the polynomial at chosen points. These methods are generally termed black-box sparse interpolation.

First, we examine a new approach to the standard sparse interpolation problem over large finite fields and approximate complex numbers. This builds on recent work by Garg and Schost (2009), improving the complexity of the fastest existing algorithm over large finite fields, and the numerical stability of the state of the art for approximate sparse interpolation. The primary new tool that we introduce is a randomised method to distinguish terms in the unknown polynomial based on their coefficients, which we term diversification. Using this technique, our new algorithms gain practical and theoretical efficiency over previous methods by avoiding the need to use factorization algorithms as a subroutine.

We gratefully acknowledge the helpful and useful comments by Éric Schost as well as the Symbolic Computation Group members at the University of Waterloo on preliminary versions of the work in this chapter.

7.1 Background

Polynomial interpolation is a long-studied and important problem in computer algebra and symbolic computation. Given a way to evaluate an unknown polynomial at any chosen point, and an upper bound on the degree, the interpolation problem is to determine a representation for the polynomial. In *sparse* interpolation, we are also given an upper bound on the number of nonzero terms in the unknown polynomial, and the output is returned in the sparse representation. Generally speaking, we seek algorithms whose cost is polynomially-bounded by the size of the output, i.e., the sparse representation size of the unknown polynomial.

Sparse interpolation has numerous applications in computer algebra and engineering. Numerous mathematical computations suffer from the problem of *intermediate expression swell*, whereby the size of polynomials encountered in the middle of a computation is much larger than the size of the input or the output. In such cases where the output is known to be sparse, sparse interpolation can eliminate the need to store large intermediate expressions explicitly, thus greatly reducing not only the intermediate storage requirement but also the computational cost. Important examples of such problems are multivariate polynomial factorization and system solving (Canny, Kaltofen, and Yagati, 1989; Kaltofen and Trager, 1990; Díaz and Kaltofen, 1995, 1998; Javadi and Monagan, 2007, 2009).

Solving non-linear systems of multivariate polynomials with approximate coefficients has numerous practical applications, especially in engineering (see for example Sommese and Wampler, 2005). Homotopy methods are a popular way of solving such systems and related problems by following the paths of single solution points from an initial, easy system, to the target system of interest. Once enough points are known in the target system, sparse interpolation methods are used to recover a polynomial expression for the actual solution. These techniques generally fall under the category of hybrid symbolic-numeric computing, and in particular have been applied to solving non-linear systems (e.g., Sommese, Verschelde, and Wampler, 2001, 2004; Stetter, 2004) and factoring approximate multivariate polynomials (e.g., Kaltofen, May, Yang, and Zhi, 2008).

Sparse interpolation is also of interest in theoretical computer science. It is a non-trivial generalisation of the important problem of *polynomial identity testing*, or PIT for short. Generally speaking, the PIT problem is to identify whether the polynomial represented by a given algebraic circuit is identically zero. Surprisingly, although this question is easily answered using randomisation and the classical results of Demillo and Lipton (1978); Zippel (1979); Schwartz (1980), no *deterministic* polynomial-time algorithm is known. In fact, even derandomising the problem for circuits of depth four would have important consequences (Kabanets and Impagliazzo, 2004). We will not discuss this problem further, but the point the reader to the excellent recent surveys of Saxena (2009) and Shpilka and Yehudayoff (2010).

7.1.1 Problem definition

Let $f \in \mathsf{F}[x_1, \ldots, x_n]$ have degree less than d. A *black box* for f is a function which takes as input a vector $(a_1, \ldots, a_n) \in \mathsf{F}^n$ and produces $f(a_1, \ldots, a_n) \in \mathsf{F}$. The cost of the black box is the number of operations in F required to evaluate it at a given input.

Clausen, Dress, Grabmeier, and Karpinski (1991) showed that, if only evaluations over the ground field F are allowed, then for some instances at least $\Omega(n^{\log t})$ black box probes are required. Hence if we seek polynomial-time algorithms, we must extend the capabilities of the black box. To this end, Díaz and Kaltofen (1998) introduced the idea of an *extended domain black box* which is capable of evaluating $f(b_1, \ldots, b_n) \in \mathsf{E}$ for any $(b_1, \ldots, b_n) \in \mathsf{E}^n$ where E is any extension field of F. That is, we can change every operation in the black box to work over an extension field, usually paying an extra cost per evaluation proportional to the size of the extension.

Motivated by the case of black boxes that are division-free algebraic circuits, we will use the following model which we believe to be fair and cover all previous relevant results. Again we use the notation of $\mathsf{M}(m)$ for the number of field operations required to multiply two dense univariate polynomials with degrees less than m, and $\tilde{O}(m)$ to represent any function bounded by $m(\log m)^{O(1)}$.

Definition 7.1. *Let* $f \in \mathsf{F}[x_1,\ldots,x_n]$ *and* $\ell > 0$. *A* remainder black box *for* f *with size* ℓ *is a procedure which, given any monic square-free polynomial* $g \in \mathsf{F}[y]$ *with* $\deg g = m$, *and any* $h_1,\ldots,h_n \in \mathsf{F}[y]$ *with each* $\deg h_i < m$, *produces* $f(h_1,\ldots,h_n) \operatorname{rem} g$ *using at most* $\ell \cdot \mathsf{M}(m)$ *operations in* F.

This definition is general enough to cover the algorithms we know of over finite fields, and we submit that the cost model is fair to the standard black box, extended domain black box, and algebraic circuit settings. The model also makes sense over complex numbers, as we will see.

7.1.2 Interpolation over finite fields

We first summarize previously known univariate interpolation algorithms when F is a finite field with q elements and identify our new contributions here. For now, let $f \in \mathbb{F}_q[x]$ have degree less than d and sparsity t. We will assume we have a remainder black box for f with size ℓ. Since field elements can be represented with $O(\log q)$ bits, a polynomial-time algorithm will have cost polynomial in ℓ, t, $\log d$, and $\log q$.

For the dense output representation, one can use the classical method of Newton, Waring, and Lagrange to interpolate in $\tilde{O}(\ell d)$ time (von zur Gathen and Gerhard, 2003, §10.2).

The algorithm of Ben-Or and Tiwari (1988) for sparse polynomial interpolation, and in particular the version developed by Kaltofen and Yagati (1989), can be adapted to arbitrary finite fields. Unfortunately, these algorithms require t discrete logarithm computations in $\mathbb{F}_q^*$, whose cost is small if the field size q is chosen carefully (as in Kaltofen (2010b)), but not in general. For arbitrary (and potentially large) q, we can take advantage of the fact that each discrete logarithm that needs to be computed falls in the range $[0,1,\ldots,d-1]$. The "kangaroo method" of Pollard (1978, 2000) can, with high probability, compute such a discrete log with $O(\sqrt{d})$ field operations. Using this algorithm makes brings the total worst-case cost of Ben-Or and Tiwari's algorithm to $O(t\ell + t^2 + t\sqrt{d})$.

This chapter builds most directly on the work of (Garg and Schost, 2009), who gave the first polynomial-time algorithm for sparse interpolation over an arbitrary finite field. Their algorithm works roughly as follows. For very small primes p, use the black box to compute f modulo $x^p - 1$. A prime p is a "good prime" if and only if all the terms of f are still distinct modulo $x^p - 1$. If we do this for all p in the range of roughly $O(t^2 \log d)$, then there will be sufficient good primes to recover the unique symmetric polynomial over $\mathbb{Z}[y]$ whose roots are the exponents of nonzero terms in f. We then factor this polynomial to find those exponents, and correlate with any good prime image to determine the coefficients. The total cost is $\tilde{O}(\ell t^4 \log^2 d)$ field operations. Using randomization, it is easy to reduce this to $\tilde{O}(\ell t^3 \log^2 d)$.

	Probes	Probe degree	Computation cost	Total cost
Dense	d	1	$\widetilde{O}(d)$	$\widetilde{O}(\ell d)$
Ben-Or & Tiwari	$O(t)$	1	$O(t^2+t\sqrt{d})$	$\widetilde{O}(\ell t+t^2+t\sqrt{d})$
Garg & Schost	$\widetilde{O}(t^2 \log d)$	$\widetilde{O}(t^2 \log d)$	$\widetilde{O}(t^4 \log^2 d)$	$\widetilde{O}(\ell t^4 \log^2 d)$
Randomized G & S	$\widetilde{O}(t \log d)$	$\widetilde{O}(t^2 \log d)$	$\widetilde{O}(t^3 \log^2 d)$	$\widetilde{O}(\ell t^3 \log^2 d)$
Ours	$O(\log d)$	$\widetilde{O}(t^2 \log d)$	$\widetilde{O}(t^2 \log^2 d)$	$\widetilde{O}(\ell t^2 \log^2 d)$

Table 7.1: Sparse univariate interpolation over large finite fields,
with black box size ℓ, degree d, and t nonzero terms

Observe that the coefficients of the symmetric integer polynomial in Garg & Schost's algorithm are bounded by $O(d^t)$, which is much larger than the $O(d)$ size of the exponents ultimately recovered. Our primary contribution over finite fields of size at least $\Omega(t^2 d)$ is a new algorithm which avoids evaluating the symmetric polynomial and performing root finding over $\mathbb{Z}[y]$. As a result, we reduce the total number of required evaluations and develop a randomized algorithm with cost $\widetilde{O}(\ell t^2 \log^2 d)$, which is roughly quadratic in the input and output sizes. Since this can be deterministically verified in the same time, our algorithm (as well as the randomized version of Garg & Schost) is of the Las Vegas type.

The relevant previous results mentioned above are summarized in Table 7.1, where we assume in all cases that the field size q is "large enough". In the table, the "probe degree" refers to the degree of g in each evaluation of the remainder black box as defined above.

7.1.3 Multivariate interpolation

Any of the univariate algorithms above can be used to generate a multivariate polynomial interpolation algorithm in at least two different ways. For what follows, write $\rho(d,t)$ for the number of remainder black box evaluations required by some univariate interpolation algorithm, $\Delta(d,t)$ for the degree of the remainder in each evaluation, and $\psi(d,t)$ for the number of other field operations required besides black box calls. Observe that these correspond to the first three columns in Table 7.1.

The first way to adapt a univariate interpolation algorithm to a multivariate one is Kronecker substitution: given a remainder black box for an unknown $f \in \mathsf{F}[x_1,\ldots,x_n]$, with each partial degree less than d, we can easily construct a remainder black box for the univariate polynomial $\hat{f} = f(x, x^d, x^{d^2}, \ldots, x^{d^{n-1}}) \in \mathsf{F}[x]$, whose terms correspond one-to-one with terms of f. This is the approach taken for instance in Kaltofen (2010b, §2) for the interpolation of multivariate polynomials with rational coefficients. The cost is simply the cost of the chosen underlying univariate algorithm, with the degree increased to d^n.

The other method for constructing a multivariate interpolation algorithm is due to Zippel (1990). The technique is inherently probabilistic and works variable-by-variable, at each step solving a number of $\hat{t} \times \hat{t}$ transposed Vandermonde systems, for some $\hat{t} \le t$. Specifically, each system is of the form $Ax = b$, where A is a $\hat{t} \times \hat{t}$ matrix of scalars from the coefficient field $\mathbb{F}_q$. The vector v consists of the output of $\hat{t}$ remainder black box evaluations, and so its

	Kronecker	Zippel
Dense	$\tilde{O}(\ell d^n)$	$\tilde{O}(\ell n t d)$
Ben-Or & Tiwari	$\tilde{O}(\ell t + t^2 + t d^{n/2})$	$\tilde{O}(n t^3 + n t^2 \sqrt{d} + \ell n t^2)$
Garg & Schost	$\tilde{O}(\ell n^2 t^4 \log^2 d)$	$\tilde{O}(\ell n t^5 \log^2 d)$
Randomized G & S	$\tilde{O}(\ell n^2 t^3 \log^2 d)$	$\tilde{O}(\ell n t^4 \log^2 d)$
Ours	$\tilde{O}(\ell n^2 t^2 \log^2 d)$	$\tilde{O}(\ell n t^3 \log^2 d)$

Table 7.2: Sparse multivariate interpolation over large finite fields, with black box size ℓ, n variables, degree d, and t nonzero terms

elements are in $\mathbb{F}_q[y]$, and the system must be solved modulo some $g \in \mathbb{F}_q[y]$, as specified by the underlying univariate algorithm. Observe however that since A does not contain polynomials, computing $x = A^{-1}b$ requires no modular polynomial arithmetic. In fact, using the same techniques as Kaltofen and Yagati (1989, §5), employing fast dense bivariate polynomial arithmetic, each system can be solved using

$$O\Big(\mathsf{M}\big(t \cdot \Delta(d,t)\big) \cdot \log\big(t \cdot \Delta(d,t)\big)\Big)$$

field operations.

Each transposed Vandermonde system gives the remainder black box evaluation of each of $\hat{t}$ univariate polynomials that we are interpolating in that step. The number of such systems that must be solved is therefore $\rho(d,t)$, as determined by the underlying univariate algorithm. Finally, each of the $\hat{t}$ univariate interpolations proceeds with the given evaluations. The total cost, over all iterations, is

$$\tilde{O}\big(\ell n t \cdot \Delta(d,t) \cdot \rho(d,t)\big)$$

field operations for the remainder black box evaluations, plus

$$\tilde{O}\big(n t \psi(d,t) + \ell n t \cdot \Delta(d,t)\big)$$

field operations for additional computation. Zippel (1990) used the dense algorithm for univariate interpolation; using Ben-Or and Tiwari's algorithm instead was studied by Kaltofen and Lee (2003).

Table 7.2 summarizes the cost of the univariate algorithms mentioned above applied to sparse multivariate interpolation over a sufficiently large finite field, using Kronecker's and Zippel's methods.

For completeness, we mention a few more results on closely related problems that do not have a direct bearing on the current study. Grigoriev, Karpinski, and Singer (1990) give a parallel algorithm with small depth but which is not competitive in our model due to the large number of processors required. A practical parallel version of Ben-Or and Tiwari's algorithm has been developed by (Javadi and Monagan, 2010). (Kaltofen, Lakshman, and Wiley, 1990) and (Avendaño, Krick, and Pacetti, 2006) present modular algorithms for interpolating polynomials with rational and integer coefficients. However, their methods do not seem to apply to finite fields.

7.1.4 Approximate Polynomial Interpolation

In Section 7.4 we consider the case of approximate sparse interpolation. Our goal is to provide both a numerically more robust practical algorithm, but also the first algorithm which is provably numerically stable, with no heuristics or conjectures. We define an "ϵ-approximate black box" as one which evaluates an unknown t-sparse target polynomial $f \in \mathbb{C}[x]$, of degree d, with relative error at most $\epsilon > 0$. Our goal is to build a t-sparse polynomial g such that $\left\|f - g\right\| \leq \epsilon \left\|f\right\|$. A bound on the degree and sparsity of the target polynomial, as well as ϵ, must also be provided. In Section 7.4 we formally define the above problem, and demonstrate that the problem of sparse interpolation is well-posed. We then adapt our variant of the (Garg and Schost, 2009) algorithm for the approximate case, prove it is numerically accurate in terms of the relative error of the output, and analyze its cost. We also present a full implementation in Section 7.5 and validating experiments.

Recently, a number of numerically-focussed sparse interpolation algorithms have been presented. The algorithm of (Giesbrecht, Labahn, and Lee, 2009) is a numerical adaptation of (Ben-Or and Tiwari, 1988), which samples f at $O(t)$ randomly chosen roots of unity $\omega \in \mathbb{C}$ on the unit circle. In particular, ω is chosen to have (high) order at least the degree, and a randomization scheme is used to avoid clustering of nodes which will cause dramatic ill-conditioning. A relatively weak theoretical bound is proven there on the randomized conditioning scheme, though experimental and heuristic evidence suggests it is much better in practice. (Cuyt and Lee, 2008) adapt Rutishauser's qd algorithm to alleviate the need for bounds on the partial degrees and the sparsity, but still evaluate at high-order roots of unity. Approximate sparse rational function interpolation is considered by (Kaltofen and Yang, 2007) and (Kaltofen, Yang, and Zhi, 2007), using the Structured Total Least Norm (STLN) method and, in the latter, randomization to improve conditioning. Approximate sparse interpolation is also considered for integer polynomials by (Mansour, 1995), where a polynomial-time algorithm is presented in quite a different model from ours. In particular the evaluation error is absolute (not relative) and the complexity is sensitive to the bit length of the integer coefficients.

Note that all these works evaluate the polynomial only on the unit circle. This is necessary because we allow and expect f to have very large degree, which would cause a catastrophic loss of precision at data points of non-unit magnitude. Similarly, we assume that the complex argument of evaluation points is exactly specified, which is again necessary because any error in the argument would be exponentially magnified by the degree.

The primary contribution of our work in Section 7.4 below is to provide an algorithm with both rigorously provable relative error and good practical performance. Our algorithm typically requires $\tilde{O}(t^2 \log^2 d)$ evaluations at primitive roots of unity of order $\tilde{O}(t^2 \log d)$ (as opposed to order d in previous approaches). We guarantee that it finds a t-sparse polynomial g such that $\left\|g - f\right\| \leq 2\epsilon \left\|f\right\|$. An experimental demonstration of the numerical robustness is given in Section 7.5.

7.2 Sparse interpolation for generic fields

For the remainder, we assume the unknown polynomial f is always univariate. This is without loss of generality, as we can use the Kronecker substitution as discussed previously. The exponential increase in the univariate degree only corresponds to a factor of n increase in $\log \deg f$, and since our algorithms will ultimately have cost polynomial in $\log \deg f$, polynomial time is preserved.

Assume a fixed, unknown, t-sparse univariate polynomial $f \in \mathsf{F}[x]$ with degree at most d. We will use a remainder black box for f to evaluate $f \operatorname{rem}(x^p - 1)$ for small primes p. We say p is a "good prime" if the sparsity of $f \operatorname{rem}(x^p - 1)$ is the same as that of f itself — that is, none of the exponents are equivalent modulo p.

The minimal bit length required so that a randomly chosen prime is with high probability a good prime is shown in the following lemma.

Lemma 7.2. *Let $f \in \mathsf{F}[x]$ be a t-sparse polynomial with degree d, and let*

$$\lambda = \max\left(21, \left\lceil \frac{5}{3} t(t-1) \ln d \right\rceil\right).$$

A prime chosen at random in the range $\lambda, \ldots, 2\lambda$ is a good prime for f with probability at least $1/2$.

Proof. Write $e_1, \ldots, e_t$ for the exponents of nonzero terms in f. If p is a bad prime, then p divides $(e_j - e_i)$ for some $i < j$. Each $e_j - e_i \le d$, so there can be at most $\log_\lambda d = \ln d / \ln \lambda$ primes that divide each $e_j - e_i$. There are exactly $\binom{t}{2}$ such pairs of exponents, so the total number of bad primes is at most $(t(t-1)\ln d)/(2\ln\lambda)$.

