DELAYED POLYNOMIAL ARITHMETIC AND APPLICATIONS

by

Paul Vrbik
B.Sc., McMaster University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Mathematics

SIMON FRASER UNIVERSITY
Fall 2008

APPROVAL

Name: Paul Vrbik

Degree: Master of Science

Title of thesis: Delayed Polynomial Arithmetic and Applications

Examining Committee: Dr. Jason Bell
Assistant Professor, Mathematics
Chair

Dr. Michael Monagan
Professor, Mathematics
Senior Supervisor

Dr. Petr Lisonek
Associate Professor, Mathematics

Dr. Nils Bruin
Assistant Professor, Mathematics
External Examiner

Date Approved: December 3, 2008

Abstract

The goal of this thesis is to develop an environment for doing delayed polynomial arithmetic. We present the various known 'heap' methods for multiplication and division and adapt them to create a high performance implementation in C.

We also present a storage-minimizing variant of this package that allows us to address some fundamental problems with Bareiss' fraction-free method for calculating determinants and the subresultant algorithm. We show that the memory usage for both of these algorithms can be linearly related to the size of the output instead of the intermediate values.

To my mom for always asking if I have enough to eat.
To my dad for twenty-four years of tutoring.
To Irene for keeping me grounded when I'm floating away.
To Michelle for making me laugh when no one else can.
To Lola for reminding me of the light at the end of the tunnel.

"I really hate this damn machine,
I wish that they would sell it,
It never does quite what I want,
Just only what I tell it."

— The Programmer's Lament

Acknowledgments

I would like to thank...

My supervisor Dr. Michael Monagan. From him I have learned much more than can be expressed in one page. He has instilled in me a sense of pragmatic elegance for code writing that had formerly eluded me. For his patience, insight and guidance I am deeply grateful.

Roman Pearce who supported me during the implementation phases of my thesis. He has proved himself as a better resource than the internet and literature combined. If not for him I would surely still be coding.

The good people at Maplesoft (particularly Dr. Jürgen Gerhard). By no coincidence the entire C-library was written while I was an intern at the company. I was truly moved by their kindness and their help was invaluable to me.

My former teacher Dr. Jacques Carette. His support and interest in my career is somewhat inexplicable but immensely appreciated. Without him I would not be where I am today. By way of this acknowledgment I hope to finally start repaying him for all he has done.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Preliminaries

1.1 Introduction

Computer algebra emerged in the 1970's evolving out of the more established field of artificial intelligence. Today, computer algebra is not only regarded as entirely its own research area but also as a profitable enterprise. The commercialization of products like Maple and Mathematica have more than proved that there is a substantial interest in doing exact computations in academic and industrial settings. Unlike numerical programs like MatLab, computer-algebra systems can represent symbolic mathematical objects like polynomials, exact rational numbers, and algebraic numbers. This essential advantage allows symbolic systems to solve a range of problems that are inaccessible to those restricted solely to numerics.

How we represent symbolic objects, notably polynomials, is a crucial and often difficult question. There are many representations, each with unique advantages and weaknesses. We discuss one such representation in Section 4.1 that has become more practical due to the larger word size of modern processors. How we use this structure to do polynomial arithmetic is the central theme of this thesis. As computer-algebra systems spend most of their time doing arithmetic of this type, this is a worthwhile endeavor.

However, a quick look at the literature will show that there is an extensive amount of research on this topic. One may appropriately question the relevance of a thesis that could only be a survey of known results. Nevertheless, we intend to carry out the polynomial arithmetic in a novel way, by using lazy computation. Lazy computation refers to an environment where calculations are made only when absolutely necessary. We will see that

there are many benefits to computing in this manner. Chapter 2 discusses the design of such a library, and demonstrates the complexity of the resulting algorithms.

We also develop a storage-minimizing variant of this library that addresses a fundamental problems with Bareiss' fraction-free method for calculating determinants and the subresultant algorithm (the normal algorithms use too much space). Their implementations are given in Chapter 3.

1.2 Definitions

We now introduce the basic definitions and concepts required for developing algorithms involving polynomials.

Definition 1.2.1. A *monomial* in variables $x_1, x_2, \ldots, x_n$ is a product in the form

$$x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \ldots \cdot x_n^{\alpha_n},$$

where the exponents $\alpha_1, \ldots, \alpha_n$ are non-negatvie integers. The *total degree* of a monomial is the sum $\alpha_1 + \ldots + \alpha_n$.

It is convenient to let $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ be the n-tuples $(x_1, \ldots, x_n)$ and $(\alpha_1, \ldots, \alpha_n)$ respectively. We then set

$$\boldsymbol{x}^{\boldsymbol{\alpha}} = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \ldots \cdot x_n^{\alpha_n}$$

and let $|\boldsymbol{\alpha}| = \alpha_1 + \ldots + \alpha_n$ (the total degree). When referring to multiple monomials we will also use additional greek letters, i.e. $\boldsymbol{x}^{\boldsymbol{\beta}}$ and $\boldsymbol{x}^{\boldsymbol{\gamma}}$.

Definition 1.2.2. A *polynomial* f in variables $x_1, \ldots, x_n$ with coefficients in the ring k is a finite linear combination (with coefficients in k) of monomials in the form

$$f = \sum_{\boldsymbol{\alpha}} a_{\boldsymbol{\alpha}} \boldsymbol{x}^{\boldsymbol{\alpha}}$$

with $a_{\boldsymbol{\alpha}} \in k$. The set of all polynomials in the variables $x_1, \ldots, x_n$ with coefficients in the ring k is denoted $k[x_1, \ldots, x_n]$.

Our examples typically use polynomials, usually refered to as f, g, and h, with only a few variables. When this is the case it will be more convenient to dispense with subscripts and use the variables x, y, and z. For instance we let $f = x^2y^3z + xyz^2 - 7xy + 4$ instead of $g = x_1^2x_2^3x_3 + x_1x_2x_3^2 - 7x_1x_2 + 4$.

Definition 1.2.3. Let $f = \sum_{\alpha} a_{\alpha}\boldsymbol{x}^{\alpha}$ be a polynomial in $k[x_1, \ldots, x_n]$.

1. a_{α} is called the *coefficient* of the monomial $\boldsymbol{x}^{\alpha}$.
2. The product of a coefficient and a monomial, $a_{\alpha}\boldsymbol{x}^{\alpha}$, is called a *term* of f. When we refer to a term of f it is assumed that $a_{\alpha} \neq 0$. The total degree of a term is the degree of its associated monomial.
3. The *degree* of f, denoted $\mathtt{deg}(f)$ is the maximum total degree among all f's terms.

Example 1.2.4. For $f = 5x^2y^3z + xyz^2 - 7xy + 4$.

1. The expression $-7xy$ is a term of f, its coefficient is -7.
2. The degree of f is six ($\mathtt{deg}(f)$=6).

When f is unknown, we will refer to its non-zero terms as f_1, f_2, f_3 and so on, so that $f = f_1 + \ldots + f_n$. However, in order to do this, we must impose some sort of ordering on f's terms. Incidentally this is also an essential ingredient of most algorithms involving polynomials.

Definition 1.2.5. A *monomial ordering* on the monomials in $k[x_1, \ldots, x_n]$ is a relation $\geqslant_{ord}$ satisfying:

1. $\geqslant_{ord}$ is a total order on the monomials in $k[x_1, \ldots, x_n]$.
2. $\boldsymbol{x}^{\alpha} \geqslant_{ord} \boldsymbol{x}^{\beta} \Rightarrow \boldsymbol{x}^{\alpha}\boldsymbol{x}^{\gamma} \geqslant_{ord} \boldsymbol{x}^{\beta}\boldsymbol{x}^{\gamma}$ for monomials $\boldsymbol{x}^{\alpha}, \boldsymbol{x}^{\beta}$ and $\boldsymbol{x}^{\gamma}$.
3. $\geqslant_{ord}$ is a well ordering. That is, any non-empty subset consisting of monomials from $k[x_1, \ldots x_n]$ has a $\geqslant_{ord}$-least element.

We will extend this ordering to the *terms* of $k[x_1, \ldots, x_n]$ by saying $a_{\alpha}\boldsymbol{x}^{\alpha} \geqslant_{ord} a_{\beta}\boldsymbol{x}^{\beta} \Leftrightarrow \boldsymbol{x}^{\alpha} \geqslant_{ord} \boldsymbol{x}^{\beta}$.

Definition 1.2.6 (Lexicographic Order). We say $\boldsymbol{x}^{\alpha} \geqslant_{lex} \boldsymbol{x}^{\beta}$ when the first (from the left) non-zero entry of the vector difference $\alpha - \beta$ is positive.

Example 1.2.7. For monomials in the ring $\mathbb{Z}[x, y, z]$ we have:

1. $xy^2z \geqslant_{lex} y^3z^4$ since $(1, 2, 1) - (0, 3, 4) = (1, -1, -4)$
2. $x \geqslant_{lex} z^{10}$ since $(1, 0, 0) - (0, 0, 10) = (1, 0, -10)$

It should be noted that the lexicographic ordering is *not* invariant with respect to a permutation of the variables. For instance, if we instead use $\mathbb{Z}[z, y, x]$ in the previous example, we now have $xy^2z \leqslant_{lex} y^3z^4$ and $z^{10} \geqslant_{lex} x$. This is because there is an assumed ordering of the components of the exponent vector $\boldsymbol{\alpha}$. In the ring $\mathbb{Z}[x, y, z]$ it is implied that $x \geqslant_{lex} y \geqslant_{lex} z$, likewise for $\mathbb{Z}[z, y, x]$, $z \geqslant_{lex} y \geqslant_{lex} x$. In general, for any monomial ordering $\geqslant_{ord}$ on the polynomial ring $k[x_1, \ldots, x_n]$, it is always implicit that $x_1 >_{ord} \ldots >_{ord} x_n$.

Definition 1.2.8 (Graded Lexicographic Ordering). We say that $\boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{grlex} \boldsymbol{x}^{\boldsymbol{\beta}}$ when

$$|\boldsymbol{\alpha}| > |\boldsymbol{\beta}|, \text{ or } |\boldsymbol{\alpha}| = |\boldsymbol{\beta}| \text{ and } \boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{lex} \boldsymbol{x}^{\boldsymbol{\beta}}.$$

That is, we will first compare monomials using their total degree, and "break ties" using the lex order.

Example 1.2.9. For monomials in the ring $\mathbb{Z}[x, y, z]$

1. $xy^2z^3 \geqslant_{grlex} x^3y^2$ since $|(1, 2, 3)| = 6 > |(3, 2, 0)| = 5$
2. $xy^2z^4 \geqslant_{grlex} xyz^5$ since $|(1, 2, 4)| = |(1, 1, 5)|$ and $xy^2z^4 \geqslant_{lex} xyz^5$.

To illustrate the difference between lex and grlex ordering, we write terms of a polynomial, first in descending lexicographic ordering, then in descending graded lexicographic ordering, both with $x \geqslant_{lex} y \geqslant_{lex} z$:

$$f(x, y, z) = 4x^4y + x^2yz + x^2z^2 + 3y^4z + y^3z + y^3 + y^2z + yz^3 + yz^2 + 7z^6 \qquad \text{(lex)}$$
$$f(x, y, z) = 7z^6 + 4x^4y + 3y^4z + x^2yz + x^2z^2 + y^3z + yz^3 + y^3 + y^2z + yz^2 \qquad \text{(grlex)}$$

From now on, whenever possible, we will write polynomial terms in descending graded lexicographic order. For clarity, but also to emphasize that our proofs and algorithms work with *any* monomial ordering, we will use $\succ$ instead of $\geqslant_{ord}$.

1.3 Polynomial Arithmetic

Before discussing arithmetic on polynomials, we must first define how arithmetic works on monomials. We have already seen sums of terms when defining polynomials, but we now deal with the following special case:

Definition 1.3.1. For two terms $a\boldsymbol{x}^{\boldsymbol{\alpha}}$ and $b\boldsymbol{x}^{\boldsymbol{\beta}}$ in $k[x_1, \ldots, x_n]$, if $\boldsymbol{\alpha} = \boldsymbol{\beta}$ then

$$a\boldsymbol{x}^{\boldsymbol{\alpha}} + b\boldsymbol{x}^{\boldsymbol{\beta}} = (a +_k b)\boldsymbol{x}^{\boldsymbol{\alpha}}$$

where $+_k$ is the ring addition of k.

A new operation is the term product, given by

Definition 1.3.2. For two terms $a\boldsymbol{x}^{\boldsymbol{\alpha}}$ and $b\boldsymbol{x}^{\boldsymbol{\beta}}$ in $k[x_1, \ldots, x_n]$ the product $a\boldsymbol{x}^{\boldsymbol{\alpha}} \cdot b\boldsymbol{x}^{\boldsymbol{\beta}}$ be given by

$$a\boldsymbol{x}^{\boldsymbol{\alpha}} \cdot b\boldsymbol{x}^{\boldsymbol{\beta}} = (a \times_k b)\boldsymbol{x}^{\boldsymbol{\alpha}+\boldsymbol{\beta}}$$

where $\times_k$ is the ring multiplication from k.

Example 1.3.3. In $\mathbb{Z}[x, y]$, $2x^2y^3 + 3x^2y^3 = 5x^2y^3$ and $4x^4y^3z^6 \cdot 2x^2yz^3 = 8x^6y^4z^9$.

We can now easily define polynomial addition and multiplication using the notation from the previous section.

Definition 1.3.4. For the polynomials $f = f_1 + \cdots + f_n$ and $g = g_1 + \cdots + g_m$ in some $k[x_1, \ldots, x_k]$ the *polynomial sum*, $f + g$, is given by

$$f + g = f_1 + \cdots + f_n + g_1 + \cdots + g_m.$$

Definition 1.3.5. For any polynomial ring $k[x_1, \ldots, x_k]$ let the product of the term f_i and a polynomial $g = g_1 + \cdots + g_m$ be

$$f_i \cdot g = f_i \cdot g_1 + \cdots + f_i \cdot g_m.$$

Then we can define the product of the polynomial $f = f_1 + \cdots + f_n$ and g to be

$$f \cdot g = f_1 \cdot g + \cdots + f_n \cdot g.$$

However, from a computational perspective, these definitions are not complete. We would never write the polynomial sum of $f = x + y$ and $g = x + y$ as $f + g = x + y + x + y$. The true challenge comes from "collecting like terms".

Definition 1.3.6. Two terms $a\boldsymbol{x}^{\alpha}$ and $b\boldsymbol{x}^{\beta}$ in $k[\boldsymbol{x}]$ are called *like terms* if $\boldsymbol{\alpha} = \boldsymbol{\beta}$.

Example 1.3.7. The like terms of $f = 2x + 3x + 4x + 7y^2 + 3y^2 + z$ are $2x$, $3x$ and $4x$ as well as $7y^2$ and $3y^2$.

When we "collect the like terms" of f, we produce a *unique* polynomial $\tilde{f}$ such that $f = \tilde{f}$, where $\tilde{f}$ has no like terms. This is also called the *simplified form* of f.

Example 1.3.8. Collecting the like terms of f from Example 1.3.7 yields $f = 9x + 10y^2 + z$.

Definition 1.3.9. For some $f = f_1 + f_2 + \ldots + f_n \in k[x_1, \ldots, x_k]$ in simplified form and some corresponding monomial ordering $\succ$, the *leading term* of f, denoted $\mathrm{LT}(f)$, is the term f_i such that $f_i \succ f_j$ for all $j \neq i$. In other words $\mathrm{LT}(f)$ is the $\succ$-greatest element among f's terms.

If we let $a_{\alpha}\boldsymbol{x}^{\alpha} = \mathrm{LT}(f)$ then the *leading monomial* of f denoted $\mathrm{LM}(f)$ is $\boldsymbol{x}^{\alpha}$.

Example 1.3.10. Let $f = 8x^3y^2 - x^3z^3 + 2x^4 + z$ in $\mathbb{Z}[x, y, z]$. With lex ordering, $\mathrm{LT}(f) = 2x^4$, and with grlex ordering $\mathrm{LT}(f) = -x^3z^3$.

Definition 1.3.11. A polynomial $f = f_1 + \cdots + f_n$ is in *standard form* with respect to a monomial order $\succ$ if:

1. f_i and f_j are not like terms when $i \neq j$
2. $f_i \neq 0$ for $1 \leqslant i \leqslant n$
3. $f_i \succ f_{i+1}$ for $1 \leqslant i < n$.

We will denote the number of terms of a polynomial f in standard form by $\#f$.

From now on we will assume that the polynomials we are working with are in standard form. Our algorithms will be required to return polynomials in *standard form* as well. This *is* possible using the machinery developed in the previous section. Using monomial orders, we can systematically iterate over terms of a polynomial in a consistent manner. For example, consider Algorithm 1 for adding two polynomials.

In this case Algorithm 1 is a straightforward adaptation of Definition 1.3.4. To do the same with multiplication would not be as prudent. Recall Definition 1.3.5 for polynomial multiplication. To calculate $f \cdot g$ in this manner could require $O(mn^2)$ monomial comparisons.

Algorithm 1 Addition (A merge)

Input: $f = f_1 + \cdots + f_n$, $g = g_1 + \cdots + g_m$, where f and g are in standard form.
Output: $h = f + g = h_1 + \cdots + h_l$ such that h is in standard form.

```
1: (i, j, k) ← (1, 1, 1);
2: while i ≤ n or j ≤ m do
3:    if i ≤ n and j ≤ m and f_i and g_j are like terms then
4:       t ← f_i + f_j;
5:       if t ≠ 0 then
6:          h_k ← t;
7:          (i, j, k) ← (i + 1, j + 1, k + 1);
8:       else
9:          (i, j) ← (i + 1, j + 1);
10:      end if
11:   else if i ≤ n and f_i ≻ g_j then
12:      h_k ← f_i;
13:      (i, k) ← (i + 1, k + 1);
14:   else if j < m then
15:      h_k ← g_j;
16:      (j, k) ← (j + 1, k + 1);
17:   end if
18: end while
19: return h
```

Example 1.3.12. Let $f = x^n + x^{n-1} + \cdots + x$ and $g = y^m + y^{m-1} + \cdots + y$ which are in the standard form for monomial order $\geqslant_{lex}$. The product of f and g is given as

$$\begin{aligned} f \cdot g &= (x^n + \cdots + x)(y^m + \cdots + y) \\ &= (x^n y^m + \cdots + x^n y) + \cdots + (xy^m + \cdots + xy) \\ &= x^n \cdot g + x^{n-1} g + \cdots + x \cdot g. \end{aligned}$$

When merging, the sum $x^n \cdot g + x^{n-1} \cdot g$ requires m monomial comparisons and yields a polynomial that is $2m$ terms long (there will be no like terms in this sum, in fact there are no like terms in the entire product). This means that $(x^n \cdot g + x^{n-1} \cdot g) + x^{n-2} \cdot g$ requires $2m$ comparisons yielding a $3m$ long term. Continuing in this fashion means that we will do

$$m \sum_{i=1}^{n-1} i = m \binom{n(n-1)}{2}$$

or $O(mn^2)$ monomial comparisons.

