Lazy and Forgetful Polynomial Arithmetic and Applications

Paul Vrbik *
University of Western Ontario
Department of Computer Science
London, ON Canada
pvrbik@csd.uwo.ca

Michael Monagan
Simon Fraser University
Department of Mathematics
Burnaby, B.C. Canada
mmonagan@cecm.sfu.ca

ABSTRACT

We present lazy and forgetful algorithms for adding, multiplying and dividing multivariate polynomials. The lazy property allows us to compute the i-th term of a polynomial without doing the work required to compute all the terms. The forgetful property allows us to forget earlier terms that have been computed to save space.

For example, given polynomials A, B, C, D, E we can compute the exact quotient $Q = \frac{A \times B - C \times D}{E}$ without explicitly computing the numerator $A \times B - C \times D$ which can be much larger than any of A, B, C, D, E and Q.

As applications we apply our lazy and forgetful algorithms to reduce the maximum space needed by the Bareiss fraction-free algorithm for computing the determinant of a matrix of polynomials and the extended Subresultant algorithm for computing the inverse of an element in a polynomial quotient ring.

1. INTRODUCTION

Let D be an integral domain and $R = D[x_1, x_2, ..., x_n]$ be a polynomial ring. Let $f = f_1 + f_2 + ... + f_n$ be a polynomial in R where each term f_i of f is of the form $f_i = a_i X_i$ where $a_i \in D$ and X_i is a monomial in $x_1, ..., x_n$. Two terms $a_i X_i$, $a_j X_j$ are *like terms* if $X_i = X_j$. We say f is in *standard form* if $a_i \neq 0$ and $X_1 \succ X_2 \succ \ldots \succ X_n$ in a monomial ordering $\succeq$. This form is often called the *sparse distributed form* for polynomials in R. In what follows we use $\#f$ to indicate the number of terms of f for readability.

Let f, g be polynomials in R in standard form. In this paper we present lazy and forgetful algorithms for computing the i-th term of $f + g$, $f \times g$ and $f \div g$ in a manner where calculations are done only when necessary. For example, if asked to compute the first two terms of the product $f \times g$ our algorithms will compute $f_1 g_1$, $f_2 g_1$ and $f_1 g_2$ only. Lazy algorithms were first introduced into computer algebra systems by Burge and Watt [1] where they were used in Scratchpad II for power series arithmetic. The idea was that if one had computed the n-th term of a power series, but needed another term, one should not have to redo the entire computation to get it. Lazy power series solved this. But not all of the lazy power-series algorithms were efficient. For example, the most obvious algorithm for computing $exp(f(x))$ to $O(x^n)$ requires $O(n^3)$ arithmetic operations whereas the lazy algorithm required $O(n^4)$ many. Watt showed how to reduce this to $O(n^2)$ [11].

van der Hoven considers lazy algorithms for multiplication of power series to $O(x^n)$ which are asymptotically fast [10]. A lazy analogue of Karatsuba's divide and conquer algorithm is given which does $O(n^{\log_2 3})$ arithmetic operations (the same as the as non-lazy algorithm) but uses $O(n \log n)$ space, an increase of a factor of $\log n$. van der Hoven also gives a lazy multiplication based on the FFT which does $O(n \log^2 n)$ arithmetic operations, a factor of log n more than the non-lazy multiplication. However, all af these results assume *dense* power series and our interest is the sparse case.

Our lazy algorithm for polynomial multiplication is a variant of Johnson's heap method [6] and our lazy division algorithm is based on the heap division algorithm of Monagan and Pearce [8]. To illustrate these variations let us develop an algorithm for multiplication.

A naive algorithm for multiplying $f = f_1 + ... + f_n$ by $g = g_1 + ... + g_m$ computes

$$f \times g = ((f \times g_1 + f \times g_2) + f \times g_3) + \ldots + f \times g_m$$

where additions are done using a simple merge. For dense polynomials this method does $O(\#f \#g)$ monomial comparisons but for sparse polynomials it may do $O(\#f \#g^2)$.