From Rosser and Schoenfeld (1962, Corollary 3 to Theorem 2), the total number of primes in the range $\lambda, \ldots, 2\lambda$ is at least $3\lambda/(5\ln\lambda)$ when $\lambda \ge 21$, which is at least $t(t-1)\ln d/\ln\lambda$, at least twice the number of bad primes. □

Now observe an easy case for the sparse interpolation problem. If a polynomial $f \in \mathsf{F}[x]$, has all coefficients distinct; that is, $f = \sum_{1 \le i \le t} c_i x^{e_i}$ and $c_i = c_j \Rightarrow i = j$, then we say f is *diverse*. To interpolate a diverse polynomial $f \in \mathsf{F}[x]$, we first follow the method of Garg and Schost (2009) by computing $f \operatorname{rem}(x^{p_i} - 1)$ for "good primes" p_i such that the sparsity of $f \operatorname{rem}(x^{p_i} - 1)$ is the same as that of f. Since f is diverse, $f \operatorname{rem}(x^{p_i} - 1)$ is also diverse and in fact each modular image has the same set of coefficients. Using this fact, we avoid the need to construct and subsequently factor the symmetric polynomial in the exponents. Instead, we correlate like terms based on the (unique) coefficients in each modular image, then use simple Chinese remaindering to construct each exponent e_i from its image modulo each p_i. This requires only $O(\log d)$ remainder black box evaluations at good primes, gaining a factor of t improvement over the randomized version of Garg and Schost (2009) for diverse polynomials.

In the following sections, we will show how to choose an $\alpha \in \mathsf{F}$ so that $f(\alpha x)$ — which we can easily construct a remainder black box for — is diverse. With such a procedure, Algorithm 7.1 gives a Monte Carlo algorithm for interpolation over a general field.

Algorithm 7.1: Generic interpolation

```
Input: $\mu \in \mathbb{R}_{>0}$, $T, D, q \in \mathbb{N}$, and a remainder black box for unknown $T$-sparse $f \in \mathsf{F}[x]$
        with $\deg f < D$
Output: $t \in \mathbb{N}$, $e_1, \ldots, e_t \in \mathbb{N}$, and $c_1, \ldots, c_t \in \mathsf{F}$ such that $f = \sum_{1 \le i \le t} c_i x^{e_i}$
$t \leftarrow 0$
$\lambda \leftarrow \max\left(21, \left\lceil \frac{5}{3} T(T-1) \ln D \right\rceil\right)$
for $\lceil \log_2(3/\mu) \rceil$ primes $p \in \{\lambda, \ldots, 2\lambda\}$ do
    Use black box to compute $f_p = f(x) \operatorname{rem} (x^p - 1)$
    if $f_p$ has more than $t$ terms then
        $t \leftarrow$ sparsity of $f_p$
        $\varrho \leftarrow p$
$\alpha \leftarrow$ element of $\mathsf{F}$ such that $\Pr[f(\alpha x)$ is not diverse$] < \mu/3$
$g_\varrho \leftarrow f(\alpha x) \operatorname{rem} (x^\varrho - 1)$
$c_1, \ldots, c_t \leftarrow$ nonzero coefficients of $g_\varrho$
$e_1, \ldots, e_t \leftarrow 0$
for $\lceil 2 \ln(3/\mu) + 4 (\ln D)/(\ln \lambda) \rceil$ primes $p \in \{\lambda, \ldots, 2\lambda\}$ do
    Use black box to compute $g_p = f(\alpha x) \operatorname{rem} (x^p - 1)$
    if $g_p$ has exactly $t$ nonzero terms then
        for $i = 1, \ldots, t$ do Update $e_i$ with exponent of $c_i$ in $g_p$ modulo $p$ via Chinese
        remaindering
for $i = 1, \ldots, t$ do $c_i \leftarrow c_i \alpha^{-e_i}$
return $f(x) = \sum_{1 \le i \le t} c_i x^{e_i}$
```

Theorem 7.3. *With inputs as specified, Algorithm 7.1 correctly computes the unknown polynomial f with probability at least $1-\mu$. The total cost in field operations (except for step 8) is*

$$O\left(\ell\cdot\left(\frac{\log D}{\log T+\log\log D}+\log\frac{1}{\mu}\right)\cdot \mathsf{M}\left(T^2\log D\right)\right).$$

Proof. The for loop on line 3 searches for the true sparsity t and a single good prime ϱ. Since each prime p in the given range is good with probability at least 1/2 by Lemma 7.2, the probability of failure at this stage is at most $\mu/3$.

The for loop on line 12 searches for and uses sufficiently many good primes to recover the exponents of f. The product of all the good primes must be at least D, and since each prime is at least λ, at least $(\ln D)/(\ln\lambda)$ good primes are required.

Let $n=\lceil 2\ln(3/\mu)+4(\ln D)/(\ln\lambda)\rceil$ be the number of primes sampled in this loop, and $k=\lceil(\ln D)/(\ln\lambda)\rceil$ the number of good primes required. We can derive that $(n/2-k)^2\geq(\ln(3/\mu)+k)^2>(n/2)\ln(3/\mu)$, and therefore $\exp(-2(\frac{n}{2}-k)^2/n)<\mu/3$. Using Hoeffding's Inequality (Hoeffding, 1963), this means the probability of encountering fewer than k good primes is less than $\mu/3$.

Therefore the total probability of failure is at most μ. For the cost analysis, the dominating cost will be the modular black box evaluations in the last for loop. The number of evaluations in this loop is $O(\log(1/\mu)+(\log D)/(\log\lambda))$, and each evaluation has cost $O(\ell\cdot\mathsf{M}(\lambda))$. Since the size of each prime is $\Theta((\log D)/(\log T+\log\log D))$, the complexity bound is correct as stated. □

In case the bound T on the number of nonzero terms is very bad, we could choose a smaller value of λ based on the true sparsity t before line 8, improving the cost of the remainder of the algorithm.

In addition, as our bound on possible number of "bad primes" seems to be quite loose. A more efficient approach in practice would be to replace the for loop on line 12 with one that starts with a prime much smaller than λ and incrementally searches for larger primes, until the product of all good primes is at least D. We could choose the lower bound to start searching from based on lower bounds on the birthday problem. That is, assuming (falsely) that the exponents are randomly distributed modulo p, start with the least p that will have no exponents collide modulo p with high probability. This would yield an algorithm more sensitive to the true bound on bad primes, but unfortunately gives a worse formal cost analysis.

7.3 Sparse interpolation over finite fields

We now examine the case that the ground field F is the finite field with q elements, which we denote $\mathbb{F}_q$. First we show how to effectively diversify the unknown polynomial f in order to complete Algorithm 7.1 for the case of large finite fields. Then we show how to extend this to a Las Vegas algorithm with the same complexity.

7.3.1 Diversification

For an unknown $f \in \mathbb{F}_q[x]$ given by a remainder black box, we must find an α so that $f(\alpha x)$ is diverse. A surprisingly simple trick works: evaluating $f(\alpha x)$ for a randomly chosen nonzero $\alpha \in \mathbb{F}_q$.

Theorem 7.4. *For $q \geq T(T-1)D+1$ and any T-sparse polynomial $f \in \mathbb{F}_q[x]$ with $\deg f < D$, if α is chosen uniformly at random from $\mathbb{F}_q^*$, the probability that $f(\alpha x)$ is diverse is at least $1/2$.*

Proof. Let $t \leq T$ be the exact number of nonzero terms in f, and write $f = \sum_{1 \leq i \leq t} c_i x^{e_i}$, with nonzero coefficients $c_i \in \mathbb{F}_q^*$ and $e_1 < e_2 < \cdots < e_t$. So the ith coefficient of $f(\alpha x)$ is $c_i \alpha^{e_i}$.

If $f(\alpha x)$ is *not* diverse, then we must have $c_i \alpha^{e_i} = c_j \alpha^{e_j}$ for some $i \neq j$. Therefore consider the polynomial $A \in \mathbb{F}_q[y]$ defined by

$$A = \prod_{1 \leq i < j \leq t} \left(c_i y^{e_i} - c_j y^{e_j} \right).$$

We see that $f(\alpha x)$ is diverse if and only if $A(\alpha) \neq 0$, hence the number of roots of A over $\mathbb{F}_q$ is exactly the number of unlucky choices for α.

The polynomial A is the product of exactly $\binom{t}{2}$ binomials, each of which has degree less than D. Therefore

$$\deg A < \frac{T(T-1)D}{2},$$

and this also gives an upper bound on the number of roots of A. Hence $q-1 \geq 2 \deg A$, and at least half of the elements of $\mathbb{F}_q^*$ are not roots of A, yielding the stated result. □

Using this result, given a black box for f and the exact sparsity t of f, we can find an $\alpha \in \mathbb{F}_q$ such that $f(\alpha x)$ is diverse by sampling random values $\alpha \in \mathbb{F}_q$, evaluating $f(\alpha x) \operatorname{rem} x^p - 1$ for a single good prime p, and checking whether the polynomial is diverse. With probability at least $1-\mu$, this will succeed in finding a diversifying α after at most $\lceil \log_2(1/\mu) \rceil$ iterations. Therefore we can use this approach in Algorithm 7.1 with no effect on the asymptotic complexity.

7.3.2 Verification

So far, Algorithm 7.1 over a finite field is probabilistic of the Monte Carlo type; that is, it may give the wrong answer with some controllably-small probability. To provide a more robust Las Vegas probabilistic algorithm, we require only a fast way to check that a candidate answer is in fact correct. To do this, observe that given a modular black box for an unknown T-sparse $f \in \mathbb{F}_q[x]$ and an explicit T-sparse polynomial $g \in \mathbb{F}_q[x]$, we can construct a modular black box for the $2T$-sparse polynomial $f - g$ of their difference. Verifying that $f = g$ thus reduces to the well-studied problem of deterministic polynomial identity testing.

The following algorithm is due to (Bläser, Hardt, Lipton, and Vishnoi, 2009) and provides this check in essentially the same time as the interpolation algorithm; we restate it in Algorithm 7.2 for completeness and to use our notation.

Algorithm 7.2: Verification over finite fields

```
Input: $T, D, q \in \mathbb{N}$ and remainder black box for unknown $T$-sparse $f \in \mathbb{F}_q[x]$ with
       $\deg f \le D$
Output: ZERO iff $f$ is identically zero
for the least $(T-1)\log_2 D$ primes $p$ do
    Use black box to compute $f_p = f \operatorname{rem}(x^p - 1)$
    if $f_p \neq 0$ then return NONZERO
return ZERO
```

Theorem 7.5. *Algorithm 7.2 works correctly as stated and uses at most*

$$O\left(\ell T \log D \cdot \mathsf{M}\left(T \log D \cdot (\log T + \operatorname{loglog} D)\right)\right)$$

field operations.

Proof. For correctness, notice that the requirements for a "good prime" for identity testing are much weaker than for interpolation. Here, we only require that a single nonzero term not collide with any other nonzero term. That is, every bad prime p will divide $e_j - e_1$ for some $2 \le j \le T$. There can be $\log_2 D$ distinct prime divisors of each $e_j - e_1$, and there are $T-1$ such differences. Therefore testing that the polynomial is zero modulo $x^p - 1$ for the first $(T-1)\log_2 D$ primes is sufficient to guarantee at least one nonzero evaluation of a nonzero T-sparse polynomial.

For the cost analysis, the prime number theorem (Bach and Shallit, 1996, Theorem 8.8.4), tells us that the first $(T-1)\log_2 D$ primes are each bounded by $O(T \cdot \log D \cdot (\log T + \operatorname{loglog} D))$. The stated bound follows directly. □

This provides all that we need to prove the main result of this section:

Theorem 7.6. *Given $q \ge T(T-1)D + 1$, any $T, D \in \mathbb{N}$, and a modular black box for unknown T-sparse $f \in \mathbb{F}_q[x]$ with $\deg f \le D$, there is an algorithm that always produces the correct polynomial f and with high probability uses only $\widetilde{O}\left(\ell T^2 \log^2 D\right)$ field operations.*

Proof. Use Algorithms 7.1 and 7.2 with $\mu = 1/2$, looping as necessary until the verification step succeeds. With high probability, only a constant number of iterations will be necessary, and so the cost is as stated. □

For the small field case, when $q \in O(T^2 D)$, the obvious approach would be to work in an extension E of size $O(\log T + \log D)$ over $\mathbb{F}_q$. Unfortunately, this would presumably increase the cost of each evaluation by a factor of $\log D$, potentially dominating our factor of T savings compared to the randomized version of (Garg and Schost, 2009) when the unknown polynomial has very few terms and extremely high degree.

In practice, it seems that a much smaller extension than this is sufficient in any case to make each $\gcd(e_j - e_i, q - 1)$ small compared to $q - 1$, but we do not yet know how to prove any tighter bound in the worst case.

7.4 Approximate sparse interpolation algorithms

In this section we consider the problem of interpolating an approximate sparse polynomial $f \in \mathbb{C}[x]$ from evaluations on the unit circle. We will generally assume that f is t-sparse:

$$f = \sum_{1 \le i \le t} c_i x^{e_i} \text{ for } c_i \in \mathbb{C} \text{ and } e_1 < \cdots < e_t = d. \tag{7.1}$$

We require a notion of size for such polynomials, and define the coefficient 2-norm of $f = \sum_{0 \le i \le d} f_i x^i$ as

$$\|f\| = \sqrt{\sum_{0 \le i \le d} |f_i|^2}.$$

The following identity relates the norm of evaluations on the unit circle and the norm of the coefficients. As in Section 7.2, for $f \in \mathbb{C}[x]$ is as in (7.1), we say that a prime p is a *good prime* for f if $p \nmid (e_i - e_j)$ for all $i \neq j$.

Lemma 7.7. *Let $f \in \mathbb{C}[x]$, p a good prime for f, and $\omega \in \mathbb{C}$ a pth primitive root of unity. Then*

$$\|f\|^2 = \frac{1}{p} \sum_{0 \le i < p} \left|f(\omega^i)\right|^2.$$

Proof. See Theorem 6.9 in the previous chapter. □

We can now formally define the approximate sparse univariate interpolation problem.

Definition 7.8. *Let $\epsilon > 0$ and assume there exists an unknown t-sparse $f \in \mathbb{C}[x]$ of degree at most D. An ϵ-*approximate black box *for f takes an input $\xi \in \mathbb{C}$ and produces a $\gamma \in \mathbb{C}$ such that $|\gamma - f(\xi)| \le \epsilon |f(\xi)|$.*

That is, the relative error of any evaluation is at most ϵ. As noted in the introduction, we will specify our input points exactly, at (relatively low order) roots of unity. The *approximate sparse univariate interpolation problem* is then as follows: given $D, T \in \mathbb{N}$ and $\delta \ge \epsilon > 0$, and an ϵ-approximate black box for an unknown T-sparse polynomial $f \in \mathbb{C}[x]$ of degree at most D, find a T-sparse polynomial $g \in \mathbb{C}[x]$ such that $\|f - g\| \le \delta \|g\|$.

The following theorem shows that t-sparse polynomials are well-defined by good evaluations on the unit circle.

Theorem 7.9. *Let $\epsilon > 0$ and $f \in \mathbb{C}[x]$ be a t-sparse polynomial. Suppose there exists a t-sparse polynomial $g \in \mathbb{C}[x]$ such that for a prime p which is good for both f and $f - g$, and pth primitive root of unity $\omega \in \mathbb{C}$, we have*

$$|f(\omega^i) - g(\omega^i)| \le \epsilon |f(\omega^i)| \quad \text{for } 0 \le i < p.$$

Then $\|f - g\| \le \epsilon \|f\|$. Moreover, if $g_0 \in \mathbb{C}[x]$ is formed from g by deleting all the terms not in the support of f, then $\|f - g_0\| \le 2\epsilon \|f\|$.

Proof. Summing over powers of ω we have

$$\sum_{0\le i<p} |f(\omega^i) - g(\omega^i)|^2 \le \epsilon^2 \sum_{0\le i<p} |f(\omega^i)|^2.$$

Thus, since p is a good prime for both $f-g$ and f, using Lemma 7.7, $p\cdot\left\|f-g\right\|^2 \le \epsilon^2\cdot p\cdot\left\|f\right\|^2$ and $\left\|f-g\right\| \le \epsilon\left\|f\right\|$.

Since $g-g_0$ has no support in common with f

$$\left\|g-g_0\right\| \le \left\|f-g\right\| \le \epsilon\left\|f\right\|.$$

Thus

$$\begin{aligned}\left\|f-g_0\right\| &= \left\|f-g+(g-g_0)\right\| \\ &\le \left\|f-g\right\| + \left\|g-g_0\right\| \le 2\epsilon\left\|f\right\|. \quad \square\end{aligned}$$

□

In other words, any t-sparse polynomial whose values are very close to f must have the same support except possibly for some terms with very small coefficients.

7.4.1 Computing the norm of an approximate sparse polynomial

As a warm-up exercise, consider the problem of computing an approximation for the 2-norm of an unknown polynomial given by a black box. Of course we this is an easier problem than actually interpolating the polynomial, but the technique bears some similarity. In practice, we might also use an approximation for the norm to normalize the black-box polynomial, simplifying certain computations and bounds calculations.

Let $0<\epsilon<1/2$ and assume we are given an ϵ-approximate black box for some t-sparse polynomial $f\in\mathbb{C}[x]$. We first consider the problem of computing $\left\|f\right\|$.

Algorithm 7.3: Approximate norm

```
Input: $T, D \in \mathbb{N}$ and $\epsilon$-approximate black box for unknown $T$-sparse $f \in \mathbb{C}[x]$ with
       $\deg f \le D$
Output: $\sigma \in \mathbb{R}$, an approximation to $\left\|f\right\|$
$\lambda \leftarrow \max\left(21, \left\lceil \frac{5}{3} t(t-1)\ln d \right\rceil\right)$
Choose a prime $p$ randomly from $\{\lambda, \ldots, 2\lambda\}$
$\omega \leftarrow \exp(2\pi i/p)$
$w \leftarrow (f(\omega^0), \ldots, f(\omega^{p-1})) \in \mathbb{C}^p$ computed using the black box
return $(1/\sqrt{p}) \cdot \|w\|$
```

Theorem 7.10. *Algorithm 7.3 works as stated. On any invocation, with probability at least* $1/2$, *it returns a value* $\sigma\in\mathbb{R}_{\ge 0}$ *such that*

$$(1-2\epsilon)\left\|f\right\| < \sigma < (1+\epsilon)\left\|f\right\|.$$

Proof. Let $v = (f(\omega^0), \ldots, f(\omega^{p-1})) \in \mathbb{C}^p$ be the vector of exact evaluations of f. Then by the properties of our ϵ-approximate black box we have $w = v + \epsilon\Delta$, where $|\Delta_i| < |f(\omega^i)|$ for $0 \leq i < p$, and hence $\|\Delta\| < \|v\|$. By the triangle inequality $\|w\| \leq \|v\| + \epsilon\|\Delta\| < (1+\epsilon)\|v\|$. By Lemmas 7.2 and 7.7, $\|v\| = \sqrt{p}\left\|f\right\|$ with probability at least 1/2, so $(1/\sqrt{p}) \cdot \|w\| < (1+\epsilon)\left\|f\right\|$ with this same probability.

