mathematics

MDPI

Article

Algorithms and Data Structures for Sparse Polynomial Arithmetic

Mohammadali Asadi, Alexander Brandt *, Robert H. C. Moir and Marc Moreno Maza

Department of Computer Science, University of Western Ontario, London, ON N6A 5B7, Canada; masadi4@uwo.ca (M.A.); rmoir3@uwo.ca (R.H.C.M.); moreno@csd.uwo.ca (M.M.M.)
* Correspondence: abrandt5@uwo.ca

Received: 1 February 2019; Accepted: 12 May 2019; Published: 17 May 2019

Abstract: We provide a comprehensive presentation of algorithms, data structures, and implementation techniques for high-performance sparse multivariate polynomial arithmetic over the integers and rational numbers as implemented in the freely available Basic Polynomial Algebra Subprograms (BPAS) library. We report on an algorithm for sparse pseudo-division, based on the algorithms for division with remainder, multiplication, and addition, which are also examined herein. The pseudo-division and division with remainder operations are extended to multi-divisor pseudo-division and normal form algorithms, respectively, where the divisor set is assumed to form a triangular set. Our operations make use of two data structures for sparse distributed polynomials and sparse recursively viewed polynomials, with a keen focus on locality and memory usage for optimized performance on modern memory hierarchies. Experimentation shows that these new implementations compare favorably against competing implementations, performing between a factor of 3 better (for multiplication over the integers) to more than 4 orders of magnitude better (for pseudo-division with respect to a triangular set).

Keywords: sparse polynomials; polynomial arithmetic; normal form; pseudo-division; pseudo-remainder; sparse data structures

1. Introduction

Technological advances in computer hardware have allowed scientists to greatly expand the size and complexity of problems tackled by scientific computing. Only in the last decade have sparse polynomial arithmetic operations (Polynomial arithmetic operations here refers to addition, subtraction, multiplication, division with remainder, and pseudo-division) and data structures come under focus again in support of large problems which cannot be efficiently represented densely. Sparse polynomial representations was an active research topic many decades ago out of necessity; computing resources, particularly memory, were very limited. Computer algebra systems of the time (which handled multivariate polynomials) all made use of sparse representations, including ALTRAN [1], MACSYMA [2], and REDUCE [3]. More recent work can be categorized into two streams, the first dealing primarily with algebraic complexity [4,5] and the second focusing on implementation techniques [6,7]. Recent research on implementation techniques has been motivated by the *efficient* use of memory. Due to reasons such as the processor–memory gap ([8] Section 2.1) and the memory wall [9], program performance has become limited by the speed of memory. We consider these issues foremost in our algorithms, data structures, and implementations. An early version of this work appeared as [10].

Sparse polynomials, for example, arise in the world of polynomial system solving—a critical problem in nearly every scientific discipline. Polynomial systems generally come from real-life applications, consisting of multivariate polynomials with rational number coefficients. Core routines for determining solutions to polynomial systems (e.g., Gröbner bases, homotopy methods, or triangular decompositions) have driven a large body of work in computer algebra. Algorithms, data structures, and implementation techniques for polynomial and matrix data types have seen particular attention. We are motivated in our work on sparse polynomials by obtaining efficient implementations of triangular decomposition algorithms based on the theory of regular chains [11].

Our aim for the work presented in this paper is to provide highly optimized sparse multivariate polynomial arithmetic operations as a foundation for implementing high-level algorithms requiring such operations, including triangular decomposition. The implementations presented herein are freely available in the BPAS library [12] at www.bpaslib.org. The BPAS library is highly focused on performance, concerning itself not only with execution time but also memory usage and cache complexity [13]. The library is mainly written in the C language, for high-performance, with a simplified C++ interface for end-user usability and object-oriented programming. The BPAS library also makes use of parallelization (e.g., via the CILK extension [14]) for added performance on multi-core architectures, such as in dense polynomial arithmetic [15,16] and arithmetic for big prime fields based on Fast Fourier Transform (FFT) [17]. Despite these previous achievements, the work presented here is in active development and not yet been parallelized.

Indeed, parallelizing sparse arithmetic is an interesting problem and is much more difficult than parallelizing dense arithmetic. Many recent works have attempted to parallelize sparse polynomial arithmetic. Sub-linear parallel speed-up is obtained for the relatively more simple schemes of Monagan and Pearce [18,19] or Biscani [20], while Gastineau and Laskar [7,21] have obtained near-linear parallel speed-up but have a much more intricate parallelization scheme. Other works are quite limited: the implementation of Popescu and Garcia [22] is limited to floating point coefficients while the work of Ewart et al. [23] is limited to only 4 variables. We hope to tackle parallelization of sparse arithmetic in the future, however, we strongly believe that one should obtain an optimized serial implementation before attempting a parallel one.

Contributions and Paper Organization

Contained herein is a comprehensive treatment of the algorithms and data structures we have established for high-performance sparse multivariate polynomial arithmetic in the BPAS library. We present in Section 2 the well-known sparse addition and multiplication algorithms from [24] to provide the necessary background for discussing division with remainder (Section 3), an extension of the exact division also presented in [24]. In Section 4 we have extended division with remainder into a new algorithm for sparse pseudo-division. Our presentation of both division with remainder and pseudo-division has two levels: one which is abstract and independent of the supporting data structures (Algorithms 3 and 5); and one taking advantage of heap data structures (Algorithms 4 and 6). Section 5 extends division with remainder and pseudo-division to algorithms for computing normal forms and pseudo-division with respect to a triangular set; the former was first seen in [25] and here we extend it to the case of pseudo-division. All new algorithms are proved formally.

In support of all these arithmetic operations we have created a so-called alternating array representation for distributed sparse polynomials which focuses greatly on data locality and memory usage. When a recursive view of a polynomial (i.e., a representation as a univariate polynomial with multivariate polynomial coefficients) is needed, we have devised a succinct recursive representation which maintains the optimized distributed representation for the polynomial coefficients and whose conversion to and from the distributed sparse representation is highly efficient. Both representations are explained in detail in Section 6. The efficiency of our algorithms and implementations are highlighted beginning in Section 7, with implementation-specific optimizations, and then Section 8, which gathers our experimental results. We obtain speed-ups between a factor of 3 (for multiplication over the integers) and a factor of 18,141 (for pseudo-division with respect to a triangular set).

2. Background

2.1. Notation and Nomenclature

Throughout this paper we use the notation R to denote a ring (commutative with identity), $\mathbb{D}$ to denote an integral domain, and $\mathbb{K}$ to denote a field. Our treatment of sparse polynomial arithmetic requires both a distributed and recursive view of polynomials, depending on which operation is considered. For a distributed polynomial $a \in R[x_1, \ldots, x_v]$, a ring R, and variable ordering $x_1 < x_2 < \cdots < x_v$, we use the notation

$$a = \sum_{i=1}^{n_a} A_i = \sum_{i=1}^{n_a} a_i X^{\alpha_i},$$

where n_a is the number of (non-zero) terms, $0 \neq a_i \in R$, and α_i is an exponent vector for the variables $X = (x_1, \ldots, x_v)$. A term of a is represented by $A_i = a_i X^{\alpha_i}$. We use a lexicographical term order and assume that the terms are ordered decreasingly, thus $\mathrm{lc}(a) = a_1$ is the *leading coefficient* of a and $\mathrm{lt}(a) = a_1 X^{\alpha_1} = A_1$ is the *leading term* of a. If a is not constant the greatest variable appearing in a (denoted $\mathrm{mvar}(a)$) is the *main variable* of a. The maximum sum of the elements of α_i is the *total degree* (denoted $\mathrm{tdeg}(a)$). The maximum exponent of the variable x_i is the *degree with respect to x_i* (denoted $\deg(a, x_i)$). Given a term A_i of a, $\mathrm{coef}(A_i) = a_i$ is the coefficient, $\mathrm{expn}(A_i) = \alpha_i$ is the exponent vector, and $\deg(A_i, x_j)$ is the component of α_i corresponding to x_j. We also note the use of a simplified syntax for comparing monomials based on the term ordering; we denote $X^{\alpha_i} > X^{\alpha_j}$ as $\alpha_i > \alpha_j$.

To obtain a recursive view of a non-constant polynomial $a \in R[x_1, \ldots, x_v]$, we view a as a univariate polynomial in $\tilde{R}[x_j]$, with x_j called the *main variable* (denoted $\mathrm{mvar}(a)$) and where $\tilde{R} = R[x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_v]$. Usually, x_j is chosen to be x_v and we have $a \in R[x_1, \ldots, x_{v-1}][x_v]$. Given a term A_i of $a \in \tilde{R}[x_j]$, $\mathrm{coef}(A_i) \in R[x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_v]$ is the coefficient and $\mathrm{expn}(A_i) = \deg(A_i, x_j) = \deg(A_i)$ is the degree. Given $a \in \tilde{R}[x_j]$, an exponent e picks out the term A_i of a such that $\deg(A_i) = e$, so we define in this case $\mathrm{coef}(a, x_j, e) := \mathrm{coef}(A_i)$. Viewed specifically in the recursive way $\tilde{R}[x_j]$, the leading coefficient of a is an element of $\tilde{R}$ called the *initial* of a (denoted $\mathrm{init}(a)$) while the degree of a in the main variable x_j is called the *main degree* (denoted $\mathrm{mdeg}(a)$), or simply degree where the univariate view is understood by context.

2.2. Addition and Multiplication

Adding (or subtracting) two polynomials involves three operations: joining the terms of the two summands; combining terms with identical exponents (possibly with cancellation); and sorting of the terms in the sum. A naïve approach computes the sum $a + b$ term-by-term, adding a term of the addend (b) to the augend (a), and sorting the result at each step, in a manner similar to *insertion sort*. (This sorting of the result is a crucial step in any sparse operation. Certain optimizations and tricks can be used in the algorithms when it is known that the operands are in some sorted order, say in a *canonical form*. For example, obtaining the leading term and degree is much simpler, and, as is shown throughout this paper, arithmetic operations can exploit this sorting.) This method is inefficient and does not take advantage of the fact that both a and b are already ordered. We follow the observation of Johnson [24] that this can be accomplished efficiently in terms of operations and space by performing a single step of *merge sort* on the two summands, taking full advantage of initial sorting of the two summands. One slight difference from a typical merge sort step is that like terms (terms with identical exponent vectors) are combined as they are encountered. This scheme results in the sum (or difference) being automatically sorted and all like terms being combined. The algorithm is very straightforward for anyone familiar with merge sort. The details of the algorithm are presented in ([24], p. 65). However, for completeness we present the algorithm here using our notation (Algorithm 1).

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

```
$(i,j,k) := 1$
while $i \le n_a$ and $j \le n_b$ do
    if $\alpha_i < \beta_j$ then
        $c_k := b_j$; $\gamma_k := \beta_j$
        $j := j+1$
    else if $\alpha_i > \beta_j$ then
        $c_k := a_i$; $\gamma_k := \alpha_i$
        $i := i+1$
    else
        $c_k := a_i + b_j$; $\gamma_k := \alpha_i$
        $i := i+1$; $j := j+1$
        if $c_k = 0$ then
            continue    #Do not increment $k$
    $k := k+1$
end
while $i \le n_a$ do
    $c_k := a_i$; $\gamma_k := \alpha_i$
    $i := i+1$; $k := k+1$
while $j \le n_b$ do
    $c_k := b_j$; $\gamma_k := \beta_j$
    $j := j+1$; $k := k+1$
return $c = \sum_{\ell=1}^{k-1} c_\ell X^{\gamma_\ell}$
```

Multiplication of two polynomials follows the same general idea of addition: Make use of the fact that the multiplier and multiplicand are already sorted. Under our sparse representation of polynomials multiplication requires production of the product terms, combining terms with equal exponents, and then sorting the product terms. A naïve method computes the product $a \cdot b$ (where a has n_a terms and b has n_b terms) by distributing each term of the multiplier (a) over the multiplicand (b) and combining like terms:

$$c = a \cdot b = (a_1 X^{\alpha_1} \cdot b) + (a_2 X^{\alpha_2} \cdot b) + \cdots .$$

This is inefficient because all $n_a n_b$ terms are generated, whether or not like terms are later combined, and then all $n_a n_b$ terms must be sorted, and like terms combined. Again, following Johnson [24], we can improve algorithmic efficiency by generating terms in sorted order.

We can make good use of the sparse data structure for

$$a = \sum_{i=1}^{n_a} a_i X^{\alpha_i}, \quad \text{and} \quad b = \sum_{j=1}^{n_b} b_j X^{\beta_j},$$

based on the observation that for given α_i and β_j, it is always the case that $X^{\alpha_{i+1}+\beta_j}$ and $X^{\alpha_i+\beta_{j+1}}$ are less than $X^{\alpha_i+\beta_j}$ in the term order. Since we always have $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$, it is possible to generate product terms in order by merging n_a "streams" of terms computed by multiplying a single term of a distributed over b,

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

and then choosing the maximum term from the "heads" of the streams. We can consider this as an n_a-way merge where at each step, we select the maximum term from among the heads of the streams, making it the next product term, removing it from the stream in the process. The new head of the stream where a term is removed will then be the term to its right.

This sub-problem of selecting the maximum term among n_a different terms can be solved efficiently by making use of a priority queue data structure, which can be implemented as a heap (see Section 6.3 for implementation details). The virtue of using a heap was noticed by Johnson [24], but the description of his algorithm was left very abstract and did not make explicit use of a priority queue.