In 1974 Johnson showed that we can do much better than $O(mn^2)$ monomial comparisons by merging the terms of $fg_1, \ldots, fg_m$ *simultaneously* using a heap. To elaborate, let $f = f_1 + \cdots + f_n$ and $g = g_1 + \cdots + g_m$ be in standard form with respect to $\succ$. Reducing $f \times g$ to standard form is equivalent to sorting the list

$$L = [f_1g_1, \ldots, f_ng_1, f_1g_2, \ldots, f_ng_2, \ldots, f_1g_m, \ldots, f_ng_m]$$

with respect to $\succ$ and then collecting like terms.

The sorting problem can be reduced to merging the set of *sorted* sequences

$$S = \{(f_1g_1, \ldots, f_ng_1), (f_1g_2, \ldots, f_ng_2), \ldots, (f_1g_m, \ldots, f_ng_m)\}$$

where each individual sequence, $S[j] = (f_1g_j, \ldots, f_ng_j)$, satisfies $f_1g_j \succ \ldots \succ f_ng_j$. We can refine our search for the largest term of L by exploiting this ordering and creating a list $H = [f_1g_1, \ldots, f_1g_m]$ consisting of the first elements of each sequence. Since f_1g_j is the largest term of $S[j]$, we are guaranteed that the $\succ$-largest term of H is the $\succ$-largest term of L. Merging is just the repeated application of this idea.

Namely, to merge the elements of S we do the following:

1. Create a list $H = [f_1g_1, \ldots, f_1g_m]$ and an empty sequence F.

2. Find the $\succ$-largest element of H, say f_ig_j (if there are multiple $\succ$-largest terms, pick the one with least i).

3. Add f_ig_j to the last term of F if they are like terms (this constitutes collecting like terms). Otherwise make f_ig_j the next term of F.

4. If $i < n$ then (in the list H) replace f_ig_j with $f_{i+1}g_j$ (the $\succ$-next largest term of $S[j]$). Otherwise, replace f_ig_j with 0.

5. Repeat steps 2 to 4 while there are non-zero terms of H.

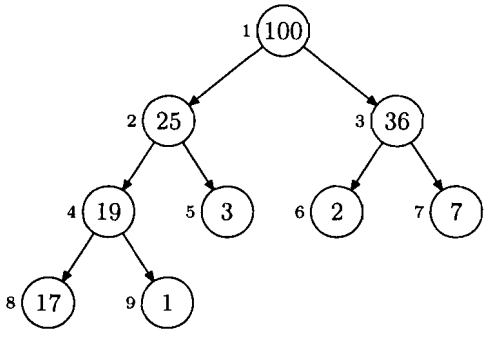
To find the largest element of H in a naive way requires $m - 1$ monomial comparisons; in general we will have to do this $O(nm)$ times, yielding $O(nm^2)$ monomial comparisons. Johnson's contribution was to make H a heap, instead of a list. A heap is a binary tree-based storage structure that yields constant-time access to its largest element, and has efficient extraction and insertion.

Definition 1.3.13. A *heap* is a partially ordered set $(\{H_1, H_2, \ldots, H_n\}, \geq)$ such that for all $1 \leq i \leq n/2$

$$H_i \geq H_{2i} \text{ and } H_i \geq H_{2i+1}.$$

A heap is typically implemented using an array, where H_i is stored at array position i, because arrays are compact and yield constant time access to terms. To remove H_1 (the $\geq$-largest element) requires $O(\log n)$ comparisons. Removing the largest element of the heap is called *extraction*. Inserting an element requires $O(\log n)$ comparisons [14].

Example 1.3.14. The array (or vector) $H = [100, 25, 36, 19, 3, 2, 7, 17, 1]$ is a heap that corresponds to the following binary tree.



Now, finding the largest term of H requires zero monomial comparisons, but inserting into and extracting from H is no longer free. In our example, the heap has m elements, and we insert/extract to/from it $O(nm)$ times at a cost of $O(\log m)$ monomial comparisons per insertion/extraction. This totals $O(nm \log m)$ monomial comparisons, which is better then the naive $O(nm^2)$.

For this reason we will be using Johnson's algorithm for multiplication (and division). However, in Chapter 2 it will be of great importance that our algorithms use as few terms of f and g as possible. Since step one of Johnson's method uses every term of g, we will provide an optimization that avoids this.

Claim 1.3.15. (Using the notation given in the explanation of merging.) If $f_1 g_j$ is in the heap H, then no term of the sequences $S[j+1], \ldots, S[m]$ can be the largest term of H.

Proof. By the definition of monomial ordering (property 2 of Definition 1.2.5) we have: if $g_j \succ g_{j+1} \succ \ldots \succ g_m$, then $f_1 g_j \succ f_1 g_{j+1} \succ \ldots \succ f_1 g_m$. As $f_1 g_{j+1}, \ldots, f_1 g_m$ are (respectively) the $\succ$-largest terms of $S[j+1], \ldots, S[m]$, it follows that $f_1 g_j$ is $\succ$-larger than

any term of $S[j+1], \ldots, S[m]$ (see Figure 1.1). The claim is an immediate consequence of this. □

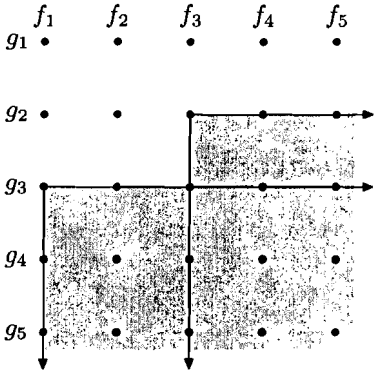


Figure 1.1: Points represent the terms of the product $f \cdot g$. The shaded region covers the terms that *must* be $\succ$-less than f_3g_2 or f_1g_3. The j-th row consists of the terms of $S[j]$. As we see, the term f_1g_3 shades a region which includes all points of $S[4], S[5], \ldots$ as posited in Claim 1.3.15.

Using this claim we can ensure that no unnecessary terms are put in the heap. That is, we will not begin inserting terms of the sequence $S[j+1]$ until the term f_1g_j has been extracted from the heap (Claim 1.3.15 ensures that these terms could not be the $\succ$-largest). In other words, we do not introduce new terms of g unless we have to. The same can be said of the terms of f, since $f_{i+1}g_j$ will only be inserted if f_ig_j is extracted.

By using this population strategy for the heap, we have addressed the requirement of using as few terms of f and g as possible (for *this* method). The replacement scheme is illustrated by Figure 1.2 and is used in the restatement of the merge process given below.

To merge elements of S with heaps, we do the following:

1. Create a heap $H = [f_1g_1]$ and an empty sequence F.
2. Extract the $\succ$-largest element of H, say f_ig_j.
3. Add f_ig_j to the last term of F if they are like terms (this constitutes collecting like terms). Otherwise, make f_ig_j the next term of F.
4. If $i < n$ then insert $f_{i+1}g_j$ (the $\succ$-next largest term of $S[j]$) into the heap.

5. If $i = 1$ and $j < m$ then add $f_1 g_{j+1}$ to the heap (that is, begin merging $S[j+1]$ with $S[1], \ldots, S[j]$).

6. Repeat steps 2 to 5 until the heap is empty.

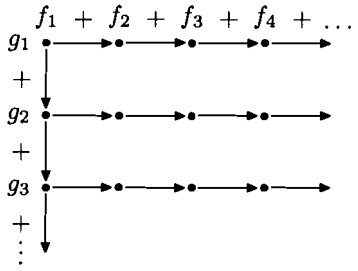


Figure 1.2: Points represent terms of the product $f \cdot g$, arrows indicate which terms are inserted when a term has been extracted from the heap. For example, after $f_1 g_1$ (the top left point) is removed, the terms $f_2 g_1$ and $f_1 g_2$ are added.

Finally, Algorithm 2 gives the implementation of polynomial multiplication using these ideas (the algorithm is our minor variant to Johnson's algorithm).

Theorem 1.3.16. *To find the standard form of $f \times g$, the heap multiplication algorithm will use $O(\#f \cdot \#g + \#g)$ space and do $O(\#f \cdot \#g \cdot \log \#g)$ monomial comparisons (the same as the unmodified Johnson's algorithms).*

Proof. The size of the heap is not affected by line (8), as this merely replaces the term coming out of the heap in line (6). The only place the heap can grow is in line (11), which is bounded by the number of terms of g. Therefore $O(\#g)$ space is required for the heap. Since the product $f \times g$ has $O(\#f \cdot \#g)$ many terms, the total amount of storage required will be $O(\#f \cdot \#g + \#g)$.

Extracting/inserting from/to a heap with $\#g$ elements does $O(\log \#g)$ monomial comparisons. Since every term of the product passes through the heap, we do $O(\#f\#g)$ extractions/insertions totaling $O(\#f\#g \log \#g)$ monomial comparisons[15]. □

Remark 1.3.17. It is possible to improve multiplication so that the number of monomial comparisons done are $O(\#f\#g \log \min(\#f, \#g))$ and amount of storage used is $O(\#f\#g + \min(\#f, \#g))$. If we were allowed to switch the order of the input (i.e. calculate $g \times f$ instead

Algorithm 2 Multiplication

Input: $f = f_1 + \cdots + f_n$, $g = g_1 + \cdots + g_m$, where f and g are in standard form.
Output: $h = f \times g = h_1 + \cdots + h_l$ such that h is in standard form.

```
1: Initialize a heap H and insert (f_1 g_1, 1, 1) {Order the heap by ≻ on the monomials in
   the first position.}
2: k ← 1;
3: while H is not empty do
4:    t ← 0;
5:    repeat
6:       Extract (s, i, j) ← H_max from the heap and assign t ← t + s;
7:       if i < n then
8:          Insert (f_{i+1} g_j, i + 1, j) into H;
9:       end if
10:      if i = 1 and j < m then
11:         Insert (f_1 g_{j+1}, 1, j + 1) into H;
12:      end if
13:   until (H is empty) or (t and H_max are not like terms).
14:   if t ≠ 0 then
15:      h_k ← t;
16:      k ← k + 1;
17:   end if
18: end while
19: return h
```

of $f \times g$) this could potentially make the heap much smaller. Recalling the replacement scheme from Figure 1.2 we can see the heap would be quite large if g had many terms. This would *not* be the case if it was f with many terms. Unfortunately, as we will see later, we will not be able to predict how many terms f or g have. So, we must quote the worst case scenario in our complexities (in fact this will be emphasized later by using $\max(\#f, \#g)$ in our complexities).

This heap approach for doing polynomial arithmetic can be extended to polynomial division as well. We assume the reader is familiar with polynomial division in one variable and extend this idea to polynomials of many variables.

Definition 1.3.18. The monomial $\boldsymbol{x}^{\boldsymbol{\alpha}} \in k[x_1, \ldots, x_n]$ is *divisible* by $\boldsymbol{x}^{\boldsymbol{\beta}}$, denoted $\boldsymbol{x}^{\boldsymbol{\beta}} | \boldsymbol{x}^{\boldsymbol{\alpha}}$, when the vector difference $\boldsymbol{\alpha} - \boldsymbol{\beta}$ contains no negative elements.

Example 1.3.19. In $\mathbb{Q}[x, y, z]$ we have that

1. $4x^4y^2$ is divisible by x^2y because $(4, 2, 0) - (2, 1, 0) = (2, 1, 0)$ has no negative entries.

2. zxy is not divisible by z^2 because $(1,1,1)-(2,0,0)=(-1,0,0)$ has a negative entry.

Definition 1.3.20. In $k[x_1,\ldots,x_n]$, when the monomial $\boldsymbol{x}^{\boldsymbol{\alpha}}$ is divisible by $\boldsymbol{x}^{\boldsymbol{\beta}}$, the division of the term $a\boldsymbol{x}^{\boldsymbol{\alpha}}$ by $b\boldsymbol{x}^{\boldsymbol{\beta}}$ with $b\neq 0$, denoted $ax^{\alpha}\div bx^{\beta}$, is given as

$$ax^{\alpha}\div bx^{\beta}=\frac{a}{b}\boldsymbol{x}^{\boldsymbol{\alpha}-\boldsymbol{\beta}}.$$

In this case it is necessary that b is an invertible element of k. To eliminate this requirement we will always assume that k is a field (denoted $\mathbb{F}$) when doing division.

Example 1.3.21. In $\mathbb{Q}[x,y,z]$ we have $9x^2y^3z^4\div 3xy^2z^2=3xyz^2$.

The goal in the one variable case, when dividing f by g in $\mathbb{F}[x]$, is to determine a *quotient* q and *remainder* r such that:

$$f=q\cdot g+r,$$

where $r=0$ or $\deg(r)<\deg(g)$. This is typically accomplished by the "long division" method, known to anyone who has completed an entry level calculus course. The procedure involves multiplying g by some appropriate term t, so that $f-t\cdot g$ "cancels out" the leading term of f. At every step f is set to $f-t\cdot g$ and the process is repeated until f is zero. In the cases where no such t exists, a term is added to the remainder r.

The process for doing polynomial division in many variables is the similar. In this case we take the leading term with respect to the monomial ordering $\succ$. We illustrate this with the following example.

Example 1.3.22. Working in $\mathbb{Q}[x,y,z]$ we divide $f=x^5z^2+x^4y+x^2y^2z+x^3z+x^2z^2+y^2$ by $g=x^2z+1$ using $\succ$ (the graded lexical order).

$$\begin{array}{rll}
 & q=x^3z+y^2+z & r=x^4y-z\\
x^2z+1\ \overline{)\,x^5z^2+x^4y+x^2y^2z+x^3z+x^2z^2+y^2} & & \\
\underline{x^5z^2+x^3z} & & \\
\underline{x^4y+x^2y^2z+x^2z^2+y^2} & & \to x^4y\\
x^2y^2z+x^2z^2+y^2 & & \\
\underline{x^2y^2z+y^2} & & \\
x^2z^2 & & \\
\underline{x^2z^2+z} & & \\
\underline{-z} & & \to -z\\
0 & &
\end{array}$$

In the first step we multiply $\mathrm{LT}(g) = x^2z$ by the monomial x^3z to eliminate $\mathrm{LT}(f) = x^5z^2$. We see that the result of $f - (x^2z) \cdot g$ has a leading term that is not divisible by $\mathrm{LT}(g)$ so x^4y is added to the remainder. The rest of the example carries on in this fashion.

To give polynomial division a stricter definition we have the theorem:

Theorem 1.3.23. *Let $\mathbb{F}$ be a field and let g be a non-zero polynomial in $\mathbb{F}[x_1, \ldots, x_k]$ with some monomial order $\succ$. Then every $f \in \mathbb{F}[x_1, \ldots, x_k]$ can be written as*

$$f = q \cdot g + r,$$

where $q, r \in \mathbb{F}[x_1, \ldots, x_n]$ and either $r = 0$ or no term of r is divisible by $LT(g)$. Furthermore, q and r are unique.

Proof of existence. The proof is constructive in that we present the division algorithm to show the existence of q and r.

Input: f, g in $\mathbb{F}[x_1, \ldots, x_k]$.
Output: q and r satisfying the conditions of Theorem 1.3.23.

1: $(q, r, p) \leftarrow (0, 0, f)$;
2: **while** $p \neq 0$ **do**
3: **if** $\mathrm{LT}(g) \,|\mathrm{LT}(p)$ **then**
4: $t \leftarrow \frac{\mathrm{LT}(p)}{\mathrm{LT}(g)}$
5: $(p, q) = (p - t \cdot g, q + t)$
6: **else**
7: $(p, r) = (p - \mathrm{LT}(p), r + \mathrm{LT}(p))$
8: **end if**
9: **end while**
10: **return** (q, r)

First let

$$\mathrm{LI}(n) = f - q \cdot g - (p + r) = 0,$$

which is the so-called loop invariant for iteration n. We demonstrate that this predicate holds throughout the entire execution of the algorithm and that $r = 0$ or that no term of r is divisible by $\mathrm{LT}(g) = g_1$.

For the base case at iteration zero we have $p = f$, $q = 0$ and $r = 0$ so

$$\mathrm{LI}(0) = f - q \cdot g - (p + r) = f - f = 0.$$

For the induction assume at loop $n-1$ that $\mathrm{LI}(n-1) = f - q \cdot g - (p+r) = 0$ and $r = 0$ or no term of r is divisible by g_1. Now, at loop n we have two cases given by the 'if' statement in line (3) of the division algorithm. In the case where $\mathrm{LT}(g) \,|\mathrm{LT}(p)$ we have:

$$t = \frac{\mathrm{LT}(p)}{\mathrm{LT}(g)} \qquad p = p - t \cdot g \qquad q = q + t$$

which means the loop invariant becomes

$$\begin{aligned} \mathrm{LI}(n) &= f - (q+t) \cdot g - (p - t \cdot g + r) \\ &= f - q \cdot g - (p+r) + (-t \cdot g + t \cdot g) \\ &= \mathrm{LI}(n-1) + 0 \\ &= 0 \end{aligned}$$

where $\mathrm{LI}(n-1) = 0$ by the induction hypothesis. Since r is unchanged from iteration $n-1$, in this case, it maintains the desired property.

In the case where $\mathrm{LT}(g)$ does not divide $\mathrm{LT}(p)$ we have:

$$p = p - \mathrm{LT}(p) \qquad r = r + \mathrm{LT}(p)$$

so the loop invariant is now

$$\begin{aligned} \mathrm{LI}(n) &= f - q \cdot g - (p - \mathrm{LT}(p) + r + \mathrm{LT}(p)) \\ &= f - q \cdot g - (p+r) \\ &= \mathrm{LI}(n-1) \\ &= 0 \end{aligned}$$

via the induction hypothesis.

Let us rewrite $r = r + \mathrm{LT}(p)$ to be $r_n = r_{n-1} + \mathrm{LT}(p)$ so that r_{n-1} is the remainder from iteration $n-1$. It is the case that the term $\mathrm{LT}(p)$ is not divisible by g_1 and by the induction hypothesis r_{n-1} has no term divisible by g_1 either. Collectively this means that $r_n = r_{n-1} + \mathrm{LT}(p)$ has no term that is divisible by g_1 as desired. We have thus shown that the loop invariant is held at every iteration with an r that has no term divisible by g_1. To complete the proof, we show that the algorithm terminates. The value of p is updated on lines (5) and (7), and in both cases the leading term of p is eliminated. Since p is strictly greater then $p - \mathrm{LT}(p)$ with respect to the monomial ordering (otherwise $\mathrm{LT}(p) =$

$\mathrm{LT}(p - \mathrm{LT}(p))$, a contradiction), the values of p yield a strictly descending sequence on $\succ$. Since $\succ$ is a well ordering, this must be a finite sequence with a least element ([9, Section 2.2 Lemma 2]). Therefore, we may only eliminate the leading term of p finitely many times, until $p = 0$. Now suppose that the division algorithm terminates at iteration M (when the terminating condition $p = 0$ is met). This gives

$$\begin{aligned} &\Rightarrow \mathrm{LI}(M) = f - q \cdot g - (0 + r) \\ &\Rightarrow 0 = f - q \cdot g - r \\ &\Rightarrow f = q \cdot g + r \end{aligned}$$

where no term of r is divisible by g_1 as Theorem 1.3.23 requires. □

Proof of uniqueness. Suppose that we have

$$f = q \cdot g + r = q' \cdot g + r'$$

where $q \neq q'$ and $r \neq r'$. Neither r or r' has a term divisible by g_1 but $(q - q') \cdot g$ does. Since $r - r' = (q - q') \cdot g$, $r - r'$ must have a term that is divisible by g_1. This is only possible when $r - r' = 0$ which contradicts our assumption. Therefore q and r must be unique. □

It would be tempting to simply use the algorithm in Theorem 1.3.23 to do polynomial division. But, we can show that this classical implementation suffers from many weaknesses.

