HIGH PERFORMANCE SPARSE MULTIVARIATE POLYNOMIALS: FUNDAMENTAL DATA STRUCTURES AND ALGORITHMS

(Spine title: High Performance Multivariate Polynomials)
(Thesis format: Monograph)

by

Alexander Brandt

Graduate Program
in
Computer Science

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

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

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Examiners:

. . . .
Dr. Steven Beauchemin

Supervisor:

. . . .
Dr. Marc Moreno Maza

. . . .
Dr. Robert M. Corless

. . . .
Dr. David Jeffrey

The thesis by

Alexander Brandt

entitled:

High Performance Sparse Multivariate Polynomials: Fundamental Data Structures and Algorithms

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

. . . .
Date

. . . .
Chair of the Thesis Examination Board

Abstract

Polynomials may be represented sparsely in an effort to conserve memory usage and provide a succinct and natural representation. Moreover, polynomials which are themselves sparse – have very few non-zero terms – will have wasted memory and computation time if represented, and operated on, densely. This waste is exacerbated as the number of variables increases. We provide practical implementations of sparse multivariate data structures focused on data locality and cache complexity.

We look to develop high-performance algorithms and implementations of fundamental polynomial operations, using these sparse data structures, such as arithmetic (addition, subtraction, multiplication, and division) and interpolation. We revisit a sparse arithmetic scheme introduced by Johnson in 1974, adapting and optimizing these algorithms for modern computer architectures, with our implementations over the integers and rational numbers vastly outperforming the current wide-spread implementations. We develop a new algorithm for sparse pseudo-division based on the sparse polynomial division algorithm, with very encouraging results. Polynomial interpolation is explored through univariate, dense multivariate, and sparse multivariate methods. Arithmetic and interpolation together form a solid high-performance foundation from which many higher-level and more interesting algorithms can be built.

Keywords: Sparse Polynomials, Polynomial Arithmetic, Polynomial Interpolation, Data Structures, High-Performance, Polynomial Multiplication, Polynomial Division, Pseudo-Division

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In the world of computer algebra and scientific computing, high-performance is paramount. Due to the exact nature of symbolic computations (in fact that is one of the defining characteristics of computer algebra) it is possible to be far more expressive than in numerical methods, leading to more complicated mathematical formulas and theories. But as these formulas, problems, and data sets grow large, the intermediate expressions which arise during the process of solving these problems grow even larger. This *expression swell* is one reason why computer algebra has historically been seen as slow and impractical. Compared to that of numerical analysis, where floating point numbers occupy a fixed amount of space regardless of their value, symbolic computation uses arbitrary precision, allowing numbers to grow and grow. Hence, careful implementation is needed to avoid such drastic slow downs and make symbolic computation able to be practical and useful in solving mathematical and scientific problems.

One such problem we are concerned with is polynomial system solving. Of course, this is an important problem with applications in every scientific discipline. Algorithms for solving these systems typically rely on some *core operation* after some algebraic tricks attempt to reduce the system's complexity. This core operation could be based on Gröbner bases or triangular decompositions. We are motivated by the efforts to obtain an efficient and open source implementation of triangular decompositions using the theory of regular chains [4]. Such algorithms [19] have already been integrated into the computer algebra system MAPLE as part of the REGULARCHAINS library [53]. However, we believe that we can do better.

The *Basic Polynomial Algebra Subprograms* (BPAS) library [2] is an open-source library for high performance polynomial operations, such as arithmetic, real root isolation, and polynomial system solving. It is mainly written in C for performance, with a C++ wrapper interface for portability, object-oriented programming, and end-user usability. Moreover, it makes use of the CILK extension [52] for parallelization and improved performance on multi-core processors. It is within this library that we include our sparse polynomial data structures and high-performance implementations for fundamental operations in support of triangular decompositions and regular chains. The algorithms and implementations presented here are published within the BPAS library.

In particular, we provide highly optimized implementations of sparse multivariate polynomials over the integers and rational numbers. This includes memory-efficient data structures, finely-tuned arithmetic (addition, multiplication, division, pseudo-division), as well as interpolation. The operations of arithmetic and interpolation are absolutely fundamental to any algorithm dealing with polynomials. The fundamental nature of arithmetic should be obvious. Of course one cannot hope to implement any form of high-performance mathematical algorithm without the basic arithmetic also being high-performance. From this standpoint we look to build from the ground up, providing the fastest arithmetic possible so that it can be put to use in higher level algorithms.

The fundamental nature of interpolation is less obvious without considering evaluation-interpolation schemes as part of *modular methods*. To combat the expression swell characteristic of symbolic computations, the approach of modular methods is to solve a large number of simplified problems where the values for the variables are carefully chosen (evaluation) and, using the results of each of these simplified problems, a solution to the original problem is reconstructed (interpolation) [75]. These methods are used extensively throughout computer algebra [32, Chapter 5].

In either case, our optimized, high-performance implementations of sparse polynomial data structures and algorithms provide a solid foundation from which we can build upon to better the algorithms and implementations available in computer algebra and scientific computing.

1.1 Existing Computer Algebra Systems and Software

The ALTRAN system for multivariate rational functions [38], developed in the late sixties at Bell Labs, was one of the first computer algebra systems. It too made use of sparse polynomials in its computations for effective memory usage and performance.

Since that time, many computer algebra systems, proprietary and open source, have been developed. The most notable of which is likely MAPLE. It began in 1982, and since then has found its way into scientific computing, industrial applications, and many university classrooms. Vast amounts of research has gone into developing MAPLE, and, as of 2011, has become a leader in performance because of this [56–59].

This is not to outshine the many other computer algebra systems which are also leaders in their own right. To name a few, there are MATHEMATICA [72], TRIP [31], MAGMA [14], SINGULAR [65], CoCoA [35], and *Symbolic Math Toolbox* of MATLAB. However, only SINGULAR and CoCoA are both free and open-source systems. Other common libraries which are focused on number theory, but provide powerful univariate symbolic computation nonetheless are NTL [71] and FLINT [39]. SymPy [55], a Python library for symbolic computations is actively being developed and excels in usability, a property for which Python is well known.

Maple and Trip are both known to use sparse polynomials in their computations. Interestingly, these are the overwhelmingly highest performing implementations [56–58]. This goes to show, that although recent decades were spent in the computer algebra community developing dense algorithms for polynomial arithmetic [12, 46, 69], all is not lost in the world of sparse polynomials.

With the work presented here on optimized sparse polynomials, we hope to make the BPAS library a leader in performance as well as availability through an open-source codebase.

1.2 Reinventing the Sparse Polynomial Wheel?

Several decades ago, algorithms and implementations of sparse polynomials was a major topic of research. The seminal articles on sparse polynomial arithmetic by Johnson [42], and sparse polynomial interpolation by Zippel [76] are two excellent examples from the seventies. The ALTRAN system of the late sixties [38], made exclusive use of sparse representations for its multivariate rational functions.

Alas, these algorithms and technologies are now nearly 50 years old. In terms of computing technology, they may as well be millennia old. The motivation for sparse algorithms and representations at the time was a result of limited computing resources. The computing memory available then was not even comparable to modern computers. As late as 1980, computer scientists were still using memory chips of a mere 64 megabytes [**hennessy2011computer**]. Today, 64 *giga*bytes is common even in consumer-level computers, let alone high performance server units. As a result of this limited memory, algorithms had been developed to minimize the amount of memory used.

On modern computer architectures, the amount of memory used is much less important. Rather, we are far more concerned with *how* memory is used. This is due to the *memory wall* – "[the] point system performance is totally determined by memory speed; making the processor faster won't affect the wall-clock time to complete an application."[73, p. 21]. The authors of [73] suggest a possible solution: make caches work better, having cache hits occur as frequently as possible. In their point of view, this was an architectural problem for computers.

Nonetheless, it could also be considered an algorithmic problem. We know that the speeds of computer processors and computer memory is diverging exponentially as technologies improve [**hennessy2011computer**]. Therefore, we must be concerned with how we use and traverse memory to avoid latent memory accesses. Computer scientists and programmers need to be aware of memory usage and memory access patterns in order to minimize the impact of waiting for latent memory. The formalization of *cache complexity* [29] is one step toward this. Apart from a usual time complexity analysis of an algorithm, which is only concerned with number of computational steps, cache complexity measures how an algorithm makes use of memory, and therefore also an important indicator of performance given the current *processor-memory gap* [**hennessy2011computer**] (see Section 2.1).

It is with this mindset that we look to implement sparse polynomial data structures and algorithms. We are concerned with performance on modern day architectures, by handling memory effectively, optimizing for cache complexity, and minimizing the impact of the processor-memory gap. Hence, while sparse polynomials have been around for many years, they had been developed using wildly different frames of reference and motivating factors than are no longer applicable.

1.3 Contributions

Throughout all of the work we present here, we are looking to give a fresh implementation of sparse polynomial algorithms for modern day architectures. We look to answer the following questions:

(1) How can the limitations of modern architecture be handled in practical implementations of existing algorithms?
(2) How can we adapt the ideas of classical sparse algorithms to new operations?
(3) How worthwhile is the effort to more finely implement (and make publicly available) existing high-performance algorithms?

We believe these questions have been answered thoroughly over the course of this thesis.

As mentioned in the previous section, our implementations are concerned with efficient memory management, memory traversal, and thus cache complexity. It is with this perspective that we approach our implementations. From Johnson's sparse addition, multiplication and division, to Zippel's sparse interpolation, we make great efforts to optimize memory usage through data structures and practical implementations with low cache complexity.

Using the ideas of Johnson, and the ideas we discovered through the process of implementing the basic polynomial arithmetic functions (addition, subtraction, multiplication, division), we propose a new algorithm for sparse *pseudo-division*. Modeling this algorithm off sparse polynomial division, we implement it effectively, with very promising results.

The good news regarding these results are not limited to our new algorithm. The existing algorithms of Johnson's sparse addition, multiplication, and division are also implemented in an optimal manner. We are not the first to do so. In [56–59] Monagan and Pearce have also realized Johnson's decades old algorithms on modern computers. Their implementation is exclusive to the proprietary software within MAPLE. However, with some of their optimization techniques in publication, we analyze and adjust their optimizations while adding some of our own, in order to fine-tune performance and achieve 50-100% speedup in comparison to their implementation. The story is similar for interpolation where we have developed an implementation of Lagrange interpolation which also out-performs that available in MAPLE. This time, however, we are an order of magnitude faster. Hence, it is indeed worthwhile to spend effort on fine-tuning existing algorithms, as the resulting speed up can be dramatic with careful implementation.

To completely and accurately answer these questions, we organize this thesis as follows. Chapter 2 begins the thesis by provided a background of knowledge for the many concepts touched on throughout later chapters. Since the subject matter of this thesis includes an interesting overlap of algorithms, practical implementations, and mathematical concepts, we provide the necessary details that either a computer scientist or a mathematician may need to appreciate the other side. This includes: computer memory architectures and cache complexity; rings, integral domains, fields, polynomials, and Gröbner bases; a short review of existing sparse polynomial techniques; and, a formalization of interpolation and some classical methods.

Chapter 3 discusses the effective encoding and data structures required to represent polynomials as effectively as possible on modern computers, while making them favourable for use in our algorithms. These fundamental algorithms are presented in Chapter 4. There, algorithms, implementations, optimization techniques, and experimentation is presented for polynomial addition, subtraction, multiplication, division, and pseudo-division.

Lastly, Chapter 5, discusses the problem of polynomial interpolation in the cases of univariate, dense multivariate, and sparse multivariate. Here, we also consider some limitations of symbolic computing, offering a possible solution using numerical methods.

Chapter 2

Background

Computer algebra lies in an interesting overlap of computer science and mathematics, drawing on both areas to fulfill its needs. In this background section, we introduce topics necessary to discuss our algorithms and implementations of polynomials. This includes some mathematics which computer scientists may be unfamiliar with, as well as some computer science aspects mathematicians may be unfamiliar with.

We discuss computer organization and memory hierarchy (Section 2.1); the understanding of memory is important for developing high-performance algorithms. We also discuss some fundamentals of algebra, such as rings and operations defined within them (Section 2.2). A more specific presentation of polynomials, their representations, and a short history of sparse polynomial algorithms is given in Sections 2.3 and 2.4. Lastly, we present the problems of interpolation and curve fitting as well as some their classical solutions (Section 2.5).

2.1 Memory, Cache, and Locality

The structure of computer memory is no longer just important to the computer architect. Any programmer should have at least a basic understanding of its workings in order to produce good quality code. The reason for this is (somewhat) recent advancements in technology. The speed of processors has eclipsed the speed of computer memory many times over [**hennessy2011computer**]. This difference is called the *processor-memory gap*. Decades ago, the speed of processors and memory were comparable, but the amount of memory available was limited. The problem now is different. There is (relatively) ample amounts of memory but it is slow to access compared to the speed of the processor. Because of this, programmers must change the way that they program in order make full use of the hardware.

Without considering a computer's memory architecture, a program will almost surely underutilise the processor, as it sits idle waiting for latent memory accesses. Computer architects have attempted to close the processor-memory gap by creating a *memory hierarchy.* That is, a hierarchy of different forms of memory with increasing speeds but decreasing size. This forms the classical memory pyramid, with L1 cache at the top, followed by L2 and L3 cache, dynamic main memory, and finally hard disks (virtual memory).

L1 cache is the fastest memory, but has the smallest capacity. Ideally, for the best performance, we would want all of our data stored in L1 cache for the duration of a program or algorithm. But due to the small capacity of L1 cache this is generally impossible. Not all hope is lost, as one can make use of the *principle of locality* — programs tend to reuse data and instructions which they have used recently [**hennessy2011computer**]. Therefore we have *data locality* and *instruction locality*; the differentiation between the two is sometimes useful.

The principle of locality was put into the design of the memory hierarchy. Therefore, the choice of which data is stored in cache over main memory is a result of simply caching the data which has been most recently used. In practice, locality is implemented in a cache structure by *evicting* the least recently used item once a cache has reached capacity and attempts to load an additional item. Programmers must make use of this fact, by programming in such a way that exploits locality, and does not destroy it. There are two types of locality and we must be concerned with both. *Temporal locality*, which says that the most recently accessed items will likely be accessed again soon, and *spatial locality*, which says that the items adjacent to the most recently accessed items are also likely to be accessed soon. It is important to use these facts in programming. If an algorithm (or data structure) is implemented effectively, memory access time is greatly reduced by exploiting caching and locality [**hennessy2011computer**]. This is evident in our discussion of polynomial data structures (in particular, see Figures 3.4 and 3.5).

2.1.1 Cache Performance and Cache Complexity

Patterson and Hennessy [**hennessy2011computer**] are keen to highlight that a good cache architecture does necessarily give rise to good algorithms. Different algorithms use cache differently and the performance of one algorithm on one memory architecture does not imply the performance of another algorithm. Cache performance is a tricky thing to measure.

Cache performance is generally characterized by *cache hits* and *cache misses.* A hit occurs when an item being accessed is already contained in the cache. A miss occurs when an item being accessed is not contained in the cache. Consequently, then it must exist in some lower level of memory and be loaded into higher level memory. The *miss penalty* is a quantity describing the cost of waiting for latent memory accesses to lower levels of memory. Since caches are built up by many layers working together, this penalty can be difficult to quantify precisely. The penalty depends on the item being accessed and how many layers away it is stored from the top-most cache. It could be in L2 cache, where the penalty is relatively small, or it could be stored in a page of virtual memory, which is much more costly. See [**hennessy2011computer**] for details.

To avoid these architecture-dependent details of memory hierarchies and miss penalties, an *idealized cache model* [29] has been formulated. This model assumes a single cache with a single backing, arbitrarily large, main memory. In this model, the cache is broken in *cache lines*, that is, the smallest unit of data that can move in and out of cache. One cache line always consists of sequential memory addresses. It is assumed that the cache can fit Z memory words and each line holds L words of that memory. Therefore, a cache has $^{Z}/_{L}$ lines. Note a memory word is a fixed constant for a particular architecture, usually 4 or 8 bytes. This constant factor is therefore usually ignored, particularly in asymptotic analysis.

Using this model, it is possible not only to derive the normal time complexity of an algorithm, but also *cache complexity.* Cache complexity is a way to measure and characterize and algorithm's performance with respect to cache misses. It is parameterized by input size (n), Z, and L. It is also common to ignore Z and L in the analysis as they can be be implied by context, or could become simple constants in the complexity analysis which get removed by asymptotics (*big-Oh* notation).

Just as with time complexity, one wishes to minimize cache complexity in order to improve the theoretical (and indeed practical) performance of an algorithm. We have explained how caches work in reality, maintaining the most recently used items in cache. The ideal cache model works very similarly, but chooses to evict items which are next referenced furthest in the future, with items which are never referenced again being considered as the furthest possible in the future.

By this model, cache complexity can be reduced by exploiting locality as much as possible. The better locality that a program possesses, the better its cache performance. It is rather simple in that regard. But other aspects also affect cache performance. The size of data items plays an important role. Since the cache has a limited size (Z), if the individual data items being accessed require fewer bytes to store, than this will also improve cache complexity without altering locality. So, in summary, the two basic ways to improve cache complexity are:

(1) Exploit (data) locality to ensure memory is accessed sequentially or in an adjacent manner, and
(2) Reduce the amount of memory required for each data item so that more items can fit in cache at once. [1]

It is clear that, given the processor-memory gap, practical implementations of any algorithms must be aware of cache performance and cache complexity. Naturally, our

[1] The same idea can technically be used for instructions as well, but they generally have fixed sizes depending on the processor's instruction set architecture. Nonetheless, one can reduce the number of instructions used, say within a loop, to obtain a similar effect.

algorithms are conscious of their memory usage and data locality in order to achieve high performance. The implementations details of these are explained in Sections 4.2.1, 4.3.1, 4.4.1, and 5.1.

For more details on computer architecture, memory, cache, and performance, Patterson and Hennessy's *Computer Architecture* provides plenty [**hennessy2011computer**]. For a more detailed description of the ideal cache model and cache complexity, [29] and [66] are the defining works.

2.2 Algebra

This section is intended to introduce those unfamiliar with the mathematical technicalities of algebra, such as the average computer scientist, to a simple selection of concepts needed for our discussion of polynomial arithmetic and interpolation. This includes, rings, integrals domains, fields, the specifics of polynomials, and polynomials as rings. The appendix of *Modern Computer Algebra* [32] provides a useful overview and additional details of all of these concepts. Here, we present only the ones necessary for our purposes.

2.2.1 The Many Flavours of Rings

A commutative *ring* (with identity) is a set R endowed with two binary operations, denoted $+$ and $\times$. These operations need not be the usual addition and multiplication, but they must satisfy the following conditions:

(1) R is a commutative *group* under $+$ with identity 0,
  (i) *Associative*: $\forall\, a, b, c \in R$, $(a + b) + c = a + (b + c)$,
  (ii) *Identity*: $\exists\, 0 \in R \;\; \forall\, a \in R$, $a + 0 = a$,
  (iii) *Inverse*: $\forall\, a \in R \;\; \exists\, a^{-1} \in R$, $a + a^{-1} = 0$, and
  (iv) *Commutative*: $\forall\, a, b \in R$, $a + b = b + a$;
(2) $\times$ is associative and commutative;
(3) R has identity 1 for $\times$; and
(4) $\times$ is distributive over $+$, $\forall\, a, b, c \in R \;\, a \times (b + c) = (a \times b) + (a \times c)$,
and $(b + c) \times a = (b \times a) + (c \times a)$.

We assume that all rings are commutative and with a multiplicative identity unless explicitly stated. Such commutative rings are still quite general. Rings can be extended in many ways to obtain different specializations ("flavours") with different properties.

An *integral domain*, $\mathbb{D}$, (sometimes simply *domain*) is a ring with the added property:

(1) the only *zero divisor* in $\mathbb{D}$ is 0.

A zero divisor is an non-zero element $a \in R$ where, for some $b \neq 0 \in R$, we have $a \times b = 0$. Thus, integral domains are suitable for looking at divisibility and exact division, without needing to worry[2] about zero divisors. A *unit* (or *invertible element*) is an element $a \in R$ with a *multiplicative inverse*. That is, there exists $b \in R$ such that we have $ab = 1$.

Extending divisibility slightly with the notion of a *greatest common divisor* (GCD) then we get a *GCD domain* — an integral domain where any two elements have a GCD between them. If R is a commutative ring with a multiplicative identity and if a, b, d are elements of R, we say that d is a *common divisor* of a and b if d divides both a and b; furthermore, we say that d is a *greatest common divisor* of a and b if any common divisor of a and b divides d as well.

The notion of a GCD domain is commonly used in computer algebra but it is rarely discussed in algebra textbooks, where the related (but not equivalent) concept is that of a *unique factorization domain (UFD).* An integral domain U is a UFD whenever every non-zero element of U writes as a product of irreducible elements, this factorization being unique up to the ordering of those irreducible elements and up to an invertible factor; for more details see `https://en.wikipedia.org/wiki/Unique_factorization_domain`. Clearly, any two non-zero elements of a UFD have a GCD but the existence of an algorithm for computing such a GCD requires additional properties.

A fundamental example of UFDs where GCDs can be effectively computed are Euclidean domains. An integral domain $\mathbb{D}$ is an *Euclidean domain* whenever there exists a function $|\cdot|$ mapping every non-zero element of $\mathbb{D}$ to a non-negative integer such that for all $a, b \in \mathbb{D}$, with $b \neq 0$, there exists $(q, r) \in \mathbb{D} \times \mathbb{D}$ such that we have $a = qb + r$ and either $r = 0$ or $|r| < |b|$ holds; for a such pair (q, r) the elements q and r are called *quotient* and *remainder* in the *Euclidean division* of a by b. The pair (q, r) is not necessarily unique (for instance in the case of the ring $\mathbb{Z}$ of the integer numbers) and additional properties may be required to make it unique (such as $r \geq 0$ in the case of $\mathbb{Z}$).

A *field*, $\mathbb{K}$, is an integral domain in which every non-zero element is a unit. This is a powerful property and it means that every element is divisible by every non-zero element in $\mathbb{K}$. Equivalently, all divisions result in a 0 remainder.

A fundamental example of field construction is the field of fractions of an integral domain. This is a natural adaptation of the construction of the field $\mathbb{Q}$ of rational numbers from the ring $\mathbb{Z}$ of integer numbers, to the case of an arbitrary integral domain $\mathbb{D}$. For details, see `https://en.wikipedia.org/wiki/Field_of_fractions`.

All of these types of rings build upon the previous. Indeed, they form a strict class inclusion:

$$\text{ring} \supset \text{integral domain} \supset \text{GCD domain} \supset \text{UFD} \supset \text{Euclidean domain} \supset \text{field}$$

[2] Indeed, if R is an integral domain and a, b, q, q' are elements of R such that $b \neq 0$ and $a = b\,q = b\,q'$ both hold, then we have $q = q'$; in other words, if b divides a the quotient of a by b is uniquely defined.

This inclusion is useful when we wish to speak generically about one type of ring. It then always applies to every subset of that type. This is evident beginning with *polynomial rings*.

2.2.2 Polynomials: Rings, Definitions, and Notations

A polynomial, as most know it, is a mathematical function in some variables, which is a linear combination of multiplicative combinations of those variables. For example, $p(x, y) = 5x^3y^2 + 3xy + 4x + 1$. The multiplicative combinations are the sub-expressions x^3y^2, xy, x, and $1 = x^0y^0$. But, polynomials are far more sophisticated than that.

From the previous example, you can see there are essentially two parts which make up each *term* of a polynomial, the numerical *coefficient* and the multiplicative combination of the variables. This multiplicative combination is called a *monomial.* We say that the coefficients belong to some ring, R, and that the polynomial is formed over that base ring.

Polynomials themselves form rings, as one can add and multiply polynomials together (see Section 2.2.3). Hence, we say *polynomials over* R to mean a ring of polynomials whose coefficients belong to R. However, we must also distinguish between different classes of polynomials over the same base ring. This is done by specifying the *variables* (indeterminates) of the polynomials. Hence, our example polynomial $p(x, y)$ would be a polynomial over $\mathbb{Z}$ with variables x and y. The ring formed by such polynomials can be denoted by $\mathbb{Z}[x, y]$.

Generally, a polynomial ring in the variables $x_1, \ldots, x_v$ over the base ring R, is denoted by $R[x_1, \ldots, x_v]$. When $v = 1$ we say $R[x_1] = R[x]$ are the *univariate* polynomials over R. When $v > 1$ we say $R[x_1, \ldots, x_v]$ are the *multivariate* polynomials over R, where v should be implied by context, chosen explicitly, or left as a general parameter.

Since R can be any ring, and polynomials themselves form rings, then it is natural to define polynomials recursively as well. One can view a bi-variate polynomial in $R[x, y]$ as being a polynomial over $R[x]$ with variable y (or equivalently, as being a polynomial over $R[y]$ with variable x). To be explicit, a recursive view can be denoted as $R[x][y]$ to imply the ring is $R[x]$ and y the variable.

Moreover, just as rings have different flavours, so too do polynomial rings. The characterization of a polynomial ring depends on the properties of its base ring (and the number of variables it has, depending on it viewing it recursively or not). For univariate polynomials over R, if R is a ring then $R[x]$ is a ring. If R is an integral domain then $R[x]$ is an integral domain. If R is a UFD then $R[x]$ is a UFD: this is Gauss's theorem, [32, Theorem 6.8]. However, this is where the similarities end. If R is a Euclidean domain then $R[x]$ is only a UFD, and division with remainder may be lost. However, we note that if a divisor is *monic* — its leading coefficient is 1 — or, more generally, its leading coefficient is unit in R, then division with remainder *is* possible. This implies that if R is a field then $R[x]$ is a Euclidean domain; in a field every non-zero element is a unit.

Using these rules for univariate polynomials, and the recursive definition of a multivariate polynomial, we obtain the following for $v \geq 2$:

(1) $R[x_1, \ldots, x_v]$ is a ring if R is a ring,
(2) $R[x_1, \ldots, x_v]$ is an integral domain if R is an integral domain,
(3) $R[x_1, \ldots, x_v]$ is a UFD if R is a UFD,
(4) $R[x_1, \ldots, x_v]$ is a UFD if R is a Euclidean domain.

After generalizing polynomials to their rings, we look at specific polynomials and define some aspects of their internal structure. For a given non-zero polynomial $p \in R[x_1, \ldots, x_v]$ we have the following:

(1) the *leading term* of p, $lt(p)$, is the first non-zero term of p, coefficient and monomial;
(2) the *leading monomial* of p, $lm(p)$, is the monomial of the leading term;
(3) the *leading coefficient* of p, $lc(p)$, is the coefficient of the leading term;
(5) the *(total) degree* of p, $deg(p)$, is the is the maximal sum of exponents of a single non-zero term of p;[3]
(5) the *partial degree* of p with respect to x_i, $deg(p, x_i)$, is the maximum exponent of x_i in any non-zero term of p;
(6) the *main variable* of p, $mvar(p)$, is the variable of highest order appearing in p whose partial degree is positive;
(7) the *main degree* of p, $mdeg(p)$, is the degree of the main variable of p, $deg(p, mvar(p))$.

But what do we mean by "first" term or "highest order" variable? Of course, there must be a defined ordering for something to be first. There are various *monomial orderings* (equivalently, *term orderings*) which are used to sort the terms in a polynomial. A monomial ordering is any *total order* that is is compatible with monomial multiplications [30, Section 3.1]. The ordering $\leq_m$ is a monomial ordering if for all monomials m_1, m_2, m_3 the following properties hold:

(1) $m_1 \leq_m m_2$ and $m_2 \leq_m m_1$ imply $m_1 = m_2$,
(2) $m_1 \leq_m m_2$ and $m_2 \leq_m m_3$ imply $m_1 \leq_m m_3$,
(3) $m_1 \leq_m m_1$ holds,
(4) either $m_1 \leq_m m_2$ or $m_2 \leq_m m_1$ holds,
(5) $1 \leq_m m_1$ holds,
(6) $m_1 \leq_m m_2$ implies $m_3 m_1 \leq_m m_3 m_2$.

Properties (1), (2), (3) and (4) are *antisymmetry*, *transitivity*, *reflexivity* and *totality*.

Two common orderings are *lexicographical* and *degree lexicographical*. Both begin by choosing an ordering of the variables themselves. Throughout our discussion we will assume $x_1 > x_2 > \cdots > x_v$. Next, let us denote a monomial $x_1^{e_1} x_2^{e_2} \ldots x_v^{e_v}$ as simply as a

[3] We use the convention that the degree of 0 is $-\infty$

sequence of its exponents, $(e_1, e_2, \ldots, e_v)$. Let $a = (a_1, a_2, \ldots, a_v)$ and $b = (b_1, b_2, \ldots, b_v)$ be two monomials. Then, we have:

$$\text{Lexicographical:} \qquad a \leq_{lex} b \iff \begin{cases} a_i < b_i, \text{ for some } i, \\ a_j = b_j, \text{ for all } j < i \end{cases}$$

$$\text{Degree Lexicographical:} \quad a \leq_{deglex} b \iff \begin{cases} deg(a) < deg(b), \text{ or} \\ deg(a) = deg(b), \text{ and } a <_{lex} b. \end{cases}$$

For bivariate monomials, lexicographical ordering looks like:

$$x^n y^n > x^{n-1} y^n > \cdots > x y^n > \cdots > x > y^n > y^{n-1} > \cdots > y > 1.$$

For bivariate monomials, degree lexicographical ordering looks like:

$$x^n y^n > x^n y^{n-1} > x^{n-1} y^n > \cdots > x^2 y > x y^2 > x^2 > xy > y^2 > x > y > 1.$$

For our purposes, we use lexicographical ordering. It offers particular computational advantages for our implementation (see Section 3.1). Since we fix this throughout, let us simply notation, using $\leq$ to mean $\leq_{lex}$ when comparing monomials or terms.

Speaking of term orders, it is also worthwhile to discuss the divisibility of two monomials. Let $m_1 = x_1^{e_{11}} \ldots x_v^{e_{v1}}$ and $m_2 = x_1^{e_{12}} \ldots x_v^{e_{v2}}$ be monomials. m_1 divides m_2, denoted by $m_1 | m_2$, if $e_{i1} \leq e_{i2}$, for $1 \leq i \leq v$. Also, $m_1 | m_2 \implies m_1 \leq_m m_2$ for any monomial ordering, $\leq_m$. From this, we get the divisibility of polynomial terms. For two polynomial terms, $t_1 = c_1 m_1$, $t_2 = c_2 m_2$, $t_1 | t_2$ if and only if $c_1 | c_2$ and $m_1 | m_2$. This notion of divisibility will play a role in discussing Gröbner bases (Section 2.2.4).

For a polynomial $p \in R[x_1, \ldots, x_v]$ there are several different notations which we may use to define it, depending on our needs. Where the ring and variables are fixed at the beginning of a discussion, such as the statement at the beginning of this paragraph, then we can use simply $p,\ q,\ f,\ g,\ a,\ b,$ *etc.* Where we want to be explicit about the number of variables, or if the number of variables changes throughout a discussion, we may use $p(x_1, \ldots, x_v)$, $q(x_1, x_2, x_3)$, *etc.* When we are interested in particular terms of a polynomial, we may write it in summation notation:

$$a \;=\; \sum_{i=1}^{n_a} a_i x_1^{e_{1i}} \ldots x_v^{e_{vi}} \;=\; \sum_{i=1}^{n_a} a_i X^{\alpha_i} \;=\; \sum_{i=1}^{n_a} A_i$$

$a_i \in R$ is the i^{th} coefficient, and $e_{1i}, \ldots, e_{vi}$ are exponents of the i^{th} monomial. To simplify notation, often a multivariate monomial will be written as X^{α_i}, where a capital letter X denotes the sequence of variables $x_1, \ldots, x_v$ and $\alpha_i = e_{1i}, \ldots, e_{vi}$ is a v-tuple or multi-index of exponents for monomial i, often called an *exponent vector*. It can also be convenient to talk about a term as a whole. Thus we use A_i to denote the term $a_i X^{\alpha_i}$. In this summation notation, the polynomial a has n_a terms (or simply n if there is only one polynomial to discuss).

In this summation notation terms are sorted decreasingly according to lexicographical ordering. While the sort is important, the particular ordering used is not so much. The sorting is important for defining "leading" or "first", but it is also crucial for obtaining a *canonical representation* of a polynomial. That is, a unique representation such that if two representations are equal then the object they represent must also be equal. Such a representation is computationally important in order to efficiently perform operations such as degree, leading term, and equality testing.

There are two strategies for discussing the terms of a polynomial. Either *dense* or *sparse*. Briefly, a dense representation includes terms whose coefficients are zero, while a sparse representation does not. Unless explicitly stated, we assume a sparse representation, so that all a_i are *non-zero*. Hence, n_a is the number of non-zero terms only. Section 2.3 provides more details on different polynomial representations, comparing and contrasting them.

In general, we use lowercase Latin letters to denote polynomials, lowercase Latin letters with subscripts for coefficients, and corresponding lowercase Greek letters with subscripts for exponent vectors. Capital Latin letters with subscripts represent polynomial terms.

2.2.3 Arithmetic in a Polynomial Ring

For a ring to be a ring, it must have two binary operators, +, and ×. For a polynomial ring these are called polynomial addition and polynomial multiplication, respectively. The operators are essentially the same for any polynomial ring, where polynomial addition (multiplication) relies on the addition operator (multiplication operator) of the base ring.

The addition of two polynomials is rather simple. For every *like-term* between the two polynomials, add their coefficients together and put this term in the sum, otherwise, put the same term in the sum as appears in the operands. Like-terms are terms which share the same monomial. Written more mathematically:

Definition 2.1 (Polynomial Addition)

Let $a, b \in R[x_1, \ldots, x_v]$, $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i}$, $b = \sum_{i=1}^{n_b} b_i X^{\beta_i}$ be *dense* polynomials, such that we include zero coefficients. Further, let $n = \max\{n_a, n_b\}$, and prepend zeros to the polynomial which was smaller to make them equal length. Then:

$$a \; + \; b \; = \; \sum_{i=1}^{n} (a_i + b_i) X^{\alpha_i}$$

Note that polynomial subtraction is essentially the same, simply subtracting coefficients instead of adding them.

The multiplication of two polynomials requires polynomial addition, and is essentially the sum of repeated distributions. Whether the polynomials are represented sparsely or densely, this scheme will work fine.

Definition 2.2 (Polynomial Multiplication)

Let $a, b \in R[x_1, \ldots, x_v],\ a = \sum_{i=1}^{n_a} a_i X^{\alpha_i},\ b = \sum_{i=1}^{n_b} b_i X^{\beta_i}$ then:

$$a \ \times \ b \ = \ \sum_{i=1}^{n_a} \left(\sum_{j=1}^{n_b} (a_i \times b_j) X^{\alpha_i + \beta_j} \right)$$

The outer sum works as polynomial addition while the inner sum works more like building up terms of a polynomial — for a fixed i and varying j, $\alpha_i + \beta_j$ are all unique. Note that $\alpha_i + \beta_j$ is a component-wise summation as they are both multi-indices.

For division, it is more tricky. Finding an explicit formula for the quotient and remainder of two polynomials is not easy, but, an algorithm can easily define the process.

Let $\mathbb{D}$ be an integral domain and let $a, b \in \mathbb{D}[x]$ with $b \neq 0$ and $\mathrm{lc}(b)$ be a unit in $\mathbb{D}$. Let $\mathbb{K}$ be the field of fractions of $\mathbb{D}$. Viewing a, b as polynomials in $\mathbb{K}[x]$ and since $\mathbb{K}[x]$ is a Euclidean domain, any pair quotient-remainder (q, r) in the Euclidean division of a by b satisfy $a \ = \ bq + r$ together with $r = 0$ or $\deg(r) < \deg(b)$. It is not hard to verify that (q, r) is unique and that Algorithm 2.1 computes it. Moreover, Algorithm 2.1 shows that the coefficients of q and r are actually in $\mathbb{D}$. Therefore, the division with remainder of a by b, as polynomials in $\mathbb{D}[x]$, is well-defined.