In Algorithm 2 we give our heap-based multiplication algorithm. This algorithm makes use of a few specialized functions to interface with the heap and the heads of streams contained therein. We provide here a simplified yet complete interface consisting of four functions. (Please note that algorithms for insertion and removal from a heap are standard and provided in any good reference on data structures and algorithms (see, e.g., [26]).) **heapInitialize**(a, B_1) initializes the heap by initiating n_a streams, where the head of the i-th stream is $A_i \cdot B_1$. Each of these heads are inserted into the heap. **heapInsert**(A_i, B_j) adds the product of the terms A_i and B_j to the heap. It is important to note, however, that the heap does not need to store the actual product terms but can store instead only the indices of the two factors, with their product only being computed when elements are removed from the heap. (This strategy is actually required in the case of pseudo-division (Section 7.4) where the streams themselves are updated over the course of the algorithm.) The exponent vector of the monomial must be computed on insertion, though, since this determines the insertion location (priority) in the heap. **heapPeek**$()$ returns the exponent vector γ of the top element in the heap and the stream index s from which the product term was formed, i.e., s such that the top element comes from the stream $A_s \cdot B$. Please note that nothing is removed from the heap by **heapPeek**$()$. **heapExtract**$()$ removes the top element of the heap, providing the product term. If the heap is empty **heapPeek**$()$ will return $\gamma = (-1, 0, \ldots, 0)$, which is, by design, less than any exponent of any polynomial term because the first element is -1. We therefore abuse notation and write $\gamma = -1$ for an empty heap.

Algorithm 2 HEAPMULTIPLYPOLYNOMIALS(a,b)

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

```
if $n_a = 0$ or $n_b = 0$ then
    return 0
$k := 1; C_1 := 0$
$s = 1;\ \gamma := \alpha_1 + \beta_1$  # Maximum possible value of $\gamma$
heapInitialize$(a, B_1)$
for $i = 1$ to $n_a$ do
    $f_i := 1$  # Indices of the current head of each stream
while $\gamma > -1$ do  # $\gamma = -1$ when the heap is exhausted
    if $\gamma \neq \mathrm{expn}(C_k)$ and $\mathrm{coef}(C_k) \neq 0$ then
        $k := k + 1$
        $C_k := 0$
    $C_k := C_k +$ heapExtract$()$
    $f_s := f_s + 1$
    if $f_s \leq n_b$ then
        heapInsert$(A_s, B_{f_s})$
    $(\gamma, s) :=$ heapPeek$()$  # Get degree and stream index of the top of the heap
end
if $C_k = 0$ then $k := k - 1$
return $c = \sum_{\ell=1}^{k} C_\ell = \sum_{\ell=1}^{k} c_\ell X^{\gamma_\ell}$
```

We note that while this algorithm seems simple in pseudo-code, its implementation, especially with respect to the heap, requires many subtle optimizations to achieve good performance. The discussions of such improvements are left to Section 7. Nonetheless the algorithm presented here is complete and correct.

Proposition 1. *Algorithm 2 terminates and is correct.*

Proof. Let $a, b \in R[x_1, \ldots, x_v]$. If either $n_a = 0$ or $n_b = 0$ then $a = 0$ or $b = 0$, in which case $c = 0$ and we are done. Otherwise, $c \neq 0$ and we initialize the heap with n_a pairs (A_i, B_1), $i = 1, \ldots, n_a$, we initialize the stream element indices f_i to 1, and we set $C_1 = 0$. We initially set $\gamma = \alpha_1 + \beta_1$, the maximum possible for polynomials a and b, and a guaranteed term of the product. This also serves to enter the loop for the first time. Since C_1 was initially set to 0, $C_k = 0$, so the first condition on line 9 is met, but not the second, so we move to line 12. Lines 12 through 15 extract the top of the heap, add it to C_k (giving $C_1 = A_1B_1$), and insert the next element of the first stream into the heap. This value of C_1 is correct. Since we add the top element of each stream to the heap, the remaining elements to be added to the heap are all less than at least one element in the heap. The next **heapPeek**() sets γ to one of $\alpha_2 + \beta_1$ or $\alpha_1 + \beta_2$ (or -1 if $n_a = n_b = 1$), and sets s accordingly. Subsequent passes through the loop must do one of the following: (1) if $C_k \neq 0$ and there exists another term with exponent $\mathrm{expn}(C_k)$, add it to C_k; (2) if $C_k = 0$, add to C_k the next greatest element (since for sparse polynomials we store only non-zero terms); or (3) when $C_k \neq 0$ and the next term has lower degree ($\gamma_k > \gamma$), increase k and then begin building the next C_k term. Cases (1) and (2) are both handled by line 12, since the condition on line 9 fails in both cases, respectively because $\gamma = \mathrm{expn}(C_k)$ or because $C_k = 0$. Case (3) is handled by lines 9–12, since $\gamma \neq \mathrm{expn}(C_k)$ and $C_k \neq 0$ by assumption. Hence, the behavior is correct. The loop terminates because there are only n_b elements in each stream, and lines 14–15 only add an element to the heap if there is a new element to add, while every iteration of the loop always removes an element from the heap at line 12. □

3. Division with Remainder

3.1. Naïve Division with Remainder

We now consider the problem of multivariate division with remainder, where the input polynomials are $a, b \in \mathbb{D}[x_1, \ldots, x_v]$, with $b \neq 0$ being the divisor and a the dividend. While this operation is well-defined for $a, b \in \mathbb{D}[x_1, \ldots, x_v]$ for an arbitrary integral domain $\mathbb{D}$, provided that $\mathrm{lc}(b)$ is a divisor of the content of both a and b, we rather assume, for simplicity, that the polynomials a and b are over a field. We can therefore specify this operation as having the inputs $a, b \in \mathbb{K}[x_1, \ldots, x_v]$, and outputs $q, r \in \mathbb{K}[x_1, \ldots, x_v]$, where q and r satisfy (We note due to its relevance for the algorithms presented in Section 5 that $\{b\}$ is a Gröbner basis of the ideal it generates and the stated condition here on the remainder r is equivalent to the condition that r is reduced with respect to the Gröbner basis $\{b\}$ (see [27] for further discussion of Gröbner bases and ideals)):

$$a = qb + r, \text{where } r = 0 \text{ or } \mathrm{lt}(b) \text{ does not divide any term in } r.$$

In an effort to achieve performance, we continue to be motivated by the idea of producing terms of the result (quotient and remainder) in sorted order. However, this is much trickier in the case of division in comparison to multiplication. We must compute terms of both the quotient and remainder in order, while simultaneously producing terms of the product qb in order. We must also produce these product terms while q is being generated term-by-term throughout the algorithm. This is not so simple, especially in implementation.

In the general "long division" of polynomials (see Section 2.4 of [28]) one repeatedly obtains the product of a newly computed quotient term and the divisor, and then updates the dividend with the difference between it and this product. Of course, this is computationally wasteful and not ideal, since at each step of this long division one needs only the leading term of this updated dividend to compute the next quotient term. Thus, before concerning ourselves with a heap-based algorithm, we consider a computationally efficient division algorithm which does not perform this continued updating of the dividend. This algorithm, which is a special case of the algorithm in Theorem 3 of Section 2.3 in [27] , is presented as Algorithm 3.

Algorithm 3 DIVIDEPOLYNOMIALS(a,b)

$a, b \in \mathbb{K}[x_1, \dots, x_v]$, $b \neq 0$; return $q, r \in \mathbb{K}[x_1, \dots, x_v]$ such that $a = qb + r$ where $r = 0$ or $\text{lt}(b)$ does not divide any term in r (r is reduced with respect to the Gröbner basis $\{b\}$).

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

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

Proposition 2. *Algorithm 3 terminates and is correct. ([27], pp. 61–63)*

3.2. Heap-Based Division with Remainder

It is clear from Algorithm 3 that multivariate division reduces to polynomial multiplication (through the product qb) and polynomial subtraction. What is not obvious is the efficient computation of the term $\tilde{r} = \text{lt}(a - qb - r)$. Nonetheless, we can again use heap-based multiplication to keep track of the product qb. The principal difference from multiplication, where all terms of both factors are known from the input, is that the terms of q are computed as the algorithm proceeds. This idea of using a heap to monitor $q \cdot b$ follows that of Johnson [24] for his exact univariate division. We extend his algorithm to multivariate division with remainder.

In terms of the wording of the multiplication algorithm, we set q to the multiplier and b to the multiplicand, distributing q over b, so the streams are formed from a single term of q, while the stream moves along b. By having q in this position it becomes relatively easy to add new streams into the computation as new terms of q are computed. Using the notations of our heap-division algorithm (Algorithm 4), the crucial difference between heap-based multiplication and heap-based division is that each stream does not start with $Q_\ell B_1$. Rather, the stream begins at $Q_\ell B_2$ since the product term $Q_\ell B_1$ is cancelled out by construction.

The management of the heap to compute the product qb uses several of the functions described for Algorithm 2. Specifically **heapPeek**(), **heapInsert**$(\cdot, \cdot)$, and **heapExtract**(). However, **heapExtract**() is modified slightly from its definition in multiplication. For division it combines removal of the top heap element *and* insertion of the next element of the stream (if there is a next) from which the top element originated. In this algorithm we use δ to denote the exponent of the top term in the heap of $q \cdot b$. Similar to multiplication, we abuse notation and let $\delta = -1$ if the heap is empty.

Finally, having settled the details of the product qb, what remains is to efficiently compute the leading term of $a - qb - r$. This is handled by a case discussion between the maximum term (in the sense of the term order) of a which has yet to be cancelled out and the maximum term of the product qb which has yet to be used to cancel out something. Then, by construction, when a newly generated term goes to the remainder it exactly cancels out one term of $a - qb$. This case discussion is evident in lines 4, 7, and 10 of Algorithm 4, while Proposition 3 formally proves the correctness of this approach.

Algorithm 4 HEAPDIVIDEPOLYNOMIALS(a,b)

$a, b \in \mathbb{K}[x_1, \ldots, x_v]$, $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i} = \sum_{i=1}^{n_a} A_i$, $b \neq 0 = \sum_{j=1}^{n_b} b_j X^{\beta_j} = \sum_{j=1}^{n_b} B_j$; return $q, r \in \mathbb{K}[x_1, \ldots, x_v]$ such that $a = qb + r$ where $r = 0$ or B_1 does not divide any term in r (r is reduced with respect to the Gröbner basis $\{b\}$).

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

Proposition 3. *Algorithm 4 terminates and is correct.*

Proof. Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[x_1, \ldots, x_v]$ with $\mathrm{tdeg}(b) > 0$. If $b \in \mathbb{K}$ then this degenerate case is simply a scalar multiplication by b_1^{-1} and proceeds as in Proposition 2. Then $r = 0$ and we are done. Otherwise, $\mathrm{tdeg}(b) > 0$ and we begin by initializing $q, r = 0$, $k = 1$ (index into a), $\ell = 0$ (index into q), and $\delta = -1$ (heap empty condition) since the heap is initially empty. The key change from Algorithm 3 to obtain Algorithm 4 is to use terms of qb obtained from the heap to compute $\tilde{r} = \mathrm{lt}(a - qb - r)$. There are then three cases to track: (1) $\tilde{r}$ is an uncancelled term of a; (2) $\tilde{r}$ is a term from $(a - r) - (qb)$, i.e., the degree of the greatest uncancelled term of a is the same as the degree of the leading term of qb; and (3) $\tilde{r}$ is a term of $-qb$ with the property that the rest of the terms of $a - r$ are smaller in the term order. Let $a_k X^{\alpha_k} = A_k$ be the greatest uncancelled term of a. The three cases then correspond to conditions on the ordering of δ and α_k. The term $\tilde{r}$ is an uncancelled term of a (Case 1) either if the heap is empty (meaning either that no terms of q have yet been computed or all terms of qb have been removed), or if $\delta > -1$ but $\delta < \alpha_k$. In either of these two situations $\delta < \alpha_k$ holds and $\tilde{r}$ is chosen to be A_k. The term $\tilde{r}$ is a term from the difference $(a - r) - (qb)$ (Case 2) if both A_k and the top term in the heap have the same exponent vector ($\delta = \alpha_k$). Lastly, $\tilde{r}$ is a term of $-qb$ (Case 3) whenever $\delta > \alpha_k$ holds. Algorithm 4 uses the above observation to compute $\tilde{r}$ by adding conditional statements to compare the components of δ and α_k. Terms are only removed from the heap when $\delta \geq \alpha_k$ holds, and thus we "consume" a term of qb. Simultaneously, when a term is removed from the heap, the next term from the given stream, if it exists, is added to the heap (by the definition of **heapExtract**()). The updating of q and r with the new leading term $\tilde{r}$ is almost the same as Algorithm 3, with the exception that when we update the quotient, we also initialize a new stream with Q_ℓ in the multiplication of $q \cdot b$. This stream is initialized with a head of $Q_\ell B_2$ because $Q_\ell B_1$, by construction, cancels a unique term of the expression $a - qb - r$. In all three cases, either the quotient is updated, or the remainder is updated. It follows from the case discussion of δ and α_k that the leading term of $a - qb - r$ is non-increasing for each loop iteration and the algorithm therefore terminates by Proposition 2. Correctness is implied by the condition that $\tilde{r} = 0$ at the end of the algorithm together with the fact that all terms of r satisfy the condition $\mathrm{lt}(b) \nmid R_k$. □

4. Pseudo-Division

4.1. Naïve Pseudo-Division

The pseudo-division algorithm is essentially a univariate operation. Accordingly, we denote polynomials and terms in this section as being elements of $\tilde{\mathbb{D}}[x_1, \ldots, x_{v-1}][x_v] = \mathbb{D}[x]$ for an arbitrary integral domain $\tilde{\mathbb{D}}$. It is important to note that while the algorithms and discussion in this section are specified for univariate polynomials they are, in general, multivariate polynomials, and thus the coefficients of these univariate polynomials are in general themselves multivariate polynomials.