Consider line (5) in the algorithm given in Theorem 1.3.23 where we update p by $p - t \cdot g$. Recall that our addition algorithm works as a merge, repeatedly comparing monomials until the terms of one input polynomial are exhausted. This becomes problematic when one of the polynomials is much larger than the other as seen in Example 1.3.24.

Example 1.3.24. Let $f = x^n + 1$, $g = y^n + y^{n-1} + \cdots + y$ and $h = f \cdot g = (x^n y^n + x^n y^{n-1} + \cdots + x^n y) + (y^n + y^{n-1} + \cdots + y)$. Consider the division of h by f:

$$\begin{array}{rl} q = y^n + y^{n-1} + \ldots & \\ x^n + 1 \;\overline{)x^n y^n + x^n y^{n-1} + \ldots + x^n y + y^n + \ldots + y} & \\ \underline{x^n y^n + y^n} & \rightarrow \text{does } n \text{ monomial comparisons} \\ x^n y^{n-1} + \ldots + x^n y + \ldots + y & \\ \underline{x^n y^{n-1} + y^{n-1}} & \rightarrow \text{does } n - 1 \text{ monomial comparisons} \\ \vdots & \end{array}$$

As we can see, to merge $x^i y^i + y^i$ for some i will require i monomial comparisons. In total the number of comparisons will be

$$\sum_{i=1}^{n} i = \frac{n^2}{2} + \frac{n}{2},$$

which is an order of magnitude larger then the product $(x^n + 1) \cdot q$ and requires a large amount of auxiliary space.

As with multiplication, we can use heaps and a replacement scheme to overcome the problems in Example 1.3.24 and drastically reduce the amount of monomial comparisons. The heap itself will be used to store the intermediate p's and our replacement scheme will allow us to minimize this heap's size.

The key observation is that we only require the leading term of p to calculate the next term of the quotient. Since we are ordering the heap with $\succ$, the leading term will always be the $\succ$-largest element (or, in the case where this element has like terms in the heap, the sum of the top few elements of the heap). All we need is a replacement scheme that guarantees that the heap will produce the terms of $f - q \cdot g$ in $\succ$-descending order.

To be more clear, we will be merging the set of sequences

$$\{(f_1, \ldots, f_n), (-q_1 g_1, \ldots, -q_k g_1), \ldots, (-q_1 g_m, \ldots, -q_k g_m)\}.$$

An alternate way to understand this merge is to see the heap as storing the difference

$$f - \sum_{i=1}^{m} (q_1 + q_2 + \ldots + q_k) \times g_i$$

where g and q are of lengths m and k respectively and the terms q_i may be unknown.

As with multiplication, we replace a term coming out of the heap with the $\succ$-next largest term in the sequence it was taken from. That is, we replace f_i with f_{i+1} and $-q_i g_j$ with $-q_{i+1} g_j$ (we also use the optimization that says only add $-q_1 g_{j+1}$ after removing $-q_1 g_j$). However, it is possible that we remove $-q_{i-1} g_j$ before q_i is known, in which case we would not be able to insert the term $-q_i g_j$. But, since $-q_i g_j$ can certainly not be required to calculate q_i, there must already be a sufficient amount of terms in the heap to establish q_i. Therefore, we can just remember the terms that should have been added to the heap, and eventually add them once q_i has been calculated. In the heap-division algorithm, this is referred to as 'sleeping'.

The heap division algorithm [15] is given by Algorithm 3.

Algorithm 3 Heap-Division

Input: f, $g \neq 0$ in $\mathbb{F}[x_1, \ldots, x_n]$.
Output: q and r, in standard form, satisfying the conditions of Theorem 1.3.23.

1: **if** $f = 0$ **then**
2: **return** $(0, f)$
3: **end if**
4: Initialize a new heap H and insert f_1 into H;
5: $(q, r, s) \leftarrow (0, 0, 2)$;
6: **while** H is not empty **do**
7: $t \leftarrow 0$;
8: **repeat**
9: Extract $x \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + x$;
10: **if** $x = f_i$ and f_{i+1} exists **then**
11: Insert f_{i+1} into H;
12: **else if** $x = g_i q_j$ and q_{j+1} is calculated **then**
13: Insert $-g_i q_{j+1}$ into H;
14: **else if** $x = g_i q_j$ and q_{j+1} is not yet calculated **then**
15: s $\leftarrow$ s + 1; {Sleep $-g_i q_{j+1}$}
16: **end if**
17: **if** $x = g_i q_1$ and g_{i+1} exists **then**
18: Insert $-g_{i+1} q_1$ into H;
19: **end if**
20: **until** (H is empty) or (t and H_{max} are not like terms).
21: **if** $t \neq 0$ and $g_1 | t$ **then**
22: Make t/g_1 the next term of q;
23: **for** k from 2 to s **do**
24: Insert $-g_k \cdot t/g_1$ into H; {Insert all terms that are sleeping into H}
25: **end for**
26: **else if** $t \neq 0$ **then**
27: Make t the next term of r;
28: **end if**
29: **end while**
30: **return** (q, r)

Theorem 1.3.25. *The heap-division algorithm requires* $O(\#g + \#q + \#r + 1)$ *storage for terms and does* $O((\#f + \#q\#g) \cdot \log \#g)$ *monomial comparisons.*

Proof. The size of the heap H, denoted $|H|$ is unaffected by lines (11) and (13) since these lines only replace terms coming out of the heap. Line (15) merely increments s and does not increase $|H|$. The only place where H can grow is line (18) in which a new term of g is added to the heap, this is clearly bounded by $\#g$. Accounting for the storage needed for the solution, we need $O(\#g)$ storage for the heap and $O(\#q + \#r)$ for the quotient and remainder, totaling $O(\#g + \#q + \#r + 1)$.

All terms of f and $q \cdot g$ are added to the heap, which is $\#f + \#q\#g$ terms. Passing this many terms through a heap of size $\#g$ requires $O((\#f + \#q\#g) \log \#g)$ monomial comparisons [14]. □

In this section we have defined polynomials, monomial orderings and explored how to use these definitions in algorithms to do polynomial arithmetic. In the next section we will derive a variant of these algorithms that will allow us to do *delayed* polynomial arithmetic.

Chapter 2

Lazy and Forgetful Polynomials

The term 'lazy evaluation' (or 'delayed computation') refers to the technique of postponing a computation until its value is known to be needed. This notion is the defining feature of most functional programming languages like Haskell or MosML. In these languages, the use of lazy evaluation allows for the creation of infinite lists, or 'streams' [16]. To overcome the problem of needing an infinite amount of memory to store an infinite list, a stream mimics an infinite list by storing a head element and a subsequent rule to get you from one element to the next, called a tail function (much like the base case and inductive step in an inductive proof). Any element of the stream can be accessed, but it is only calculated when its value is needed.

We can adapt this idea to do polynomial operations [19]. If we order the terms of the polynomial with some monomial ordering $\succ$, then we can number the terms from 1 to n, where 1 is the $\succ$-largest term. If we then restrict our algorithms to only return a single term from a polynomial (for example if $f = x^2y + xy^2 + 7$ then $f_3 = 7$) then we can also be lazy in our calculations. To calculate the n-th term from the sum of two polynomials would *not* require the calculation of the $(n+1)$-st term. We could simply halt after enough work has been done. Furthermore, if we saved intermediate results from this calculation, then the m-th term where $m \leq n$ could be 'calculated' instantaneously.

In this section we present the variations of the algorithms given in Chapter 1 that will allow us to compute in a lazy manner.

2.1 Motivation

Our motivation comes from an intermediate calculation in Buchberger's algorithm [9]. Buchberger's algorithm transforms a set of generators for a polynomial ideal into a Gröbner basis [9]. At each iteration the algorithm generates many 'S-Polynomials', namely the value

$$S(f,g) = \frac{L}{\mathrm{LT}(f)} \cdot f - \frac{L}{\mathrm{LT}(g)} \cdot g$$

where $L = \mathtt{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$. This calculation appears to be wasteful as it is known that $S(f,g)$ reduces to zero when $\mathtt{gcd}(\mathrm{LT}(f,), \mathrm{LT}(g)) = 1$. Why completely determine f and g in this case when only the leading terms are needed?

An environment where one could determine $\mathrm{LT}(f)$ without calculating f in its entirety could theoretically speed up Buchberger's algorithm significantly. However, although this may be the case, this undertaking is perhaps too ambitious to be carried out in this thesis. Thankfully, investigations exposed simpler algorithms that could benefit from delayed arithmetic in the same way. We discuss these applications in Sections 3.1 and 3.2.

2.2 Lazy Polynomials

We begin development of an environment for performing lazy arithmetic with polynomials by first discussing what this entails. Our primary objective will be to compute the n-th term of some arbitrary polynomial expression, say $f \times g$, using as few terms of f and g as possible. Our notion of 'work' or 'amount of computation' will then be a measure of how many monomial comparisons are done to determine this n-th term as well as the work necessary to calculate the individual terms of f and g that are used. Our goal will be to create algorithms that do as little work as possible (one may characterize this as laziness).

Unfortunately we are not able to measure the number of monomial comparisons required to calculate *only* the n-th term of $f \times g$ in general. We can only give an upper bound on the number of monomial comparisons required to calculate *every* term and take for granted that it takes less monomial comparisons to calculate fewer than all terms (which, as we saw in the motivation is to our advantage).

This restriction also prevents us from calculating the work done to produce individual terms of f and g as well. We can merely assume that *some* work will be done in order to calculate a given term of f and g and strive to devise algorithms that use as few of these terms as possible.

In Chapter 1, we have developed heap methods for polynomial arithmetic and established the amount of monomial comparisons that each method requires. Although we can't prove that this is the minimal amount of comparisons possible (it is still open whether multiplication is $O(\#f \cdot \#g \log \min(\#f, \#g))$ we demonstrated that these algorithms are performing substantially better than classical methods. We also devised a replacement scheme that ensured the heap contained only those terms essential for the calculation of the next term. This property will be crucial now, as it ensures that no work is done calculating unneeded terms of f and g.

In this sense, we can claim that our algorithms are doing a minimal amount of work in general. We use algorithms that avoid doing any *unnecessary* work (like calculating unneeded terms of f and g) and use *few* (but not provably minimal) monomial comparisons by way of our heap algorithms. By this premise, we can at least superficially conclude that we are doing a minimal amount of work to calculate a single term.

We will now begin the development of the lazy algorithms that utilize these ideas. These algorithms will be a variation of the heap algorithms given in Chapter 1, which were deliberately designed in a manner to be conducive to these lazy ambitions.

Definition 2.2.1. A *lazy* or *delayed polynomial*, F, is an approximation of the polynomial $f = f_1 + \cdots + f_n$ (in standard form), given by

$$F^N = \sum_{i=1}^{N} f_i$$

where $0 \leq N \leq n$.

The terms $F_1, \ldots, F_N$ are called the *forced terms* of F and the nonzero terms of $f - F^N$ are called the *delayed terms* of F. We denote the number of forced terms of a delayed polynomial F by $|F|$. Note that it is always the case that $F_i = f_i$ for all i and that F^{N+1} is a better approximation to f than F^N when $N < n$.

A delayed polynomial must satisfy the following conditions regarding computation. If F is approximating a polynomial expression of the type $g + h$, $g \times h$, or $g \div h$ (either the quotient or remainder of the division) then:

1. Calculating F_i when F_i is a forced term requires no monomial comparisons. That is: all the forced terms of F should be cached for re-access.

2. The scheme to calculate a delayed term of F uses as few terms of g and h as possible.

3. The number of monomial comparisons required to calculate every term of F is no more then the number of monomial comparisons done by the heap algorithms given in Chapter 1.

From now on it will be necessary to distinguish regular polynomials from those that are delayed. When a polynomial is delayed we will denote it with capitals letters, typically F, G or H and continue using lower case letters when they are not.

Remark 2.2.2. The polynomial $f = f_1 + \cdots + f_n$ that we are approximating is unknown. Since this also means that n is unknown we are unable to say if F^N exists. For instance, if $f = x + y$ then F^3 would not have a value by Definition 2.2.1. To resolve this we append an infinite amount of zeros to the end of f so that $f = f_1 + \cdots + f_n + 0 + 0 + \cdots$ (now $F^3 = x + y + 0$). For clarity we will continue omitting zero terms when describing a polynomial.

This admits the useful notation F^∞ which is the lazy polynomial with no delayed terms. That is $F^\infty = f$ when F is approximating f.

Example 2.2.3. Let $f = x^3y^2 + xy + y + 1$ and let F approximate f.

$$F^2 = x^3y^2 + xy \qquad F_3 = y \qquad F_{100} = 0$$

Let us refine our focus and address the problem of determining the n-th term of a polynomial when it is the result of some operation. That is, given delayed polynomials F and G that approximate f and g respectively, can we determine the n-th term of $f + g$ or $f \times g$?

The new difficulties we will encounter are caused by the mysterious nature of the terms of F and G. Since these delayed polynomials are merely approximations of the polynomials we are operating on, we can rarely speculate about the qualities of these terms (like how many terms there are). What we can assume, since $F_i = f_i$ (where f is in standard form for some monomial ordering $\succ$) is that $F_i \succ F_{i+1}$, which all of our algorithms will exploit. Furthermore, since accessing F_i will presumably do some calculation, we would like to access as few terms of F (and G) as possible. [1]

First, let us dispel of the notion that we could use naive algorithms for this purpose. These algorithms are ill suited to our goal, as they prematurely access terms. To best

[1] In fact this is the most important feature of our algorithms. The entire environment is set up so that we can force as few terms of a delayed polynomial as possible.

illustrate this, recall the first step of naive multiplication, where we merge $f_1 \cdot g$ with $f_2 \cdot g$. To accomplish this, it would be necessary to use *every* term of g. Division is worse. When updating f by $f - t \cdot g$, all terms of both f and g would have to be used.

Remark 2.2.4. It is usually not a problem to use every term of a polynomial in a calculation (the terms are traditionally known). In fact Johnson's [12] algorithms typically begin by populating the heap with all the terms of some polynomial. The slight modifications to these algorithms given in Chapter 1 were done to overcome this problem.

We will find it is simple to modify the algorithms of Chapter 1 to calculate a specific term of a polynomial. To demonstrate this we modify heap multiplication (Algorithm 2) to return the n-th term of a product. The modification is given by Algorithm 4. We will see the modifications of division and addition later, as part of the complete delayed algorithms, so we will not present them.

The first of the changes are made to the conditions on lines (7) and (10) of Algorithm 4. Since we do not know the length of f (and g) we are unable to determine if some term f_i exists. Instead we can check if f_i is zero, in which case the previous term was the last non-zero term. This strategy will be repeated in all our algorithms and should be interpreted as checking if f_i exists. The next change is a reflection of our goal, line (17) returns the n-th term once it is calculated. This is easy to determine as the algorithm is incrementing through the terms of $f \times g$ in $\succ$-order. Of course, if the algorithm terminates before $k = N$, this means that the n-th term does not exist. So, to be consistent with our definitions, we return zero (line (21)).

While being able to calculate the N-th term of a polynomial is a good start we would also expect no work be repeated to calculate $F_{N-1}, \ldots, F_1$ after calculating F_N. We can make this change by passing our algorithms any pertinent partial solution (i.e the approximation F^N). By doing this our algorithms can be thought of more as a method for picking up where the last calculation left off. However, in order to accomplish this the partial solutions have to remember the state of the algorithm they were passed to. They must remember the heap the calculation was using and local variables that would otherwise get erased. For now, assume that this information is associated with the partial solution in *some* way, which we will resolve in Chapter 4.

Now we can modify Algorithm 4 so it does not waste computation if more terms are needed from the same polynomial. Recall that $|F|$ is the number of forced terms of the

Algorithm 4 Calculate N-th term of a product

Input: The delayed polynomials F and G so that $F^{\infty} = f$ and $G^{\infty} = g$ and a positive integer N.

Output: The N-th term of the product $f \times g$.

```
1: Initialize a heap H and insert (F_1 G_1, 1, 1) {Order the heap by ≻ on the monomials in
   the first position.}
2: k ← 1;
3: while H is not empty do
4:   t ← 0;
5:   repeat
6:     Extract (s, i, j) ← H_max from the heap and assign t ← t + s;
7:     if F_{i+1} ≠ 0 then
8:       Insert (F_{i+1} G_j, i + 1, j) into H;
9:     end if
10:    if i = 1 and G_{j+1} ≠ 0 then
11:      Insert (F_1 G_{j+1}, 1, j + 1) into H;
12:    end if
13:  until (H is empty) or (t and H_max are not like terms).
14:  if t ≠ 0 then
15:    k ← k + 1;
16:  end if
17:  if k = N then
18:    return t;
19:  end if
20: end while
21: return 0;
```

delayed polynomial F. The implementation is given by Algorithm 5.

It is a simple task to extend this idea to addition, which is given by Algorithm 6. In this case the algorithm does not require a heap since it is only doing a simple merge. Instead what must be remembered by the solution X is how far along F and G the merge is. Or specifically, the indices i and j of the next terms from F and G to be merged.

Lastly we modify division (Algorithm 7). The division algorithm returns two polynomials, the quotient and the remainder, so there can be some ambiguity about which quantity the n-th term is referring to. In our implementation we return the n-th term of the quotient, but with a trivial change it could easily be the n-th term of the remainder instead. A key feature of the algorithm is that it updates *both* Q and R. That is to say that no work would be repeated to calculate terms of the remainder if they were forced while calculating terms

of the quotient.

Remark 2.2.5. This feature also yields an algorithm for divisibility testing. Suppose Q and R are the quotient and remainder which result when we pass F and G to Algorithm 7; then

$$F|G \Leftrightarrow R_1 = 0.$$

Testing in this manner will require less computation than if the entire remainder was to be computed (provided that the remainder has more than one term).

2.3 Forgetful Polynomials

There is a variant of delayed polynomial arithmetic that has some useful properties. Consider that the operations from the previous section can be composed to form polynomial expressions. That is, we could use delayed arithmetic to calculate the n-th term of say, $A \cdot B - C \cdot D$. When we did this we stored the intermediate calculations (namely the products $A \cdot B$ and $C \cdot D$) to provide quick re-access to terms. But, if re-access was not required we could have easily "forgotten" these terms instead. A "forgetful" operation will be like a delayed operation but intermediate terms won't be stored. For this reason, forgetful operations are quite useful when expanding composed polynomial expressions with large intermediate subexpressions. This is because we are ensuring we use the minimal amount of memory (much in the same spirit as doing minimal computation) which has clear advantages.

