Sparse Polynomial Arithmetic with the BPAS Library

Mohammadali Asadi, Alexander Brandt(✉), Robert H. C. Moir, and Marc Moreno Maza

Department of Computer Science, The University of Western Ontario, London, Canada
{masadi4,abrandt5,rmoir3}@uwo.ca, moreno@csd.uwo.ca

Abstract. We discuss algorithms for pseudo-division and division with remainder of multivariate polynomials with sparse representation. This work is motivated by the computations of normal forms and pseudo-remainders with respect to regular chains. We report on the implementation of those algorithms with the BPAS library.

1 Introduction

General-purpose polynomial system solvers, like MAPLE's `solve` command, combine different algorithms using various polynomial data-types. Consider, as input for such a solver, a polynomial system coming from a real life application, typically consisting of sparse multivariate polynomials with rational number coefficients. A pre-processing phase, using sparse polynomial data-types, attempts to reduce the number of equations, variables or the total degree, say by exploiting properties like symmetries. Then a *core engine*, say based on Gröbner bases, a homotopy method, or triangular decompositions, determines a representation of the real or complex solutions of the input system; this step generally requires a change of polynomial representation (e.g. dense data-types) together with a change of coefficient type (e.g. to finite fields when modular methods are used). Finally, the representation computed by the core engine is converted to one which is more "explicit" or convenient to an end-user; in fact, a return to the original sparse polynomial data-type is likely to take place.

Core engines of polynomial systems solvers have driven a large body of work in the computer algebra community. In particular, algorithms and implementation techniques supporting the polynomial and matrix data-types used by those core engines have received great attention. In contrast, until a decade ago, the implementation of sparse polynomial arithmetic, which is the default data-type for general-purpose computer algebra systems, like MAPLE, MATHEMATICA, SAGE, and SINGULAR, was often less optimized. Nevertheless, we should mention pioneer works like the seminal article of Johnson [11] in 1974.

Research works conducted in the last decade on sparse polynomial arithmetic operations[1] and data-types can essentially be categorized into two streams. The

[1] Polynomial arithmetic operations refers here to addition, multiplication, division and pseudo-division.

V. P. Gerdt et al. (Eds.): CASC 2018, LNCS 11077, pp. 32–50, 2018.
https://doi.org/10.1007/978-3-319-99639-4_3

first one deals primarily with algebraic complexity, see the works of van der Hoeven and Lecerf [10] and those of Arnold and Roch [1]. The latter focuses on implementation techniques, see the works of Monagan and Pearce [15,19], and those of Gastineau and Laskar [5,6]. The present work subscribes to this second stream. We are motivated by obtaining efficient implementation of triangular decomposition algorithms based on the theory of regular chains [4]. To be precise, we aim at adapting the algorithms of the `RegularChains` library [13] to the *Basic Polynomial Algebra Subprograms* (BPAS). This latter library is written mainly in C language, for high performance, wrapped in a C++ interface to make use of object-oriented programming and for end-user usability. The `Cilk` extension [12] is used for multi-threading, targeting multi-core processors. BPAS is already equipped with parallel dense polynomial arithmetic over finite fields [20] and the integers [3]. BPAS is publicly available in source at www.bpaslib.org.

We report in this paper on the implementation with the BPAS library of elementary arithmetic operations for multivariate polynomials represented with sparse data-types. In Sect. 2, we start by discussing multiplication and division with remainder, following the papers [11,15,19]. Then, we propose an algorithm for pseudo-division using similar principles. Our presentation of both division with remainder and pseudo-division has two levels: one abstract level independent of the supporting data-structures (see Algorithms 1 and 3) and one level taking advantage of heap data-structures (see Algorithms 2 and 4). This presentation allows us to formally prove those algorithms.