Pseudo-division is essentially a fraction-free division: instead of dividing a by $h = \mathrm{lc}(b)$ (once for each term of the quotient q), a is multiplied by h to ensure that the polynomial division can occur without being concerned with divisibility limitations of the ground ring. The outputs of a pseudo-division operation are the pseudo-quotient q and pseudo-remainder r satisfying

$$h^\ell a = qb + r, \ \deg(r) < \deg(b), \tag{1}$$

where ℓ satisfies the inequality $0 \leq \ell \leq \deg(a) - \deg(b) + 1$. When $\ell < \deg(a) - \deg(b) + 1$ the pseudo-division operation is called *lazy* or *sparse*.

Under this definition, the simple multivariate division algorithm (Algorithm 3) can be readily modified for pseudo-division by accounting for the required factors of h. This enters in two places: (i) each time a term of a is used, we must multiply the current term A_k of a by h^ℓ, where ℓ is the number of quotient terms computed so far, and (ii) each time a quotient term is computed we must multiply all the previous quotient terms by h to ensure that $h^\ell a = qb + r$ will be satisfied. Algorithm 5 presents this basic pseudo-division algorithm modified from the simple multivariate division algorithm.

Algorithm 5 PSEUDODIVIDEPOLYNOMIALS(a,b)
$a, b \in \mathbb{D}[x]$, $b \neq 0$, $h = \mathrm{lc}(b)$; return $q, r \in \mathbb{D}[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r) < \deg(b)$.

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

It is important to note that because pseudo-division is univariate, all of the quotient terms are computed before any remainder terms are computed. This is because we can always carry out a pseudo-division step, and produce a new quotient term, provided that $\deg(b) \leq \deg(\mathrm{lt}(h^\ell a - qb - r))$, where $r = 0$. When $\deg(b) > \deg(\mathrm{lt}(h^\ell a - qb - r))$ then the quotient is done being computed and we have $r = h^\ell a - qb$, satisfying the conditions (1) of a pseudo-remainder. The following proposition proves the correctness of our pseudo-division algorithm.

Proposition 4. *Algorithm 5 terminates and is correct.*

Proof. Let $\mathbb{D}$ be an integral domain and let $a, b \in \mathbb{D}[x]$ with $\beta = \deg(b) > 0$. If $\deg(b) = 0$, $b = h$ and the divisibility test on line 4 always passes, all generated terms go to the quotient, and we get a remainder of 0 throughout the algorithm. Essentially this is a meaningless operation. q becomes $h^{n_a - 1}a$ and the formula (1) holds with $r = 0$ and the convention that $\deg(0) = -\infty$. We proceed assuming $\deg(b) > 0$. We initialize $q, r, \ell = 0$. It is enough to show that for each loop iteration, the degree of $\tilde{r}$ strictly decreases. Since the degree of $\tilde{r}$ is finite, $\tilde{r}$ is zero after finitely many iterations. We use superscripts to denote the values of the variables of Algorithm 5 on the i-th iteration. We have two possibilities for each i, depending on whether or not $x^\beta \mid \tilde{r}^{(i)}$ holds: (1) $Q_\ell = \tilde{r}^{(i)}/x^\beta$, Q_ℓ being a new quotient term; or (2) $R_k = \tilde{r}^{(i)},\ R_k$ being a new remainder term. In Case 1 we update only the quotient term so $r^{(i+1)} = r^{(i)}$; in Case 2 we update only the remainder term so $q^{(i+1)} = q^{(i)}$.

Suppose, then, that $\tilde{r}^{(i)}$ has just been used to compute a term of q or r, and we now look to compute $\tilde{r}^{(i+1)}$. Depending on whether or not $x^\beta \mid \tilde{r}^{(i)}$ we have:
Case 1: $x^\beta \mid \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $Q_\ell = \tilde{r}^{(i)}/x^\beta$. Here, because we are still computing quotient terms, $r^{(i+1)} = r^{(i)} = 0$. Thus,

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

In the second last line, where $r^{(i)} = 0$ appears, notice that since $\tilde{r}^{(i)} = \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $h \in \mathbb{D}$, we can ignore h for the purposes of choosing a term with highest degree and we have therefore that $\mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)} - \tilde{r}^{(i)}) < \mathrm{lt}(\tilde{r}^{(i)})$. Also, the expression $Q_\ell(b - B_1)$ has leading term $Q_\ell B_2$ which is strictly less than $\tilde{r}^{(i)} = Q_\ell x^\beta$, by the ordering of the terms of b. Hence $\tilde{r}^{(i+1)}$ is strictly less than $\tilde{r}^{(i)}$.
Case 2: $x^\beta \nmid \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$ and $R_k = \tilde{r}^{(i)}$

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

Similar to Case 1, $\tilde{r}^{(i)} = \mathrm{lt}(h^\ell a - q^{(i)}b - r^{(i)})$, thus the difference between $(h^\ell a - q^{(i)}b - r^{(i)})$ and $\tilde{r}^{(i)}$ must have a leading term strictly less than $\tilde{r}^{(i)}$. The loop therefore terminates. The correctness is implied by the condition that $\tilde{r} = 0$ at the end of the loop. The condition $\deg(r) < \deg(b)$ is met because the terms are only added to the remainder when $x^\beta \nmid \tilde{r}$ holds, i.e., when it is always the case that $\deg(h^\ell a - qb) < \deg(b)$. $\ell \leq \deg(a) - \deg(b) + 1$ holds because ℓ is only incremented when a new quotient term is produced (i.e., $x^\beta \mid \tilde{r}$) and the maximum number of quotient terms is $\deg(a) - \deg(b) + 1$. □

4.2. Heap-Based Pseudo-Division

Optimization of Algorithm 5 using a heap proceeds in much the same way as for division. The only additional concern to handle to reach Algorithm 6 is how to account for factors of h in the computation of $\mathrm{lt}(h^\ell a - qb - r)$. Handling this requires adding the same number of factors of h to A_k that have been added to the quotient up to a given iteration, that is, h^ℓ. The number ℓ is incremented when the previous quotient terms are multiplied by h prior to adding a new quotient term. Other than this, the changes to Algorithm 5 to reach Algorithm 6 follow exactly the analogous changes to Algorithm 3 to reach Algorithm 4. These observations therefore yield the following algorithm and proposition.

Algorithm 6 HEAPPSEUDODIVIDEPOLYNOMIALS(a,b)

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

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

Proposition 5. *Algorithm 6 terminates and is correct.*

Proof. Let $\mathbb{D}$ be an integral domain and $a, b \in \mathbb{D}[x]$ with $\deg(b) > 0$. If $b \in \mathbb{D}$ then this degenerate case proceeds as in Proposition 4. Then $r = 0$ with $\deg(r) = -\infty < 0 = \deg(b)$ and we are done. Observe that there are two main conditionals (lines 5–12 and 13–20) in the while loop. Given Proposition 4, it is enough to show that the first conditional computes $\mathrm{lt}(h^\ell a - qb - r)$ and the second uses $\tilde{r}$ to add terms to either q or r, depending on whether or not $x^\beta \mid \tilde{r}$. We initialize $q, r = 0$, $k = 1$ (index into a), $\ell = 0$ (index into q), $\delta = -1$ (heap empty condition) since the heap is initially empty. The central change to Algorithm 5 to reach Algorithm 6 is to take terms of qb from the heap to compute $\tilde{r} = \mathrm{lt}(h^\ell a - qb - r)$. Three cases must then be tracked: (1) $\tilde{r}$ is a term of $h^\ell a$ that has not yet been cancelled; (2) $\tilde{r}$ is a term from $(h^\ell a - r) - (qb)$; and (3) $\tilde{r}$ is a term of $-qb$ such that all remaining terms of $h^\ell a - r$ have smaller degree. Notice that all the terms of q are computed before the terms of r since this is essentially univariate division with respect to the monomials. Therefore, we can ignore r in the sub-expression $h^\ell a - r$. Thus, computing $\mathrm{lt}(h^\ell a - qb - r)$ in order simply requires computing terms of $(h^\ell a - qb)$ in order. These three cases for computing $\tilde{r}$ are handled by the first conditional. Let $a_k X^{\alpha_k} = A_k$ be the greatest uncancelled term of a. In Case 1, the heap is either empty (indicating that no terms of q have been computed yet or all terms of qb have been extracted) or $\deg(qb) = \delta > -1$ but $\delta < \alpha_k$. In either situation $\delta < \alpha_k$ holds and $\tilde{r}$ is chosen to be A_k. The term $\tilde{r}$ is a term from the difference $(h^\ell a - qb)$ (Case 2) if both A_k and the top term of the heap have the same degree ($\delta = \alpha_k$) and $\tilde{r}$ is chosen to be the difference of $h^\ell A_k$ and the greatest uncancelled term of qb. Lastly, $\tilde{r}$ is a term of $-qb$ (Case 3) in any other situation, i.e., $\delta > \alpha_k$. Thus, the first conditional computes $\mathrm{lt}(h^\ell a - qb - r)$, *provided* that the second conditional correctly adds terms to q and r. The second conditional adds terms to the quotient when $x^\beta \mid \mathrm{lt}(h^\ell a - qb)$ holds. Since each new quotient term adds another factor of h, we must first multiply all previous quotient terms by h. We then construct the new quotient term to cancel $\mathrm{lt}(h^\ell a - qb)$ by setting $Q_{\ell+1} = \mathrm{lt}(h^\ell a - qb)/x^\beta$, as in Algorithm 5. Since $Q_\ell B_1$ cancels a term of $(h^\ell a - qb)$ by construction, then line 18 initializing a new stream with $Q_\ell B_2$ is also correct. If, on the other hand, $x^\beta \nmid \mathrm{lt}(h^\ell a - qb)$, all remaining $\tilde{r}$ terms are remainder terms, which are correctly added by line 20. □

While the algorithmic shift between heap-based multivariate division (Algorithm 4) and heap-based pseudo-division (Algorithm 6) is very straight forward, the change of coefficient domain from simple numerical coefficients to full multivariate polynomials (when $\mathbb{D}$ is a polynomial ring) leads to many implementation challenges. This affects lines 6, 9 and 14 of Algorithm 6 in particular because they can involve multiplication of multivariate polynomials. These issues are discussed in Section 7.4.

5. Multi-Divisor Division and Pseudo-Division

One natural application of multivariate division with remainder is the computation of the normal form with respect to a Gröbner basis, which is a kind of multi-divisor division. Let $\mathbb{K}$ be a field and $\mathcal{B} = \{b_1, \ldots, b_k\}$ be a Gröbner basis with $b_j \in \mathbb{K}[x_1, \ldots, x_v]$ for $1 \leq i \leq k$. Then we can compute the normal form r of a polynomial $a \in \mathbb{K}[x_1, \ldots, x_v]$ (together with the quotients q_j) with respect to $\mathcal{B}$ by Algorithm 21.11 from [28], yielding $a = q_1 t_1 + \cdots + q_k t_k + r$, where r is reduced with respect to $\mathcal{B}$. This naïve normal form algorithm makes repeated calls to a multivariate division with remainder algorithm, thus we can take advantage of our optimized heap-based division (Algorithm 4).

We can offer algorithmic improvements in some cases where the set of divisors forms a triangular set, i.e., where the main variables of $t_j \in \{t_1, \ldots, t_k\}$ are pairwise different. Note that a triangular set $\mathsf{T} = \{t_1, \ldots, t_k\}$, with $t_j \in \mathbb{K}[x_1, \ldots, x_v]$ and $\mathrm{mvar}(t_k) > \cdots > \mathrm{mvar}(t_1)$, is called *normalized* if, for every polynomial of T, every variable appearing in its initial is free, i.e., is not the main variable of another polynomial of T. In the case where a normalized triangular set is also zero-dimensional (i.e., $k = v$) so that being normalized implies that $\mathrm{init}(t_i) \in \mathbb{K}$ holds, the triangular set T is actually a Gröbner basis for the ideal it generates.

For such zero-dimensional normalized (also known as Lazard) triangular sets it is possible to use a recursive algorithm (Algorithm 7) which is taken from [25]. Since the algorithm is recursive we appropriately use the recursive representation of the polynomials. If $v = 1$, the desired result is obtained by simply applying normal division with remainder. Otherwise the coefficients of a with respect to $x_v = \mathrm{mvar}(t_v)$ are polynomials belonging to $\mathbb{K}[x_1, \ldots, x_{v-1}]$ because T is a triangular set. The coefficients of a are reduced with respect to the set $\{t_1, t_2, \ldots, t_{v-1}\}$ by means of a recursive call, yielding a polynomial r. At this point, r is divided by t_v by applying the division algorithm. Since this operation can lead to an increase in degree of for the variables less than x_v, the coefficients of r are reduced with respect to $\{t_1, \ldots, t_{v-1}\}$ by means of a second recursive call.