We can make some straightforward modifications to our delayed algorithms to accomplish this forgetful environment. Essentially all that is required is the removal of lines that save terms to the solution polynomial (i.e. lines that look like $X_i \leftarrow \square$) and eliminating any references to previous terms (or even multiple references to a current term). To emphasize this change we will limit our access to a polynomial by way of a `next` command.

Definition 2.3.1. For some delayed polynomial F and monomial order $\succ$, the `next` command returns the $\succ$-next un-calculated term of a polynomial (eventually returning only zeros) and satisfies

$$\texttt{next}\,(F) + \texttt{next}\,(F) + \texttt{next}\,(F) + \cdots = F^{\infty}$$

and

$$\texttt{next}\,(F) \succ \texttt{next}\,(F) \succ \cdots \succeq 0 \succeq 0 \cdots .$$

Definition 2.3.2. A *forgetful polynomial* is a delayed polynomial that is accessed solely via the `next` command. That is, intermediate terms of F are not stored and can only be accessed *once*. If the functionality to re-access terms is restored in any way (i.e. by caching the intermediate results in memory), F is no longer considered a forgetful polynomial. Thus, for a forgetful polynomial F, calculating F_{n+1} forfeits access to the terms F_1 through F_n, even if these terms have never been accessed.

Although it would be ideal to have all of our forgetful routines take forgetful polynomials as input and return forgetful polynomials as output, we will see this is not possible without caching previous results. Multiplication for instance can not accept forgetful polynomials as input (it *will* be able to return a forgetful polynomial). This is because regardless of the scheme used to calculate $f \cdot g$, it is necessary to multiply every term of f with g. Since we are limited to single time access to terms this task is impossible. If we calculate $f_1 g_2$ we can not calculate $f_2 g_1$ and vice versa.

The problems of multiplication percolate to other algorithms that have multiplication as an intermediate step. Specifically, our division algorithm can not accept a forgetful dividend as it must be repeatedly multiplied by terms of the quotient (for the same reason the quotient can not be forgetful). However, we will see that the dividend *can* be forgetful which is a highly desirable feature (see Chapter 3).

The only "fully" forgetful (forgetful input and output) routine we can have is addition, given by Algorithm 8. The variant of multiplication that takes as input delayed polynomials, returning a forgetful polynomial, is a trivial change to Algorithm 5. In this case all that must be done is to remove the 'if' statement in line (28) so that the $\succ$-next, instead of the N-th, term is returned. As this is not a significant change, we will not give pseudocode for multiplication.

Lastly, division will be given as a special purpose algorithm that will be useful in some specific applications. Division will take as input a forgetful dividend and delayed divisor returning a fully forced quotient and remainder. Aside from enabling division (given by Algorithm 9) to accept a forgetful dividend, there are no other improvements.

Theorem 2.3.3. *When adding F and G with the forgetful addition, the worst case storage complexity for the algorithm is $O(1)$.*

Proof. At any given time the Algorithm 8 will only have to remember ans, t_F and t_G. □

Theorem 2.3.4. *When multiplying f (with $\#f$ terms) by g (with $\#g$ terms) the worst case storage complexity for forgetful multiplication is $O(max(\#f, \#g))$ (the storage required for the heap).*

Proof. For delayed multiplication (Algorithm 5) the size of the heap H, denoted $|H|$, is unaffected by line (18) since this merely replaces the term coming out of the heap on line (16). The only place the heap can grow is on line (21), but this is bounded by the number of non-zero terms of G^∞. Since $G^\infty = f$ or $G^\infty = g$ (depending on the order of the input), G^∞ can have at most $\max(\#f, \#g)$ non-zero terms. The solution will require at most $O(\#f \cdot \#g)$ space, totaling $O(\#f \cdot \#g + \max(\#f, \#g))$ storage for delayed multiplication.

A quick inspection of Algorithm 5 will show that the only time a previous term of the product is used is on line (3) and (30). In both cases the term is merely being *re-accessed* and is not used to compute a new term of the product. Since we do not store or re-access terms of a forgetful polynomial, we can eliminate the storage needed to do this, and reduce the space complexity to $O(\max(\#f, \#g))$ as desired. □

Algorithm 5 Delayed Multiplication

Input: The delayed polynomials F and G so that $F^{\infty} = f$ and $G^{\infty} = g$, a positive integer N (the desired term), and the delayed polynomial X so that $X^{\infty} = f \times g$.

Output: The N-th term of the product $f \times g$.

1: **if** $N \leq |X|$ **then**
2: $\{X_N$ has already been calculated.$\}$
3: **return** X_N;
4: **end if**
5: **if** |X|=0 **then**
6: $\{X$ has no information.$\}$
7: Initialize a heap H and insert $(F_1G_1, 1, 1)$ {Order the heap by $\succ$ on the monomials in the first position.}
8: $k \leftarrow 1$;
9: **else**
10: Let H be the heap associated with X.
11: $k \leftarrow$ number of elements in H;
12: **end if**
13: **while** H is not empty **do**
14: $t \leftarrow 0$;
15: **repeat**
16: Extract $(s, i, j) \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + s$;
17: **if** $F_{i+1} \neq 0$ **then**
18: Insert $(F_{i+1}G_j, i + 1, j)$ into H;
19: **end if**
20: **if** $i = 1$ and $G_{j+1} \neq 0$ **then**
21: Insert $(F_1G_{j+1}, 1, j + 1)$ into H;
22: **end if**
23: **until** (H is empty) or (t and H_{max} are not like terms)
24: **if** $t \neq 0$ **then**
25: $X_k \leftarrow t$;
26: $k \leftarrow k + 1$;
27: **end if**
28: **if** $k = N$ **then**
29: Associate the heap H with X.
30: **return** X_k;
31: **end if**
32: **end while**
33: Associate the (empty) heap H with X.
34: **return** 0;

Algorithm 6 Delayed Addition

Input: The delayed polynomials F and G so that $F^{\infty} = f$ and $G^{\infty} = g$, a positive integer N (the desired term), and the delayed polynomial X so that $X^{\infty} = f + g$.
Output: The N-th term of the sum $f + g$.

1: **if** $N \leq |X|$ **then**
2: $\{X_N$ has already been calculated.$\}$
3: **return** X_N;
4: **end if**
5: **if** |X|=0 **then**
6: $\{X$ has no information.$\}$
7: $(i, j, k) \leftarrow (1, 1, 1)$;
8: **else**
9: Set i and j to the values associated with X;
10: $k \leftarrow |X|$;
11: **end if**
12: **while** $F_i \neq 0$ or $G_j \neq 0$ **do**
13: **if** F_i and G_j are like terms **then**
14: $t \leftarrow F_i + G_j$;
15: **if** $t \neq 0$ **then**
16: $X_k \leftarrow t$;
17: $(i, j, k) \leftarrow (i + 1, j + 1, k + 1)$;
18: **else**
19: $(i, j) \leftarrow (i + 1, j + 1)$;
20: **end if**
21: **else if** $F_i \neq 0$ and $F_i \succ G_j$ **then**
22: $X_k \leftarrow F_i$;
23: $(i, k) \leftarrow (i + 1, k + 1)$
24: **else if** $G_j \neq 0$ **then**
25: $X_k \leftarrow G_j$
26: $(j, k) \leftarrow (j + 1, k + 1)$
27: **end if**
28: **if** $k = N$ **then**
29: Associate i and j with X;
30: **return** X_k;
31: **end if**
32: **end while**
33: Associate i and j with X;
34: **return** 0;

Algorithm 7 Delayed Heap-Division

Input: The delayed polynomials F and G so that $F^\infty = f$ and $G^\infty = g$, a positive integer N (the desired term), and the delayed polynomials Q and R so that $f = g \cdot Q^\infty + R^\infty$.
Output: The N-th term of the quotient from $f \div g$.

1: **if** $F_1 = 0$ **then return** 0; **end if**
2: **if** $N \leq |Q|$ **then**
3: $\{Q_N$ has already been calculated.$\}$
4: **return** Q_N;
5: **end if**
6: **if** $|Q| = 0$ **then**
7: $\{Q$ has no information.$\}$
8: Initialize a new heap H and insert F_1 into H;
9: $s \leftarrow 2$;
10: **else**
11: Let H be the heap associated with Q;
12: **end if**
13: **while** H is not empty **do**
14: $t \leftarrow 0$;
15: **repeat**
16: Extract $x \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + x$;
17: **if** $x = F_i$ and $F_{i+1} \neq 0$ **then**
18: Insert F_{i+1} into H;
19: **else if** $x = G_i Q_j$ and Q_{j+1} is forced **then**
20: Insert $-G_i Q_{j+1}$ into H;
21: **else if** $x = G_i Q_j$ and Q_{j+1} is delayed **then**
22: s $\leftarrow$ s + 1; {Sleep $-G_i Q_{j+1}$}
23: **end if**
24: **if** $x = G_i Q_1$ and $G_{i+1} \neq 0$ **then**
25: Insert $-G_{i+1} Q_1$ into H;
26: **end if**
27: **until** (H is empty) or (t and H_{max} are not like terms).
28: **if** $t \neq 0$ and $g_1 | t$ **then**
29: $Q_{|Q|+1} \leftarrow t/G_1$; {Now $Q_{|Q|+1}$ is a forced term.}
30: **for** k from 2 to s **do**
31: Insert $-G_k \cdot t/G_1$ into H; {Insert all terms that are sleeping into H}
32: **end for**
33: **else if** $t \neq 0$ **then**
34: $R_{|R|+1} \leftarrow t$; {Now $R_{|R|+1}$ is a forced term.}
35: **end if**
36: **if** $|Q| = N$ **then** Associate the heap H with Q; **return** Q_N; **end if**
37: **end while**
38: Associate the (empty) heap H with Q;
39: **return** 0;

Algorithm 8 Forgetful Addition

Input: The forgetful polynomials F and G so that $F^{\infty} = f$ and $G^{\infty} = g$ and the forgetful polynomial X so that $X^{\infty} = f + g$.

Output: The $\succ$-next delayed term of X.

```
1: if |X|=0 then
2:   {X has no information.}
3:   (t_F, t_G) ← (next(F), next(G));
4: else
5:   Set t_F and t_G to the values associated with X;
6: end if
7: while t_F ≠ 0 or t_G ≠ 0 do
8:   if t_F and t_G are like terms then
9:     ans ← t_F + t_G;
10:    (t_F, t_G) ← (next(F), next(G))
11:  else if t_F ≠ 0 and t_F ≻ t_G then
12:    ans ← t_F;
13:    t_F ← next(F)
14:  else if t_G ≠ 0 then
15:    ans ← t_G
16:    t_G ← next(G)
17:  end if
18:  if ans ≠ 0 then
19:    Associate t_F and t_G with X;
20:    return ans;
21:  end if
22: end while
23: Associate t_F and t_G with X;
24: return 0;
```

Algorithm 9 Specialized Heap-Division

Input: The forgetful polynomial F and delayed polynomial G so that $F^{\infty} = f$ and $G^{\infty} = g$.
Output: The delayed polynomials Q and R so that $f = g \cdot Q^{\infty} + R^{\infty}$.

1: $t_F \leftarrow \texttt{next}(F)$;
2: **if** $t_F = 0$ **then**
3: Set Q and R to zero.
4: **return** Q and R.
5: **end if**
6: Initialize a new heap H and insert t_F into H;
7: $s \leftarrow 2$;
8: **while** H is not empty **do**
9: $t \leftarrow 0$;
10: **repeat**
11: Extract $x \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + x$.
12: **if** $x = t_F$ **then**
13: $t_F = \texttt{next}(F)$
14: **if** $t_F \neq 0$ **then**
15: Insert t_F into H;
16: **end if**
17: **else if** $x = G_i Q_j$ and Q_{j+1} is forced **then**
18: Insert $-G_i Q_{j+1}$ into H;
19: **else if** $x = G_i Q_j$ and Q_{j+1} is delayed **then**
20: s $\leftarrow$ s + 1; {Sleep $-G_i Q_{j+1}$}
21: **end if**
22: **if** $x = G_i Q_1$ and $G_{i+1} \neq 0$ **then**
23: Insert $-G_{i+1} Q_1$ into H;
24: **end if**
25: **until** (H is empty) or (t and H_{max} are not like terms).
26: **if** $t \neq 0$ and $g_1 | t$ **then**
27: $Q_{|Q|+1} \leftarrow t/G_1$; {Now $Q_{|Q|+1}$ is a forced term.}
28: **for** k from 2 to s **do**
29: Insert $-G_k \cdot t/G_1$ into H; {Insert all terms that are sleeping into H}
30: **end for**
31: **else if** $t \neq 0$ **then**
32: $R_{|R|+1} \leftarrow t$; {Now $R_{|R|+1}$ is a forced term.}
33: **end if**
34: **end while**
35: **return** Q and R;

Chapter 3

Applications

3.1 Bareiss' Algorithm

Finding determinants and solving linear systems is a fundamental problem in computing with a rich history. Leibnitz was the first to use determinants in 1693, with Cramer presenting a determinant based method for solving linear equations in the late 1700s. Gaussian elimination, which solves linear systems by doing a succession of elementary matrix operations, was formally outlined by Gauss in a publication in 1809. In contemporary computing, solving linear systems poses as much of a problem as it did centuries ago. Practice shows that there are inherent weaknesses in Gaussian elimination. For instance, when using this algorithm on matrices with real entries, calculating differences of real numbers required to do the row reductions becomes problematic because they can lead to numeric instabilities. Using exact computation as one would do in symbolic computing is also problematic. Although working exactly with fractions eliminates any numerical issues, the use of GCD's to reduce these fractions has a substantial cost. This cost becomes especially significant if we are trying to solve a system with rational function coefficients.

One solution to this problem would be to avoid fractions altogether. This is the basic idea behind "fraction free" Gaussian elimination. In this method, instead of normalizing the rows to do row reductions, we multiply each row by some factor so that the leading element in each row is the same. For example, to eliminate the leading entry in $[7, 2, 3]$ using the row $[5, 3, 2]$ we would have to multiply the rows by five and seven respectively. This gives $5 \cdot [7, 2, 3] - 7 \cdot [5, -3, 2] = [0, 31, 1]$ and puts a relatively large term in the diagonal. This deficiency is a crucial one as it turns out that there can be an exponentially large increase

in the digit length of the diagonal entries. This will cause the fraction-free scheme to be *slower* then the ordinary Gaussian elimination scheme [10, p. 392].

There is a "fraction free" approach for calculating determinants that eliminates this problem. It is given by Algorithm 10 and is due to Bareiss [2] who noted that the method was first known to Jordan. The algorithm does exact divisions over any integral domain to avoid fractions and explosions of the matrix entries. For now it is enough to understand that it takes as input a matrix $\mathbf{M}$, with (i,j)-th entry $(\mathbf{M})_{i,j}$, and returns $\mathbf{M}$'s determinant. The details of the algorithm and a proof of its correctness will be given in the following sections.

Algorithm 10 Bareiss Algorithm

Input: $\mathbf{M}$ an n-square matrix with entries over an integral domain $\mathcal{D}$ with $x_{k,k}^{(k)}$ all non-zero.
Output: The determinant of $\mathbf{M}$. In general, for $i > k$, we have

$$(\mathbf{M})_{i,k} = x_{i,k}^{(k)}, \text{ and } (\mathbf{M})_{k,i} = x_{k,i}^{(k)}.$$

1: $(\mathbf{M})_{0,0} \leftarrow 1$;
2: **for** $k = 1$ to $n-1$ **do**
3: **for** $i = k+1$ to n **do**
4: **for** $j = k+1$ to n **do**
5: $(\mathbf{M})_{i,j} \leftarrow \frac{(\mathbf{M})_{k,k}(\mathbf{M})_{i,j} - (\mathbf{M})_{i,k}(\mathbf{M})_{k,j}}{(\mathbf{M})_{k-1,k-1}}$ {Exact division.}
6: **end for**
7: **end for**
8: **end for**
9: **return** $(\mathbf{M})_{n,n}$

3.1.1 The Problem With Bareiss' Algorithm

We restrict to matrices with entries from the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots, x_n]$. Although line (5) (an *exact* division which defines the Bareiss algorithm) can be credited for keeping the diagonal entires a reasonable size, the intermediate calculations it must do can often get out of hand. It is quite possible (in fact typical) that this calculation (of the form $\frac{A \cdot B - C \cdot D}{E}$) produces a numerator that is *much* larger than the corresponding quotient and denominator. This happens when A, B, C, D, E are sparse polynomials.

Example 3.1.1. Consider the so-called symmetric Toeplitz matrix with entries from the

polynomial ring $\mathbb{Z}[x_1, x_2, \ldots, x_9]$ generated by $[x_1, \ldots, x_9]$,

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_9 \\ x_2 & x_1 & x_2 & \cdots & x_8 \\ x_3 & x_2 & x_1 & \cdots & x_7 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_9 & \cdots & x_3 & x_2 & x_1 \end{bmatrix}.$$

When calculating the determinant of this matrix using Bareiss' algorithm the last division (in line (5) of Algorithm 10) will have a dividend of 128,530 terms, whereas the divisor and quotient will only have 427 and 6,090 terms respectively.

So, in practice one often finds that an inordinate amount of memory is used to compute what can be a relatively small determinant. This deficiency is the reason that Bareiss' algorithm does not experience more widespread use (for example it is implemented in Maple but not used).

To overcome this problem recall that we introduced forgetful operations in Chapter 2. In doing so we managed to minimize the amount of memory used to expand a polynomial expression. The upshot of this is that now we can construct the quotient of $\frac{A \cdot B - C \cdot D}{E}$ without having to store $A \cdot B - C \cdot D$ in its entirety (in fact the forgetful algorithms were designed to do precisely this calculation).

Theorem 3.1.2. *Calculating $Q = \frac{A \cdot B - C \cdot D}{E}$ (an exact division) with forgetful operations requires at most $O(\max(\#A, \#B) + \max(\#C, \#D) + 1 + \#E + \#Q)$ storage.*

Proof. We have from Theorem 2.3.4 that the products $A \cdot B$ and $C \cdot D$ will require at most $\max(\#A, \#B)$ and $\max(\#C, \#D)$ space, where the difference of these products requires $O(1)$ space by Theorem 2.3.3. Since there is no remainder because the division is exact, the division algorithm will use $O(\#E + \#Q)$ storage by Theorem 1.3.25. Summing these complexities gives the desired result. □

Remark 3.1.3. The implications of this theorem and can be observed in the package that implements the Bareiss Algorithm with forgetful polynomials. If we measure the amount of memory used by the exact division on line (5) we would expect this to have a linear relationship with the size of its inputs. We can observe this relationship in Figures 3.1, 3.2, and 3.3. Each figure is a log-log plot where the horizontal axis represents the value

$\max(\#A, \#B) + \max(\#C, \#D) + 1 + \#E + \#Q$ from some arbritrary iteration, and the vertical access measures maximum memory usage (in bits) of the associated division.