In Sect. 3, we discuss the implementation of the algorithms presented in Sect. 2 within the BPAS library; we highlight the differences between our implementation and that realized in Maple by Monagan and Pearce. Note that, currently, all the BPAS code for sparse polynomial arithmetic is entirely serial C code, that is, multi-threading is not used yet. We stress the fact that, while algorithms for division with remainder (Algorithms 1 and 2) may look similar to their counterparts for pseudo-division (Algorithms 3 and 4), implementation of the latter is by far more challenging than that of the former. Indeed, pseudo-division is essentially a univariate operation. Thus, when used in the context of multivariate polynomials, careful data-structure manipulations are needed to optimize both memory usage and access time to terms of polynomials, see Sect. 3.5. Section 4 gathers our experimental results. For multivariate polynomials over the integers (for which both BPAS and Maple have optimized implementation), BPAS is usually faster with a speedup factor typically between 2 to 3, see Figs. 5, 6 and 8. For multivariate polynomials over the rational numbers (for which only BPAS has an optimized implementation), BPAS is faster than Maple by 2 to 3 orders of magnitude, see Figs. 3, 4 and 7. This is particularly true for the computation of normal forms, see Fig. 9.

2 Sparse Polynomial Arithmetic

For the treatment of sparse polynomial arithmetic we require both a distributed and recursive view of polynomials, depending on the operation. For a distributed

polynomial $a \in \mathbb{D}[x_1, \ldots, x_m]$, for an integral domain $\mathbb{D}$ and variable ordering $x_1 < x_2 < \cdots < x_m$, 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, $0 \neq a_i \in \mathbb{D}$, α_i is an exponent vector for the variables $X = (x_1, \ldots, x_m)$. A term of a is represented by $A_i = a_i X^{\alpha_i}$. We assume that the terms are ordered (decreasing) lexicographically, so that $\mathrm{lc}(a) = a_1$ is the *leading coefficient* of a, $\mathrm{lm}(a) = X^{\alpha_1}$ is the *leading monomial* of a, and $\mathrm{lt}(a) = a_1 X^{\alpha_1}$ is the *leading term* of a. If a is not constant, the greatest variable appearing in a is the *main variable* of a (denoted $\mathrm{mvar}(a)$). 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. Then, $\deg(a, x_j)$ is the maximum value of $\deg(A_i, x_j)$ among all terms A_i of a.

For a recursive view of a non-constant polynomial $a \in \mathbb{D}[x_1, \ldots, x_m]$, again with $x_1 < x_2 < \cdots < x_m$, we view a as a univariate polynomial in $R[x_j]$, with $x_j = \mathrm{mvar}(a)$ is the largest variable occurring in a, and where $R = \mathbb{D}[x_1, \ldots, x_{j-1}]$. Viewed in $R[x_j]$, the leading coefficient of a is the *initial* of a (denoted $\mathrm{init}(a)$). Given a term A_i of $a \in R[x_j]$, $\mathrm{coef}(A_i) \in \mathbb{D}[x_1, \ldots, x_{j-1}]$ and $\mathrm{expn}(A_i) = \deg(A_i, x_j)$.

Addition (or subtraction) of two polynomials requires joining the terms of the two summands, combining terms with identical exponents (with possible cancellation) and then sorting the terms of the sum. A naïve approach is to compute the sum $a + b$ term-by-term, adding a term of the addend (b) to the augend (a), sorting at each step, in a manner similar to *insertion sort*. An efficient algorithm instead uses *merge sort*, taking advantage of the fact that the terms of a and b are already ordered. For details of the algorithm see [11, p. 65].

Multiplication of two polynomials requires generating the terms of the product, combining terms with equal exponents and sorting the product terms. A naïve approach is to compute 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 they have equal exponents, and the $n_a n_b$ terms must be sorted. Again, following Johnson [11], we can obtain more efficient algorithms by generating terms in sorted order.