Algorithm 7 TriangularSetNormalForm (a,T)

Given $a \in \mathbb{K}[x_1, \ldots, x_v]$, $\mathsf{T} = \{t_1, \ldots, t_v\} \subset \mathbb{K}[x_1, \ldots, x_v]$, with $x_1 = \mathrm{mvar}(t_1) < \cdots < x_v = \mathrm{mvar}(t_v)$ and $\mathrm{init}(t_1), \ldots, \mathrm{init}(t_v) \in \mathbb{K}$, returns $\mathbf{q} = \{q_1, \ldots, q_v\} \subset \mathbb{K}[x_1, \ldots, x_v]$ and $r \in \mathbb{K}[x_1, \ldots, x_v]$ such that $a = q_1 t_1 + \cdots + q_v t_v + r$, with r is reduced (in the Gröbner bas) with respect to the Lazard triangular set T.

```
1: if $v = 1$ then
2:     $(q_1, r) := \textsc{HeapDividePolynomials}(a, t_1)$
3: else
4:     for $i = 0$ to $\deg(a, x_v)$ do
5:         $(\mathbf{q}^{(i)} := \{q_1^{(i)}, \ldots, q_{v-1}^{(i)}\}, r^{(i)}) := \textsc{TriangularSetNormalForm}(\mathrm{coef}(a, x_v, i), \{t_1, \ldots, t_{v-1}\})$
6:     end for
7:     $\mathbf{q} := 0$
8:     $r := \sum_i r^{(i)} {x_v}^i$
9:     for $j = 1$ to $v - 1$ do
10:        $q_j := q_j + \sum_i q_j^{(i)} {x_v}^i$
11:    end for
12:    $(\tilde{q}, r) := \textsc{HeapDividePolynomials}(r, t_v)$; $q_v := q_v + \tilde{q}$
13:    for $i = 0$ to $\deg(r, x_v)$ do
14:        $(\mathbf{q}^{(i)} := \{q_1^{(i)}, \ldots, q_{v-1}^{(i)}\}, r^{(i)}) := \textsc{TriangularSetNormalForm}(\mathrm{coef}(r, x_v, i), \{t_1, \ldots, t_{v-1}\})$
15:    end for
16:    execute Lines 8–11
17: end if
18: return $(\mathbf{q}, r)$
```

Proposition 6. *Algorithm 7 terminates and is correct [25].*

This approach can be extended to pseudo-division of a polynomial by a triangular set, an operation that is important in triangular decomposition algorithms, in the case that the triangular set is normalized. The pseudo-remainder r and pseudo-quotients q_j of a polynomial $a \in \mathbb{K}[x_1, \ldots, x_v]$ pseudo-divided by a triangular set $\mathsf{T} = \{t_1, \ldots, t_k\}$ must satisfy

$$ha = q_1 t_1 + \cdots + q_k t_k + r, \qquad \deg(r, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j)) \text{ for } 1 \leq j \leq k, \tag{2}$$

where h is a product of powers of the initials (leading coefficients in the univariate sense) of the polynomials of T. If this condition is satisfied then r is said to be *reduced with respect to* T, again using the convention that $\deg(r) = -\infty$ if $r = 0$.

The pseudo-remainder r can be computed naïvely in k iterations where each iteration performs a single pseudo-division step with respect to each main variable in decreasing order $\mathrm{mvar}(t_k), \mathrm{mvar}(t_{k-1}), \ldots, \mathrm{mvar}(t_1)$. The remainder is initially set to a and is updated during each iteration. This naïve algorithm is inefficient for two reasons. First, since each pseudo-division step can increase the degree of lower variables in the order, if a is not already reduced with respect to T, the intermediate pseudo-remainders can experience significant coefficient swell. Second, it is inefficient in terms of data locality because each pseudo-division step requires performing operations on data distributed throughout the polynomial.

A less naïve approach is a recursive algorithm that replaces each of the k pseudo-division steps in the naïve algorithm with a recursive call, amounting to k iterations where multiple pseudo-division operations are performed at each step. This algorithm deals with the first inefficiency issue of coefficient swell, but still runs into the issue with data locality. To perform this operation more efficiently we conceive a recursive algorithm (Algorithm 8) based on the recursive normal form algorithm (Algorithm 7). Using a recursive call for each coefficient of the input polynomial a ensures that we work only on data stored locally, handling the second inefficiency of the naïve algorithm.

Algorithm 8 TriangularSetPseudoDivide (a, T)

Given $a, t_1, \ldots, t_k \in \mathbb{K}[x_1, \ldots, x_v]$, $\mathsf{T} = \{t_1, \ldots, t_k\}$, with $\mathrm{mvar}(t_1) < \cdots < \mathrm{mvar}(t_k)$ and $\mathrm{init}(t_j) \notin \{\mathrm{mvar}(t_i) \mid t_i \in \mathsf{T}\}$ for $1 \leq j \leq k$, returns $\mathbf{q} = \{q_1, \ldots, q_k\} \subset \mathbb{K}[x_1, \ldots, x_v]$ and $r, h \in \mathbb{K}[x_1, \ldots, x_v]$ such that $ha = q_1 t_1 + \cdots + q_k t_k + r$, where r is reduced with respect to T.

```
1:  if $k = 1$ then
2:      $(q_1, r, e) :=$ HeapPseudoDividePolynomials$(a, t_1)$; $h = \mathrm{init}(t_1)^e$
3:  else
4:      $x_m := \mathrm{mvar}(t_k)$
5:      for $i = 0$ to $\deg(a, x_m)$ do
6:          $(\mathbf{q}^{(i)} := \{q_1^{(i)}, \ldots, q_{k-1}^{(i)}\}, r^{(i)}, h^{(i)}) :=$ TriangularSetPseudoDivide$(\mathrm{coef}(a, x_m, i), \{t_1, \ldots, t_{k-1}\})$
7:      end for
8:      $\mathbf{q} = 0$
9:      $h_1 := \mathrm{lcm}(h^{(i)}), 0 \leq i \leq \deg(a, x_m)$
10:     $r := \sum_i (h_1 / h^{(i)})\, r^{(i)} x_m^i$
11:     for $j = 1$ to $k - 1$ do
12:         $q_j := q_j + \sum_i (h_1 / h^{(i)})\, q_j^{(i)} x_m^i$
13:     end for
14:     if $\mathrm{mvar}(r) = x_m$ then
15:         $(\tilde{q}, r, \tilde{e}) :=$ HeapPseudoDividePolynomials$(r, t_k)$
16:         $\tilde{h} = \mathrm{init}(t_k)^{\tilde{e}}$
17:         for $j = 1$ to $k - 1$ do
18:             $q_j := q_j \tilde{h}$
19:         end for
20:         $q_k := \tilde{q}$
21:         for $i = 0$ to $\deg(r, x_m)$ do
22:             $(\mathbf{q}^{(i)} := \{q_1^{(i)}, \ldots, q_{k-1}^{(i)}\}, r^{(i)}, h^{(i)}) :=$ TriangularSetPseudoDivide$(\mathrm{coef}(r, x_m, i), \{t_1, \ldots, t_{k-1}\})$
23:         end for
24:         $h_2 := \mathrm{lcm}(h^{(i)}), 0 \leq i \leq \deg(r, x_m)$
25:         for $j = 1$ to $k$ do
26:             $q_j := q_j h_2$
27:         end for
28:         execute Lines 9–13 with $h_2$ replacing $h_1$
29:         $h := h_1 \tilde{h} h_2$
30:     else
31:         $h := h_1$; $q_k = 0$
32:     end if
33: end if
34: return $(\mathbf{q}, r, h)$
```

Proposition 7. *Algorithm 8 terminates and is correct.*

Proof. The central difference between this algorithm and Algorithm 7 is the change from division to pseudo-division. By Proposition 5 the computed pseudo-remainders are reduced with respect to their divisor. The fact that the loops of recursive calls are all for a triangular set with one fewer variables ensures that the total number of recursive calls is finite, and the algorithm terminates. If $k = 1$, then Proposition 5 proves correctness of this algorithm, so assume that $k > 1$.

We must first show that lines 4–13 correctly reduce a with respect to the polynomials $\{t_1, \ldots, t_{k-1}\}$. Let $c_i = \mathrm{coef}(a, x_m, i)$, so $a = \sum_{i=0}^{\deg(a,x_m)} c_i x_m^i$. Assuming the correctness of the algorithm, the result of these recursive calls are $q_j^{(i)}$, $r^{(i)}$ and $h^{(i)}$ such that $h^{(i)} c_i = \sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}$, where $\deg(r^{(i)}, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j))$ and $h^{(i)} = \prod_{j=1}^{k-1} \mathrm{init}(t_j)^{e_j}$ for some non-negative integers e_j. It follows that $c_i = \left(\sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}\right) / h^{(i)}$. We seek a minimal h_1 such that $h_1 a = \sum_i h_1 c_i x_m^i = \sum_i (h_1/h^{(i)}) \left(\sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}\right) x_m^i$ is denominator-free, which is easily seen to be $\mathrm{lcm}(h^{(i)})$. This then satisfies the required relation of the form (2), with h_1 in place of h, by taking $q_j = \sum_i (h_1/h^{(i)}) q_j^{(i)} t_j x_m^i$ and $r = \sum_i (h_1/h^{(i)}) r_j^{(i)} x_m^i$. This follows from the conditions $\deg(r^{(i)}, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j))$ since h_1 contains none of the main variables of $\{t_1, \ldots, t_{k-1}\}$ because T is normalized.

If at this point $\mathrm{mvar}(r) \neq x_m$, then no further reduction needs to be done and the algorithm finishes with the correct result by returning $(q_1, \ldots, q_{k-1}, 0, r, h_1)$. This is handled by the else clause on lines 30 and 31 of the conditional on lines 14–32. If, on the other hand, $\mathrm{mvar}(r) = x_m$, we must reduce r with respect to t_k. Proposition 5 proves that after executing line 15, $\deg(r, \mathrm{mvar}(t_k)) < \deg(t_k, \mathrm{mvar}(t_k))$, and together with lines 16–20 implies that with the updated pseudo-quotients

$$\tilde{h} h_1 a = \sum_{j=1}^{k} q_j t_j + r. \tag{3}$$

Since the pseudo-division step at line 15 may increase the degrees of the variables of r less than x_m in the variable ordering, we must issue a second set of recursive calls to ensure that (2) is satisfied. Again, given the correctness of the algorithm, it follows that the result of the recursive calls on lines 21–23 taking as input $r = \sum_{i=0}^{\deg(r,x_m)} c_i x_m^i$, with $c_i = \mathrm{coef}(r, x_m, i)$, are $q_j^{(i)}$, $r^{(i)}$ and $h^{(i)}$ such that $h^{(i)} c_i = \sum_{j=1}^{k-1} q_j^{(i)} t_j + r^{(i)}$, where $\deg(r^{(i)}, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j))$. Combining these results as before and taking $h_2 = \mathrm{lcm}(h^{(i)})$ it follows that

$$h_2 r = \sum_{j=1}^{k-1} \tilde{q}_j t_j + \tilde{r} \tag{4}$$

satisfies a reduction condition of the form (2) with $\tilde{q} = \sum_i (h_2/h^{(i)}) q_j^{(i)} t_j x_m^i$ and $\tilde{r} = \sum_i (h_2/h^{(i)}) r_j^{(i)} x_m^i$, again because T is normalized. Multiplying (3) by h_2 and using Equation (4) yields $h_2 \tilde{h} h_1 a = \sum_{j=1}^{k} h_2 q_j t_j + h_2 r = \sum_{j=1}^{k} h_2 q_j t_j + \sum_{j=1}^{k-1} \tilde{q}_j t_j + \tilde{r} = \sum_{j=1}^{k-1} (h_2 q_j + \tilde{q}_j) t_j + h_2 q_k t_k + \tilde{r}$, which gives the correct conditions for updating the pseudo-quotients on lines 25–27, with the $\tilde{q}_j$ and $\tilde{r}$ computed at line 28. Now $\tilde{r}$ is reduced with respect to x_m because r is and with respect to $\mathrm{mvar}(t_1), \ldots, \mathrm{mvar}(t_{k-1})$ because of the above argument, so that the correct overall multiplier is $h = h_2 \tilde{h} h_1$, set on line 29. The algorithm is therefore correct. □

6. Data Structures

Polynomial arithmetic is fundamental to so many algorithms that it should naturally be optimized as much as possible. Although algorithm choice is important for this, so too is making use of appropriate data structures. When programming for modern computer architectures we must be concerned with the processor–memory gap: the exponentially increasing difference between processor speeds and memory-access time. We combat this gap with judicial memory usage and management. In particular, the *principle of locality* and cache complexity describe how to obtain performance by maximizing memory accesses that make best use of modern memory hierarchies (i.e., data locality). Basically, this means that the same memory address should be accessed frequently or, at the very least, accesses should be adjacent to those most recently accessed. Our implementation adheres to this principle through the use of memory-efficient data structures with optimal data locality. We see later (in Section 7) that our algorithms have implementation-specific optimizations to exploit this locality and minimize cache complexity.

This section begins by reviewing our memory-efficient data structures for both sparse distributed (Section 6.1) polynomials and sparse recursive polynomials (Section 6.2). The latter is interesting as the data structure is still flat and distributed but allows for the polynomial to be viewed recursively. Then, we discuss the implementation of our heap data structure (Section 6.3) which is specialized and optimized for use in polynomial multiplication.

6.1. A Sparse Distributed Polynomial Data Structure

The most simple and common scheme for sparsely representing a polynomial is a linked list, or some similar variation of data blocks linked together by pointers [6,29,30]. This representation allows for very easy manipulation of terms using simple pointer manipulation. However, the indirection created by pointers can lead to poor locality while the pointers themselves must occupy memory, resulting in memory wasted to encode the structure rather than the data itself. More efficient sparse data structures have been explored by Gastineau and Laskar [29], where *burst tries* store monomials in the TRIP computer algebra system, and Monagan and Peace [30], where the so-called POLY data structure for MAPLE closely stores monomials in a dense array. In both cases, the multi-precision coefficients corresponding to those monomials are stored in a secondary structure and accessed by either indices stored alongside the monomials (in the case of TRIP) or pointers (in the case of MAPLE).