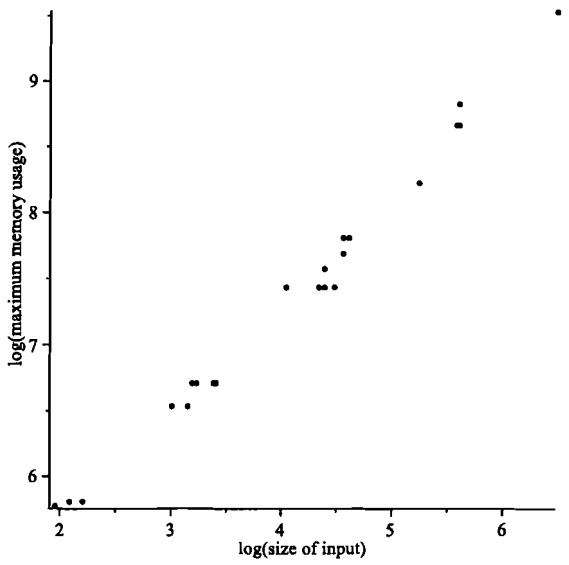


Figure 3.1: Maximum memory usage for the exact division of the 'forgetful' Bareiss' Algorithm when given the matrix A corresponding to the Maple code:

```
A := LinearAlgebra[ToeplitzMatrix](var,nops(var),symmetric);
```

where `var:=[seq(y[i],i=1..7)]`.

3.1.2 Sylvester's Identity

Our goal in this section is to prove that the division in the Bareiss algorithm is exact [3]. To begin we introduce the notation and definitions necessary for this section. Denote the set of $m \times n$ matrices with entries in the domain $\mathcal{D}$ by $\mathcal{D}^{m \times n}$. Also, let the (i, j)-th entry of a matrix $\mathbf{M} \in \mathcal{D}^{m \times n}$ be denoted as $(\mathbf{M})_{i,j}$.

Definition 3.1.4. The *determinant* of a matrix $\mathbf{M} \in \mathcal{D}^{n \times n}$, denoted $\det(\mathbf{M})$, is recursively defined as follows:

1. $\det\left(\begin{bmatrix} x_{1,1} \end{bmatrix}\right) = x_{1,1}$ for any $x_{1,1} \in \mathcal{D}$.

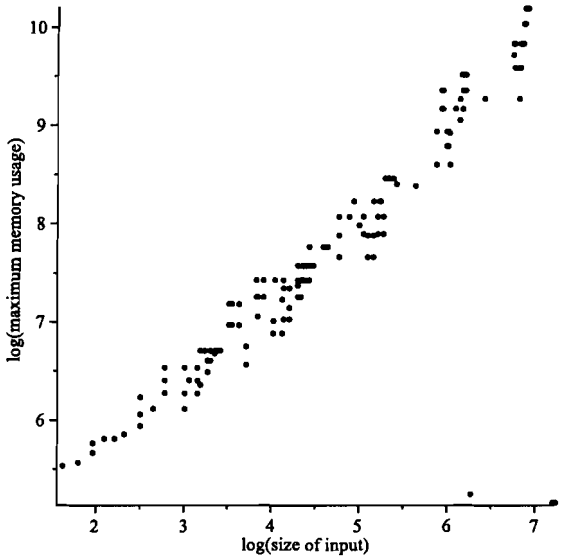


Figure 3.2: Maximum memory usage for the exact division of the 'forgetful' Bareiss' Algorithm when given the matrix A corresponding to the Maple code:

$$\texttt{A} := \texttt{LinearAlgebra[ToeplitzMatrix]([op(var), op(var)], 2 * nops(var), symmetric);}$$

where `var:=[seq(y[i],i=1..6)]`.

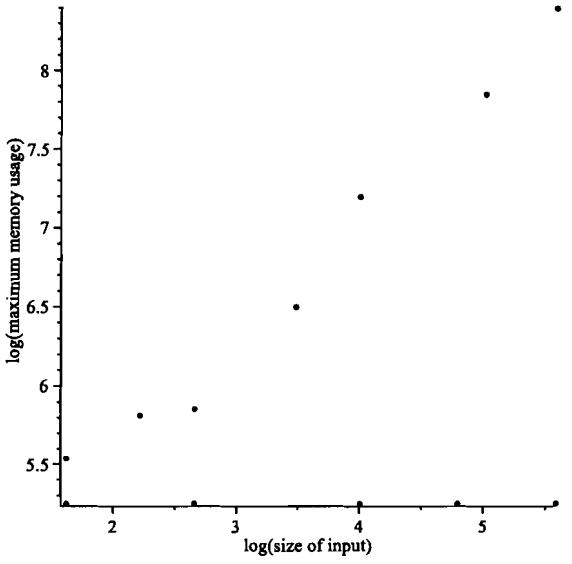


Figure 3.3: Maximum memory usage for the exact division of the 'forgetful' Bareiss' Algorithm when given the matrix A corresponding to the Maple code:

$$\texttt{A} := \texttt{LinearAlgebra} : -\texttt{VandermondeMatrix}([\texttt{op}(\texttt{var}), \texttt{op}(\texttt{var})]);$$

where `var:=[seq(y[i],i=1..5)]`.

2. $\det(\mathbf{M}) = \sum_{j=1}^{m} (\mathbf{M})_{i,j} \cdot [\mathbf{M}]_{i,j}$ for any $1 \leq i \leq n$.

where $[\mathbf{M}]_{i,j}$ denotes the (i,j)-*cofactor* of $\mathbf{M}$ given as

$$[\mathbf{M}]_{i,j} \equiv (-1)^{i+j} \det(\mathbf{M}[i;j])$$

and $\mathbf{M}[i;j]$ is the matrix obtained by removing the i-th row and j-th column from $\mathbf{M}$.

We briefly note, without proof, some useful properties of the determinant. For $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathcal{D}^{n \times n}$, r some constant in $\mathcal{D}$, and $\mathbf{0}$ the zero matrix of dimension $n \times n$ we have:

1. $\det(\mathbf{A} \cdot \mathbf{B}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$

2. $\det(r\mathbf{A}) = r^n \det(\mathbf{A})$

3. $\det\left(\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{C} \end{bmatrix}\right) = \det\left(\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{B} & \mathbf{C} \end{bmatrix}\right) = \det \mathbf{A} \cdot \det \mathbf{C}$

Definition 3.1.5. The *adjoint* of a matrix $\mathbf{M} \in \mathcal{D}^{n \times n}$ is the matrix $\text{adj}(\mathbf{M})$ whose (i,j)-th entry is the (j,i)-cofactor of $\mathbf{M}$. Namely

$$(\text{adj}(\mathbf{M}))_{i,j} = [\mathbf{M}]_{j,i}$$

(note the transposed subscripts).

For the remainder of this section let $\mathbf{M} \in \mathcal{D}^{n \times n}$ with $(\mathbf{M})_{i,j} = x_{i,j}$. It is easy to see that

$$x_{i,1}[\mathbf{M}]_{j,1} + x_{i,2}[\mathbf{M}]_{j,2} + \cdots + x_{i,n}[\mathbf{M}]_{j,n} = \begin{cases} 0, & \text{if } i \neq j \\ \det(\mathbf{M}), & \text{if } i = j \end{cases}. \tag{3.1}$$

When $i = j$ this is precisely the definition of the determinant of $\mathbf{M}$. When $i \neq j$ this sum can be regarded as the determinant of the matrix obtained by replacing $\mathbf{M}$'s j-th row with $\mathbf{M}$'s i-th row. Now we can subtract $\mathbf{M}$'s j-th row from $\mathbf{M}$'s i-th row to obtain a row of zeros. This enables us to choose an i for part 2 of Definition 3.1.4, where every $(\mathbf{M})_{i,j} = 0$, resulting in a zero determinant.

Using (3.1), we can show that

$$\mathbf{M} \cdot \text{adj}(\mathbf{M}) = \det(\mathbf{M}) \cdot \mathbf{I} \tag{3.2}$$

since any entry $(\mathbf{M} \cdot \mathrm{adj}\,(\mathbf{M}))_{i,j} = \sum_{k=1}^{n} x_{i,k} \left[\mathbf{M}\right]_{j,k}$ which is only non-zero when $i = j$, for which it is the determinant of $\mathbf{M}$. Taking the determinant of both sides of (3.2) we derive

$$\det(\mathbf{M} \cdot \mathrm{adj}\,(\mathbf{M})) = \det(\det(\mathbf{M}) \cdot \mathbf{I}) \tag{3.3}$$

$$\det(\mathbf{M}) \cdot \det(\mathrm{adj}\,(\mathbf{M})) = \det(\mathbf{M})^{n} \tag{3.4}$$

$$\det(\mathrm{adj}\,(\mathbf{M})) = \det(\mathbf{M})^{n-1} \tag{3.5}$$

and if we assume that that $\det(\mathbf{M})$ is an invertible element of $\mathcal{D}$ we get

$$(\mathbf{M})^{-1} = (\det(\mathbf{M}))^{-1} \mathrm{adj}\,(\mathbf{M}). \tag{3.6}$$

Bareiss algorithm is based on Sylvester's identity;

Lemma 3.1.6. Sylvester's identity

$$\left(x_{k-1,k-1}^{(k-1)}\right)^{n-k} \det(\mathbf{M}) = \det\left(\begin{bmatrix} x_{k,k}^{(k)} & x_{k,k+1}^{(k)} & \cdots & x_{k,k}^{(n)} \\ x_{k+1,k}^{(k)} & x_{k+1,k+1}^{(k)} & \cdots & x_{k+1,n}^{(k)} \\ \vdots & & \ddots & \vdots \\ x_{n,k}^{(k)} & x_{n,k+1}^{(k)} & \cdots & x_{n,n}^{(k)} \end{bmatrix}\right).$$

In order to develop this lemma we state and prove two additional lemmas. We begin by expressing $\mathbf{M}$ in the form

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{X} \end{bmatrix}$$

where $\mathbf{A}$ is $(k-1)$-square and $\mathbf{X}$ is $(n-(k-1))$-square. Let us also assume that $\delta = \det(\mathbf{A}) \neq 0$.

Lemma 3.1.7.

$$\delta^{n-k} \det(\mathbf{M}) = \det(\delta\mathbf{X} - \mathbf{C} \cdot \mathrm{adj}(\mathbf{A}) \cdot \mathbf{B})$$

Proof. If we express

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & (\mathbf{A})^{-1}\mathbf{B} \\ \mathbf{0} & \mathbf{X} - \mathbf{C}\,(\mathbf{A})^{-1}\mathbf{B} \end{bmatrix}$$

then, by taking the determinant of both sides, we get

$$\begin{aligned}
\det(\mathbf{M}) &= \det\left(\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & \mathbf{I} \end{bmatrix}\right) \cdot \det\left(\begin{bmatrix} \mathbf{I} & (\mathbf{A})^{-1}\mathbf{B} \\ \mathbf{0} & \mathbf{X} - \mathbf{C}(\mathbf{A})^{-1}\mathbf{B} \end{bmatrix}\right) \\
&= \det(\mathbf{A}) \cdot \det(\mathbf{I}) \cdot \det(\mathbf{I}) \cdot \det\left(\mathbf{X} - \mathbf{C}(\mathbf{A})^{-1}\mathbf{B}\right) \\
&= \delta \cdot \det\left(\mathbf{X} - \mathbf{C}(\mathbf{A})^{-1}\mathbf{B}\right) \\
\Rightarrow \delta^{n-k}\det\mathbf{M} &= \delta^{n-(k-1)}\det\left(\mathbf{X} - \mathbf{C}(\mathbf{A})^{-1}\mathbf{B}\right) \\
&= \det\left(\delta\mathbf{X} - \mathbf{C}\left(\delta(\mathbf{A})^{-1}\right)\mathbf{B}\right) \qquad \text{By determinant property 2.} \\
&= \det(\delta\mathbf{X} - \mathbf{C} \cdot \operatorname{adj}(\mathbf{A}) \cdot \mathbf{B}) \qquad \text{By (3.6).}
\end{aligned}$$

□

To study the structure of $\mathbf{X} - \mathbf{C}(\mathbf{A})^{-1}\mathbf{B}$, we introduce:

Definition 3.1.8. The *"(r,s)-bordered matrix of order k"* of a matrix $\mathbf{M}$ where $(\mathbf{M})_{i,j} = x_{i,j}$ is defined to be

$$\mathbf{M}_{r,s}^{(k)} \equiv \left[\begin{array}{cccc|c} x_{1,1} & x_{1,2} & \cdots & x_{1,k-1} & x_{1,s} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k-1} & x_{2,s} \\ \vdots & \vdots & & \vdots & \vdots \\ x_{k-1,1} & x_{k-1,2} & \cdots & x_{k-1,k-1} & x_{k-1,s} \\ \hline x_{r,1} & x_{r,2} & \cdots & x_{r,k-1} & x_{r,s} \end{array}\right]$$

for $k \leq \min(r,s)$. For the case where $k = 1$ we let $\mathbf{M}_{r,s}^{(1)} = \begin{bmatrix} x_{r,s} \end{bmatrix}$. We also define

$$x_{r,s}^{(k)} = \det\left(\mathbf{M}_{r,s}^{(k)}\right) \tag{3.7}$$

and note that $\mathbf{M} = \mathbf{M}_{n,n}^{(n)}$ and $\det(\mathbf{M}) = x_{n,n}^{(n)}$.

Example 3.1.9. The matrix

$$\mathbf{M} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & x_{3,5} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & x_{4,5} \\ x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & x_{5,5} \end{bmatrix}$$

has

$$\mathbf{M}^{(2)}_{3,5} = \begin{bmatrix} x_{1,1} & x_{1,5} \\ x_{3,1} & x_{3,5} \end{bmatrix} \quad \mathbf{M}^{(3)}_{4,3} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{4,1} & x_{4,2} & x_{4,3} \end{bmatrix} \quad \mathbf{M}^{(3)}_{4,5} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,5} \\ x_{2,1} & x_{2,2} & x_{2,5} \\ x_{4,1} & x_{4,2} & x_{4,5} \end{bmatrix}.$$

In order to state our next lemma, we first consider an arbitrary entry of $\delta\mathbf{X}-\mathbf{C}\cdot\mathrm{adj}\,(\mathbf{A})\,\mathbf{B}$. For $r \geq k$ and $s \geq k$ we have

$$\begin{aligned}
(\delta\mathbf{X} - \mathbf{C}\cdot\mathrm{adj}\,(\mathbf{A})\,\mathbf{B})_{r-(k-1),s-(k-1)} &= \delta x_{r,s} - (\mathbf{C}\cdot\mathrm{adj}\,(\mathbf{A})\,\mathbf{B})_{r-(k-1),s-(k-1)} && (3.8)\\
&= \delta x_{r,s} - \sum_{i=1}^{k-1} (\mathbf{C})_{r-(k+1),i}\cdot(\mathrm{adj}\,(\mathbf{A})\,\mathbf{B})_{i,s-(k+1)} && (3.9)\\
&= \delta x_{r,s} - \sum_{i=1}^{k-1} (\mathbf{C})_{r-(k+1),i}\cdot\sum_{j=1}^{k-1}(\mathrm{adj}\,(\mathbf{A}))_{i,j}\cdot(\mathbf{B})_{j,s-(k+1)} && (3.10)\\
&= \delta x_{r,s} - \sum_{i=1}^{k-1} x_{r,i}\cdot\sum_{j=1}^{k-1}[\mathbf{A}]_{j,i}\,x_{j,s} && (3.11)
\end{aligned}$$

where

$$\det\left(\mathbf{M}^{(k)}_{r,s}\right) = \det\begin{bmatrix} & & & x_{1,s} \\ & \mathbf{A} & & \vdots \\ & & & x_{k-1,s} \\ x_{r,1} & \cdots & x_{r,k-1} & x_{r,s} \end{bmatrix} = \det(\mathbf{A})\cdot x_{r,s} - \sum_{i=1}^{k-1} x_{r,i}\cdot\sum_{j=1}^{k-1}[\mathbf{A}]_{j,i}\,x_{j,s}$$

which is precisely the expression given in (3.11). This immediately yields:

Lemma 3.1.10. *For $r \geq k$ and $s \geq k$*

$$(\delta\mathbf{X} - \mathbf{C}\cdot adj\,(\mathbf{A})\,\mathbf{B})_{r,s} = x^{(k)}_{r,s}.$$

Now, combining the last two lemmas, and noting that $\delta = x^{(k-1)}_{k-1,k-1}$, proves Sylvester's identity: Lemma 3.1.6.

3.1.3 Correctness of Bareiss' Algorithm

We now present a proof of the correctness of Bareiss' algorithm for fraction-free determinant computation. With the notations of the previous section, Lemma 3.1.6 with $n-k = 1$ (called

the "first order" Sylvester identity) reads

$$x_{k-1,k-1}^{(k-1)} \det\left(\mathbf{M}_{i,j}^{(k+1)}\right) = \det\left(\begin{bmatrix} x_{k,k}^{(k)} & x_{k,j}^{(k)} \\ x_{i,j}^{(k)} & x_{i,j}^{(k)} \end{bmatrix}\right)$$

which further implies

$$x_{i,j}^{(k+1)} = \frac{x_{k,k}^{(k)} x_{i,j}^{(k)} - x_{i,k}^{(k)} x_{k,j}^{(k)}}{x_{k-1,k-1}^{(k-1)}}. \tag{3.12}$$

We see that (3.12) is another way of expressing a determinant. That is, one can compute a determinant by this 'update' formula, as a combination of determinants of sub matrices. The crucial feature of (3.12) is that the division is exact, thus avoiding fractions *and* preventing explosion of numbers in the diagonal. This is the defining step of Bareiss' algorithm; Algorithm 10.

The correctness of this algorithm is easily proved by induction on k, and by an appeal to equation (3.12). We can also show the division in line (5) is exact because $(\mathbf{M})_{k-1,k-1} = x_{k-1,k-1}^{(k-1)}$, or by another appeal to (3.12). Hence, all the computed values remain inside the domain $\mathcal{D}$.