Notice that we require the leading coefficient of the divisor to be a unit in $\mathbb{D}$. When this is not the case, the division over the base ring elements, $lc(c)/lc(b)$ is not usually defined (unless $lc(b)$ exactly divides every coefficient in a, in which case we need not require $lc(b)$ to be a unit). Moreover, since polynomials over integral domains and with positive degree are never units in their ring, then the polynomial division defined is essentially only for univariate polynomials. Multivariate division is much more interesting. It comes about as a consequence of *Gröbner bases.* Therefore, we leave its discussion to Section 2.2.4 regarding Gröbner bases.

For now, Algorithm 2.1 shows univariate polynomial division using a dense representation for ease of notation (notice summations are indexed from 0). Sparse division is discussed thoroughly in Sections 2.4 and 4.3.

Algorithm 2.1 polynomialDivision(a,b)
$a, b \in \mathbb{D}[x]$, $a = \sum_{i=0}^{n_a} a_i x^{\alpha_i}$, $b = \sum_{i=0}^{n_b} b_i x^{\beta_i}$, $lc(b)$ is a unit in R; returns $q, r \in \mathbb{D}[x]$ where q is quotient, r remainder, and $a = qb + r$, $deg(r) < deg(b)$.

```
r := a
for $i = 0$ to $n_a - n_b$ do
    if $deg(r) = n_a - i$ then
        $q_i := lc(r)/lc(b)$
        $r := r - q_i x^{(n_a - n_b + i)} b$
    else
        $q_i := 0$
return $q = \sum_{i=0}^{n_a - n_b} q_i x^{(n_a - n_b + i)}$, $r$
```

When division is not possible due to the limitations of the base ring, such as when the divisor is not monic, there is still an option to perform *pseudo-division*.

Pseudo-division

Pseudo-division is a generalization of the idea of division with remainder on polynomials. The idea behind pseudo-division is to ensure that polynomial division occurs, if it can occur, without worrying about the restrictions of the base ring. In pseudo-division the division by the leading coefficient of the divisor is avoided entirely. Hence, neither do we require the divisor to be monic, the leading coefficient of the divisor to be a unit, nor that the polynomials be defined over an integral domain. Pseudo-division is defined for polynomials over any base ring [48]. Hence, this operation (while essentially univariate) can be defined for multivariate polynomials when they are viewed recursively.

Definition 2.3 (Pseudo-Division)

Let $a, b, q, r \in R[x], b \neq 0$, $lc(b) = h$, $\deg(a) \geq \deg(b)$. Then q is the pseudo-quotient and r the pseudo-remainder, satisfying the equation

$$h^{deg(a)-deg(b)+1} a \ = \ bq + r, \ deg(r) < deg(b).$$

Notice that the exponent on h is equivalent to the maximum number of *division steps* that can occur. That is, the number of times one might actually perform a division in classical polynomial division. Therefore, this ensures the "division"[4] of coefficients can always occur. A classical algorithm for pseudo-division is shown in Algorithm 2.2. Notice that this algorithm is essentially the same as Algorithm 2.1 with the minor addition of multiplying by h at each division step.

[4] In implementation, one avoids multiplication by h and division entirely by simply ignoring both all together. By construction, they cancel each other out anyways.

Algorithm 2.2 naïvePseudoDivision(a,b)

$a, b \in R[x_2, \ldots x_v][x_1] = R[x]$, $\deg(b) > 0$;
return $q, r \in R[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$.

```
$q$:=0; $r$:=$a$
$\ell$:=0; $h$:=lc($b$)
while $\deg(r) \geq \deg(b)$ do
    $k := \deg(r) - \deg(b)$
    $q := hq + \mathrm{lc}(r)x^k$
    $r := hr - \mathrm{lc}(r)x^k b$
    $\ell := \ell + 1$
return $(q, r, \ell)$
```

We also note that one can define a *lazy* pseudo-division in the sense that the exponent on h is not exactly $deg(a) - deg(b) + 1$. Rather, the exponent is equal to the precise number of times one would need to multiply by h to make the division work. Of course, this has many practical benefits in implementation. Pseudo-division is discussed further in Section 4.4 with its practical implementation discussed in Section 4.4.1.

For more details and algorithms regarding polynomial arithmetic, see [48, Section 4.6] and [Sections 2, 6.12, and 25][32]

2.2.4 Gröbner Bases, Ideals, and Reduction

The subject of Gröbner bases is a rich area within algebraic geometry, finding many theoretical and practical applications. It is concerned with, among many others, ideals. We refer the reader to [30] for a rather succinct description of Gröbner bases, and [6] for a more detailed description. A survey of the many applications of Gröbner bases can be found in [17].

We begin by defining *ideals.* An ideal I is a subset of a ring R with the properties:

(1) $\forall\, a, b \in I,\ a + b \in I$
(2) $\forall\, a \in I,\ r \in R,\ ar \in I$

If property (2) is not commutative, we can call I a right (or left, depending on which side r appears in (2)) ideal of R.

An ideal can be *generated* by a set of elements $a_1, \ldots, a_n \in R$, denoted by:

$$\langle a_1, \ldots, a_n \rangle = \{a_1 r_1 + \ldots a_n r_n \mid r_1, \ldots, r_n \in R\}$$

and it is said that $a_1, \ldots, a_n$ form the *basis* of the ideal or is the *generating set* of the ideal. If $A = \{a_1, \ldots, a_n\}$, a useful short-hand is to denote $\langle a_1, \ldots, a_n \rangle = \langle A \rangle$. It is worth noting that the same ideal can have many different generating sets.

A natural problem arises from the discussion of ideals. Does a particular ring element belong to a particular ideal? This is the *ideal membership problem.*

Problem 2.1 (Ideal Membership Problem)

$$\text{For an ideal } I \subseteq R \text{ and } f \in R, \text{ is } f \in I?$$

Gröbner bases yield a theoretically and computationally effective way of solving this problem. Simply, a Gröbner basis is special generating set of an ideal by which the ideal membership problem is easily solved. In order to formalize the solution to the ideal membership problem we must discuss *reduction*.

For the remainder of our discussion, we consider polynomials in the ring $\mathbb{K}[x_1, \ldots, x_v]$ where $\mathbb{K}$ is a field. Gröbner bases are usually discussed in this context as all theorems and results hold in the case of a field. However, we note that it is still possible to work with Gröbner bases for multivariate polynomials over rings [1, Chapter 4]. Further, we must also fix some term order. We will see why this important. The particular one used is not of great importance, but it should remain fixed throughout. We use lexicographical ordering (see Section 2.2.2).

Reduction is simply a generalization of polynomial division. Let $f, g, h, r \in \mathbb{K}[x_1, \ldots, x_v]$ be multivariate polynomials. It is said that f reduces to h modulo g *in one step* if and only if $lt(g)$ divides some term, t of f with the result h being:

$$h = f - \frac{t}{lt(g)}g \tag{2.1}$$

This reduction in one step is denoted:

$$f \xrightarrow{g} h$$

Similarly, f *reduces* to h modulo g if and only if a sequence of reductions in one step produce h from f. That is,

$$f \xrightarrow{g} h_1 \xrightarrow{g} h_2 \xrightarrow{g} \ldots \xrightarrow{g} h$$

This reduction is denoted:

$$f \xrightarrow{g}_{+} h$$

Moreover, we say a polynomial r is the remainder of f with respect to g if $f \xrightarrow{g}_{+} r$ and r is *reduced* with respect to g. Where reduced means either:

(1) $r = 0$, or
(2) no terms of r are divisible by the leading term of g.

One can see the similarities between reduction and division. Indeed, if f and g are univariate polynomials, then reduction exactly corresponds to polynomial division with remainder. Equation 2.1 looks very much like one division step, $r = a - qb$. It is for this reason that reduction is often seen as multivariate polynomial division. To be precise, multivariate polynomial division is a special case of reduction.

If we let $q = t/lt(g)$ be the quotient from one reduction step, then we can accumulate a quotient from the many steps over an entire reduction, to obtain a full quotient alongside the remainder. Then, we get the following:

Definition 2.4 (Multivariate Polynomial Division with Remainder)

Let $a, b, q, r \in \mathbb{K}[x_1, \ldots, x_v], b \neq 0$. Then q is the quotient and r the remainder, satisfying the equation

$$a \;=\; bq + r,\; r \text{ is reduced with respect to } b$$

We discuss our variation of multivariate division and its algorithms in Section 4.3

Reduction is more general than as has been described so far. Reduction in its full form replaces g by a set of polynomials $G = \{g_1, \ldots, g_n\}$. Thus, it is also sometimes referred to as *multi-divisor polynomial division.* If there were not enough synonyms already, the remainder of a multi-divisor polynomial division is also called a *normal form.* We discuss this in Section 4.5.

We say f reduces to h modulo G if and only if there exists a sequence, say $(i_1, \ldots, i_m)$, of reductions in one step, using one of $g_i \in G$ for each step.

$$f \xrightarrow{g_{i_1}} h_1 \xrightarrow{g_{i_2}} h_2 \xrightarrow{g_{i_3}} \ldots \xrightarrow{g_{i_m}} h$$

$$f \xrightarrow{G}_{+} h$$

We note that not every polynomial in G must be used for a reduction, and, it is very possible to use the same polynomial several times. In the case of multiple divisors, a polynomial r is reduced with respect to G if either:

(1) $r = 0$, or
(2) no terms of r are divisible by $lt(g_i)$ for every $g_i \in G$.

From this, we can see why it is important to fix a term order when discussing reductions. As we need to decide the leading term of the divisor, this changes depending on the term order used. Moreover, the term order then impacts the decision of whether a polynomial is reduced or not.

For example, under lexicographical ordering, the polynomial $f = x^2 + xy^4$ is reduced with respect to $g = x^3 + xy^3$ because x^3 does not divide either x^2 or xy^4. However, under degree lexicographical ordering, $f = xy^4 + x^2$ and $g = xy^3 + x^3$ and now f is no longer reduced with respect to g as xy^3 divides xy^4.

A similar problem occurs with multiple divisors. Let $f = x^2y$, $G = \{g_1, g_2\}$ with $g_1 = x^2$ and $g_2 = xy - y^2$,

$$f \xrightarrow{g_1} 0 \qquad \text{but} \qquad f \xrightarrow{g_2} xy^2 \xrightarrow{g_2} y^3,$$

and yet both 0 and y^3 are reduced with respect to G. Clearly, the order in which the divisors are applied makes a difference. So, reduction is an ambiguous problem. At least, that is the case when the divisor set is a general set of polynomials.

This is troublesome. Particularly because one solution to the identity membership problem is the following: for $G = \{a_1, \ldots, a_n\}$ and $\langle G \rangle = I \subseteq R$, $f \in R$,

$$f \xrightarrow{G}_{+} 0 \implies f \in I.$$

Since the order of application influences the final result, we cannot say the converse, that if $f \in I$ then $f \xrightarrow{G}_{+} 0$. However, Gröbner bases give us just that. They are special sets such that that the process of reduction with respect to them always yields a unique remainder. Therefore, if G is a Gröbner basis then

$$f \xrightarrow{G}_{+} 0 \iff f \in I.$$

This if and only if relation is a very important property of Gröbner bases. Algorithm 21.33 in [32] shows that every ideal admits a Gröbner basis, and it can be computed effectively. Hence, these bases allow effective and practical solving of the ideal membership problem, among many other problems. See Chapter 21 in [32] and all of [17] for many other such problems and applications.

2.2.5 Algebraic Geometry

Here, we give only a very brief review of some definitions which will become useful in later theorems. In particular, we are interested in hyperplanes, hypersurfaces, and their degrees.

An ordinary 2-dimensional surface exists in a 3-dimensional space. A hypersurface is just a generalization of a surface to arbitrary dimension. Consider a space — usually affine or Euclidean, the particulars of which are not important — of dimension n, say $\mathbb{K}^n$. This is the *ambient space*. Then, hypersurfaces are sub-spaces of dimension $n-1$. Equivalently, they have *codimension* 1 with their ambient space.

Hypersurfaces are defined by a single polynomial in $\mathbb{K}[x_1, \ldots, x_n] = \mathbb{K}[X]$ as $p(X) = 0$. A hypersurface is said to have degree d if the polynomial defining it has total degree d. A *hyperplane* is simply a hypersurface of degree 1. For example, the hypersurface defined by $p(x, y, z) = 2x^2yz + 3xz + y = 0$ is a hypersurface of degree 3 in $\mathbb{K}^3$, with the ambient space being $\mathbb{K}^4$.

We refer the reader to [51] for more details on geometry, algebraic varieties, and the like.

2.2.6 (Numerical) Linear Algebra

Beyond the very basics of linear algebra, like vectors, matrices, and systems of equations (with which we assume the reader is familiar), linear algebra is a very broad topic, even compared to algebra as a whole. Here, we only highlight a specific topic which some may not be familiar.

Given a matrix, $\mathbf{A} \in \mathbb{R}^{n\times m}$, a *singular value decomposition* (SVD) of $\mathbf{A}$ is a factorization into the form $\mathbf{UDV}^T$ where $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\mathbf{V} \in \mathbb{R}^{m\times m}$ are orthonormal and $\mathbf{D} \in \mathbb{R}^{n\times m} = diag(\sigma_1, \sigma_2, \ldots, \sigma_p)$, $p = min\{n, m\}$. The diagonal entries of $\mathbf{D}$ are known as the *singular values* of $\mathbf{A}$. Generally, algorithms produce $\mathbf{D}$ such that the singular values appear in a decreasing order, with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$ [23]. The diagonal matrix of the SVD gives the rank of $\mathbf{A}$ as r:

$$\mathbf{A} = \mathbf{UDV}^T = \mathbf{U}\begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \\ & & & & & \end{bmatrix}\mathbf{V}^T$$

If $\mathbf{A}$ is of full rank then $r = p = min\{m, n\}$. Otherwise, $\sigma_{r+1} = \cdots = \sigma_p = 0$.

In numerical analysis, the singular value decomposition of a matrix provides a great deal of information regarding the (numerical) properties of the matrix, such as, effective rank, nearness to singularity, and the matrix's condition number [33]. Whereas the actual rank of a matrix is given as the number of non-zero singular values, if $\sigma_{r+1}, \ldots, \sigma_p$ are very small but non-zero then it is possible to say that $\mathbf{A}$ is *numerically rank deficient.* This is important as the *condition number* of $\mathbf{A}$ — a measure of the sensitivity of a problem to numerical errors and perturbations — is given by $\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_p}$. Of course, if $\sigma_1 \gg \sigma_p \approx 0$ then the condition number will be very large and the ill-conditioning of $\mathbf{A}$ may cause drastic errors in further computations [23]. Thus, it can then be convenient to explicitly set $\sigma_{r+1} = \cdots = \sigma_p = 0$ and regard $\mathbf{A}$ as rank-deficient, with rank r.

Of importance to us is the ability to use a singular value decomposition as one possible solution to the *linear least squares problem* (Section 2.5.3). Other methods include QR-Factorization. For further review in numerical linear algebra we recommend reviewing [23].

2.3 Representing Polynomials

For the sake of representing polynomials effectively there are two aspects to consider. Algorithms operating on these polynomials can be evaluated by their running time and/or memory usage. Generally, they work inversely, in the sense that more memory means the algorithm can run more quickly (in theory) while using less memory requires more steps from the algorithm. This interplay is an important consideration, especially considering the memory configuration of modern computers (see Section 2.1). Ideally, we want both time and memory to be minimized.

Let us begin with an example: addition of two univariate polynomials. Let $a, b \in R[x]$ such that

$$a = a_n x^n + a_{n-1} + x^{n-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i \tag{2.2}$$

$$b = b_m x^m + b_{m-1} + x^{m-1} + \cdots + b_1 x + b_0 = \sum_{i=0}^{m} b_i x^i \tag{2.3}$$

In this representation, which we call a *dense* representation, notice that the indices of the coefficients match the exponent of their associated monomial. All terms are represented regardless of the value of its coefficient, in particular, terms with a 0 coefficient. This provides advantages for notations and computation. Assuming $n = m$ (otherwise, simply extend the smaller polynomial by adding zero terms so that the sizes match) then the addition, $a + b$ is simply:

$$a + b = \sum_{i=0}^{n} (a_i + b_i) x^i.$$

In contrast, for a sparse representation, it is not as simple as matching indices and adding coefficients, one must also match monomials, requiring more computational steps within the algorithm.

Consider a possible implementation of this dense univariate representation. An obvious solution is to encode each polynomial as simply an array of coefficients. In this scheme, the exponent on the monomial corresponding to each coefficient is implied by the coefficient's index (Figure 2.1) A polynomial addition algorithm in this representation is dead simple; it is nothing more than iterating through two arrays, summing corresponding indices. In this dense array representation, one can also immediately query things such as leading monomial, leading coefficient, number of terms, and degree.

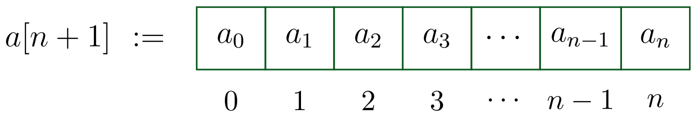


Figure 2.1: Array representation of a dense univariate polynomial.

Consider however, encoding the polynomial $x^{10000} - 1$ in this dense array representation. Although there are only four defining items (coefficients 1 and -1, and the monomials x^{10000} and x^0) we would need to create an array of size 10001 to encode the polynomial. Moreover, if this polynomial was used in some algorithm, say polynomial addition, much of the time of the algorithm would be spend summing coefficients where at least one operand was 0, a very wasteful operation.

This leads to the discussion of the *sparsity* of a polynomial itself [5]. In this sense, a polynomial *is* dense or sparse regardless of how it is represented. One says a polynomial is sparse if it has relatively few non-zero terms compared to the maximum number of terms possible for a polynomial with the same degree. Similarly, a polynomial is dense if it has relatively few zero terms.

A polynomial can either be *represented densely* or *represented sparsely*, regardless of whether it is dense or sparse. Naturally, sparse representations work best when they are used to represent sparse polynomials. The same goes for dense.

Of importance here, is deciding how to represent multivariate polynomials. For a fixed maximum partial degree, the number of multivariate polynomial terms increases exponentially in the number of variables. Even with a partial degree of no more than 2, multivariate polynomials in 3 variables have up to 27 terms while 10 variables already have up to 59049 terms. A dense representation would have to encode at least that many terms, a prohibitively large number as degrees and number of variables increase. However, multivariate polynomials are rarely dense in computer algebra problems. Generally speaking, multivariate polynomials are sparse, either with many variables, each of low degree, or a few variables, each of high degree but very few non-zero terms [27].

Mathematically, we can write sparse polynomials using summation notation as well, with slightly different notation. This is the same notation as was presented in Section 2.2.2, which we repeat here for completeness.

$$a \;=\; \sum_{i=1}^{n_a} a_i x_1^{e_{1i}} \dots x_v^{e_{vi}} \;=\; \sum_{i=1}^{n_a} a_i X^{\alpha_i}$$

In this summation notation, the n_a terms are sorted decreasingly according to lexicographical ordering. $a_i \in R$ is the i^{th} coefficient, and $e_{1i}, \dots, e_{vi}$ are exponents of the i^{th} monomial. To simplify notation, often a multivariate monomial will be written as X^{α_i}, where a capital letter X denotes the sequence of variables $x_1, \dots, x_v$ and $\alpha_i = e_{1i}, \dots, e_{vi}$ is a v-tuple or multi-index of exponents. We leave the discussion on the implementation of sparse representations to Chapter 3, where many details and implementation strategies are discussed.

Lastly, the idea of a *recursive* representation should not be forgotten. That is, representing a multivariate polynomial $p \in R[x_1, \dots, x_v]$ explicitly as $p' \in R[x_2, \dots, x_v][x_1]$.

[5] The sparsity of a polynomial can depend on the basis in which it is represented. In general, we use the monomial basis and our discussion of sparsity is with respect to this basis.

This representation is very useful computationally when wishing to perform essentially univariate operations on a multivariate polynomial. For example, pseudo-division (Sections 2.2.3 and 4.4), greatest common divisor, content and primitive part[32, Section 6.2], and subresultants [32, Section 6.10] are all essentially univariate operations.

We can easily define such a representation, again in summation notation, as:

$$f(x_1, x_2, \ldots, x_v) \;=\; \sum_{i=1}^{n} \; g_i(x_2, \ldots, x_v) \;\cdot\; x_1^i$$

This representation is tricky to implement in the sense that coefficients are themselves polynomials. However, it is also essentially univariate in its representation. Thus, it could be convenient to use a dense representation, like arrays, where the index implies the exponents on the main variable (x_1) and the entries in the array are the polynomial coefficients ($g_i(x_2, \ldots, x_v)$). In a C-like programming language the coefficients in the array could just be *pointers* to other polynomials, such as in Figure 2.2. We discuss our more efficient recursive sparse implementation in Section 3.4.

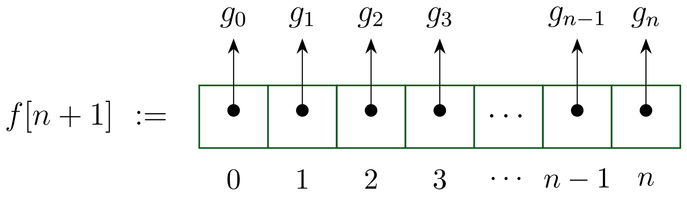


Figure 2.2: Array representation of a dense recursive multivariate polynomial.

For further discussion on polynomial representations in various computer algebra systems, [27] provides a good overview. For a detailed look at the implementations and representations of Maple and Singular see [60].

2.4 Working with Sparse Polynomials

As we saw in the last section, dense polynomial representations provide some computational advantages, such as very simple algorithms and the ability to instantly query information like degree, number of terms, and coefficients at a particular index. Many of these advantages are lost when working with sparse representations.

Without even considering implementation, one can see the difficulties of working with sparsely represented polynomials. Consider the question *does this polynomial contain the monomial* x^d, for some d. In a dense representation, $p = \sum_{i=0}^{n} p_i x^i$, the answer is immediate: does $p_d = 0$? For a sparse representation, $p = \sum_{i=1}^{n} p_i x^{\rho_i}$, one must iterate through all ρ_i, determining if any equal d (sorting the terms helps with this, but does not eliminate the problem entirely). It is much the same for addition, one cannot just match indices and add those corresponding coefficients. Arithmetic in general is more cumbersome for a sparse representation.

So why do we care about sparse polynomials? The short answer is memory. Computer memory is almost always the limiting factor. Historically, the amount of memory available limited the size of problems that computers could solve and how they solved them. Today, this is less of a concern. On modern architectures we are more concerned with how memory is used (Section 2.1).

Many algorithms were still (and are) developed for dense representations [48, Section 4.3.3 (How fast can we multiply?)]. Methods like Karatsuba [46], Toom-Cook [12], and Schönhage-Strassen [69] have been developed that are asymptotically very fast. However, for sparse (multivariate) polynomials, whose dense representations are prohibitively large, these algorithms are ineffective. Moreover, considering the processor-memory gap, we would like to maintain good cache-complexity for sparse polynomials, which is impossible if they are represented densely. Hence, we want to work with sparse polynomials represented sparsely.

Two very important works which focused on sparse polynomials considered their arithmetic [42] and their interpolation [76]. Sparse polynomial arithmetic in [42] was left unnoticed for many years, until its rediscovery in the late 2000s by Monagan and Pearce [56, 58, 59]. These algorithms for sparse arithmetic can be adapted to modern architecture with good cache complexity. Indeed, this is the main subject matter of Chapter 4. Here, we present the original sparse algorithms of [42] as a base to work from and to refer to later. Sparse interpolation is left to the discussion of Section 5.3.

The ideas of sparse arithmetic are built up in succession. Division relies on multiplication; multiplication relies on addition. We begin with addition. Let us fix the two operands of a binary polynomial operation as a and b with the result being c.

$$a = \sum_{i=1}^{n_a} a_i X^{\alpha_i} \qquad b = \sum_{j=1}^{n_b} b_j X^{\beta_j} \qquad c = \sum_{k=1}^{n_c} c_i X^{\gamma_k}$$

Addition (or subtraction) of two polynomials requires joining the terms of the two summands, combining like-terms (with possible cancellation) and then sorting the terms of the sum. Sorting is necessary in this case in order to maintain a canonical representation — an issue which will come up again for every operation. A naïve approach is to compute the sum $a+b$ term-by-term, adding a term of the addend (b) to the augend (a), inserting it in its proper position among the terms of a so that the term order is maintained. One could think of this as analogous to *insertion sort*.

This method is inefficient and does not take advantage of the fact that both a and b are already ordered. This can be accomplished more efficiently in terms of operations, and space, by performing a modified *merge sort* of the two summands. This takes advantage of the fact that the two summands are already sorted. The addition operation then combines terms with identical exponents as they are encountered (where the sum or difference of coefficients is computed), with the output being automatically sorted. This algorithm is presented in Algorithm 2.3. Subtraction is essentially the same by only replacing the addition of like-term coefficients with their subtraction.

Algorithm 2.3 addPolynomials(a,b)

$a, b \in R[x_1, \ldots, x_v],\ a = \sum_{i=1}^{n_a} a_i X^{\alpha_i},\ b = \sum_{j=1}^{n_b} b_j X^{\beta_j};$
return $c = a + b = \sum_{k=1}^{n_c} c_k X^{\gamma_k} \in R[x_1, \ldots, x_v]$

```
$(i, j, k) := 1$
while $i \le n$ and $j \le m$ do
    if $\alpha_i < \beta_j$ then
        $c_k := b_j$
        $\gamma_k := \beta_j$
        $j := j + 1$
    else if $\alpha_i > \beta_j$ then
        $c_k := a_i$
        $\gamma_k := \alpha_i$
        $i := i + 1$
    else
        $c_k := a_i + b_j$
        $\gamma_k := \alpha_i$
        $i := i + 1$
        $j := j + 1$
        if $c_k = 0$ then
            continue   #Don't increment $k$
    $k := k + 1$
end
while $i \le n$ do
    $c_k := a_i$
    $\gamma_k := \alpha_i$
    $i := i + 1$
    $k := k + 1$
while $j \le m$ do
    $c_k := b_j$
    $\gamma_k := \beta_j$
    $j := j + 1$
    $k := k + 1$
return $c = \sum_{\ell=1}^{k} c_\ell X^{\gamma_\ell}$
```

Much like addition, polynomial multiplication requires generating the terms of the product, combining like-terms among the product terms, and then sorting the product terms. A naïve approach is to compute the product $a \cdot b$ by distributing each term of the multiplier (a) over the multiplicand (b), combining like terms, and sorting: $c = a \cdot b = (a_1 X^{\alpha_1} \cdot b) + (a_2 X^{\alpha_2} \cdot b) + \cdots$. This is inefficient because all $n_a n_b$ terms are generated, whether or not they combine as like-terms. Also, the final $n_a n_b$ terms must be sorted.

We can obtain more efficient algorithms by generating terms in sorted order. In such a way, like-terms are immediately found and combined as early as possible. The sparse structure a and b is put to good use by observing that for a given α_i and β_j we always have that $X^{\alpha_{i+1}+\beta_j}$ and $X^{\alpha_i+\beta_{j+1}}$ are less than $X^{\alpha_i+\beta_j}$ in the term order. Given that $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$, $X^{\alpha_i+\beta_j} \ge X^{\alpha_i}$, and $X^{\alpha_i+\beta_j} \ge X^{\beta_j}$, we can generate terms of the product in order by merging n_a "streams" of terms obtained by multiplying a single term of a distributed over b.

$$a \cdot b = \begin{cases} (a_1 \cdot b_1)\, X^{\alpha_1+\beta_1} & + (a_1 \cdot b_2)\, X^{\alpha_1+\beta_2} + (a_1 \cdot b_3)\, X^{\alpha_1+\beta_3} + \dots \\ (a_2 \cdot b_1)\, X^{\alpha_2+\beta_1} & + (a_2 \cdot b_2)\, X^{\alpha_2+\beta_2} + (a_2 \cdot b_3)\, X^{\alpha_2+\beta_3} + \dots \\ \vdots & \\ (a_{n_a} \cdot b_1)\, X^{\alpha_{n_a}+\beta_1} & + (a_{n_a} \cdot b_2)\, X^{\alpha_{n_a}+\beta_2} + (a_{n_a} \cdot b_3)\, X^{\alpha_{n_a}+\beta_3} + \dots \end{cases}$$

We can consider this like an n_a*-way* merge sort, where at each step, we select the maximum term from the heads of the streams and use it at the next term in the product, removing it from the stream in the process. The new head of the stream where a term is removed is then the term to its right. To be computationally effective in implementation, the product $(a_i \cdot b_j)X^{\alpha_i+\beta_k}$ is only actually computed when it is removed from a stream (see Section 4.2.1 for implementation details). If there is no unique maximum, then the maximums are all like-terms and we can select all such terms and add their coefficients together to form a single term of the product. This is shown in the algorithm (Algorithm 2.4) by accumulating sums of products in c_k (line 16), and only updating k when the maximum degree "drops" and the resulting coefficient is non-zero (line 11).

We use a variable to keep track of the index of the head of each stream, and do a brute-force search over the those heads for the maximum. We use the variable f_s to give the "column" index of each stream, where s is the index of the row (stream). Thus, f_s picks out the current head element of stream s. Further, we use the index I to denote the index of the first non-empty stream. In this n_a-way merge, since we have $X^{\alpha_i+\beta_j} > X^{\alpha_{i+1}+\beta_j}$ *and* $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$, then the streams will become "empty" in order of increasing a_i. Maintaining I provides advantages for notation and computations.

The last of the arithmetic operations discussed in [42] is exact division. Division is a direct application of multiplication. In fact, it is simply formulated as a multiplication in which one of the operands is continuously updating. We explain it fully in Section 4.3 where it is extended to division with remainder. We also expand on the rather terse original presentation of the algorithm.

2.5 Interpolation & Curve Fitting[6]

Interpolation has a long history with many applications. Arguably, the history of interpolation is more rich in numerical analysis but, nonetheless, it is fundamental to both symbolic and numeric computation. Precisely, interpolation is the process of, given a set of (possibly multivariate) points, π_i, and values, β_i, finding a function, f, such that $f(\pi_i) = \beta_i$. In essence, interpolation is the process of transforming a set of discrete data points into a function. This function may be the exact function which produced the data

[6] The unpublished work of George Miminis, *Introduction to Scientific Computing*, is to thank for the lovely details interconnecting basis polynomials, interpolation, and curve fitting.

Algorithm 2.4 multiplyPolynomials(a,b)
$a, b \in R[x_1, \dots, x_v],\ a = \sum_{i=1}^{n_a} a_i X^{\alpha_i},\ b = \sum_{j=1}^{n_b} b_j X^{\beta_j};$
return $c = a + b = \sum_{k=1}^{n_c} c_k X^{\gamma_k} \in R[x_1, \dots, x_v]$

```
if $n_a = 0$ or $n_b = 0$ then
    return $0$
$k := 1$; $c_1 := 0$
$\gamma_1 := \alpha_1 + \beta_1$
for $i = 1$ to $n_a$ do
    $f_i := 1$
end for
$I := 1$
while $I \leq n_a$ do
    $s := \max_s \{\alpha_s + \beta_{f_s} \mid I \leq s \leq n_a\}$
    if $\gamma_k \neq \alpha_s + \beta_{f_s}$ then
        if $c_k \neq 0$ then
            $k := k + 1$
            $c_k := 0$
        $\gamma_k := \alpha_a + \beta_{f_s}$
    $c_k := c_k + a_s b_{f_s}$
    $f_s := f_s + 1$
    if $f_s > n_b$ then
        $I = s + 1$
end
return $c = \sum_{\ell=1}^{k} c_\ell X^{\gamma_\ell}$
```

points (if sufficient information about the function is known), or simply equal to the underlying function at the set of points provided. Approximation (curve fitting) relaxes the constraint that the function f must equal β_i at each interpolation node, π_i. Rather, one hopes to find f that fits the data "as close as possible" (see Section 2.5.3).

Of course, many engineering applications need exactly this sort of transformation where data points are mere observations, and a function needs to be determined in order to continue the mathematical analysis. The succinct representation of a data set as a function provides a starting point for many other algorithms. These needs range from evaluating the function at various (non-observed) points [48, Section 4.6.2], to analyzing its derivative or integral.

There are various flavours of interpolation including nearest-neighbour, linear, polynomial, and trigonometric. The differences in these flavours depend on the choice of *basis functions* for the interpolation. That is, a set of functions such that other functions can be generated as a linear combination of the basis functions. The well-known Taylor series [18, Section 1.1] of a function shows that a function can be written as a linear combination of the set of monomials $\{1, x, x^2, x^3, \dots\}$. Hence, these monomials form a basis of functions, yielding *polynomial interpolation*. Other function bases may be a set of linear functions, the trigonometric functions, or, as we will see, specifically defined functions.

Let us call such a generic set of basis functions $\phi_j(x)$, $1 \leq j \leq m$. As we wish to interpolate the function using this basis of functions and the point-value pairs (π_i, β_i),

$1 \leq i \leq n$, then we can easily create a system of linear equations describing this,

$$\alpha_1\phi_1(\pi_i) + \alpha_2\phi_2(\pi_i) + \alpha_3\phi_3(\pi_i) + \cdots = \beta_i,$$

where α_i are the desired coefficients of our basis functions ϕ_j. In matrix notation, these linear equations are represented as:

$$\mathbf{Ax} = \mathbf{b}$$

$$\begin{bmatrix} \phi_1(\pi_1) & \phi_2(\pi_1) & \dots & \phi_m(\pi_1) \\ \phi_1(\pi_1) & \phi_2(\pi_2) & \dots & \phi_m(\pi_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\pi_n) & \phi_2(\pi_n) & \dots & \phi_m(\pi_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

If $\phi_1 = 1$, $\phi_2 = x$, $\dots$, $\phi_m = x^{m-1}$ then we have a *monomial basis* resulting in polynomial interpolation. Of course, this system of linear equations could be solved directly to obtain the coefficients of our linear combination of ϕ_j. However, this method is a little brutal (in the sense of brute-force). Moreover, this connection between linear algebra and interpolation is an important one. In particular, we note the appearance of the Vandermonde matrix. If we let $\phi_1 = 1$, $\phi_2 = x$, $\dots$, $\phi_m = x^{m-1}$, as in polynomial interpolation, then the our matrix of points (let us call this the *sample matrix*) becomes a Vandermonde matrix over $(\pi_1, \pi_2, \dots, \pi_n)$.

$$\begin{bmatrix} 1 & \pi_1 & \pi_1^2 & \dots & \pi_1^{m-1} \\ 1 & \pi_2 & \pi_2^2 & \dots & \pi_2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \pi_n & \pi_n^2 & \dots & \pi_n^{m-1} \end{bmatrix}$$

Vandermonde matrices are known to be ill-conditioned for real values [7], providing another reason to avoid solving the system of linear equations directly (at least numerically). As we will see in Sections 2.5.1 and 2.5.2 there exists more efficient and direct ways to calculate the coefficients α_j for the monomial basis instead of simply solving the system of equations. However, theoretically they provide some intuition about interpolation.