Our distributed polynomial representation stores both coefficients and monomials side-by-side in the same array. This representation, aptly named an *alternating array*, improves upon data locality; the coefficient and monomial which together make a single polynomial term are optimally local with respect to each other. This decision is motivated by the fact that in arithmetic, coefficients are accessed alongside their associated monomials (say to perform a multiplication or combine like terms). In practice, this array structure is augmented by a simple C-struct holding three items: the number of terms; the number of variables; and a pointer to the actual array. This seemingly simple structure abstracts away some complexities in both the coefficients and monomials. We begin with the coefficients.

Due to the nature of arbitrary-precision coefficients, in our case either integers or rational numbers (We actually have two nearly identical yet distinct alternating array implementations. One implementation holds integer coefficients while the other holds rational number coefficients), we cannot say they are fully stored in the array. We make use of the GNU Multiple Precision Arithmetic (GMP) Library [31] for our coefficients. The implementation of arbitrary-precision numbers in this library is broken into two distinct parts, which we will call the *head* and the *tree*. The head contains metadata about the tree, as well as a pointer to the tree, while the tree itself is what holds the numerical data. By the design of the GMP library users only ever interact with the head. Thus, our alternating array representation holds the heads of the GMP numbers directly in the array rather than pointers or indices to some other structure, which in turn would hold the heads of the GMP numbers. Figure 1 depicts an arbitrary polynomial of n terms stored in an alternating array, highlighting the GMP tree structure.

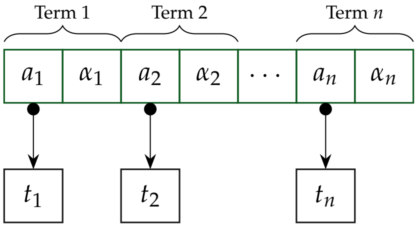


Figure 1. An alternating array representation of n terms showing GMP trees as $t_1, t_2, \ldots, t_n$, GMP heads as $a_1, a_2, \ldots, a_n$, and monomials as $\alpha_1, \alpha_2, \ldots, \alpha_n$. One head and tree together make a single arbitrary-precision number.

The alternating array diagram in Figure 1 may be misleading at first glance, since it appears that pointers are still being used; however, these pointers are completely internal to GMP and are unavoidable. Hence, where other structures use indices ([29], Figure 2) or pointers ([6], Figure 3) to a separate array of GMP coefficients, that coefficient array also further contains these pointers to GMP trees. Our implementation thus removes one level of indirection compared to these other schemes. We do note, however, that the data structure described in [6,30] includes an additional feature which automatically makes use of machine-precision integers stored directly in the data structure, rather than GMP integers, if coefficients are small enough.

Next, we discuss the implementation of monomials. Under a fixed variable ordering it becomes unnecessary to store the variables themselves with the monomial, and so we only store the exponent vector. This greatly reduces the required memory for a monomial. However, even more memory is saved via *exponent packing*. Using bit-masks and bit-shifts, multiple partial degrees, each taking a small non-negative value, can easily be stored in a single machine word (usually 64 bits). This fact should be obvious by looking at the binary representation of a non-negative integer on a computer. Integers are stored in some fixed size, typically 32 or 64 bits, and, when positive or unsigned, have many leading 0 bits. For example, 5 as a 32-bit integer is $0b00000000000000000000000000000101$. By using a predetermined number of bits for each partial degree in an exponent vector, it becomes easy to partition the 64 bits to hold many integers. Our alternating array thus holds a single machine word directly in the array for packing each exponent vector.

Exponent packing has been in use at least since the 60s in ALTRAN [1], but also in more recent works such as [4,32]. Our implementation differs from others in that exponents are packed unevenly, i.e., each exponent is given a different number of bits in which to be encoded. This is motivated by two factors. First, 64 bits is rarely evenly divided among the number of variables, meaning some bits could be wasted. Second, throughout the process of operations such as pseudo-division or triangular decomposition the degrees of lower-ordered variables often increase more drastically than higher-ordered variables, and so we give more bits to the lower-ordered variables. This can allow for large computations to progress further without failing or having to revert to an *unpacked* exponent vector. One final highlight on exponent packing (first emphasized in [32]) is that monomial comparisons and monomial multiplications respectively reduce to a single machine-integer comparison and a single machine-integer addition. This result drastically reduces the time to complete monomial comparisons, and thus sort monomials, a huge part of sparse polynomial arithmetic.

6.2. A Sparse Polynomial Data Structure for Viewing Polynomials Recursively

We take this section to describe our recursive polynomial data structure. That is not to say that the data structure itself is recursive, rather the polynomial is viewed recursively, as a univariate polynomial with multivariate polynomial coefficients. In general, polynomials are stored using the distributed representation; however, some operations, such as pseudo-division, require a specifically univariate view of the polynomial. Thus, we have created an in-place, very fast conversion between the distributed and recursive representations, amounting to minimal overhead in both memory usage and time. As a result, we can use the same distributed representation everywhere, only converting as required. This recursive representation is shown in Figure 2.

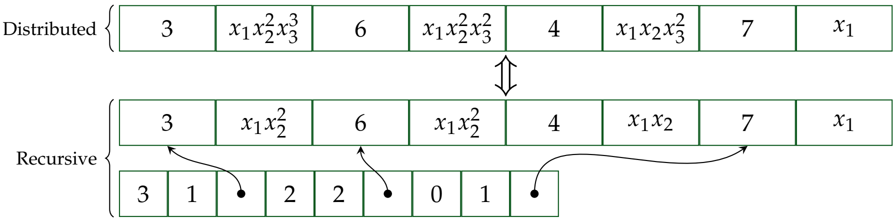


Figure 2. A distributed polynomial representation and its corresponding recursive polynomial representation, showing the additional secondary array. The secondary array alternates between: (1) degree of the main variable, (2) size of the coefficient polynomial, and (3) a pointer to the coefficient polynomial,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

To view a polynomial recursively we begin by (conceptually) partitioning its terms into blocks based on the degree of the main (highest-ordered) variable. Since our polynomials are stored using a lexicographical term order, the terms of the polynomial are already sorted based on the degree of the main variable. Moreover, terms within the same block are already stored in lexicographical order with respect to the remaining variables. Therefore, each block will act as a multivariate polynomial coefficient of the univariate polynomial in the main variable. The partitioning is done in-place, without any memory movement, simply by maintaining an offset into the alternating array which signifies the beginning of a particular coefficient, in the recursive sense.

We create a secondary auxiliary array which holds these offsets, the number of terms in each polynomial coefficient, and the degree of the main variable. Simultaneously, the degree of the main variable in the original alternating array is set to 0. The degree of the main variable then does not pollute the polynomial coefficient arithmetic. This secondary array results in minimal overhead, particularly because its size is proportional to only the number of unique values of the degree of the main variable. Figure 2 highlights this secondary array as part of the recursive structure.

6.3. Heaps Optimized for Polynomial Multiplication

The largest effort required of our sparse multiplication algorithm (and thus also that of our division and pseudo-division algorithms) is to sort the terms of the product. Our algorithm makes use of a heap to do this sorting (much like heap sort), and thus arithmetic performance is largely dependent in the performance of the heap. Briefly, a heap is a data structure for efficiently obtaining the maximum (or minimum) from a continually updating collection of elements. This is achieved by using a binary tree, which stores key-value pairs, with a special *heap property*—children are always less than their parents in the tree. A more complete discussion of heaps can be found in ([26], Section 2.4).

The optimizations used in our heap implementation focus on two aspects, minimizing the working memory of the heap and minimizing the number of comparisons. The need for the latter should be obvious, while the need for the former is more subtle. Due to the encoding of a heap as a binary tree, parent nodes and child nodes are not adjacent to each other in memory; the heap must essentially perform random memory accesses across all its elements. In the sense of locality and cache usage, this is not ideal, yet unavoidable. Therefore, we look to minimize the size of the heap in hopes that it will entirely fit in cache and allow for quick access to all its elements.

The first optimization is due to [33] which reduces the number of comparisons required to remove the maximum element of the heap by a factor of two. The usual implementation of a heap removes the root node, swapping a leaf node into the hole, and then filtering it downward to re-establish the heap property. This requires two comparisons per level to determine which path to travel down. Instead, one can continuously promote the larger of the hole's two children until the hole is a leaf node. This requires only one comparison per level.

The second optimization called *chaining* reduces both the required number of comparisons and the amount of working memory for the heap. This technique is common in the implementation of hash tables for *conflict resolution* ([26], Chapter 3). Whenever a “conflict” occurs (when two elements are found to be equal) they form a *chain*, or linked list. Each node’s key remains in the heap, but the values are now linked lists. Elements found to be equal simply add their value to the chain rather than insert a new element. This minimizes the number of elements in the heap but also allows extracting an entire chain, and therefore many elements, at the cost of removing a single element. This heap organization is presented in Figure 3.

In the context of polynomial multiplication, the exponent vector of the product term is the key while the value is a linked list of coefficients of the product. For our multiplication algorithm (Algorithm 2) we must also know from which stream a particular product term originated, and so should also store the stream index. However, to minimize the space required for the heap, while also storing the stream index (i.e., the multiplier term’s index), we do not store the product term’s coefficient at all and instead store the indices of the multiplier and multiplicand terms which together would produce a particular product term’s coefficient. We do not need the coefficient of the product term to do the sorting, and so storing indices is more efficient. Moreover, delaying the multiplication of coefficients has benefits for locality. With chaining, removing the maximum element actually removes an entire chain of like terms, then the coefficient multiplication and addition of like terms can be done simultaneously.

Similar heap optimizations, including chaining, have been used in [6]. In contrast with our implementation, chaining in [32] used pointers to multiplier and multiplicand terms rather than indices. Integer indices (32 bits) are twice as efficient in memory usage as pointers on 64-bit machines, improving the overall memory and cache usage of the heap (and multiplication in general).

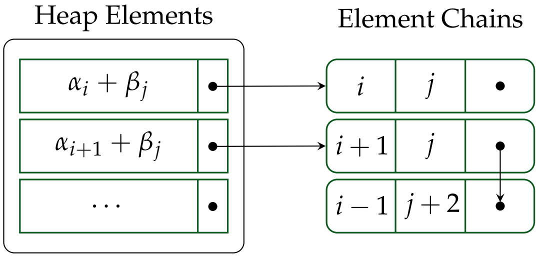


Figure 3. A heap of product terms, showing element chaining and index-based storing of coefficients. In this case, terms $A_{i+1} \cdot B_j$ and $A_{i-1} \cdot B_{j+2}$ have equal monomials and are chained together.

7. Implementation

As discussed in the previous section, our data structures are memory-efficient with exceptional data locality. Now, in this section, we describe the implementation-specific optimizations of our algorithms, such as memory management techniques and efficient use of our data structures. These implementations exploit the locality of the data structures to minimize cache complexity and improve performance. Formal cache complexity estimates of these algorithms are presented in [34]; we exclude them here and instead focus on motivations and techniques for reducing cache complexity in general.

We begin in Section 7.1 describing how to exploit our data structure for an optimized “in-place” addition (or subtraction) operation. Next, we discuss our implementations of multiplication (Section 7.2), division with remainder (Section 7.3), and pseudo-division (Section 7.4), all based on our heap data structure described above (Section 6.3). Lastly, we examine the application of these operations in our implementation of normal form and pseudo-division by a triangular set (Section 7.5).

7.1. In-Place Addition and Subtraction

An “in-place” algorithm suggests that the result is stored back into the same data structure as one of operands (or the only operand). This strategy is often motivated by either limited available memory resources or working with data that is too large to consider making a complete copy for the result. For our purposes, we are concerned with neither of these since our polynomial representations use relatively small amounts of memory. Hence, in-place operations are only of interest if they can improve running time. Generally speaking, in-place algorithms require more operations and more movement of data than out-of-place alternatives, making them most useful when the data set being sorted is so large that a copy cannot be afforded. For example, in-place merge sort has been a topic of discussion for decades, however, these implementations run 25–200% *slower* than an out-of-place implementation [35–37].

Due to the similarities between merge sort and polynomial addition (subtraction) it would seem unlikely that an in-place scheme would lead to performance benefits. However, our in-place addition becomes increasingly faster than out-of-place addition as coefficient sizes increase. This in-place addition scheme is not technically in-place, but it does exploit the structure of GMP numbers (as shown in Figure 1) for in-place coefficient arithmetic. In-place addition builds the resulting polynomial out-of-place but reuses the GMP *trees* of one of the operand polynomials. Rather than allocating a new GMP number—and thus a new GMP tree—in the resulting polynomial, we simply copy the head of one GMP number (and the pointer to its existing tree) into the new polynomial’s alternating array, performing the coefficient arithmetic in-place. This saves on memory allocation and memory copying, and benefits from the improved performance of GMP when using in-place arithmetic ([31], Section 3.11).

These surprising results are highlighted in Figure 4 where out-of-place addition and its in-place counterpart are compared for various polynomial sizes with varying coefficient sizes. In-place addition has a speed-up factor of up to 3 for the coefficient sizes tested, with continued improvements as coefficient sizes grow larger. In-place arithmetic is put to use in pseudo-division to reduce the cost of polynomial coefficient arithmetic and improve the performance of pseudo-division itself. See Section 7.4 for this discussion.