We make good use of the sparse data structure for $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i}$, and $b = \sum_{j=1}^{n_b} b_j X^{\beta_j}$, by observing that for given α_i and β_j, we always have that $X^{\alpha_{i+1}+\beta_j}$ and $X^{\alpha_i+\beta_{j+1}}$ are less than $X^{\alpha_i+\beta_j}$ in the term order. Given that $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$ we can generate terms of the product in order by merging n_a "streams" of terms obtained by multiplying a single term of a distributed over b,

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

and then selecting the maximum term from the heads of the streams. The new head of the stream where a term is removed is then the term to its right in that

stream. We can efficiently handle this sub-problem of selecting the maximum term by storing the heads of the streams in a priority queue, which we implement using a binary max-heap. We minimize the size of the heap by choosing the order of multiplicative factors such that $n_a \leq n_b$, which we are free to do since multiplication is commutative. Because the heap multiplication algorithm was specified completely by Johnson, we refer the reader to [11], which discusses the algorithm and provides pseudo-code.

2.1 Division

We now consider the problem of multivariate division, where the input polynomials $a, b \in \mathbb{D}[x_1, \ldots, x_m]$, with $b \notin \mathbb{D}$ being the divisor and a the dividend. We assume that $\mathbb{D}$ is a field. Hence $\{b\}$ is a Gröbner basis of the ideal it generates. Thus, we can specify the output as $q, r \in \mathbb{D}[x_1, \ldots, x_m]$ satisfying $a = qb + r$, such that r is reduced with respect to b treated as a Gröbner basis.

Division presents a more tricky problem in terms of heap-optimization. We must compute terms of the quotient and remainder in order, and produce terms of the product qb in order, as terms of q are generated in the execution of the algorithm. To see how this can be done without a heap, consider Algorithm 1, which computes q and r term by term by computing $\tilde{r} = \mathrm{lt}(a - qb - r)$ at each step. This works for multivariate division because introducing a new quotient term whenever $\mathrm{lt}(b) \mid \tilde{r}$ ensures that any subsequent terms of $a - qb - r$ that do not satisfy this condition will be remainder terms. This allows terms of both q and r to be computed in order.

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

Proof. It is enough to show that for each iteration of the loop, the term $\tilde{r}$ decreases strictly. It follows from the axioms of a term order that $\tilde{r}$ becomes zero after finitely many iterations. We denote the values of the variables of Algorithm 1 on the i-th iteration by superscripts. For each i, depending on whether or not $\mathrm{lt}(b) \mid \tilde{r}^{(i)}$ holds, we have two possibilities:

- $Q_\ell = \tilde{r}^{(i)}/B_1$, where Q_ℓ is a new quotient term;
- or $R_k = \tilde{r}^{(i)}$, where R_k is a new remainder term.

We provide the proof for the first case. The second case is similar but essentially trivial. Since $\tilde{r}^{(i)} = Q_\ell B_1$ holds by assumption, we have

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

The remainder r is reduced with respect to $\{b\}$ because all terms R_k of r satisfy $\mathrm{lt}(b) \nmid R_k$ by construction. □

Heap-optimization can then be applied to Algorithm 1 by using a heap to keep track of the computation of the product qb. This is a special case of heap multiplication. The major difference from multiplication, where all terms of both factors are known at the outset, is that q is computed as the algorithm proceeds, which forces q to be the multiplier and b the multiplicand. Thus, each stream consists of a term Q_ℓ of q distributed over b. Another difference from multiplication is that each stream is initiated with the term $Q_\ell B_2$, because $Q_\ell B_1$ need not be computed because it is canceled out by construction.

The management of the heap to compute a product ab requires a number of specialized functions. We provide here a simplified interface consisting of three functions. **heapInsert**(A_i, B_j) adds the product of A_i and B_j to the heap[2]. **heapPeek**() gets the exponent vector ε of the top element in the heap. **heapExtract**() removes the top element of the heap *and* inserts the next element of the stream from which the top element came from. That is, if there are any elements remaining in that stream. The key modification of Algorithm 1 to reach Algorithm 2 is to use terms of qb from the heap to compute $\tilde{r} = \mathrm{lt}(a - qb - r)$. This requires tracking three cases: (1) $\tilde{r}$ is an uncanceled term of a; (2) $\tilde{r}$ is a term of the difference $(a - r) - (qb)$; and (3) $\tilde{r}$ is a term of $-qb$ such that all remaining terms of $a - r$ are smaller in the term order.