If our sample matrix turned out to be singular, then clearly the coefficients α_j cannot be uniquely determined and hence our interpolating function cannot be unique. In the case of polynomial interpolation then, the interpolating polynomial is unique if the Vandermonde matrix is non-singular, which is the case if the points π_i are pairwise distinct [68], provided that $n = m$. That is, the number of points is equal to the number of basis polynomials.

In contrast, if $n > m$, it is not possible to find a unique interpolating function in general. Then, interpolation becomes the problem of *curve fitting*. Whereas interpolation supposes that we find a f that exactly interpolates the points, that is, $f(\pi_i) = \beta_i$, curve fitting relaxes this equality and instead wishes to *minimize* the difference $f(\pi_i)$ and β_i.

For example, with a monomial basis, if we wish our function to have a maximum degree d then we must use $d+1$ points to obtain a unique polynomial with degree d. However, if our points come from some experimental observations, it may be useful to include all such observations and exceed $d+1$ points. In this case, we are fitting a curve (polynomial) to the data as best as possible, instead of exactly interpolating. We review this procedure in Section 2.5.3

2.5.1 Lagrange Interpolation

Lagrange presented his famous *Lagrange interpolation* centuries ago, in 1795, and his interpolation method is still widely used [48]. This scheme interpolates a function using *Lagrange basis polynomials*. Using our notation of ϕ_j as our basis functions and (π_i, β_i) as point-value pairs, then ϕ_j is defined as:

$$\begin{aligned}\phi_j(x) &= \frac{(x-\pi_1)\ldots(x-\pi_{j-1})(x-\pi_{j+1})\ldots(x-\pi_n)}{(\pi_j-\pi_1)\ldots(\pi_j-\pi_{j-1})(\pi_j-\pi_{j+1})\ldots(\pi_j-\pi_n)} \\ &= \prod_{\substack{i=1 \\ i\neq j}}^{n} \frac{(x-\pi_i)}{(\pi_j-\pi_i)}\end{aligned}$$

We note than the denominator is actually a single composite number, and the numerator is a simple product of degree 1 polynomials. Hence, each ϕ_j is a degree $n-1$ polynomial. Assuming that $m = n$, in order to obtain a unique interpolating polynomial, we define the *Lagrange interpolating polynomial* as the summation of each of these ϕ_j basis functions:

$$f(x) = \sum_{j=1}^{m} \beta_j \phi_j(x)$$

As f is a simple sum of degree $n-1$ polynomials, then it is also a degree $n-1$ polynomial.

Notice that, by construction, $\phi_j(x)$ either equals 1 or 0 at each of the π_i points. It is easy to see that $\phi_j(\pi_i) = 0$ for $i \neq j$, as one of the factors of the numerator will become zero, zeroing the entire function. For $\phi_j(\pi_j)$, then the factors of the numerator exactly match the denominators:

$$\begin{aligned}\phi_j(\pi_j) &= \frac{(\pi_j-\pi_1)\ldots(\pi_j-\pi_{j-1})(\pi_j-\pi_{j+1})\ldots(\pi_j-\pi_n)}{(\pi_j-\pi_1)\ldots(\pi_j-\pi_{j-1})(\pi_j-\pi_{j+1})\ldots(\pi_j-\pi_n)} \\ &= 1\end{aligned}$$

Since f is a simple linear combination of these q_j polynomials, then of course f meets the requirement of an interpolating polynomial as $f(\pi_i) = \beta_i$. This is easily seen by expanding f:

$$\begin{aligned} f(x) &= \beta_1\phi_1(x) \ + \cdots + \beta_j\phi_j(x) \ + \cdots + \beta_m\phi_m(x) \\ \implies f(\pi_j) &= \beta_1\phi_1(\pi_j) + \cdots + \beta_j\phi_j(\pi_j) + \cdots + \beta_m\phi_m(\pi_j) \\ &= \ \ \beta_1 \cdot 0 \quad + \cdots + \quad \beta_j \cdot 1 \quad + \cdots + \quad \beta_m \cdot 0 \\ &= \ \ \beta_j \end{aligned}$$

What is worth highlighting is the directness of obtaining an interpolating polynomial by this method. It is a simple construction without any real computation. However, to obtain a more succinct representation, one should expand all of the factors of each ϕ_j and combine them to obtain a single degree $n-1$ polynomial[7] This carries with it some high one-time arithmetic costs for expansion, but then a succinct representation is known. Another difficulty with is that the π_i must be pairwise distinct. Of course this is the case, otherwise a factor in the denominators of the ϕ_j basis polynomials would be $(\pi_i - \pi_i)$, leading to division by 0. This same result was seen by the singularity of the Vandermonde matrix obtained from the monomial basis using linear systems to find the interpolating polynomial. Moreover, the same connection between Lagrange and the Vandermonde matrix hints that Lagrange interpolation is numerically unstable as well.

2.5.2 Newton Interpolation

Whereas Lagrange interpolation is a very direct method but high in its initial arithmetic computations, and where solving a system of linear equations is neither direct nor light in arithmetic, *Newton's interpolating polynomial* fits halfway between. Newton's method for finding an interpolating polynomial is less direct than Lagrange but requires less arithmetic to obtain a simplified interpolant. Newton interpolation uses the polynomial basis:

$$\begin{aligned} \phi_1(x) &= 1, \\ \phi_2(x) &= (x-\pi_1), \\ \phi_3(x) &= (x-\pi_1)(x-\pi_2), \\ \phi_4(x) &= (x-\pi_1)(x-\pi_2)(x-\pi_3) \\ &\ \vdots \\ \phi_j(x) &= \prod_{i=1}^{j-1}(x-\pi_i) \end{aligned}$$

Much like Lagrange, these basis polynomials are constructed in a way that become 0 for some input $x = \pi_i$. For example, $\phi_2(\pi_1) = 0$, $\phi_3(\pi_1) = \phi_3(\pi_2) = 0$, and so on. The

[7] Numerically, this expansion is not a good idea, especially if two interpolation points are close together. Using the barycentric forms would be best [23].

polynomials are then linearly combined to produce the interpolating polynomial. Here, we are again assuming that the number of basis functions equals the number of points.

$$f = \sum_{j=1}^{m} \left(\alpha_i \prod_{i=1}^{j-1} (x - \pi_i) \right)$$

Let the empty product (when j=1) be $\prod_{i=1}^{0} = 1$.

The coefficients of the interpolating polynomial, α_i, can be computed directly using *divided differences* [23]. However, it is also possible to set up a system of linear equations by assuming $\phi_j(\pi_j) = \beta_j$. The sample matrix generated by this is a lower triangular matrix, so the system of equations can be solved by a simple forward substitution.

$$\begin{bmatrix} 1 & & & & \\ 1 & (\pi_2 - \pi_1) & & & \\ 1 & (\pi_3 - \pi_1) & (\pi_3 - \pi_1)(\pi_3 - \pi_2) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & (\pi_n - \pi_1) & (\pi_n - \pi_1)(\pi_n - \pi_2) & \dots & \prod_{i=1}^{n-1}(\pi_n - \pi_j) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_n \end{bmatrix}$$

2.5.3 Curve Fitting and Linear Least Squares

In the previous two sections, we have assumed that the number of points (n) is equal to the number of basis functions (m). In the case that the number of points exceeds the number of basis functions, generally, it is impossible to find a unique function which exactly interpolates all of the points. Comparatively, curve fitting, in general, is about finding some smooth function to "best fit" the collection of data and not necessarily interpolate it exactly.

It is not necessary that this fitted curve be formed by a basis of functions in the strict sense. Rather, one may simply want to *estimate the parameters* of some model, whether those be coefficients, exponents, etc. However, if we restrict this model to be a polynomial then we do obtain the known polynomial basis (or one of its variants) where the model parameters are simply coefficients. This type of fit is useful when looking to interpolate (in the statistical sense) new values from the observed ones [61].

Unfortunately, in this same case, symbolic methods can become troublesome. Consider if error was present in the data, then symbolic interpolation techniques would exactly fit the the error as well, which is naturally undesirable. Such error can easily occur during the collection of data, say by limitations of the measuring devices. This data can be described as inexact or *noisy*. Then, using exact symbolic methods is not well suited (and will likely fail entirely), hence, we make use of numerical methods in these circumstances.

One such numerical method is the least squares method for curve fitting. Least squares attempts to fit the curve to the data by minimizing the sum of squares of distances between the fitted curve and the data points [61]. For such a curve, f, this is:

$$\sum_{i=1}^{n} (f(\pi_i) - \beta_i)^2$$

This sum of squares can be modeled as the square of the Euclidean norm of the *residual vector*, $\mathbf{r}$.

$$||\mathbf{r}||_2^2 \; = \; ||(r_i)||_2^2 \; = \; ||(f(\pi_i) - \beta_i)||_2^2$$

This is only one possible such "best fit" for the collection of data. Using norms other than the Euclidean norm leads to other best fits.

For *linear least squares* where the model parameters are combined only linearly, as is the case for a polynomial model, then we can model the data points, model parameters, and residual vector as a system of linear equations. Here, we have $\mathbf{A} \in \mathbb{R}^{n\times m}, \mathbf{x} \in \mathbb{R}^m, \mathbf{b} \in \mathbb{R}^n$, $n > m$. Again using the Euclidean norm, we arrive at a typical definition of linear least squares [23, 33]:

$$\mathbf{x} \; = \; \min_{\mathbf{x}} ||\mathbf{r}||_2^2 \; = \; \min_{\mathbf{x}} ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2$$

One possible method to solve this problem is by the so-called *normal-equations.* Assuming $rank(A) = m$, it can be shown [23, Appendix C] that the minimal solution to the linear least squares problem is the same solution $\mathbf{x}$ of:

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$$

These are the normal equations. Assuming that $\mathbf{A}$ has full rank, then $\mathbf{A}^T\mathbf{A}\mathbf{x} \in \mathbb{R}^{m\times m}$ will be a non-singular, square matrix. Hence, a unique solution exists, and the system is no longer over-determined. One can then solve this system by any normal means.

However, this is not a numerically stable method. Given the condition number of $\mathbf{A}$ as $\kappa(\mathbf{A})$, the condition number of $\mathbf{A}^T\mathbf{A}$ is $\kappa(\mathbf{A}^T\mathbf{A}) = (\kappa(\mathbf{A}))^2$ [49]. Hence, any ill-conditioning present in $\mathbf{A}$ is made drastically worse in the normal equations and any algorithm making use of normal equations will be inherently unstable. Moreover, if $rank(\mathbf{A}) < m$ then $\mathbf{A}^T\mathbf{A}$ could be singular, and thus, have no unique solution.

More stable numerical algorithms rely on orthogonal factorizations, like QR-factorization, or singular value decomposition. These methods are discussed in Section 5.4.

2.6 Symbols and Notation

In this section we provide a summary of the symbols and notations used throughout the text for quick reference.

Rings

- R is a generic ring.
- $\mathbb{D}$ is a generic integral domain.
- $\mathbb{K}$ is a generic field.
- $\mathbb{Z}$ is the set of integers.
- $\mathbb{Q}$ is the set of rational numbers.
- $\mathbb{R}$ is the set of real numbers.
- $\mathbb{N}^0$ is the set of natural numbers including zero, or equivalently, the non-negative integers.

Polynomials

- Lowercase Latin letters, a, b, f, g, q, r, *etc.* are polynomials.
- Uppercase Latin letters with subscripts, A_i, B_i, Q_i, R_i, *etc.* are polynomial terms.
- Greek letters are, generally, tuples or multi-indices, such as exponent vectors or multi-dimensional points.
- $\mathrm{lt}(f)$ is the leading term of f.
- $\mathrm{lc}(f)$ is the leading coefficient of f.
- $\deg(f)$ or $\deg(F_i)$ is the degree of a polynomial or polynomial term, respectively.
- $\mathrm{coef}(F_i)$ is the coefficient of a polynomial term.
- n_a or $\#(a)$ denote the number of non-zero terms in the polynomial a.
- $x_1, x_2, \dots, x_v$ are variables or indeterminants.
- X is a collection of variables that will be clear by context.

Interpolation and Linear Algebra

- ϕ and ψ are bases functions of an interpolation.
- (π_i, β_i) is the set of point-value pairs in an interpolation.
- Bold uppercase Latin letters, $\mathbf{A}, \mathbf{C}$, are matrices.
- Bold lowercase Latin letters, $\mathbf{b}, \mathbf{x}$, are vectors.

Chapter 3

Memory-Conscious Polynomial Representations

The effective use of computer memory is a leading concern of high-performance programming. Although the amount of memory available to programmers on modern computers is vastly larger than the standard only a few years ago, the speed (or response time) of that memory has barely improved. Therefore, it is often the *memory wall* which limits the performance of an algorithm. That is, the latency of instructions waiting for data to be retrieved from memory is the limiting factor of a program's wall-clock time [73]. Processors are therefore underutilized.

A programmatic attempt to help push the memory wall closer towards the processing wall is related to *cache complexity* (see Section 2.1.1). That is, a study of how frequently a *cache miss* occurs during the running of an algorithm. Since cache and memory are precisely concerned with the data in which an algorithm operates, of course optimizing the data's structure (think layout in memory) will have benefits for the cache complexity. In particular, the more compact a representation, the fewer bytes a single data element requires in memory, and thus the more data elements can fit in cache at once, improving cache complexity.

Other programmatic ways of improving cache complexity, and thus a program's performance, occur by optimizing the way an algorithm accesses a data structure. These implementations details remain for the next chapters: polynomial arithmetic, Chapter 4, and polynomial interpolation, Chapter 5. Here, we discuss the data structures in which we can encode the data of a polynomial.

The basic structure of a polynomial is a simple collection of terms. Since we are interested in sparse representations, these collections encode only the non-zero terms. That collection is augmented with some *meta-data* of sorts. Information about the polynomial such as the number of terms, total allocated space, and variable symbols. This header information is rather succinct and requires essentially constant memory to store. The real effort is in effectively representing this collection of terms. But before we consider collections, how can we effectively encode a single term? This is not as obvious as one would think.

3.1 Coefficients, Monomials, and Exponent Packing

A single polynomial term is a rather simple structure; we have a coefficient (whether an integer or rational number) and pair it with a monomial. Because we require arbitrary-precision integers and rational numbers, we encode coefficients using GMP (GNU Multi-Precision arithmetic) numbers [37]. GMP is a leader in performance for arbitrary-precision arithmetic and thus we rely on their highly optimized implementation in C. For monomials, there is more work to do.

The defining characteristics of a monomial are its variables and their associated exponents. However, since we have imposed a variable ordering (as part of the term ordering) there is no need for the symbols of those variables to be stored in the monomial itself. We need only the exponents. In particular, as we have denoted a monomial by X^{α_i}, where α_i is a multi-index, it is natural to simply encode this multi-index. We call this encoding an *exponent vector*.

The simplest strategy for encoding an exponent vector is just that, a vector (array) of exponents (unsigned integers). Say we wish to encode an exponent vector of v variables. Then, as a typical unsigned integer is 32 bits, the corresponding exponent vector would require $32v$ bits or $4v$ bytes of memory. However, this is an inefficient use of memory.

Consider the binary representation of the number 5 in a 32-bit unsigned integer. It is $0b00000000000000000000000000000101$. Obviously, there are many leading zeros in this representation. For polynomials, and in particular multivariate polynomials, exponents very rarely require the full space of a 32-bit integer to be encoded. That is, they rarely ever reach close to a value of $2^{32} - 1 = 4294967295$. One might consider using a `short` data type instead. However, this is still 16-bits or a value of $2^{16} - 1 = 65535$, still rather large.

Therefore, we look to improve upon this representation to avoid the wasted space of unnecessary leading 0s. One strategy is called *exponent packing*, a method of encoding multiple integers into a single (64-bit) machine word. Basically, the bits of a word are partitioned (conceptually, not actually) into sections, where each section holds the bits to encode a single integer. Using bit-masks and shifts, multiple integers, each of small absolute value, can effectively be stored in a single 64-bit machine word. The idea of exponent packing has been employed at least since ALTRAN in the late 60s [38] and more recently in [59] and [40].

Some systems, like Maple, also encode the total degree of the monomial in the single 64-bit word. This is more useful when operating under a term ordering which makes uses of degrees in its ordering, like degree lexicographical. It is unneeded in a lexicographical ordering as we use. Further, this scheme wastes bits which could be used for additional variables or higher degrees. In particular, under the scheme used in Maple monomials are limited to 21 variables each with a maximum degree of 3 [60]. Our representation does not encode total degree, therefore we can encode up to 32 variables, each of maximum degree 3, a substantial difference. Moreover, in polynomial system solving, degrees of lower ordered variables often increase much quicker than those of high ordered variables. Thus, in our implementation, we pack exponents disproportionately within the machine word, giving more bits to lower ordered variables, ensuring all 64 bits are made useful. This again differs from the variation of exponent packing in Maple where each variable is given the same number of bits and, therefore, when 64 is not divisible by the number of variables, wastes bits. Figure 3.1 shows the packing for a monomial in 3 variables. The monomial with maximum degree which can be represented in this way is thus $x_1^{65535}x_2^{1048575}x_3^{268435455}$.

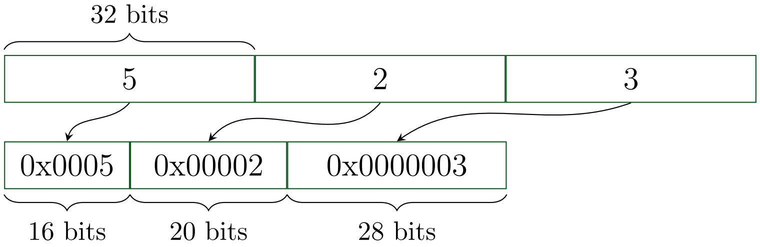


Figure 3.1: A 3-variable exponent vector packed into a single machine word.

The benefits of exponent packing are obvious in terms of memory savings. But, we also gain an important computational advantage since we use a single machine word now for each monomial. Comparison and multiplication of monomials reduces to machine word comparison and addition, respectively. As we saw in the algorithms for sparse addition (Algorithm 2.3) and sparse multiplication (Algorithm 2.4) monomial comparisons are an integral part of arithmetic operations in order to maintain an ordered, canonical representation. Therefore, the savings are twofold, both in memory usage and number of machine instructions.

Putting this all together, we obtain a very succinct representation of a single polynomial term. One GMP coefficient, and one long unsigned integer (64-bit word). GMP integers (rational numbers) require 16 bytes (32 bytes) to encode[1], therefore, a single polynomial term only requires 24 bytes (40 bytes) to encode entirely. Notice this is a fixed amount regardless of the number of variables. That is a very nice property. With these numbers then, we can effectively represent many millions (44.7 million, to be precise, for integer coefficients) of polynomial terms in under 1 gigabyte of memory – a relatively small amount on modern machines.

[1] This is not strictly true. As these are arbitrary-precision integers, of course they can grow arbitrarily large. However, their main structure – and the only structure one can interact with directly – has this fixed size.

3.2 Linked Lists

Now that we have an effective representation of a single polynomial term the next step is to collect multiple terms effectively into a single data structure in order to encode an entire polynomial. The simplest scheme to represent this collection would be a *linked list* – a list of nodes linked together using pointers [70, Section 1.3]. Each node in the list is used to encode a single piece of data; in our context, a single term of the polynomial. Figure 3.2 shows the polynomial $13x^2y^3 + 5x^2y + 7y^3z$ encoded as a linked list.

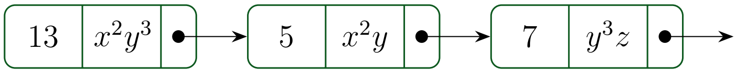


Figure 3.2: The polynomial $13x^2y^3 + 5x^2y + 7y^3z$ encoded as a linked list.

This representation makes handling and manipulating terms very easy with simple pointer manipulation. Moreover, new elements can easily be added or removed from the list. This seems like a very natural operation for sparsely represented polynomials. However, there are other operations which perform poorly for linked lists. In particular, indexing, counting the number of terms, and finding a particular monomial all require traversing the entire list. These $O(n)$ operations are not reasonable.

Moreover, the indirection created by pointers could possibly lead to poor locality for successive nodes in the list, making this scheme inefficient for cache. Further still, a node requires 8 bytes worth of memory to store the pointer to the next node. Hence, the representation of a single polynomial term increased from 24 bytes (40 for rational numbers) to 32 (48) bytes. That is a substantial percentage spent encoding (meaningless) structure, instead of pure data.

Thus, we aim to remove this overhead of storing pointers, as well as looking to minimize indirection and possibly poor data locality. Packing these nodes (polynomial terms) tightly into an array would solve both of these problems.

3.3 Alternating Arrays

The alternating array representation packs terms side-by-side in an array, effectively alternating between coefficients and monomials (hence the name). This also follows the terminology introduced in 1997 in the `BasicMath` library, part of the European Project FRISCO `https://cordis.europa.eu/project/rcn/31471_en.html`; see also [16].

A coefficient and its corresponding monomial are side-by-side in memory and are thus optimally local with respect to each other. Similar schemes have been used in Maple [57, 60]. However, in the case of Maple, their scheme uses pointers into a parallel array to store the arbitrary-precision coefficients, whereas we store the arbitrary-precision coefficients directly in the array (see Section 4.1 for further discussion). Of course, this increases the locality of a coefficient with respect to its monomial. In contrast, storing the coefficient directly in the array also causes the monomials (coefficients) of adjacent polynomial terms to be further apart in memory, decreasing locality in that regard. While this is the case, we argue that a monomial without its coefficient nearby is essentially useless, one uses both simultaneously, especially in the case of arithmetic. We also note that this array data structure in Maple is limited to integer polynomials while all other polynomials use an inefficient sum-of-products encoding [60]. In contrast, our alternating array representation supports both integer and rational number coefficients.[2].

Of course, in our alternating array representation, terms are stored in decreasing lexicographical order to maintain a canonical representation. This, in addition to the array data structure allows for basic operations to occur very efficiently. Operations like degree, leading coefficient, indexing, and number of terms are all constant-time operations. Conversely, unlike linked lists, alternating arrays do not allow for easy insertion or removal of elements in arbitrary position within the array. This is justified due to the many other memory savings we obtain by removing pointer overhead and indirection. Moreover, our algorithms both use and produce polynomial terms in order (see Chapter 4). Hence, inserting in the middle of the list is never really needed.

For our alternating array implementation, we again make use of GMP to store the coefficients and use exponent packing with long unsigned integers to encode the exponent vectors. The elements of the array alternate between coefficients and monomials in a tightly-packed manner. There is no overhead whatsoever. Therefore, we obtain the best encoding , with no overhead, for a collection of polynomial terms, using only 24 bytes (40 for rational number) per polynomial term. Figure 3.3 shows a polynomial encoded as an alternating array.

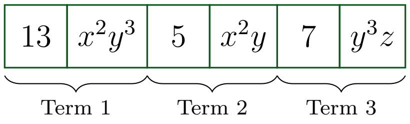
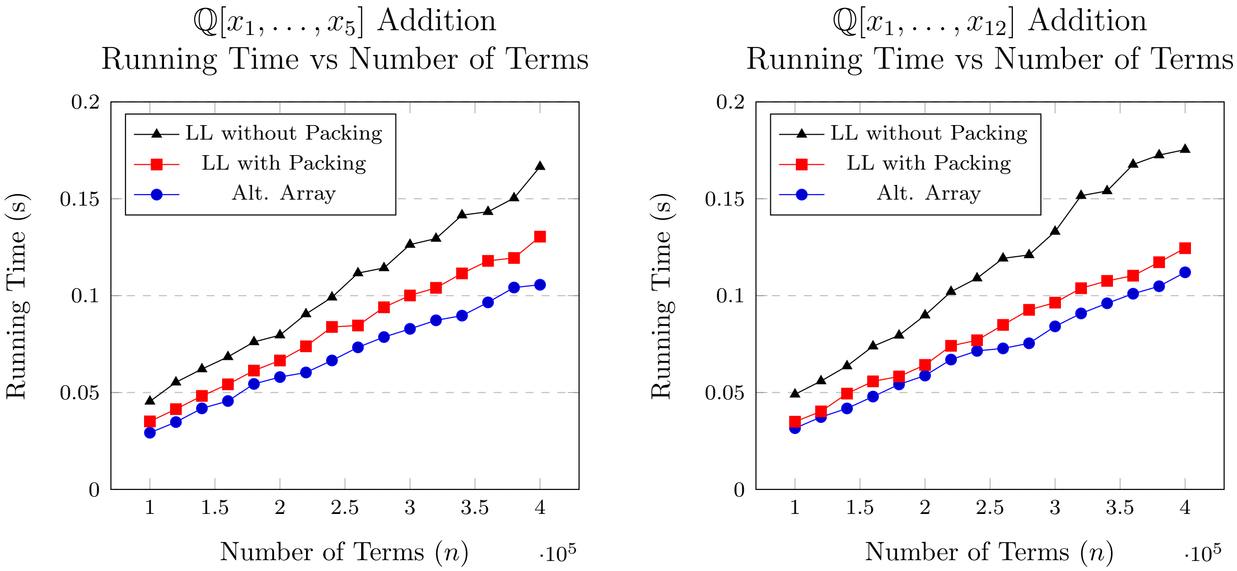


Figure 3.3: The polynomial $13x^2y^3 + 5x^2y + 7y^3z$ encoded as an alternating array.

Using polynomial addition as an example, we compare the effects of memory usage and data structures in Figure 3.4. In this plot we run the addition algorithm for polynomials over $\mathbb{Q}$ with various numbers of non-zero terms and variables. We note the algorithm for doing the addition is exactly the same. The only difference is the data structures themselves. We can see that while the curves are essentially linear in the number of terms they have different slopes. The linked list running time is quickly diverging from that of alternating arrays. We account for this discrepancy by the reduced indirection and size of individual data elements between the linked list and alternating array representations. Even more drastically, the variation without exponent packing

[2] Using floating point coefficients is also arbitrarily easy to implement as they essentially form a field that is a subset of the rational numbers. In fact, we use such polynomials for numerical interpolation (Section 5.4)

diverges even more quickly. This is a result of the increased bytes required to encode each monomial, which is exacerbated as the number of variables increase.



(a) 5 variables

(b) 12 variables

Figure 3.4: Comparing linked list, with and without exponent packing, and alternating array implementations of polynomial addition. Rational number polynomials in 5 and 12 variables are added together for various sizes.

Addition is a good algorithm for comparing data structures because sparse addition results in very little arithmetic work. Only like-terms are combined via arithmetic, while the other terms are simply combined by memory movement and sorting them into the proper term order. Since these are sparse polynomials we expect few like-terms to actually occur during a polynomial addition. Hence, it could be considered a *memory-bound* problem. We can analyze the cache complexity to see the effects of how using memory impacts performance. For the purposes of cache analysis we must make certain assumptions. In particular, about the memory usage and traversal of the GMP coefficients.

Hypothesis 3.1 (GMP Cache Analysis Assumptions) *Let us assume that the GMP coefficients are sufficiently small (say, less than 64 bits) and that coefficient the* data *is stored within the main GMP struct which, in turn, is stored within the alternating array structure. This is not strictly true, but is a fair assumption to analyze the cache usage of the polynomial arithmetic algorithms (see Figure 4.1 and surrounding discussion for further details on GMP usage within the alternating array structure).*

Under the ideal cache model (Section 2.1.1), where the cache has L words per line, and assuming 8 bytes per word with a cache size that is at least three lines (one for each operand and one for the sum), $Z \geq 3L$, then we have the following estimates for the number of cache misses. In the worse case, due to indirection in the linked list pointers, one could have a cache complexity of $O(4n)$ as each node can cause a cache miss. However, in practice this is not the case. As nodes are allocated one at a time, in order, then they also align in memory. For linked lists (with exponent packing) over the integers and rational numbers, this results in a cache complexity of

$$O\left(1+\frac{32}{8}\frac{4n}{L}\right)=O\left(1+16\frac{n}{L}\right) \text{ and}$$
$$O\left(1+\frac{48}{8}\frac{4n}{L}\right)=O\left(1+24\frac{n}{L}\right),$$

respectively. For alternating arrays over the integers and rational numbers we have,

$$O\left(1+\frac{24}{8}\frac{4n}{L}\right)=O\left(1+12\frac{n}{L}\right) \text{ and}$$
$$O\left(1+\frac{40}{8}\frac{4n}{L}\right)=O\left(1+20\frac{n}{L}\right),$$

respectively. There is a multiplicative factor of 4, as in the worst case the sum has a size equal to twice the input. Figure 3.5 shows the actual number of cache misses occurring for various sizes of operand polynomials encoded as: linked lists without exponent packing, linked lists with exponent packing, and alternating arrays. These results mimic the differences in performance for actual running time.

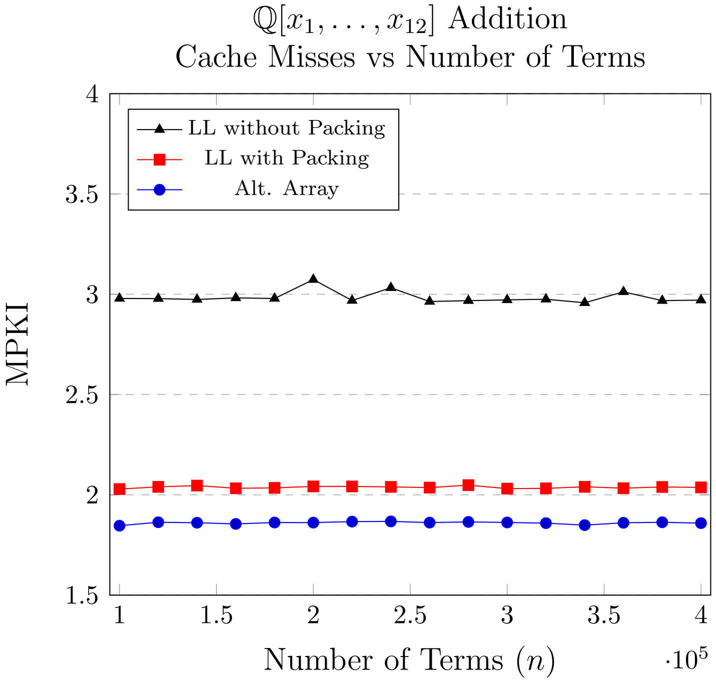


Figure 3.5: Comparing cache misses for rational number polynomial addition where polynomials are encoded as linked lists, with and without exponent packing, and alternating arrays. These polynomials are in 12 variables. The y-axis shows cache misses per one thousand instructions (MPKI).

3.4 Recursive Arrays

The previous alternating array representation can be seen as a *distributed* multivariate representation. That is, it encodes polynomials such as $f \in R[x_1, \ldots, x_v]$. It would be inefficient to traverse and manipulate in a recursive way, that is, viewing the polynomial as essentially univariate with polynomial coefficients: $f \in R[x_2, \ldots, x_v][x_1]$. But, this is the precise structure we need to in order to work with pseudo-division. In order to have an effective data structure to implement a sparse pseudo-division algorithm, we introduce a new polynomial data type, the *recursive array*. This structure is designed to view polynomials in this univariate, recursive way in order to efficiently operate on them within the semantics of pseudo-division.

Recall from Section 2.3, a dense recursive representation. Where coefficients are themselves polynomials and an array of pointers to these polynomials encodes the main recursive polynomial. This representation is repeated in Figure 3.6 for clarity. Here, a polynomial has parts of its data implied by the structure. Namely, the exponents of x_1 for each term are implied by the index of polynomial coefficients, g_i.

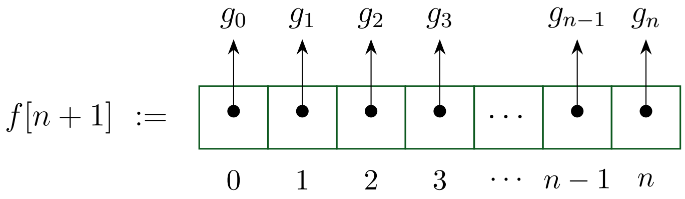


Figure 3.6: Array representation of a dense recursive multivariate polynomial.

If we merge this idea of pointers to polynomial coefficients with the idea of alternating arrays, we get to our idea of a recursive array. Essentially, it is a univariate alternating array, which alternates between coefficients and monomials, just like the general alternating array. It differs by the fact that coefficients are simply pointers to other polynomials (encoded using the general alternating array) and that monomials, since univariate, are encoded by a single integer instead of an exponent vector. However, it is more sophisticated than that. Instead of just arbitrary pointers to polynomial coefficients, these pointers are rather offsets into a single array containing all the polynomial coefficients packed side-by-side. This is best explained by discussing how we convert from a distributed view to a recursive view, re-using data and its structure.

This recursive polynomial representation uses an in-place, very fast conversion between the normal distributed representation and the recursive one. This amounts to minimal overhead and allows a single distributed-view polynomial to be passed around, and those operations needing recursive views can convert very quickly to that view. Of course, an in-place conversion is beneficial to avoid memory movement and reduce the working memory required for the algorithm.

To view the polynomial recursively, we begin by blocking the alternating array representation of the distributed polynomial based on the degrees of the main variable. Each block groups together terms which have equal degree with respect to the main variable. As our polynomials are ordered lexicographically, then all terms are already in order with respect to the degree of the main variable, and, moreover, within a block, all terms are also sorted lexicographically with respect to all of the remaining variables. Because of this, we can create these blocks in-place, without any memory movement, simply by maintaining the offset into the array for the beginning of each block.

Next, we create a secondary alternating array to store these offsets. This array alternates between an exponent of the main variable and a pointer to the original array which is offset to point to the beginning of the block that corresponds to the that main variable exponent. Note that we also store the size of each block. This is convenient when we need to do coefficient arithmetic as those coefficients are themselves polynomials that must know their size to perform arithmetic. In addition, as we traverse the original alternating array to determine where to form partitions, we zero out the degree of the main variable for every monomial. This ensures that the degree of the main variable does not pollute the arithmetic of the polynomial coefficients. Figure 3.7 shows this secondary array structure along with the original array, highlighting the conversion process.

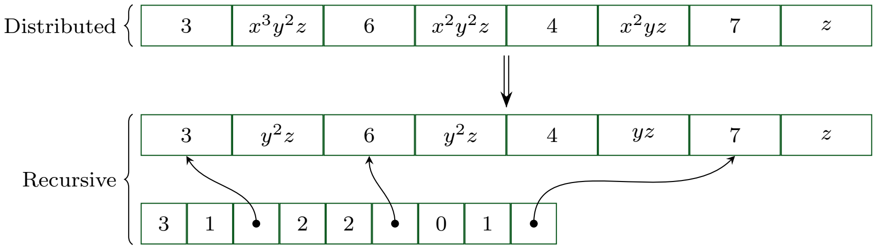


Figure 3.7: An alternating array converted to a recursive array representation, showing the additional secondary array. The secondary array alternates between: (1) exponent of the main variable, (2) size of the coefficient polynomial, and (3) a pointer to the coefficient polynomial which is simply an offset into the original distributed polynomial.

These two alternating arrays together exactly and efficiently represent the recursive view of a polynomial, having coefficients from an arbitrary polynomial ring and univariate monomials. The secondary alternating array requires little additional memory. It will have size proportional to the number of unique values of the degree of the main variable in the distributed polynomial. In practice, with sparse polynomials, this number is quite small. In the absolute worst case, for integer polynomials that are fully dense with respect to the main variable, this secondary array requires $O(\frac{2}{3}n)$ additional space. As the arbitrary-precision coefficients grow larger, or when working with rational number coefficients, this fraction becomes much smaller. This additional space becomes increasingly insignificant as the integers (rational numbers) grow in size, as they always do in pseudo-division calculations (the main algorithm making use of this recursive representation).

Lastly, it also worth noting that these sparse representations (alternating array and recursive array) are used for all of our algorithms, including division and pseudo-division.