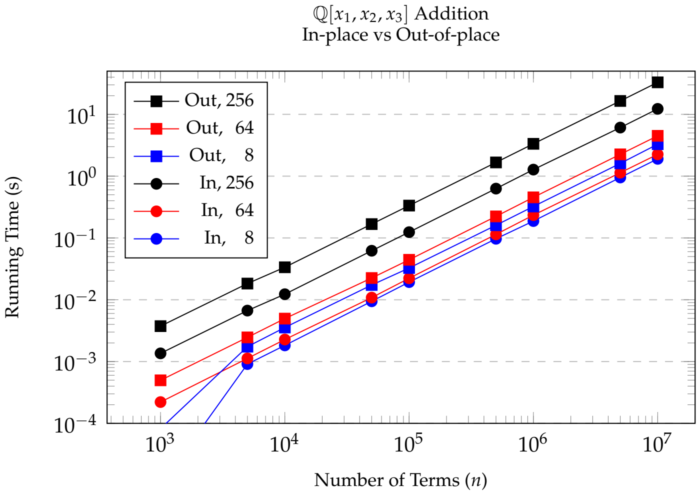


Figure 4. Comparing in-place and out-of-place polynomial addition. Random rational number polynomials in 3 variables are added together for various numbers of terms and for various coefficient sizes. The number of bits needed to encode the coefficients of the operands are shown in the legend. Notice this is a log-log plot.

7.2. Multiplication

The algorithm for polynomial multiplication (Algorithm 2) translates to code quite directly. However, we note some important implementation details to obtain better performance. Apart from the optimizations within the heap itself there are some implementation details concerning how the heap is used within multiplication to improve performance.

The first optimization makes use of the fact that multiplication is a commutative operation. Since the number of elements in the heap is equal to the number of streams, which is in turn equal to the number of terms in the multiplier (the factor a in $a \cdot b$), then we choose the multiplier to be the smaller operand, minimizing the size of the heap. The second optimization deals with the initialization of the heap. Due to the fact that for two terms A_i and B_j, $A_i \cdot B_j$ is always greater than $A_{i+1} \cdot B_j$ in the term order, then at the beginning of the multiplication algorithm it is only necessary to insert the term $A_{i+1} \cdot B_1$ after the term $A_i \cdot B_1$ has been removed from the heap.

A final optimization for multiplication deals with memory management. In particular, we know that for operands with n_a and n_b terms each, the maximum size of the product is $n_a \cdot n_b$. Therefore, we can pre-allocate this maximal size of the product (and similarly pre-allocate a maximal size n_a for the heap) before we begin computation. This eliminates any need for reallocation or memory movement, which can cause slowdowns. However, in the case where $n_a \cdot n_b$ is a very large number, say, exceeding 100 million, then we begin by only allocating 100 million terms for the product, doubling the allocation as needed in order amortize the cost of reallocation. Of course, any memory allocated in excess is freed at the end of the algorithm.

7.3. Division with Remainder

Polynomial division is essentially a direct application of polynomial multiplication. Again, we use heaps, with all the optimizations previously discussed, to produce the terms of the quotient-divisor product efficiently and in order. However, one important difference between division and multiplication is that the one of the operands of the quotient-divisor product, the quotient, is simultaneously being produced and consumed throughout the algorithm. Thus, we cannot pre-allocate space for the product or heap since the size of the quotient is unknown. Instead, we again follow a doubling of allocation strategy for the quotient and remainder to amortize the cost of reallocation. Moreover, we reallocate the space for the heap whenever we reallocate q since we know that the heap's maximum size will be equal to the number of terms in q. The added benefit of this is that the heap is guaranteed to have enough space to store all elements and does not need to check for overflow on each insert.

7.4. Pseudo-Division

As seen in Section 4.1 the algorithms for division (Algorithms 3 and 4) can easily be adapted to pseudo-division (Algorithms 5 and 6) by multiplying the dividend and quotient by the divisor's initial. However, the implementation between these two algorithms is very different. In essence, pseudo-division is a univariate operation, viewing the input multivariate polynomials recursively. That is, the dividend and divisor are seen as univariate polynomials over some arbitrary (polynomial) integral domain. Therefore, coefficients can be, and indeed are, entire polynomials themselves. Coefficient arithmetic becomes non-trivial. Moreover, the normal distributed polynomial representation would be inefficient to traverse and manipulate in this recursive way. Therefore, we use the recursive polynomial representation described in Section 6.1 with minimal overhead for conversion.

One of the largest performance concerns in this recursive view is the non-trivial coefficient arithmetic. As coefficients are now full polynomials there is more overhead in manipulating them and performing arithmetic. One important implementation detail is to perform the addition (and subtraction) of like terms in-place. Such combinations occur when computing the leading term of $h^{\ell}a - qb$ and when combining like terms in the quotient-divisor product. In-place addition, as described in Section 7.1, performs exceedingly better than out-of-place addition as the size of numerical coefficients grows, which occurs drastically during pseudo-division.

Similarly, the update of the quotient by multiplying by the initial of the divisor requires a multiplication of full polynomials. If we wish to save on memory movement we should perform this multiplication in-place as well. However, in our recursive representation (Figure 2), coefficient polynomials are tightly packed in a continuous array. To modify them in-place would require shifting all the following coefficients down the array to make room for the strictly large product polynomial. To avoid this unnecessary memory movement we modify the recursive data structure exclusively for the quotient polynomial; we break the continuous array of coefficients into many arrays, one for each coefficient. This allows them to grow without displacing the following coefficients. At the end of the algorithm, once the quotient has finished being produced, we collect and compact all of these disjoint polynomials into a single, packed array. In contrast, the remainder is never updated once its terms are produced, nor does it need to be viewed recursively, thus it is stored directly in the normal distributed representation, avoiding the unnecessary conversion out of the recursive representation.

7.5. Multi-Divisor (Pseudo-)Division

The performance of our normal form and multi-divisor pseudo-division algorithms primarily relies on the performance of the basic operations of division and pseudo-division, respectively. Hence, our normal form and multi-divisor pseudo-division algorithms gain significant performance benefits from the optimization of these lower-level operations. We only note two particular implementation details for these multi-divisor algorithms.

Firstly, the algorithms for normal form (Algorithm 7) and triangular set pseudo-division (Algorithm 8) use distributed and recursive polynomial representations, respectively, to manipulate operand polynomials appropriately for their operations. Secondly, we use in-place techniques, following the scheme of in-place addition (Section 7.1) to reduce the effects of GMP arithmetic and memory movement. Due to the recursive nature of these algorithms we can use a pre-allocation of memory as a destination to store both the final remainder and the remainder in each recursive call, continually reusing GMP coefficients.

8. Experimentation and Discussion

As we have seen in the previous two sections, our implementation has focused well on locality and memory usage in interest of obtaining performance. Indeed, as a result of the processor–memory gap this is highly important on modern architectures. The experimentation and benchmarks provided in this section substantiate our efforts where we will compare similar heap-based arithmetic algorithms provided in MAPLE [38].

Let us begin with a discussion on the quantification of sparsity with respect to polynomials. For univariate polynomials, sparsity can easily be defined as the maximum degree difference between any two successive non-zero terms. However, in the multivariate case, and in particular using lex ordering, there are infinitely many polynomial terms between x_1 and x_2, in the form of x_1^i. For multivariate polynomial, sparsity is less easily defined. Inspired by Kronecker substitution ([28], Section 8.4) we propose the following sparsity measure for multivariate polynomials adapted from the univariate case. Let $f \in R[x_1, \ldots, x_v]$ be non-zero and define $r = \max(\deg(f, x_i), 1 \leq i \leq v) + 1$. Then, every exponent vector $e := (e_1, \ldots, e_v)$ of a term of f can be viewed as an integer in a radix-r representation, $e_1 + e_2 r + \cdots + e_v r^{v-1}$. Viewing any two successive polynomial terms in f, say F_i and F_{i+1}, as integers in this radix-r representation, say c_i and c_{i+1}, we call the *sparsity* of f the smallest integer which is larger than the maximum value of $c_i - c_{i+1}$, for $1 \leq i < n_f$.

Our experimentation uses randomly generated sparse polynomials whose generation is parameterized by several variables: the number of variables v, the number of terms n, the number of bits used to encode any coefficient (denoted coefficient bound), and a sparsity value s used to compute the radix $r = \lfloor \sqrt[v]{s \cdot n} \rfloor$ for use in generating exponent vectors as just defined. Our arithmetic algorithms, and code for generating random test instances, are openly available in the BPAS library [12].

We compare our arithmetic implementations against MAPLE for both integer polynomials and rational number polynomials. Thanks to the work by Monagan and Pearce [6,18,19,32] in recent years MAPLE has become the leader in integer polynomial arithmetic. Benchmarks there clearly show that their implementation outperforms many others including that of TRIP [39], MAGMA [40], SINGULAR [41], and PARI/GP [42]. Moreover, other common libraries like FLINT [43] and NTL [44] provide only univariate polynomials, so to compare our multivariate implementation against theirs would be unfair. Hence, we compare against MAPLE with its leading high-performance implementation. In particular, MAPLE 2017 with `kernelopts(numcpus = 1)` (which forces MAPLE to run serially.) Of course, the parallel arithmetic of MAPLE described in [18,19], which has been shown to achieve up to 17x parallel speed-up, could out-perform our serial implementation in some cases, such as multiplication and division over $\mathbb{Z}$. However, to be fair, we compare serial against serial.

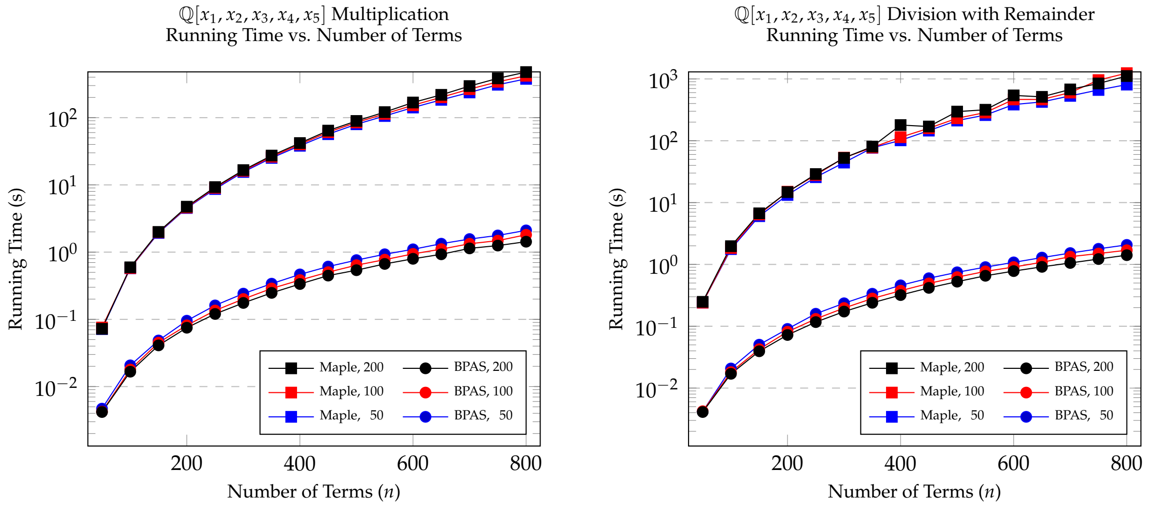
Our benchmarks were collected on a machine running Ubuntu 14.04 using an Intel Xeon X560 processor (Intel, Santa Clara, CA, USA) at 2.67 GHz with 32 KB L1 data cache, 256 KB L2 cache, and 12288 KB L3 cache, with 12 × 4 GB of DDR3 RAM at 1333 MHz. In all the following timings we present the median time among 3 trials using 3 different sets of randomly generated input polynomials.

8.1. Multiplication and Division with Remainder

We compare multiplication and division with remainder against MAPLE for both polynomials over the rational numbers and the integers. For multiplication we call `expand` in MAPLE and for division with remainder we call `Groebner:-NormalForm`. Normal form is a more general algorithm for many divisors but reduces to division with remainder in the case of a single divisor. This operation appears to be the only documented algorithm for computing division with remainder in MAPLE. The optimized integer polynomial *exact* division of [6] appears in MAPLE as the `divide` operation. It would be unfair to use our division with remainder algorithm to compute exact divisions to compare against [6] directly (although, some examples of such are shown in Section 8.4). However, internally, `Groebner:-NormalForm` clears contents to work, at least temporarily, with integer polynomials for calls to `divide` and `expand` for division and multiplication operations, respectively, each of which is indeed optimized (Contents are cleared from (rational number) polynomials, to result in an integer polynomial, either via the basis computation or directly in the call to the underlying normal form function `Groebner:-Buchberger:-nfprocs`).