To establish a lower bound on the output, note that we can make error in the evaluation relative to the output magnitude: because $\epsilon < 1/2$, $|f(\omega^i) - w_i| < 2\epsilon|w_i|$ for $0 \leq i < p$. We can write $v = w + 2\epsilon\nabla$, where $\|\nabla\| < \|w\|$. Then $\|v\| \leq (1+2\epsilon)\|w\|$, and $(1-2\epsilon)\left\|f\right\| < (1/\sqrt{p}) \cdot \|w\|$. □

7.4.2 Constructing an ϵ-approximate remainder black box

Assume that we have chosen a good prime p for a t-sparse $f \in \mathsf{F}[x]$. Our goal in this subsection is a simple algorithm and numerical analysis to accurately compute $f \operatorname{rem} x^p - 1$.

Assume that $f \operatorname{rem} x^p - 1 = \sum_{0 \leq i < p} b_i x^i$ exactly. For a primitive pth root of unity $\omega \in \mathbb{C}$, let $V(\omega) \in \mathbb{C}^{p \times p}$ be the Vandermonde matrix built from the points $1, \omega, \ldots, \omega^{p-1}$. Recall that $V(\omega) \cdot (b_0, \ldots, b_{p-1})^T = (f(\omega^0), \ldots, f(\omega^{p-1}))^T$ and $V(\omega^{-1}) = p \cdot V(\omega)^{-1}$. Matrix vector product by such Vandermonde matrices is computed very quickly and in a numerically stable manner by the Fast Fourier Transform (FFT).

Algorithm 7.4: Approximate Remainder

```
Input: An $\epsilon$-approximate black box for the unknown $t$-sparse $f \in \mathbb{C}[x]$, and $p \in \mathbb{N}$, a
       good prime for $f$
Output: $h \in \mathbb{C}[x]$ such that $\left\|(f \operatorname{rem} x^p - 1) - h\right\| \leq \epsilon \left\|f\right\|$.
1 $w \leftarrow (f(\omega^0), \ldots, f(\omega^{p-1})) \in \mathbb{C}^p$ computed using the $\epsilon$-approximate black box for $f$
2 $u \leftarrow (1/p) \cdot V(\omega^{-1}) w \in \mathbb{C}^p$ using the FFT algorithm
3 return $h = \sum_{0 \leq i < p} u_i x^i \in \mathbb{C}[x]$
```

Theorem 7.11. *Algorithm 7.4 works as stated, and*

$$\left\|(f \operatorname{rem} x^p - 1) - h\right\| \leq \epsilon \left\|f\right\|.$$

It requires $O(p \log p)$ floating point operations and p evaluations of the black box.

Proof. Because f and $f \operatorname{rem} x^p - 1$ have exactly the same coefficients (p is a good prime for f), they have exactly the same norm. The FFT in Step 2 is accomplished in $O(p \log p)$ floating point operations. This algorithm is numerically stable since $(1/\sqrt{p}) \cdot V(\omega^{-1})$ is unitary. That is, assume $v = (f(\omega_0), \ldots, f(\omega^{p-1})) \in \mathbb{C}^p$ is the vector of *exact* evaluations of f, so $\|v - w\| \leq \epsilon \|v\|$ by the black box specification. Then, using the fact that $\|v\| = \sqrt{p}\left\|f\right\|$,

$$\begin{aligned}\left\|(f \operatorname{rem} x^{p-1}) - h\right\| &= \left\|\frac{1}{p} V(\omega^{-1}) v - \frac{1}{p} V(\omega^{-1}) w\right\| \\ = \frac{1}{\sqrt{p}} \left\|\frac{1}{\sqrt{p}} V(\omega^{-1}) \cdot (v - w)\right\| &= \frac{1}{\sqrt{p}} \|v - w\| \leq \frac{\epsilon}{\sqrt{p}} \|v\| = \epsilon \left\|f\right\|.\end{aligned}$$

□

7.4.3 Creating ϵ-diversity

First, we extend the notion of polynomial diversity to the approximate case.

Definition 7.12. *Let $f \in \mathbb{C}[x]$ be a t-sparse polynomial as in (7.1) and $\delta \geq \epsilon > 0$ such that $|c_i| \geq \delta \left\| f \right\|$ for $1 \leq i \leq t$. The polynomial f is said to be ϵ-*diverse *if and only if every pair of distinct coefficients is at least $\epsilon \left\| f \right\|$ apart. That is, for every $1 \leq i < j \leq t$, $|c_i - c_j| \geq \epsilon \left\| f \right\|$.*

Intuitively, if $(\epsilon/2)$ corresponds to the machine precision, this means that an algorithm can reliably distinguish the coefficients of a ϵ-diverse polynomial. We now show how to choose a random α to guarantee ϵ-diversity.

Theorem 7.13. *Let $\delta \geq \epsilon > 0$ and $f \in \mathbb{C}[x]$ a t-sparse polynomial whose non-zero coefficients are of magnitude at least $\delta \left\| f \right\|$. If s is a prime satisfying $s > 12$ and*

$$t(t-1) \leq s \leq 3.1 \frac{\delta}{\epsilon},$$

then for $\zeta = \mathbf{e}^{2\pi \mathbf{i}/s}$ an s-PRU and $k \in \mathbb{N}$ chosen uniformly at random from $\{0, 1, \ldots, s-1\}$, $f(\zeta^k x)$ is ϵ-diverse with probability at least $\frac{1}{2}$.

Proof. For each $1 \leq i \leq t$, write the coefficient c_i in polar notation to base ζ as $c_i = r_i \zeta^{\theta_i}$, where each r_i and θ_i are nonnegative real numbers and $r_i \geq \delta \left\| f \right\|$.

Suppose $f(\zeta^k x)$ is *not* ϵ-diverse. Then there exist indices $1 \leq i < j \leq t$ such that

$$\left| r_i \zeta^{\theta_i} \zeta^{k e_i} - r_j \zeta^{\theta_j} \zeta^{k e_j} \right| \leq \epsilon \left\| f \right\|.$$

Because $\min(r_i, r_j) \geq \delta \left\| f \right\|$, the value of the left hand side is at least $\delta \left\| f \right\| \cdot \left| \zeta^{\theta_i + k e_i} - \zeta^{\theta_j + k e_j} \right|$. Dividing out $\zeta^{\theta_j + k e_i}$, we get

$$\left| \zeta^{\theta_i - \theta_j} - \zeta^{k(e_j - e_i)} \right| \leq \frac{\epsilon}{\delta}.$$

By way of contradiction, assume there exist distinct choices of k that satisfy the above inequality, say $k_1, k_2 \in \{0, \ldots, s-1\}$. Since $\zeta^{\theta_i - \theta_j}$ and $\zeta^{e_j - e_i}$ are a fixed powers of ζ not depending on the choice of k, this means

$$\left| \zeta^{k_1(e_j - e_i)} - \zeta^{k_2(e_j - e_i)} \right| \leq 2 \frac{\epsilon}{\delta}.$$

Because s is prime, $e_i \neq e_j$, and we assumed $k_1 \neq k_2$, the left hand side is at least $|\zeta - 1|$. Observe that $2\pi/s$, the distance on the unit circle from 1 to ζ, is a good approximation for this Euclidean distance when s is large. In particular, since $s > 12$,

$$\frac{|\zeta - 1|}{2\pi/s} > \frac{\sqrt{2}\left(\sqrt{3} - 1\right)/2}{\pi/6},$$

and therefore $|\zeta - 1| > 6\sqrt{2}(\sqrt{3} - 1)/s > 6.2/s$, which from the statement of the theorem is at least $2\epsilon/\delta$. This is a contradiction, and therefore the assumption was false; namely, there is at most one choice of k such that the i'th and j'th coefficients collide.

Then, since there are exactly $\binom{t}{2}$ distinct pairs of coefficients, and $s \geq t(t-1) = 2\binom{t}{2}$, $f(\zeta^k x)$ is diverse for at least half of the choices for k. □

```
Algorithm 7.5: Adaptive diversification
Input: $\epsilon$-approximate black box for $f$, known good prime $p$, known sparsity $t$
Output: $\zeta, k$ such that $f(\zeta^k x)$ is $\epsilon$-diverse, or FAIL
$s \leftarrow 1$,    $\delta \leftarrow \infty$,    $f_p \leftarrow 0$
while $s \le t^2$ and $\#\{$coeffs $c$ of $f_s$ s.t. $|c| \ge \delta\} < t$ do
    $s \leftarrow$ least prime $\ge 2s$
    $\zeta \leftarrow \exp(2\pi\mathbf{i}/s)$
    $k \leftarrow$ random integer in $\{0, 1, \ldots, s-1\}$
    Compute $f_s = f(\zeta^k x) \operatorname{rem} x^p - 1$
    $\delta \leftarrow$ least number s.t. all coefficients of $f_s$ at least $\delta$ in absolute value are pairwise $\epsilon$-distinct
if $\delta > 2\epsilon$ then return FAIL
else return $\zeta^k$
```

We note that the diversification which maps $f(x)$ to $f(\zeta^k x)$ and back is numerically stable since ζ is on the unit circle.

In practice, the previous theorem will be far too pessimistic. We therefore propose the method of Algorithm 7.5 to adaptively choose s, δ, and ζ^k simultaneously, given a good prime p.

Suppose there exists a threshold $S \in \mathbb{N}$ such that for all primes $s > S$, a random sth primitive root of unity ζ^k makes $f(\zeta^k x)$ ϵ-diverse with high probability. Then Algorithm 7.5 will return a root of unity whose order is within a constant factor of S, with high probability. From the previous theorem, if such an S exists it must be $O(t^2)$, and hence the number of iterations required is $O(\log t)$.

Otherwise, if no such S exists, then we cannot diversify the polynomial. Roughly speaking, this corresponds to the situation that f has too many coefficients with absolute value close to the machine precision. However, the diversification is not actually necessary in the approximate case to guarantee numerical stability. At the cost of more evaluations, as well as the need to factor an integer polynomial, the algorithm of (Garg and Schost, 2009) for finite fields can be adapted quite nicely for the approximate case. This is essentially the approach we outline below, and even if the diversification step is omitted, the stated numerical properties of the computation will still hold true.

7.4.4 Approximate interpolation algorithm

We now plug our ϵ-approximate remainder black box, and method for making f ϵ-diverse, into our generic Algorithm 7.1 to complete our algorithm for approximate interpolation.

Theorem 7.14. *Let $\delta > 0$, $f \in \mathbb{C}[x]$ with degree at most D and sparsity at most T, and suppose all nonzero coefficients c of f satisfy $|c| > \delta \left\|f\right\|$. Suppose also that $\epsilon < 1.5\delta/(T(T-1))$, and we are given an ϵ-approximate black box for f. Then, for any $\mu < 1/2$ we have an algorithm to produce a $g \in \mathbb{C}[x]$ satisfying the conditions of Theorem 7.9. The algorithm succeeds with* *probability at least* $1-\mu$ *and uses* $\widetilde{O}(T^2 \cdot \log(1/\mu) \cdot \log^2 D)$ *black box evaluations and floating point operations.*

Proof. Construct an approximate remainder black box for f using Algorithm 7.4. Then run Algorithm 7.1 using this black box as input. On step 8 of Algorithm 7.1, run Algorithm 7.5, iterating steps 5–7 $\lceil \log_2(3/\mu) \rceil$ times on each iteration through the while loop to choose a diversifying $\alpha = \zeta^k$ with probability at least $1-\mu/3$.

The cost comes from Theorems 7.3 and 7.11 along with the previous discussion and Theorem 7.13. □

Observe that the resulting algorithm is Monte Carlo, but could be made Las Vegas by combining the finite fields zero testing algorithm discussed in Section 7.3.2 with the guarantees of Theorem 7.9.

7.5 Implementation results

We implemented our algorithms in the MVP library, using GMP (Granlund et al., 2010) and NTL (Shoup, 2009) for the exponent arithmetic. For comparison with the algorithm of Garg and Schost (2009), we also used NTL's squarefree polynomial factorization routines. We note that, in our experiments, the cost of integer polynomial factorization (for Garg & Schost) and Chinese remaindering were always negligible.

In our timing results, "G&S Determ" refers to the deterministic algorithm as stated in Garg and Schost (2009) and "Alg 7.1" is the algorithm we have presented here over finite fields, without the verification step. We also developed and implemented a more adaptive, Monte Carlo version of these algorithms, as briefly described at the end of Section 7.2. The basic idea is to sample modulo $x^p - 1$ for just one prime $p \in \Theta(t^2 \log d)$ that is good with high probability, then to search for much smaller good primes. This good prime search starts at a lower bound of order $\Theta(t^2)$ based on the birthday problem, and finds consecutively larger primes until enough primes have been found to recover the symmetric polynomial in the exponents (for Garg & Schost) or just the exponents (for our method). The corresponding improved algorithms are referred to as "G&S MC" and "Alg 7.1++" in the table, respectively.

Table 7.3 summarizes some timings for these four algorithms on our benchmarking machine. Note that the numbers listed reflect the *base-2 logarithm* of the degree bound and the sparsity bound for the randomly-generated test cases. The tests were all performed over the finite field $\mathbb{Z}/65521\mathbb{Z}$. This modulus was chosen for convenience of implementation, although other methods such as the Ben-Or and Tiwari algorithm might be more efficient in this particlar field since discrete logarithms could be computed quickly. However, observe that our algorithms (and those from Garg and Schost) have only poly-logarithmic dependence on the field size, and so will eventually dominate.

The timings are mostly as expected based on our complexity estimates, and also confirm our suspicion that primes of size $O(t^2)$ are sufficient to avoid exponent collisions. It is satisfying but not particularly surprising to see that our "Alg 7.1++" is the fastest on all inputs, as

$\log_2 D$	T	G&S Determ	G&S MC	Alg 7.1	Alg 7.1++
12	10	3.77	0.03	0.03	0.01
16	10	46.82	0.11	0.11	0.08
20	10	—	0.38	0.52	0.33
24	10	—	0.68	0.85	0.38
28	10	—	1.12	2.35	0.53
32	10	—	1.58	2.11	0.66
12	20	37.32	0.15	0.02	0.02
16	20	—	0.91	0.52	0.28
20	20	—	3.5	3.37	1.94
24	20	—	6.59	5.94	2.99
28	20	—	10.91	10.22	3.71
32	20	—	14.83	16.22	4.24
12	30	—	0.31	0.01	0.01
16	30	—	3.66	1.06	0.65
20	30	—	10.95	6.7	3.56
24	30	—	25.04	12.42	9.32
28	30	—	38.86	19.36	13.8
32	30	—	62.53	68.1	14.66
12	40	—	0.58	0.01	0.02
16	40	—	8.98	3.7	1.54
20	40	—	30.1	12.9	8.42
24	40	—	67.97	38.34	16.57
28	40	—	—	73.69	36.24
32	40	—	—	—	40.79

Table 7.3: Finite Fields Algorithm Timings

Noise	Mean Error	Median Error	Max Error
0	4.440 e−16	4.402 e−16	8.003 e−16
$\pm 10^{-12}$	1.113 e−14	1.119 e−14	1.179 e−14
$\pm 10^{-9}$	1.149 e−11	1.191 e−11	1.248 e−11
$\pm 10^{-6}$	1.145 e−8	1.149 e−8	1.281 e−8

Table 7.4: Approximate Algorithm Stability

all the algorithms have a similar basic structure. Had we compared to the Ben-Or and Tiwari or Zippel's method, they would probably be more efficient for small sizes, but would be easily beaten for large degree and arbitrary finite fields as their costs are super-polynomial.

The implementation of the approximate algorithm uses machine `double` precision (based on the IEEE standard), the built-in C++ `complex<double>` type, and the popular Fastest Fourier Transform in the West (FFTW) package for computing FFTs (Frigo and Johnson, 2005). Our stability results are summarized in Table 7.4. Each test case was randomly generated with degree at most 2^{20} and at most 50 nonzero terms. We varied the precision as specified in the table and ran 10 tests in each range. Observe that the error in our results was often *less* than the ϵ error on the evaluations themselves.

7.6 Conclusions

We have presented two new algorithms, using the basic idea of Garg and Schost (2009), plus our new technique of diversification, to gain improvements for sparse interpolation over large finite fields and approximate complex numbers.

There is much room for improvement in both of these methods. For the case of finite fields, the limitation to *large* finite fields of size at least $\Omega(t^2 d^n)$ is in some sense the interesting case, since for very small finite fields it will be faster to use Ben-Or and Tiwari's algorithm and compute the discrete logs as required. However, the restriction on field size for our algorithms does seem rather harsh, and we would like to reduce it, perhaps to a bound of size only $t^{O(1)}$. This would have a tremendous advantage because working in an extension field to guarantee that size would only add a logarithmic factor to the overall complexity. By contrast, working an extension large enough to satisfy the conditions of our algorithm could add a factor of $\tilde{O}(\log d)$ to the complexity, which is significant for lacunary algorithms.

Over approximate complex numbers, our algorithm provides the first provably numerically stable method for sparse interpolation. However, this improved numerical stability is at the expense of extra computational cost and (more significantly) extra black-box probes. For many applications, the cost of probing the unknown function easily dominates the cost of the interpolation algorithm, and therefore reducing the number of probes as much as possible is highly desirable. Ideally, we would like to have an approximate interpolation algorithm with the numerical stability of ours, but using only $O(t)$ probes, as in Ben-Or and Tiwari's algorithm.

Another area for potential improvement is in the bounds used for the number of possible bad primes, in both algorithms. From our experiments and intuition, it seems that using primes of size roughly $O(t^2 + t \log d)$ should be sufficient, rather than the $O(t^2 \log d)$ size our current bounds require. However, proving this seems quite challenging. These and other related bounds will be discussed at greater length in the next chapter.