Even though (pseudo-)quotients and (pseudo-)remainders are often much more dense than the divisor and dividend, since we are working with multivariate polynomials, a dense representation would nonetheless grow exponentially with the number of variables and, therefore, our sparse representation is still worthwhile and efficient. We will see these data structures in use as we implement optimized arithmetic in the next chapter.

Chapter 4

Polynomial Arithmetic

By the fundamental and necessary nature of arithmetic functions we are greatly concerned with the performance of our algorithms. Thus, we must be concerned with memory usage, data locality, and cache complexity. With the ever-increasing gap between processor speeds and memory-access time, our implementation techniques focus on memory usage and management. Our implementations effectively traverse memory while making use of memory-efficient data structures with good data locality. The implementations techniques look to minimize memory usage and optimize locality in a cache-oblivious way. We will put to use the polynomial data structures discussed in the previous chapter. In particular, we use the alternating array representation (see Section 3.3), which is the best performing data structure in terms of memory usage and locality.

Using the sparse arithmetic algorithms first introduced by Johnson in [42], (see Section 2.4), we will discuss the effective implementations of multiplication (Section 4.2) and division (Section 4.3). We also extend the ideas of Johnson to a new algorithm for sparse multivariate pseudo-division (Section 4.4). Since the algorithm for addition (subtraction) has already been fully specified (Algorithm 2.3) and its implementation follows precisely from the algorithm, we will not further that discussion. However, we do note a variation of the algorithm for *in-place* addition and subtraction, that is, by reusing the structure of one of the operands to hold the result. This is described in Section 4.1.

4.1 In-place Addition and Subtraction

An in-place algorithm suggests that the result is stored back into the same data structure as one of operands (or the only operand). For our purposes, because the actual amount of memory used for our polynomial representations is rather small relative to the available memory, we are interested in in-place operations only if they improve the performance of an algorithm with respect to time. Generally speaking, in-place algorithms require more operations and more movement of data than out-of-place alternatives.

For example, in-place merge sort has been a topic of discussion for sorting algorithms for quite some time. In-place variants have been studied for the sake of minimizing the auxiliary space required by the algorithm and for minimizing the space to store the result. However, these implementations run 25-200% *slower* than an out-of-place implementation [41, 47]. As sparse polynomial addition is essentially one step of merge sort, it seems unlikely that we can gain performance using an in-place scheme.

However, this is not the case. Our in-place addition is roughly twice as fast as the out-of-place variation. Figure 4.2 shows out-of-place addition vs its in-place counterpart for various polynomial sizes with varying coefficient sizes. Clearly, in-place is winning, with a speed up factor of up to 3, and it continues to improve with larger data. Yet, from the analogy to merge sort, it should be slower. Why is it faster?

The simple answer is due to how we use GMP arbitrary-precision numbers. These numbers are broken into two parts, we call them the head and the tree. The head is the main structure which is used directly by users of the GMP library. Indeed, the head is what we store in the alternating array representation as a coefficient. The head contains metadata about the tree, such as size and allocation space. The tree is where the actual data encoding the arbitrary-precision numbers are stored. The head of each number holds a pointer to a unique tree. See Figure 4.1 which highlights the pointers to GMP trees within the alternating array structure. Naturally, the head requires relatively less memory than the tree, especially as the size of the arbitrary-precision numbers grow. Further details on the internals of GMP are specified in its user manual, [37, Section 16].

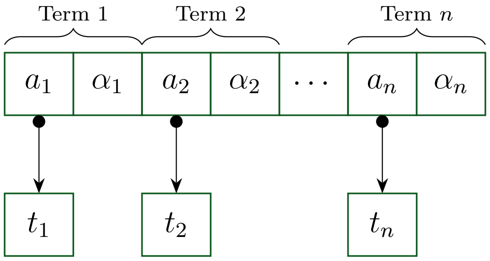


Figure 4.1: Alternating array representation showing GMP trees as t_1, t_2, t_n.

The difference between our in-place addition and out-of-place addition is the manipulation of GMP numbers. Since strict in-place merge sort is both asymptotically slower and practically slower than out-of-place, our in-place addition still uses a complete auxiliary polynomial to (temporarily) store the sum. The difference comes about from using GMP coefficients in a smart way. For an out-of-place addition, we would need to create a new GMP number, allocating a tree in the process, and store the result of the coefficient addition into the new tree (or simply copy the coefficient for non-like-terms into the new tree). This memory movement is expensive. For the in-place addition we rather re-use the trees of one of the operands to store the sums in-place. The head of each GMP number is newly created and its metadata copied to the new GMP number, but the pointer to the underlying tree is also copied instead the data within the tree. Hence, we save the time required to both allocate a new tree and to fill it. Moreover, GMP arithmetic functions have improved performance when they are done in-place [37].

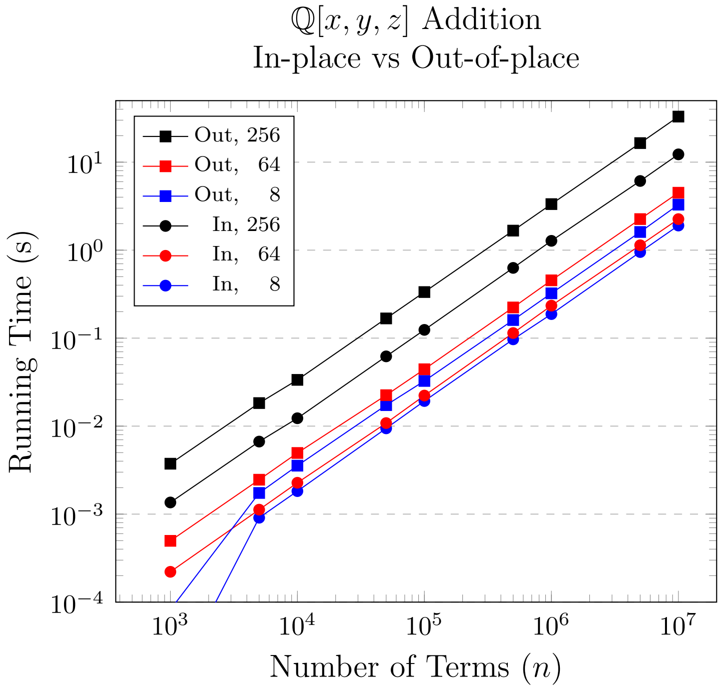


Figure 4.2: Comparing in-place and out-of-place polynomial addition. Rational number polynomials in 3 variables are added together for various sizes and for various coefficient sizes. The number of bits needed to encode the coefficients of the operands are shown in the legend. Notice this is a log-log plot.

Clearly our so-called in-place addition (subtraction) is much more effective at using memory and has improved performance because of it. In-place arithmetic is put to use in pseudo-division to once again reduce the impact of polynomial arithmetic on memory and to improve the performance of pseudo-division itself. See Section 4.4 for this discussion.

4.2 Multiplication

The movement and traversal of data in memory for multiplication is of far greater concern than for addition. Given two operands, a and b, with n_a and n_b terms, respectively, their product has upwards of $n_a n_b$ terms, compared to their sum which may have $n_a + n_b$ terms. This relatively large size of the result compared to the size of the input requires us to be particularly concerned with memory management within the algorithm. Multiplication follows addition (and merge sort) with the idea of producing terms in order, combining like-terms as soon as possible, and accessing data linearly. This is accomplished using the the n-way merge as described in Section 2.4. We reuse the notation of that section here.

For a practical implementation of this n_a-way merge, we must be smart in how we choose to find s for the maximum term $\alpha_a + \beta_{f_s}$. We can use a *priority queue* to effectively implement this selection. Priority queues are effective data structures for retrieving maximum (or minimum) elements from a continuously updating data set. One standard implementation of a priority queue is known as a *binary-tree heap* (simply called *heap* in most literature, as this is the typical heap implementation). Heaps encode a binary tree in some partially sorted order known as the *heap order* or the *heap property*. Of importance are two facts: the root of the binary tree is the maximum element, allowing for instant checking of the maximum value, and, extracting and inserting elements into the heap requires reshuffling elements to maintain the heap property. Due to the structure imposed by the heap property, this reshuffling is fast, requiring only $O(\lg n)$ operations, compared to $O(n)$ for maintaining a typical sorted array. Figure 4.3 shows an example binary-tree heap of integers. Notice the heap property: that all parent nodes are larger than their child nodes. We refer the readers to [70, Chapter 2] for further information on heaps and effective data structures.

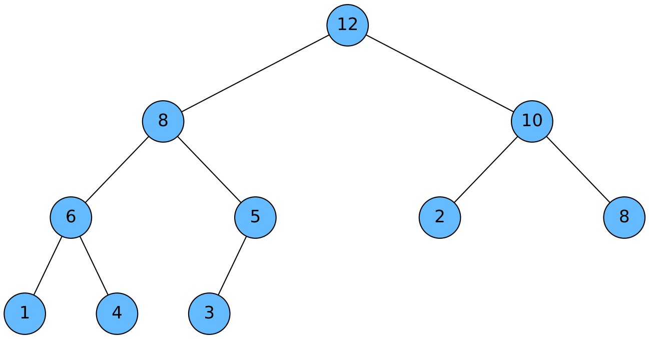


Figure 4.3: An example heap of integers, showing the heap property satisfied.

Hence, we encode our multiplication "streams" as a heap. In particular, we store the heads of each stream in the heap. When we remove an element from the heap, we simply insert its successor, the new head, into the heap (if one exists, otherwise we do nothing and the heap size is permanently reduced). In this way, the heap is very efficient in terms of memory usage and memory traversal. Details on the heap implementation and optimizations are explored in the next section.

We adapt a new algorithm from our previous polynomial multiplication algorithm (Algorithm 2.4) to include our operations with the heap. This gives the so-called heap multiplication algorithm (Algorithm 4.1). In this algorithm, the management of the heap for computing product terms requires a number of specialized functions. We provide here a simplified interface consisting of four functions. **heapInitialize**(a, B_1) initializes the heap by initiating n_a streams, where the head of the i-th stream is $A_i \cdot B_1$. Each of these heads are inserted into the heap. **heapInsert**(A_i, B_j) adds the product of the terms A_i and B_j to the heap [1]. **heapPeek**$()$ gets the exponent vector γ of the top element in the

[1] Note that the heap need not actually store product terms but can simply store the indices of the

heap and the stream index s from which the product term was formed. **heapExtract**() removes the top element of the heap, providing the product term.

Algorithm 4.1 heapMultiplyPolynomials(a,b)
$a, b \in R[x_1, \ldots, x_v]$, $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i}$, $b = \sum_{j=1}^{n_b} b_j X^{\beta_j}$;
return $c = a \cdot b = \sum_{k=1}^{n_c} c_k X^{\gamma_k} \in R[x_1, \ldots, x_v]$

```
1: if $n_a = 0$ or $n_b = 0$ then
2:     return 0
3: $k := 1$; $C_1 := 0$
4: $\gamma := \alpha_1 + \beta_1$ #Maximum possible value of $\gamma$
5: heapInitialize($a, B_1$)
6: for $i = 1$ to $n_a$ do
7:     $f_i$:=1
8: while $\gamma > -1$ do #$\gamma = -1$ when the heap is exhausted
9:     $(\gamma, s)$ := heapPeek() #Get degree and stream index of the top of the heap
10:    if $\gamma \neq \deg(C_k)$ and $\operatorname{coef}(C_k) \neq 0$ then
11:        $k := k + 1$
12:        $C_k := 0$
13:    $C_k := C_k +$ heapExtract()
14:    $f_s := f_s + 1$
15:    if $f_s \leq n_b$ then
16:        heapInsert($A_s, B_{f_s}$)
17: end
18: return $c = \sum_{\ell=1}^{k} C_\ell = \sum_{\ell=1}^{k} c_\ell X^{\gamma_\ell}$
```

There are very clear similarities between the heap multiplication algorithm and the standard multiplication. Indeed, it is only modified to more efficiently select the next product term from the n_a-way merge. We conclude this section with the following proposition, that the heap multiplication algorithms terminates and is correct.

Proposition 4.1 *Algorithm 4.1 terminates and is correct.*

Proof: Let $a, b \in R[x]$ for R a commutative ring with identity. If either $n_a = 0$ or $n_b = 0$ then $a = 0$ or $b = 0$, in which case $c = 0$ and we are done. Otherwise, $c \neq 0$ and we initialize the heap with n_a pairs (A_i, B_1), $i = 1, \ldots, n_a$, we initialize the stream element indices f_i to 1, and we set $C_1 = 0$. We initially set $\gamma = \alpha_1 + \beta_1$, the maximum possible for polynomials a and b, as they are sorted in decreasing order. This is slightly redundant but serves to enter the loop for the first time.

Since we have added the top element of each stream to the heap, the remaining elements to be added to the heap are all less than some element in the heap. The initial **heapPeek**() sets $\gamma = \alpha_1 + \beta_1$ and $s = 1$. Since C_1 was initially set to 0, $C_k = 0$, so the condition on line 10 is met, but not that of line 11 so we move to line 14. Lines 14 through 17 extract the top of the heap, add it to C_k (giving $C_1 = A_1 B_1$), and insert the next element of the first stream into the heap. This value of C_1 is correct.

Subsequent passes through the loop must either (1) add to C_k a term of $\deg(C_k)$

two factors, with the product only computed when elements of the heap are removed. This strategy is needed for pseudo-division (Section 4.4) where the quotient terms are updated over the course of the algorithm.

(if one exists) or if $C_k = 0$, we add to C_k the next greatest element or (2) increase k and begin building the C_{k+1} term when $C_k \neq 0$ (since for sparse polynomials we store only non-zero terms). Case (1) is handled by line 13 if $\gamma = \deg(C_k)$ or lines 10 and 13 if $C_k = 0$. Case (2) is handled by lines 10-13, since $\gamma \neq \deg(C_k)$ and $C_k \neq 0$ by assumption. Hence, the behaviour is correct.

The loop terminates because there are only n_b elements in each stream, and lines 15-16 only add an element to the heap if there is a new element to add, while every iteration of the loop always removes an element from the heap at line 13. □

Consider using our alternating array representation with integer coefficients to encode the polynomials in Algorithm 4.1. We can then estimate this algorithm's cache complexity. Following the previous assumptions on GMP coefficients (see Hypothesis 3.1) let the product coefficients also be small enough to be directly stored. Let us also assume that the entire heap fits in cache; this is a reasonable assumption due to the relatively small amount of auxiliary space needed to encode it (see the next subsection). Then cache misses occur by:

(i) iterating through a (**heapInitialize**()),
(ii) iterating through b (**heapInsert**()), and
(iii) appending resulting product terms to c ($C_k := 0$, $C_k := C_k +$ **heapExtract**()).

Assuming a word size of 8 bytes then the number of cache misses as a result of (i) is $O(1 + 3n_a/L)$ while the number as a result of (ii) is $O(1 + 3n_b/L)$. In the worst case c will have a number of terms equal to $n_a \cdot n_b$. Hence, the number of cache misses as a result of (iii) is $O(1 + 3n_an_b/L)$. Then, assuming the heap fully fits in the cache, the cache complexity of HEAPMULTIPLYPOLYNOMIALS is

$$O\left(1 + \frac{3n_an_b}{L} + \frac{3n_a}{L} + \frac{3n_b}{L}\right)$$

We can relax the assumption that the heap fully fits in cache using the following generalizations of binary heap performance. Say the heap contains N elements, then:

(i) an **heapInsert**() accesses $O(\lg(N))$ heap elements,
(ii) an **heapExtract**() accesses $O(\lg(N))$ heap elements, and
(iii) a **heapPeek**() is free.

Then, in addition to cache misses resulting from iterating through a, b, and c, each **heapExtract**() and **heapInsert**() can also incur $O(\lg(N)/L)$ cache misses per operation. Since one extract produces one term of the product, this results in $O(2\lg(N)n_an_b/L)$ cache misses. Moreover, each time a product term is appended to c, it may cause a cache miss as the **heapExtract**() could have evicted the previously cached terms of c. In contrast, iterating through a and b is now encapsulated by the heap operations. Using the fact that the size of the heap is $N = n_a$, then the cache complexity of HEAPMULTIPLY-

Polynomials is

$$O\left(1 + n_a n_b + \frac{3n_a}{L} + \frac{2\lg(n_a) n_a n_b}{L}\right)$$

4.2.1 Implementation

In the previous section we saw a modified algorithm using a heap in order to effectively choose the maximum elements among the multiplication streams. The actual implementation of this heap and algorithm require some tricks to gain as much performance as possible. In this section we present these implementation details.

Our multiplication algorithm, like all our sparse algorithms, is motivated by the need to produce the terms of the result in-order. This need arises for its computational and memory-based advantages. Multiplication can be seen to follow a 3-step coarse algorithm: (1) generate the terms of the result, (2) combine terms with equal monomials, and (3) sort the resulting terms to regain a canonical representation. By producing terms in-order then step (3) can be avoided altogether. Moreover, we can combine like-terms as soon as they are found and minimize the number of intermediate terms created. This reduces the amount of working memory needed for our algorithms and simplifies terms as soon as possible.

The problems of intermediate terms and combining like-terms is far worse in multiplication than addition as the number of product terms can be up to $n_a n_b$. Notice that, as the algorithm is stated, the terms of the left operand, a, are distributed over the terms of b, producing n_a "streams" in the process. We can reorder operands such that a always has fewer terms than b. This is computationally advantageous as the size of our heap is determined by the number of active streams, and, the size of the heap directly impacts the cost of inserting and removing elements from it. Since we only store the heads of each stream in the heap, the preference is for the number of streams to be few and each of them to be long. This is accomplished by choosing a to the smaller of the operands, which we are free to do since multiplication is commutative.

To minimize working space, we do not compute the whole stream in advance, but rather only produce the product of the two terms at the head of a stream. Moreover, this product is computed in two distinct steps. First, the product monomial is computed as the product term is inserted into the heap. This monomial is required for comparisons within the heap. In contrast, the product coefficient is not required for comparisons or choosing the maximum. Thus, we delay the calculation of the product coefficient until we have extracted its corresponding monomial from the heap and are about to commit it to the product polynomial (commit here being only an append to the product polynomial as terms are produced in-order). This improves the data locality for coefficient arithmetic and combining of like-terms as the multiplication of the extracted product term and its addition with like-terms can be done simultaneously.

With this in mind, the implementation varies slightly from the algorithm presented in Section 4.1. Rather than extract one term and insert its successor to the heap right away, we continue to extract terms from that heap as long as they are like-terms and then insert back all successors of these extracted elements at once. This reduces the overall amount of work needed to extract and insert terms from the heap. Clearly, if we extract more frequently than we insert, the number of terms in the heap shrinks. This is only temporary, as we will generally insert the successors of those terms again, but this temporary shrink nonetheless impacts performance for the better.

We must also consider effectively storing the resulting product polynomial. Using our alternating array representation, it is slightly tricky to continuously append to the end of the array, as the heap multiplication algorithm suggests. We must be wary that we do not overflow the array allocation during the append. Moreover, if we do in fact overflow the initial array allocation, then we must reallocate the array, extending its size, resulting in unnecessary memory movement and a waste of processor time. To remedy this, we pre-allocate an array for the product with its maximum possible size, $n_a \cdot n_b$. This minimizes memory movement and reallocation required throughout the computation of appending product terms to the product polynomial. It is reasonable to pre-allocate this maximal size as our inputs are sparse polynomials and so we expect fairly few like-terms to be produced, and therefore, combined. Also, as we have seen in Section 3.3, our polynomials are very efficiently represented and so the maximum $n_a \cdot n_b$ terms still consumes relatively little memory.

Heap Optimizations

The performance of our code is very dependent on the implementation of its data-structures, and in particular, heaps. Aside from coefficient arithmetic, all of the work for multiplying polynomials comes from obtaining the ordering of product terms. Hence, the heap, whose purpose is to produce terms in the required order, takes the majority of the effort of our algorithm. Optimizing the heap is then necessary for performance.

The simplest optimization is with respect to memory management. Say we are multiplying a and b with $n_a < n_b$. Then, during the course of the algorithm, we will have at most n_a streams active at once time. Hence we can pre-allocate space in the heap for exactly n_a elements as that will be the exact number of streams to consider. This is advantageous for two reasons. The first is that, just like pre-allocation of space for the product polynomial, we avoid possible reallocation and useless memory movement when inserting into the heap and overrunning the allocated space. The second is that the insertion code does not need to bother with checking for overflow; we are guaranteed to have enough space in the heap. This completely removes that if-block from our heap insertion code, simplifying it and making it more efficient. This blind assumption is safe for our restricted and well-controlled use of the heap data structure.

Further optimizations for the heap have two focuses. First, to minimize the number of monomial comparisons required to produce terms in sorted order, and second, to minimize the working memory of the heap. Minimizing comparisons should be an obvious improvement because fewer comparisons means more efficient sorting of the product terms. As for reducing the working memory of the heap, it is important to note that a binary-tree heap, as in our implementation, essentially performs random memory access across all of its elements. Memory access is expensive in terms of performance and so minimizing the working memory of the heap minimizes the amount of memory that it needs to traverse. With a small enough working memory the heap can fully fit into cache memory for much cheaper random access.

The first and most simple optimization to reduce the number of monomial comparisons is not within the heap itself but how we use the heap in multiplication. Recall that for each stream in the multiplication, product terms are strictly decreasing within the one stream, $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$. One could say they are strictly decreasing along a "row" of the streams. But notice also that across streams product terms are strictly decreasing with respect to a fixed term of b. That is, $X^{\alpha_i+\beta_j} > X^{\alpha_{i+1}+\beta_j}$. One could say they are strictly decreasing along a "column" of the streams. Hence, we only need to insert the product $X^{\alpha_{i+1}+\beta_j}$ into the heap once $X^{\alpha_i+\beta_j}$ has been removed from it. In general, this has no effect on the heap size since $X^{\alpha_{i+1}+\beta_{j-1}}$ (that is, the element preceding $X^{\alpha_{i+1}+\beta_j}$ in its own stream) is already in the heap and so $X^{\alpha_{i+1}+\beta_j}$ will not yet be inserted as it is not yet the head of its stream. But, we can optimize the initialization of the heap to only insert the first term of a stream, $X^{\alpha_i+\beta_1}$, once the first term of the preceding stream, $X^{\alpha_{i-1}+\beta_1}$ has been extracted. This minimizes the number of terms initially in the heap, allowing for much faster heap insertion and extraction up until all streams are active and $X^{\alpha_{n_a}+\beta_1}$ gets inserted.

The next optimization is applied to how elements are extracted from the heap. Of course, this operation is crucial as extraction is what essentially does the sorting of our product terms. Standard implementations of a binary-tree heap remove the maximum element at the root, swap a leaf element into that newly empty slot, and *sink* that former leaf node to restore the heap property. Sinking here meaning shuffling elements in the heap in order to maintain the heap property. A heap of size n requires $O(2 \log n)$ comparisons and $O(\log n)$ element swaps to restore the heap property using a sink. It was noted in [36] that extraction can be implemented more efficiently, using only $O(\log n)$ comparisons, by removing the root node and then pushing upward (promoting) children to fill the hole. This factor of 2 reduction in comparisons is not an insignificant improvement, especially because extracting from the heap is fundamental to our multiplication algorithm. This same heap improvement was also noted by Monagan and Pearce in [59] for their implementation of integer polynomial arithmetic.

Our final heap optimization is concerned with both minimizing the working memory of the heap and the number of comparisons it needs to operate. The technique of *chaining* can be used to drastically reduce the number of elements in heap at one time. Chaining is a common technique in the implementation of *hash tables* for *collision resolution* (see [70, Chatper 3]). In this scheme, elements found to have the same key in a single hash table are chained together to form a linked list. We modify this scheme for use in a heap. In our case, elements are polynomial terms and their keys are monomials. Therefore, by augmenting our heap with chaining, we obtain a heap of linked lists. Chaining reduces the number of elements in the heap as terms with equal monomials only need to occupy a single data element in the heap. This is represented by the monomial being stored in the heap itself with a pointer to the linked list of corresponding coefficients. With fewer elements in the heap fewer comparisons are needed to maintain the heap property and, moreover, the working space of the heap itself is reduced. As elements are being inserted or extracted, if equal monomials are found, then chains are formed. With increasing density of the operand polynomials, the amount of chaining in the heap increases. As noted in [59], when chained heaps are used for completely dense (no zero terms) polynomial multiplication, the heap size reduces to 1 at all times as all elements chain together.

Chaining is a great advantage for performance. In particular, we are able to extract an *entire chain* for the cost of extracting a single element. Therefore, we are effectively extracting many terms at once from the heap for the cost of a single extraction. It is worth noting that, due to the way a binary-heap is traversed, it is possible that not all elements of equal monomial in the heap are chained together in a single chain. Because of this, one should continue to extract chains until the maximum degree in the heap no longer matches the first one extracted during a single round of extractions. This follows the scheme described in the previous sub-section where like-terms are all extracted at once, combined together, and then their successors are inserted at once.

The final optimization for the product heap involves further reducing the working space of the heap. With chaining, the coefficients of the product terms are already not stored directly in the heap, but they still play a role in overall auxiliary memory needed for the algorithm. With our alternating array representation of polynomials it is very easy to directly index the operand polynomials to access the appropriate coefficient. Thus, our heap only stores the *indices* of the operand coefficients which together form the coefficient of a particular product term. This reduces the memory required for each product coefficient in the heap from 32 bytes (in the case of rational number coefficients) down to 8 bytes. Similar schemes using pointers to coefficients have been examined in [57, 59] but indices are even more succinct than pointers. Figure 4.4 shows a chained heap along with the idea of storing indices in place of the coefficients.

4.2.2 Experimentation

In the past two sections we introduced multiplication along with many of its implementation details and optimizations. Moreover, we are claiming *high performance* implementations of our arithmetic, hence we must, and do, show this with some nice results.

We compare our implementation against Maple for both integer polynomials and rational number polynomials. Over the past 10 years or so, Maple has become the leader in integer polynomial arithmetic thanks to the extensive work of Monagan and

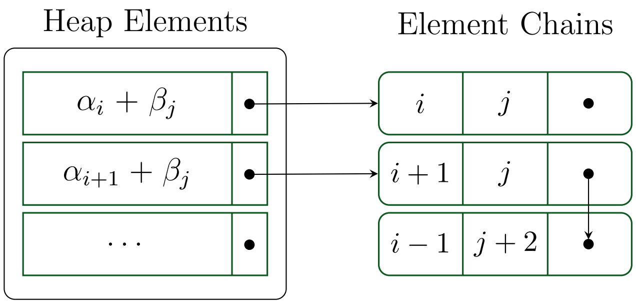


Figure 4.4: A heap of product terms, showing element chaining and index-based storing of coefficients. In this case, terms $A_{i+1} \cdot B_j$ and $A_{i-1} \cdot B_{j+2}$ have equal monomials and are chained together.

Pearce [56–59]. Benchmarks in these papers provide clear indication that their implementation outperforms many other computer algebra systems including: Trip, Magma, Singular, and Pari. Moreover, other common systems like FLINT [39] and NTL [71] provide only univariate polynomial implementations, meaning the comparison against our multivariate implementation would be unfair. Therefore, we compare our implementations against the leading high-performance implementation that is provided by Maple in particular, Maple 2017.

We begin with some preliminaries. In particular, we wish to quantify *sparsity* since we are interested in sparse polynomials. For multivariate polynomials, the notion of sparsity is very difficult to quantify. For univariate polynomials, sparsity is easily defined as the maximum degree difference between any two successive non-zero terms in a polynomial. However in the multivariate case, and in particular using lex ordering, there are infinitely many polynomial terms between, say, x and y, in the form of y^i. Although sparsity is not so easily defined for multivariate polynomials, we propose the following adaptation of the univariate case to the multivariate one, inspired by Kronecker substitution [32, Section 8.4].

Let $f \in R[x_1, \ldots, x_v]$ be non-zero and define $r = \max(\deg(f, x_i), 1 \leq i \leq v) + 1$. Then, every exponent vector $(e_1, \ldots, e_v)$ in f can be viewed as a integer in a radix-r representation, $e_1 + e_2 r + \cdots + e_v r^{v-1}$. Viewing any two successive polynomial terms in f as integers in this radix-r representation, say r_i and r_{i+1}, we call the *sparsity* of f the smallest number which is larger than the maximum value of $r_i - r_{i+1}$, for $1 \leq i < n_f$. Note, that using this definition, a sparsity of two is then fully dense, as the difference between any two exponent vectors is 1.

For our experiments, sparse polynomials were randomly generated using the following parameters: number of variables v, number of terms n, sparsity s, and maximum number of bits in any coefficient. Then, exponent vectors are generated as radix r representations with v digits and r computed as $\lfloor \sqrt[v]{s \cdot n} \rfloor$.

Throughout all benchmarks presented in this section, and those following, our benchmarks were collected on a machine with an Intel Xeon X560 processor at 2.67 GHz, 32KB L1 data cache, 256KB L2 cache, 12288KB L3 cache, and 48GB of RAM.

We begin by comparing the effect that our heap optimizations have on our algorithm. In particular we compare with and without heap chaining (Table 4.1).

		Without Chaining		With Chaining	
n	s	Time (s)	Comparisons	Time (s)	Comparisons
250	2	0.05363	3,364,390	0.02166	388,339
	50	0.05563	3,665,990	0.04878	2,930,800
	100	0.05502	3,681,180	0.05095	3,073,490
	250	0.05201	3,701,710	0.05366	3,200,640
500	2	0.21764	3,364,390	0.07761	388,339
	50	0.23372	3,665,990	0.18502	2,930,800
	100	0.23022	3,681,180	0.19975	3,073,490
	250	0.22604	3,701,710	0.21271	3,200,640
1000	2	0.96485	15,339,500	0.30952	1,375,600
	50	1.06024	16,551,900	0.77558	12,517,900
	100	1.03442	16,656,000	0.82393	13,063,000
	250	1.01055	16,730,400	0.89683	13,612,100

Table 4.1: Comparing the effect of heap implementations with and without chaining on polynomial multiplication. We fix the number of variables, v to 3, and the the coefficient bound to 6. Sparsity, s, and number of terms, n, of the operand polynomials vary as indicated.

Next, we look at the implementation of our integer polynomials. In Figure 4.5 we compare our implementation to Maple for various levels of sparsity and number of terms, fixing the coefficient bound to 128. It is for integer polynomials that Maple has a highly-optimized implementation. In all cases except dense multiplication, BPAS performs favourably. This is expected, really, because our algorithms are designed specifically for sparse polynomials. It is likely that Maple automatically chooses a dense multiplication algorithm (such as Karatsuba or Toom-Cook, which are asymptotically faster than any known sparse algorithm) when the polynomial is dense, as opposed to the sparse algorithms of Monagan and Pearce.

In Figure 4.6 we compare our implementation of rational number polynomial multiplication to that of Maple. In this case, rational number polynomials did not receive the same high performance treatment as integer polynomials. Hence, we can see the drastic differences between our implementation and the one which currently exists in Maple. We note that, in our implementation, the more sparse the polynomial, the faster multiplication occurs. This is due to fewer like-terms being created and thus having to perform much less coefficient arithmetic.

In order to reiterate the effects of choosing a proper data structure, actual cache miss rates for multiplication are presented in Figure 4.7. This plot compares multiplication using linked lists without exponent packing, linked lists with exponent packing, and alternating arrays. We note that the downward slope of the cache miss rates are due to more work being done *per polynomial term* in the multiplication, therefore there are more

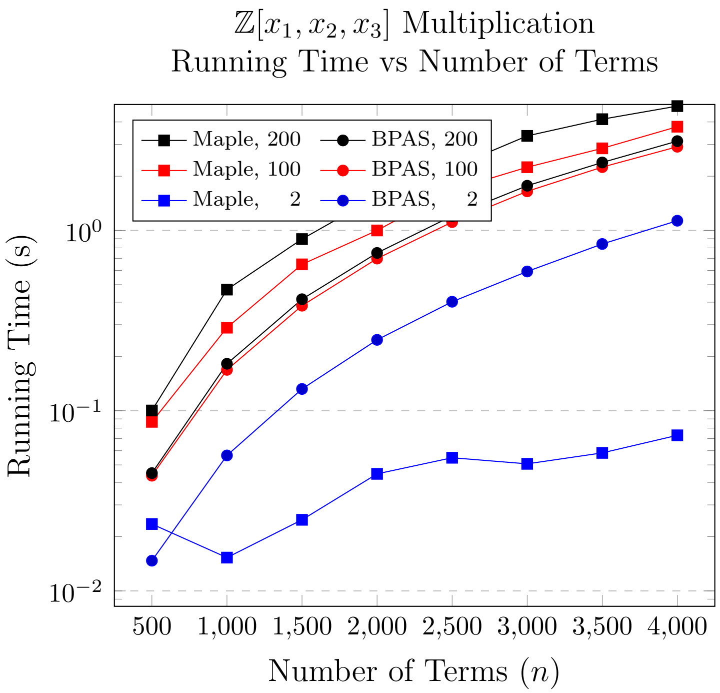


Figure 4.5: Comparing multiplication of integer polynomials. The number of variables is fixed at 3, and the coefficient bound at 128. The number of terms vary on the x-axis, while the sparsity varies as noted in the legend.

instructions operating on each polynomial term, leading to a smaller ratio of cache misses per thousand instructions. Moreover, we note that the alternating array MPKI is slightly higher only in the sense of *ratios*. Alternating arrays result in much fewer instructions executed throughout the program and thus a higher ratio. The number of alternating array cache misses is actually much less than that of linked lists with exponent packing.

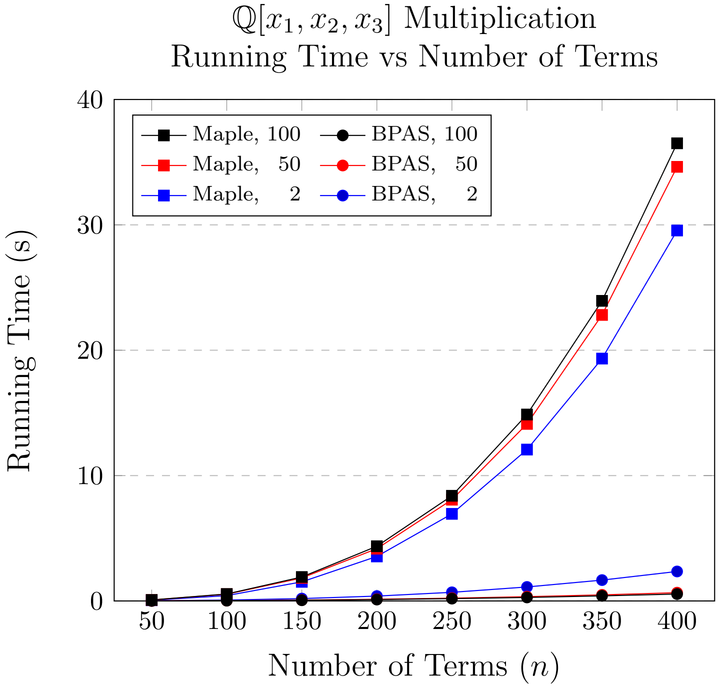


Figure 4.6: Comparing multiplication of rational number polynomials. The number of variables is fixed at 3, and the coefficient bound at 128. The number of terms vary on the x-axis, while the sparsity varies as noted in the legend.