Proof of Bareiss Algorithm Correctness. The base case for $k = 1$, where $i, j > k$ gives

$$\begin{aligned}(\mathbf{M})_{i,j} &= \frac{(\mathbf{M})_{i,j}(\mathbf{M})_{1,1} - (\mathbf{M})_{i,1}(\mathbf{M})_{1,j}}{(\mathbf{M})_{0,0}} \\ &= \frac{x_{i,j}^{(1)} x_{1,1}^{(1)} - x_{i,1}^{(1)} x_{1,j}^{(1)}}{x_{0,0}^{(0)}} \\ &= x_{i,j}^{(2)} \text{ by (3.12)}\end{aligned}$$

For the induction, assume that at iteration $k-1$ it is true that $(\mathbf{M})_{i,j} = x_{i,j}^{(k)}$ for $i, j \geq k$. Now, for the k-th iteration, we have $i = k+1, \cdots, n$ and $j = k+1, \cdots, n$, meaning $i, j \geq k$. Therefore, by our assumption, we have

$$\begin{aligned}(\mathbf{M})_{i,j} &= \frac{(\mathbf{M})_{i,j}(\mathbf{M})_{k,k} - (\mathbf{M})_{i,k}(\mathbf{M})_{k,j}}{(\mathbf{M})_{k-1,k-1}} \\ &= \frac{x_{i,j}^{(k)} x_{k,k}^{(k)} - x_{i,k}^{(k)} x_{k,j}^{(k)}}{x_{k-1,k-1}^{(k-1)}} \\ &= x_{i,j}^{(k+1)} \text{ by (3.12).}\end{aligned}$$

The induction terminates when $k = n-1$, $i = n$, $j = n$; we then have $(\mathbf{M})_{n,n} = x_{n,n}^{(n)} = \det(\mathbf{M})$ as desired. □

3.2 The Subresultant Algorithm

The resultant is a fundamental operation in computer algebra that plays a central role in many algorithms. A resultant can be constructed as the determinant of a "Sylvester matrix", a matrix formed by repeating the coefficients of two polynomials in a particular way. As we have seen, calculating a determinant is not necessarily a simple task, especially when the entries of the matrix are polynomials (as is the case here).

For this reason the resultant is typically computed by using a polynomial remainder sequence (or PRS), which allows us to bypass explicit determinant calculation. We will review a succession of PRSs that will each address weaknesses in its predecessor. These problems are similar to those of determinant calculation in that one will have to develop a fraction-free method that avoids explosions of intermediate terms.

However, as observed with the previously developed fraction-free method, we may still encounter problems in the arithmetic. We saw that the Bareiss algorithm, despite utilizing a system that overcomes many initial obstacles, may still be impractical in application. The subresultant algorithm for calculating a resultant has a similar flaw, and fortunately a similar solution.

The subresultant algorithm [4],[8] is given by Algorithm 11. The operations $\texttt{deg}_x$, `prem` and $\texttt{lcoeff}_x$, stand for the degree in x, pseudo-remainder, and leading coefficient in x (respectively). All of these will be defined later. The problematic calculation occurs when finding the pseudo-remainder on line (4). It can be easily demonstrated, especially when u and v are sparse polynomials in many variables, that $\tilde{r}$ is very large relative to the dividend and quotient given by the division on line (6). In fact $\tilde{r}$ can be much larger than the resultant $\text{Res}(u, v, x)$. To resolve this we will have the pseudo-remainder return a forgetful polynomial so that the numerator on line (6) will not have to be explicitly stored. This is precisely the same strategy used in the Bareiss calculation.

3.2.1 The Subresultant PRS

In the following sections we will develop the mathematics for working with resultants. The ultimate goal is to construct a polynomial remainder sequence (abbreviated PRS) that will compute resultants.

Sylvester's resultant is defined for univariate polynomials, but this will not restrict us from taking the resultant of polynomials in many variables as well. This is because we can

Algorithm 11 Subresultant Algorithm

Input: The polynomials $u, v \in \mathcal{D}[x]$ where $\texttt{deg}_x(u) > \texttt{deg}_x(v)$.
Output: $r = \text{Res}(u, v, x)$.

```
1: (g, h) ← (1, −1);
2: while deg_x(v) ≠ 0 do
3:    d ← deg_x(u) − deg_x(v);
4:    r̃ ← prem(u, v, x); {r̃ is big.}
5:    u ← v;
6:    v ← r̃ ÷ (−g · h^d); {Exact division with v much smaller than r̃.}
7:    g ← lcoeff_x(u);
8:    h ← (−g)^d ÷ h^{d−1}; {Exact division.}
9: end while
10: r ← v;
11: return r;
```

regard any multivariate polynomial $f \in k[x_1, \ldots, x_n]$ as a univariate polynomial in x_1 with coefficients from the ring $k[x_2, \ldots, x_n]$.

Example 3.2.1. $f = xy + xyz^2 + y^3z + x^2z^3 + xy^3z + yz^4 \in \mathbb{Z}[x, y, z]$ can be re-written as $f = (z^3)x^2 + (y + y^3z + yz^2)x + (yz^4 + y^3z) \in \mathbb{Z}[y, z][x]$.

To emphasize that our coefficients may be polynomials we will use the ring $\mathcal{D}[x]$ (for some UFD $\mathcal{D}$). Now, when referring to the degree of a polynomial, we will use $\texttt{deg}_x(f)$ to indicate the polynomial's degree in *only* the variable x. We also define $\texttt{lcoeff}_x(f)$ (leading coefficient of f) to be the coefficient from the terms f_i with largest $\texttt{deg}_x(f_i)$. For example, $\texttt{deg}_x(f) = 2$ and $\texttt{lcoeff}_x(f) = z^3$ when f is taken from Example 3.2.1.

Definition 3.2.2. For polynomials $f, g \in \mathcal{D}[x]$, with positive degree, write

$$f = a_0x^m + \ldots + a_m$$
$$g = b_0x^n + \ldots + b_n$$

where $a_0 \neq 0$, $b_0 \neq 0$ and $m > 0$, $n > 0$. We define the *Sylvester matrix* of f and g with

respect to x, denoted $\mathrm{Syl}(f,g,x)$, as the $(m+n)$-square matrix

$$\mathrm{Syl}(f,g,x) = \underbrace{\begin{bmatrix} a_0 & & & & b_0 & & & \\ a_1 & a_0 & & & b_1 & b_0 & & \\ a_2 & a_1 & \ddots & & b_2 & b_1 & \ddots & \\ \vdots & a_2 & \ddots & a_0 & \vdots & b_2 & \ddots & b_0 \\ a_m & \vdots & \ddots & a_1 & b_n & \vdots & \ddots & b_1 \\ & a_m & & a_2 & & b_n & & b_2 \\ & & \ddots & \vdots & & & \ddots & \vdots \\ & & & a_m & & & & b_n \end{bmatrix}}_{n \text{ columns}} \underbrace{\qquad\qquad}_{m \text{ columns}}$$

(where the blanks are understood to be zeros).

Definition 3.2.3. The *resultant* of f and $g \in \mathcal{D}[x]$ with respect to x, denoted by $\mathrm{Res}(f,g,x)$, is given by

$$\mathrm{Res}(f,g,x) = \det\left(\mathrm{Syl}(f,g,x)\right).$$

Example 3.2.4. For the polynomials $f = xy - 1$ and $g = x^2 + y^2 - 4$, regarding f and g as polynomials in x with coefficients over $\mathbb{Z}[y]$, we have

$$\mathrm{Res}(f,g,x) = \det\left(\begin{bmatrix} y & 0 & 1 \\ -1 & y & 0 \\ 0 & -1 & y^2-4 \end{bmatrix}\right) = y^4 - 4y^2 + 1.$$

As previously stated we would like to avoid the explicit use of determinants. To do this we will use a PRS, which (as its name implies) is a sequence of polynomial remainders. A simple and well known PRS is the sequence constructed by Euclid's algorithm for gcd calculation. Given $f, g \in \mathcal{D}[x]$ with $\mathtt{deg}_x(f) \geq \mathtt{deg}_x(g)$ this algorithm constructs a sequence of univariate polynomials $R_0 = f, R_1 = g, \cdots, R_k \in \mathcal{D}[x]$ where

$$\begin{aligned} R_0 &= R_1 \cdot Q_1 + R_2 \quad \text{with } \mathtt{deg}_x(R_2) < \mathtt{deg}_x(R_1) \\ R_1 &= R_2 \cdot Q_2 + R_3 \quad \text{with } \mathtt{deg}_x(R_3) < \mathtt{deg}_x(R_2) \\ &\vdots \\ R_{k-2} &= R_{k-1} \cdot Q_2 + R_k \quad \text{with } \mathtt{deg}_x(R_k) < \mathtt{deg}_x(R_{k-1}) \\ R_{k-1} &= R_k \cdot Q_k. \end{aligned}$$

The names Q and R were deliberately chosen to reflect that Q_i and R_{i+1} are the quotient and remainder from the division $R_i \div R_{i-1}$. The essential property of the sequence is that $\gcd(R_{i-1}, R_i) = \gcd(R_i, R_{i+1})$ which implies that R_k is an associate of the $\gcd(f, g)$ in $\mathcal{D}[x]$.

In general, polynomial remainder sequences enable us to calculate a variety of values in different ways. For instance there are variations of the PRS for Euclid's algorithm that are fraction-free and others that do exact divisions to keep coefficients small. This is usually done by using pseudo-division to create a sequence of polynomial pseudo-remainders.

Definition 3.2.5. Let $\mathcal{D}$ be a UFD and $f, g \in \mathcal{D}[x]$ with $f \neq 0$, $g \neq 0$. Let $\alpha = \mathtt{lcoeff}_x(g)^{\delta+1}$ where $\delta = \deg_x(f) - \deg_x(g)$. Then the *pseudo-remainder* $\tilde{r}$ of f divided by g is defined as the remainder of αf divided by g. The *pseudo-quotient* $\tilde{q}$ is similarly defined as the quotient of the same division. Thus $\alpha f = g\tilde{q} + \tilde{r}$ with $\tilde{r} = 0$ or $\deg_x(\tilde{r}) < \deg_x(g)$ by Theorem 1.3.23.

Theorem 3.2.6. *Let $\mathcal{D}$ be a UFD and let g be a nonzero polynomial in $\mathcal{D}[x]$. Then for every nonzero $f \in \mathcal{D}[x]$ the pseudo-quotient $\tilde{q}$ and pseudo-remainer $\tilde{r}$ of f, g are in the ring $\mathcal{D}[x]$. Furthermore, $\tilde{q}$ and $\tilde{r}$ are unique.*

Proof. The proof is similar to that of Theorem 1.3.23 for regular division. It can be shown that the modified division algorithm yields exact coefficient arithmetic; thereby generating $\tilde{q}, \tilde{r} \in \mathcal{D}[x]$ (instead of potentially in $\mathcal{D}/\mathcal{D}[x]$). □

Example 3.2.7. For $f = 3x^3 + x^2 + x + 5, g = 5x^2 - 3x + 1 \in \mathbb{Z}[x]$, dividing f by g would produce the quotient and remainder

$$q = \frac{3}{5}x + \frac{14}{25} \quad \text{and} \quad r = \frac{52}{25}x + \frac{111}{25}.$$

Whereas, if we premultiplied f by 5^2 and divided $5^2 f$ by g we would get a pseudo-quotient and pseudo-remainder

$$\tilde{q} = 15x + 14 \quad \text{and} \quad \tilde{r} = 52x + 111.$$

Moreover, no fractions appear while executing the division algorithm thereby avoiding calculations in $\mathbb{Q}$.

Remark 3.2.8. The previously developed division algorithms can be used to do pseudo-division. To do this we multiply the dividend by the appropriate α and the division algorithm

will return the correct pseudo-remainder and pseudo-quotient while avoiding fractions. We will assume this algorithm has been implemented as $\texttt{prem}(f, g, x)$, which returns the pseudo-remainder $\tilde{r}$ of $f \div g$ when the polynomials are considered univariate in x. We will also assume the existence $\texttt{pquo}(f, g, x)$ which returns the pseudo-quotient $\tilde{q}$.

Remark 3.2.9. Since we are regarding the polynomials as univariate, we must modify the monomial ordering to produce the correct leading terms. The new monomial ordering must calculate the leading term of $x + y^2z^3$ to be x and not yz^3 as $\geqslant_{grlex}$ would imply. One such ordering is a product ordering [10] defined by $f_i >_{ord} f_j$ when $\texttt{deg}_x(f_i) > \texttt{deg}_x(f_j)$ or when $\texttt{deg}_x(f_i) = \texttt{deg}_x(f_j)$ and $f_i \geqslant_{grlex} f_j$.

Definition 3.2.10 (PRS)**.** For $\mathcal{D}$ a UFD, let f and g be polynomials in $\mathcal{D}[x]$ with $\texttt{deg}_x(f) \geqslant \texttt{deg}_x(g)$. A *polynomial remainder sequence* (PRS) for f and g is a sequence of polynomials $R_0, \cdots, R_k \in \mathcal{D}[x]$ satisfying the following conditions.

1. $R_0 = f$, $R_1 = g$
2. $\alpha_i \cdot R_{i-1} = Q_i \cdot R_i + \beta_i \cdot R_{i+1}$ with $\alpha_i, \beta_i \in \mathcal{D}$ for $i = 1, 2, \ldots, k-1$.
3. $\texttt{prem}(R_{k-1}, R_k, x) = 0$.

The subresultant PRS is the culmination of improvements that were incrementally made to the basic Euclidean PRS. To best understand these improvements we will review a collection of examples used by Geddes et al. [10] and Knuth [13], starting with Euclid's algorithm.

Example 3.2.11 (Euclid's Algorithm)**.** Let $f, g \in \mathbb{Z}[x]$ be given by

$$f = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$
$$g = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

Since $\mathbb{Z}$ is not a field it will be necessary to carry out divisions in $\mathbb{Q}[x]$. The sequence of remainders given by Euclid's algorithm (which is the PRS with $\alpha_i = \beta_i = 1$) are

$$\begin{aligned}
R_2 &= -\frac{5}{9}x^4 + \frac{1}{9}x^2 - \frac{1}{3}, \\
R_3 &= -\frac{117}{25}x^2 - 9x + \frac{411}{25}, \\
R_4 &= \frac{233150}{19773}x - \frac{102500}{6591}, \\
R_5 &= -\frac{1288744821}{543589225}.
\end{aligned}$$

Which implies that f and g are relatively prime since their greatest common divisor is a unit in $\mathbb{Q}[x]$.

Example 3.2.11 exhibits the same problem as calculating determinants. The use of gcds to simplify fractions becomes significant (especially when the coefficient ring is a polynomial ring). One resolution to this problem would be to avoid fractions altogether by using pseudo-remainders in lieu of the usual remainders.

Example 3.2.12 (WWGCD). Let f and g be as in Example 3.2.11. The PRS with $\beta_i = 1$ and $\alpha_i = \texttt{lcoeff}_x(R_i)^{\delta_i+1}$ where $\delta_i = \texttt{deg}_x(R_{i-1}) - \texttt{deg}_x(R_i)$ is the sequence of pseudo-remainders formed by direct pseudo-division at each step and is given by

$$\begin{aligned}
R_2 &= -15x^4 + 3x^2 - 9, \\
R_3 &= 15795x^2 + 30375x - 59535, \\
R_4 &= 1254542875143750x - 1654608338437500, \\
R_5 &= 12593338779550074310093114199218750 0.
\end{aligned}$$

Although the method employed in the previous example succeeds in obtaining a gcd while working entirely within $\mathbb{Z}[x]$, it fails (utterly) to keep the coefficients a workable size. In fact the coefficient growth is so explosive that this algorithm has aptly been described as the "world's worst" gcd algorithm (WWGCD).

Ideally we would like an algorithm that can control coefficient explosion while avoiding unnecessary gcd calculations. One may recall that a similar problem was (theoretically) resolved in the previous section by using Sylvester's identity. Bareiss used this identity to develop a scheme that did *exact* divisions at each step, simultaneously controlling explosions of intermediate pivots and avoiding gcds. In much the same spirit, Collins [8] and Brown [5] independently developed the subresultant PRS algorithm that accomplishes the same feat.

Definition 3.2.13. The *subresultant PRS* [4] is defined by

$$\alpha_i = r_i^{\delta_i+1}, \quad \beta_1 = (-1)^{\delta_1+1}, \quad \beta_i = -r_{i-1} \cdot \Psi_i^{\delta_i} \quad \text{for} \quad 2 \leqslant i \leqslant k$$

where $r_i = \texttt{lcoeff}_x(R_i)$, $\delta_i = \texttt{deg}_x(R_{i-1}) - \texttt{deg}_x(R_i)$ and Ψ_i is defined by

$$\Psi_1 = -1, \quad \Psi_i = (-r_{i-1})^{\delta_{i-1}} \cdot \Psi_{i-1}^{1-\delta_{i-1}} \quad \text{for} \quad 2 \leqslant i \leqslant k.$$

(Note β_i, r_i and Ψ_i do no involve x).

Example 3.2.14. When applied to the polynomials f and g from Example 3.2.11, the subresultant PRS gives

$$\begin{aligned} R_2 &= 15x^4 - 3x^2 + 9, \\ R_3 &= 65x^2 + 125x - 245, \\ R_4 &= 9326x + 12300, \\ R_5 &= 260708 = \mathrm{Res}(f, g, x). \end{aligned}$$

The subresultant algorithm produces $R_k = \mathrm{Res}(R_0, R_1, x)$, by the update formula

$$R_{i+1} = \frac{\mathtt{prem}(R_{i-1}, R_i, x)}{\beta_i} \qquad (i = 1, \ldots, k-1) \tag{3.13}$$

where β_i is given by Definition 3.2.13. Unfortunately it is not easy to see why the sequence of β_i's work. Superficially, equation (3.13) would impy that R_{i+1} lies in $\mathcal{D}/\mathcal{D}[x]$ ($\mathcal{D}/\mathcal{D}$ is the quotient field of $\mathcal{D}$) rather than in $\mathcal{D}[x]$. Moreover, it is not clear from Definition 3.2.13 that Ψ_i (and therefore β_i) belong to $\mathcal{D}$. In fact, both Collins and Brown noted they were unable to show that Ψ_i was in $\mathcal{D}$ and proceeded in their proofs with $\beta_i, \Psi_i \in \mathcal{D}/\mathcal{D}$. Not until two decades later was a proof given by Ho and Ken Yap [11] who used subresultant chains to show that the Ψ_i's could be regarded as subdeterminants of matrices consisting of the coefficients of R_0 and R_1.

To show that R_i lies in $\mathcal{D}[x]$ is also highly non-trivial. In a follow up to Brown's original paper [4], Brown and Traub [6] developed the Fundamental Theory of Subresultants which reduces the j-th subresultant (the determinant of a submatrix of $\mathrm{Syl}(f, g, x)$) to an explicit product. By proving that each R_i given by the subresultant PRS was a particular j-th subresultant (the 0-th subresultant being the classical one), Collins and Brown simultaneously proved that every $R_i \in \mathcal{D}[x]$ and that the subresultant PRS was correct.

Combining the results of Ho, Yap, Brown and Collins gives $R_{i+1} \in \mathcal{D}[x]$ and $\beta_i \in \mathcal{D}$ for $i = 2, \ldots, k-1$. Since the pseudo-remainder must lie in $\mathcal{D}[x]$ we conclude that the division given by equation (3.13) is exact. That is, the division will generate no remainder or fractions and will produce a quotient in $\mathcal{D}[x]$. We will take this for granted and state (without proof) the following theorem.

Theorem 3.2.15 (Ho, Yap 1996). *For $\mathcal{D}$ a UFD the subresultant PRS produces R_0, R_1,...,$R_k \in \mathcal{D}[x]$, a finite sequence of polynomials, with $\beta_i, \Psi_i \in \mathcal{D}$ for $i = 1 \ldots k-1$, such that $R_k = Res(R_0, R_1, x)$.*

3.2.2 Correctness of the Subresultant Algorithm

We now give a proof of the correctness of Algorithm 11 (the Subresultant algorithm). We show that the variables of this algorithm satisfy the relations of Definition 3.2.13.