> The original paraphernalia for the lottery had been lost long ago, and the black box now resting on the stool had been put into use even before Old Man Warner, the oldest man in town, was born. Mr. Summers spoke frequently to the villagers about making a new box, but no one liked to upset even as much tradition as was represented by the black box. There was a story that the present box had been made with some pieces of the box that had preceded it, the one that had been constructed when the first people settled down to make a village here. Every year, after the lottery, Mr. Summers began talking again about a new box, but every year the subject was allowed to fade off without anything's being done. The black box grew shabbier each year: by now it was no longer completely black but splintered badly along one side to show the original wood color, and in some places faded or stained.
>
> —Shirley Jackson, *The Lottery* (1948)

Chapter 8

Sparsest shift interpolation

The interpolation algorithms presented in Chapter 7 produce a representation of the unknown $f \in \mathsf{R}[x]$ as a sparse polynomial in the standard power basis $1, x, x^2, \ldots$. Here we consider instead interpolation in the *shifted power basis* $1, (x-\alpha), (x-\alpha)^2, \ldots$, for some $\alpha \in \mathsf{R}$. This is useful because even polynomials with many nonzero coefficients in the standard basis may have very few terms in the shifted basis, for some very carefully chosen shift α. The algorithm we present works over $\mathbb{Q}[x]$ and produces the *sparsest shift* α, as well as the sparse representation in the α-shifted power basis, in polynomial-time in the size of the output.

The basic idea of our algorithm was first presented at the MACIS 2007 conference (Giesbrecht and Roche, 2007). Significant improvements to those methods were later made and published in the journal Computational Complexity (Giesbrecht and Roche, 2010). We are indebted to Igor Shparlinski for pointing out a few very useful references on analytic number theory that we will discuss, as well as to Erich Kaltofen and Éric Schost for sharing some pre-prints and for other useful discussions on these topics.

8.1 Background

In the last chapter, we saw that for polynomial interpolation algorithms, the choice of representation of the output is absolutely crucial. The sparse interpolation algorithms we reviewed computed the representation of an unknown f in the sparse representation, written as

$$f(x) = b_0 + b_1 x^{d_1} + b_2 x^{d_2} + \cdots + b_s x^{d_s}. \tag{8.1}$$

In this chapter, we will present a new algorithm that instead interpolates into the sparse representation in the α-shifted power basis as in

$$f(x) = c_0 + c_1(x-\alpha)^{e_1} + c_2(x-\alpha)^{e_2} + \cdots + c_t(x-\alpha)^{e_t}. \tag{8.2}$$

With f as in (8.2), we say that α is a *t-sparse shift* of f because the representation has exactly t non-zero and non-constant terms in this basis. When α is chosen so that t is absolutely minimal in (8.2), we call this the *sparsest shift* of f.

8.1.1 Previous results

The most significant challenge here is computing the sparsest shift $\alpha \in \mathbb{Q}$. Computing this value from a set of evaluation points was stated as an open problem by Borodin and Tiwari (1991). Actually, their problem concerned the *decidability* of the problem for an unchosen set of evaluation points. This question is still open; all algorithms that we know of for sparse interpolation require a black box for evaluation so that the interpolated points can be chosen by the algorithm, and we do not depart from this general approach.

Grigoriev and Karpinski (1993) presented the first non-trivial algorithm to compute the sparsest shift of a polynomial. Their algorithm actually computes the sparsest shift from a black box for evaluation, but the complexity of the method is greater than the cost of dense interpolation. Therefore without loss of generality we can assume for their problem that the dense representation of f is known in advance, and they show how to compute the sparsest shift α of f. The authors admit that their algorithm's dependence on the degree is probably not optimal. Our result confirms this by giving a sparsest-shift interpolation whose cost is polynomial in the logarithm of the degree.

A more efficient algorithm to compute the sparsest shift of a polynomial given in the dense representation was developed by Lakshman and Saunders (1996). Although our method is entirely different, we crucially rely on their theorem on the uniqueness and rationality of the sparsest shift (Fact 8.1 below).

Using the early interpolation version of Ben-Or and Tiwari's interpolation algorithm from Kaltofen and Lee (2003), but treating $f(x+\alpha)$ as a polynomial with indeterminate coefficients, Giesbrecht, Kaltofen, and Lee (2003) give even more efficient algorithms for computing the sparsest shift of a given polynomial.

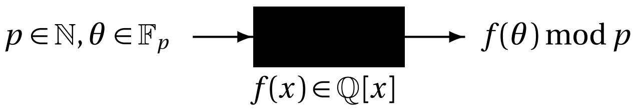
In all these cases, the size of the output in the sparsest-shifted representation could be exponentially smaller than the size of the input, represented in the standard power basis. To see that this is true, one need only polynomials such as $f(x) = (x+1)^n$. The sparsest-shifted representation size of this polynomial is proportional to $\log n$, but the size using the standard power basis — even in the sparse representation — is proportional to n itself.

This motivates our problem of computing the sparsest shift directly from a black box for evaluating f at chosen points, avoiding the need to ever write down f in the standard power basis. Surprisingly, even though we are solving a seemingly more difficult problem, the cost of our new algorithm is competitive with the previous results just mentioned, as we will discuss later.

8.1.2 Problem statement and black box model

We present new algorithms to interpolate $f \in \mathbb{Q}[x]$, given a black box for evaluation, in time proportional to the size of the sparsest-shift representation corresponding to (8.2).

The black box model we use is somewhat different from those of the previous chapter:



Given a prime p and an element θ in $\mathbb{F}_p$, the black box computes the value of the unknown polynomial evaluated at θ over the field $\mathbb{Z}_p$. (An error is produced exactly in those unfortunate circumstances that p divides the denominator of $f(\theta)$.) We generally refer to this as a *modular black box*. To account for the reasonable possibility that the cost of black box calls depends on the size of p, we define κ to be an upper bound on the number of field operations in $\mathbb{F}_p$ used in black box evaluation, for a given polynomial $f \in \mathbb{Q}[x]$. As with the remainder black box, our motivation for this definition really comes from the idea of an algebraic circuit for f. Given such a circuit (over $\mathbb{Q}$), we could easily construct our modular black box, and the parameter κ would correspond to the size of the circuit (which was called ℓ in the previous chapter).

Some kind of extension to the standard black box, such as the modular black box proposed here, is in fact necessary, since the value of a polynomial of degree n at any point other than $0, \pm 1$ will typically have n bits or more. Thus, any algorithm whose complexity is proportional to $\log n$ cannot perform such an evaluation over $\mathbb{Q}$ or $\mathbb{Z}$. Other possibilities might include allowing for evaluations on the unit circle in some representation of a subfield of $\mathbb{C}$, or returning only a limited number of bits of precision for an evaluation. A similar model and approach to ours is employed by Bläser et al. (2009) for deterministic identify testing of polynomials.

To be precise about our notion of size, we extend the definition of size(f) from Chapter 6 to cover shifted-sparse polynomials. Recall that size(q) for $q \in \mathbb{Q}$ is the number of machine words needed to represent q in an IMM. So if we write $q = \frac{a}{b}$ with $a \in \mathbb{Z}$, $b \in \mathbb{N}$, and $\gcd(a,b) =$

1, and if the machine word size is w, then $\text{size}(q) = \lceil \log_2(|a|+1)/w \rceil + \lceil \log_2(b+1)/w \rceil + 1$. For a rational polynomial f as in (8.2), define:

$$\text{size}(f) = \text{size}(\alpha) + \sum_{i=0}^{t} \text{size}(c_i) + \sum_{i=1}^{t} \text{size}(e_i). \tag{8.3}$$

The upper bound we will use for simplicity is:

$$\text{size}(f) \leq \text{size}(\alpha) + t\left(H(f) + \text{size}(n)\right), \tag{8.4}$$

where $H(f)$ is defined as $\max_{0 \leq i \leq t} \text{size}(c_i)$. Also, recall as in previous chapters that $\mathsf{M}(r)$ is the cost in ring operations of a dense degree-r multiplication, and $\mathsf{N}(a)$ is the cost in word operations on an IMM of computing the product of two integers, both with absolute value at most a.

Our algorithms will have polynomial complexity in the smallest possible $\text{size}(f)$.

The remainder of this chapter is structured as follows. In Section 8.2 we show how to find the sparsest shift from evaluation points in $\mathbb{F}_p$, where p is a prime with some special properties provided by some "oracle". In Section 8.3 we show how to perform sparse interpolation given a modular black box for a polynomial. In Section 8.4 we show how to generate primes such that a sufficient number satisfy the conditions of our oracle, and discuss the effect of some number-theoretic conjectures on our algorithms. Section 8.5 provides the complexity analysis of our algorithms. We conclude in Section 8.6, and introduce some open questions.

8.2 Computing the Sparsest Shift

For a polynomial $f \in \mathbb{Q}[x]$, we first focus on computing the sparsest shift $\alpha \in \mathbb{Q}$ so that $f(x+\alpha)$ has a minimal number of non-zero and non-constant terms. This information will later be used to recover a representation of the unknown polynomial.

8.2.1 The polynomial $f^{(p)}$

Here, and for the remainder of this paper, for a prime p and $f \in \mathbb{Q}[x]$, define $f^{(p)} \in \mathbb{F}_p[x]$ to be the unique polynomial with degree less than p which is equivalent to f modulo $x^p - x$ and with all coefficients reduced modulo p. Observe that this is very similar to what could be produced by the remainder black box of Chapter 7, but not quite the same. The key difference is that we simultaneously reduce the coefficients and the polynomial itself. Essentially, the remainder black box allows us to work in larger domains (when the unknown polynomial has coefficients in finite fields, for example), whereas our modular black box here allows us to work in the *smaller* domain of integers modulo p.

From Fermat's Little Theorem, we see immediately that $f^{(p)}(\alpha) \equiv f(\alpha) \bmod p$ for all $\alpha \in \mathbb{F}_p$. Hence $f^{(p)}$ can be found by evaluating f at each point $0, 1, \ldots, p-1$ modulo p and using dense interpolation over $\mathbb{F}_p[x]$.

Notice that, over $\mathbb{F}_p[x]$, $(x-\alpha)^p \equiv x-\alpha \mod x^p - x$, and therefore $(x-\alpha)^{e_i} \equiv (x-\alpha)^k$ for any $k \neq 0$ such that $e_i \equiv k \bmod (p-1)$. The smallest such k is in the range $\{1, 2, \ldots, p\}$; we now define this with some more notation. For $a \in \mathbb{Z}$ and positive integer m, define $a \operatorname{rem}_1 m$ to be the unique integer in the range $\{1, 2, \ldots, m\}$ which is congruent to a modulo m. As usual, $a \operatorname{rem} m$ denotes the unique congruent integer in the range $\{0, 1, \ldots, m-1\}$.

If f is as in (8.2), then by reducing term-by-term we can write

$$f^{(p)}(x) = (c_0 \operatorname{rem} p) + \sum_{i=1}^{t} (c_i \operatorname{rem} p)(x - \alpha_p)^{e_i \operatorname{rem}_1 (p-1)}, \tag{8.5}$$

where α_p is defined as $\alpha \operatorname{rem} p$. Hence, for some $k \leq t$, α_p is a k-sparse shift for $f^{(p)}$. That is, the polynomial $f^{(p)}(x + \alpha_p)$ over $\mathbb{F}_p[x]$ has at most t non-zero and non-constant terms.

Computing $f^{(p)}$ from a modular black box for f is straightforward. First, use p black-box calls to determine $f(i) \operatorname{rem} p$ for $i = 0, 1, \ldots, p-1$. Recalling that κ is the number of field operations in $\mathbb{F}_p$ for each black-box call, the cost of this step is $O(p\kappa \mathsf{N}(p))$ word operations. Second, we use the well-known divide-and-conquer method to interpolate $f^{(p)}$ into the dense representation (see, e.g., Borodin and Munro, 1975, Section 4.5). Since $\deg f^{(p)} < p$, this step can be performed in $O(p \log^2 p \, \mathsf{N}(p))$ word operations.

Furthermore, for any $\alpha \in \mathbb{F}_p$, the dense representation of $f^{(p)}(x+\alpha)$ can be computed in exactly the same way as the second step above, simply by shifting the indices of the already-evaluated points by α. This immediately gives a naïve algorithm for computing the sparsest shift of $f^{(p)}$: compute $f^{(p)}(x+\gamma)$ for $\gamma = 0, 1, \ldots, p-1$, and return the γ that minimizes the number of non-zero, non-constant terms. The cost in word operations of this approach is $O(p^2 \log^2 p \, \mathsf{N}(p))$, which for our applications will often be less costly than the more sophisticated approaches of, e.g., Lakshman and Saunders (1996) or (Giesbrecht et al., 2003), precisely because p will not be very much larger than $\deg f^{(p)}$.

8.2.2 Overview of Approach

We will make repeated use of the following fundamental theorem from Lakshman and Saunders (1996):

Fact 8.1. *Let F be an arbitrary field and $f \in \mathsf{F}[x]$, and suppose $\alpha \in \mathsf{F}$ is such that $f(x+\alpha)$ has t non-zero and non-constant terms. If $\deg f \geq 2t+1$ then α is the unique sparsest shift of f.*

From this we can see that, if α is the unique sparsest shift of f, then $\alpha_p = \alpha \operatorname{rem} p$ is the unique sparsest shift of $f^{(p)}$ *provided that* $\deg f^{(p)} \geq 2t+1$. This observation provides the basis for our algorithm.

The input to the algorithms will be a modular black box for evaluating a rational polynomial, as described above, and bounds on the maximal size of the unknown polynomial. Note that such bounds are a necessity in any type of black-box interpolation algorithm, since otherwise we could never be sure that the computed polynomial is really equal to the black-box function at *every* point. Specifically, we require $B_A, B_T, B_H, B_N \in \mathbb{N}$ such that

$$\begin{aligned} \text{size}(\alpha) &\leq B_A, \\ t &\leq B_T, \\ \text{size}(c_i) &\leq B_H, \quad \text{for } 0 \leq i \leq t, \\ \text{size}(n) &\leq B_N. \end{aligned}$$

By considering the following polynomial:

$$c(x-\alpha)^n + (x-\alpha)^{n-1} + \cdots + (x-\alpha)^{n-t+1},$$

we see that these bounds are independent — that is, none is polynomially-bounded by the others — and therefore are all necessary.

We are now ready to present the algorithm for computing the sparsest shift α almost in its entirety. The only part of the algorithm left unspecified is an *oracle* which, based on the values of the bounds, produces primes to use. We want primes p such that $\deg f^{(p)} \geq 2t+1$, which allows us to recover one modular image of the sparsest shift α. But since we do not know the exact value of t or the degree n of f over $\mathbb{Q}[x]$, we define some prime p to be a *good prime for sparsest shift computation* if and only if $\deg f^{(p)} \geq \min\{2B_T+1, n\}$. For the remainder of this section, "good prime" means "good prime for sparsest shift computation." Our oracle indicates when enough primes have been produced so that at least one of them is guaranteed to have been a good prime, which is necessary for the procedure to terminate. The details of how to construct such an oracle will be considered in Section 8.4.

Our algorithm for computing the sparsest shift is presented in Algorithm 8.1.

Theorem 8.2. *With inputs as specified, Algorithm 8.1 correctly returns a sparsest shift α of f.*

Proof. Let f, B_A, B_T, B_H, B_N be the inputs to the algorithm, and suppose t, α are as specified in (8.2).

First, consider the degenerate case where $n \leq 2B_T$, i.e., the bound on the sparsity of the sparsest shift is at least half the actual degree of f. Then, since each $f^{(p)}$ can have degree at most n (regardless of the choice of p), the condition of Step 6 will never be true. Hence Steps 10–14 will eventually be executed. The size of coefficients over the standard power basis is bounded by $2B_T B_A + B_H$ since $\deg f \leq 2B_T$, and therefore f will be correctly computed on Step 5. In this case, Fact 8.1 may not apply, i.e., the sparsest shift may not be unique, but the algorithms from Giesbrecht et al. (2003) will still produce a sparsest shift of f.

Now suppose instead that $n \geq 2B_T+1$. The oracle eventually produces a good prime p, so that $\deg f^{(p)} \geq 2B_T+1$. Since $t \leq B_T$ and $f^{(p)}$ has at most t non-zero and non-constant terms in the $(\alpha \operatorname{rem} p)$-shifted power basis, the value computed as α_p on Step 8 is exactly $\alpha \operatorname{rem} p$, by Fact 8.1. The value of P will also be set to $p > 1$ here, and can only increase. So the condition of Step 10 is never true. Since the numerator and denominator of α are both bounded above by 2^{B_A}, we can use rational reconstruction to compute α once we have the image modulo P for $P \geq 2\alpha^2$. Therefore, when we reach Step 15, we have enough images α_p to recover and return the correct value of α. □

Algorithm 8.1: Computing the sparsest shift

```
Input: A modular black box for an unknown polynomial $f \in \mathbb{Q}[x]$, bounds
       $B_A, B_T, B_H, B_N \in \mathbb{N}$ as described above, and an oracle which produces primes
       and indicates when at least one good prime must have been produced
Output: A sparsest shift $\alpha$ of $f$.
$P \leftarrow 1$,    $\mathcal{G} \leftarrow \emptyset$
while size$(P) \le 2B_A + 1$ do
    $p \leftarrow$ new prime from the oracle
    Evaluate $f(i) \operatorname{rem} p$ for $i = 0, 1, \ldots, p-1$
    Use dense interpolation to compute $f^{(p)}$
    if $\deg f^{(p)} \ge 2B_T + 1$ then
        Use dense interpolation to compute $f^{(p)}(x+\gamma)$ for $\gamma = 1, 2, \ldots, p-1$
        $\alpha_p \leftarrow$ the unique sparsest shift of $f^{(p)}$
        $P \leftarrow P \cdot p$,    $\mathcal{G} \leftarrow \mathcal{G} \bigcup \{p\}$
    else if $P = 1$ and oracle indicates $\ge 1$ good prime has been produced then
        $q \leftarrow$ least prime such that size$(q) > 2B_T B_A + B_H$ (computed directly)
        Evaluate $f(i) \operatorname{rem} q$ for $i = 0, 1, \ldots, 2B_T$
        Compute $f \in \mathbb{Q}[x]$ with $\deg f \le 2B_T$ by dense interpolation in $\mathbb{F}_q[x]$ followed by
        rational reconstruction on the coefficients
        return A sparsest shift $\alpha$ computed by a univariate algorithm from Giesbrecht
        et al. (2003) on input $f$
return The unique $\alpha = a/b \in \mathbb{Q}$ such that $|a|, b \le 2^{B_A}$ and $a \equiv b\alpha_p \bmod p$ for each $p \in \mathcal{G}$,
using Chinese remaindering and rational reconstruction
```

We still need to specify which algorithm to use to compute the sparsest shift of a densely-represented $f \in \mathbb{Q}[x]$ on Step 14. To make Algorithm 8.1 completely deterministic, we should use the univariate symbolic algorithm from Giesbrecht et al. (2003, Section 3.1), although this will have very high complexity. Using a probabilistic algorithm instead gives the following, which follows directly from the referenced work.