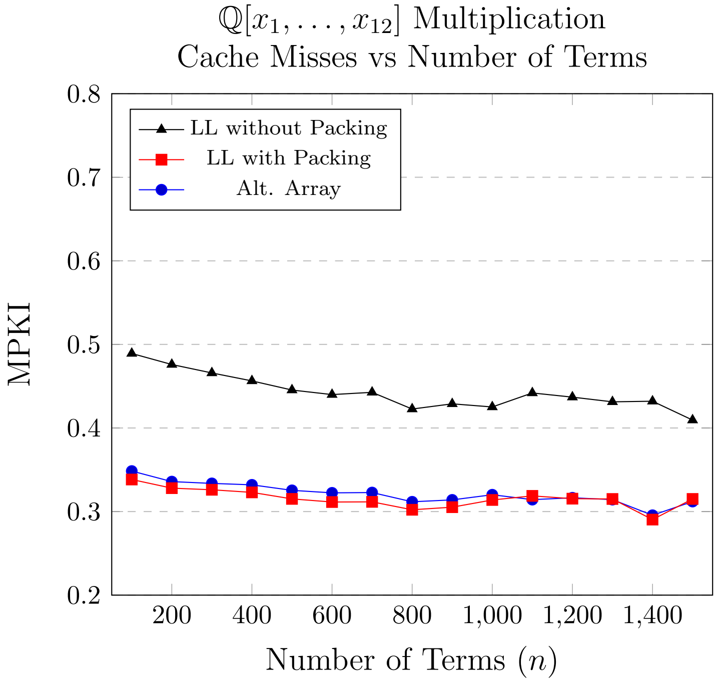


Figure 4.7: Comparing cache misses for rational number multiplication in polynomials encoded as linked lists, with and without exponent packing, and alternating arrays. These polynomials are in 12 variables. The y-axis shows cache misses per one thousand instructions (MPKI).

4.3 Division

We now consider the problem of multivariate division, where the input polynomials are $a, b \in \mathbb{K}[x_1, \ldots, x_v]$, with $b \neq 0$ being the divisor and a the dividend. We assume that $\mathbb{K}$ is a field. Hence, $\{b\}$ is a Gröbner basis of the ideal it generates. Thus, we can specify division just as one would for reduction in the sense of Gröbner bases (see Definition 2.4). Note that this division, and its associated algorithms, is well-defined for $a, b, q, r \in D[x_1, \ldots, x_v]$ for an arbitrary integral domain D provided that b is monic (or at least its leading term divides all coefficients of a). We assume here however, for ease of discussion, that the coefficients are elements of a field.

Division presents a more tricky problem than multiplication in the sense of producing terms in order. We must compute terms of both the quotient and remainder in order, while simultaneously producing terms of the product qb in order. We must also produce these product terms all the while q is being updated throughout the algorithm. This is not so simple, especially in implementation.

As a starting point, we do not yet consider the heap variation of division. We begin by presenting what is essentially a small extension of the univariate division algorithm (Algorithm 2.1) to multivariate polynomials. It relies on repeated multiplication of quotient with divisor, and subtracting this product from the current remainder. This algorithm is presented in Algorithm 4.2.

In this algorithm, the quotient and remainder, q and r, are computed term by term by computing $\tilde{r} = \mathrm{lt}(a - qb - r)$ at each step. This works for multivariate division by deciding whether $\tilde{r}$ should belong to the remainder or the quotient at each step. If $\mathrm{lt}(b) \mid \tilde{r}$ then we perform this division and obtain a new quotient term. Otherwise, we obtain a new remainder term. In either case, this $\tilde{r}$ *was* the leading term of the expression $a - qb - r$ and now either belongs to q or r. Therefore, In the next step, the old $\tilde{r}$ was added to either q or r and thus will now cancel itself out, resulting in a new leading term to be produced from the expression $a - qb - r$. This new leading term is strictly less (in the sense of its monomial) than the preceding $\tilde{r}$.

Algorithm 4.2 DIVIDEPOLYNOMIALS(a,b)

$a, b \in \mathbb{K}[x_1, \ldots, x_v]$, $b \neq 0$; return $q, r \in \mathbb{K}[x_1, \ldots, x_v]$ such that $a = qb + r$ where r is reduced with respect to the Gröbner basis $\{b\}$.

```
$q := 0$; $r := 0$
while ($\tilde{r} := \mathrm{lt}(a - qb - r)) \neq 0$ do
    if $\mathrm{lt}(b) \mid \tilde{r}$ then
        $q := q + \tilde{r}/\mathrm{lt}(b)$
    else
        $r := r + \tilde{r}$
end
return $(q, r)$
```

Proposition 4.2 *Algorithm 4.2 terminates and is correct.*

Proof: Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[x_1, \ldots, x_v]$. If $deg(b) = 0$ holds, then the divisibility test on line 3 always passes, all generated terms go to the quotient, and we get have a remainder of zero throughout the algorithm. Essentially, this is a scalar multiplication by b^{-1}. Then of course $r = 0$ is reduced and we are done.

We initialize $q, r = 0$. It is enough to show that for each iteration of the loop, the degree of $\tilde{r}$ strictly decreases. It follows from the axioms of a term order that $\tilde{r}$ becomes zero after finitely many iterations.

From now on, we assume $\deg(b) > 0$. We denote the values of the variables of Algorithm 4.2 on the i-th iteration by superscripts. For each i, depending on whether or not $\mathrm{lt}(b) \mid \tilde{r}^{(i)}$ holds, we have two possibilities: (1) $Q_\ell = \tilde{r}^{(i)}/B_1$, where Q_ℓ is a new quotient term; or (2) $R_k = \tilde{r}^{(i)}$, where R_k is a new remainder term. Notice in case 1 we update the quotient term so $r^{(i+1)} = r^{(i)}$. In case 2 we update the remainder term so $q^{(i+1)} = q^{(i)}$. Suppose that $\tilde{r}^{(i)}$ has just been used to compute a term of q or r, and we now look to compute $\tilde{r}^{(i+1)}$. Depending on whether or not $\mathrm{lt}(b) \mid \tilde{r}^{(i)}$ we have:

Case 1: $\mathrm{lt}(b) \mid \mathrm{lt}(a - q^{(i)}b - r^{(i)})$ and $Q_\ell = \tilde{r}^{(i)}/B_1$

$$\begin{aligned}
\tilde{r}^{(i+1)} = \mathrm{lt}(a - q^{(i+1)}b - r^{(i+1)}) &= \mathrm{lt}(a - ([q^{(i)} + Q_\ell]b) - r^{(i+1)}) \\
&= \mathrm{lt}(a - ([q^{(i)} + Q_\ell]b) - r^{(i)}) \\
&= \mathrm{lt}(a - (q^{(i)}b + Q_\ell b + r^{(i)})) \\
&= \mathrm{lt}(a - (q^{(i)}b + r^{(i)} + (\tilde{r}^{(i)} - \tilde{r}^{(i)}) + Q_\ell b)) \\
&= \mathrm{lt}(a - (q^{(i)}b + r^{(i)} + \tilde{r}^{(i)} + Q_\ell b - \tilde{r}^{(i)})) \\
&= \mathrm{lt}(a - (q^{(i)}b + r^{(i)} + \tilde{r}^{(i)} + Q_\ell(b - B_1)) \\
&= \mathrm{lt}([(a - q^{(i)}b - r^{(i)}) - \tilde{r}^{(i)}] - [Q_\ell(b - B_1)]) \\
&< \mathrm{lt}(\tilde{r}^{(i)}) = \tilde{r}^{(i)}
\end{aligned}$$

In the second last line notice that, since $\tilde{r}^{(i)} = \mathrm{lt}(a - q^{(i)}b - r^{(i)})$, then their difference must have a leading term strictly less than $\tilde{r}^{(i)}$. Also, the expression $Q_\ell(b - B_1)$ has leading term $Q_\ell B_2$ which is strictly less than $\tilde{r}^{(i)} = Q_\ell B_1$, by the ordering on the terms of b. Hence $\tilde{r}^{(i+1)}$ is strictly less than $\tilde{r}^{(i)}$.

Case 2: $\mathrm{lt}(b) \nmid \mathrm{lt}(a - q^{(i)}b - r^{(i)})$ and $R_k = \tilde{r}^{(i)}$

$$\begin{aligned}
\tilde{r}^{(i+1)} = \mathrm{lt}(a - q^{(i+1)}b - r^{(i+1)}) &= \mathrm{lt}(a - q^{(i)}b - (r^{(i)} + R_k)) \\
&= \mathrm{lt}((a - q^{(i)}b - r^{(i)}) - \tilde{r}^{(i)}) \\
&< \mathrm{lt}(\tilde{r}^{(i)}) = \tilde{r}^{(i)}
\end{aligned}$$

Similarly, $\tilde{r}^{(i)} = \mathrm{lt}(a - q^{(i)}b - r^{(i)})$, thus their difference must have a leading term strictly less than $\tilde{r}^{(i)}$.

The loop therefore terminates. The correctness is implied by the condition that $\tilde{r} = 0$ at the end of the loop together with the fact that the all of the terms in r satisfy the condition that $\mathrm{lt}(b) \nmid R_k$ by construction. □

The multivariate division algorithm is essentially just polynomial multiplication in the form of $q \cdot b$ and polynomial subtraction. We can then use a heap to keep track of this quotient-divisor product. However, this must be treated as a special case of heap multiplication. The major difference from multiplication, where all terms of both factors are known from the start, is that q is computed as the algorithm proceeds, which forces q to be the multiplier and b the multiplicand. In terms of the wording of the multiplication algorithm, we distribute q over b, producing "streams" which are formed from a single term of q, and the head of the steam moves along b. By having q in this position, it becomes relatively easy to add new streams as new terms of q are computed. One crucial difference is that each stream does not start with $Q_\ell B_1$. Rather, the steam begins at $Q_\ell B_2$ since the product term $Q_\ell B_1$ is cancelled out by construction.

The management of the heap to compute the product qb uses several of the functions described for Algorithm 4.1. Namely, **heapPeek**(), **heapInsert**, and **heapExtract**(). However, **heapExtract**() is modified slightly from its definition in multiplication. It removes the top element of the heap *and* inserts back the next element of the stream (if there is a next) from which the top element came. In this algorithm we use δ to be the exponent of the top term in the heap of $q \cdot b$. If the heap is empty, we let $\delta = (-1, 0, \ldots, 0)$, which will be less than any exponent of any polynomial term on account of the first element being -1. We therefore abuse notation and write $\delta = -1$ for an empty heap.

Algorithm 4.3 heapDividePolynomials(a,b)

$a, b \in \mathbb{K}[x_1, \ldots, x_v]$, $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i} = \sum_{i=1}^{n_a} A_i$, $b \neq 0 = \sum_{j=1}^{n_b} b_j X^{\beta_j} = \sum_{j=1}^{n_b} B_j$;
return $q, r \in \mathbb{K}[x_1, \ldots, x_v]$ such that $a = qb + r$ where r is reduced with respect to the Gröbner basis $\{b\}$.

```
$(q, r, l) := 0$
$k := 1$
while ($\delta :=$ heapPeek()) $> -1$ or $k \leq n_a$ do
    if $\delta < \alpha_k$ then
        $\tilde{r} := A_k$
        $k := k + 1$
    else if $\delta = \alpha_k$ then
        $\tilde{r} := A_k -$ heapExtract()
        $k := k + 1$
    else
        $\tilde{r} := -$ heapExtract()
    if $B_1 \mid \tilde{r}$ then
        $\ell := \ell + 1$
        $Q_\ell := \tilde{r}/B_1$
        $q := q + Q_\ell$
        heapInsert($Q_\ell, B_2$)
    else
        $r := r + \tilde{r}$
end
return $(q, r)$
```

Proposition 4.3 *Algorithm 4.3 terminates and is correct.*

Proof: Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[x_1, \ldots, x_v]$ with $\deg(b) > 0$. If $b \in \mathbb{K}$ then this degenerative case is simply a scalar multiplication by b^{-1}. And proceeds as in Proposition 4.2. Then $r = 0$ is reduced and we are done.

We initialize $q, r = 0$, $k = 1$ (index into a), $\ell = 0$ (index into q), and $\delta = -1$ (heap empty condition) since the heap is initially empty. The key modification of Algorithm 4.2 to reach Algorithm4.3 is to use terms of qb from the heap to compute $\tilde{r} = \mathrm{lt}(a - qb - r)$. This requires tracking three cases: (1) $\tilde{r}$ is an uncancelled term of a; (2) $\tilde{r}$ is a term of the difference $(a - r) - (qb)$; and (3) $\tilde{r}$ is a term of $-qb$ such that all remaining terms of $a - r$ are smaller in the term order.

Let $a_k X^{\alpha_k} = A_k$ be the greatest uncancelled term of a. Then, the three cases correspond to conditions on the ordering of δ and α_k. The term $\tilde{r}$ is an uncancelled term of a (case 1) either if the heap is empty (indicating that no terms of q have yet been computed or all terms of qb have been extracted), or if $\delta > -1$ but $\delta < \alpha_k$. In either of these two situations $\delta < \alpha_k$ holds and $\tilde{r}$ is chosen to be A_k. The term $\tilde{r}$ is a term of the difference $(a - r) - (qb)$ (case 2) if both A_k and the top term in the heap have the same exponent vector ($\delta = \alpha_k$). Lastly, $\tilde{r}$ is a term of $-qb$ (case 3) whenever $\delta > \alpha_k$ holds.

Algorithm 4.3 uses this observation to compute $\tilde{r}$ by adding conditional statements to compare the components of δ and α_k. Terms are only extracted from the heap when $\delta \geq \alpha_k$ holds, and thus we "consume" a term of qb. Simultaneously, when a term is extracted from the heap, the next term from the given stream, if there is one, is added back to the heap (defined behaviour of **heapExtract**()). The updating of q and r with the new leading term $\tilde{r}$ is almost identical to Algorithm 4.2, except that for when we update the quotient, was also initialize a new stream with Q_ℓ in the multiplication of $q \cdot b$. This stream is initialized with a head of $Q_\ell B_2$ because $Q_\ell B_1$, by construction, cancels a unique term of the expression $a - qb - r$.

In all three cases, either the quotient is updated or the remainder is updated. By Proposition 4.2, together with the case discussion of δ and α_k, the leading term of $a - qb - r$ is strictly decreasing for each loop iteration. □

Using the same assumptions as multiplication, we can analyze the cache complexity of Algorithm 4.3. However, since in our sparse representation the dividend and divisor are not parameterized by their degrees, then the number of terms which will appear in the quotient and remainder are not known. We denote the number of terms which will result in q and r together as N in order to estimate the cache complexity without any other assumptions on dividend or divisor. For division, cache misses occur by:

(i) iterating through a,
(ii) iterating through b,
(iii) iterating through q and r, and
(iv) heap element accesses through insertion and extraction.

Notice that a is not operated on as part of the heap. Hence, terms of a may fully be evicted from the cache by the heap operations. This is unlikely due to the relatively small encoding of the heap, but in the worst case each access to A_k in the algorithm will result in a miss. Hence, the number of cache misses from iterating through a is $O(n_a)$. Iterating through r is similar, leading to $O(N)$ cache misses in the worst case. The size of the quotient dictates the size of the heap in division and the number of extractions is equal to the size of the quotient plus the size of the remainder. Hence, the number of cache misses as a result of heap operations is $O(3\lg(N)N/L)$. We have a factor of 3 as each extract automatically performs another insert along with the explicit insert for each new quotient term. Assuming a cache size of $Z \geq 3L$, the number of cache misses for heapDividePolynomials is:

$$O\left(1 + \frac{3\lg(N)N}{L} + n_a + N\right)$$

Assuming the heap fully fits in cache, this simplifies the expression, removing the factor added by the heap operations, but adding $O(3n_b/L)$ now for iterating through b.

$$O\left(1 + \frac{3n_b}{L} + \frac{3n_a}{L} + N\right)$$

N still appears without a denominator because, in the worst case, each iteration of the loop appends to the opposite of q and r from which was operated on in the previous iteration. This could cause a cache miss each time.

4.3.1 Implementation

Division is essentially a direct application of multiplication. We again use heaps, with all of its optimizations, as presented in Section 4.2.1. There is little to add in terms of optimizations apart from those already mentioned. But, we do note the following detail on memory allocation and re-allocation which is specific to division.

Division differs from multiplication where instead of producing the product terms of the two input operands, we must produce product terms between the divisor and the continually updating quotient. This poses problems for memory management since we do not know ahead of time the sizes of the quotient or remainder. In multiplication we are able to pre-allocate $n_a \cdot n_b$ space for the product because that is the maximum number of product terms. The indeterminate number of quotient and remainder terms does not allow for such one-time allocation and we must continually check for producing more terms than the number for which we have allocated space. We begin by allocating n_a space for the quotient and remainder, as generally the dividend is larger than the divisor. Then, if more terms are produced than we have currently allocated for, we double the current allocation. Doubling the capacity during reallocation is a common technique for data structures in order for the re-allocation cost be *amortized* away [70, Sections 1.3,1.4].

Whenever we reallocate space for the quotient we also reallocate space for the same number of terms in the heap. Recall that the maximum number of terms in the heap is equal to the number of quotient terms (the left-hand operand in the multiplication, as we are distributing quotient terms over the divisor in the multiplication). So, this memory allocation for the heap is safe, in the sense that the heap will never overflow that allocation. The performance benefits are much like multiplication where the heap was precisely the maximum size possible, n_a; the code does not need to check for overflow on each insert into the heap as it is guaranteed to have enough space.

4.3.2 Experimentation

Much like multiplication, we compare our optimized implementation of the heap-based polynomial division against that of Maple. The set up is the same, using number of terms, coefficient bound, and sparsity as defined for the multiplication experimentation (Section 4.2.2). We provide benchmarks for both polynomial division over the integers (with monic divisor) and polynomial division over the rational numbers, Figures 4.8 and 4.9, respectively. In these benchmarks we use increasing sizes of dividend and divisor along with various amounts of sparsity.

In either case, it is important to note their odd shape. It is clearly not a smooth increase in running time with increasing input size (although trends are clearly evident). The reasoning for this is because our dividend and divisor are sparse. Polynomial division results in quotients and remainders which are much more dense than their producing dividends and divisors. This is clear from a simple example like $(x^{10000} - 1)/(x - 1)$ where the resulting quotient has 10000 terms. Since we use randomly generated *sparse* polynomials, degrees of the quotients and remainders can vary by a significant amount within a fixed number of terms for the dividend and divisor. A polynomial in 200 terms with a sparsity of 15 has a much lower degree than a polynomial in 200 terms with a sparsity of 30. Therefore, increased sparsity is able to create higher degree quotients and remainders.

The main trend which should be noticed is the large gap between our implementation and Maple's, with ours being the distinctly faster version. The two algorithms diverge rapidly in their running time as polynomial size and sparsity increases. Speed-up factors upwards of 6 is evident with only moderately sized inputs. Evidently, the more sparse the polynomial the longer the computation. Again, this is caused by the quotient and remainder densifying. As sparsity increased, Maple performed increasingly worse. Likely, this is due to poor memory management, often reallocating and moving data as the quotients and remainders are produced.

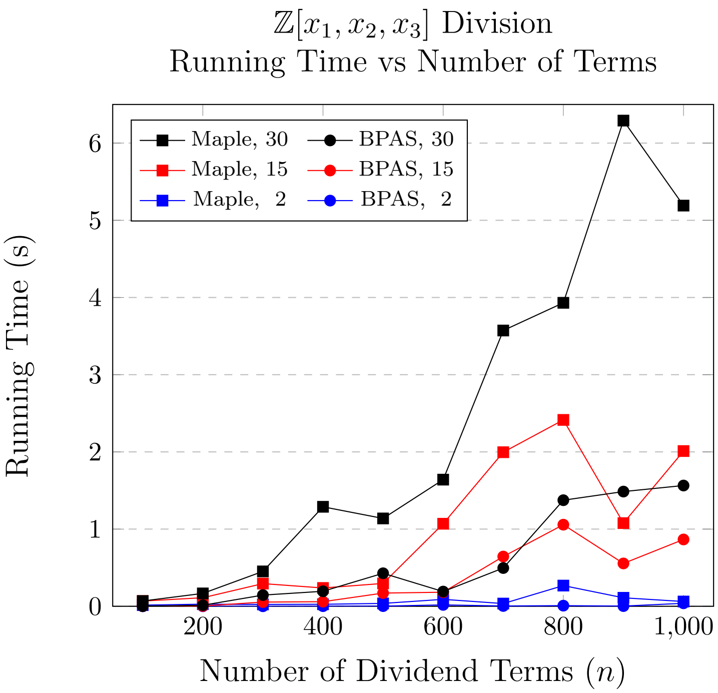


Figure 4.8: Comparing division of integer polynomials. The number of variables is fixed at 3, and the coefficient bound at 128. The number of terms in the divisor is $n/2$. The number of terms in the dividend vary on the x-axis, while the sparsity varies as noted in the legend.

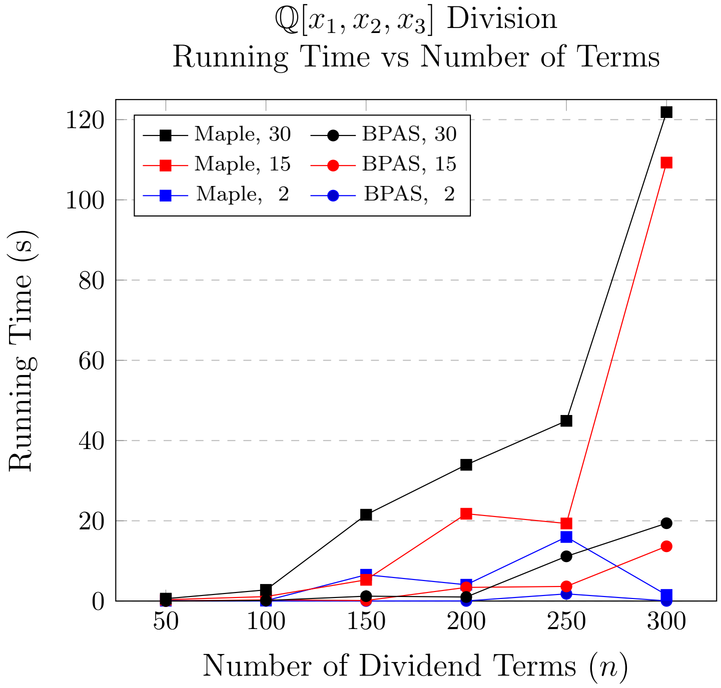


Figure 4.9: Comparing division of rational number polynomials. The number of variables is fixed at 3, and the coefficient bound at 128. The number of terms in the divisor is $n/2$. The number of terms in the dividend vary on the x-axis, while the sparsity varies as noted in the legend.

4.4 Pseudo-Division

The pseudo-division algorithm is an essentially univariate operation. Therefore, we denote polynomials and terms in this section as being elements of $\tilde{\mathbb{D}}[x_2, \ldots, x_v][x_1] = \mathbb{D}[x]$ for an arbitrary integral domain $\tilde{\mathbb{D}}$. It is important to note that while the algorithms and discussion in this section are specified for univariate polynomials they are, in general, multivariate polynomials.

Pseudo-division can really be thought of as a fraction-free division: rather than dividing a by $h = \mathrm{lc}(b)$ for each term of the quotient q, it multiplies a by h in order to ensure that the polynomial division can occur without being concerned with limitations of the base ring. In the definition of pseudo-division, Definition 2.3, it states that at the end of the operation, a should be multiplied by $h^{deg(a)-deg(b)+1}$. However, this is rather coarse. Really, if the quotient ends up with ℓ terms, then the result need only satisfy $h^\ell a = qb+r$. This is because we multiply the dividend by h once per *division step* that is actually performed. There is one such division step per term in the quotient. Of course, if all terms of the dividend and divisor are non-zero, and therefore they are dense, then $\ell = deg(a) - deg(b) + 1$. But this equality is not strict if the operands are sparse. Rather, $\ell \leq deg(a) - deg(b) + 1$. We call this variation where ℓ is less, a *lazy* pseudo-division.

Using the heap-optimized division algorithm, we propose a new algorithm for lazy (sparse) pseudo-division. In this case, rather than computing term-by-term in the multivariate sense, we compute term-by-term in the univariate sense. Thus, the product terms in the heap tracking the product $q \cdot b$ are elements of $\mathbb{D} = \tilde{\mathbb{D}}[x_2, \ldots, x_v]$, and therefore actually multivariate polynomials.

An important consequence of pseudo-division being univariate is that all of the quotient terms are computed first and then all of the remainder terms are computed. This is because we can always carry out a pseudo-division step provided that $\deg(b) \leq \deg(\mathrm{lt}(h^\ell a - qb))$, where $\mathrm{lt}(h^\ell a - qb)$ is the equivalent of $\tilde{r}$ from Algorithm 4.2 when $r = 0$. Thus, we adopt the same symbol for it in Algorithm 4.4 which is the extension of Algorithm 4.2 to pseudo-division. The are only two differences between these algorithms. The first is that each time we compute a new pseudo-quotient term we do so as $\tilde{r}/x^\gamma$, where $\gamma = \deg(b)$ (fraction free division), rather than $\tilde{r}/B_1 = \tilde{r}/(hx^\gamma)$ as before. The second difference, and the reason why we can do this simplified division with $\tilde{r}$, is that we add a factor of h to both a and q at each division step, as specified by the pseudo-division definition.

The division algorithm (Algorithm 4.2) carries over with only minor changes required for proper accounting of the factors of h. This enters in two places: (1) each time a term of a is used, we must multiply the current term A_k of a by h^ℓ, where ℓ is the number of quotient terms computed so far; (2) each time a quotient term is computed we must multiply all of the previous quotient terms by h to ensure that $h^\ell a = qb + r$ will be satisfied. This latter condition introduces an additional consideration for heap management, namely that rather than storing terms of qb, we must store pointers to terms of q and b, since the terms of q will change (by factors of h) as new quotient terms are introduced. No special accounting is needed for the remainder terms because they are all computed after the quotient has been computed and we are done multiplying by factors of h.

Algorithm 4.4 pseudoDividePolynomials(a,b)
$a, b \in \mathbb{D}[x]$, $b \neq 0$; return $q, r \in \mathbb{D}[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r) < \deg(b)$.

```
 1: $(q, r, \ell) := 0$
 2: $h := \mathrm{lc}(b)$; $\gamma = \deg(b)$
 3: while $(\tilde{r} := \mathrm{lt}(h^\ell a - qb - r)) \neq 0$ do
 4:     if $x^\gamma \mid \tilde{r}$ then
 5:         $q := hq + \tilde{r}/x^\gamma$
 6:         $\ell := \ell + 1$
 7:     else
 8:         $r := r + \tilde{r}$
 9: end
10: return $(q, r, \ell)$
```

Proposition 4.4 *Algorithm 4.4 terminates and is correct.*

Proof: Let $\mathbb{D}$ be an integral domain and $a, b \in \mathbb{D}[x]$ with $\gamma = \deg(b) > 0$. If $deg(b) = 0$, then the divisibility test on 4 always passes, all generated terms go to the quotient, and we get a remainder of zero throughout the algorithm. Essentially this is a meaningless operation. q becomes $b^{\ell-1}a$ and the formula holds as $r = 0$ along with the convention that $\deg(0) = -\infty$.

We initialize $q, r, \ell = 0$. It is enough to show that for each iteration of the loop, the degree of $\tilde{r}$ strictly decreases. Since the degree of $\tilde{r}$ is finite, $\tilde{r}$ is zero after finitely many iterations.

We denote the values of the variables of Algorithm 4.4 on the i-th iteration by superscripts. For each i, depending on whether or not $x^\gamma \mid \tilde{r}^{(i)}$ holds, we have two possibilities: (1) $Q_\ell = \tilde{r}^{(i)}/x^\gamma$, where Q_ℓ is a new quotient term; or (2) $R_k = \tilde{r}^{(i)}$, where R_k is a new remainder term. Notice that in case 1 we update the quotient term so $r^{(i+1)} = r^{(i)}$, In case 2 we update the remainder term so $q^{(i+1)} = q^{(i)}$. Suppose that $\tilde{r}^{(i)}$ has just been used to compute a term of q or r, and we now look to compute $\tilde{r}^{(i+1)}$. Depending on whether or not $x^\gamma \mid \tilde{r}^{(i)}$ we have:

Case 1: $x^\gamma \mid \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $Q_\ell = \tilde{r}^{(i)}/x^\gamma$

$$\begin{aligned}
\tilde{r}^{(i+1)} = \mathrm{lt}(h^{\ell+1}a - q^{(i+1)}b - r^{(i+1)}) &= \mathrm{lt}(h^{\ell+1}a - ([hq^{(i)} + Q_\ell]b) - r^{(i+1)}) \\
&= \mathrm{lt}(h^{\ell+1}a - ([hq^{(i)} + Q_\ell]b) - r^{(i)}) \\
&= \mathrm{lt}(h^{\ell+1}a - (hq^{(i)}b + Q_\ell b) - r^{(i)}) \\
&= \mathrm{lt}(h^{\ell+1}a - [hq^{(i)}b + (h\tilde{r}^{(i)} - h\tilde{r}^{(i)}) + Q_\ell b] - r^{(i)}) \\
&= \mathrm{lt}(h^{\ell+1}a - [hq^{(i)}b + h\tilde{r}^{(i)} + Q_\ell(b - hx^\gamma)] - r^{(i)}) \\
&= \mathrm{lt}(h^{\ell+1}a - [hq^{(i)}b + h\tilde{r}^{(i)} + Q_\ell(b - B_1)] - r^{(i)}) \\
&= \mathrm{lt}(h[h^\ell a - q^{(i)}b - \tilde{r}^{(i)}] - Q_\ell(b - B_1) - r^{(i)}) \\
&= \mathrm{lt}(\big(h[h^\ell a - q^{(i)}b - \tilde{r}^{(i)}] - r^{(i)}\big) - Q_\ell(b - B_1)) \\
&< \mathrm{lt}(\tilde{r}^{(i)}) = \tilde{r}^{(i)}.
\end{aligned}$$

In the second last line notice that, since $\tilde{r}^{(i)} = \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $h \in \mathbb{D}$, then we can ignore h for the purposes of choosing a term with highest degree and we have that $\mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)} - \tilde{r}^{(i)}) < \mathrm{lt}(\tilde{r}^{(i)})$. Also, the expression $Q_\ell(b - B_1)$ has leading term $Q_\ell B_2$ which is strictly less than $\tilde{r}^{(i)} = Q_\ell x^\gamma$, by the ordering of the terms of b. Hence $\tilde{r}^{(i+1)}$ is strictly less than $\tilde{r}^{(i)}$.

Case 2: $x^\gamma \nmid \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $R_k = \tilde{r}^{(i)}$

$$\begin{aligned}
\tilde{r}^{(i+1)} = \mathrm{lt}(h^\ell a - q^{(i+1)}b - r^{(i+1)}) &= \mathrm{lt}(h^\ell a - q^{(i)}b - (r^{(i)} + R_k)) \\
&= \mathrm{lt}((h^\ell a - q^{(i)}b - r^{(i)}) - \tilde{r}^{(i)}) \\
&< \mathrm{lt}(\tilde{r}^{(i)}) = \tilde{r}^{(i)}
\end{aligned}$$

Similarly, $\tilde{r}^{(i)} = \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$, thus their difference must have a leading term strictly less than $\tilde{r}^{(i)}$.

The loop therefore terminates. The correctness is implied by the condition that $\tilde{r} = 0$ at the end of the loop. The condition $\deg(r) < \deg(b)$ is ensured because the only times terms are added to the remainder is when $x^\gamma \nmid \tilde{r}$ holds, that is, until $\deg(h^\ell a - qb) < \deg(b)$ holds. □

Heap-optimization of Algorithm 4.4 proceeds in much the same way as for division. The only additional consideration required for Algorithm 4.5 is the accounting for factors of h in the computation of $\mathrm{lt}(h^\ell a - qb - r)$. This only requires adding as many factors of h to A_k that have been added to the quotient up to the current iteration. Since ℓ terms have been added to q, we multiply A_k by h^ℓ each time we use one of the terms. Additional factors of h are added when the previous quotient is multiplied by h prior to the computation of the next quotient term. Other than this, the shift from Algorithm 4.4 to Algorithm 4.5 follows the analogous shift between Algorithms 4.2 and 4.3 exactly. We therefore have the following:

Algorithm 4.5 heapPseudoDividePolynomials(a,b)
$a, b \in \mathbb{D}[x]$, $a = \sum_{i=1}^{n_a} a_i x^{\alpha_i} = \sum_{i=1}^{n_a} A_i$, $0 \neq b = \sum_{j=1}^{n_b} b_j x^{\beta_j} = \sum_{j=1}^{n_b} B_j$;
return $q, r \in \mathbb{D}[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r) < \deg(b)$.

```
1: $(q, r, l) := 0$
2: $h := \text{lc}(b)$; $\gamma := \deg(b)$
3: $k := 1$
4: while ($\delta :=$ heapPeek()) $> -1$ or $k \leq n_a$ do
5:     if $\delta < \alpha_k$ then
6:         $\tilde{r} := h^\ell A_k$
7:         $k := k + 1$
8:     else if $\delta = \alpha_k$ then
9:         $\tilde{r} := h^\ell A_k -$ heapExtract()
10:        $k := k + 1$
11:    else
12:        $\tilde{r} := -$ heapExtract()
13:    if $x^\gamma \mid \tilde{r}$ then
14:        $q := hq$
15:        $\ell := \ell + 1$
16:        $Q_\ell := \tilde{r}/x^\gamma$
17:        $q := q + Q_\ell$
18:        heapInsert($Q_\ell, B_2$)
19:    else
20:        $r := r + \tilde{r}$
21: end
22: return $(q, r, \ell)$
```

Proposition 4.5 *Algorithm 4.5 terminates and is correct.*

Proof: Let $\mathbb{D}$ be an integral domain and $a, b \in \mathbb{D}[x]$ with $\deg(b) > 0$. If $b \in \mathbb{D}$ then this degenerative case proceeds as in Proposition 4.4. Then $r = 0$ with $deg(r) = -\infty < 0 = deg(b)$ and we are done.

Observe that there are two main conditionals (lines 5–12 and 13–20) in the while loop. Given Proposition 4.4, it is enough to show that the first conditional computes $\text{lt}(h^\ell a - qb - r)$ and the second uses $\tilde{r}$ to add terms to either q or r, depending on whether or not $\gamma \mid \tilde{r}$.

We initialize $q, r = 0$, $k = 1$ (index into a), $\ell = 0$ (index into q), $\delta = -1$ (heap empty condition) since the heap is initially empty. The key modification of Algorithm 4.4 to reach Algorithm 4.5 is to use terms of qb from the heap to compute $\tilde{r} = \text{lt}(h^\ell a - qb - r)$. This requires tracking three cases: (1) $\tilde{r}$ is an uncancelled term of $h^\ell a$; (2) $\tilde{r}$ is a term of the difference $(h^\ell a - r) - (qb)$; and (3) $\tilde{r}$ is a term of $-qb$ such that all remaining terms of $h^\ell a - r$ have smaller degree. Notice that all of the terms of q are computed before the terms of r since this is essentially univariate division with respect to the monomials. Therefore we can ignore r in the sub-expression $h^\ell a - r$. Thus, computing $\text{lt}(h^\ell a - qb - r)$ in order simply requires computing terms of $(h^\ell a - qb)$ in order.

These three cases for computing $\tilde{r}$ are handled by the first conditional. Let $a_k X^{\alpha_k} = A_k$ be the greatest uncancelled term of a. In case 1, either the heap is empty (indicating that no terms of q have yet to be computed or all terms of qb have been extracted) or if $\deg(qb) = \delta > -1$ but $\delta < \alpha_k$. In either situation $\delta < \alpha_k$ holds and $\tilde{r}$ is chosen to be A_k. The term $\tilde{r}$ is a term of the difference $(h^\ell a - qb)$ (case 2) if both A_k and the top term of the heap have the same degree ($\delta = \alpha_k$) and $\tilde{r}$ is chosen to be the difference of $h^\ell A_k$ and the greatest uncancelled term of qb. Lastly, $\tilde{r}$ is a term of $-qb$ (case 3) in any other situation, that is, $\delta > \alpha_k$. Thus, the first conditional computes $\mathrm{lt}(h^\ell a - qb - r)$, *provided* that the second conditional correctly adds terms to q and r.