Proof of Subresultant Algorithm Correcntess. We show (by induction on i) that the variables of Algorithm 11 satisfy

$$d = \delta_i \qquad \tilde{r} = \beta_i R_{i+1} \qquad u = R_i \tag{3.14}$$

$$v = R_{i+1} \qquad g = \mathtt{lcoeff}_x(R_i) = r_i \qquad h = \Psi_{i+1} \tag{3.15}$$

at the end of iteration i. Collectively these equations serve as our loop invariant.

For the base case let us determine the values of (3.14) and (3.15) at the end of iteration one. Recall that initially, before the loop is entered (what can be considered iteration zero), we have $u = R_0$, $v = R_1$, $g = 1$ and $h = -1$. Noting the difference between $\tilde{r}$ (the pseudo-remainder) and r_i (the leading coefficient of R_i), the first iteration of Algorithm 11 does

$$\begin{aligned}
&d \leftarrow \mathtt{deg}_x(u) - \mathtt{deg}_x(v) = \mathtt{deg}_x(R_0) - \mathtt{deg}_x(R_1) = \delta_1 \\
&\tilde{r} \leftarrow \mathtt{prem}(u, v, x) = \mathtt{prem}(R_0, R_1, x) = \beta_1 R_2 \\
&u \leftarrow v = R_1 \\
&v \leftarrow \tilde{r} \div (-g \cdot h^d) = \beta_1 \cdot R_2 \div \underbrace{((-1) \cdot (-1)^{\delta_1})}_{\beta_1} = R_2 \\
&g \leftarrow \mathtt{lcoeff}_x(u) = \mathtt{lcoeff}_x(R_1) = r_1 \\
&h \leftarrow (-g)^d \div h^{d-1} = (-r_1)^{\delta_1} \div (-1)^{\delta_1 - 1} = (-r_i)^{\delta_1} \cdot \Psi_1^{1-\delta_1} = \Psi_2
\end{aligned}$$

and thereby satisfies the loop invariant.

For the inductive step we will assume that the loop invariant holds at iteration i and infer from this that it holds at iteration $i+1$ as well. At iteration $i+1$ (using the induction

hypothesis) we have

$$d \leftarrow \mathtt{deg}_x(u) - \mathtt{deg}_x(v) = \mathtt{deg}_x(R_i) - \mathtt{deg}_x(R_{i+1}) = \delta_{i+1} \tag{3.16}$$

$$\tilde{r} \leftarrow \mathtt{prem}(u, v, x) = \mathtt{prem}(R_i, R_{i+1}, x) = \beta_{i+1} \cdot R_{i+2} \tag{3.17}$$

$$u \leftarrow v = R_{i+1} \tag{3.18}$$

$$v \leftarrow \tilde{r} \div (-g \cdot h^d) = (\beta_{i+1} \cdot R_{i+2}) \div \underbrace{(-r_i \cdot \Psi_{i+1}^{\delta_{i+1}})}_{\beta_{i+1}} = R_{i+2} \tag{3.19}$$

$$g \leftarrow \mathtt{lcoeff}_x(u) = \mathtt{lcoeff}_x(R_{i+1}) = r_{i+1} \tag{3.20}$$

$$h \leftarrow (-g)^d \div h^{d-1} = (-r_{i+1})^{\delta_{i+1}} \div \Psi_{i+1}^{\delta_{i+1}-1} = (-r_{i+1})^{\delta_{i+1}} \cdot \Psi_{i+1}^{1-\delta_{i+1}} = \Psi_{i+2} \tag{3.21}$$

showing that the loop invariant holds for any arbitrary iteration i. Namely it holds at $i = k-1$ when $v = R_k$ and $\mathtt{deg}_x(v) = 0$. We have from Theorem 3.2.15 that $R_k = \mathrm{Res}(R_0, R_1, x)$ which implies that the subresultant algorithm returns $v = \mathrm{Res}(R_0, R_1, x)$ as desired. □

3.2.3 The Extended Subresultant Algorithm

Given a UFD $\mathcal{D}$ and non-constant polynomial $m \in \mathcal{D}[x]$, we can form the quotient ring $\mathcal{D}[x]/\langle m \rangle$. When m is an irreducible element of $\mathcal{D}[x]$ (that is, there is no non-constant $t \in \mathcal{D}[x]$ such that $t \neq m$ and t divides m), this quotient ring will be a field. Of course, when working in fields it is natural to ask if there is a systematic way of finding inverses. By modifying the the subresultant algorithm we will be able to do this by finding $s, t \in \mathcal{D}[x]$ such that $s \cdot u + t \cdot m = \mathrm{Res}(u, m, x)$. In this case $\mathtt{deg}_x(s) < \mathtt{deg}_x(m)$ and the inverse of $u \in \mathcal{D}[x]/\langle m \rangle$ is $s/\mathrm{Res}(u, m, x)$.

Definition 3.2.16. Let $R_0, \ldots, R_k$ be the PRS given by the subresultant PRS. Let $Q_i = \mathtt{pquo}(R_i, R_{i-1})$ and α_i and β_i be defined as they were in Definition 3.2.13. The following relations give the *extended subresultant PRS* implemented by Algorithm 12.

$$S_0 = 1 \qquad S_1 = 0 \qquad S_{i+1} = (\alpha_i \cdot S_{i-1} - Q_i \cdot S_i)/\beta_i$$
$$T_0 = 0 \qquad T_1 = 1 \qquad T_{i+1} = (\alpha_i \cdot T_{i-1} - Q_i \cdot T_i)/\beta_i.$$

We now give a proof that Algorithm 12 generates s and t with the desired property: $s \cdot u + t \cdot v = \mathrm{Res}(u, v, x)$.

Proof of Correctness of Extended Subresultant Algorithm. We can adapt the proof from the

Algorithm 12 Extended Subresultant Algorithm

Input: The polynomials $u, v \in \mathcal{D}[x]$ where $\mathtt{deg}_x(u) \geq \mathtt{deg}_x(v)$ and $v \neq 0$.
Output: $r = \mathrm{Res}(u, v, x)$ and $s, t \in \mathcal{D}[x]$ satisfying $s \cdot u + t \cdot v = \mathrm{Res}(u, v, x)$.

1: $(g, h) \leftarrow (1, -1)$;
2: $(s_0, s_1, t_0, t_1) \leftarrow (1, 0, 0, 1)$;
3: **while** $\mathtt{deg}_x(v) \neq 0$ **do**
4: $\quad d \leftarrow \mathtt{deg}_x(u) - \mathtt{deg}_x(v)$;
5: $\quad \tilde{r} \leftarrow \mathtt{prem}(u, v, x)$;
6: $\quad \tilde{q} \leftarrow \mathtt{pquo}(u, v, x)$; {$\tilde{r}$ and $\tilde{q}$ are computed simultaneously.}
7: $\quad u \leftarrow v$;
8: $\quad \alpha \leftarrow \mathtt{lcoeff}_x(v)^{d+1}$;
9: $\quad s \leftarrow \alpha \cdot s_0 - s_1 \cdot \tilde{q}$;
10: $\quad t \leftarrow \alpha \cdot t_0 - t_1 \cdot \tilde{q}$;
11: $\quad (s_0, t_0) \leftarrow (s_1, t_1)$;
12: $\quad v \leftarrow \tilde{r} \div (-g \cdot h^d)$;
13: $\quad s_1 \leftarrow s \div (-g \cdot h^d)$
14: $\quad t_1 \leftarrow t \div (-g \cdot h^d)$
15: $\quad g \leftarrow \mathtt{lcoeff}_x(u)$;
16: $\quad h \leftarrow (-g)^d \div h^{d-1}$;
17: **end while**
18: $(r, s, t) \leftarrow (v, s_1, t_1)$;
19: **return** r, s, t;

previous section by adding the equations

$$\tilde{q} = Q_i \qquad s_0 = S_i \qquad s_1 = S_{i+1} \tag{3.22}$$

$$\alpha = \alpha_i \qquad t_0 = T_i \qquad t_1 = T_{i+1} \tag{3.23}$$

to the loop invariant. We will omit the details of this proof so as to avoid repetition, and assume that the equations (3.14), (3.15), (3.22) and (3.23) hold at the end of iteration i for Algorithm 12.

What remains to be shown is that Algorithm 12 produces values s and t such that $s \cdot u + t \cdot v = \text{Res}(u, v, x)$. We prove (by induction on i) that

$$S_i \cdot R_0 + T_i \cdot R_1 = R_i \tag{3.24}$$

for $i \geq 0$ and then use (3.22) and (3.23) to relate this back to the algorithm. Specifically, we will show that the variables s_1 and t_1 satisfy

$$s_1 \cdot R_0 + t_1 \cdot R_1 = \text{Res}(R_0, R_1, x) \tag{3.25}$$

on termination of Algorithm 12.

For the base case recall that $S_0 = T_1 = 1$ and $S_1 = T_0 = 0$ by Definition 3.2.16 which trivially satisfies (3.24) for both $i = 0$ and $i = 1$.

For the induction hypothesis assume that (3.24) is satisfied for $i = n - 1$ and $i = n$. Setting $i = n - 1$ and $i = n$ in (3.24) gives the relations

$$S_{n-1} \cdot R_0 + T_{n-1} \cdot R_1 = R_{n-1} \quad \text{and} \quad S_n \cdot R_0 + T_n \cdot R_1 = R_n.$$

To conclude the induction we see that setting $i = n + 1$ in the left hand side of (3.24) and using Definition 3.2.16 gives,

$$\begin{aligned}
S_{n+1} \cdot R_0 + T_{n+1} \cdot R_1 &= \frac{\alpha_n \cdot S_{n-1} - Q_n \cdot S_n}{\beta_n} \cdot R_0 + \frac{\alpha_n \cdot T_{n-1} - Q_n \cdot T_n}{\beta_n} \cdot R_1 \\
&= \frac{\alpha_n \cdot (S_{n-1} \cdot R_0 + T_{n-1} \cdot R_1) - Q_n \cdot (S_n \cdot R_0 + T_n \cdot R_1)}{\beta_n} \\
&= \frac{\alpha_n \cdot R_{n-1} - Q_n \cdot R_n}{\beta_n} \text{ by induction.} \\
&= \frac{\beta_n \cdot R_{n+1}}{\beta_n} \text{ by Definition 3.2.10.} \\
&= R_{n+1}
\end{aligned}$$

showing that (3.24) is held for arbitrary i. Namely (3.24) will hold at $i = k - 1$ when $R_k = \text{Res}(R_0, R_1, x)$ as guaranteed by Theorem 3.2.15.

Now, using the equations of (3.22) and (3.23) we have that at the end of iteration $k - 1$ of Algorithm 12 that the variables s_1 and t_1 satisfy

$$s_1 \cdot R_0 + t_1 \cdot R_1 = S_k \cdot R_0 + T_k \cdot R_1 = R_k = \text{Res}(R_0, R_1, x).$$

Thus Algorithm 12 returns s and t with the desired property. □

Chapter 4

Implementation

Here we present the main ideas behind the implementation (in C) of the algorithms discussed in this thesis. We will not give any complete listings since they are long and optimized to a point where they do not much resemble the pseudo-code. Instead we will discuss the data structures and key ideas necessary to carry out the implementation. We will also show how the compiled code behaves in the Maple environment. Since speed and space are primary concerns, this implementation has not been generalized to handle polynomials with coefficients from an arbitrary ring. We will only allow coefficients from the set of integers given by the C data type `int` (the so-called 'machine integers'). It would be a straightforward modification to use arbitrary integers as coefficients, but this would require the use of the GMP library for arbitrary precision arithmetic.

4.1 Packed Monomial Represenation

There are many ways to represent polynomials in a computer language [17]. We can classify a representation as being recursive or distributed, sorted or not sorted, sparse or dense, variables in or variables out, which already gives us at least sixteen representations. Since the timings given in Section 4.4 are compared with Maple, and also to contrast our representation against something, let us discuss Maple's representation.

At its core Maple can represent sums, products, and names. In this environment a polynomial has the natural representation as a sum of products (product actually refers to exponentiation). Figure 4.1 demonstrates the maple representation of a simple polynomial.

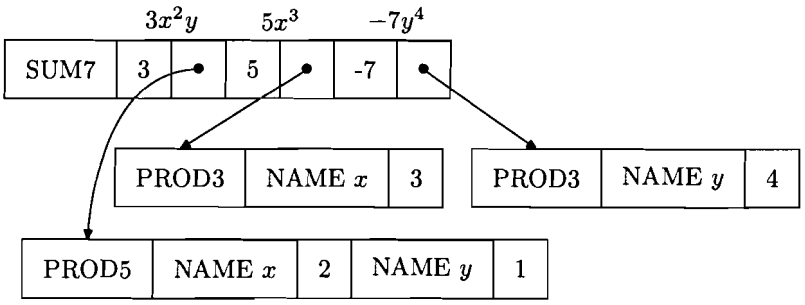


Figure 4.1: $3x^2y + 5x^3 - 7y^4$ as represented by Maple.

This representation is distributed (a recursive representation regards every polynomial being in a single variable, where coefficients may be polynomials as well), not sorted (the terms are in no particular $\succ$-order), sparse (0 terms are not represented), and the variables are in the structure. The motivation for creating polynomials like this was to satisfy the initial Maple ambition of doing calculus. Being able to quickly identify something as a sum or product makes taking derivatives very natural.

However, this representation is not very well suited for doing polynomial arithmetic on a large scale. For one it is not compact, each term requires the creation of a product structure, and these structures are strewn across memory which is generally undesirable. Additionally, the terms are not given in $\succ$-order and are therefore not in standard form, a requisite for each one of our algorithms.

But the most pronounced deficiency with this representation is the problem of monomial comparison and multiplication. As we have seen our algorithms are governed mainly by the speed at which we can do these operations. With Maple's representation, in order to compare or multiply monomials we must first locate the name (variable) in the product structure and then identify its exponent. Repeating this task for each variable, since the variables can be given in any order, requires at least linear time to accomplish and does a linear amount of integer arithmetic (linear time in the number of variables.)

The packed structure given and analyzed by Monagan and Pearce [14] is based on Bachmann and Schönemann's scheme [1], which overcomes these problems. Instead of using a special product structure, Bachmann and Schöenmann's scheme (used by Singular) stores

a total degree and a vector of exponents. For example the monomial $x^2y^3z^5$, with the ordering $x \succ y \succ z$, would be represented by $< 10, 2, 3, 5 >$ (in general $\boldsymbol{x}^{\boldsymbol{\alpha}}$ is represented by $\langle|\boldsymbol{\alpha}|, \alpha_1, \ldots, \alpha_n\rangle$). Now we get constant time access to the exponents since the vector is ordered.

Monagan and Pearce also discussed how to reduce the amount of arithmetic needed to test $\succ$-order to one machine instruction. To understand why this is a huge improvement recall that $\boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{lex} \boldsymbol{x}^{\boldsymbol{\beta}}$ when $(\boldsymbol{\alpha} - \boldsymbol{\beta})$'s leftmost non-zero entry is positive. In order to test this we search for i such that $\boldsymbol{\alpha}_i - \boldsymbol{\beta}_i < 0$. In the case where $\boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{lex} \boldsymbol{x}^{\boldsymbol{\beta}}$ we will never find such an i and end up checking every difference. This means we usually do a linear (in the number of variables) amount of arithmetic to test $\succ$-order.

The trick is not to represent the exponent vector $< 10, 2, 3, 5 >$ as a list, but rather as a single integer so that monomial comparisons and multiplications become single integer comparisons and additions. To accomplish this consider the (now) typical 64-bit word.[1] The register, literally a sequence of 64 switches, is where the processor stores numbers. A 64-bit register can represent numbers in the range of 0 to 18446744073709551615 or $2^{64} - 1$. When numbers are in this range, computations involving them can be done directly in the processor. These computations are called machine instructions and are the fastest computations a processor can do.

To demonstrate how we pack a monomial into an integer let us pack a specific monomial first, then generalize after. We can place the exponent vector $< 10, 2, 3, 5 >$ into a 32-bit register by doing the following: partition the register into 4 equal pieces, give each number in the exponent vector a $\frac{32}{4} = 8$ bit representation, and then concatenate these binary numbers into one large bit sequence, the result is a 32-bit integer. This process is depicted in Figure 4.2.

The generalization of this process is not hard to see. All that is needed to pack more variables is more partitions of the register. In the case where this division is not exact we give all extra bits to the first partition that represents the total degree. This is because the total degree will always be the largest number being packed.

The clear drawback of this representation is that not all monomials can be packed. We are unable to represent monomials with exponents so large that they can not fit into a partition. To exacerbate this problem these partitions shrink as more variables are introduced.

[1] Although 64-bit processors didn't become common in personal computers till 2003 they have existed in supercomputers since the 1960s.

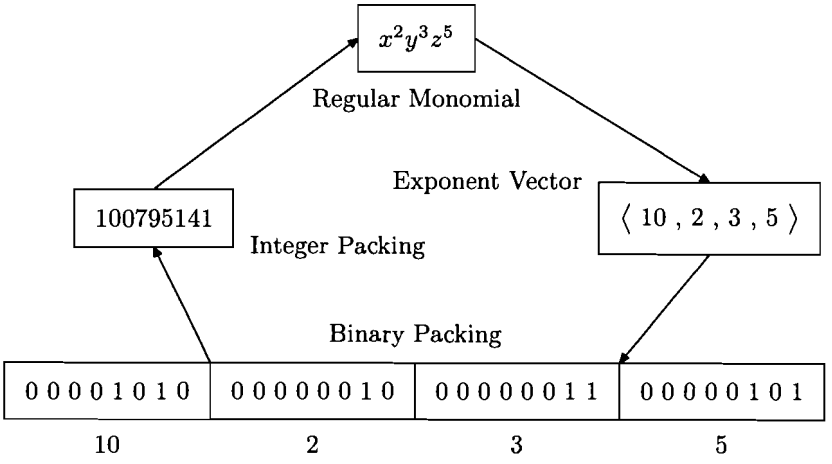


Figure 4.2: The packing of the term $x^2y^3z^5$ is demonstrated. The exponent vector is packed by converting its elements into bit strings and concatenating the result. The whole box in the Binary Packing represents a 32-bit register and the subdivisions are the 8-bit partitions. The integer that represents this 32-bit long sequence is 100795141.

Fortunately, in practice (especially when doing gcd's and Gröbner basis calculations), it is rarely the case that monomials have exponents that are very large. Table 4.1 lists the maximum degrees for a given number of variables for both 64-bit and 32-bit processors.

The packing we have just described is best suited for use with the graded lexicographic ordering (for just lexical ordering we could simply leave out the total degree and pack in the exact same way).