To reduce the number of monomial comparisons one could instead compute a list of terms in the product

$$L = [f_1 g_1, \ldots, f_n g_1, f_1 g_2, \ldots, f_n g_2, \ldots, f_1 g_m, \ldots, f_n g_m],$$

sort L using $O(\#f \#g \log \#f \#g)$ monomial comparisons, then add terms with like monomials. But this method requires space for storing $\#f \#g$ terms which is poor for dense polynomials where the product $h = \#f \#g$ may have only $O(\#f + \#g)$ terms.

The sorting problem can be reduced to doing a simultaneous

*We gratefully acknowledge the support of the MITACS NCE of Canada.

m-ary merge on the set of *sorted* sequences

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

Johnson's contribution was to use a heap H, initialized to contain the terms $f_1g_1, f_1g_2, \ldots, f_1g_m$, to merge the m sequences. Since the number of terms in the heap never exceeds $\#g$, inserting into and extracting terms from H costs $O(\log \#g)$ monomial comparisons per insertion/extraction. Since all $\#f\#g$ terms are eventually inserted and extracted from the heap, the algorithm does a total of $O(\#f\#g \log \#g)$ monomial comparisons.

A lazy algorithm should use as few terms of f and g as possible. But Johnson's method uses every term of g (to initialize the heap H). We give an optimization that avoids this.

Claim 1. Let $S[j] = (f_1g_j, \ldots, f_ng_j)$. If f_1g_j is in the heap H, then no term of the sequences $S[j+1], \ldots, S[m]$ can be the $\succ$-largest term of H.

PROOF. By the definition of a monomial ordering we have: if $g_j \succ g_{j+1} \succ \ldots \succ g_m$, then $f_1g_j \succ f_1g_{j+1} \succ \ldots \succ f_1g_m$. As $f_1g_{j+1}, \ldots, f_1g_m$ are (respectively) the $\succeq$-largest terms of $S[j+1], \ldots, S[m]$, it follows that f_1g_j is $\succeq$-larger than any term of $S[j+1], \ldots, S[m]$. The claim is an immediate consequence of this. □

Using this claim we can ensure that no unnecessary terms are put in the heap. 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 ensures that these terms could not be the $\succeq$-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. To merge elements of S using heaps and a replacement scheme we do the following:

1. Create a heap $H = [f_1g_1]$ and an empty sequence F.
2. Extract the $\succeq$-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 $i < \#f$ then insert $f_{i+1}g_j$ (the $\succeq$-next largest term of $S[j]$) into the heap.
5. If $i = 1$ and $j < \#g$ then add f_1g_{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.

This heap algorithm for doing polynomial multiplication can be extended to polynomial division. Recall that when we do $f \div g$ we are trying to construct the quotient q and remainder r such that $f - qg - r = 0$. We use a heap to store the sum $f - qg$ by merging the set of $\#q + 1$ sequences

$$\{(f_1, \ldots, f_n), (-q_1g_1, \ldots, -q_kg_1), \ldots, (-q_1g_m, \ldots, -q_kg_m)\}.$$

Alternatively we may see the heap as storing the sum

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

where $\#g = m$, $\#q = k$ and the terms q_i may be unknown.

As with multiplication, we replace a term coming out of the heap with the $\succeq$-next largest term in the sequence it was taken from. That is, we replace f_i with f_{i+1} and $-q_ig_j$ with $-q_{i+1}g_j$ (we also use the optimization that says only add $-q_1g_{j+1}$ after removing $-q_1g_j$). 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 $-q_ig_j$. But, since $-q_ig_j$ can certainly not be required to calculate q_i, the terms needed to determine q_i must already be in the heap. 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 lazy division algorithm, this is referred to as 'sleeping'.