Theorem 8.3. *If the "two projections" algorithm of Giesbrecht et al. (2003, Section 3.3) is used on Step 14, then Steps 10–14 of Algorithm 8.1 can be performed with*

$$O(B_T^2 \mathsf{M}(B_T^4 B_A + B_T^3 B_H))$$

word operations, plus $O(\kappa B_T \mathsf{M}(B_T B_A + B_H))$ word operations for the black-box evaluations.

The precise complexity analysis proving that the entire Algorithm 8.1 has cost polynomial in the bounds given depends heavily on the size and number of primes p that are used, and so must be postponed until Section 8.5.1, after our discussion on choosing primes.

Example 8.4. *Suppose we are given a modular black box for the following unknown polynomial:*

$$\begin{aligned} f(x) = x^{15} - 45x^{14} + 945x^{13} - 12285x^{12} + 110565x^{11} - 729729x^{10} \\ + 3648645x^9 - 14073345x^8 + 42220035x^7 - 98513415x^6 + \\ 177324145x^5 - 241805625x^4 + 241805475x^3 - 167403375x^2 \\ + 71743725x - 14348421, \end{aligned}$$

along with the bounds $B_A = 4$, $B_T = 2$, $B_H = 4$, and $B_N = 4$. Under the simplistic assumption of single bit-length words, i.e., $w = 1$, one may easily confirm that $f(x) = (x-3)^{15} - 2(x-3)^5$, and hence these bounds are actually tight.

Now suppose the oracle produces $p = 7$ in Step 3. We use the black box to compute each $f(0), f(1), \ldots, f(6)$ in $\mathbb{F}_7$, and dense interpolation to compute

$$f^{(7)}(x) = 5x^5 + 2x^4 + 3x^3 + 6x^2 + x + 4.$$

Since $\deg f^{(7)} = 5 \geq 2B_T + 1$, *we move on to Step 8 and compute each $f^{(7)}(x+\gamma)$ with $\gamma = 1, 2, \ldots, 6$. Examining these, we see that $f^{(7)}(x+3) = 5x^5 + x^3$ has the fewest non-zero and non-constant terms, and so set α_7 to 3 on Step 8. This means the sparsest shift must be congruent to 3 modulo 7. This provides a single modular image for use in Chinese remaindering and rational reconstruction on Step 15, after enough successful iterations for different primes p.*

8.2.3 Conditions for Success

We have seen that, provided $\deg f > 2B_T$, a good prime p is one such that $\deg f^{(p)} > 2B_T$. The following theorem provides (quite loose) sufficient conditions on p to satisfy this requirement.

Theorem 8.5. *Let $f \in \mathbb{Q}[x]$ as in (8.2) and $B_T \in \mathbb{N}$ such that $t \leq B_T$. Then, for some prime p, the degree of $f^{(p)}$ is greater than $2B_T$ whenever the following hold:*

- $c_t \not\equiv 0 \mod p$;
- $\forall i \in \{1, 2, \ldots, t-1\}$, $e_t \not\equiv e_i \mod (p-1)$;
- $\forall i \in \{1, \ldots, 2B_T\}$, $e_t \not\equiv i \mod (p-1)$.

Proof. The first condition guarantees that the last term of $f^{(p)}(x)$ as in (8.5) does not vanish. We also know that there is no other term with the same degree from the second condition. Finally, the third condition tells us that the degree of the last term will be greater than $2B_T$. Hence the degree of $f^{(p)}$ is greater than $2B_T$. □

For purposes of computation it will be convenient to simplify the above conditions to two non-divisibility requirements, on p and $p-1$ respectively:

Corollary 8.6. *Let f, B_T, B_H, B_N be as in the input to Algorithm 8.1 with* $\deg f > 2B_T$. *Then there exist $C_1, C_2 \in \mathbb{N}$ with* $\text{size}(C_1) \leq 2B_H$ *and* $\text{size}(C_2) \leq B_N(3B_T - 1)$ *such that* $\deg f^{(p)} > 2B_T$ *whenever $p \nmid C_1$ and $(p-1) \nmid C_2$.*

Proof. Write f as in (8.2). We will use the sufficient conditions given in Theorem 8.5. Write $|c_t| = a/b$ for $a, b \in \mathbb{N}$ relatively prime. In order for $c_t \operatorname{rem} p$ to be well-defined and not zero, neither a nor b can vanish modulo p. This is true whenever $p \nmid ab$. Set $C_1 = ab$. Since $\text{size}(a), \text{size}(b) \leq B_H$, $\text{size}(C_1) = \text{size}(ab) \leq 2B_H$.

Now write

$$C_2 = \prod_{i=1}^{t-1} (e_t - e_i) \cdot \prod_{i=1}^{2B_T} (e_t - i).$$

We can see that the second and third conditions of Theorem 8.5 are satisfied whenever $(p-1) \nmid C_2$. Now, since each integer e_i is distinct and positive, and e_t is the greatest of these, each $(e_t - e_i)$ is a positive integer less than e_t. Similarly, since $e_t = \deg f > 2B_T$, each $(e_t - i)$ in the second product is also a positive integer less than e_t. Therefore, using the fact that $t \leq B_T$, we see $C_2 \leq e_t^{3B_T - 1}$. Furthermore, $\text{size}(e_t) \leq B_N$, so we know that $\text{size}(C_2) \leq B_N(3B_T - 1)$. □

A similar criteria for success is required in (Bläser et al., 2009), and they employ Linnik's theorem which gives a polynomial-time algorithm, with a high exponent. Linnik's theorem was also employed in (Giesbrecht and Roche, 2007) to yield a much more expensive polynomial-time algorithm for finding sparse shifts than the one presented here.

8.3 Interpolation

Once we know the value of the sparsest shift α of f, we can trivially construct a modular black box for the t-sparse polynomial $f(x+\alpha)$ using the modular black box for f. Therefore, for the purposes of interpolation, we can assume $\alpha = 0$, and focus only on interpolating a t-sparse polynomial $f \in \mathbb{Q}[x]$ given a modular black box for its evaluation.

The approach we use is very similar to the techniques of the previous chapter, except that the modular black box differs from a remainder black box in a subtle way, that the coefficients and the exponents are both reduced (although not modulo the same integer!). This complication makes our analysis much more involved here. Also, the fact that we must use very small primes means that the diversification techniques of Chapter 7 will not work.

For convenience, we restate the notation for f and $f^{(p)}$, given a prime p:

$$f = c_0 + c_1 x^{e_1} + c_2 x^{e_2} + \cdots + c_t x^{e_t}, \tag{8.6}$$

$$f^{(p)} = (c_0 \operatorname{rem} p) + (c_1 \operatorname{rem} p) x^{e_1 \operatorname{rem}_1 (p-1)} + \cdots + (c_t \operatorname{rem} p) x^{e_t \operatorname{rem}_1 (p-1)}. \tag{8.7}$$

Again, we assume that we are given upper bounds B_H, B_T, and B_N on $\max_i \operatorname{size}(c_i)$, t, and $\operatorname{size}(\deg f)$, respectively. We also re-introduce the notation $\tau(f)$, which as in previous sections is defined to be the number of distinct non-zero, non-constant terms in the univariate polynomial f.

This algorithm will again use the polynomials $f^{(p)}$ for primes p, but now rather than a degree condition, we need $f^{(p)}$ to have the maximal number of non-constant terms. So we define a prime p to be a *good prime for interpolation* if and only if $\tau(f^{(p)}) = t$. Again, the term "good prime" refers to this kind of prime for the remainder of this section.

Now suppose we have already used modular evaluation and dense interpolation (as in Algorithm 8.1) to recover the polynomials $f^{(p)}$ for k distinct good primes $p_1, \ldots, p_k$. We therefore have k images of each exponent e_i modulo $(p_1 - 1), \ldots, (p_k - 1)$. Write each of these polynomials as:

$$f^{(p_i)} = c_0^{(i)} + c_1^{(i)} x^{e_1^{(i)}} + \cdots + c_t^{(i)} x^{e_t^{(i)}}. \tag{8.8}$$

Note that it is *not* generally the case that $e_j^{(i)} = e_j \operatorname{rem}_1 (p_i - 1)$. Because we don't know how to associate the exponents in each polynomial $f^{(p_i)}$ with their pre-image in $\mathbb{Z}$, a simple Chinese remaindering on the exponents will not work. Possible approaches are provided by (Kaltofen, 1988), (Kaltofen et al., 1990) or (Avendaño et al., 2006). However, the most suitable approach for our purposes is the clever technique of (Garg and Schost, 2009), based on ideas of Grigoriev and Karpinski (1987), as we saw in the previous chapter. We interpolate the polynomial

$$g(z) = (z - e_1)(z - e_2) \cdots (z - e_t), \tag{8.9}$$

whose coefficients are symmetric functions in the e_i's. Given $f^{(p_i)}$, we have all the values of $e_j^{(i)} \operatorname{rem}_1 (p_i - 1)$ for $j = 1, \ldots, t$; we just don't know the order. But since g is not dependent on the order, we can compute $g \bmod (p_i - 1)$ for $i = 1, \ldots, k$, and then find the roots of $g \in \mathbb{Z}[x]$ to determine the exponents $e_1, \ldots, e_t$. Of course, this is exactly the step that our diversification technique avoided in the last chapter, but again this is not possible in this case because the primes p that we use are too small.

Once we know the exponents, we recover the coefficients from their images modulo each prime. The correct coefficient in each $f^{(p)}$ can be identified because the residues of the exponents modulo $p-1$ are unique, for each chosen prime p. This approach is made explicit in the following algorithm.

Algorithm 8.2: Sparse Polynomial Interpolation over $\mathbb{Q}[x]$

```
Input: A modular black box for unknown $f \in \mathbb{Q}[x]$, bounds $B_H$ and $B_N$ as described
       above, and an oracle which produces primes and indicates when at least one
       good prime must have been returned
Output: $f \in \mathbb{Q}[x]$ as in (8.6)
$Q \leftarrow 1, \quad P \leftarrow 1, \quad k \leftarrow 1, \quad t \leftarrow 0$
while size$(P) \le 2B_H + 1$ or size$(Q) < B_N$
or the oracle does not guarantee a good prime has been produced do
    $p_k \leftarrow$ new prime from the oracle
    Compute $f^{(p_k)}$ by black box calls and dense interpolation
    if $\tau(f^{(p_k)}) > t$ then
        $Q \leftarrow p_k - 1$, $P \leftarrow p_k$, $t \leftarrow \tau(f^{(p_k)})$, $p_1 \leftarrow p_k$, $f^{(p_1)} \leftarrow f^{(p_k)}$, $k \leftarrow 2$
    else if $\tau(f^{(p_k)}) = t$ then
        $Q \leftarrow \mathrm{lcm}(Q, p_k - 1), \quad P \leftarrow P \cdot p_k, \quad k \leftarrow k + 1$
for $i \in \{1, \ldots, k-1\}$ do
    $g^{(p_i)} \leftarrow \prod_{1 \le j \le t} (z - e_j^{(i)}) \bmod p_i - 1$
Construct $g = a_0 + a_1 z + a_2 z^2 + \cdots + a_t z^t \in \mathbb{Z}[x]$ such that $g \equiv g^{(p_i)} \bmod p_i - 1$ for
$1 \le i < k$, by Chinese remaindering
Factor $g$ as $(z - e_1)(z - e_2) \cdots (z - e_t)$ to determine $e_1, \ldots, e_t \in \mathbb{Z}$
for $1 \le i \le t$ do
    for $1 \le j \le k$ do
        Find the exponent $e_{\ell_j}^{(j)}$ of $f^{(p_j)}$ such that $e_{\ell_j}^{(j)} \equiv e_i \bmod p_j - 1$
    Reconstruct $c_i \in \mathbb{Q}$ by Chinese remaindering from residues $c_{\ell_1}^{(1)}, \ldots, c_{\ell_k}^{(k)}$
Reconstruct $c_0 \in \mathbb{Q}$ by Chinese remaindering from residues $c_0^{(1)}, \ldots, c_0^{(k)}$
```

The following theorem follows from the above discussion.

Theorem 8.7. *Algorithm 8.2 works correctly as stated.*

Again, this algorithm runs in polynomial time in the bounds given, but we postpone the detailed complexity analysis until Section 8.5.2, after we discuss how to choose primes from the "oracle". Some small practical improvements may be gained if we use Algorithm 8.2 to interpolate $f(x+\alpha)$ after running Algorithm 8.1 to determine the sparsest shift α, since in this case we will have a few previously-computed polynomials $f^{(p)}$. However, we do not explicitly consider this savings in our analysis, as there is not necessarily any asymptotic gain.

Now we just need to analyze the conditions for primes p to be good. This is quite similar to the analysis of the sparsest shift algorithm above, so we omit many of the details here.

Theorem 8.8. *Let f, B_T, B_H, B_N be as above. There exist $C_1, C_2 \in \mathbb{N}$ with* $\operatorname{size}(C_1) \le 2B_H B_T$ *and* $\operatorname{size}(C_2) \le \frac{1}{2} B_N B_T (B_T - 1)$ *such that $\tau(f^{(p)})$ is maximal whenever $p \nmid C_1$ and $(p-1) \nmid C_2$.*

Proof. Let f be as in (8.6), write $|c_i| = a_i/b_i$ in lowest terms for $i = 1, \ldots, t$, and define

$$C_1 = \prod_{i=1}^{t} a_i b_i, \quad C_2 = \prod_{i=1}^{t} \prod_{j=i+1}^{t} (e_j - e_i).$$

Now suppose p is a prime such that $p \nmid C_1$ and $(p-1) \nmid C_2$. From the first condition, we see that each $c_i \bmod p$ is well-defined and nonzero, and so none of the terms of $f^{(p)}$ vanish. Furthermore, from the second condition, $e_i \not\equiv e_k \bmod p-1$ for all $i \ne j$, so that none of the terms of $f^{(p)}$ collide. Therefore $f^{(p)}$ contains exactly t non-constant terms. The bounds on C_1 and C_2 follow from the facts that each $\operatorname{size}(a_i), \operatorname{size}(b_i) \le B_H$ and each difference of exponents $(e_j - e_i)$ has size at most B_N. □

8.4 Generating primes

We now turn our attention to the problem of generating primes for the sparsest shift and interpolation algorithms. We first present our algorithm for generating suitable primes, and we make use of powerful results from analytic number theory to estimate its running time. However, experience suggests that the true complexity of our algorithm is even better than we can prove. We briefly examine some related mathematical problems and indications towards the true cost of our method.

8.4.1 Prime generation algorithm

Recall that the algorithms above for sparsest shift computation and interpolation assumed we had an "oracle" for generating good primes, and indicating when at least one good prime must have been produced. We now present an explicit and analyzed algorithm for this problem.

The definition of a "good prime" is not the same for the algorithms in Section 8.2 and Section 8.3. However, Corollary 8.6 and Theorem 8.8 provide a unified presentation of sufficient conditions for primes being "good". Here we call a prime which satisfies those sufficient conditions a *useful prime*. So every useful prime is good (with the bounds appropriately specified for the relevant algorithm), but some good primes might not be useful.

We first describe a set $\mathscr{P}$ of primes such that the number and density of useful primes within the set is sufficiently high. We will assume that there exist numbers C_1, C_2, and useful primes p are those such that $p \nmid C_1$ and $(p-1) \nmid C_2$. The numbers C_1 and C_2 will be unknown, but we will assume we are given bounds β_1, β_2 such that $\log_2 C_1 \le \beta_1$ and $\log_2 C_2 \le \beta_2$. Suppose we want to find ℓ useful primes. We construct $\mathscr{P}$ explicitly, of a size guaranteed to contain enough useful primes, then enumerate it.

The following fact is immediate from (Mikawa, 2001), though it has been somewhat simplified here, and the use of (unknown) constants is made more explicit. This will be important in our computational methods.

For $q \in \mathbb{Z}$, let $S(q)$ be the smallest prime p such that $q \mid (p-1)$.

Fact 8.9 (Mikawa 2001)**.** *There exists a constant* $\mu > 0$*, such that for all* $n > \mu$*, and for all integers* $q \in \{n,\ldots,2n\}$ *with fewer than* $\mu n/\log^2 n$ *exceptions, we have* $S(q) < q^{1.89}$.

Our algorithms for generating useful primes require explicit knowledge of the value of the constant μ in order to run correctly. So we will assume that we know μ in what follows. To get around the fact that we do not, we simply start by assuming that $\mu = 1$, and run any algorithm depending upon it. If the algorithm fails we simply double our estimate for μ and repeat. At most a constant number of doublings is required. We make no claim this is particularly practical.

For convenience we define

$$\Upsilon(x) = \frac{3x}{5\log x} - \frac{\mu x}{\log^2 x}.$$

Theorem 8.10. *Let* $\log_2 C_1 \le \beta_1$, $\log_2 C_2 \le \beta_2$ *and* ℓ *be as above. Let* n *be the smallest integer such that* $n > 21$, $n > \mu$ *and* $\Upsilon(n) > \beta_1 + \beta_2 + \ell$*. Define*

$$\mathcal{Q} = \{q \text{ prime} : n \le q < 2n \text{ and } S(q) < q^{1.89}\}, \quad \mathcal{P} = \{S(q) : q \in \mathcal{Q}\}.$$

Then the number of primes in $\mathcal{P}$ *is at least* $\beta_1 + \beta_2 + \ell$*, and the number of useful primes in* $\mathcal{P}$*, such that* $p \nmid C_1$ *and* $(p-1) \nmid C_2$*, is at least* ℓ*. For all* $p \in \mathcal{P}$ *we have* $p \in O((\beta_1 + \beta_2 + \ell)^{1.89} \cdot \log^{1.89}(\beta_1 + \beta_2 + \ell))$.

Proof. By (Rosser and Schoenfeld, 1962), the number of primes between n and $2n$ is at least $3n/(5\log n)$ for $n \ge 21$. Applying Fact 8.9, we see $\#\mathcal{Q} \ge 3n/(5\log n) - \mu n/\log^2 n$ when $n \ge \max\{\mu, 21\}$. Now suppose $S(q_1) = S(q_2)$ for $q_1, q_2 \in \mathcal{Q}$. If $q_1 < q_2$, then $S(q_1) > q_1^2$, a contradiction with the definition of $\mathcal{Q}$. So we must have $q_1 = q_2$, and hence

$$\#\mathcal{P} = \#Q \ge \Upsilon(n) > \beta_1 + \beta_2 + \ell.$$

We know that there are at most $\log_2 C_1 \le \beta_2$ primes $p \in \mathcal{P}$ such that $p \mid C_1$. We also know that there are at most $\log_2 C_2 \le \beta_2$ primes $q \in \mathcal{Q}$ such that $q \mid C_2$, and hence at most $\log_2 C_2$ primes $p \in \mathcal{P}$ such that $p = S(q)$ and $q \mid (p-1) \mid C_1$. Thus, by construction $\mathcal{P}$ contains at most $\beta_1 + \beta_2$ primes that are not useful out of $\beta_1 + \beta_2 + \ell$ total primes.