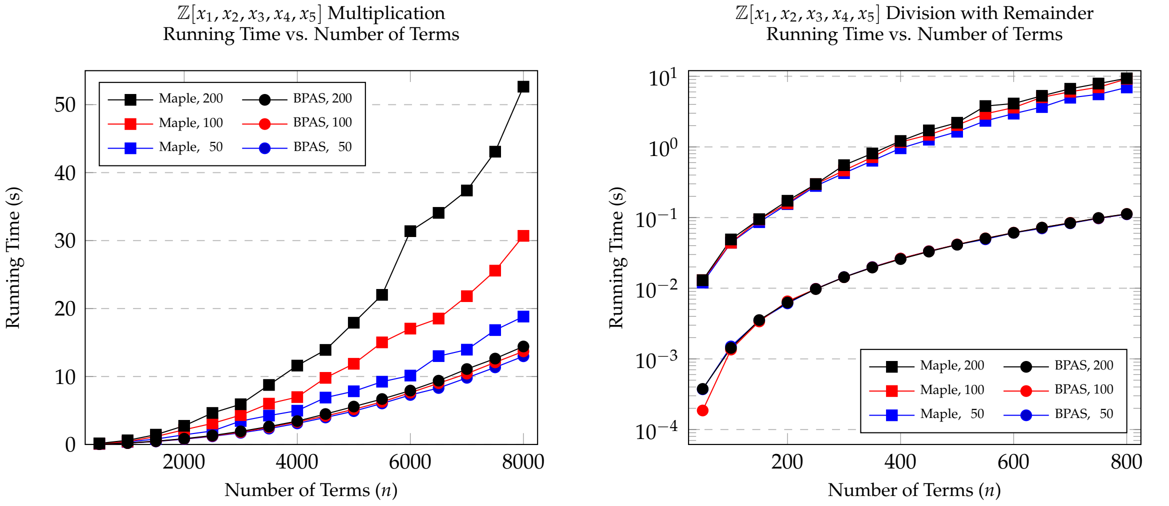
We begin by comparing our multiplication and division with remainder algorithms for polynomials over the rationals. MAPLE does not have an optimized data structure for polynomials with rational number coefficients [30], so this benchmark is meant to highlight the necessity of memory-efficient data structures for algorithmic performance. The plot in Figure 5a shows the performance of multiplication over $\mathbb{Q}$ for polynomials in 5 variables of varying sparsity and number of terms. The parameters specified determine how both the multiplier and multiplicand where randomly generated. The plot in Figure 5b shows the performance of division with remainder over $\mathbb{Q}$ for polynomials in 5 variables of varying sparsity and number of terms. For this division, we construct two polynomials f and g using the parameters specified and then perform the division $(f \cdot g + f)/g$. The disparity in running times between BPAS and MAPLE is very apparent, with multiple orders of magnitude separating the two. We see speed-ups of 333 for multiplication and 731 for division with remainder.

The same set of experiments were performed again for integer polynomials. Figure 6a,b shows multiplication and division with remainder, respectively, for polynomials over $\mathbb{Z}$. In this case, MAPLE features a more optimized data structure for polynomials over $\mathbb{Z}$ and performs relatively much better. However, BPAS still outperforms MAPLE with a speed-up factor of up to 3 for multiplication and 67 for division with remainder. The speed-up factors continue to grow as sparsity increases. This growth can be attributed to the fact that as sparsity increases, the number of like terms produced during a multiplication decreases. Hence, there is less coefficient arithmetic and many more terms in the product, highlighting the effects of better locality and memory management.



(**a**) Multiplication. (**b**) Division with remainder. $(fg + f)/g$ is performed.

Figure 5. Comparing multiplication and division with remainder over $\mathbb{Q}$. Polynomials are in 5 variables and the coefficient bound is 128. The sparsity varies as noted in the legend.

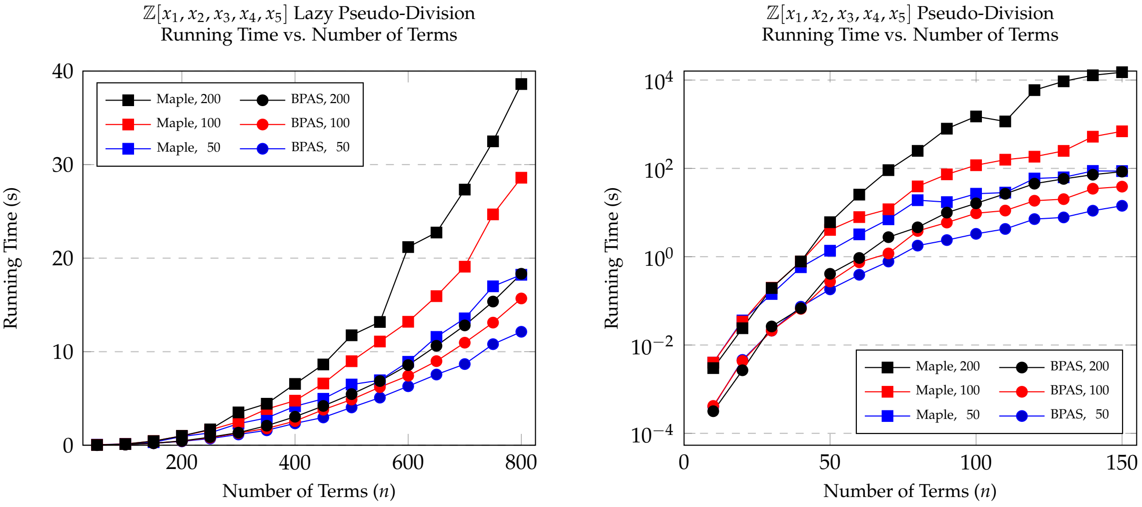


(**a**) Multiplication. (**b**) Division with remainder. $(fg + f)/g$ is performed.

Figure 6. Comparing multiplication and division with remainder over $\mathbb{Z}$. Polynomials are in 5 variables and the coefficient bound is 128. The sparsity varies as noted in the legend.

8.2. Pseudo-Division

We next compare the implementations of pseudo-division over $\mathbb{Z}$. We perform the pseudo-division of $(f \cdot g + f)$ by g for randomly generated f and g. However, since pseudo-division is essentially univariate, the randomly generated polynomials go through a secondary cleaning phase where the degree of the main variable is spread out evenly such that each polynomial coefficient, in the recursive sense, is the same size. This stabilizes the running time for randomly generated polynomials with the same number of terms. Figure 7b shows the running time of non-lazy pseudo-division, that is, ℓ is forced to be $\deg(a) - \deg(b) + 1$ in the pseudo-division equation $h^{\ell}a = qb + r$. Figure 7a shows a lazy pseudo-division, where ℓ is only as large as is needed to perform the pseudo-division. For lazy pseudo-division we see a speed-up factor of up to 2 while for non-lazy pseudo-division we see a speed-up factor of up to 178. A non-lazy pseudo-division's running time is usually dominated by coefficient polynomial arithmetic and performs much slower than the lazy version. Moreover, the gap between BPAS and MAPLE is much greater for non-lazy pseudo-division; increasing sparsity became a big problem in MAPLE, taking several hours to perform a single pseudo-division. Again, an increase in sparsity creates an increase in the number of terms in a polynomial product. Therefore, with our efficient memory management and use of data structures, increasing sparsity has little effect on our performance, in contrast to that of MAPLE. In MAPLE we call `prem` and `sprem` for non-lazy and lazy pseudo-division, respectively.

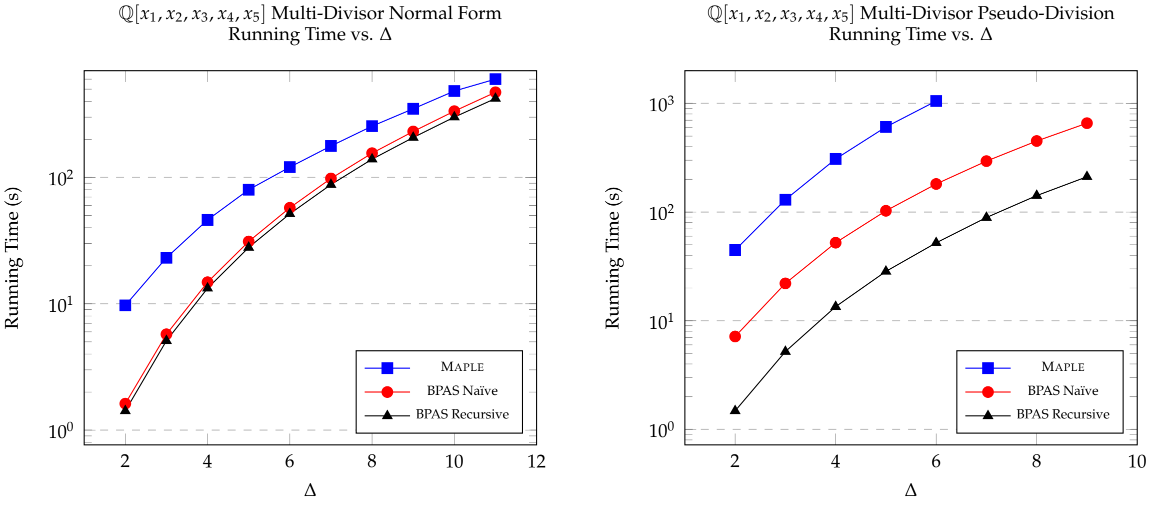


(**a**) Lazy pseudo-division of $fg + f$ by g. (**b**) Non-lazy pseudo-division of $fg + f$ by g.

Figure 7. Comparing lazy and non-lazy pseudo-division over $\mathbb{Z}$. Polynomials are in 5 variables and the coefficient bound is 128. The sparsity varies as noted in the legend.

8.3. Multi-Divisor Division and Pseudo-Division

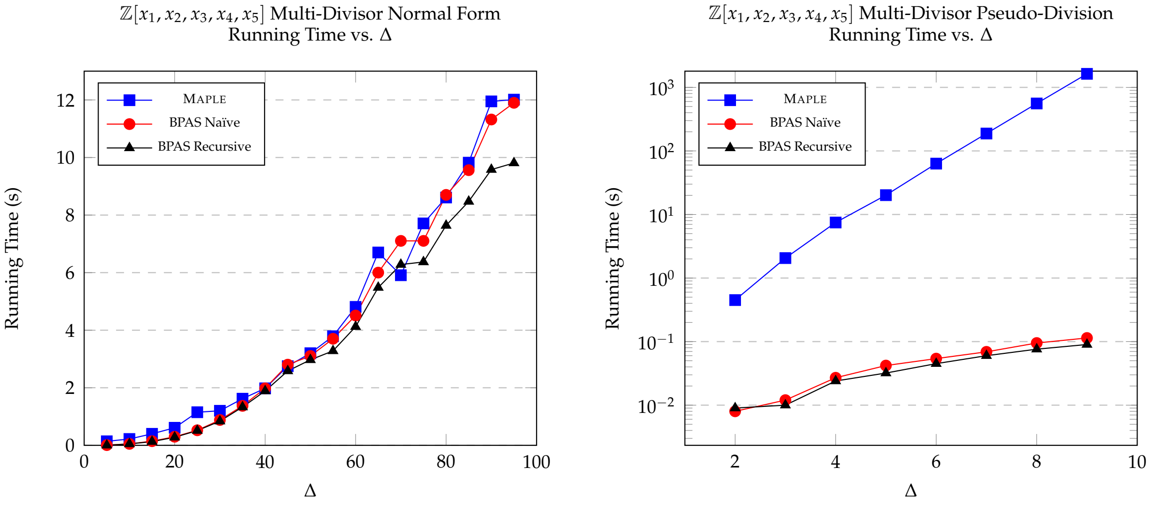
For comparing multi-divisor division (normal form) and pseudo-division with respect to a triangular set, we require more structure to our operands. For these experiments we use a zero-dimensional normalized (Lazard) triangular set. For our benchmarks we use polynomials with 5 variables, say x_1, x_2, x_3, x_4, x_5, and thus a triangular set of size 5 ($\mathbf{T} = \{t_1, t_2, t_3, t_4, t_5\}$). The polynomials in the divisor set and dividend (a) are always fully dense and have the following degree pattern. For some positive integer Δ we let $\deg(a, x_1) = 2\Delta$, $\deg(a, x_i) = \lg(\Delta)$, $\deg(a, x_1) - \deg(t_1, x_1) = \Delta$ and $\deg(a, x_i) - \deg(t_i, x_i) = 1$ for $1 < i \leq 5$. There is a large gap in the lowest variable, but a small gap in the remaining variables, a common structure of which the recursive algorithms can take advantage. For both polynomials over $\mathbb{Q}$ (Figure 8a,b) and over $\mathbb{Z}$ (Figure 9a,b) we compare the naïve and recursive algorithms for both normal form and pseudo-division by a triangular set against MAPLE. For normal form we call MAPLE's `Groebner:-NormalForm` with respect to the `rem` while for triangular set pseudo-division we implement Algorithm 8 in MAPLE using `prem`. Since `prem` is a non-lazy pseudo-division, we similarly perform non-lazy pseudo-division in our implementations for a fair comparison. In general, the normal form results are relatively close, particularly in comparison to the differences between timings for pseudo-division. Our pseudo-division implementation sees several orders of magnitude speed-up against MAPLE thanks to our recursive scheme and optimized single-divisor pseudo-division.



(**a**) Normal form.

(**b**) Triangular set pseudo-division.

Figure 8. Comparing normal form and triangular set pseudo-division over $\mathbb{Q}$. For each, the naïve algorithm, the recursive algorithm, and the algorithm within MAPLE are compared. Polynomials are in 5 variables and the coefficient bound is 128.



(**a**) Normal form.

(**b**) Triangular set pseudo-division.

Figure 9. Comparing normal form and triangular set pseudo-division over $\mathbb{Z}$. For each, the naïve algorithm, the recursive algorithm, and the algorithm within MAPLE are compared. Polynomials are in 5 variables and the coefficient bound is 128.