Theorem 4.1.1. *Suppose $\boldsymbol{x}^{\boldsymbol{\alpha}}$ and $\boldsymbol{x}^{\boldsymbol{\beta}}$ pack to the integers A and B respectively and let b be the number of bits allocated for a given partition, then:*

1. $\boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{grlex} \boldsymbol{x}^{\boldsymbol{\beta}}$ *when* $A > B$.
2. $\boldsymbol{x}^{\boldsymbol{\alpha}} \times \boldsymbol{x}^{\boldsymbol{\beta}}$ *packs to* $A + B$ *when* $\boldsymbol{deg}(\boldsymbol{x}^{\boldsymbol{\alpha}}) + \boldsymbol{deg}(\boldsymbol{x}^{\boldsymbol{\beta}}) < 2^b$.
3. $\boldsymbol{x}^{\boldsymbol{\alpha}} \div \boldsymbol{x}^{\boldsymbol{\beta}}$ *packs to* $A - B$ *when* $\boldsymbol{x}^{\boldsymbol{\beta}} | \boldsymbol{x}^{\boldsymbol{\alpha}}$.

Proof. Let $\boldsymbol{x}^{\boldsymbol{\alpha}}$ and $\boldsymbol{x}^{\boldsymbol{\beta}}$ be monomials that correspond to the exponent vectors $\langle |\boldsymbol{\alpha}|, \alpha_1, \ldots, \alpha_n \rangle$ and $\langle |\boldsymbol{\beta}|, \beta_1, \ldots, \beta_n \rangle$. Suppose when we pack these vectors we get the bit sequences $A = a_0 \circ a_1 \circ \ldots \circ a_n$ and $B = b_0 \circ b_1 \circ \ldots \circ b_n$, where $\circ$ denotes concatenation and a_i is the binary representation of α_i when $i > 0$ and $|\boldsymbol{\alpha}|$ when $i = 0$ (and similarly so for B with β).

Table 4.1: Degree limitations when packing monomials in a 64-bit or 32-bit register. The heading #bits refers to the number of bits alloted for packing a single exponent.

	64-bit		32-bit	
#variables	#bits	max deg	#bits	max deg
2	21	2,097,151	10	1023
3	16	65,535	8	255
4	12	2047	6	6
5	10	1023	5	31
6	9	511	4	15
7	8	255	4	15
8	7	127	3	7
9	6	63	3	7
10	5	31	2	3
11	5	31	2	3
15	4	15	2	3
21	3	7	1	1
31	2	3	1	1

1. Suppose $A > B$ then we have $a_0 \circ \ldots \circ a_n > b_0 \circ \ldots \circ b_n$ which means there is an $i \geqslant 0$ such that $a_i > b_i$ where $a_j = b_j$ for $0 \leqslant j < i$. If $i = 0$ then we have $\boldsymbol{\alpha}_0 > \boldsymbol{\beta}_0 \Rightarrow \boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{grlex} \boldsymbol{x}^{\boldsymbol{\beta}}$ by Definition 1.2.8. If $i \neq 0$ then the integer representation for a_i and b_i satisfy $\boldsymbol{\alpha}_i > \boldsymbol{\beta}_i$ when $i > 0$ where $\boldsymbol{\alpha}_j = \boldsymbol{\beta}_j$ for $0 < j < i$. This also means that $\boldsymbol{x}^{\boldsymbol{\alpha}} \geqslant_{grlex} \boldsymbol{x}^{\boldsymbol{\beta}}$ by Definition 1.2.8.

2. The exponent vector for $\boldsymbol{x}^{\boldsymbol{\alpha}} \times \boldsymbol{x}^{\boldsymbol{\beta}}$ is $\langle |\boldsymbol{\alpha} + \boldsymbol{\beta}|, \alpha_1 + \beta_1, \ldots, \alpha_n + \beta_n \rangle$. We have that $|\boldsymbol{\alpha}| + |\boldsymbol{\beta}| > (\alpha_i + \beta_i)$ for $i > 0$. Therefore when $|\boldsymbol{\alpha} + \boldsymbol{\beta}| < 2^b$ it is the case that $(\alpha_i + \beta_i) < 2^b$ for $i > 0$. In other words, when the sum of the total degrees are small enough to fit into there partition, so must the individual sums of exponents fit as well. Therefore the bit sequence $(a_0 + b_0) \circ (a_1 + b_1) \circ \ldots \circ (a_n + b_n)$ will represent the exponent vector for $\boldsymbol{x}^{\boldsymbol{\alpha}} \times \boldsymbol{x}^{\boldsymbol{\beta}}$, which is the binary number representing $A + B$.

3. When $\boldsymbol{x}^{\boldsymbol{\beta}} | \boldsymbol{x}^{\boldsymbol{\alpha}}$ the exponent vector for $\boldsymbol{x}^{\boldsymbol{\alpha}} \div \boldsymbol{x}^{\boldsymbol{\beta}}$ is given by $\langle |\boldsymbol{\alpha} - \boldsymbol{\beta}|, \alpha_1 - \beta_1, \ldots, \alpha_n - \beta_n \rangle$. Since there is no i where $\alpha_i - \beta_i < 0$ each $a_i - b_i$ yields a positve binary number that must fit in its partition. The resulting bit sequence would be $(a_0 - b_0) \circ (a_1 - b_1) \circ \ldots \circ (a_n - b_n)$ represented by the integer $A - B$.

□

Now, in order to check $\boldsymbol{x}^{\alpha} \geqslant_{grlex} \boldsymbol{x}^{\beta}$ or calculate $\boldsymbol{x}^{\alpha} \times \boldsymbol{x}^{\beta}$, all we must do is compare the monomial's integer representations with $>$ (regular greater than) or add the representations. Either of which can be done in one machine instruction. Reducing these tasks, that were once potentially linear, to one machine instruction is a huge improvement indeed.

To represent a polynomial using the packed representation is easy. A polynomial will be two arrays, one integer array for the monomials, and another array (of any data type) for the coefficients. This representation is depicted in Figure 4.3. We order the arrays so that the monomial at position i in the monomial array has its coefficient at position i in the coefficient array. If we order the monomial array by $>$ (regular greater than), and reorder the coefficient array accordingly, then the terms of the polynomial will be in $\succ$-descending order, as required by our algorithms.

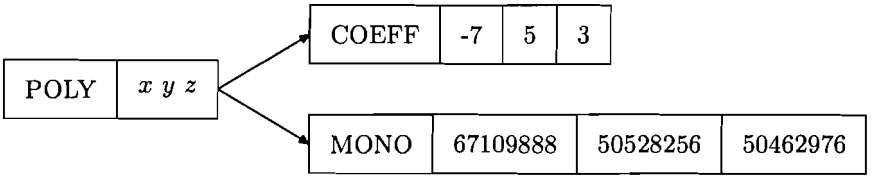


Figure 4.3: $3x^2y + 5x^3 - 7y^4$ in the packed representation.

4.2 Data Types

We now give the unabridged data structures used in the C-implementation of this library. The first structure we introduce is that of a single term. A term consists of an integer that represents a monomial and (in this case) an integer coefficient. The modifier `unsigned` allows us to use the leading bit of the register that would otherwise be used to indicate the sign of the integer. In creating this structure we are deviating slightly from the packed representation given in the previous section, but not in any meaningful way.

Listing 4.1: The term structure.

```
1 struct term {
2     unsigned int mono;      //monomials
3     int coeff;              //coefficients
4 };
5 typedef struct term TermType;
```

The second, and most important structure, allows us to declare a delayed polynomial in the C-language. The creation of this structure was paramount to the development of the library and its relevance should not be understated. The crucial observation, that prompted lines (7)-(10) of the listing below, is that a polynomial can be regarded as the application of some method (addition, multiplication, division) to two other polynomials. In this respect H should not be regarded as a simple list of terms, but rather a pointer to a method that can calculate an arbitrary term.

Listing 4.2: The delayed polynomial structure.

```
1  struct poly {
2      int N;                    //number of forced terms
3      TermType *terms;          //forced terms, the last of which must be zero
4      char *var;                //variables
5      int NumVars;              //number of variables
6      //This poly = Method(F,G) where method is amoung ADD,MULT,DIVIDE//
7      struct poly *F1;
8      struct poly *F2;
9
10     TermType (*Method)(int n, struct poly *F, struct poly *G, struct poly *H);
11     int state[6];             //local variables for Method
12     /*Special state for a Division
13     state[0] = Sleep
14     state[1] = ActiveG
15     state[2] = P (a machine prime)
16     state[3] = 0 (quotient) / 1 (remainder)
17
18     if state[3]=0 then
19         state[4] = address of remainder
20     else
21         state[4] = address or quotient
22     end if*/
23
24     /*members of the Heap look like [ mono[f(i)*g(j)] , i, j ] */
25     HeapType *Heap;           //local heap for Method
26 };
27
28 typedef struct poly PolyType;
```

The basic methods are addition, multiplication, and subtraction (division is a slight variant). Based on the delayed algorithms given in Chapter 2 these methods are designed to "pick up where another calculation left off". They return the n-th term from the application of a method to F and G. H stores the result and also maintains information of how the last computation ended.

Listing 4.3: Methods.

```
1 TermType h_ADD (int n, PolyType *F, PolyType *G, PolyType *H);
2 TermType h_SUB (int n, PolyType *F, PolyType *G, PolyType *H);
3 TermType h_MULT (int n, PolyType *F, PolyType *G, PolyType *H);
```

The procedure Term produces the n-th term of the delayed polynomial F, calculating it if necessary. This procedure allows us to follow the pseudo-code given in Chapter 2 more directly as $\text{Term}(i, F) = F_i$.

Listing 4.4: Term.

```
1 TermType Term (int n, PolyType *F) {
2     if (n>F->N) {
3         return F->Method(n,F->F1,F->F2,F);
4     }
5     return F->terms[n];
6 };
```

The final loose end we must tie up is how to represent polynomials that are not delayed. That is, how do we define the initial few polynomials we start from, or what happens when a polynomial has no more delayed terms. For this we have the special Immediate method. An Immediate polynomial is one with no delayed terms. Once a polynomial has forced all of its terms its method is changed to Immediate.

Listing 4.5: Immediate.

```
1 TermType Immediate(int n, struct poly *F, struct poly *G, struct poly *H) {
2     if (n>F->N) {
3         return F->terms[F->N+1];          //terms[F->N+1] = 0
4     };
5     return F->terms[n];
6 };
```

4.3 Sample Session

To use this library in a more meaningful way we have imported it into Maple. In order to do this we compiled the C-code into a shared library and built a custom wrapper. A wrapper allows external code to be interfaced with Maple to be used in a session. For simple procedures Maple can automatically generate these wrappers, but when more complex data structures are required (which is most of the time) these wrappers have to be coded by hand (hence the name custom wrapper), which is a non-trivial task.

We begin by reading the file `DelayedPoly.mpl` which contains all the procedures (some externally linked) to work with delayed polynomials. Our examples will be in $\mathbb{Z}[x, y, z]$ and our monomial ordering is set to graded lexical. The first command invoked is `MakePoly` which converts a Maple represented polynomial to an Immediate delayed polynomial.

```
> read "DelayedPoly.mpl":
> var:=[x,y,z];
                              var := [x, y, z]
> f:=randpoly(var,coeffs=rand(1..1));
                              2    3      2  3      3        4
             f := x y + x y z  + y  z + x  z  + x y  z + y z
> g:=randpoly(var,coeffs=rand(1..1));
                        2      3      3    5    4      2    2
                 g := x  + x y  + y z  + x  + x  z + x  y z
> F:=MakePoly(f,var);
                           F := "External/0xe6d60"
> G:=MakePoly(g,var);
                           G := "External/0x3a030"
```

`External/` indicates the address in memory where F can be found. It is of no relevance to the Maple session but is useful for debugging.

We can now perform our first operation in the lazy environment. We will see that all that is received by operating on two delayed polynomials is yet another address in memory (of another delayed polynomial). Unlike the F and G, which are Immediate polynomials, H is fully delayed and is pointing to a method to calculate its n-th term.

```
> H:=ADD(F,G);
```

```
H := "External/0x20f298"
```

To access single terms of H is easily done with the `Term` procedure. However this procedure is doing some additional work converting the packed representation back to Maple form. As this is the case there is an additional command `RawTerm` which returns the term in packed representation doing no unnecessary work.

```
> RawTerm(3,H);
                              1, 84017410
> Term(3,H,var);
                                2    2
                               x  y z
> Term(100,H,var);
                                  0
```

It is often useful to calculate H^{∞} and display its terms. This is done by forcing terms until zero is reached, indicating that all terms have been calculated. For this we have two procedures `flook` (force look), which forces and prints every raw term, and `alook` (algebraic look) which forces and converts every term back to Maple, summing the results to form a Maple polynomial.

```
> flook(H);
[ [1,84213760],[1,84148225],[1,84017410],[1,84017155],[1,83952385],
[1,83886340],[1,67175168],[1,67174658],[1,67109633],[1,67109123],
[1,33685504],[1,33620224],[0,0] ]

> alook(H,var);
 5    4      2    2    2 3      3        4      3        2    3
x  + x  z + x  y z  + x  z  + x y  z + y z  + x y  + x y z  + y  z

          3    2
     + y z  + x  + x y
> expand( alook(H,var) - (f+g) );
                                  0
```

It should be noted that Maple's polynomial representation has no implied order for terms. That is, the terms of the polynomial given by `alook` are in no particular order, and are

in no way representing the actual order which is intact for the list of packed monomials given by `flook`. Despite this, `alook` gives us a convenient means of checking correctness as illustrated above.

Multiplication and subtraction work as one would expect, so we will not give any exposition of their behavior. Since division is sufficiently different we give a sample session below. In this case, forcing terms of the quotient Q may force terms of the remainder R. This prompted the creation of the procedure `ForcedTerms` which returns the number of forced terms of a delayed polynomial.

```
> var:=[x,y,z]:
> f:=randpoly(var)*randpoly(var);
                        2       2  3         3           4
f := (87 x y - 56 x y z  - 62 x  z  + 97 x y  z - 73 y z )

          3       2  2       3           3       4         3
    (-82 y  + 80 x  z  - 44 y  z + 71 y z  - 17 x  y - 75 x  y z)

> g:=randpoly(var);
               2         2       2  2           2         3           4
      g := 6 y  + 74 x y  + 72 x  z  + 37 x y z  - 23 x y  z + 87 x z

> F:=MakePoly(f,var):
> G:=MakePoly(g,var):
> Q,R:=DIVIDE(F,G,101):
> Term(2,Q,var);
                                      3
                                -35 x  y z
> ForcedTerms(R);
                                      2
> Term(3,R,var);
                                   2  4  4
                                29 y  z  x
> ForcedTerms(Q);
                                      2
```

```
> Expand(f-g*alook(Q,var)-alook(R,var)) mod 101;
                                    0
```

4.4 Timings

To conclude this chapter we time the delayed multiply and divide routines using some moderately large examples. To be more precise first we take f and g in $\mathbb{Z}_{503}[x, y, z]$ to ensure that the coefficients in $f \times g$ don't fall outside the range of a 32-bit processor ($[-2^{-31}, \ldots, 2^{31}]$). That is, we use `Expand() mod 503` when defining f and g. Then we use the Maple `time()` command to measure the runtime of `force(H)` and `force(Q)` when `H:=MULT(F,G)` and `Q,R:=DIVIDE(H,F,503)`.

In order to compare our times we also provide timings for Maple's `expand(f*g)` and `Divide(h,f) mod 503` where `h:=Expand(f*g) mod 503`. It should be noted that the disparity in the timings is not an indication that the delayed routines are fundamentally better. The improvements are largely the result of computing in a compiled C-library (as oppose to the Maple kernel) and using a packed monomial representation. We only provide the Maple times to show that computing in a delayed fashion does not result in an overall slowdown that makes it worse than the usual methods.

			$f \times g$		$(fg) \div f$	
	$\#f, \#g$	$\#(fg)$	DelayedPoly	Maple 11	DelayedPoly	Maple 11
$f = (1+x+y+z)^{25}$ $g = f + 1$	3276	23426	4.98s	12.34s	4.94s	18.30s
$f = (1+x+y^2+z^3)^{20}$ $g = (1+z+y^2+x^3)^{20}$	1771	78960	1.36s	6.26s	1.45s	12.55s
$f = (1+x+y^3+z^5)^{20}$ $g = (1+z+y^3+x^5)^{20}$	1771	180585	1.40s	8.19s	1.45s	12.55s

Table 4.2: The first, second, and third rows correspond to a dense, somewhat sparse, and very sparse example (respectively). The notation $\#f$ indicates the number of terms in the polynomial f as is the same of $\#g$ and $\#(fg)$. The time is given in seconds.

Bibliography

[1] Olaf Bachman and Hans Schönemann. Monomial representations for Gröbner bases computations. In *Proceedings of ISSAC*, pages 309–316. ACM Press, 1998.

[2] E. F. Bareiss. Sylvester's identity and multisptep integer-preserving Gaussian elimination. *J. Math. Comp.*, 103:565–578, 1968.

[3] E. F. Bareiss. Computational solutions of matrix problems over an integral domain. *J. Inst. Maths. Appl.*, 10:68–104, 1972.

[4] W. S. Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the Association for Computing Machinery*, 18(4):478–504, October 1971.

[5] W. S. Brown. The subresultant PRS algorithm. *ACM Transactions on Mathematical Software*, 4(3):273–249, September 1978.

[6] W. S. Brown and J. F. Traub. On Euclid's algorithm and the theory of subresultants. *Journal of the Associtation for Computain Machinery*, 18(4):505–514, October 1971.

[7] B. Buchberger, G. E. Collins, and R. Loos with R. Albrecht, editors. *Computer Algebra Symbolic and Algebraic Computation.* Springer-Verlag, 1982.

[8] G. E. Collins. Subresultants and reduced polynomial remainder sequences. *J. ACM*, 14(1):128–142, 1967.

[9] David Cox, John Little, and Donald O'Shea. *Ideals, Varieties, and Algorithms.* Springer, third edition, 2007.

[10] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992.

[11] Chung-Hen Ho and Chee Keng Yap. The habicht approach to subresultants. *Journal of Symbolic Computation*, 21:1–14, 1996.

[12] Stephen C. Johnson. Sparse polynomial arithemetic. *ACM SIGSAM Bulletin*, 8(3):63–71, 1974.

[13] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Publishing Company, 1981.

[14] Michael Monagan and Roman Pearce. *Polynomial Division using Dynamic Arrays, Heaps, and Packed Exponent Vectors*, volume 4770 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[15] Michael Monagan and Roman Pearce. Sparse polynomial arithmetic using a heap. *Journal of Symbolic Computation - Special Issue on Milestones In Computer Algebra*, 2008. Submitted.

[16] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[17] David Stoutemyer. Which polynomial representation is best? In *Proceedings of the 1984 Macsyma Users Conference*, pages 221–244, Schenectady, N.Y., 1984.

[18] Akira Terui. Recursive polynomial remainder sequence and its subresultants. Preprint, January 2008.

[19] S. M. Watt. A fixed point method for power series computation. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, volume 358 of *Lecture Notes in Computer Science*. Spinger Verlag, July 1989.

[20] Chee-Keng Yep. Linear systems. Lecture Notes, September 1995.