2. LAZY ARITHMETIC

The intended purpose of working in a lazy way is to improve performance by avoiding unnecessary calculations. To apply this to polynomial arithmetic we restrict access to a polynomial to that of a single term. For instance, to calculate the n-th term of a sum of two polynomials would *not* require the calculation of the $(n+1)$-st term. Furthermore, if we saved intermediate results from this calculation, then the i-th term where $i \leq n$ could be 'calculated' instantaneously.

Definition 1. A *lazy polynomial*, F, is an approximation of the polynomial $f = f_1 + \ldots + f_n$ (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 lazy polynomial F by $|F|$ (and to be consistent let $\#F = |F^\infty| = \#f$). Note that it is always the case that $F_i = f_i$ for all i and that F^{N+1} is a better approximation of f than F^N when $N < n$.

A lazy polynomial must satisfy two conditions regarding computation: all the forced terms of F are cached for reaccess and calculating a delayed term of F uses as few terms of g and h as possible.

From now on it will be necessary to distinguish regular polynomials from those that are lazy. 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 1. The polynomial $f = f_1 + \ldots + f_n$ 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 1. To resolve this we append an infinite amount of zeros to the end of f so that $f = f_1 + \ldots + f_n + 0 + 0 + \ldots$ (now $F^3 = x+y+0$). 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.

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. First note that naive algorithms prematurely access terms. To best illustrate this, recall the first step of naive multiplication, where we merge $f_1 \times g$ with $f_2 \times g$. To accomplish this, it would be necessary to use *every* term of g. Division is similar. When updating f by $f - q_i \times g$, all terms of g are needed.

We use the heap methods for division and multiplication and a simple merge for addition. Since these methods build the result in $\succeq$-order anyways, we simply halt and return once n non-zero terms are generated. But we also require no work to be repeated to calculate $X_{N-1}, \ldots, X_1$ after calculating X_N. To achieve this we pass our algorithms the approximation X^N which must also remember the state of the algorithm it was passed to. Specifically, it must remember the heap the calculation was using and local variables that would otherwise get erased (we will assume that this information is associated with X^N in *some* way and can be retrieved and updated).

Lazy algorithms for doing addition, multiplication and division are now presented. Note that the algorithm for division returns terms of the quotient (while updating the remainder), but could easily be modified to instead return terms of the remainder (while updating the quotient). Complexity results for multiplication and division follow their respective algorithms.

Algorithm 1 - Lazy 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: **if** $F_i + G_j \neq 0$ **then**
15: $X_k \leftarrow F_i + G_j$;
16: $(i, j, k) \leftarrow (i + 1, j + 1, k + 1)$;
17: **else**
18: $(i, j) \leftarrow (i + 1, j + 1)$;
19: **end if**
20: **else if** $F_i \neq 0$ and $F_i \succ G_j$ **then**
21: $X_k \leftarrow F_i$;
22: $(i, k) \leftarrow (i + 1, k + 1)$
23: **else if** $G_j \neq 0$ **then**
24: $X_k \leftarrow G_j$
25: $(j, k) \leftarrow (j + 1, k + 1)$
26: **end if**
27: **if** $k = N$ **then**
28: Associate i and j with X;
29: **return** X_k;
30: **end if**
31: **end while**
32: Associate i and j with X;
33: **return** 0;

Algorithm 2 - Lazy Multiplication

Input: The lazy polynomials F and G so that $F^\infty = f$ and $G^\infty = g$, a positive integer N (the desired term), and the lazy 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 $\succeq$ 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;

Theorem 1. To force every term of X (that is to completely determine the standard form of $f \times g$) in Algorithm 2, requires $O(\#f\#g \log \#g)$ monomial comparisons, space for a heap with at most $\#g$ terms, and space for $O(\#f\#g)$ terms of the product.

Proof. The size of the heap is not effected by line 18, as this merely replaces the term coming out of the heap in line 16. The only place the heap can grow is on line 21, 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 at most $\#f\#g$ many terms it will require $O(\#f\#g)$ space.

Extracting/inserting from/to a heap with $\#g$ elements does

$O(\log \#g)$ many 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[9]. □

Remark 2. It is possible to improve multiplication so that the heap requires space for only $\min(\#f, \#g)$ terms and the number of monomial comparisons done is $O(\#f\#g \log \min(\#f, \#g))$. If $\#f < \#g$ and we could switch the order of the input (i.e. calculate $g \times f$ instead of $f \times g$) then the heap would be of size $\#f$. But we we may not know $\#f$ and $\#g$! So, we must quote the worst case scenario in our complexities (in fact we will emphasize this by using $\max(\#f, \#g)$ in our space complexities).

Algorithm 3 - Lazy 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 $f = g \times Q^\infty + R^\infty$.

Output: The N-th term of the quotient from $f \div g$.

1: **if** $F_1 = 0$ **then**
2: **return** 0
3: **end if**
4: **if** $N \leq |Q|$ **then**
5: {Q_N has already been calculated.}
6: **return** Q_N;
7: **end if**
8: **if** $|Q| = 0$ **then**
9: {Q has no information.}
10: Initialize a new heap H and insert F_1 into H;
11: $s \leftarrow 2$;
12: **else**
13: Let H be the heap associated with Q;
14: **end if**
15: **while** H is not empty **do**
16: $t \leftarrow 0$;
17: **repeat**
18: Extract $x \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + x$;
19: **if** $x = F_i$ and $F_{i+1} \neq 0$ **then**
20: Insert F_{i+1} into H;
21: **else if** $x = G_iQ_j$ and Q_{j+1} is forced **then**
22: Insert $-G_iQ_{j+1}$ into H;
23: **else if** $x = G_iQ_j$ and Q_{j+1} is delayed **then**
24: s ← s + 1; {Sleep $-G_iQ_{j+1}$}
25: **end if**
26: **if** $x = G_iQ_1$ and $G_{i+1} \neq 0$ **then**
27: Insert $-G_{i+1}Q_1$ into H;
28: **end if**
29: **until** (H is empty) or (t and H_{max} are not like terms)
30: **if** $t \neq 0$ and $g_1|t$ **then**
31: $Q_{|Q|+1} \leftarrow t/G_1$; {Now $Q_{|Q|+1}$ is a forced term.}
32: **for** k from 2 to s **do**
33: Insert $-G_k \cdot t/G_1$ into H; {Insert all terms that are sleeping into H}
34: **end for**
35: **else**
36: $R_{|R|+1} \leftarrow t$; {Now $R_{|R|+1}$ is a forced term.}
37: **end if**
38: **if** $|Q| = N$ **then**
39: Associate the heap H with Q.
40: **return** Q_N;
41: **end if**
42: **end while**
43: Associate the (empty) heap H with Q;
44: **return** 0;

Theorem 2. To force every term of Q and R (that is to completely determine q and r such that $f = g \times q + r$) in Algorithm 3 requires $O((\#f + \#q\#g) \log \#g)$ many monomial comparisons, space for a heap with $O(\#g)$ terms, and space for $O(\#q + \#r)$ terms of the solution.

Proof. The size of the heap H, denoted $|H|$ is unaffected by lines 20 and 22 since these lines only replace terms coming out of the heap. Line 24 merely increments s and does not increase $|H|$. The only place where H can grow is line 27 in which a new term of g is added to the heap, this is clearly bounded by $\#g$. It is clear that we require $O(\#q + \#r)$ space to store the quotient and remainder.

All terms of f and $q \times g$ 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 [8]. □

3. FORGETFUL ARITHMETIC

There is a variant to delayed polynomial arithmetic that has some useful properties. Consider that the operations from the previous section can be compounded to form polynomial expressions. That is, we could use delayed arithmetic to calculate the n-th term of say, $A \times B - C \times D$. When we do this we store the intermediate calculations (namely the products $A \times B$ and $C \times D$) to provide quick re-access to terms. But, if re-access was not required we could 'forget' these terms instead. A 'forgetful' operation is like a delayed operation but intermediate terms won't be stored. For this reason, forgetful operations are potentially useful when expanding compounded polynomial expressions with large intermediate subexpressions.

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. For some delayed polynomial F and monomial order $\succeq$, the `next` command returns the $\succeq$-next uncalculated term of a polynomial (eventually returning only zeros) and satisfies $\texttt{next}(F) + \texttt{next}(F) + \texttt{next}(F) + \ldots = F^\infty$ and $\texttt{next}(F) \succ \texttt{next}(F) \succ \ldots \succ \texttt{next}(F) = 0 = \texttt{next}(F) = \ldots$.

Definition 3. 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 to be 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, 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 $f \times g$, 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.

For the same reason our division algorithm can not accept a forgetful divisor as it must be repeatedly multiplied by terms of the quotient (thus the quotient can not be forgetful either). However, we will see that the dividend *can* be forgetful which is a highly desirable feature (see Section 4). The only 'fully' forgetful (forgetful input and output) arithmetic operation we can have is addition (although polynomial differentiation and scalar multiplication can also fully forgetful).

The variant of multiplication that takes as input delayed polynomials, returning a forgetful polynomial, is a trivial change to Algorithm 2. In this case all that must be done is to remove the 'if' statement on line 28 so that the $\succeq$-next, instead of the N-th, term is returned. As this is not a significant change, we will not present and algorithm for forgetful multiplication. 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 lazy divisor returning a *fully forced* quotient and remainder. Aside from enabling division to accept a forgetful dividend, there are no other improvements.

Theorem 3. When multiplying f by g the worst case storage complexity for forgetful multiplication is $O(max(\#f, \#g))$ (the storage required for the heap).

Proof. A quick inspection of Algorithm 2 will show that the only time a previous term of the product is used is on line 3 and line 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 nor re-access terms of a forgetful polynomial, we can eliminate the storage needed to do this requiring only space for a heap with $\max(\#f, \#g)$ terms. □

Algorithm 4 - Forgetful Addition

Input: 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: $(t_F, t_G) \leftarrow (\texttt{next}(F), \texttt{next}(G))$;
4: **else**
5: Set t_F and t_G to the values associated with X;
6: **end if**
7: **while** $t_F \neq 0$ or $t_G \neq 0$ **do**
8: **if** t_F and t_G are like terms **then**
9: $ans \leftarrow t_F + t_G$;
10: $(t_F, t_G) \leftarrow (\texttt{next}(F), \texttt{next}(G))$
11: **else if** $t_F \neq 0$ and $t_F \succ t_G$ **then**
12: $ans \leftarrow t_F$;
13: $t_F \leftarrow \texttt{next}(F)$
14: **else if** $t_G \neq 0$ **then**
15: $ans \leftarrow t_G$
16: $t_G \leftarrow \texttt{next}(G)$
17: **end if**
18: **if** $ans \neq 0$ **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;

Theorem 4. 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 3 will only have to remember three values : ans, t_F and t_G. □

Algorithm 5 - Forgetful Division

Input: A 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 $f = g \times Q^\infty + R^\infty$.

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 ← 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**
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;

In its current form Algorithm 5 returns a fully forced quotient Q and reaminder R. It is straightforward to modify this algorithm to return a forgetful remainder instead. We simply have line 32 return t instead of saving a term to the remainder and change line 35 to return 0 (for when terms of R have been exhausted). In the interest of space we will assume this modification has been done as:

Algorithm 6 - Forgetful Division (with forgetful remainder)

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

Output: The delayed polynomial Q and forgetful polynomial R so that $f = g \times Q^{\infty} + R^{\infty}$.

Theorem 5. In algorithm 6, when calculating $f \div g$ the space required (*including space for the input*) to force every term of the forgetful remainder R is:

1. Space for a heap with $\#g$ terms.
2. Space for $\#q$ terms of the quotient.
3. Space for $\#g$ terms of the divisor.
4. Space for *one* term of the dividend f.

Proof.

1. As there has been no change to the division algorithm, Theorem 2 implies the heap has $\#g$ many terms.
2. To fully force every term of a delayed polynomial Q requires storage for $\#q$ many terms.
3. As G is a delayed polynomial that will be fully forced during the execution we require space to store $\#g$ many terms for the divisor.
4. As F is a forgetful polynomial we are restricted to only accessing one term from F at a time (where no previously calculated terms are cached). Therefore we only require space to store one term of f.

□

4. APPLICATIONS

We give two applications of forgetful polynomial arithmetic: the Bareiss algorithm and the Subresultant algorithm. These algorithms both have a deficiency in that intermediate calculations can become quite large with respect to the algorithms output. By using forgetful operations we can bypass the need to explicitly store intermediate polynomials and thus reduce the operating space of the each algorithm significantly.

4.1 The Bareiss Algorithm

The Bareiss algorithm is 'fraction free' approach for calculating determinants due to Bareiss [3] who noted that the method was first known to Jordan. The algorithm does exact divisions over any integral domain to avoid fractions.

The Bareiss algorithm is given below. In the case where $\mathbf{M}_{k,k} = 0$ (which prevents us from dividing by $\mathbf{M}_{k,k}$ in the next step) it would be straightforward to add code (between lines 2 and 3) to find a non-zero pivot. That is, if $\mathbf{M}_{k-1,k-1} = 0$ and there is some $\mathbf{M}_{k-1,i} \neq 0$ for $i = k, \ldots, n$ then we can exchange the i-th column with the $(k-1)$-th column. For the purpose of this exposition we assume no pivoting is required.

Algorithm 6 - Bareiss Algorithm

Input: $\mathbf{M}$ an n-square matrix with entries over an integral domain $\mathcal{D}$.

Output: The determinant of $\mathbf{M}$.

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** $\mathbf{M}_{n,n}$

The problem is the exact division in line 5. In the final division where the determinant $M_{n,n}$ is obtained by dividing by $M_{n-1,n-1}$ the dividend must be larger than the determinant. It is quite possible (in fact typical) that this calculation (of the form $\frac{A \times B - C \times D}{E}$) produces a dividend that is *much* larger than the corresponding quotient and denominator. This final division can be the bottleneck of the entire algorithm.

Example 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 6) will have a dividend of 128,530 terms, whereas the divisor and quotient will only have 427 and 6,090 terms respectively.

To overcome this problem recall that we use forgetful arithmetic to construct the quotient of $\frac{A \times B - C \times D}{E}$ without having to store $A \times B - C \times D$ in its entirety (in fact the forgetful algorithms were designed to do precisely this calculation).

Theorem 6. Calculating $Q = \frac{A \times B - C \times D}{E}$ (an exact division) with forgetful operations requires space for at most

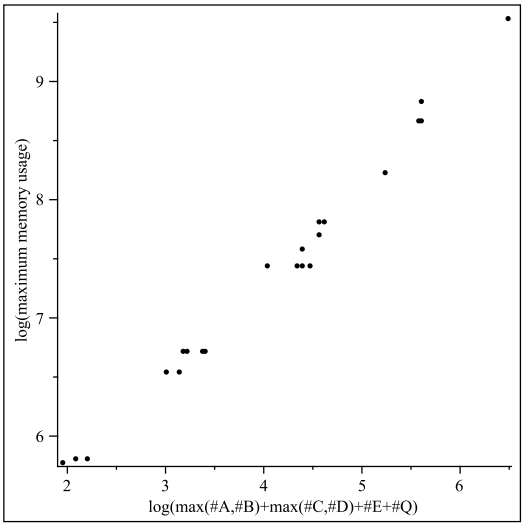

Figure 1: Maximum memory usage for line 5 of the Bareiss Algorithm given the symmetric Toeplitz matrix generated by $[x_1, \ldots, x_7]$ when using forgetful polynomials.

$O(\max(\#A, \#B) + \max(\#C, \#D) + 1 + \#E + \#Q)$ terms at any one time.

PROOF. We have from Theorem 3 that the products $A \times B$ and $C \times D$ require at most $\max(\#A, \#B)$ and $\max(\#C, \#D)$ space, where the difference of these products requires $O(1)$ space by Theorem 4. Since there is no remainder because the division is exact, the division algorithm will use $O(\#E+\#Q)$ storage by Theorem 2. Summing these complexities gives the desired result. □

We have implemented a package of lazy and forgetful algorithms for polynomial arithmetic in C. The implications of this theorem and can be observed in our implementation of the Bareiss Algorithm with forgetful polynomials. In figure 1 we have measured the amount of memory used by the exact division on line 5. The figure shows a linear relationship with the size of the input polynomials A, B, C, D and E.

4.2 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. The extended subresultant algorithm will be able to do this by finding $s, t \in \mathcal{D}[x]$ such that $s \cdot u + t \cdot m = \text{Res}(u, m, x)$. In this case $\texttt{deg}_x(s) < \texttt{deg}_x(m)$ and the inverse of $u \in \mathcal{D}[x]/\langle m \rangle$ is $s/\text{Res}(u, m, x)$.

Our interest is finding subresultants (and inverses) in $\mathcal{D}[x]$ when $\mathcal{D} = \mathbb{Z}$ or $\mathcal{D} = \mathbb{Z}[y, z, \ldots]$. The Subresultant algorithm uses pseudo-division instead of ordinary division (which the Euclidean algorithm uses) to avoid computing with fractions in the quotient field $\mathcal{D}/\mathcal{D}$. We recall the definition of pseudo-remainder and pseudo-quotient.

Definition 4. Let $\mathcal{D}$ be an integral domain and $f, g \in \mathcal{D}[x]$ with $f \neq 0$, $g \neq 0$. Let $\alpha = \texttt{lcoeff}_x(g)^{\delta+1}$ where $\delta = \texttt{deg}_x(f) - \texttt{deg}_x(g)$. 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 $\texttt{deg}_x(\tilde{r}) < \texttt{deg}_x(g)$ (the division algorithm ensures this is the case).

One can show (e.g. see Ch. 2. of Geddes et. al. [5]) that pseudo-quotient $\tilde{q}$ and pseudo-remainder $\tilde{r}$ are elements of $\mathcal{D}[x]$ and are unique.

The *extended* Subresultant algorithm [7] is given by Algorithm 7. The operations $\texttt{deg}_x$, prem, pquo, and $\texttt{lcoeff}_x$, stand for the degree in x, pseudo-remainder, pseudo-quotient and leading coefficient in x (respectively).

ALGORITHM 7 - EXTENDED SUBRESULTANT ALGORITHM

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

1: $(g, h) \leftarrow (1, -1)$;
2: $(s_0, s_1, t_0, t_1) \leftarrow (1, 0, 0, 1)$;
3: **while** $\texttt{deg}_x(v) \neq 0$ **do**
4: $\quad d \leftarrow \texttt{deg}_x(u) - \texttt{deg}_x(v)$;
5: $\quad \tilde{r} \leftarrow \texttt{prem}(u, v, x)$;
6: $\quad \tilde{q} \leftarrow \texttt{pquo}(u, v, x)$; {$\tilde{r}$ and $\tilde{q}$ are computed simultaneously.}
7: $\quad u \leftarrow v$;
8: $\quad \alpha \leftarrow \texttt{lcoeff}_x(v)^{d+1}$;
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: $\quad g \leftarrow \texttt{lcoeff}_x(u)$;
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;

The problematic calculation occurs when finding the pseudo-remainder on line 5. 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 12. In fact $\tilde{r}$ can be much larger than the resultant $\text{Res}(u, v, x)$.

Example 2. Consider the two polynomials $f = \sum_{i=1}^{9} x_i + \sum_{i=1}^{9} x_i^3$ and $g = \sum_{i=1}^{9} x_i^2$ in $\mathbb{Z}[x_1, \ldots, x_9]$. When we apply the extended subresultant algorithm to these polynomials we find that in the last iteration, the pseudo-remainder $\tilde{r}$ has $426,252$ terms but the quotient v has only $15,085$ (v is the resultant in this case).

To resolve this we will let the pseudo-remainder be a forgetful polynomial so that the numerator on line 12 will not

have to be explicitly stored. This is can be accomplished by using the Algorithm 6 since (when f and g regarded as univariate polynomials in x) calculating $\texttt{prem}(f, g, x)$ is equivalent to calculating $\texttt{divide}(\alpha^{\delta+1} f, g)$ where $\alpha = \texttt{lcoeff}_x(g)$ and $\delta = \texttt{deg}_x(f) - \texttt{deg}_x(g)$. Note, in order to implement pseudo-division, the monomial ordering used must satisfy $Yx^n \succ Zx^{n-1}$ for all monomials Y and Z.

5. IMPLEMENTATION

We have implemented a C-library for doing lazy (and forgetful) arithmetic for polynomials with coefficients that are machine integers. In our implementation we are representing monomials as single machine integers (which allows us to compare and multiply monomials in one machine instruction). This representation, analyzed by Monagan and Pearce [8], is based on Bachmann and Schönemann's scheme [2].

We found that there was no significant loss in efficiency when computing with this package. That is, to force every term of a lazy (or forgetful) product, quotient, or remainder was never slower than doing the same calculation with Maple (in fact we were two to three times faster). The C-structure we are using to represent a lazy polynomial is given bellow.

Listing 1: The delayed polynomial structure.

```
1  struct poly {
2      int N;
3      TermType *terms;
4      struct poly *F1;
5      struct poly *F2;
6
7      TermType (*Method)
8      (int n, struct poly *F,
9      struct poly *G, struct poly *H);
10
11     int state[6];
12     HeapType *Heap;
13 };
14
15 typedef struct poly PolyType;
```

The variable N is the number of forced terms, and $F1$ and $F2$ are two other lazy polynomials which the procedure `Method` (among `ADD`, `MULT` and `DIVIDE`) is applied to. As previously discussed `Method` requires three inputs, two lazy polynomials to operate on, and a third lazy polynomial where the solution is stroed (and where the current heap can be found). The array `state[6]` is a place to put local variables that get erased but need to be maintained, and `Heap` is the heap which the procedure `Method` uses.

It is useful to define a procedure `Term` which produces the n-th term of the delayed polynomial F, calculating it if necessary. This procedure enables us to follow the pseudo-code given more directly as `Term(i,F)` $= F_i$.

Listing 2: 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  };
```

Many details about the implementation have been omitted but it is this structure and procedure that are the two most important building blocks for development.

6. FUTURE WORK

Our original motivation for considering lazy algorithms for polynomials was an intermediate calculation in Buchberger's algorithm. Buchberger's algorithm transforms a set of generators for a polynomial ideal into a Gröbner basis [4]. 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 $L = \texttt{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$. This calculation appears to be wasteful as it is known that $S(f,g)$ reduces to zero when $\texttt{gcd}(\mathrm{LT}(f,), \mathrm{LT}(g)) = 1$. 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.

7. REFERENCES

[1] W. H. Burge and S. M. Watt. Infinite Structures in Scratchpad II. *Proc. EUROCAL '87*, Springer-Verlad LNCS **378**, 1989.

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

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

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

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

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

[7] Marc Moreno Maza and Renaud Rioboo. Polynomial gcd computations over towers of algebraic extensions. In Gérard D. Cohen, Marc Giusti, and Teo Mora, editors, *AAECC*, volume 948 of *Lecture Notes in Computer Science*, pages 365–382. Springer, 1995.

[8] 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.

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

[10] Joris van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Computation*, 11(1-000), 2002.

[11] 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.