8.4. Structured Problems

To further test our implementations on structured examples, rather than random, we look at two problems proposed by Monagan and Pearce in [6,45] and a classic third problem. First is the *sparse 10 variable problem*. In this problem $f_1 = (\sum_{i=1}^{9} (x_i x_{i+1} + x_i) + x_{10}x_1 + x_{10} + 1)^d$ and $g_1 = (\sum_{i=1}^{10} (x_i^2 + x_i) + 1)^d$. The multiplication $h_1 = f_1 \cdot g_1$ and the division $q_1 = h_1 / f_1$ are performed. Second is the *very sparse 5 variable problem*. In this problem $f_2 = (1 + x_1 + x_2^2 + x_3^3 + x_4^5 + x_5^7)^d$ and $g_2 = (1 + x_1^7 + x_2^5 + x_3^3 + x_4^2 + x_5)^d$. The multiplication $h_2 = f_2 \cdot g_2$ and the division $q_2 = h_2 / f_2$ are performed. Lastly, a classic problem in polynomial factorization and division, $f_3 = x^d - 1$ and $g_3 = x - 1$, performing f_3 / g_3. Let us call this the dense quotient problem. The sparsity of the dividend is at a maximum, but the quotient produced is completely dense.

In these problems the coefficients are strictly machine-word sized, i.e., less than 64-bits. We concede that MAPLE uses more advanced techniques for coefficient arithmetic, using machine-word integers and arithmetic, if possible. This contrasts with our implementation which uses only arbitrary-precision integers. It is expected then for MAPLE to out-perform BPAS on these examples with machine-integer coefficients. However, this is only the case for the first two problems. To focus exclusively on the polynomial algorithms and not the integer arithmetic, we repeat the first two problems with arbitrary-precision coefficients, each term in f_1, g_1, f_2, g_2 is given a random positive integer coefficient using a coefficient bound. Table 1 shows the execution time and memory used (for all inputs and outputs) for these problems for various values of d. Multiplication again shows speed-up of a factor between 1.2 and 21.6, becoming increasingly better with increasing sparsity and number of terms. Division here is exact and, in comparison to division with remainder, MAPLE performs much better, likely thanks to their so-called *divisor heap* [32]. Only as sparsity increases does BPAS out-perform MAPLE. In all multi-precision cases, however, memory usage in BPAS is significantly better, being less than half that of MAPLE.

Table 1. Comparing multiplication and division on the three structured problems.

Operation	d	Coef. Bound	BPAS		MAPLE	
			Time (s)	Memory (MiB)	Time (s)	Memory (MiB)
Multiplication	4	1	4.28	172.11	1.78	79.31
$h_1 = f_1 \cdot g_1$	4	64	8.22	353.27	9.58	810.23
	5	64	155.51	2481.37	221.71	5569.25
Division	4	64	7.84	353.29	6.465	816.03
$q_1 = h_1 / f_1$	5	64	154.08	2509.42	124.37	5583.50
Multiplication	12	1	3.61	702.14	2.835	439.21
$h_2 = f_2 \cdot g_2$	12	32	7.62	1878.96	52.80	4026.29
	15	32	51.61	8605.52	1114.23	18,941.05
Division	12	32	8.09	1919.28	10.35	4033.57
$q_2 = h_2 / f_2$	15	32	57.09	8627.16	58.906	18,660.94
Division	1,000,000	1	0.18	38.59	1.505	164.54
$q_3 = f_3 / g_3$	10,000,000	1	1.87	522.65	23.63	1102.21

9. Conclusions and Future Work

In this paper, we have described algorithms and data structures for the high-performance sparse polynomial arithmetic as implemented in the freely available BPAS library. We have considered polynomials both over the integers and the rationals, where others have ignored the rationals; arithmetic over the rationals is important for areas such as Gröbner bases and polynomial system solving. The operations of multiplication, and division, have been extended from [24] to also include division with remainder and a new algorithm for sparse pseudo-division. We employ these fundamental algorithms for use in the mid-level algorithms of normal form and pseudo-division with respect to a triangular set. Our experimentation against MAPLE highlights how the proper treatment of locality and data structures can result huge improvements in memory usage and running time. We achieve orders of magnitude speed-up (for arithmetic over the rationals and non-lazy pseudo-division over the integers) or up to a factor of 67 (for other operations over the integers).

In the future we hope to apply these techniques for locality and arithmetic optimization to obtain efficient computations with regular chains and triangular decompositions. Following the design goals of the BPAS library we plan to apply parallelization to both the arithmetic operations presented in this paper and to upcoming work on triangular decompositions.

Author Contributions: Conceptualization, A.B. and M.M.M.; software, M.A. and A.B.; formal analysis, R.H.C.M.; investigation, M.A. and A.B.; writing—original draft preparation, M.A., A.B., R.H.C.M., M.M.M.; supervision, M.M.M.; project administration, M.M.M.; funding acquisition, M.M.M.

Funding: This research was funded by IBM Canada Ltd (CAS project 880) and Natural Sciences and Engineering Research Council of Canada (NSERC) CRD grant CRDPJ500717-16.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hall, A.D., Jr. The ALTRAN system for rational function manipulation-a survey. In Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, Los Angeles, CA, USA, 23–25 March 1971; ACM: New York, NY, USA, 1971; pp. 153–157.
2. Martin, W.A.; Fateman, R.J. The MACSYMA system. In Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, Los Angeles, CA, USA, 23–25 March 1971; ACM: New York, NY, USA, 1971; pp. 59–75.
3. Hearn, A.C. REDUCE: A user-oriented interactive system for algebraic simplification. In *Symposium on Interactive Systems for Experimental Applied Mathematics, Proceedings of the Association for Computing Machinery Inc. Symposium, Washington, DC, USA, 1 August 1967*; ACM: New York, NY, USA, 1967; pp. 79–90.
4. Van der Hoeven, J.; Lecerf, G. On the bit-complexity of sparse polynomial and series multiplication. *J. Symb. Comput.* **2013**, *50*, 227–254, doi:10.1016/j.jsc.2012.06.004. [CrossRef]
5. Arnold, A.; Roche, D.S. Output-Sensitive Algorithms for Sumset and Sparse Polynomial Multiplication. In Proceedings of the ISSAC 2015, Bath, UK, 6–9 July 2015; pp. 29–36. [CrossRef]
6. Monagan, M.B.; Pearce, R. Sparse polynomial division using a heap. *J. Symb. Comput.* **2011**, *46*, 807–822. [CrossRef]
7. Gastineau, M.; Laskar, J. Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems. In Proceedings of the CASC, Berlin, Germany, 9–13 September 2013; pp. 100–115.
8. Hennessy, J.L.; Patterson, D.A. *Computer Architecture: A Quantitative Approach*, 4th ed.; Morgan Kaufmann: San Francisco, CA, USA, 2007.
9. Wulf, W.A.; McKee, S.A. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Comput. Archit. News* **1995**, *23*, 20–24. [CrossRef]
10. Asadi, M.; Brandt, A.; Moir, R.H.C.; Moreno Maza, M. Sparse Polynomial Arithmetic with the BPAS Library. In Proceedings of the Computer Algebra in Scientific Computing—20th International Workshop (CASC 2018), Lille, France, 17–21 September 2018; pp. 32–50. [CrossRef]
11. Chen, C.; Moreno Maza, M. Algorithms for computing triangular decomposition of polynomial systems. *J. Symb. Comput.* **2012**, *47*, 610–642. [CrossRef]
12. Asadi, M.; Brandt, A.; Chen, C.; Covanov, S.; Mansouri, F.; Mohajerani, D.; Moir, R.H.C.; Moreno Maza, M.; Wang, L.X.; Xie, N.; et al. Basic Polynomial Algebra Subprograms (BPAS). 2018. Available online: http://www.bpaslib.org (accessed on 16 May 2019).
13. Frigo, M.; Leiserson, C.E.; Prokop, H.; Ramachandran, S. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* **2012**, *8*, 4. [CrossRef]
14. Leiserson, C.E. Cilk. In *Encyclopedia of Parallel Computing*; Springer: Boston, MA, USA, 2011; pp. 273–288. [CrossRef]
15. Moreno Maza, M.; Xie, Y. Balanced Dense Polynomial Multiplication on Multi-Cores. *Int. J. Found. Comput. Sci.* **2011**, *22*, 1035–1055. [CrossRef]
16. Chen, C.; Covanov, S.; Mansouri, F.; Moreno Maza, M.; Xie, N.; Xie, Y. Parallel Integer Polynomial Multiplication. *arXiv* **2016**, arXiv:1612.05778.
17. Covanov, S.; Mohajerani, D.; Moreno Maza, M.; Wang, L.X. Big Prime Field FFT on Multi-core Processors. In Proceedings of the ISSAC, Beijing, China, 15–18 July 2019; ACM: New York, NY, USA, 2019.
18. Monagan, M.B.; Pearce, R. Parallel sparse polynomial multiplication using heaps. In Proceedings of the ISSAC, Seoul, Korea, 29–31 July 2009; pp. 263–270.

19. Monagan, M.; Pearce, R. Parallel sparse polynomial division using heaps. In Proceedings of the PASCO, Grenoble, France, 21–23 July 2010; ACM: New York, NY, USA, 2010; pp. 105–111.
20. Biscani, F. Parallel sparse polynomial multiplication on modern hardware architectures. In Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation, Grenoble, France, 22–25 July 2012; ACM: New York, NY, USA, 2012; pp. 83–90.
21. Gastineau, M.; Laskar, J. Parallel sparse multivariate polynomial division. In Proceedings of the PASCO 2015, Bath, UK, 10–12 July 2015; pp. 25–33. [CrossRef]
22. Popescu, D.A.; Garcia, R.T. Multivariate polynomial multiplication on gpu. *Procedia Comput. Sci.* **2016**, *80*, 154–165. [CrossRef]
23. Ewart, T.; Hehn, A.; Troyer, M. VLI–A Library for High Precision Integer and Polynomial Arithmetic. In Proceedings of the International Supercomputing Conference, Leipzig, Germany, 16–20 June 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 267–278.
24. Johnson, S.C. Sparse polynomial arithmetic. *ACM SIGSAM Bull.* **1974**, *8*, 63–71. [CrossRef]
25. Li, X.; Moreno Maza, M.; Schost, É. Fast arithmetic for triangular sets: From theory to practice. *J. Symb. Comput.* **2009**, *44*, 891–907. [CrossRef]
26. Sedgewick, R.; Wayne, K. *Algorithms*, 4th ed.; Addison-Wesley: Boston, MA, USA, 2011.
27. Cox, D.A.; Little, J.; O'shea, D. *Ideals, Varieties, and Algorithms*, 2 ed.; Springer: New York, NY, USA, 1997.
28. Von zur Gathen, J.; Gerhard, J. *Modern Computer Algebra*, 2 ed.; Cambridge University Press: New York, NY, USA, 2003.
29. Gastineau, M.; Laskar, J. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. In Proceedings of the Computational Science—ICCS 2006, 6th International Conference, Reading, UK, 28–31 May 2006; Part II, pp. 446–453. [CrossRef]
30. Monagan, M.; Pearce, R. The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. *ACM Commun. Comput. Algebra* **2015**, *48*, 166–186. [CrossRef]
31. Granlund, T.; others. *GNU MP 6.0 Multiple Precision Arithmetic Library*; Samurai Media Limited: Surrey, UK, 2015.
32. Monagan, M.; Pearce, R. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In Proceedings of the CASC 2007, Bonn, Germany, 16–20 September 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 295–315.
33. Gonnet, G.H.; Munro, J.I. Heaps on heaps. *SIAM J. Comput.* **1986**, *15*, 964–971. [CrossRef]
34. Brandt, A. High Performance Sparse Multivariate Polynomials: Fundamental Data Structures and Algorithms. Master's Thesis, The University of Western Ontario, London, ON, Canada, 2018.
35. Huang, B.C.; Langston, M.A. Practical in-place merging. *Commun. ACM* **1988**, *31*, 348–352. [CrossRef]
36. Katajainen, J.; Pasanen, T.; Teuhola, J. Practical in-place mergesort. *Nord. J. Comput.* **1996**, *3*, 27–40.
37. Dalkilic, M.E.; Acar, E.; Tokatli, G. A simple shuffle-based stable in-place merge algorithm. *Procedia Comput. Sci.* **2011**, *3*, 1049–1054. [CrossRef]
38. Waterloo Maple Inc. MAPLE *2017—The Essential Tool for Mathematics*; Waterloo Maple Inc.: Waterloo, ON, Canada, 2017.
39. Gastineau, M.; Laskar, J. TRIP: A Computer Algebra System Dedicated to Celestial Mechanics and Perturbation Series. *ACM Commun. Comput. Algebra* **2011**, *44*, 194–197. [CrossRef]
40. Bosma, W.; Cannon, J.; Playoust, C. The Magma algebra system. I. The user language. *J. Symb. Comput.* **1997**, *24*, 235–265. [CrossRef]
41. Decker, W.; Greuel, G.M.; Pfister, G.; Schönemann, H. SINGULAR 4-1-1—A Computer Algebra System for Polynomial Computations. 2018 Available online: http://www.singular.uni-kl.de (accessed on 15 March 2019).
42. The PARI Group, Univ. Bordeaux. *PARI/GP Version* `2.3.3`. 2008. Available online: http://pari.math.u-bordeaux.fr/ (accessed on 15 March 2019).
43. Hart, W.; Johansson, F.; Pancratz, S. FLINT: Fast Library for Number Theory. V. 2.4.3. Available online: http://flintlib.org (accessed on 15 March 2019).

44. Shoup, V. NTL: A Library for Doing Number Theory. Available online: www.shoup.net/ntl/ (accessed on 15 March 2019).
45. Monagan, M.B.; Pearce, R. Sparse polynomial multiplication and division in Maple 14. *ACM Commun. Comput. Algebra* **2010**, *44*, 205–209. [CrossRef]