To analyze the size of the primes in $\mathcal{P}$, we note that to make $\Upsilon(n) > \beta_1 + \beta_2 + \ell$, we have $n \in \Theta((\beta_1 + \beta_2 + \ell) \cdot \log(\beta_1 + \beta_2 + \ell))$ and each $q \in \mathcal{Q}$ satisfies $q \in O(n)$. Elements of $\mathcal{P}$ will be of magnitude at most $(2n)^{1.89}$ and hence $p \in O((\beta_1 + \beta_2 + \ell)^{1.89} \log^{1.89}(\beta_1 + \beta_2 + \ell))$. □

Given β_1, β_2 and ℓ as above (where $\log_2 C_1 \le \beta_1$ and $\log_2 C_2 \le \beta_2$ for unknown C_1 and C_2), we generate the primes in $\mathcal{P}$ as follows.

Start by assuming that $\mu = 1$, and compute n as the smallest integer such that $\Upsilon(n) > \beta_1 + \beta_2 + \ell$, $n \ge \mu$ and $n \ge 21$. List all primes between n and $(2n)^{1.89}$ using a Sieve of Eratosthenes, and store these primes in a data structure that allows constant-time membership queries. For instance, we could simply use a length-$(2n)^{1.89}$ bit array where the ith bit is set iff i is prime. In practice, it would be more efficient to use a hash table for this purpose.

For each prime q between n and $2n$, determine $S(q)$, if it is less than $q^{1.89}$, by simply checking if $kq+1$ is prime for $k = 1, 2, \ldots, \lfloor q^{0.89} \rfloor$. If we find a prime $p = S(q) < q^{1.89}$, add p to $\mathscr{P}$. This is repeated until $\mathscr{P}$ contains $\beta_1 + \beta_2 + \ell$ primes. If we are unable to find this number of primes, we have underestimated μ (since Theorem 8.10 guarantees their existence), so we double μ and restart the process. Obviously in practice we would not redo primality tests already performed for smaller μ, so really no work need be wasted.

Theorem 8.11. *For $\log_2 C_1 \le \beta_1$, $\log_2 C_2 \le \beta_2$, ℓ, and n as in Theorem 8.10, we can generate $\beta_1 + \beta_2 + \ell$ elements of $\mathscr{P}$ with*

$$O((\beta_1+\beta_2+\ell)^{1.89} \cdot \log^{1.89}(\beta_1+\beta_2+\ell) \cdot \operatorname{logloglog}(\beta_1+\beta_2+\ell))$$

word operations. At least ℓ of the primes in $\mathscr{P}$ will be useful.

Proof. The method and correctness follows from the above discussion. The Sieve of Eratosthenes can be run with $O(n^{1.89} \operatorname{logloglog} n)$ bit operations (see Knuth, 1981, §4.5.4), including the construction of the bit-array as described above. Each primality test of $kq+1$ then takes constant time, and there are at most $n^{1.89}$ such tests. Since

$$n \in O((\beta_1+\beta_2+\ell) \cdot \log(\beta_1+\beta_2+\ell)),$$

the stated complexity follows. □

8.4.2 Using smaller primes

The analysis of our methods will be significantly improved when more is discovered about the behavior of the least prime congruent to one modulo a given prime, which we have denoted $S(q)$. This is a special case of the more general question of the least prime in an arbitrary arithmetic progression, a well-studied problem in the mathematical literature. Recall from Section 1.4 that Linnik's theorem guarantees us that $S(q) \ll q^C$ for some constant C. A series of results has sought explicit bounds for C, with the most recent progress by Xylouris (2009) giving $C \le 5.2$.

Assuming the extended Riemann hypothesis (ERH), this can be reduced unconditionally to $S(q) \ll q^2 \ln^2 q$, as reported by Bach (1990) and Heath-Brown (1992). The result of Mikawa (2001) that we employed above is even stronger, and does not require the ERH. However, recall that Mikawa's bound of $S(q) \ll q^{1.89}$ is not shown to hold for all q, and this is why we had to handle exceptions.

What is the true asymptotic behaviour of $S(q)$? No one knows for certain, but it seems that the answer is $S(q) \sim q \ln^2 q$. Granville and Pomerance (1990) conjecture this as an asymptotic lower bound for $S(q)$, and earlier Heath-Brown (1978) conjectured a upper bound of the same form. Today this is known as Wagstaff's conjecture, after the more general statement by Wagstaff (1979), which was also supported by an extensive (at that time) computational search, confirming the conjecture for $q < 10^6$. He also gave a heuristic argument that this should be the true asymptotic growth rate of the least prime in arithmetic progressions.

We conducted our own computational search to investigate the size of $S(q)$ and have found that $S(q) < 2q\ln^2 q$ for all primes q such that $2q\ln^2 q < 2^{64}$ — that is, all q for which $S(q)$ fits in a 64-bit machine word. As a point of reference, this limit is approximately $q < 10^{16}$. In fact, this search was the original motivation for some of the tricks in modular arithmetic discussed in Chapter 2. Our computation also used the result of Jaeschke (1993) which shows that only 9 Miller-Rabin trials are necessary to deterministically check primality for integers of this size.

The results of our computational search make a strong statement about the true complexity of our algorithms. On machines with at most 64-bit words (which is to say, nearly any modern computer), our algorithms cannot possibly use primes p larger than 2^{64}, since in this case the dense polynomial $f^{(p)}$ would not fit into addressable memory. (Recall the lengthy discussion of word-size in the IMM model from Chapter 1.) This means that, at least on modern hardware, the true cost of our algorithms will be given by assuming $S(q) \in O(q\log^2 q)$. In any case, the actual cost of the algorithms discussed — on any machine — will be a reflection of the true behavior of $S(q)$, even before it is completely understood by us.

Even more improvements might be possible if this rather complicated construction is abandoned altogether, as useful primes would naturally seem to be relatively plentiful. In particular, one would expect that if we randomly choose primes p directly from a set which has, say, $4(\beta_1 + \beta_2 + \ell)$ primes, we might expect that the probability that $p \mid C_1$ or $(p-1) \mid C_2$ to less than, say, 1/4. Proving this directly appears to be difficult. Perhaps most germane results to this are lower bounds on the Carmichael Lambda function (which for the product of distinct primes $p_1, \ldots, p_m$ is the LCM of $p_1 - 1, \ldots, p_m - 1$), which are too weak for our purposes. See (Erdös, Pomerance, and Schmutz, 1991).

8.5 Complexity analysis

We are now ready to give a formal complexity analysis for the algorithms presented in Section 8.2 and Section 8.3. For all algorithms, the complexity is polynomial in the four bounds B_A, B_T, B_H, and B_N defined in Section 8.2.2, as well as the word-size w in the IMM model. Since these are each bounded above by size(f), our algorithms will have polynomial complexity in the size of the output if these bounds are sufficiently tight.

It may be surprising to see the word-size w appear in the complexity, so we briefly give an intuitive explanation for this phenomenon. Recall that the bounds B_A, B_H, and B_N above are bounds on the number of words used to store the relevant values in the output. And our algorithm analysis of course will count word operations. Because of this, doing integer computations on very small integers much smaller than 2^w is wasteful in some sense. That is, in order to get the best complexity in word operations, we should always make sure to get the most computational power out of those word operations that we can.

Now observe that our algorithms will *definitely* be wasteful in this sense, because of the primes p that drive the cost of the whole algorithm. We are doing many computations in the field $\mathbb{F}_p$, and so to maximize the value of word operations, we would normally want every p to be close to the maximum value that will fit in a machine word, 2^w. However, the cost of our algorithm depends not only on the size of p, but on the *value* of p as well, since the algorithm repeatedly generates dense polynomials with degrees bounded by p. So the ps must be chosen as small as possible to avoid making the whole algorithm intractable. This inevitably means that computations modulo p will be wasteful, and the parameter w that appears in the complexity measures indicates the degree of this wastefulness.

As a comparative illustration, suppose we were given an instance of the long integer multiplication problem, encoded in words, but chose to operate only on bits. Then the size-n input would have bit-length wn, and our cost (in word operations) would be proportional to the latter value, reflecting the fact that every word operation was wasteful by only computing with single bits. A similar phenomenon is happening here. Observe, however, that as w must be bounded by the logarithm of the input size, the effect of factors of w in the complexity will not be very significant.

8.5.1 Complexity of Sparsest Shift Computation

Algorithm 8.1 gives our algorithm to compute the sparsest shift α of an unknown polynomial $f \in \mathbb{Q}[x]$ given bounds B_A, B_T, B_H, and B_N and an oracle for choosing primes. The details of this oracle are given in Section 8.4.

To choose primes, we set $\ell = 2B_A w + 1$, and $\beta_1 = 2B_H w$ and $\beta_2 = B_N(3B_T - 1)w$ (according to Corollary 8.6 and the definitions of $\beta_1, beta_2, \ell$ in the previous section). i For the sake of notational brevity, define $B_\Sigma = (B_A + B_H + B_N B_T)w$ so that $\beta_1 + \beta_2 + \ell \in O(B_\Sigma)$.

Theorem 8.12. *Suppose $f \in \mathbb{Q}[x]$ is an unknown polynomial given by a black box, with bounds B_A, B_T, B_H, and B_N given as above. If $\deg f > 2B_T$, then the sparsest shift $\alpha \in \mathbb{Q}$ of f can be computed deterministically using*

$$O\left(w\, B_A\, B_\Sigma^{3.78} \cdot \log^{5.78} B_\Sigma\right)$$

word operations, plus $O(\kappa B_\Sigma^{2.89} \log^{1.89} B_\Sigma)$ word operations for the black-box evaluations.

Proof. Algorithm 8.1 will always satisfy the conditions of Step 2 and terminate after $2B_A w + 1$ good primes have been produced by the oracle.

Using the oracle to choose primes, and because $\beta_1 + \beta_2 + \ell \in O(B_\Sigma)$,

$$(B_\Sigma^{1.89} \cdot \log^{1.89} B_\Sigma \cdot \text{logloglog}\, B_\Sigma)$$

word operations are used to compute all the primes on Step 3, by Theorem 8.11. This cost does not dominate the complexity. Also, by Theorem 8.10, each chosen p is bounded by $O(B_\Sigma^{1.89} \log^{1.89} B_\Sigma)$.

All black-box evaluations are performed on Step 4. There are p evaluations at each iteration, and $O(B_\Sigma)$ iterations, and observe that $p < B_\Sigma^{O(1)}$, and we can safely assume this latter value is word-sized, since B_Σ is polynomially-related to the instance size. Therefore every operation in $\mathbb{F}_p$ takes $O(1)$ word operations, a total cost of $O(\kappa B_\Sigma p)$ word operations. Using the fact that $p \in O(B_\Sigma^{1.89} \log^{1.89} B_\Sigma)$ gives the stated result.

Steps 10–14 are never executed when $\deg f > 2B_T$. Step 15 is only executed once and never dominates the complexity.

Dense polynomial interpolation over $\mathbb{F}_p$ is performed at most $O(B_\Sigma)$ times on Step 5 and $O(p)$ times at each of $O(wB_A)$ iterations through Step 7. Since $p \gg B_\Sigma$, the latter step dominates. Using asymptotically fast methods, each interpolation of $f^{(p)}(x+\gamma)$ uses $O(\mathsf{M}(p)\log p)$ field operations in $\mathbb{F}_p$, each of which again uses just a constant number of word operations. Also, from Chapter 1, recall that we can assume $\mathsf{M}(p) \in O(p \log p)$ by encoding the polynomials in $\mathbb{F}_p[x]$ into long integers. This gives a total cost over all iterations of $O(wB_A\, p^2 \log^2 p)$ (a slight abuse of notation here since the value of p varies). Again, using the fact that $p \in O(B_\Sigma^{1.89} \log^{1.89} B_\Sigma)$ gives the stated result. □

To simplify the discussion somewhat, consider the case that we have only a *single* bound on the size of the output polynomial, say $B_f \geq \text{size}(f)$. By setting each of B_T, B_H, and B_N equal to B_f, and because $w \in O(\log \text{size}(f))$, we obtain the following comprehensive result:

Corollary 8.13. *If an unknown polynomial $f \in \mathbb{Q}[x]$ has shifted-lacunary size bounded by B_f, then the sparsest shift α of f can be computed using $O\left(B_f^{8.56} \log^{10.56} B_f\right)$ word operations, plus $O\left(\kappa B_f^{5.78} \log^{4.78} B_f\right)$ word operations for the black-box evaluations.*

Proof. The stated complexities follow directly from Theorem 8.12 above, using the fact that $B_\Sigma \in O(wB_f^2)$. Using the single bound B_f, we see that these costs always dominate the cost of Steps 10–14 given in Theorem 8.3, and so we have the stated general result. □

In fact, if we have no bounds at all *a priori*, we could start by setting B_f to some small value (perhaps dependent on the size of the black box or κ), running Algorithm 8.1, then doubling B_f and running the algorithm again, and so forth until the same polynomial f is computed in successive iterations. This can then be tested on random evaluations. Such an approach yields an output-sensitive polynomial-time algorithm which should be correct on most input, though it could certainly be fooled into early termination.

Somewhat surprisingly, our algorithm is competitive even with the best-known sparsest shift algorithms which require a (dense) $f \in \mathbb{Q}[x]$ to be given explicitly as input. By carefully constructing the modular black box from a given $f \in \mathbb{Q}[x]$, and being sure to set $B_T < (\deg f)/2$, we can derive from Algorithm 8.1 a deterministic sparsest-shift algorithm with bit complexity close to the fastest algorithms in Giesbrecht et al. (2003); the dependence on degree n and sparsity t will be somewhat less, but the dependence on the size of the coefficients size(f) is greater.

To understand the limits of our computational techniques (as opposed to our current understanding of the least prime in arithmetic progressions) we consider the cost of our algorithms under the assumption that $S(q) \in O(q \log^2 q)$, as discussed in the previous section. In this case the sparsest shift α of an unknown polynomial $f \in \mathbb{Q}[x]$, whose size in the sparsest-shifted representation is bounded by B_f, can be computed using $\tilde{O}(B_f^5)$ word operations. As noted in the previous section, we have verified computationally that $S(q) \leq 2q \ln^2 q$ for all 64-bit primes $p = S(q)$. This would suggest the above complexity for all sparsest-shift interpolation problems that we would expect to encounter.

8.5.2 Complexity of Interpolation

The complexity analysis of the sparse interpolation algorithm given in Algorithm 8.2 will be quite similar to that of the sparsest shift algorithm above. Here, we need

$$\ell = w \cdot \max\{2B_H + 1, B_N\}$$

good primes to satisfy the conditions of Step 3, and from Theorem 8.8, we set $\beta_1 = 2wB_HB_T$ and $\beta_2 = \frac{1}{2}wB_NB_T(B_T - 1)$. Hence for this subsection we set

$$B_S = wB_T(B_H + B_NB_T),$$

so that $\beta_1 + \beta_2 + \ell \in O(B_S)$.

Theorem 8.14. *Suppose $f \in \mathbb{Q}[x]$ is an unknown polynomial given by a modular black box, with bounds B_T, B_H, B_N, and B_S given as above. The sparse representation of f as in (8.2) can be computed with $O(B_S^{2.89} \log^{3.89} B_S)$ word operations, plus $O(\kappa B_S^{2.89} \log^{1.89} B_S)$ word operations for the black-box evaluations.*

Proof. As in the sparsest-shift computation, the cost of choosing primes in Step 4 is

$$O(B_S^{1.89} \log^{1.89} B_S \cdot \text{logloglog}\, B_S)$$

word operations, and each chosen prime p_k satisfies $p_k \in O(B_S^{1.89} \log^{1.89} B_S)$.

The cost of Step 5 is $O(p \log^2 p)$ word operations at each of the $O(B_S)$ iterations, for a total cost of $O(pB_S \log^2 p)$ word operations. The black-box evaluations all occur on this step, and their total cost is $O(\kappa pB_S)$ word operations, which gives the stated cost.

We can compute each $g^{(p_i)}$ in Step 11 using $O(\mathsf{M}(t) \log t)$ ring operations modulo $p_i - 1$. Note that k is bounded by $O(\ell)$, which in turn is $O(w \cdot (B_H + B_N))$. This gives the total cost in word operations for all iterations of this step as $O(w \cdot (B_H + B_N)B_T \log^2 B_T)$.

Step 12 performs t Chinese Remainderings each of k modular images, and the size of each resulting integer is bounded by B_N, for a cost of $O(B_TB_N \log^2 B_N)$ word operations.

To factor g in Step 13, we can use Algorithm 14.17 of von zur Gathen and Gerhard (2003), which has a total cost in word operations of

$$O\left(B_T^3 B_N^2 \log^2 B_T \cdot (\log B_T + \log B_N)^2\right)$$

because the degree of g is t, g has t distinct roots, and each coefficient has size bounded by $O(B_TB_N)$.

In Step 16, we must first compute the modular image of $e_i \bmod p_j - 1$ and then look through all t exponents of $f^{(p_j)}$ to find a match. This is repeated tk times. We can use fast modular reduction to compute all the images of each e_i using $O(B_N \log^2 B_N)$ bit operations, so the total cost is $O(wB_T^2(B_H + B_N) + B_T B_N \log^2 B_N)$ word operations.

Finally, we perform Chinese remaindering and rational reconstruction of $t+1$ rational numbers, each of whose size is bounded by B_H, for a total cost of $O(B_T B_H \log^2 B_H)$ word operations.

Therefore we see that the complexity is always dominated by the dense interpolation in Step 5. □

Once again, by having only a single bound on the size of the output, the complexity measures are greatly simplified.

Corollary 8.15. *If the lacunary representation size of an unknown polynomial $f \in \mathbb{Q}[x]$ is bounded by B_f, then that representation can be interpolated from a modular black box using $O\left(B_f^{8.67} \log^{6.78} B_f\right)$ word operations, plus $O\left(\kappa B_f^{8.67} \log^{4.78} B_f\right)$ word operations for the black box evaluations.*

Similar improvements to those discussed at the end of Section 8.5.1 can be obtained under stronger (but unproven) number theoretic assumptions.

8.6 Conclusions and Future Work

Here we provide the first algorithm to interpolate an unknown univariate rational polynomial into the sparsest shifted power basis in time polynomial in the size of the output. The main tool we have introduced is mapping down modulo small primes where the sparse shift is also mapped nicely. This technique could be useful for other problems involving sparse polynomials as well, although it is not clear how it would apply in finite domains where there is no notion of "size".