The second conditional adds terms to the quotient when $x^\gamma \mid \mathrm{lt}(h^\ell a - qb)$ holds. Since each new quotient term adds another factor of h, we must first multiply all previous quotient terms by h. We then construct the new quotient term to cancel $\mathrm{lt}(h^\ell a - qb)$ by setting $Q_{\ell+1} = \mathrm{lt}(h^\ell a - qb)/x^\gamma$, as in Algorithm 4.4. Since $Q_\ell B_1$ cancels a term of $(h^\ell a - qb)$ by construction, then line 18 initializing a new stream with $Q_\ell B_2$ is also correct. If, on the other hand, $x^\gamma \nmid \mathrm{lt}(h^\ell a - qb)$, all remaining $\tilde{r}$ terms are remainder terms, which are correctly added by line 20. □

In order to estimate the cache complexity of heapPseudoDividePolynomials, we must notice that coefficients are, in general, polynomials. Whether they be polynomials or simple numbers let us denote cache misses as a result of multiplication over $\mathbb{D}$ to be C_m. Then, the cache complexity is much like heapDividePolynomials with the addition of updating the quotient ($q := hq$) and multiplying A_k by h^ℓ. Again, let N denote the number of terms in both the quotient and remainder. The number of cache misses resulting from heap operations is $O(3\lg(N)N/L)$, the number of cache misses for updating the quotient is $O(C_m N^2)$ (the update can occur N times), and the number of cache misses for multiplying A_k by h^ℓ is $O(C_m n_a)$ (this multiplication occurs n_a times). Hence, the number of cache misses for Algorithm 4.5 is:

$$O\left(1 + \frac{3\lg(N)N}{L} + C_m n_a + C_m N^2\right)$$

4.4.1 Implementation

As we saw in the previous section, the algorithm for division can easily be adapted for pseudo-division. With only the modification of multiplying the dividend and quotient by the divisor's initial, we obtained an algorithm for pseudo-division that efficiently produces terms in order. However, the implementation between these two algorithms is very different. In essence, pseudo-division is a univariate operation, viewing the input multivariate polynomials recursively. That is, the dividend and divisor are seen as univariate polynomials over some arbitrary (polynomial) integral domain. Therefore, coefficients can be, and indeed are, entire polynomials themselves. Coefficient arithmetic becomes non-trivial. We can make use of the recursive array structure as seen in Section 3.4 in order to efficiently work with polynomials in this recursive manner.

With the recursive view of a polynomial efficiently implemented, it is then important to consider efficiency of coefficient arithmetic. As coefficients are now full polynomials there is more overhead in manipulating them and performing arithmetic. One important implementation detail is to perform the addition (and subtraction) of like-terms in-place

(Section 4.1). Such combinations occur when computing the leading term of $h^\ell a - qb$ and when combining like-terms in the quotient-divisor product. In-place addition allows for the re-use of underlying GMP data. Therefore, performance of in-place addition compared to out-of-place becomes increasingly better as coefficients grow throughout the pseudo-division algorithm.

Similarly, the update of the quotient by multiplying by the initial of the divisor, requires a multiplication of full polynomials. If we wish to save on memory movement we should perform this multiplication in place. However, notice that, in our recursive representation, coefficient polynomials are tightly packed in a continuous array. To modify them in place would require shifting all following coefficients down the array to make room for the strictly large product polynomial. To avoid this unnecessary memory movement, we modify the recursive data structure exclusively for the pseudo-quotient polynomial. We break the continuous array of coefficients into many arrays, one for each coefficient. This allows them to grow without displacing the following terms. This representation is shown in Figure 4.10. At the end of the algorithm, once the quotient has finished being produced, we collect and compact all of these disjoint polynomials into a single, packed array. In contrast, the remainder is never updated once its terms are produced. Moreover, we do not require any recursively viewed operations on the remainder. Hence, as terms of the remainder are produced, we store them directly in the normal, distributed representation, avoiding conversion out of the recursive representation and any memory overhead of the additional recursive array.

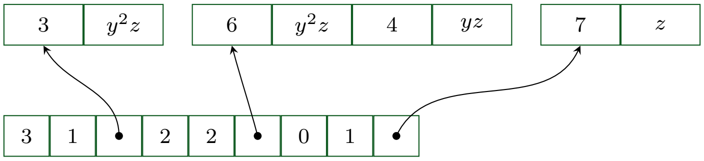


Figure 4.10: An example recursive polynomial representation for a pseudo-quotient polynomial. Notice how the coefficient polynomials are in disjoint arrays, allowing them to grow in-place without affecting other coefficient polynomials.

Just as with division, the remainder and quotient are of indeterminate size and so we do not pre-allocate a maximum space for them. We follow the same techniques of division, continuously doubling the allocated space for either quotient or remainder as they grow to fill their previously allocated space.

Lastly, our final optimization is common among other sparse pseudo-division algorithms, such as `sprem` in Maple. We perform a divisibility test between a newly produced quotient term and the initial of the divisor. If division is exact, we avoid one multiplication of the quotient with the divisor's initial, and the newly produced quotient term is replaced by its quotient calculated by the exact division. This divisibility test is little overhead as the test usually fails very early. Often, this divisibility test is instead performed by a GCD calculation in order to always multiply the quotient by the smallest possible polynomial instead of the full initial of the divisor. However, we note that multivariate GCD calculations can result in excessive overhead.

4.4.2 Experimentation

Just as in multiplication and division, we proceed to perform benchmarking for pseudo-division with both rational number and integer polynomials. These experiments are, again, done using Maple as a comparison point, since Maple has the current leading implementation of integer polynomial arithmetic [56–59].

We note one slight variation in the terminology of this round of experimentation compared to that of multiplication or division. In particular, the meaning of sparsity, coefficient bound, and number of terms has changed. Since pseudo-division is essentially univariate, the polynomials are defined recursively, belonging to the ring $\mathbb{D}[x_2, \ldots, x_v][x_1]$. So, the number of terms means the number of terms of the polynomial with respect to this recursive representation (essentially the number of distinct values of the exponents on x_1 in the polynomial). Similarly, sparsity is defined in the univariate sense, being the smallest integer which is greater than the maximum degree difference (with respect to x_1) of any two adjacent terms in the polynomial. Coefficient bound now, in addition to describing the maximum number of bits needed to encode numerical coefficients of the base ring, also defines the number of terms in the polynomial coefficients which belong to $\mathbb{D}[x_2, \ldots, x_v]$. For example, with a coefficient bound of 3 and number of terms to be 10, if distributed out, the polynomial actually has 30 terms.

For both integer polynomials (Figure 4.11) and rational number polynomials (Figure 4.12) we fix the number of terms in the polynomial and vary sparsity. The importance for doing this is that our pseudo-division algorithm is particularly concerned with effectively operating with sparse polynomials. The algorithm essentially provides no benefit when sparsity is low. In those cases, the naive variation (Algorithm 2.2) works just as well as our heap-based algorithm. Moreover, for a fixed sparsity and increasing number of terms, the computations become limited by (numerical) coefficient arithmetic (coefficients grow immense during pseudo-division by repeated multiplication by h).

Thus, the benchmarks provided by varying sparsity more drastically is both important and interesting. We note that, since sparsity is defined in the univariate sense, and we are working over multivariate polynomials, a sparsity of 10 is actually very sparse, especially so with more variables in the coefficient polynomials. It is clear from the plots that running times between our algorithm and Maple's implementation quickly diverge. For integer polynomials in particular, the running time of Maple is following a nearly exponential trend as sparsity increases while our implementation is close to linear.

As a comparison point, we also implement an optimized version of the naive pseudo-division algorithm (Algorithm 2.2). This highlights the benefits gained purely from algorithmic differences as both implementations make use of the same data structure and optimization techniques. These experiments are highlighted in Figure 4.13, where the scale of the plot is more conducive to direct comparison.

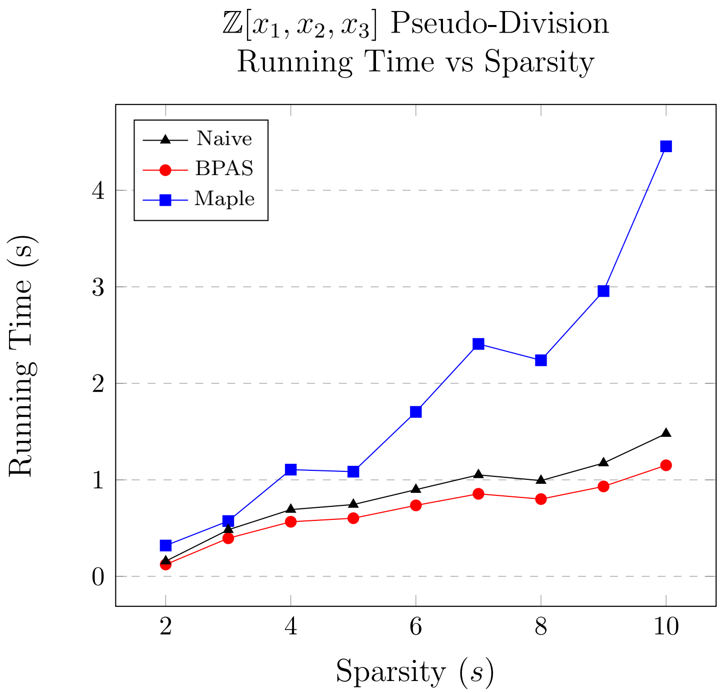


Figure 4.11: Comparing pseudo-division of integer polynomials. The number of variables is fixed at 3, and the coefficient bound at 3. The number of terms in the dividend is 175, and the number of terms in the divisor is 50. Sparsity varies on the x-axis.

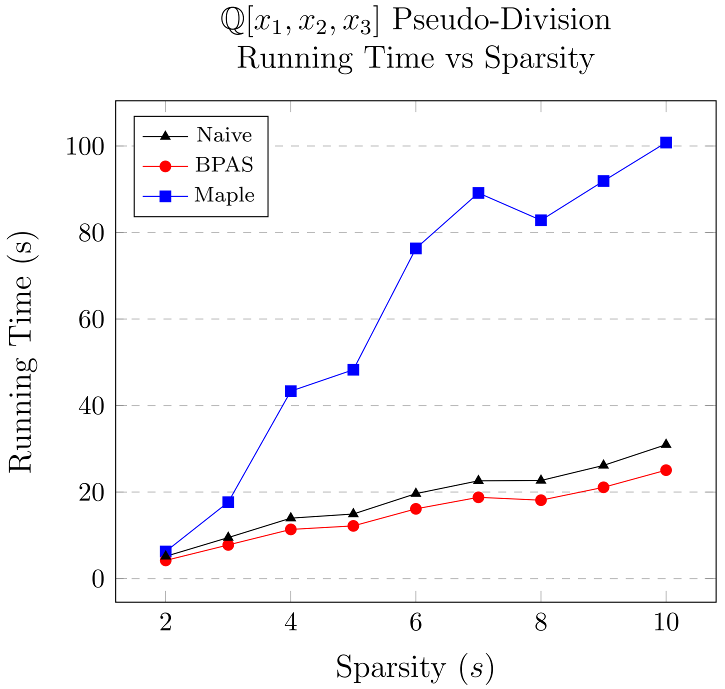


Figure 4.12: Comparing pseudo-division of rational number polynomials. The number of variables is fixed at 3, and the coefficient bound at 3. The number of terms in the dividend is 175, and the number of terms in the divisor is 50. Sparsity varies on the x-axis.

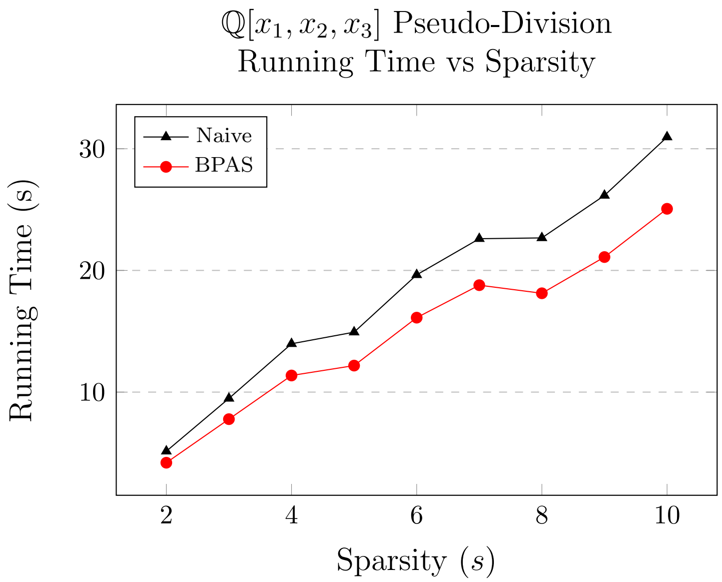


Figure 4.13: Comparing naive and heap-based pseudo-division of rational number polynomials. The number of variables is fixed at 3, and the coefficient bound at 3. The number of terms in the dividend is 175, and the number of terms in the divisor is 50. Sparsity varies on the x-axis.

Beyond randomly generated instances we also compare our implementation with MAPLE on specific examples found in the literature. For each example we begin with a known polynomial system and perform a triangular decomposition on it. We then pseudo-divide a related polynomial by the main polynomial (highest-ordered) of the triangular decomposition. That is, the divisor comes the triangular decomposition while the dividend is some other polynomial. The first example (Example 4.1) [28, Example 7.2] uses $(xyz)^i$ as the dividend while the second example (Example 4.2) [13, Example Rose] uses z^i as the dividend. Table 4.2 displays the results for these two examples. We use the `prem` command in MAPLE for computing pseudo-remainder and pseudo-quotient, comparing it to our own (non-lazy) pseudo-division implementation. Various values of i are chosen to increase the complexity of the problem and solution. We note the remarkable difference in memory usage between BPAS and MAPLE. Our implementation uses 10-15 times less memory to perform the same operation.

Example 4.1

$$\begin{cases} x^2yz + xy^2z + xyz^2 + xyz + xy + xz + yz, \\ x^2y^2z + xy^2z^2 + x^2yz + xyz + yz + x + z, \\ x^2y^2z^2 + x^2y^2z + xy^2z + xyz + xz + z + 1 \end{cases} \tag{4.1}$$

Example 4.2

$$\begin{cases} 7y^4 - 20x^2, \\ 2160x^2z^4 + 1512xz^4 + 315z^4 - 4000x^2 - 2800x - 490, \\ -10080000x^4z^3 - 28224000x^3z^3 - 15288000x^2z^3 - 1978032xz^3 - 180075z^3 - 23520000x^4yz^2 \\ -41395200x^3yz^2 - 26726560x^2yz^2 - 7727104xyz^2 - 852355yz^2 + 40320000x^6y^2z + 28800000x^5y^2z \\ +21168000x^3y^2z + 4939200x^2y^2z + 347508xy^2z + 67200000x^5y^3 + 94080000x^4y^3 + 40924800x^3 * y^3 \\ +2634240x^2y^3 - 2300844xy^3 - 432180y^3 \end{cases} \tag{4.2}$$

i	BPAS	Maple
50	1.80 (0.03GB)	1.77 (0.38GB)
100	26.01 (0.27GB)	43.00 (2.27GB)
150	136.31 (0.96GB)	235.39 (15.47GB)
200	434.35 (2.18GB)	907.18 (33.31GB)

(a) Example 1

i	BPAS	Maple
50	4.50 (0.03GB)	6.670 (0.60GB)
100	68.78 (0.27GB)	82.66 (4.97GB)
150	327.14 (0.89GB)	387.91 (11.79GB)
200	1011.96 (2.10GB)	1195.51 (20.10GB)

(b) Example 2

Table 4.2: Comparing pseudo-division on examples of triangular decompositions coming from the literature. Table (a) shows pseudo-division of $(xyz)^i$ by the main polynomial of the triangular decomposition of Example 4.1. Table (b) shows pseudo-division of z^i by the main polynomial of the triangular decomposition of Example 4.2

4.5 Normal Form and Multi-Divisor Pseudo-Division

We mention, briefly, a direct application of our division and pseudo-division algorithms. Indeed, we claim them to be fundamental and foundation to higher-leveled algorithms. Such natural extensions exist as normal form with respect to Gröbner bases and multi-divisor pseudo-division (or more practically, pseudo-division by a triangular set), which each extend multivariate polynomial division and pseudo-division, respectively.

We saw reduction in the sense of Gröbner bases in Section 2.2.4. Normal form is one name for the remainder of a polynomial that is reduced with respect to a Gröbner basis. Since reduction occurs with respect to an arbitrary Gröbner basis, that is, one with possibly more than one element, our multivariate division algorithm (Algorithms 4.2 and 4.3) would be insufficient. Our multivariate division algorithms use only a single divisor, while normal form requires dividing by many divisors multiple times. However, just as the general reduction algorithm is explained by the repeated application of *reduction in one step* (Equation 2.1), one can define a normal form algorithm which is simply repetitive calls to the basic multivariate division algorithm with one divisor. This method is described in [3] with [3, Algorithm 7] in particular showing a specialized normal form where the divisors form a triangular set.

In a very similar manner, one can describe pseudo-division with respect to many divisors simultaneously. This is similar to normal form except that each reduction in one step is done by the fraction-free pseudo-division instead of the normal multivariate division. This process is called multi-divisor pseudo-division or, in specializes cases, pseudo-division with respect to a triangular set [4]. Practical implementations of pseudo-division with respect to a triangular set, using single-divisor pseudo-division as the foundational operation, are provided in [3, Algorithms 6 and 8]

Chapter 5

Symbolic and Numeric Polynomial Interpolation

There are various flavours of interpolation including nearest-neighbour, linear, polynomial, and trigonometric. However, polynomial interpolation is generally the most useful, and indeed, "must be considered fundamental" [23, p. 330]. It is useful to both symbolic and numeric computations. From real analysis, the *Weierstrass approximation theorem* [5, Theorem 17.7] (also called *StoneWeierstrass theorem*) tells us that, within some region, any function continuous in that region can be accurately approximated by a polynomial. Moreover, polynomials themselves are very easy to work with. Their derivatives and integrals are themselves polynomials, and are easily computable. Hence, approximation of a function by a polynomial, and thus polynomial interpolation, is used most often [18, Chapter 3]. Various forms of polynomial interpolation exist, such as, Lagrange, Newton's, Hermite, and cubic splines. Lagrange interpolation "is in most cases the method of choice for dealing with polynomial interpolants" [10, p. 501].

Although Lagrange interpolation is praised analytically, it is traditionally numerically unstable. However, given the right approaches and algorithms (such as through the formulas of barycentric interpolation [10]), it is still a viable numerical option. In such cases, it is even more stable than Newton's interpolation which has been the preferred method before improvements to Lagrange were made [23]. Classically, Lagrange interpolation is a *univariate* interpolation method. In this case, Lagrange is well-behaved and well-studied, and indeed we will use it as a method for univariate interpolation, and later for sparse multivariate interpolation (Section 5.3), where univariate interpolation forms the basis of these algorithms.

Although multivariate interpolation can be done by a generalization of the univariate Lagrange interpolation [26, 67, 68], it is a difficult, troublesome, and "particularly annoying" [26, p. 1] problem due to numerical issues [23] and limitations on the geometric configuration of the points being interpolated (see Section 5.2.2). Nonetheless, ideas from Lagrange interpolation extend to other (dense) multivariate interpolation schemes

(Section 5.2).

Indeed, the configuration of the points being interpolated are an important aspect to consider. Some algorithms (not withstanding geometric limitations) can interpolate a polynomial where the points and values, (π_i, β_i), are supplied (see Section 5.2). Other algorithms [8, 24, 75, 76] require a *black-box* representation of the underlying function in order to perform the interpolation using pre-determined (or random) points (see Section 5.3). A black-box is some method which, given some point π_i, returns the value of the underlying function at that point. *Straight line programs* and *arithmetic circuit representations* are possible implementations of black boxes [32].

These black-box algorithms require that the values obtained for the underlying function be *exact*. In contrast, the values we wish to interpolate may be *noisy*, especially when the values come from experimental observations. In such cases, symbolic methods are not well suited, and hence numerical methods should be employed. In particular, if the number of observations *over-determine* the interpolating polynomial (say because we restrict the degree of the interpolating polynomial), then we have moved from the problem of interpolation to that of *curve fitting*. This can often occur in real-world experimentation. Both the problems of noisy data and over-determined systems can be solved with the same technique using numerical methods (Section 5.4).

Throughout the following sections we use the following notations. π_i are (possibly multivariate) points which we wish to interpolate. β_i are the corresponding values of the underlying function evaluated at π_i. The function resulting from the interpolation, the *interpolant*, is denoted by f.

5.1 Univariate Polynomial Interpolation

Univariate polynomial interpolation has long been studied. In Section 2.5 we gave a short description of various interpolation techniques like Lagrange, Newton's, and a brute-force solution to a system of linear equations. Of course, just as with our polynomial arithmetic, our interpolation implementation uses exact arithmetic (using GMP multi-precision numbers). Hence, we are not subject to the numerical concerns which often plague Lagrange interpolation. Because of this, along with the fact that the Lagrange interpolating polynomial is so simple to obtain directly, we do not bother implementing Newton's. We will also note that solving systems of equations exactly is a very slow process (a sentiment echoed in [23]). Our Lagrange method is an *order of magnitude* faster than solving a system of linear equations exactly (see Figure 5.1), even using a state of the art linear system solver (see Section 5.3).

Recall the *Lagrange basis polynomials* (Section 2.5.1):

$$\begin{aligned}\phi_j(x) &= \frac{(x-\pi_1)\dots(x-\pi_{j-1})(x-\pi_{j+1})\dots(x-\pi_n)}{(\pi_j-\pi_1)\dots(\pi_j-\pi_{j-1})(\pi_j-\pi_{j+1})\dots(\pi_j-\pi_n)}\\ &= \prod_{\substack{i=1\\ i\neq j}}^{m} \frac{(x-\pi_i)}{(\pi_j-\pi_i)}\\ &= \frac{q_j(x)}{d_j}\end{aligned}$$

and the *Lagrange interpolating polynomial*:

$$f(x) = \sum_{j=1}^{m} \beta_j \phi_j(x)$$

Based on our efficient polynomial representations (Chapter 3) we wish to obtain a unique *distributed* polynomial for our interpolant. Our algorithm does so by computing a distributed form for each ϕ_j and efficiently performing the summation See Algorithms 5.1 and 5.2.

An interesting aspect of this algorithm is balancing the arithmetic costs with the memory costs. There is a lot of repetition among the factors of each of the $\phi_j(\pi)$ basis polynomials. Naturally, it makes sense to cache[1] these factors instead of computing them multiple times. This is especially true for the factors in the denominators. Although there are many such factors, once expanded, they reduce to a single rational number, d_j. So, we pre-compute and cache each d_j. To be precise, we actually multiply it by β_j to obtain ${}^{\beta_j}/_{d_j}$ and cache this rational number. This is nice as, after this initial pre-computation, β_j is never needed again. The list of values is iterated exactly once. In contrast, the list of points must be iterated through n times to compute the factors $(\pi_i - \pi_j)$ for each possible combination of i and j.

Next, we also construct and cache the n possible degree-1 polynomial factors ($factor_i = (x - \pi_i)$) appearing in the q_j polynomials. Of course these factors will each be used $n-1$ times, each one per q_j polynomial with $j \neq i$. Hence, the trade-off for memory usage is worth the computational savings. Moreover, we then only need to iterate through the list of points once to obtain all of these factors. And at this stage, we no longer need the list of points either for remainder of the algorithm.

After all of this pre-computation completes, we look to be as efficient as possible with the use of memory for effective cache usage. We construct f iteratively, and efficiently,

[1] Cache here is meant in the programmatic sense, where temporary values are stored in temporary memory during the course of an algorithm instead of recomputing them multiple times.

using our in-place polynomial addition algorithm (Section 4.1):

$$\begin{aligned}
f^{(0)} &:= 0\\
f^{(1)} &:= f^{(0)} \quad + \beta_1\phi_1\\
&\;\vdots\\
f^{(i)} &:= f^{(i-1)} + \beta_i\phi_i\\
&\;\vdots\\
f := f^{(n)} &:= f^{(n-1)} + \beta_n\phi_n
\end{aligned}$$

We call this in-place addition addPolynomials_InPlace.

What remains is to compute the distributed form of each ϕ_j by multiplying together the pre-computed factors $(\pi - \pi_i)$, $i \neq j$. Notice that one of the operands of the multiplication is always a degree-1 polynomial. Hence, we have implemented a specific algorithm to efficiently multiply a generic polynomial by a univariate polynomial with degree 1 (Algorithm 5.2) instead of using generic polynomial multiplication. Moreover, this multiplication by a binomial is done *in-place*. Since we are accumulating the factors into a single resulting polynomial (ϕ_j) this is only natural, and again, is an effective use of memory.

Combining all of these aspects together we obtain our Lagrange interpolation algorithm, LagrangeInterpolation, Algorithm 5.1.

Algorithm 5.1 LagrangeInterpolation(π_i,β_i)

$\pi_i, \beta_i \in \mathbb{Q}$, $1 \le i \le n$; returns the unique interpolating polynomial $f(x) \in \mathbb{Q}[x]$.

```
#Set-up denominators
$d_j = 1, 1 \le j \le n$
for $j = 1$ to $n$ do
    for $i = 1$ to $n,\ i \neq j$ do
        $d_j = d_j * (\pi_j - \pi_i)$
    $d_j = \frac{\beta_j}{d_j}$
#Set-up numerator factors
for $i = 1$ to $n$ do
    $factor_i = (x - \pi_i)$
#Produce each $q_j$ in turn, reusing $p$ variable, and sum into $f$
$f = 0$
for $j = 1$ to $n$ do
    $p = 1$
    for $i = 1$ to $n$ do
        if $i \neq j$ then
            $prod$ = multiplyByBinomial_InPlace($prod$, $factor_i$)
    $prod = d_j \cdot prod$
    $f$ =addPolynomials_InPlace($f, prod$)
return $f$
```

Algorithm 5.2 MultiplyByBinomial_InPlace(f,b)

$f \in \mathbb{D}[x_1, \dots, x_v]$, $b = cx_i + d$, $c, d \in \mathbb{D}$, $1 \leq i \leq n$; returns nothing, the result is computed in-place for f.

```
$g$ = empty polynomial of $n_f$ terms
for $i = 1$ to $n_f$ do
    $g_i = df_i$
    #Note $x_i \cdot f_i$ is only a simple addition of exponent vectors
    $f_i = cx_i \cdot f_i$
addPolynomials_InPlace($f, g$)
```

Using our LagrangeInterpolation algorithm, we compare it against the brute-force linear equations solution for various numbers of interpolation points. The Lagrange method is more than an order of magnitude faster than the linear equations method. We also compare our implementation against Maple's `CurveFitting:-PolynomialInterpolation` method for univariate interpolation with a monomial basis. Our Lagrange implementation is also nearly an order of magnitude faster than Maple's. These results are shown in Figure 5.1. Note on the plot the y-axis has a log scale.

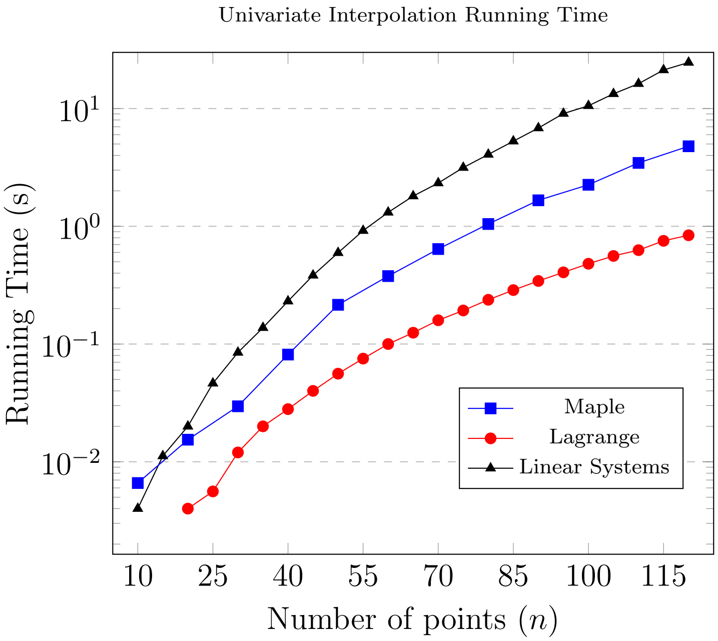


Figure 5.1: Comparing our LagrangeInterpolation algorithm to the brute-force linear system solving. For $n = 10, 15$ Lagrange was too fast to measure. The y-axis has a log-scale.

We conclude with possibilities for yet further improvement. Notice that the computation of each ϕ_j is independent, using read-only access to some cached data elements. It should be very easy then to compute each ϕ_j in parallel. Moreover, under this scheme we could use *parallel reduction* [63] in order to effectively sum f using a binary-tree of partial sums to minimize addition operations.

Nonetheless, this efficient implementation of univariate interpolation will be very helpful as a starting point for further algorithms. As we will see, the sparse multivariate interpolation algorithm (Section 5.3) relies heavily on univariate interpolation.

5.2 Dense Multivariate Interpolation

We have seen how it is possible to set up a system of linear equations in order to solve the problem of interpolation (Section 2.5). Using the set of univariate monomials $\{1, x, x^2, x^3, \dots\}$ as basis functions, as in (univariate) polynomial interpolation, the *sample matrix* becomes a Vandermonde matrix. This same idea can be generalized to the set of multivariate monomials, assuming the number of variables is known. Such a set over 3 variables could look like:

$$\{1,\ x,\ y,\ z,\ xy,\ xz,\ yz,\ xyz,\ x^2y,\ x^2z,\ xy^2,\ y^2z,\ xz^2,\ yz^2, x^3,\ y^3,\ z^3,\ \dots\}$$

What is important to observe is how many monomials exist in this list even with small degree. With a maximum (total) degree of 3 there are only 4 univariate monomials. However, in 3 variables there are 17 such monomials. In general, say there are v variables, each with maximum partial degree d_i. Then, the maximum number of v-variable monomials with respective partial degrees less than or equal to d_i is

$$\prod_{i=1}^{v}(d_i+1)$$

If $d_i = d$ for all variables, then we obtain a more familiar looking equation: $(d+1)^v$. If one were to consider d as a total degree instead, then the number of multivariate monomials is $\binom{v+d}{d}$ [22]. In either case, clearly the number of such monomials grows very large very quickly.

If one were to set up a system of equations for the multivariate monomial basis, the sample matrix would very quickly grow *huge*. This is important for two reasons. First, solving a system of equations requires $O(n^3)$ operations [23], n being the dimension of the matrix. Second, we require n sample points and function values to solve this system. Evaluating the function may be costly depending on the application and underlying function. Since the dimension of the sample matrix is given by the number of basis functions, say $n = (d+1)^v$, then we get that solving this system is roughly $O((d+1)^{3v})$. But, when degrees and number of variables are low, this is not a necessarily horrible solution.

Therefore, it is by solving a system of linear equations using the multivariate monomials as the basis functions that we arrive at a scheme for dense multivariate interpolation. Other schemes exist, such as [74, Section 14.1], where each variable is interpolated one after another, by a univariate algorithm. However, this method by solving a single system of equations is more direct. Moreover, the ideas (and parts of the implementation) of this dense interpolation scheme will carry over into sparse interpolation, albeit with many tricks. We then dedicate the remainder of this section to describing our implementation of linear system solving using exact arithmetic.

Highly optimized libraries exist for solving systems of linear equations exactly [9, 39, 71]. Through comparative benchmarks the Integer Matrix Library (IML) [21] has been shown to be the current leader for solving systems of arbitrary-precision integers [20]. We make use of this library to solve the linear systems constructed by dense interpolation. This library solves systems using the following general scheme:

(1) Convert the input system to a prime field, $\mathbb{Z}/p^2$, for a suitably small prime so that the integer can fit within the mantissa of a floating point number.
(2) Perform numerical linear algebra using BLAS [11].
(3) Reconstruct the solution.

However, this library is limited to working with integer matrices. Thus, for the case of rational numbers, we must first convert the system of linear equations to one over the integers. This is a very simple process. Let $\mathbf{A} \in \mathbb{Q}^{n\times n}$, $\mathbf{b}, \mathbf{x} \in \mathbb{Q}^{n\times 1}$ and $\mathbf{C} \in \mathbb{Z}^{n\times n}$, $\mathbf{d} \in \mathbb{Z}^{n\times 1}$. We compute the least common multiple of the denominators in $\mathbf{A}$ and $\mathbf{b}$, $d_{\mathbf{A}}$ and $d_{\mathbf{b}}$, respectively, factoring them out as a scalar multiple. We then proceed as follows.

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \frac{1}{d_{\mathbf{A}}}\tilde{\mathbf{A}}\mathbf{x} &= \frac{1}{d_{\mathbf{b}}}\tilde{\mathbf{b}} \\ d_{\mathbf{b}}\tilde{\mathbf{A}}\mathbf{x} &= d_{\mathbf{A}}\tilde{\mathbf{b}} \\ \mathbf{C}\mathbf{x} &= \mathbf{d} \end{aligned}$$

Notice that our solution vector, $\mathbf{x}$, has not changed. Hence, the solution to this integer system is the same as the original rational number systems. Indeed, we have only multiplied by a factor of 1 twice, as ${}^{d_{\mathbf{A}}}/_{d_{\mathbf{A}}}$ and ${}^{d_{\mathbf{b}}}/_{d_{\mathbf{b}}}$. With this scheme for solving rational number linear systems we can proceed with dense multivariate interpolation for rational number polynomials.

Before we can construct the linear system, there is one difficulty for multivariate interpolation not found in univariate interpolation: the choice of (multivariate) monomials. The *degree bound* – the maximal allowed degree of the interpolant – of a univariate polynomial interpolant is implicit by the number of data points. That degree is one less than the number of points. In general, one additional point is needed to interpolate one additional (partial) degree. For example, to move from x^2y to x^2y^2 one additional point is needed. However, that same point could be used to move from x^2y to x^3y. The number of points is ambiguous, for multivariate monomials for determining the partial degrees of the interpolant. Rather, explicit degree bounds must be given for each variable to unambiguously determine the set of (multivariate) monomials to use as the function basis.

Say we wish to interpolate a polynomial over $\mathbb{D}[x, y, z]$ with degree bounds $d_x = 2$, $d_y = 1$, $d_z = 1$. We can enumerate all monomials with partial degrees less than or equal

[2] A prime field is simply a field with a finite number of elements. These elements of integers ranging in value from 0 to $p - 1$.

to these to obtain the set of monomials for the interpolation:

$$\{1, x, y, z, xy, xz, yz, xyz, x^2, x^2y, x^2z, x^2yz\}$$

It is more clear to write these out explicitly as monomials in three variables:

$$\{x^0y^0z^0,\ x^0y^1z^0,\ x^1y^1z^0,\ x^1y^0z^1,\ x^1y^1z^1,\ x^2y^0z^0,\ x^2y^1z^0,\ x^2y^0z^1,\ x^2y^1z^1\}$$

This shows the set up of one row of the resulting sample matrix. Simply, for each point π_i, evaluate that point at each monomial in the set and use that as row i in the sample matrix.