Let ε be the exponent vector of the top term of the heap computation of qb. If the heap is empty, we let $\varepsilon = (-1, 0, \ldots, 0)$, which will be less than any exponent of any polynomial term on account of the first element being -1. We therefore abuse notation and write $\varepsilon = -1$ for an empty heap. Let A_k be the greatest uncanceled term of a. Then, the three cases correspond to conditions on the ordering of ε and $\mathrm{expn}(A_k)$. The term $\tilde{r}$ is an uncanceled term of a (case 1) either if the heap is empty (indicating that no terms of q have yet been computed or all terms of qb have been extracted) or if $\varepsilon > -1$ but $\varepsilon < \mathrm{expn}(A_k)$. In either of these two situations $\varepsilon < \mathrm{expn}(A_k)$ holds. The term $\tilde{r}$ is a term of the difference $(a - r) - (qb)$ (case 2) if both A_k and ε have the same exponent ($\varepsilon = \mathrm{expn}(A_k)$). And $\tilde{r}$ is a term of $-qb$ (case 3) whenever $\varepsilon > \mathrm{expn}(A_k)$ holds.

Algorithm 2 uses this observation to compute $\tilde{r}$ by adding a conditional to compare the ranks of ε and $\mathrm{expn}(A_k)$. Terms are only extracted from the heap when $\varepsilon \geq \deg(A_k)$ holds; and when a term is extracted the next term from the given stream, if there is one, is added to the heap (defined behaviour of **heapExtract**()). The adding of new terms to q and r is almost identical to Algorithm 1, except that for quotient terms we initiate a new stream starting with $Q_\ell B_2$ (because $Q_\ell B_1$ is canceled by construction). Together with Proposition 1, then, we have established the following proposition.

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

[2] Note that the heap need not actually store product terms but can simply store the indices of the two factors, with the product only computed when elements of the heap are removed. This strategy is needed for pseudodivision, discussed below, where the quotient terms are updated in the course of the algorithm.

Algorithm 1. DIVIDE(a,b)
$a, b \in \mathbb{D}[x_1, \ldots, x_m]$, mdeg$(b) > 0$, return $q, r \in \mathbb{D}[x_1, \ldots, x_m]$ such that $a = qb + r$ where r is reduced with respect to the Gröbner basis $\{b\}$.

```
1: q := 0; r := 0
2: while r̃ := lt(a − qb − r) ≠ 0 do
3:     if lt(b) | r̃ then
4:         q := q + r̃/lt(b)
5:     else
6:         r := r + r̃
7:     end if
8: end while
9: return (q, r)
```

Algorithm 2. DIVIDE(a,b)
$a, b \in \mathbb{D}[x_1, \ldots, x_m]$, mdeg$(b) > 0$, return $q, r \in \mathbb{D}[x_1, \ldots, x_m]$ such that $a = qb + r$ where r is reduced with respect to the Gröbner basis $\{b\}$.

```
1: q := 0; r := 0
2: k := 1; ℓ := 0
3: while ε := heapPeek() > −1 or k ≤ n_a do
4:     if ε < expn(A_k) then
5:         r̃ := A_k
6:         η := expn(A_k); k := k + 1
7:     else if ε = expn(A_k) then
8:         r̃ := A_k − heapExtract()
9:         η := ε; k := k + 1
10:    else
11:        r̃ := −heapExtract()
12:        η := ε
13:    end if
14:    if expn(B_1) | η then
15:        ℓ := ℓ + 1; Q_ℓ := r̃/B_1; q := q + Q_ℓ
16:        heapInsert(Q_ℓ, B_2)
17:    else
18:        r := r + r̃
19:    end if
20: end while
21: return (q, r)
```

Algorithm 3. PSEUDODIVIDE(a,b,x)
$a, b \in \mathbb{D}[x]$, $\deg(b, x) > 0$, returns $q, r \in \mathbb{D}[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r, x) < \deg(b, x)$.

```
1: q := 0; r := 0; h := lc(b); ