Lazy and Forgetful Polynomial Arithmetic and Applications

Michael Monagan[1] and Paul Vrbik[2]

[1] Simon Fraser University, Department of Mathematics, Burnaby, B.C. Canada
[2] The University of Western Ontario, Department of Computer Science, London, ON Canada

Abstract. We present lazy and forgetful algorithms for 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

Lazy algorithms were first introduced into computer algebra systems by Burge and Watt [3] where they were used in Scratchpad II for power series arithmetic. 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 in [3] required $O(n^4)$. In [9] Watt showed how to reduce this to $O(n^2)$.

van der Hoeven considers lazy algorithms for multiplication of power series to $O(x^n)$ which are asymptotically fast [8]. 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 Hoeven 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 of these results assume *dense* power series and our interest is the sparse case.

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 $X_1 \succ X_2 \succ \cdots \succ X_n$ 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.

Let f, g be polynomials in the standard form. Johnson's [5] multiplication algorithm is based on the observation that multiplying $f = f_1 + \cdots + f_n$ by $g = g_1 + \cdots + g_m$ can be done by executing a simultaneous m-ary merge on the set of *sorted* sequences

$$S = \{(f_1 g_1, \ldots, f_n g_1), \ldots, (f_1 g_m, \ldots, f_n g_m)\}.$$

Johnson used a heap H, initialized to contain the terms $f_1g_1, f_1g_2, \ldots, f_1g_m$, to merge the m sequences. The number of terms in this heap never exceeds $\#g$ and inserting into and extracting terms from H costs $O(\log \#g)$ monomial comparisons per insertion/extraction. Therefore, 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 and requires auxiliary space for at most $\#g$ terms in the heap plus space for the output.

Monagan and Pearce [6] extended this heap algorithm to polynomial division. Recall that when we do $f \div g$ we are trying to construct the quotient q and remainder r such that $f - qg - r = 0$. One could use a heap to store the sum $f - qg$ by merging the set of $\#g + 1$ sorted sequences

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

where $m = \#g$ and $k = \#q$. Alternatively we may see the heap as storing the sum $f - \sum_{i=1}^{m} g_i \times (q_1 + q_2 + \cdots + q_k)$.

These heap algorithms dealt only with the so-called zealous (non-lazy) polynomials. Our contributions are the variations of these algorithms that enable us to compute in a lazy and forgetful manner.

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. Furthermore, we save intermediate results from this calculation so that the i-th term where $i \leq n$ will be 'calculated' instantaneously.

Definition 1. *A* lazy 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 $N \geq 0$. To ensure F^N is always defined we let $f_i = 0$ when $i > n$. This admits the useful notation $F^\infty = f$.*

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 $\#F = |F^\infty| = \#f$).*

A lazy polynomial must satisfy two conditions regarding computation: all the forced terms of F are cached for re-access and calculating a delayed term of F should force as few terms as possible.

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. We will use the heap methods for division and multiplication and a simple merge for addition. Since these methods build the result in $\succeq$-order anyway, we simply halt and return once n non-zero terms are generated. But, in order to initially populate the heap one polynomial must be fully forced. We give an optimization that avoids this.

Claim. Let f, g be polynomials in the standard form and $S[j] = (f_1 g_j, \dots, f_n g_j)$. If $f_1 g_j$ is in the heap H, then no term of the sequences $S[j+1], \dots, S[m]$ can be the $\succ$-largest term of H.

Proof. By the definition of a monomial ordering we have: if $g_j \succ g_{j+1} \succ \dots \succ g_m$, then $f_1 g_j \succ f_1 g_{j+1} \succ \dots \succ f_1 g_m$. As $f_1 g_{j+1}, \dots, f_1 g_m$ are (respectively) the $\succeq$-largest terms of $S[j+1], \dots, S[m]$, it follows that $f_1 g_j$ is $\succeq$-larger than any term of $S[j+1], \dots, S[m]$. The claim is an immediate consequence of this.

This claim gives a natural replacement scheme that ensures no term is prematurely calculated and put in the heap. For multiplication this is reflected in lines (13)-(15) of Algorithm 2. For division 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 $-q_i g_j$ 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, 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’.

We also require no work to be repeated to calculate $X_{N-1}, \dots, X_1$ after calculating X_N. To achieve this we pass our algorithms the approximation X^N, which must also record the state of the algorithm that generated it. 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 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 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 $N \le |X|$ then {$X_N$ has already been calculated.} return $X_N$; end if
 2: if $|X| = 0$ then
 3:     {$X$ has no information.}
```

```
4:     Initialize a heap $H$ and insert $(F_1G_1, 1, 1)$; {Order the heap by $\succeq$ on the mono-
       mials in the first position.}
5:     $k \leftarrow 1$;
6:  else
7:     Let $H$ be the heap associated with $X$;
8:     $k \leftarrow$ number of elements in $H$;
9:  end if
10: while $H$ is not empty do
11:    $t \leftarrow 0$;
12:    repeat
13:       Extract $(s, i, j) \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + s$;
14:       if $F_{i+1} \neq 0$ then Insert $(F_{i+1}G_j, i + 1, j)$ into $H$; end if
15:       if $i = 1$ and $G_{j+1} \neq 0$ then Insert $(F_1G_{j+1}, 1, j + 1)$ into $H$;  end if
16:    until ($H$ is empty) or ($t$ and $H_{max}$ are not like terms);
17:    if $t \neq 0$ then $(X_k, k) \leftarrow (t, k + 1)$; end if
18:    if $k = N$ then Associate the heap $H$ with $X$; return $X_k$; end if
19: end while
20: Associate the (empty) heap $H$ with $X$;
21: 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. Proceeding as in [7], the size of the heap is not effected by line 14, as this merely replaces the term coming out of the heap in line 13. The only place the heap can grow is on line 15, 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. As every term of the product passes through the heap, we do $O(\#f\#g)$ extractions/insertions totaling $O(\#f\#g \log \#g)$ monomial comparisons.

Remark 1. 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 $\max(\#f, \#g)$).

Algorithm 3 - Lazy Division

```
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 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 return 0; end if
2: if $N \leq |Q|$ then {$Q_N$ has already been calculated.} return $Q_N$;
3: if $|Q| = 0$ then
```

```
 4:   {$Q$ has no information.}
 5:   Initialize a new heap $H$ and insert $F_1$ into $H$;
 6:   $s \leftarrow 2$;
 7: else
 8:   Let $H$ be the heap associated with $Q$;
 9: end if
10: while $H$ is not empty do
11:   $t \leftarrow 0$;
12:   repeat
13:     Extract $x \leftarrow H_{max}$ from the heap and assign $t \leftarrow t + x$;
14:     if $x = F_i$ and $F_{i+1} \neq 0$ then
15:       Insert $F_{i+1}$ into $H$;
16:     else if $x = G_i Q_j$ and $Q_{j+1}$ is forced then
17:       Insert $-G_i Q_{j+1}$ into $H$;
18:     else if $x = G_i Q_j$ and $Q_{j+1}$ is delayed then
19:       $s \leftarrow s + 1$; {Sleep $-G_i Q_{j+1}$}
20:     end if
21:     if $x = G_i Q_1$ and $G_{i+1} \neq 0$ then Insert $-G_{i+1} Q_1$ into $H$; end if
22:   until ($H$ is empty) or ($t$ and $H_{max}$ are not like terms)
23:   if $t \neq 0$ and $g_1 | t$ then
24:     $Q_{|Q|+1} \leftarrow t/G_1$; {Now $Q_{|Q|+1}$ is a forced term.}
25:     for $k$ from 2 to $s$ do
26:       Insert $-G_k \cdot t/G_1$ into $H$; {Insert all terms that are sleeping into $H$}
27:     end for
28:   else
29:     $R_{|R|+1} \leftarrow t$; {Now $R_{|R|+1}$ is a forced term.}
30:   end if
31:   if $|Q| = N$ then Associate the heap $H$ with $Q$; return $Q_N$; end if
32: end while
33: Associate the (empty) heap $H$ with $Q$;
34: 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. Proceeding as in [6], the size of the heap H, denoted $|H|$ is unaffected by lines 15 and 17 since these lines only replace terms coming out of the heap. Line 19 merely increments s and does not increase $|H|$. The only place where H can grow is line 21 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.

3 Forgetful Arithmetic

We propose a variant to lazy polynomial arithmetic that has useful properties. Consider that the operations from the previous section can be composed to form polynomial expressions. For example, we could use lazy arithmetic to calculate the n-th term of say, $A \times B - C \times D$. When we do this we store the intermediate terms But, if re-access was not required we could 'forget' these terms. A 'forgetful' operation is like a lazy operation but intermediate terms won't be stored. Forgetful operations are potentially useful when expanding compounded polynomial expressions with large intermediate subexpressions.

We can make some straightforward modifications to our lazy 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 lazy polynomial F and monomial order $\succeq$, the* `next` *command returns the $\succeq$-next un-calculated term of a polynomial (eventually returning only zeros).*

Remark 2. A forgetful polynomial F satisfies: $\texttt{next}\,(F) \succ \texttt{next}\,(F) \succ \cdots \succ \texttt{next}\,(F) = 0 = \texttt{next}\,(F) = \cdots$ and $\texttt{next}\,(F) + \texttt{next}\,(F) + \texttt{next}\,(F) + \cdots = F^{\infty}$.

Definition 3. *A* forgetful polynomial *is a lazy 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 any term but the current term 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. Consider multiplication for instance. Assuming that we must multiply each term of f by each term of g and we are limited to single time access to terms, this task is impossible. For if we calculate $f_1 g_2$ we cannot then 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, the dividend *can* be forgetful which is a highly desirable feature (see Section 5). The only 'fully' forgetful (forgetful input and output) arithmetic operation we can have is addition (although polynomial differentiation and scalar multiplication are also fully forgetful).

The variant of multiplication that takes as input lazy 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 18 so that the $\succeq$-next, instead of the N-th, term is returned. As this is not a significant change, we will not present an algorithm for forgetful multiplication. Division will take as input a forgetful dividend and lazy divisor returning a *fully forced* quotient and remainder.

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 2 and line 18. 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 5 - Forgetful Division

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

In its current form Algorithm 5 returns a fully forced quotient Q and remainder R. It is straightforward to modify this algorithm to return a forgetful remainder instead. We simply have line 29 return t instead of saving a term to the remainder and change line 32 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 lazy polynomial G so that $F^\infty = f$ and $G^\infty = g$.

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

Theorem 4. *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 lazy polynomial Q requires storage for $\#q$ many terms.
3. As G is a lazy 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 Implementation

We have implemented a C-library for doing lazy (and forgetful) arithmetic for polynomials with coefficients that are machine integers modulo p, for p some machine prime. In our implementation we represent monomials as single machine integers (which allows us to compare and multiply monomials in one machine instruction). This representation, analyzed by Monagan and Pearce [6], is based on Bachmann and Schönemann's scheme [1]. The C-structure we are using to represent a lazy polynomial is given below.

Listing 1.1. The lazy polynomial structure.

```
struct poly {
    int N;
    TermType *terms;
    struct poly *F1;
    struct poly *F2;
    TermType (*Method)(int n, struct poly *F,
                struct poly *G, struct poly *H);
    int state[6];
    HeapType *Heap;
};
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`, `DIVIDE`, and `DONE`) 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 stored (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.

The procedure `Term` produces the n-th term of the lazy polynomial F, calculating it if necessary, enabling us to follow the pseudo-code given more directly as `Term(i,F)` $= F_i$.

Listing 1.2. Term.

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

Many details about the implementation have been omitted but we note that we have built a custom wrapper that interfaces the C-library with Maple (a non-trivial technical feat). This allows us to manipulate polynomials in a lazy way at the Maple level but do calculations at the C level.

Benchmarks are given in Table 1 where we see that calculating in a lazy/forgetful manner is $3-5$ times slower than calculating directly with Monagan and Pearce's SDMP package (see [7]) or Singular. Roman Pearce pointed out that this is because we are not using *chaining* in our heap implementations. Chaining is a technique where like terms are grouped together in a linked list to dramatically reduce the number of monomial comparisons in the heap operations. In [7], Monagan and Pearce show that chaining improves the performance of multiplication and division using heaps by a factor of $3-5$.

Table 1. Benchmarks for Maple's SDMP package [7], Singular, and our lazy package on sparse examples.

	$f \times g \mod 503$			$(fg) \div f \mod 503$		
	SDMP	Singular	Lazy	SDMP	Singular	Lazy
$f=(1+x+y^2+z^3)^{20}$ $g=(1+z+y^2+x^3)^{20}$	0.26	0.28	1.2	0.28	0.38	1.4
$f=(1+x+y^3+z^5)^{20}$ $g=(1+z+y^3+x^5)^{20}$	0.35	0.67	1.3	0.38	0.65	1.4
$f=(1+x+y^3)^{100}$ $g=(1+x^3+y)^{100}$	2.2	1.1	10.8	2.5	2.04	11.1

5 Applications

We give two similar, but nonetheless independently important, 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.

5.1 The Bareiss Algorithm

The Bareiss algorithm is 'fraction free' approach for calculating determinants 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.

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. 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 $\mathbf{M}_{n,n}$ is obtained by dividing by $\mathbf{M}_{n-1,n-1}$ 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 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 we use forgetful arithmetic to construct the quotient of $\frac{A\times B - C\times D}{E}$ without explicitly storing $A \times B - C \times D$ (the forgetful algorithms were invented to do precisely this calculation).

Theorem 5. *Calculating* $Q = \frac{A\times B-C\times D}{E}$ *(an exact division) with forgetful operations requires space for at most* $O(\max(\#A, \#B)+\max(\#C, \#D)+\#E+\#Q)$ *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 $O(1)$ since it is merely a merge. As 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.

The implications of this theorem can be observed in Table 2 where we have measured the amount of memory used by our implementation of the Bareiss algorithm with forgetful polynomials. The table shows a linear relationship with the size of the input polynomials. For $n = 8$ the total space is reduced by a factor of $57184/832 = 68$ (compared to a Bareiss implementation that explicitly stores the quotient), which is significant.

Table 2. Let $Q = \frac{A\times B-C\times D}{E}$ be the division of line 5 of the Bareiss algorithm and $\alpha = \max(\#A, \#B) + \max(\#C, \#D)$. The following is a measurement of memory used by our implementation of the Bareiss algorithm using forgetful polynomials to calculate $\mathbf{M}_{n,n}$ when given the Toeplitz matrix generated by $[x_1, \ldots, x_7]$.

n	$\#A$	$\#B$	$\#C$	$\#D$	$\#E$	$\#A\#B + \#C\#D$	$\alpha + \#E + \#Q$	32-bit words	$\succeq$-comparisons
5	12	15	17	17	4	469	106	426	2817
6	35	51	55	55	12	4810	306	944	45632
7	35	62	70	70	12	7070	326	1462	70028
8	120	182	188	188	35	57184	832	3468	720696

5.2 The Extended Subresultant Algorithm

Given a UFD $\mathcal{D}$ and non-constant polynomial $m \in \mathcal{D}[x]$, we can form the quotient ring $F[x]/\langle m\rangle$ where F is the fraction field of $\mathcal{D}$. 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 does 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 $u \in F[x]/\langle m\rangle$ is $s/\mathrm{Res}(u, m, x)$.

Our interest is finding subresultants in $\mathcal{D}[x]$ and inverses in $F[x]$ 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 regular Euclidean algorithm uses) to avoid computing with fractions in the fraction field F of $\mathcal{D}$. We recall the definition of pseudo-remainder and pseudo-quotient.

Definition 4. *Let $f, g \in \mathcal{D}[x]$. The pseudo-quotient $\tilde{q}$ and pseudo-remainder $\tilde{r}$ are the ordinary quotient and remainder of $\alpha \times f$ divided by g where $\alpha = \texttt{lcoeff}_x(g)^{\delta+1}$ and $\delta = \texttt{deg}_x(f) - \texttt{deg}_x(g)$. Thus they satisfy $\alpha f = \tilde{q}\, g + \tilde{r}$.*

One can show (e.g. see Ch. 2. of [4]) that $\tilde{q}$ and $\tilde{r}$ are elements of $\mathcal{D}[x]$. The *extended* Subresultant algorithm 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 $r = \mathrm{Res}(u, v, x) \in \mathcal{D}$ and $s, t \in \mathcal{D}[x]$ where $s \cdot u + t \cdot v = r$.
 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:     $d \leftarrow \texttt{deg}_x(u) - \texttt{deg}_x(v)$;
 5:     $(\tilde{r}, \tilde{q}) \leftarrow (\texttt{prem}(u, v, x), \texttt{pquo}(u, v, x))$; {$\tilde{r}$, $\tilde{q}$ are computed simultaneously.}
 6:     $u \leftarrow v$;
 7:     $\alpha \leftarrow \texttt{lcoeff}_x(v)^{d+1}$;
 8:     $(s, t) \leftarrow (\alpha \cdot s_0 - s_1 \cdot \tilde{q}, \alpha \cdot t_0 - t_1 \cdot \tilde{q})$;
 9:     $(s_0, t_0) \leftarrow (s_1, t_1)$;
10:     $v \leftarrow \tilde{r} \div (-g \cdot h^d)$;
11:     $(s_1, t_1) \leftarrow (s \div (-g \cdot h^d), t \div (-g \cdot h^d))$
12:     $g \leftarrow \texttt{lcoeff}_x(u)$;
13:     $h \leftarrow (-g)^d \div h^{d-1}$;
14: end while
15: $(r, s, t) \leftarrow (v, s_1, t_1)$;
16: return $r, s, t$;
```

A bottleneck 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 10. In fact $\tilde{r}$ can be much larger than the resultant $\mathrm{Res}(u, v, x)$.

Example 2. Consider the two polynomials $f = x_1^6 + \sum_{i=1}^{8} \left(x_i + x_i^3\right)$ and $g = x_1^4 + \sum_{i=1}^{8} x_i^2$ 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 $427,477$ terms but the quotient v has only $15,071$ (v is the resultant in this case).

To solve this problem we let the pseudo-remainder be a forgetful polynomial so that the numerator on line 10 does not have to be explicitly stored. This is accomplished by using Algorithm 5 since (when f and g regarded as univariate polynomials in x) calculating $\texttt{prem}(f, g, x)$ is equivalent to dividing $\alpha \times f$ by g using ordinary division with remainder. Table 3 shows the benefit of calculating in this manner. In the final iteration only a max 634+2412=3046 terms will need to be explicitly stored to calculate a pseudo-remainder with 14,692 terms. Note, in order to implement this pseudo-division in the sparse distributed form, the monomial ordering used must satisfy $Yx^n \succ Zx^{n-1}$ for all monomials Y and Z that do not involve x.

Table 3. Let $\tilde{r}, \tilde{q}$ be from line 5 and $v, -g \cdot h^d$ be from line 10 of Algorithm 7. The following is a measurement of the memory used by our implementation of the extended subresultant algorithm using forgetful polynomials to calculate $\mathrm{Res}(f, g, x_1)$ where $f = x_1^8 + \sum_{i=1}^{5} \left(x_i + x_i^3\right), g = x_1^4 + \sum_{i=1}^{5} x_i^2 \in \mathbb{Z}[x_1, \ldots, x_5]$ at iteration n.

n	$\#\tilde{r}$	$\#\tilde{q}$	$\#v$	$\#\left(-g \cdot h^d\right)$	32-bit words	$\succ$-comparisons
1	29	7	29	1	236	137
2	108	6	108	1	154	953
3	634	57	634	1	2,672	75,453
4	14,692	2412	2,813	70	83,694	25,801,600

Conclusion

We presented algorithms for lazy and forgetful polynomial arithmetic and two applications. These applications have demonstrated that the space complexity of the Bareiss algorithm and extended Subresultant algorithm can be significantly improved by using forgetful arithmetic, as proposed in this paper.

Acknowledgement We wish to thank Dr. Jürgen Gerhard, Dr. Marc Moreno Maza, Dr. Eric Schost, and Roman Pearce for their contributions to this work.

References

1. O. Bachman and H. 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. W. H. Burge and S. M. Watt. Infinite structures in Scratchpad II. In *EUROCAL 1987*, page 378. Springer-Verlad, 1989.
4. K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.
5. S. C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, 8(3):63–71, 1974.
6. M. Monagan and R. Pearce. *Polynomial Division using Dynamic Arrays, Heaps, and Packed Exponent Vectors*, volume 4770 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.
7. M. Monagan and R. Pearce. Sparse polynomial arithmetic using a heap. *Journal of Symbolic Computation - Special Issue on Milestones In Computer Algebra*, 2008. Submitted.
8. J. van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Computation*, 11(1-000), 2002.
9. 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.