The complexity of our algorithm is now fairly good compared to previous approaches (that required f to be given explicitly as input). However, as mentioned, improved results bounding the size of primes in arithmetic progressions could improve the provable complexity of our algorithm a bit further. A different approach suggested by the result of Baker and Harman (1996) might also be useful to us in choosing smaller primes. Rather than start with primes q and find larger primes p congruent to 1 modulo q, as we have done, they start with the larger prime p and give lower bounds on the greatest prime divisor of $p-1$. It is possible that this type of result could improve our complexity, but we have not yet fully investigated the possibility.

There are many further avenues to consider, the first of which might be multivariate polynomials with a shift in each variable (see, e.g., Grigoriev and Lakshman (2000)). It would be easy to adapt our algorithms to this case provided that the degree in *each variable* is more than twice the sparsity (this is called a "very sparse" shift in Giesbrecht et al. (2003)). In such cases, we could just choose random fixed values for all variables but one, then perform the univariate sparsest shift algorithm to find the sparsest shift of that variable, and proceed to all other variables.

Finding multivariate shifts in the general case seems more difficult. The precise difficult case here is actually similar to the “Zippel sparse” complexity model, when the sparsity of the sparsest shift is proportional to the partial degrees in each variable. Since the shifts may not even be unique in such situations, it seems that a new approach will be necessary for this problem.

Even more challenging would be allowing multiple shifts, for one or more variables — for example, finding sparse $g_1, \ldots, g_k \in \mathbb{Q}[x]$ and shifts $\alpha_1, \ldots, \alpha_k \in \mathbb{Q}$ such that the unknown polynomial $f(x)$ equals $g_1(x - \alpha_1) + \cdots + g_k(x - \alpha_k)$. The most general problem of this type, which we are very far from solving, might be to compute a minimal-length formula or minimal-size algebraic circuit for an unknown function. We hope that the small step taken here might provide some insight towards this ultimate goal.

Bibliography

John Abbott. Sparse squares of polynomials. *Math. Comp.*, 71(237):407–413, 2002. ISSN 0025-5718.
doi: 10.1090/S0025-5718-00-01294-1. Referenced on pages 106 and 112.

John Abbott and Anna Maria Bigatti. CoCoALib: a C++ library for doing computations in commutative algebra. Online, February 2011.
URL http://cocoa.dima.unige.it/cocoalib. Version 0.9942. Referenced on page 22.

Karl Abrahamson. Time-space tradeoffs for branching programs contrasted with those for straight-line programs. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 402–409, October 1986.
doi: 10.1109/SFCS.1986.58. Referenced on page 57.

Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988. ISSN 0001-0782.
doi: 10.1145/48529.48535. Referenced on page 11.

D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155 – 193, 1979. ISSN 0022-0000.
doi: 10.1016/0022-0000(79)90045-X. Referenced on pages 10, 11 and 14.

V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. The economical construction of the transitive closure of an oriented graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970. ISSN 0002-3264. Referenced on page 17.

Martín Avendaño, Teresa Krick, and Ariel Pacetti. Newton-Hensel interpolation lifting. *Found. Comput. Math.*, 6(1):81–120, 2006. ISSN 1615-3375.
doi: 10.1007/s10208-005-0172-3. Referenced on pages 121 and 146.

Eric Bach. Number-theoretic algorithms. *Annual Review of Computer Science*, 4(1):119–172, 1990.
doi: 10.1146/annurev.cs.04.060190.001003. Referenced on page 150.

Eric Bach and Jeffrey Shallit. *Algorithmic number theory. Vol. 1.* Foundations of Computing Series. MIT Press, Cambridge, MA, 1996. ISBN 0-262-02405-5. Referenced on pages 10 and 127.

Eric Bach and Jonathan Sorenson. Sieve algorithms for perfect power testing. *Algorithmica*, 9:313–328, 1993. ISSN 0178-4617.
doi: 10.1007/BF01228507. Referenced on page 95.

David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4: 23–35, 1990. ISSN 0920-8542.
doi: 10.1007/BF00162341. Referenced on page 37.

R. C. Baker and G. Harman. The Brun-Titchmarsh theorem on average. In *Analytic number theory, Vol. 1 (Allerton Park, IL, 1995)*, volume 138 of *Progr. Math.*, pages 39–103. Birkhäuser Boston, Boston, MA, 1996. Referenced on page 155.

Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew Odlyzko, editor, *Advances in Cryptology âĂŤ CRYPTOâĂŹ 86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Berlin / Heidelberg, 1987.
doi: 10.1007/3-540-47721-7_24. Referenced on page 29.

Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317 – 330, 1983. ISSN 0304-3975.
doi: 10.1016/0304-3975(83)90110-X. Referenced on page 5.

Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 301–309, New York, NY, USA, 1988. ACM. ISBN 0-89791-264-0.
doi: 10.1145/62212.62241. Referenced on pages 24, 119 and 122.

E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Tech. J.*, 46:1853–1859, 1967. ISSN 0005-8580. Referenced on page 7.

Daniel J. Bernstein. Detecting perfect powers in essentially linear time. *Math. Comp.*, 67(223): 1253–1283, 1998. ISSN 0025-5718.
doi: 10.1090/S0025-5718-98-00952-1. Referenced on pages 95 and 114.

Markus Bläser, Moritz Hardt, Richard J. Lipton, and Nisheeth K. Vishnoi. Deterministically testing sparse polynomial identities of unbounded degree. *Information Processing Letters*, 109(3):187 – 192, 2009. ISSN 0020-0190.
doi: 10.1016/j.ipl.2008.09.029. Referenced on pages 126, 139 and 145.

A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 294–301, New York, NY, USA, 1980. ACM. ISBN 0-89791-017-6.
doi: 10.1145/800141.804677. Referenced on page 11.

A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems.* Number 1 in Elsevier Computer Science Library; Theory of Computation Series. American Elsevier Pub. Co., New York, 1975. ISBN 0444001689 0444001565. Referenced on page 141.

Allan Borodin and Prasoon Tiwari. On the decidability of sparse univariate polynomial interpolation. *Computational Complexity*, 1:67–90, 1991. ISSN 1016-3328.
doi: 10.1007/BF01200058. Referenced on page 138.

Robert S. Boyer and J. Strother Moore. MJRTY — a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.
URL http://www.cs.utexas.edu/~moore/best-ideas/mjrty/index.html. Referenced on page 86.

Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Number 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge Univ. Press, November 2010. Referenced on pages 28 and 55.

Peter Bürgisser and Martin Lotz. Lower bounds on the bounded coefficient complexity of bilinear maps. *J. ACM*, 51:464–482, May 2004. ISSN 0004-5411.
doi: 10.1145/990308.990311. Referenced on page 56.

John F. Canny, Erich Kaltofen, and Lakshman Yagati. Solving systems of nonlinear polynomial equations faster. In *Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation*, ISSAC '89, pages 121–128, New York, NY, USA, 1989. ACM. ISBN 0-89791-325-6.
doi: 10.1145/74540.74556. Referenced on page 118.

David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991. ISSN 0001-5903.
doi: 10.1007/BF01178683. Referenced on pages 7, 20, 56 and 70.

David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, 1981. ISSN 0025-5718.
doi: 10.2307/2007663. Referenced on page 7.

Bernard Chazelle. A spectral approach to lower bounds with applications to geometric searching. *SIAM Journal on Computing*, 27(2):545–556, 1998.
doi: 10.1137/S0097539794275665. Referenced on page 11.

Michael Clausen, Andreas Dress, Johannes Grabmeier, and Marek Karpinski. On zero-testing and interpolation of k-sparse multivariate polynomials over finite fields. *Theoretical Computer Science*, 84(2):151–164, 1991. ISSN 0304-3975.
doi: 10.1016/0304-3975(91)90157-W. Referenced on page 118.

Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973. ISSN 0022-0000.
doi: 10.1016/S0022-0000(73)80029-7. Referenced on page 10.

Stephen Arthur Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966. Referenced on pages 17, 68 and 70.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
doi: 10.1090/S0025-5718-1965-0178586-1. Referenced on page 36.

Don Coppersmith and James Davenport. Polynomials whose powers are sparse. *Acta Arith*, 58:79–87, 1991. Referenced on page 106.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, September 2001. ISBN 0262032937. Referenced on page 31.

Richard E. Crandall. *Topics in advanced scientific computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0-387-94473-7. Referenced on pages 35 and 39.

Felipe Cucker, Pascal Koiran, and Steve Smale. A polynomial time algorithm for Diophantine equations in one variable. *J. Symbolic Comput.*, 27(1):21–29, 1999. ISSN 0747-7171.
doi: 10.1006/jsco.1998.0242. Referenced on page 95.

Annie Cuyt and Wen-shin Lee. A new algorithm for sparse interpolation of multivariate polynomials. *Theoretical Computer Science*, 409(2):180–185, 2008. ISSN 0304-3975.
doi: 10.1016/j.tcs.2008.09.002. Symbolic-Numerical Computations. Referenced on page 122.

James H. Davenport and Jacques Carette. The sparsity challenges. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on*, pages 3 –7, September 2009.
doi: 10.1109/SYNASC.2009.62. Referenced on page 7.

Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 499–506, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-047-0.
doi: 10.1145/1374376.1374447. Referenced on page 56.

W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3-1-2 — A computer algebra system for polynomial computations. Online, 2010.
URL http://www.singular.uni-kl.de. Referenced on page 22.

Richard A. Demillo and Richard J. Lipton. A probabilistic remark on algebraic program testing. *Information Processing Letters*, 7(4):193–195, 1978. ISSN 0020-0190.
doi: 10.1016/0020-0190(78)90067-4. Referenced on pages 105 and 118.

Angel Díaz and Erich Kaltofen. On computing greatest common divisors with polynomials given by black boxes for their evaluations. In *Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, ISSAC '95, pages 232–239, New York, NY, USA, 1995. ACM. ISBN 0-89791-699-9.
doi: 10.1145/220346.220375. Referenced on page 118.

Angel Díaz and Erich Kaltofen. FOXBOX: a system for manipulating symbolic objects in black box representation. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, ISSAC '98, pages 30–37, New York, NY, USA, 1998. ACM. ISBN 1-58113-002-3.
doi: 10.1145/281508.281538. Referenced on pages 23 and 118.

Jack Dongarra and Francis Sullivan. Guest editors' introduction to the top 10 algorithms. *Computing in Science Engineering*, 2(1):22–23, January/February 2000. ISSN 1521-9615.
doi: 10.1109/MCISE.2000.814652. Referenced on page 36.

J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LINBOX: A generic library for exact linear algebra. In Arjeh M Cohen, Xiao-Shan Gao, and Nobuki Takayama, editors, *Mathematical software*, Proc. First International Congress of Mathematical Software, pages 40–50. World Scientific, 2002.
doi: 10.1142/9789812777171_0005. Referenced on page 24.

Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35:19:1–19:42, October 2008. ISSN 0098-3500.
doi: 10.1145/1391989.1391992. Referenced on page 29.

Ahmet Duran, B. David Saunders, and Zhendong Wan. Hybrid algorithms for rank of sparse matrices. In *Proc. SIAM Conf. on Appl. Linear Algebra*, 2003. Referenced on page 71.

Mark J. Encarnación. Black-box polynomial resultants. *Information Processing Letters*, 61(4): 201–204, 1997. ISSN 0020-0190.
doi: 10.1016/S0020-0190(97)00016-1. Referenced on page 5.

P. Erdös. On the number of terms of the square of a polynomial. *Nieuw Arch. Wiskunde (2)*, 23:63–65, 1949. Referenced on page 106.

Paul Erdös, Carl Pomerance, and Eric Schmutz. Carmichael's lambda function. *Acta Arith.*, 58 (4):363–385, 1991. ISSN 0065-1036. Referenced on page 151.

Richard Fateman. Draft: Comparing the speed of programs for sparse polynomial multiplication. Online, July 2002.
URL http://www.cs.berkeley.edu/~fateman/algebra.html. Referenced on page 6.

Richard Fateman. Draft: What's it worth to write a short program for polynomial multiplication? Online, December 2008.
URL http://www.cs.berkeley.edu/~fateman/papers/shortprog.pdf. Referenced on page 71.

Michael Filaseta, Andrew Granville, and Andrzej Schinzel. Irreducibility and greatest common divisor algorithms for sparse polynomials. In *Number theory and polynomials*, volume 352 of *London Math. Soc. Lecture Note Ser.*, pages 155–176. Cambridge Univ. Press, Cambridge, 2008.
doi: 10.1017/CBO9780511721274.012. Referenced on page 8.

M. J. Fischer and S. L. Salzberg. Finding a majority among n votes: Solution to problem 81-5. *Journal of Algorithms*, 3(4):376–379, 1982. ISSN 0196-6774.
doi: 10.1016/0196-6774(82)90031-1. Referenced on page 86.

Timothy S. Freeman, Gregory M. Imirzian, Erich Kaltofen, and Lakshman Yagati. Dagwood: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Softw.*, 14:218–240, September 1988. ISSN 0098-3500.
doi: 10.1145/44128.214376. Referenced on pages 5 and 23.

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". Referenced on page 135.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.
doi: 10.1109/SFFCS.1999.814600. Referenced on page 11.

Martin Fürer. Faster integer multiplication. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-631-8.
doi: 10.1145/1250790.1250800. Referenced on pages 17 and 56.

Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science*, 410(27-29):2659–2662, 2009. ISSN 0304-3975.
doi: 10.1016/j.tcs.2009.03.030. Referenced on pages 3, 108, 117, 119, 122, 123, 127, 132, 133, 135 and 146.

Mickaël Gastineau and Jacques Laskar. TRIP: A computer algebra system dedicated to celestial mechanics and perturbation series. *SIGSAM Bull.*, 44:194–197, January 2011. ISSN 0163-5824.
doi: 10.1145/1940475.1940518. Referenced on page 22.

Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, second edition, 2003. ISBN 0521826462. Referenced on pages 17, 100, 101, 119 and 154.

Joachim von zur Gathen, Marek Karpinski, and Igor Shparlinski. Counting curves and their projections. *Computational Complexity*, 6:64–99, 1996. ISSN 1016-3328.
doi: 10.1007/BF01202042. Referenced on page 94.

Mark Giesbrecht and Daniel Roche. Interpolation of shifted-lacunary polynomials. *Computational Complexity*, 19:333–354, 2010. ISSN 1016-3328.
doi: 10.1007/s00037-010-0294-0. Referenced on page 137.

Mark Giesbrecht and Daniel S. Roche. Interpolation of shifted-lacunary polynomials [extended abstract]. In *Proc. Mathematical Aspects of Computer and Information Sciences (MACIS) 2007*, 2007. Referenced on pages 137 and 145.

Mark Giesbrecht and Daniel S. Roche. On lacunary polynomial perfect powers. In *ISSAC '08: Proceedings of the twenty-first international symposium on Symbolic and algebraic computation*, pages 103–110, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-904-3. doi: 10.1145/1390768.1390785. Referenced on pages 94 and 115.

Mark Giesbrecht and Daniel S. Roche. Detecting lacunary perfect powers and computing their roots. *Journal of Symbolic Computation*, to appear, 2011. URL http://arxiv.org/abs/0901.1848. Referenced on page 94.

Mark Giesbrecht, Erich Kaltofen, and Wen-shin Lee. Algorithms for computing sparsest shifts of polynomials in power, chebyshev, and pochhammer bases. *Journal of Symbolic Computation*, 36(3-4):401 – 424, 2003. ISSN 0747-7171. doi: 10.1016/S0747-7171(03)00087-7. ISSAC 2002. Referenced on pages 138, 141, 142, 143, 144, 153 and 155.

Mark Giesbrecht, George Labahn, and Wen-shin Lee. Symbolic-numeric sparse interpolation of multivariate polynomials. *Journal of Symbolic Computation*, 44(8):943 – 959, 2009. ISSN 0747-7171. doi: 10.1016/j.jsc.2008.11.003. Referenced on page 122.

Etienne Grandjean and J. Robson. RAM with compact memory: a realistic and robust model of computation. In E. Börger, H. Büning, M. Richter, and W. Schönfeld, editors, *Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 195–233. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3-540-54487-9_60. Referenced on page 10.

Torbjörn Granlund et al. *GNU Multiple Precision Arithmetic Library, The.* Free Software Foundation, Inc., 4.3.2 edition, January 2010. URL http://gmplib.org/. Referenced on pages 22, 68 and 133.

Andrew Granville and Carl Pomerance. On the least prime in certain arithmetic progressions. *Journal of the London Mathematical Society*, s2-41(2):193–200, April 1990. doi: 10.1112/jlms/s2-41.2.193. Referenced on page 150.

Dima Grigoriev and Marek Karpinski. A zero-test and an interpolation algorithm for the shifted sparse polynomials. In Gérard Cohen, Teo Mora, and Oscar Moreno, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 673 of *Lecture Notes in Computer Science*, pages 162–169. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56686-4_41. Referenced on page 138.

Dima Grigoriev, Marek Karpinski, and Andrew M. Odlyzko. Short proofs for nondivisibility of sparse polynomials under the extended Riemann hypothesis. *Fundam. Inf.*, 28(3-4):297–301, 1996. ISSN 0169-2968. Referenced on page 94.

Dima Yu. Grigoriev and Marek Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in NC. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 166–172, October 1987. doi: 10.1109/SFCS.1987.56. Referenced on pages 108 and 146.

Dima Yu. Grigoriev and Y. N. Lakshman. Algorithms for computing sparse shifts for multivariate polynomials. *Applicable Algebra in Engineering, Communication and Computing*, 11:43–67, 2000. ISSN 0938-1279.
doi: 10.1007/s002000050004. Referenced on page 155.

Dima Yu. Grigoriev, Marek Karpinski, and Michael F. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM Journal on Computing*, 19(6): 1059–1063, 1990.
doi: 10.1137/0219073. Referenced on page 121.

Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*, chapter 2: Finite Field Arithmetic, pages 25–74. Springer Professional Computing. Springer-Verlag, New York, 2004. Referenced on page 28.

William Hart, Fredrick Johansson, and Sebastian Pancratz. *FLINT: Fast Library for Number Theory*, version 2.0.0 edition, January 2011.
URL http://www.flintlib.org/. Referenced on page 22.

David Harvey. zn_poly: a C library for polynomial arithmetic in $\mathbb{Z}/n\mathbb{Z}[x]$. Online, October 2008.
URL http://cims.nyu.edu/~harvey/code/zn_poly/. Version 0.9. Referenced on pages 22 and 66.

David Harvey. A cache-friendly truncated FFT. *Theoretical Computer Science*, 410(27–29): 2649–2658, 2009a. ISSN 0304-3975.
doi: 10.1016/j.tcs.2009.03.014. Referenced on pages 37 and 40.

David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009b. ISSN 0747-7171.
doi: 10.1016/j.jsc.2009.05.004. Referenced on page 9.