Given any degree bounds, then we know the number of required point-value pairs needed to interpolate. Of course, the number of monomials in the function basis (and thus the size the of the linear system) is given by $n = \prod_{i=1}^{v}(d_i + 1)$. So, with degree bounds, we determine the set of monomials, the size of the linear system, and the number of required points, in order to set up a sample matrix just as in univariate interpolation. Using again the example with $d_x = 2$, $d_y = 1$, and $d_z = 1$, we get the linear system:

$$\begin{bmatrix} 1 & x|_{\pi_1} & y|_{\pi_1} & z|_{\pi_1} & \dots & x^2yz|_{\pi_1} \\ 1 & x|_{\pi_2} & y|_{\pi_2} & z|_{\pi_2} & \dots & x^2yz|_{\pi_2} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x|_{\pi_n} & y|_{\pi_n} & z|_{\pi_n} & \dots & x^2yz|_{\pi_n} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

Assuming the sample matrix is non-singular (and thus $n = m$), this system can easily be solved for the coefficients, α_i, and an interpolant uniquely constructed. But what if the sample matrix *is* singular? From univariate interpolation, we know that pairwise distinct points π_i are sufficient to ensure that the (Vandermonde) matrix be non-singular [68]. Unfortunately, it is not that simple for this matrices formed by multivariate monomials, as we will see in Section 5.2.2. But, assuming non-singularity let us discuss our implementation of dense interpolation and the technique of *early termination.* The mechanism used in early termination is also used to handle the non-singularity of the input matrix.

5.2.1 Implementing Early Termination for Multivariate Interpolation

Early termination is the process of terminating an interpolation early, once the interpolating polynomial does not change as more points are "added" to the interpolation [43, 45, 50]. Consider a subset of size k, $2 < k < n$, of the input points: $\{\pi_i \mid 1 \le i \le k\}$. If an interpolant is found to interpolate these k points, say f_k. That is, $f_k(\pi_i) = \beta_i$. Then it is possible to determine, with high probability, that this interpolant will interpolate all n points. If $f_k = f_{k+1} = \cdots = f_{k+\delta}$ then $f_k = f_n$ with probability which increases (exponentially, in the univariate case [50, Theorem 2.1]) with respect to δ. To the best of our knowledge, early termination strategies have only yet been implemented for sparse or univariate interpolations. In this section we present a simple scheme for early termination in dense multivariate interpolation.

Notice, that we say points are "added" to the interpolation. This is because in these existing early termination schemes interpolation is done via a black-box. The algorithm itself is responsible for choosing points and evaluating the underlying function. This is not a very general scheme. Sometimes a black-box in unavailable, or sometimes, the range of possible function evaluation points are from some (unknown) discrete set.

Our algorithm works more generically, working in a sort of producer-consumer pattern. This pattern in common in parallel programming where one thread produces data an another thread consumes it [54, Chapter 5]. It is also possible that both produce and both consume. The producer/consumer pattern does not have to be limited to the context of parallel synchronization. We make use of this pattern for our interpolation. The producer or *driver function* is the code supplying points and values, while the consumer or *interpolator* is our dense interpolation algorithm, accumulating the points and values. The consumer, in this case, also produces, namely the interpolant. Our interpolation algorithm thus works in a sort of state-based way, remembering all the previously "added" points. The rough pseudo-code for this early-terminating dense interpolation is presented in Algorithm 5.3.

Algorithm 5.3 denseInterpolation_Driver(d_j, π_i, β_i, δ)
$d_j \in \mathbb{N}^0$, $1 \leq j \leq v$, v is the number of variables,
$\pi_i \in \mathbb{D}^v$, $\beta_i \in \mathbb{D}$, $1 \leq i \leq n$, $n >= \prod_{j=1}^{v}(d_j + 1)$,
$\delta \geq 1$, the number of points to try in early termination;
returns $f \in \mathbb{D}[x_1, \ldots, x_v]$ or `Failure`

```
Interpolator := initializeInterpolator($v$, $d_j$)
for $i = 1$ ; $i \leq n$ ; $i$++ do
    addPoint(Interpolator, $\pi_i$, $\beta_i$)
    ($f_i$, valid) := getInterpolant(Interpolator)
    if valid then
        #Assuming $i + j \leq n$
        for $j = 1$ to $\delta$ do
            if $f_i(\pi_{i+j}) \neq \beta_{i+j}$ then
                valid := false
        if valid then
            break
    end
if valid then
    return $f_i$
else
    return Failure
```

`initializeInterpolator` is a simple function which just caches the number of variables and degree bounds for the interpolator's memory state. `addPoint` adds a point to the interpolator for consideration and `getInterpolant` asks the interpolator to try and interpolate all points added so far. This function returns a status, representing whether or not the interpolation succeeded with the i points added so far. The driver function then tests δ more points to determine if this function still matches the underlying function over more points.

What remains then it so explain how we can “interpolate all points added so far”. Essentially, the idea is to pretend the degree bounds are less than they are and interpolate a polynomial in that lower degree. Or, equivalently, choose a subset of the basis functions to use, explicitly setting the others to be have coefficients of 0. This works well for a monomial basis where the basis functions have some ordering and can be enumerated easily. In implementation, this choosing of a subset, works by simply blocking our sample matrix into an $i \times i$ sub-matrix for i points added so far, and an $n-i \times n-i$ identity sub-matrix. Similarly, the vector of values is set so the $n-i$ bottom elements are 0.

$$\left[\begin{array}{ccc|c} \phi_1(\pi_1) & \dots & \phi_i(\pi_1) & \\ \vdots & \ddots & \vdots & 0 \\ \phi_1(\pi_i) & \dots & \phi_i(\pi_i) & \\ \hline & & & \\ & 0 & & I \\ & & & \end{array}\right] \left[\begin{array}{c} \alpha_1 \\ \vdots \\ \alpha_i \\ \hline \alpha_{i+1} \\ \vdots \\ \alpha_n \end{array}\right] = \left[\begin{array}{c} \beta_1 \\ \vdots \\ \beta_i \\ \hline 0 \\ \vdots \\ 0 \end{array}\right]$$

In general, this system can be solved to find an interpolant for only the i points. However, testing even one extra point is usually enough to rule out false interpolants. For example, a line can always be used to interpolate two points, but adding a third point usually rules that out a line as an interpolant immediately. What remains is to deal with singularity in the sample matrix.

Our algorithm handles this simply by returning false for the valid state in `getInterpolant`. Moreover, during the process of attempting to solve a system of equations with a rank-deficient sample matrix, it is easy to determine the troublesome (linearly dependent) rows, say from its *row echelon form.*[3] In this case, as new points are added, instead of reducing the identity sub-matrix and adding a new row, we replace the troublesome rows with the new input point, hoping to make the matrix non-singular. But why do some points cause the sample matrix to become singular? This is not a simple question.

5.2.2 The Difficulties of Multivariate Interpolation

By the connection between Vandermonde matrices and univariate polynomial interpolation, we know that the sample matrix, in this case, is non-singular if each point, π_i, is unique. We say that the points $\pi_1, \dots, \pi_n$ are *poised* for the basis functions $\phi_1, \dots, \phi_n$ if the sample matrix produced by these points and functions is non-singular. Similarly we can say that the points are (multivariate) *polynomially poised* to mean the points are

[3] Row echelon form is the form resulting from a Gaussian elimination. That is, sort of an upper triangular form, where the bottom rows are 0 if the matrix is rank deficient.

posed for the basis functions of (multivariate) monomials. The number of points determine the number of monomials used, generally by limiting the degree of the monomials. Recall, that for v variables and a maximum total degree d, there are $n = \binom{v+d}{d}$ monomials. So, n points are polynomially poised means an interpolating polynomial exists whose degree is at most d.

There are several different theorems which determine if a set of points are polynomially poised. The first is the most succinct but least intuitive.

Theorem 5.1 *Given v variables and a maximum* total *degree of d, the $n = \binom{v+d}{d}$ points, $\pi_1, \ldots, \pi_n$ are polynomially poised if and only if they do not belong to a common algebraic hypersurface of degree $\leq d$* [64, Theorem 4.1].

Equivalently, as long as no non-zero polynomial, with degree no more than d, exists which vanishes at all points, then the points are polynomially poised. That is, there is no $p \neq 0 \in \mathbb{R}[x_1, \ldots, x_v]$, $deg(p) \leq d$ such that $p(\pi_i) = 0$ for $1 \leq i \leq n$.

The next theorem is more intuitive and is based on the aptly-called *geometric characterization condition* (condition GC) for a collection of points.

Definition 5.1 (Condition GC for the points $\pi_1, \ldots, \pi_n$)

For each point π_i there exists d distinct hyperplanes $p_{i1}, \ldots p_{id}$ such that:

(1) π_i does not lie on any of these hyperplanes,
(2) All other points lie on at least one of these hyperplanes.

Written mathematically:

$$\pi_j \in \bigcup_{\ell=1}^{d} p_{i\ell} \iff i \neq j,\ 1 \leq i, j \leq n$$

Notice that, in the univariate case, this condition reduces to the fact that all points must be distinct (since a hyperplane in one dimension is a single point). This corresponds to the result obtained via the singularity of the Vandermonde matrix. Using this geometric condition we have the following theorem regarding if points are polynomially poised.

Theorem 5.2 *Let Π be the points $\pi_1, \ldots, \pi_n \in R^v$. If Π satisfies Condition GC then Π is polynomially poised for a polynomial of degree $\leq d$* [22, Theorem 1].

For example, for a 2-dimensional collection of points and a maximum total degree of 2, the sample points are poised if, for each point, there exists two lines for which all other points lie on and the selected point does not. Figure 5.2 shows this example, a collection of points satisfying condition GC for $v = 2$, $d = 2$, and $n = 6 = \binom{v+d}{d}$.

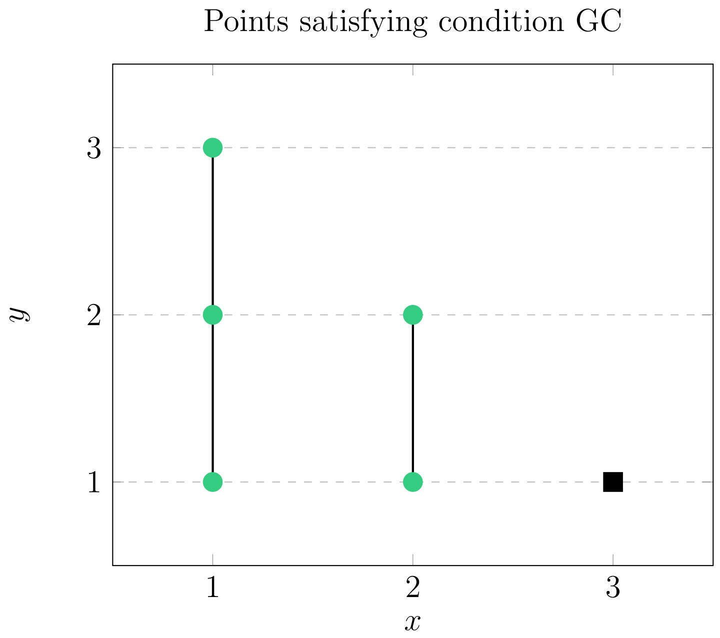


Figure 5.2: A collection of points satisfying condition GC. A selected point, and its corresponding hyperplanes, are highlighted in black.

The idea of geometric characterization can be generalized to hypersurfaces instead of hyperplanes. We call this the *generalized geometric characterization condition* (condition GGC).

Definition 5.2 (Condition GC for the points $\pi_1, \dots, \pi_n$)

For each point π_i there are d distinct hypersurfaces $s_{i1}, \dots, s_{id}$ with degrees $d_{i1}, \dots, d_{id}$ such that:

(1) π_i does not lie on any of these hypersurfaces,
(2) All other points lie on at least one of these hypersurfaces,
(3) $d_{i1} + \cdots + d_{id} = d$, for $1 \le i \le n$,

where a degree 0 hypersurface is regarded as the empty set. Written mathematically:

$$\pi_j \in \bigcup_{\ell=1}^{d} s_{i\ell} \iff i \neq j,\ 1 \le i, j \le n$$

Again, using this generalized condition, we obtain a generalized theorem regarding whether or not a collection of points are polynomially poised.

Theorem 5.3 *Let Π be the points $\pi_1, \dots, \pi_n \in \mathbb{R}^v$. If Π satisfies Condition GGC then Π is polynomially poised for a polynomial of degree $\le d$* [22, Theorem 7].

From the previous three theorems, we can see that choosing polynomially poised points is not an easy task, nor is it necessarily intuitive. If one performs the interpolation via a black-box using random points drawn from a large enough set, then the points will be poised with high probability (as we will see in the next section). But, if using user-supplied points, say points drawn from some experimental data, then it is likely the points will be colinear or at least organized in some regular pattern. Clearly, this regularity will likely cause many points to lay on the same hyperplane/hypersurface, leading to ill-poised collections of points. This problem is hard to avoid. However, as we have seen, our early-termination strategy provides an adaptive method for dealing with troublesome, ill-poised points.

5.2.3 Rational Function Interpolation

It is worthwhile to note that the same techniques of solving linear systems, along with early termination, as presented in the previous sections, can be adapted to also interpolate *rational functions*. A rational function (in $x_1, \ldots, x_v$ over $\mathbb{D}$) is nothing more than a fraction of polynomials (in $x_1, \ldots, x_v$ over $\mathbb{D}$). To be precise, they are the *field of fractions* of $\mathbb{D}[x_1, \ldots, x_v]$, $\mathbb{D}(x_1, \ldots, x_v) = \{{}^{p}/_{q} \mid p, q \in \mathbb{D}[x_1, \ldots, x_v]\}$ [32, Section 25.3]. In this section, we briefly detail one way to extend polynomial interpolation to rational functions.

For interpolating rational functions, one can solve a system of equations similar to polynomial interpolation. Say we have a rational function:

$$f(x_1, \ldots, x_v) = \frac{p(x_1, \ldots, x_v)}{q(x_1, \ldots, x_v)},$$

$$\begin{aligned} p(x_1, \ldots, x_v) &= \alpha_1 \phi_1(x_1, \ldots, x_v) + \cdots + \alpha_{m_1} \phi_{m_1}(x_1, \ldots, x_v), \\ q(x_1, \ldots, x_v) &= \alpha_{m_1+1} \psi_1(x_1, \ldots, x_v) + \cdots + \alpha_{m_1+m_2} \psi_{m_2}(x_1, \ldots, x_v), \end{aligned}$$

where m_1 and m_2 are chosen by the degree bounds on each variable (here we allow different degree bounds to be specified for the numerator p and the denominator q). For a single point-value pair, say (π_i, β_i), we can set up the equation

$$f(\pi_i) = \frac{p(\pi_i)}{q(\pi_i)} = \beta_i \implies p(\pi_i) - \beta_i q(\pi_i) = 0.$$

We can perform such a transformation to obtain many linear equations, and then, just as before, we can set up a system of linear equations for the monomial bases ϕ_i and ψ_j.

We denote the point-value pairs as (π_i, β_i) as usual.

$$\mathbf{Ax} = \mathbf{b}$$

$$\begin{bmatrix} \phi_1(\pi_1) & \dots & \phi_{m_1}(\pi_1) & -\beta_1\psi_1(\pi_1) & \dots & -\beta_1\psi_{m_2}(\pi_1) \\ \phi_1(\pi_2) & \dots & \phi_{m_1}(\pi_2) & -\beta_2\psi_1(\pi_2) & \dots & -\beta_2\psi_{m_2}(\pi_2) \\ \phi_1(\pi_3) & \dots & \phi_{m_1}(\pi_3) & -\beta_3\psi_1(\pi_3) & \dots & -\beta_3\psi_{m_2}(\pi_3) \\ \phi_1(\pi_4) & \dots & \phi_{m_1}(\pi_4) & -\beta_4\psi_1(\pi_4) & \dots & -\beta_4\psi_{m_2}(\pi_4) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \phi_1(\pi_n) & \dots & \phi_{m_1}(\pi_n) & -\beta_n\psi_1(\pi_n) & \dots & -\beta_n\psi_{m_2}(\pi_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_{m_1} \\ \alpha_{m_1+1} \\ \vdots \\ \alpha_{m_1+m_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Notice that in this system our right-hand vector, $\mathbf{b}$, is equal to the zero vector, $\mathbf{0}$. We no longer have the normal interpolation problem of solving $\mathbf{Ax} = \mathbf{b}$. We have a similar yet distinct problem, solving a *homogeneous* system of linear equations, $\mathbf{Ax} = \mathbf{0}$. Such a problem has two solutions: if $\mathbf{A}$ has full row rank then there is exactly one solution, the trivial solution, where $\mathbf{x} = \mathbf{0}$. Otherwise, there are infinitely many solutions [62]. Moreover, since we are interpolating a rational function now instead of a polynomial, it is always possible to multiply the rational function by $1 = {}^{e}/_{e},\ e \in \mathbb{D}$, to scale the non-zero coefficients of p and q however we see fit. Therefore, we *normalize* the denominator, q, by fixing one of its coefficients equal to 1. This can be accomplished by simply adding a row to the matrix which is all zeros except for a single 1, with the corresponding right-hand vector element also equal to 1. This forces our system of equations to be *heterogeneous.* As we look to normalize the denominator, we should choose a column corresponding to one of $\alpha_{m_1+1}, \dots, \alpha_{m_1+m_2}$ to hold the single non-zero element 1. Say we choose α_{m_1+1} to be 1, then we get the system of linear equations:

$$\begin{bmatrix} \phi_1(\pi_1) & \dots & \phi_{m_1}(\pi_1) & -\beta_1\psi_1(\pi_1) & \dots & -\beta_1\psi_{m_2}(\pi_1) \\ \phi_1(\pi_2) & \dots & \phi_{m_1}(\pi_2) & -\beta_2\psi_1(\pi_2) & \dots & -\beta_2\psi_{m_2}(\pi_2) \\ \phi_1(\pi_3) & \dots & \phi_{m_1}(\pi_3) & -\beta_3\psi_1(\pi_3) & \dots & -\beta_3\psi_{m_2}(\pi_3) \\ \phi_1(\pi_4) & \dots & \phi_{m_1}(\pi_4) & -\beta_4\psi_1(\pi_4) & \dots & -\beta_4\psi_{m_2}(\pi_4) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \phi_1(\pi_n) & \dots & \phi_{m_1}(\pi_n) & -\beta_n\psi_1(\pi_n) & \dots & -\beta_n\psi_{m_2}(\pi_n) \\ 0 & \dots & 0 & 1 & \dots & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_{m_1} \\ \alpha_{m_1+1} \\ \vdots \\ \alpha_{m_1+m_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

However, it is unfair to fix $\alpha_{m_1+1} = 1$ as we cannot guarantee that $\alpha_{m_1+1} \neq 0$ in the actual function f (we can only perform this arbitrary scaling for the non-zero terms of f). Therefore, we actually solve m_2 different systems of equations, fixing one of $\alpha_{m_1+j} = 1,\ 1 \leq j \leq m_2$, for each system, obtaining up to m_2 different solutions. Of course, by fixing one of the α_j to be 1, the resulting system of equations may become singular and produce no solution. Of the systems of equations which are non-singular, their results will be equal up to a scale factor. This is a result of having normalized a different term in the denominator polynomial q.

It is also possible that the matrix be non-square. Say when $n > m_1 + m_2$. Then the system can be solved in a least-squares sense (see Section 2.5.3). In this case there will indeed by m_2 different solutions, as a curve will always be fit to the data, although sometimes very poorly. Thus, we accept the solution which produces the minimal residual (as was defined in Section 2.5.3). The outliers (say when we set $\alpha_{m_1+j} = 1$ when it should in fact be 0) will have residuals which are clearly unacceptable and much higher than correct solutions.

5.3 Sparse Multivariate Interpolation

For multivariate polynomial interpolation (in symbolic computations), sparse algorithms have been the focus for decades [8, 24, 43, 50, 75, 76] and [74, Chapter 14]. Of course, this is important considering multivariate polynomials grow exponentially with the number of variables (Section 2.3). Unlike dense interpolation, these algorithms focus on interpolating functions given as black-boxes. That is, they need to determine the value of the underlying function for arbitrary points. These points may be randomly chosen, such as in probabilistic methods (Section 5.3.1), or be specific well-chosen points, such as in deterministic methods (Section 5.3.2).

5.3.1 Probabilistic Method

Zippel's work [76] presented a very important algorithm for sparse interpolation, influencing many other developments. This algorithm is probabilistic, by the fact that the resulting interpolant is correct with high probability, due to the randomness of the evaluation points chosen. Hence, this algorithm works using a black-box function, where the underlying function can be evaluated at any point.

Let the desired interpolant be $f \in \mathbb{K}[x_1, \ldots, x_v]$. The general idea of this algorithm is to view the interpolant recursively, as one in $\mathbb{K}[x_2, \ldots, x_v][x_1]$, and then interpolate the function as if it was in $\mathbb{K}[x_1]$, say f_1. With this univariate interpolant, view each of its coefficients as a polynomial in $\mathbb{K}[x_3, \ldots, x_v][x_2]$ and interpolate another univariate function as if it was in $\mathbb{K}[x_2]$ *for each coefficient*. The resulting polynomial is say $f_2 \in \mathbb{K}[x_3, \ldots, x_v][x_1, x_2]$. This process continues for each variable, interpolating one after the other. We call the step in which variable x_i is being interpolated *stage* i, with the result of stage i being the polynomial f_i. At each step, the size of the support of the interpolant f_i increases monotonically. Moreover, we call the support of f_i the *skeleton* of stage i, as we are not particularly concerned with the coefficients of stage $i < v$. The final interpolation of $f_v \in \mathbb{K}[x_1, \ldots, x_{v-1}][x_v]$ determines the scalar coefficients in $\mathbb{K}$ of f. All earlier interpolations merely determine the *support* for each variable.

The ability to interpolate polynomial coefficients like this is due to the fact that all variables of index greater than the one currently being interpolated are evaluated at fixed points. This is best shown by an example, as seen in [75, Section 4.1].

Consider wishing to obtain the interpolant $f \in \mathbb{K}[x, y, z]$ whose partial degrees are at most d for every variable. For ease of discussion, say this interpolant exactly matches the underlying black-box function. Begin by choosing an *initial point*, (x_0, y_0, z_0). Using some univariate interpolation algorithm, one can interpolate $f_1 = f(x, y_0, z_0) \in \mathbb{K}[x]$ from the points $(x_0, y_0, z_0), \ \ldots, \ (x_d, y_0, z_0)$ as inputs to the black-box. This interpolant f_1 will have some support in x. Say $f_1 = c_0x^5 + c_1x + c_2$ for $c_i \in \mathbb{K}$. Then really, this function is just $f(x, y_0, z_0) = g_0(y_0, z_0)x^5 + g_1(y_0, z_0)x + g_2(y_0, z_0)$ for some unknown polynomials $g_0, g_1, g_2 \in \mathbb{K}[y, z]$. Notice that the support of f with respect to x is fully determined as x^5, x^1, and x^0. We say that all other monomials in x with exponents different from these are *pruned*.

The next step is to interpolate polynomials in y. To do this, we could use the polynomials $f(x, y_0, z_0), f(x, y_1, z_0), \ldots, f(x, y_d, z_0)$. Since we know the support of f in x we know these polynomials would look like:

$$\begin{aligned}
f(x, y_0, z_0) &= c_0x^5 + c_1x + c_2 \\
f(x, y_1, z_0) &= c_3x^5 + c_4x + c_5 \\
f(x, y_2, z_0) &= c_6x^5 + c_7x + c_8 \\
&\vdots \\
f(x, y_d, z_0) &= c_{3d}x^5 + c_{3d+1}x + c_{3d+2}
\end{aligned}$$

From these c_i coefficients, we can consider the function g_0 evaluated at $d+1$ different points. Namely, $g_0(y_0, z_0) = c_0, \ g_0(y_1, z_0) = c_3, \ \ldots, \ g_0(y_d, z_0) = c_{3d}$. Similarly for g_1 and g_2. Thus, just as we interpolated f_1 as a univariate function in x, we could determine g_0, g_1, g_2 as a univariate functions in y. To determine these coefficients, $c_3, \ldots c_{3d+2}$, we evaluate more points using the black-box to create a system of linear equations, one system for each $f(x, y_i, z_0)$. This is much like the dense (univariate) interpolation algorithm. However, instead of using all monomials from degree 0 to d, we choose only the monomials present in the support of f with respect to x. Namely, x^5, x^1, x^0. Then, only 3 additional points are needed instead of $d+1$. For $f(x, y_1, z_0)$ this system of linear equations would look like:

$$\begin{bmatrix} x^5|_{x_1} & x^1|_{x_1} & x^0|_{x_1} \\ x^5|_{x_2} & x^1|_{x_2} & x^0|_{x_2} \\ x^5|_{x_3} & x^1|_{x_3} & x^0|_{x_3} \end{bmatrix} \begin{bmatrix} c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} f(x_1, y_1, z_0) \\ f(x_2, y_1, z_0) \\ f(x_3, y_1, z_0) \end{bmatrix}$$

Thus, we determine the coefficients $c_3, \ldots, c_{3d+2}$ using d systems of linear equations, and then interpolate 3 univariate functions in y. This results in some bivariate function f_2 in x and y. This process is then repeated to recover the variable z and obtain $f_3 = f$.

There are two things of importance to notice with this scheme.

(1) The values $x_0, x_1, \ldots, y_0, y_1, \ldots, z_0, z_1, \ldots$ are all randomly chosen, and
(2) The support obtained from each univariate interpolation is *assumed* to be correct.

Both of these aspects lead to this algorithm being probabilistic and non-deterministic. The critical aspect here is the assumption that the support is correct. Consider if $f_1 = g_0(y_0, z_0)x^5 + g_1(y_0, z_0)x + g_2(y_0, z_0) + g_3(y_0, z_0)x^3$, but, $g_3(y_0, z_0) = 0$. It would be assumed, simply from a poor choice of y_0 and z_0, that f_1 had fewer non-zero terms than it actually did. In such a case, the point (x_0, y_0, z_0) is called a *bad starting point*. Note that this phenomenon can occur during the univariate interpolation of any variable, not just from the initial point. To remedy these troubles, we choose random values from a large enough set to determine, with high probability, that the assumed support obtained for each variable is correct. This result comes from the fact that polynomials vanish at relatively few points compared to the number of possible evaluation points [75, Propositions 9, 12].

The precise probability is given by the following formula [76, Theorem 1]. Consider the true interpolant of a set of points, f, with v variables, each variable with degree no more than d, and having t non-zero terms. Then, the algorithm will return the *incorrect* result with probability less than ε if the random values chosen as the components of the randomly chosen initial point come from a set of size at least:

$$\frac{v^2dt}{\varepsilon}$$

If t, or some bound on t, is not known, then one can use the worst case bound $t = (d+1)^v$, the maximum number of terms in a degree d polynomial in v variables. The number of variables v appears in the numerator as we must choose v values to make up a single point.

We note that in Zippel's presentation, the set must actually be of size: vd^2t^2/ε [75, Proposition 10] to account for the possible singularity of the sample matrix due to randomly chosen points. As we will see in Algorithm 5.4 we remedy this just as was done in dense interpolation, by replacing rows in the sample matrix that are linearly dependent. Doing so, there is a higher chance that the matrix will initially be singular, but arithmetic throughout the entire algorithm is faster since the values come from a smaller set and are smaller in size (and hence fewer bits are needed to encode them).

Generally, the algorithm proceeds as follows:

(1) Determine the set from which random values are chosen as $\{1..B\} \subset \mathbb{Z}$, $B = v^2d(d+1)^v/\varepsilon$ (or use a bound on t, if known, instead of $(d+1)^v$),
(2) Choose an initial point $(x_{10}, x_{20}, \ldots, x_{v0})$,
(3) Interpolate $f_1(x_1, x_{20}, \ldots, x_{v0})$, and
(4) Perform stage i, for $2 \leq i \leq v$, to add each x_i to the interpolation, determining f_i.

We present *stage i* as Algorithm 5.4 and the complete probabilistic algorithm in Algorithm 5.5.

Algorithm 5.4 sparseInterpolation_StageI($\mathcal{B}$, d_i, f_{i-1}, π_0, B)
$\mathcal{B}$, the black-box for evaluation, d_i, the degree bound on variable x_i, f_{i-1}, the interpolant of stage $i-1$, π_0, the initial point $(x_{10}, \ldots, x_{v0})$, B the size of the random value set; returns f_i, the interpolant of stage i.

```
1: $(x_{i1}, \ldots x_{id_i})$ := pairwise distinct random values in $\{1..B\}$.
2: $S_{i-1} :=$ skeleton$(f_{i-1})$
3: $t := |S_{i-1}|$.
4: for $j = 1$ to $d_i$ do
5:     #Determine coefficients of $f(x_1, \ldots, x_{i-1}, x_{ij}, x_{(i+1)0}, \ldots, x_{v0})$
6:     for $k = 1$ to $t$ do
7:         $(y_0, \ldots, y_{i-1})$ := random values in $1..B$
8:         $\pi_k := (y_0, \ldots, y_{i-1}, x_{ij}, x_{(i+1)0}, \ldots, x_{v0})$
9:         $\beta_k := \mathcal{B}(\pi_k)$
10:     #Solve system of equations to determine coefficients.
11:     $\mathbf{A} \in \mathbb{K}^{t \times t}$ has one row for $S_{i-1}$ evaluated at the first $i-1$ components of $\pi_k$, $k = 1..t$
12:     while rank$(\mathbf{A}) < t$ do
13:         Repeat lines 7-11, replacing $\pi_k$, $\beta_k$ that resulted in linearly dependent rows in $\mathbf{A}$.
14:     $\mathbf{x} \in \mathbb{K}^{t \times 1} = (c_{jd_j}, \ldots, c_{jd_j+t-1})$ #the sought coefficients
15:     $\mathbf{b} \in \mathbb{K}^{t \times 1} = (\beta_k)$
16:     Solve $\mathbf{Ax} = \mathbf{b}$
17: end
18: #Interpolate $t$ polynomials in $x_i$ as the coefficients of $f_{i-1}$
19: for $k = 0$ to $t-1$ do
20:     #$x_{i0}$ is from $\pi_0$ while $c_0, \ldots, c_{t-1}$ are the coefficients of $f_{i-1}$
21:     $g_k(x_i) :=$ univariateInterpolation( $(x_{i0}, \ldots, x_{id_i})$, $(c_{jt+k}, 0 \leq j \leq d_i)$ ).
22: return $f_i := (S_{i-1}$ distributed over coefficient polynomials $g_k)$
```

Algorithm 5.5 sparseInterpolation($\mathcal{B}$, d_i, ε, T)
$\mathcal{B}$, the black-box for evaluation, d_i, the degree bound on variable x_i $(1 \leq i \leq v)$, ε, probability bound for incorrect interpolant, T, bound on number of terms in interpolant; returns f, the interpolating polynomial.

```
if $T < 0$ then
    $T := \prod_{i=1}^{v}(d_i + 1)$
$B := v^2 dT/\varepsilon$
$\pi_0 := (x_{10}, \ldots, x_{v0})$, a randomly chosen starting point with components in $\{1..B\}$
$(x_{10}, \ldots, x_{1d_1})$ := pairwise distinct random values in $\{1..B\}$
$\beta_j := \mathcal{B}(x_{1i}, x_{20}, \ldots, x_{v0})$ for $0 \leq j \leq d_i$.
$f_1 :=$univariateInterpolation( $(x_{10}, \ldots, x_{1d_1})$, $(\beta_0, \ldots, \beta_{d_1})$ )
for $i = 2$ to $v$ do
    $f_i :=$ sparseInterpolation_StageI($\mathcal{B}$, $d_i$, $f_{i-1}$, $\pi_0$, $B$)
return $f_v$
```

We summarize the departures our algorithm takes from Zippel's original probabilistic algorithm presented in [76] and [75]:

(1) There is no failure to interpolate when one of the linear systems has a singular matrix. We choose additional points to make it non-singular.
(2) From (1), the components of the randomly chosen initial point can be chosen from a much smaller sized set, one of size v^2dt/ε instead of vd^2t^2/ε.
(3) Degree bounds are given for each variable, d_i, instead of a single bound on all partial degrees, reducing the required number of systems to solve and size of univariate interpolations.

While probabilistic algorithms can be fast, usually much faster than deterministic ones, certainty is sometimes needed. In Section 5.3.3 we experiment and look at the speed of the probabilistic algorithm, while in the next section, we look at transforming the sparseInterpolation algorithm to be deterministic.

5.3.2 Deterministic Method

In [75, Section 6] and [74, Section 14.3] Zippel briefly proposes a strategy to transform his probabilistic sparse interpolation algorithm to be deterministic. However, discussion (or lack thereof) regarding this algorithm in the literature seems to be overshadowed by another deterministic algorithm presented 2 years prior by Ben-Or and Tiwari [8]. This work has been used extensively by others [24, 25, 34, 43–45, 50]. But, no attention has been given to the proposed deterministic variation of Zippel's algorithm.

Ben-Or and Tiwari's algorithm is not as straight-forward as Zippel's probabilistic one [74]. It relies on linear system solving, finding roots of polynomials, and integer factorization [8], while Zippel's uses only linear system solving and generic univariate interpolation. Moreover, Ben-Or and Tiwari's algorithm, at least in its original presentation, only works for integer polynomials [74, Section 14.4]. We do note, however, that Ben-Or and Tiwari's algorithm has asymptotically better performance and relies on fewer black-box evaluations [75, Section 1]. On the other hand, Zippel's algorithm is much simpler, and transforming Zippel's probabilistic algorithm to become deterministic is relatively straight forward. Here, we discuss this variation, and actually implement it. The deterministic and probabilistic variations are compared in Section 5.3.3.

There are two sources of probabilistic error in Zippel's sparse interpolation algorithm. The first results from randomly choosing evaluation points causing singularity in the matrix when solving the linear system (Algorithm 5.4, lines 6-11). The second is a result of assuming the support of the previous stage's interpolant is correct, where this is caused by the random choice of the initial point. As we saw in the previous section, we solved the singularity problem by simply using additional evaluation points. Another possible solution is to use evaluation points that, by construction, always create a non-singular sample matrix. This main idea is also used by Ben-Or and Tiwari's deterministic algorithm. It is given as Proposition 10 in [74]:

Proposition 5.1 *Let p be a polynomial in $U[x_1, \ldots, x_v]$ where U be a unique factorization domain with characteristic 0 and p has at most T terms. Then, if π_0 is a sequence of v different primes, then either p is identically 0 or, for one of π_0^j, $0 \le j < T$, $p(\pi_0^j) \neq 0$.*

Proof: Let m_i denote the i^{th} monomial of p evaluated at π_0 and c_i be the i^{th} coefficient of p for $1 \le i \le T$. We know the m_i are distinct as the components of π_0 are different primes, and so simply evaluating different monomials at that point maintains distinctness by unique factorization. Then, if p vanished at all of π_0^j we could set up the following system of equations:

$$\begin{aligned} c_1 + \cdots + c_T &= 0 \\ c_1 m_1 + \cdots + c_T m_T &= 0 \\ c_1 m_1^2 + \cdots + c_T m_T^2 &= 0 \\ &\vdots \\ c_1 m_1^{T-1} + \cdots + c_T m_T^{T-1} &= 0 \end{aligned}$$

This system forms a Vandermonde matrix in m_i. Since each m_i is distinct by construction, then the matrix is non-singular. Thus, all c_i must equal zero, and p is identically 0. The other option is that $p(\pi_0^j) \neq 0$ for some j. □

This proposition tells us that a *non-zero* polynomial cannot vanish at more than $T-1$ points. The same idea can be generalized to a collection of monomials:

Proposition 5.2 *For non-zero polynomials $p_1, \ldots, p_s \in U[x_1, \ldots, x_v]$ for a unique factorization domain U, where $\#(p_1) + \cdots + \#(p_s) = T$, if π_0 is a sequence of v different primes in U, then for some integer j, $0 \leq j \leq T-s$, all of $p_i(\pi_0^j)$ are non-zero.*

Proof: By Proposition 5.1 we know polynomial p_i cannot vanish at more than $\#(p_i) - 1$ points. Thus polynomials $p_1, \ldots, p_s$ can collectively vanish at no more than $T-s$ points,

$$\sum_{i=1}^{s} (\#(p_i) - 1) = \sum_{i=1}^{s} \#(p_i) - s = T - s,$$

when the components of those points are different primes. Since π_0^j, $0 \leq j \leq T-s$ is a set of $T-s+1$ points whose components are different primes, then for at least one j, all $p_i(\pi_0^j)$ must be non-zero. □

We note that in [75, Proposition 15] or [74, Proposition 102] this proposition was given as requiring $T+1$ points, based on polynomial p_i vanishing for at most $\#(p_i)$ points. But really, it vanishes for at most $\#(p_i) - 1$ points, leading to this tighter bound.

Using Proposition 5.2 we can derive a method to ensure the skeleton at each stage of the interpolation is correct. The procedure works inductively, assuming that stage $i-1$ produced the correct skeleton and then produces the correct skeleton for stage i. Consider the stage i interpolant, $f_i \in U[x_{i+1}, \ldots, x_v][x_1, \ldots, x_i]$. The skeleton of this interpolant will be incorrect if any coefficient in $U[x_{i+1}, \ldots x_v]$ vanishes at our choice of initial point. Thus, one simply performs stage i using many initial points to ensure no coefficient vanishes. Proposition 5.2 gives us the number of points required, $T-s$, where T is be the bound on the number of terms of $f = f_v$ and s is number of terms in f_i. Yet, the number of terms in f_i has not yet been determined. We can use the number of terms in f_{i-1} instead since as the number of terms increases monotonically with each interpolant.

Of course, this essentially means that each stage of the sparse interpolation algorithm must be done $O(T)$ times, greatly increasing the overall number of steps required for the algorithm in general. As we will see in Section 5.3.3, this does not necessarily result in a slower algorithm. But first, we present the modified, deterministic versions of Algorithms 5.4 and 5.5 as Algorithms 5.6 and 5.7, respectively.

Algorithm 5.6 SPARSEINTERPOLATION_STAGEI_DETERMINISTIC($\mathcal{B}$, d_i, T, f_{i-1})
$\mathcal{B}$, the black-box in v variables for evaluation, d_i, the degree bound on variable x_i, T, the bounds on number of terms in final interpolant, f_{i-1}, the interpolant of stage $i-1$, returns f_i, the interpolant of stage i..

```
$S_{i-1} :=$ skeleton($f_{i-1}$)
$t := |S_{i-1}|$.
#Perform stage $i$ a total of $T - t + 1$ times to ensure no coefficients vanish
for $\ell = 0$ to $T - t$ do
    #The initial point for this iteration of stage $i$. $p_m$ denotes the $m^{th}$ prime.
    $\pi_0 := (p_{i+1}^{\ell}, \dots, p_v^{\ell})$.
    for $j = 0$ to $d_i$ do
        #Determine coefficients of $f(x_1, \dots, x_{i-1}, x_{ij}, \pi_0)$
        $x_{ij} := p_i^j$.
        for $k = 1$ to $t$ do
            #Use structures points instead of random ones, to force non-singularity.
            $\pi_k := (p_1^{k-1}, p_2^{k-1}, \dots, p_{i-1}^{k-1}, x_{ij}, \pi_0)$
            $\beta_k := \mathcal{B}(\pi_k)$
        #Solve system of equations to determine coefficients.
        $\mathbf{A} \in \mathbb{K}^{t \times t}$ has one row for $S_{i-1}$ evaluated at the first $i-1$ components of $\pi_k$, $k = 1..t$
        $\mathbf{x} \in \mathbb{K}^{t \times 1} = (c_{jd_j}, \dots, c_{jd_j+t-1})$, the sought coefficients
        $\mathbf{b} \in \mathbb{K}^{t \times 1} = (\beta_k)$
        Solve $\mathbf{Ax} = \mathbf{b}$
    end
    #Interpolate $t$ polynomials in $x_i$ as the coefficients of $f_{i-1}$
    for $k = 0$ to $t - 1$ do
        $g_k(x_i) :=$ univariateInterpolation( $(x_{i0}, \dots, x_{id_i})$, $(c_{jt+k}, 0 \le j \le d_i)$ ).
        $f_{i\ell} := S_{i-1}$ distributed over coefficient polynomials $g_j$.
return $f_i := \max\limits_{f_{i\ell}}( \#(f_{i\ell}) )$
```

Algorithm 5.7 SPARSEINTERPOLATION_DETERMINISTIC($\mathcal{B}$, d_i, T)
$\mathcal{B}$, the black-box for evaluation, d_i, the degree bound on variable x_i $(1 \le i \le v)$, T, bound on number of terms in interpolant; returns f, the interpolating polynomial.

```
if $T < 0$ then
    $T := \prod_{i=1}^{v}(d_i + 1)$
for $\ell = 0$ to $T - 1$ do
    #$p_m$ denotes the $m^{th}$ prime.
    $\pi_0 := (p_2^{\ell}, \dots, p_v^{\ell})$
    for $j = 0$ to $d_1$ do
        $x_{1j} := p_1^j$
        $\beta_j := \mathcal{B}(x_{1j}, \pi_0)$
    $f_{1\ell} :=$ univariateInterpolation( $(x_{10}, \dots, x_{1d_1})$, $(\beta_0, \dots, \beta_{d_1})$ )
$f_1 := \max\limits_{f_{1\ell}}( \#(f_{i\ell}) )$
for $i = 2$ to $v$ do
    $f_i :=$ SPARSEINTERPOLATION_STATEI_DETERMINISTIC($\mathcal{B}$, $d_i$, $T$, $f_{i-1}$)
return $f_v$
```

5.3.3 Experimentation

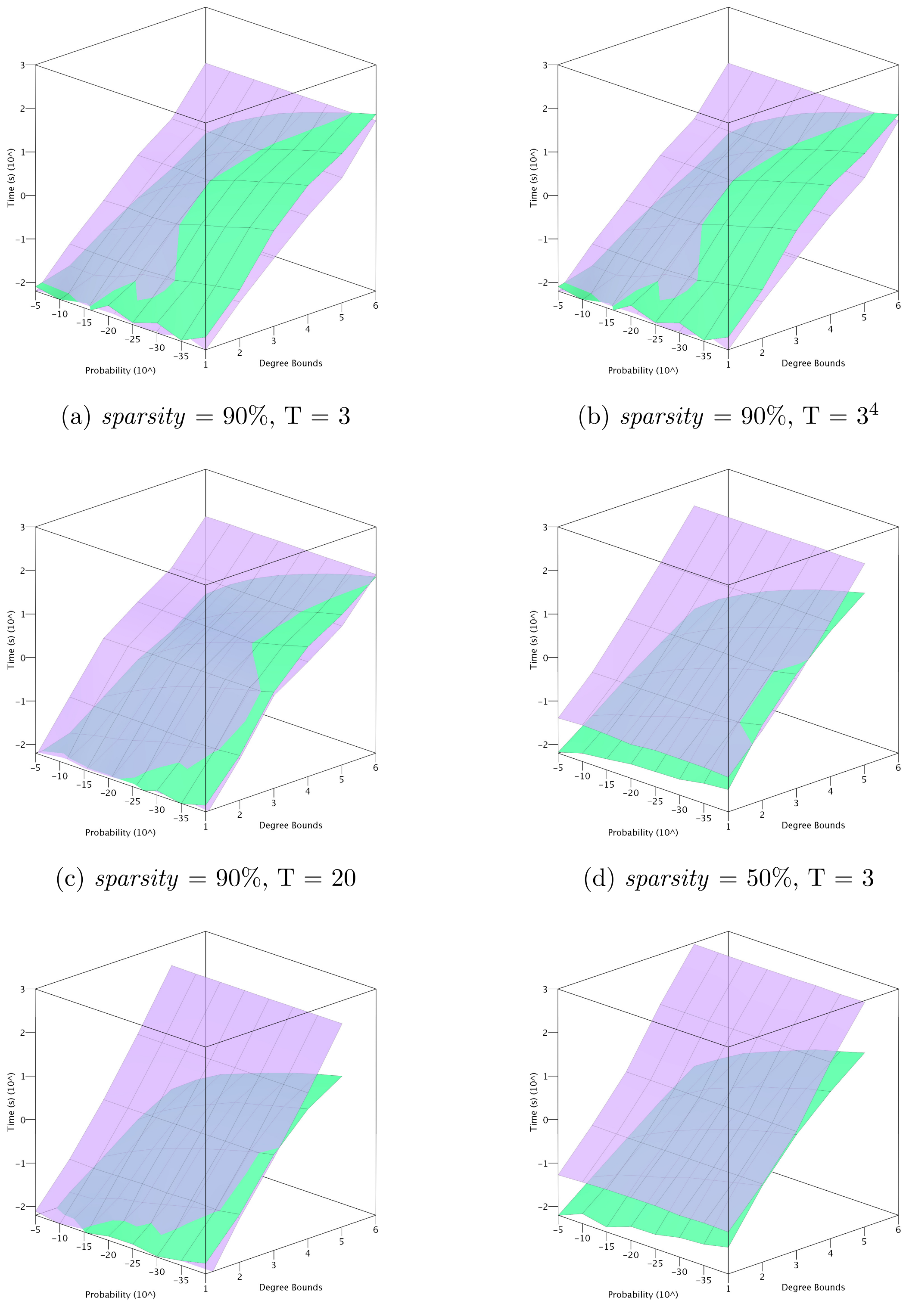
Our sparse multivariate interpolation algorithms work essentially by only univariate interpolation and linear system solving. Section 5.1 showed that our implementation of univariate system solving is quite optimized, and we know the library we use for linear system solving is also highly optimized (see Section 5.2). These two together form a basis of implementation for both variations of sparse interpolation algorithms.

Naturally, one thinks of probabilistic algorithms as running faster than deterministic ones. However, if the probability for error becomes small enough, this may cause large slow-downs. In the case of probabilistic sparse interpolation, the probability of error influences the size of the set from which random numbers are drawn. The actual number of steps in the algorithm does not change with the probability, but the size of the numbers do, and thus the cost of the arithmetic with them. The size of this set is $B = v^2 dt\varepsilon^{-1}$, with v being the number of variables, d being the bound on partial degrees in the interpolant, t being the number of terms in the interpolant, and ε, the maximum probability of returning an incorrect result. Using the integers $\{1..B\}$ as the set for random numbers, we can see these numbers require $O(\lg(B)) = O(\lg(v^2 dt\varepsilon^{-1}))$ bits to be stored. For example, consider $v = 3$, $d = 5$, $t = 20$. Then with $\varepsilon > 10^{-16}$ these integers become multi-precision, greatly slowing down the arithmetic.

In this experimentation we used *sparsity* to indicate the percentage of zero-terms in the polynomial, when compared to the maximum possible number of terms. Recall, the maximum number of terms in a polynomial in v variables each with maximum partial degree d is $(d+1)^v$. Further, the definition of T varies slightly here from its definition used in the previous sections. Instead of T being a bound on the number of terms, the bound is rather calculated as $\min(\#(f) + T,\ (d+1)^v)$ where f is the underlying polynomial being interpolated. Also, if $T < 0$ then the bound is taken to be $(d+1)^v$.

In Figures 5.3a-5.3f we compare the probabilistic and deterministic algorithms varying many different parameters. Within a single plot we vary the degree bound, d, and the probability bound, ε. The green surfaces are the running time of the probabilistic algorithm, while the transparent purple surfaces are the running time of the deterministic algorithm. For a fixed d, the probabilistic algorithm increases as $\lg(\varepsilon)$. Notice that both the probability (ε) axis and running time axis are log-scaled.

Comparing multiple plots, the left column shows the effect of varying the bounds on the number of terms in the interpolant. If the bound is tight, then the deterministic algorithm performs more favourably than if it is loose. The right column shows the impact of varying the sparsity of the underlying polynomial (and thus the sparsity of the generated interpolant). The more dense the interpolant, the more coefficient polynomials must be computed for each stage of the interpolation. Surprisingly, the probabilistic algorithm is not always the winner. When the bound on the number of terms in the interpolant is somewhat tight, and the probability of an incorrect solution is only moderately low (10^{-20} or smaller), the deterministic algorithm outperforms the probabilistic.



(a) *sparsity* = 90%, T = 3

(b) *sparsity* = 90%, T = 3[4]

(c) *sparsity* = 90%, T = 20

(d) *sparsity* = 50%, T = 3

(e) *sparsity* = 90%, T = -1

(f) *sparsity* = 10%, T = 3

Figure 5.3: Comparing probabilistic (green surfaces) and deterministic (transparent purple surfaces) sparse interpolation algorithms. The number of variables is fixed at 3.

[4] This data is intentionally repeated for visual comparison within the column.

5.4 Numerical Interpolation (& Curve Fitting)

In the previous two sections, we studied both dense and sparse polynomial interpolation using exact arithmetic. For completion, we also present here a numerical flavour of polynomial interpolation. Simply put, numerical interpolation is sometimes needed in circumstances where exact arithmetic will fail or is not well suited. These circumstances occur frequently in practice, with real-world observations and experimentation, where the precision of a measurement could be limited by the measuring device itself, or more generally, the measurements are perturbed by some noise.

With such *noisy data*, exact arithmetic is ill-suited. The noise will be maintained throughout an algorithm and also appear in the result. The issue with this is that when two values are intended to be the same, or at least converge toward the same value, they never will in the presence of noise using exact arithmetic. Numerical methods however can easily deal with such noise.

Using the same algorithm, numerical methods for interpolation can also solve the problem of curve fitting. Further still, one algorithm can be used which handles interpolation, curve fitting, as well as rank-deficient systems which can often occur (see Section 5.2.2). This algorithm is solving a system of linear equations by singular value decomposition (Section 2.2.6).

Our numerical interpolation scheme is exactly like that of our dense interpolation (Section 5.2), with the only difference being that the linear system created is solved using numerical methods instead of exact ones. Indeed, aspects like early termination can also be employed in the exact same manner. Therefore, we limit the discussion here only the the differences in solving the linear system.

We begin by setting up a system of linear equations just as in dense interpolation. We use the multivariate monomials as basis functions. With v variables and partial degree bounds for each variable, d_i, $1 \leq i \leq v$, the set of monomials becomes fixed. Say this set contains m monomials. Then, from a set of n point-value pairs (π_i, β_i), we obtain the system:

$$\mathbf{Ax} = \mathbf{b}$$
$$\begin{bmatrix} \phi_1(\pi_1) & \phi_2(\pi_1) & \dots & \phi_m(\pi_1) \\ \phi_1(\pi_1) & \phi_2(\pi_2) & \dots & \phi_m(\pi_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\pi_n) & \phi_2(\pi_n) & \dots & \phi_m(\pi_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

The solution of this system yields a function, f, which is a linear combination of the ϕ_j functions with coefficients α_j. If $m = n$ the function is an interpolant, precisely interpolating the points. That is, $f(\pi_i) = \beta_i$. If $m > n$, then the a solution can still be obtained in a least-squares sense giving a curve fit rather than an interpolant.

Using orthogonal factorizations, we can easily solve this system in either case: interpolation or curve fitting. For interpolation, simply obtain the factorization and solve the system directly, by say backward substitution. For the least-squares variation, if the matrix has full rank, then the system can also be solved directly as the resulting factorization will have a triangular (or diagonal) factor. QR-factorization is one such orthogonal factorization method. It is both efficient and numerically stable. However, it will fail if the original matrix $\mathbf{A}$ is column rank deficient, much like the method by normal equations (Section 2.5.3). Further still, it is susceptible to instability if $\mathbf{A}$ is numerically rank deficient [49]. This presents a problem as our sample matrix can often be (numerically) rank-deficient in this case of multivariate interpolation.

Singular value decomposition presents a solution. By SVD it is possible to obtain a least squares solution regardless of the rank of $\mathbf{A}$ [23]. Of course, if $\mathbf{A}$ is rank deficient, there is no unique solution to the least squares problem, but by SVD we can compute the *minimum norm solution* using the 2-norm.

We sketch, very simply, the method to solve linear least squares by SVD with $\mathbf{A} \in \mathbb{R}^{n \times m}$, $n \geq m$. Of course, if $n = m$ and $rank(\mathbf{A}) = n$, then we are solving the interpolation problem and the unique solution will have a residual of 0 (up to any numerical discrepancies).

$$\begin{aligned}
\mathbf{x} &= \min_{\mathbf{x}} ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 \\
&= \min_{\mathbf{x}} ||\mathbf{b} - \mathbf{U}\mathbf{D}\mathbf{V}^T\mathbf{x}||_2^2 \\
&= \min_{\mathbf{x}} ||\mathbf{U}^T\mathbf{b} - \mathbf{D}\mathbf{V}^T\mathbf{x}||_2^2
\end{aligned}$$

Letting $\mathbf{V}^T\mathbf{x} = \mathbf{z}$ and $\mathbf{U}^T\mathbf{b} = \mathbf{w}$ gives $\min_{\mathbf{z}} ||\mathbf{w} - \mathbf{D}\mathbf{z}||_2^2$. Since $\mathbf{D} = diag(\sigma_1, \ldots, \sigma_m)$ is a diagonal matrix, we can easily solve for $\mathbf{z}$. Say $rank(\mathbf{D}) = r \leq n$. The, we proceed by $z_k = {}^{w_k}/_{\sigma_k},\ k \leq r$. If $r < n$ then we are looking for the minimal norm solution, and we look to minimize $\mathbf{z}$. Thus, we set $z_k = 0,\ r + 1 \leq k \leq n$. Lastly, we obtain the solution in $\mathbf{x}$ by simply multiplying $\mathbf{z}$ by the orthonormal matrix $\mathbf{V}$:

$$\begin{aligned}
\mathbf{V}^T\mathbf{x} &= \mathbf{z} \\
\mathbf{V}\mathbf{V}^T\mathbf{x} &= \mathbf{V}\mathbf{z} \\
\mathbf{x} &= \mathbf{V}\mathbf{z}
\end{aligned}$$

Note that it is possible to solve a system of equations by SVD even for $n < m$. That is, fewer point-value pairs than the number of basis functions. We refer the reader to [23, 33, 49] for detailed methods and solutions in the general case.

Chapter 6

Conclusions and Future Work

The basic arithmetic of polynomials, along with their interpolation, are fundamental operations in computer algebra. These operations form the foundations of many more advanced and applicable algorithms. Of course, algorithms without arithmetic would surely be impossible. Therefore, we have given highly optimized algorithms and implementations for sparse polynomial arithmetic and interpolation to provide a high-performance basis from which more interesting algorithms can be developed. Particular interest is in polynomial system solving via triangular decompositions and regular chains.

Our concerns for high-performance stem from the knowledge of the current state of modern computer architectures, including their limitations. The *processor-memory* gap is one such limitation that was not of much concern before the 1980s. This issue can be addressed by optimizing for cache complexity and making smart choices for data locality in the development of data structures and algorithms. This motivation differs from some historical algorithms – where historical in the sense of computing technology is upwards of 50 years – which attempted to minimize the amount of memory used. This was accomplished by representing polynomials sparsely, without particular concern for *how* memory was used. Therefore, by using the classical sparse algorithms as a starting point, we have improved and adapted them for modern architectures.

In Chapter 3 we presented data structures for encoding sparse polynomials with particular interest in data locality and cache complexity, displaying results which indicate the relations between cache complexity and running time. In Chapter 4 we put these data structures to use as we adapt and optimize Johnson's sparse arithmetic algorithms [42] to modern day computers. In this chapter we also propose a new algorithm for sparse pseudo-division. We prove the algorithm's correctness and performance by implementing this algorithm and comparing its results against the classical, yet still well-implemented, algorithm as well as the commercial implementation within Maple.

Lastly, in Chapter 5, we provide various algorithms and implementation techniques for polynomial interpolation and curve fitting. In particular, we obtain a highly optimized univariate Lagrange interpolation algorithm, dense multivariate interpolation, and probabilistic sparse multivariate interpolation. We also provide an implementation of deterministic sparse multivariate interpolation, an algorithm which seems to have only ever been proposed and never implemented. Thus, we also give detailed experimentation to compare the probabilistic and deterministic variations, concluding in which situations which algorithm is optimal. We close this chapter by presenting numerical methods – a departure from the symbolic computation throughout this thesis – in order to solve problems with *noisy data* that strictly exact arithmetic admittedly cannot solve.

With these algorithms and optimal implementations, we look to better the Basic Polynomial Algebra Subprograms (BPAS) library in its quest for a high-performance implementation of triangular decomposition via regular chains. Our foundational algorithms will surely support this. But yet more is still possible. We wish to parallelize our algorithms, as others have attempted [56, 58], in order to gain even more performance on modern architectures with multi-core processors.

However, the BPAS library could be yet further improved beyond just improving its already fast arithmetic. One could hope to improve its software engineering aspects as well as end-user usability. In 1984, Zippel published an open letter in the ACM SIGSAM Bulletin titled *The Future of Computer Algebra* [77]. In this letter he praises the algorithmic and implementation developments and improvements within the computer algebra community over that past decade. However, he points to a critical area of improvement. He says, "we must build systems better organized and tuned to solve problems in particular scientific and engineering domains." He also notes that the solutions for this come from "an emphasis on building tools to build systems, rather than building systems themselves". While some improvements have been made on this front such as with Maple, Mathematica and Matlab, these systems are proprietary and expensive. There lacks a strong foundational library for computer algebra upon which other system can be built. Think of a computer algebra equivalent of the Basic Linear Algebra Subprograms (BLAS) library [11] – the ubiquitous library for linear algebra.

In mathematical and academic software, one major downfall is the negligence towards software engineering aspects, usability, and end-user considerations. Researchers are more interested in creating and implementing innovative and complex algorithms. Software maintainability, robustness, documentation, and ease of use, are all considerations which are historically missing from mathematical and research-based software. Unfortunately, this creates a massive barrier between library developers and possible users. Such barriers are artificially limiting the advancement of other works, both academic and industrial, which could make use of such new and advanced algorithms. It is our goal then to continue the progress of high-performance within BPAS using well-designed algorithms, efficient data structures, and parallelization all the while being proactive in the software engineering and usability aspects of this mathematical, high-performance library. The overlap between high-performance computing, mathematics, and software engineering is a very niche, yet undoubtedly important, area of research that is missing much needed care and attention.

Bibliography

[1] William W Adams and Philippe Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Soc., 1994.

[2] Mohammadali Asadi, Alexander Brandt, Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Davood Mohajerani, Robert H. C. Moir, Marc Moreno Maza, Ning Xie, and Yuzhen Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. `http://www.bpaslib.org`. 2018.

[3] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. "Sparse Polynomial Arithmetic with the BPAS Library (forthcoming)". In: *Computer Algebra in Scientific Computing (CASC) 2018. Proceedings of the 20th International Workshop on*. Springer. 2018.

[4] Philippe Aubry, Daniel Lazard, and Marc Moreno Maza. "On the theories of triangular sets". In: *Journal of Symbolic Computation* 28.1-2 (1999), pp. 105–124.

[5] Robert G. Bartle. *The elements of real analysis*. 2nd. Wiley New York, 1964.

[6] Thomas Becker and Volker Weispfenning. *Gröbner bases, volume 141 of Graduate Texts in Mathematics*. 1993.

[7] Bernhard Beckermann. "The condition number of real Vandermonde, Krylov and positive definite Hankel matrices". In: *Numerische Mathematik* 85.4 (2000), pp. 553–577.

[8] 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*. ACM. 1988, pp. 301–309.

[9] L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K.M. Heal, G. Labahn, J. P. May, J. McCarron, M. B. Monagan, D. Ohashi, and S. M. Vorkoetter. *Maple Programming Guide*. `www.maplesoft.com/documentation_center/maple2018/ProgrammingGuide.pdf`. 2018.

[10] Jean-Paul Berrut and Lloyd N Trefethen. "Barycentric Lagrange interpolation". In: *SIAM review* 46.3 (2004), pp. 501–517.

[11] *BLAS (Basic Linear Algebra Subprograms)*. `http://www.netlib.org/blas/`. 2017.

[12] M. Bodrato and A. Zanoni. "Integer and polynomial multiplication: towards optimal Toom-Cook matrices". In: *ISSAC*. 2007, pp. 17–24.

[13] W Boege, Rüdiger Gebauer, and Heinz Kredel. "Some examples for solving systems of algebraic equations by calculating Groebner bases". In: *Journal of Symbolic Computation* 2.1 (1986), pp. 83–98.

[14] Wieb Bosma, John Cannon, and Catherine Playoust. “The Magma algebra system I: The user language”. In: *Journal of Symbolic Computation* 24.3-4 (1997), pp. 235–265.

[16] M. Bronstein, M. Moreno Maza, and S.M. Watt. “Generic programming techniques in ALDOR”. In: *Proceedings of AWFS 2007.* 2007, pp. 72–77.

[17] Bruno Buchberger and Franz Winkler, eds. *Gröbner bases and applications.* Vol. 251. Cambridge University Press, 1998.

[18] Richard L. Burden and J. Douglas Faires. *Numerical Analysis.* 9th. Brooks/Cole, Boston, MA, 2011.

[19] Changbo Chen and Marc Moreno Maza. “Algorithms for computing triangular decomposition of polynomial systems”. In: *J. Symb. Comput.* 47.6 (2012), pp. 610–642. DOI: `10.1016/j.jsc.2011.12.023`. URL: `https://doi.org/10.1016/j.jsc.2011.12.023`.

[20] Zhuliang Chen and Arne Storjohann. “A BLAS based C library for exact linear algebra on integer matrices”. In: *Proceedings of the 2005 international symposium on Symbolic and algebraic computation.* ACM. 2005, pp. 92–99.

[21] Zhuliang Chen, Arne Storjohann, and Cory Fletcher. *IML - Integer Matrix Library.* `cs.uwaterloo.ca/~astorjoh/iml.html`. 2015.

[22] Kwok Chiu Chung and Te Hai Yao. “On lattices admitting unique Lagrange interpolations”. In: *SIAM Journal on Numerical Analysis* 14.4 (1977), pp. 735–743.

[23] Robert M Corless and Nicolas Fillion. *A graduate introduction to numerical methods.* Springer, 2013.

[24] Annie Cuyt and Wen-shin Lee. “A new algorithm for sparse interpolation of multivariate polynomials”. In: *Theoretical computer science.-Amsterdam* 409.2 (2008), pp. 180–185.

[25] Annie Cuyt and Wen-shin Lee. “Sparse interpolation of multivariate rational functions”. In: *Theoretical Computer Science* 412.16 (2011), pp. 1445–1456.

[26] Carl De Boor and Amos Ron. “On multivariate polynomial interpolation”. In: *Constructive Approximation* 6.3 (1990), pp. 287–302.

[27] Richard Fateman. “Comparing the speed of programs for sparse polynomial multiplication”. In: *ACM SIGSAM Bulletin* 37.1 (2003), pp. 4–15.

[28] Jean-Charles Faugere, Patrizia Gianni, Daniel Lazard, and Teo Mora. “Efficient computation of zero-dimensional Gröbner bases by change of ordering”. In: *Journal of Symbolic Computation* 16.4 (1993), pp. 329–344.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-oblivious algorithms”. In: *40th Annual Symposium on Foundations of Computer Science.* 1999, pp. 285–297. DOI: `10.1109/SFFCS.1999.814600`.

[30] Ralf Fröberg. *An introduction to Gröbner bases.* John Wiley & Sons, 1997.

[31] Mickaël Gastineau and Jacques Laskar. “Trip: a computer algebra system dedicated to celestial mechanics and perturbation series”. In: *ACM Communications in Computer Algebra* 44.3/4 (2011), pp. 194–197.

[32] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* 2nd ed. NY, USA: Cambridge University Press, 2003. ISBN: 0521826462.

[33] James E Gentle, Wolfgang Karl Härdle, and Yuichi Mori. *Handbook of computational statistics: concepts and methods.* Springer Science & Business Media, 2012.

[34] Mark Giesbrecht, George Labahn, and Wen-shin Lee. "Symbolic–numeric sparse interpolation of multivariate polynomials". In: *Journal of Symbolic Computation* 44.8 (2009), pp. 943–959.

[35] Alessandro Giovini and Gianfranco Niesi. "CoCoA: a user-friendly system for commutative algebra". In: *International Symposium on Design and Implementation of Symbolic Computation Systems*. Springer. 1990, pp. 20–29.

[36] Gaston H Gonnet and J Ian Munro. "Heaps on heaps". In: *SIAM Journal on Computing* 15.4 (1986), pp. 964–971.

[37] Torbjørn Granlund et al. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Free Software Foundation, Inc, 2015.

[38] Andrew D Hall Jr. "The ALTRAN system for rational function manipulation-a survey". In: *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. ACM. 1971, pp. 153–157.

[39] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. V. 2.4.3, `http://flintlib.org`.

[40] Joris van der Hoeven and Grégoire Lecerf. "On the bit-complexity of sparse polynomial and series multiplication". In: *J. Symb. Comput.* 50 (2013), pp. 227–254. DOI: `10.1016/j.jsc.2012.06.004`. URL: `https://doi.org/10.1016/j.jsc.2012.06.004`.

[41] Bing-Chao Huang and Michael A Langston. "Practical in-place merging". In: *Communications of the ACM* 31.3 (1988), pp. 348–352.

[42] Stephen C Johnson. "Sparse polynomial arithmetic". In: *ACM SIGSAM Bulletin* 8.3 (1974), pp. 63–71.

[43] Erich Kaltofen and Wen-shin Lee. "Early termination in sparse interpolation algorithms". In: *Journal of Symbolic Computation* 36.3-4 (2003), pp. 365–400.

[44] Erich Kaltofen and Lakshman Yagati. "Improved sparse multivariate polynomial interpolation algorithms". In: *International Symposium on Symbolic and Algebraic Computation*. Springer. 1988, pp. 467–474.

[45] 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*. ACM. 2007, pp. 203–210.

[46] Anatolii Alekseevich Karatsuba and Yu P Ofman. "Multiplication of many-digital numbers by automatic computers". In: *Doklady Akademii Nauk*. Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.

[47] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. "Practical in-place mergesort". In: *Nord. J. Comput.* 3.1 (1996), pp. 27–40.

[48] Donald E Knuth. *The Art of Programming, vol. 2, Semi-Numerical Algorithms*. Addison Wesley, Reading, MA, 1981.

[49] Do Q Lee. *Numerically efficient methods for solving least squares problems*. 2012.

[50] Wen-shin Lee. "Early termination strategies in sparse interpolation algorithms". PhD thesis. Raleigh, NC, USA: North Carolina State University, 2001.

[51] Solomon Lefschetz. *Algebraic Geometry*. Dover, 2005.

[52] Charles E. Leiserson. "Cilk". In: *Encyclopedia of Parallel Computing*. 2011, pp. 273–288. DOI: `10.1007/978-0-387-09766-4\_289`. URL: `https://doi.org/10.1007/978-0-387-09766-4%5C_289`.

[53] Francois Lemaire, Marc Moreno Maza, and Yuzhen Xie. "The RegularChains library in MAPLE". In: *ACM SIGSAM Bulletin* 39.3 (2005), pp. 96–97. DOI: 10.1145/1113439.1113456. URL: http://doi.acm.org/10.1145/1113439.1113456.

[54] Calvin Lin and Lawrence Snyder. *Principles of parallel programming.* Pearson, 2009.

[55] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[56] M. B. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps". In: *ISSAC*. 2009, pp. 263–270.

[57] Michael B. Monagan and Roman Pearce. "Sparse polynomial division using a heap". In: *J. Symb. Comput.* 46.7 (2011), pp. 807–822. DOI: 10.1016/j.jsc.2010.08.014. URL: https://doi.org/10.1016/j.jsc.2010.08.014.

[58] Michael Monagan and Roman Pearce. "Parallel sparse polynomial division using heaps". In: *Proceedings of PASCO 2010.* ACM. 2010, pp. 105–111.

[59] Michael Monagan and Roman Pearce. "Polynomial division using dynamic arrays, heaps, and packed exponent vectors". In: *CASC 2007.* Springer. 2007, pp. 295–315.

[60] Michael Monagan and Roman Pearce. "The design of Maple's sum-of-products and POLY data structures for representing mathematical objects". In: *ACM Communications in Computer Algebra* 48.3/4 (2015), pp. 166–186.

[61] Harvey Motulsky and Arthur Christopoulos. *Fitting models to biological data using linear and nonlinear regression: a practical guide to curve fitting.* Oxford University Press, 2004.

[62] Ferrante Neri et al. *Linear algebra for computational sciences and engineering.* Springer, 2016.

[63] J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. ISSN: 1542-7730.

[64] Peter J Olver. "On multivariate interpolation". In: *Studies in Applied Mathematics* 116.2 (2006), pp. 201–240.

[65] Gert-Martin Pfister, Gerhard Greuel, and Hans Schonemann. "Singular-A computer algebra system for polynomial computations". In: *Symbolic Computation and Automated Reasoning: The CALCULEMUS-2000 Symposium.* AK Peters/CRC Press. 2001, p. 227.

[66] Harald Prokop. "Cache-Oblivious Algorithms". MA thesis. Cambridge, MA: Massachusetts Institute of Technology, 1999.

[67] Kamron Saniee. "A Simple Expression for Multivariate Lagrange Interpolation". In: *SIAM Undergraduate Research Online (SIURO)* 1 (1 2008), pp. 1–9.

[68] Thomas Sauer and Yuan Xu. "On multivariate Lagrange interpolation". In: *Mathematics of Computation* 64.211 (1995), pp. 1147–1170.

[69] A. Schönhage and V. Strassen. “Schnelle Multiplikation großer Zahlen”. In: *Computing* 7.3-4 (1971), pp. 281–292.

[70] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Addison-Wesley, 2011. Chap. 2.

[71] Victor Shoup. *NTL: A library for doing number theory*. `www.shoup.net/ntl/`.

[72] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley, 1991.

[73] Wm A Wulf and Sally A McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[74] Richard Zippel. *Effective polynomial computation*. Vol. 241. Springer Science & Business Media, 2012.

[75] Richard Zippel. “Interpolating polynomials from their values”. In: *Journal of Symbolic Computation* 9.3 (1990), pp. 375–403.

[76] Richard Zippel. “Probabilistic algorithms for sparse polynomials”. In: *Symbolic and algebraic computation*. Springer, 1979, pp. 216–226.

[77] Richard Zippel. “The future of computer algebra”. In: *ACM SIGSAM Bulletin* 18.2 (1984), pp. 6–7.

Curriculum Vitae

Name:	Alexander Brandt
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2017 - 2018 M.Sc. Memorial University of Newfoundland St. John's, NL 2013 - 2017 B.Sc. (Hons.)
Related Work Experience:	Teaching Assistant University of Western Ontario 2017 - 2018 Research Assistant Ontario Research Center for Computer Algebra University of Western Ontario 2017 - 2018 Software Developer Whitecap Scientific Corporation 2016 - 2018 Research Assistant Memorial University of Newfoundland 2016 - 2017

Publications:

- Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. "Sparse Polynomial Arithmetic with the BPAS Library (forthcoming)". In: *Computer Algebra in Scientific Computing (CASC) 2018. Proceedings of the 20th International Workshop on.* Springer. 2018

- Alexander Brandt. "On the formalization and computational complexity of software modularization". Honours Thesis. St. John's, NL, Canada: Memorial University of Newfoundland, 2017