David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 325–329, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0150-3.
doi: 10.1145/1837934.1837996. Referenced on pages 35 and 53.

D. R. Heath-Brown. Almost-primes in arithmetic progressions and short intervals. *Mathematical Proceedings of the Cambridge Philosophical Society*, 83(03):357–375, 1978.
doi: 10.1017/S0305004100054657. Referenced on page 150.

D. R. Heath-Brown. Zero-free regions for Dirichlet L-functions, and the least prime in an arithmetic progression. *Proc. London Math. Soc.*, s3-64(2):265–338, March 1992.
doi: 10.1112/plms/s3-64.2.265. Referenced on pages 18 and 150.

Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, October 1984. ISSN 0740-7467.
doi: 10.1109/MASSP.1984.1162257. Referenced on page 36.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963. ISSN 0162-1459.
URL http://www.jstor.org/stable/2282952. Referenced on page 125.

Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, ISSAC '04, pages 290–296, New York, NY, USA, 2004. ACM. ISBN 1-58113-827-X.
doi: 10.1145/1005285.1005327. Referenced on pages 2, 40, 47 and 50.

Joris van der Hoeven. Notes on the truncated Fourier transform. Technical Report 2005-5, Université Paris-Sud, Orsay, France, 2005.
URL http://www.texmacs.org/joris/tft/tft-abs.html. Referenced on page 40.

Gerhard Jaeschke. On strong pseudoprimes to several bases. *Math. Comp.*, 61(204):915–926, 1993.
doi: 10.1090/S0025-5718-1993-1192971-8. Referenced on page 151.

Seyed Mohammad Mahdi Javadi and Michael Monagan. A sparse modular GCD algorithm for polynomials over algebraic function fields. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ISSAC '07, pages 187–194, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-743-8.
doi: 10.1145/1277548.1277575. Referenced on page 118.

Seyed Mohammad Mahdi Javadi and Michael Monagan. On factorization of multivariate polynomials over algebraic number and function fields. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 199–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0.
doi: 10.1145/1576702.1576731. Referenced on page 118.

Seyed Mohammad Mahdi Javadi and Michael Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 160–168, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0067-4.
doi: 10.1145/1837210.1837233. Referenced on page 121.

Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8:63–71, August 1974. ISSN 0163-5824.
doi: 10.1145/1086837.1086847. Referenced on pages 7, 31 and 70.

Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13:1–46, 2004. ISSN 1016-3328.
doi: 10.1007/s00037-004-0182-6. 10.1007/s00037-004-0182-6. Referenced on page 118.

E. Kaltofen. Single-factor Hensel lifting and its application to the straight-line complexity of certain polynomials. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 443–452, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7.
doi: 10.1145/28395.28443. Referenced on page 95.

Erich Kaltofen. Notes on polynomial and rational function interpolation. Unpublished manuscript, 1988. Referenced on page 146.

Erich Kaltofen. Factorization of polynomials given by straight-line programs. In *Randomness and Computation*, pages 375–412. JAI Press, 1989. Referenced on page 5.

Erich Kaltofen and Pascal Koiran. On the complexity of factoring bivariate supersparse (lacunary) polynomials. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 208–215, New York, NY, USA, 2005. ACM. ISBN 1-59593-095-7.
doi: 10.1145/1073884.1073914. Referenced on pages 7, 34 and 95.

Erich Kaltofen and Pascal Koiran. Finding small degree factors of multivariate supersparse (lacunary) polynomials over algebraic number fields. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 162–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-276-3.
doi: 10.1145/1145768.1145798. Referenced on pages 34, 95 and 116.

Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *Journal of Symbolic Computation*, 36(3-4):365–400, 2003. ISSN 0747-7171.
doi: 10.1016/S0747-7171(03)00088-9. ISSAC 2002. Referenced on pages 71, 121 and 138.

Erich Kaltofen and Barry M. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation*, 9(3):301–320, 1990. ISSN 0747-7171.
doi: 10.1016/S0747-7171(08)80015-6. Computational algebraic complexity editorial. Referenced on pages 5, 7 and 118.

Erich Kaltofen and Lakshman Yagati. Improved sparse multivariate polynomial interpolation algorithms. In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 467–474. Springer Berlin / Heidelberg, 1989.
doi: 10.1007/3-540-51084-2_44. Referenced on pages 119 and 121.

Erich Kaltofen and Zhengfeng Yang. On exact and approximate interpolation of sparse rational functions. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ISSAC '07, pages 203–210, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-743-8.
doi: 10.1145/1277548.1277577. Referenced on page 122.

Erich Kaltofen, Y. N. Lakshman, and John-Michael Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '90, pages 135–139, New York, NY, USA, 1990. ACM. ISBN 0-201-54892-5.
doi: 10.1145/96877.96912. Referenced on pages 121 and 146.

Erich Kaltofen, Zhengfeng Yang, and Lihong Zhi. On probabilistic analysis of randomization in hybrid symbolic-numeric algorithms. In *Proceedings of the 2007 international workshop*

on Symbolic-numeric computation, SNC '07, pages 11–17, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-744-5.
doi: 10.1145/1277500.1277503. Referenced on page 122.

Erich Kaltofen, John P. May, Zhengfeng Yang, and Lihong Zhi. Approximate factorization of multivariate polynomials using singular value decomposition. *Journal of Symbolic Computation*, 43(5):359–376, 2008. ISSN 0747-7171.
doi: 10.1016/j.jsc.2007.11.005. Referenced on page 118.

Erich L. Kaltofen. The "seven dwarfs" of symbolic computation. Manuscript prepared for the final report of the 1998-2008 Austrian research project SFB F013 "Numerical and Symbolic Scientific Computing," Peter Paule, director, April 2010a.
URL http://www.math.ncsu.edu/~kaltofen/bibliography/10/Ka10_7dwarfs.pdf. Referenced on page 2.

Erich L. Kaltofen. Fifteen years after DSC and WLSS2: What parallel computations I do today [invited lecture at PASCO 2010]. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 10–17, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0067-4.
doi: 10.1145/1837210.1837213. Referenced on pages 119 and 120.

A. A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 7:595–596, 1963. Referenced on pages 7, 17, 54 and 70.

Marek Karpinski and Igor Shparlinski. On the computational hardness of testing square-freeness of sparse polynomials. In Marc Fossorier, Hideki Imai, Shu Lin, and Alain Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 1719 of *Lecture Notes in Computer Science*, pages 731–731. Springer Berlin / Heidelberg, 1999.
doi: 10.1007/3-540-46796-3_47. 10.1007/3-540-46796-3_47. Referenced on page 95.

Donald E. Knuth. *The art of computer programming, Volume 2: seminumerical algorithms*. Addison-Wesley, Boston, MA, 1981. ISBN 0-201-89684-2. Referenced on pages 8, 17 and 150.

A. N. Kolmogorov and V. A. Uspenskiĭ. On the definition of an algorithm. *Uspehi Mat. Nauk*, 13(4(82)):3–28, 1958. ISSN 0042-1316.
URL http://mi.mathnet.ru/eng/umn7453. Referenced on page 10.

Leopold Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal Für die reine und angewandte Mathematik*, 92:1–122, 1882. Referenced on page 8.

Y. N. Lakshman and B. David Saunders. Sparse shifts for univariate polynomials. *Applicable Algebra in Engineering, Communication and Computing*, 7:351–364, 1996. ISSN 0938-1279.
doi: 10.1007/BF01293594. Referenced on pages 138 and 141.

Susan Landau. Factoring polynomials over algebraic number fields. *SIAM Journal on Computing*, 14(1):184–195, 1985.
doi: 10.1137/0214015. Referenced on page 110.

H. W. Lenstra, Jr. Finding small degree factors of lacunary polynomials. In *Number theory in progress, Vol. 1 (Zakopane-Kościelisko, 1997)*, pages 267–276. de Gruyter, Berlin, 1999. Referenced on pages 7, 24, 34, 95 and 116.

Xin Li, Marc Moreno Maza, Raqeeb Rasheed, and Éric Schost. The modpn library: Bringing fast polynomial arithmetic into MAPLE. *ACM Commun. Comput. Algebra*, 42:172–174, February 2009a. ISSN 1932-2240.
doi: 10.1145/1504347.1504374. Referenced on page 23.

Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: From theory to practice. *Journal of Symbolic Computation*, 44(7):891–907, 2009b. ISSN 0747-7171.
doi: 10.1016/j.jsc.2008.04.019. International Symposium on Symbolic and Algebraic Computation. Referenced on pages 30, 40 and 51.

Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley Publishing Company Advanced Book Program, Reading, MA, 1983. ISBN 0-201-13519-1. Referenced on pages 97 and 98.

U. V. Linnik. On the least prime in an arithmetic progression. II. The Deuring-Heilbronn phenomenon. *Rec. Math. [Mat. Sbornik] N.S.*, 15(57):347–368, 1944. Referenced on page 18.

Roman Maeder. Storage allocation for the Karatsuba integer multiplication algorithm. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 59–65. Springer Berlin / Heidelberg, 1993.
doi: 10.1007/BFb0013168. Referenced on page 55.

Yishay Mansour. Randomized interpolation and approximation of sparse polynomials. *SIAM Journal on Computing*, 24(2):357–368, 1995.
doi: 10.1137/S0097539792239291. Referenced on page 122.

John D. Markel. FFT pruning. *Audio and Electroacoustics, IEEE Transactions on*, 19(4):305–311, December 1971. ISSN 0018-9278.
doi: 10.1109/TAU.1971.1162205. Referenced on page 40.

M. Mignotte. An inequality about factors of polynomials. *Math. Comp.*, 28:1153–1157, 1974. ISSN 0025-5718. Referenced on page 101.

Hiroshi Mikawa. On primes in arithmetic progressions. *Tsukuba Journal of Mathematics*, 25 (1):121–153, June 2001.
URL http://hdl.handle.net/2241/100471. Referenced on pages 149 and 150.

Michael Monagan. In-place arithmetic for polynomials over z_n. In John Fitch, editor, *Design and Implementation of Symbolic Computation Systems*, volume 721 of *Lecture Notes in Computer Science*, pages 22–34. Springer Berlin / Heidelberg, 1993.
doi: 10.1007/3-540-57272-4_21. Referenced on page 28.

Michael Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In Victor Ganzha, Ernst Mayr, and Evgenii Vorozhtsov, editors,

Computer Algebra in Scientific Computing, volume 4770 of *Lecture Notes in Computer Science*, pages 295–315. Springer Berlin / Heidelberg, 2007.
doi: 10.1007/978-3-540-75187-8_23. Referenced on pages 31, 70 and 72.

Michael Monagan and Roman Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 263–270, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0.
doi: 10.1145/1576702.1576739. Referenced on pages 23 and 31.

Michael Monagan and Roman Pearce. Sparse polynomial division using a heap. *Journal of Symbolic Computation*, In Press, Corrected Proof, 2010a. ISSN 0747-7171.
doi: 10.1016/j.jsc.2010.08.014. Referenced on page 23.

Michael Monagan and Roman Pearce. Parallel sparse polynomial division using heaps. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 105–111, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0067-4.
doi: 10.1145/1837210.1837227. Referenced on page 23.

Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170): 519–521, 1985.
doi: 10.2307/2007970. Referenced on page 29.

PARI/GP, version 2.3.5. The PARI Group, Bordeaux, February 2010.
URL http://pari.math.u-bordeaux.fr/. Referenced on page 22.

Philippe Pébay, J. Maurice Rojas, and David C. Thompson. Optimizing n-variate $(n+k)$-nomials for small k. *Theoretical Computer Science*, In Press, Corrected Proof, 2010. ISSN 0304-3975.
doi: 10.1016/j.tcs.2010.11.053. Referenced on page 8.

Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995. ISSN 0166-218X.
doi: 10.1016/0166-218X(93)E0160-Z. Referenced on page 70.

David A. Plaisted. New NP-hard and NP-complete polynomial and integer divisibility problems. *Theoret. Comput. Sci.*, 31(1-2):125–138, 1984. ISSN 0304-3975.
doi: 10.1016/0304-3975(84)90130-0. Referenced on page 94.

David Alan Plaisted. Sparse complex polynomials and polynomial reducibility. *J. Comput. System Sci.*, 14(2):210–221, 1977. ISSN 0022-0000. Referenced on page 94.

J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, 32(143): 918–924, 1978. ISSN 0025-5718.
doi: 10.1090/S0025-5718-1978-0491431-9. Referenced on page 119.

J. M. Pollard. Kangaroos, monopoly and discrete logarithms. *Journal of Cryptology*, 13:437–447, 2000. ISSN 0933-2790.
doi: 10.1007/s001450010010. Referenced on page 119.

Andrew Quick. Some GCD and divisibility problems for sparse polynomials. Master's thesis, University of Toronto, 1986. Referenced on page 94.

Daniel S. Roche. Adaptive polynomial multiplication. In *Proc. Milestones in Computer Algebra (MICA)*, pages 65–72, 2008. Referenced on page 69.

Daniel S. Roche. Space- and time-efficient polynomial multiplication. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 295–302, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0.
doi: 10.1145/1576702.1576743. Referenced on page 53.

Daniel S. Roche. Chunky and equal-spaced polynomial multiplication. *Journal of Symbolic Computation*, In Press, Accepted Manuscript:–, 2010. ISSN 0747-7171.
doi: 10.1016/j.jsc.2010.08.013. Referenced on page 69.

J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Ill. J. Math.*, 6:64–94, 1962.
URL http://projecteuclid.org/euclid.ijm/1255631807. Referenced on pages 101, 110, 123 and 149.

John Savage and Sowmitri Swamy. Space-time tradeoffs for oblivious integer multiplication. In Hermann Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 498–504. Springer Berlin / Heidelberg, 1979.
doi: 10.1007/3-540-09510-1_40. Referenced on page 56.

Nitin Saxena. Progress on polynomial identity testing. *Bull. EATCS*, 99:49–79, 2009. Referenced on pages 5 and 118.

A. Schinzel. On the number of terms of a power of a polynomial. *Acta Arith.*, 49(1):55–70, 1987. ISSN 0065-1036. Referenced on pages 103 and 106.

A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.
doi: 10.1137/0209036. Referenced on pages 10, 17 and 20.

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. ISSN 0010-485X.
doi: 10.1007/BF02242355. Referenced on pages 7, 17, 56 and 70.

Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977. ISSN 0001-5903.
doi: 10.1007/BF00289470. Referenced on pages 56 and 70.

Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter. *Fast algorithms*. Bibliographisches Institut, Mannheim, 1994. ISBN 3-411-16891-9. A multitape Turing machine implementation. Referenced on page 10.

J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701–717, October 1980. ISSN 0004-5411.
doi: 10.1145/322217.322225. Referenced on pages 106 and 118.

Victor Shoup. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation*, 17(5):371–391, 1994. ISSN 0747-7171.
doi: 10.1006/jsco.1994.1025. Referenced on pages 99 and 100.

Victor Shoup. NTL: A Library for doing Number Theory. Online, August 2009.
URL http://www.shop.net/ntl/. Version 5.5.2. Referenced on pages 22, 66, 115 and 133.

Igor E. Shparlinski. Computing Jacobi symbols modulo sparse integers and polynomials and some applications. *Journal of Algorithms*, 36(2):241–252, 2000. ISSN 0196-6774.
doi: 10.1006/jagm.2000.1091. Referenced on page 95.

Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010. Referenced on page 118.

A. J. Sommese and C. W. Wampler. *Numerical solution of polynomial systems arising in engineering and science*. World Scientific, Singapore, 2005. Referenced on page 118.

Andrew J. Sommese, Jan Verschelde, and Charles W. Wampler. Numerical decomposition of the solution sets of polynomial systems into irreducible components. *SIAM Journal on Numerical Analysis*, 38(6):2022–2046, 2001.
doi: 10.1137/S0036142900372549. Referenced on page 118.

Andrew J. Sommese, Jan Verschelde, and Charles W. Wampler. Numerical factorization of multivariate complex polynomials. *Theoretical Computer Science*, 315(2-3):651–669, 2004. ISSN 0304-3975.
doi: 10.1016/j.tcs.2004.01.011. Referenced on page 118.

Henrik V. Sorensen and C. Sidney Burrus. Efficient computation of the DFT with only a subset of input or output points. *Signal Processing, IEEE Transactions on*, 41(3):1184–1200, March 1993. ISSN 1053-587X.
doi: 10.1109/78.205723. Referenced on page 40.

Hans J. Stetter. *Numerical Polynomial Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004. ISBN 0898715571. Referenced on page 118.

David R. Stoutemyer. Which polynomial representation is best? Surprises abound! In *Proc. 1984 Macsyma users' conference*, pages 221–243, Schenectady, New York, 1984. Referenced on page 6.

Emmanuel Thomé. Karatsuba multiplication with temporary space of size $\leq n$. Online, September 2002.
URL http://www.loria.fr/~thome/publis/. Referenced on page 55.

A. L. Toom. The complexity of a scheme of functional elements simulating the multiplication of integers. *Doklady Akademii Nauk SSSR*, 150:496–498, 1963. ISSN 0002-3264. Referenced on pages 17, 68 and 70.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.
doi: 10.1112/plms/s2-42.1.230. Referenced on page 10.

Jr. Wagstaff, Samuel S. Greatest of the least primes in arithmetic progressions having a given modulus. *Math. Comp.*, 33:1073–1080, 1979.
doi: 10.1090/S0025-5718-1979-0528061-7. Referenced on page 151.

André Weil. On some exponential sums. *Proc. Nat. Acad. Sci. U. S. A.*, 34:204–207, 1948. ISSN 0027-8424. Referenced on page 98.

Triantafyllos Xylouris. On Linnik's constant. Technical report, arXiv:0906.2749v1 [math.NT], 2009.
URL http://arxiv.org/abs/0906.2749. Referenced on pages 18 and 150.

Thomas Yan. The geobucket data structure for polynomials. *Journal of Symbolic Computation*, 25(3):285–293, 1998. ISSN 0747-7171.
doi: 10.1006/jsco.1997.0176. Referenced on pages 31 and 70.

David Y. Y. Yun. On square-free decomposition algorithms. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, SYMSAC '76, pages 26–35, New York, NY, USA, 1976. ACM.
doi: 10.1145/800205.806320. Referenced on page 95.

Umberto Zannier. On the number of terms of a composite polynomial. *Acta Arith*, 127(2): 157–167, 2007.
doi: 10.4064/aa127-2-5. Referenced on page 106.

Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward Ng, editor, *Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer Berlin / Heidelberg, 1979.
doi: 10.1007/3-540-09519-5_73. Referenced on pages 7, 105 and 118.

Richard Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375–403, 1990. ISSN 0747-7171.
doi: 10.1016/S0747-7171(08)80018-1. Computational algebraic complexity editorial. Referenced on pages 7, 120 and 121.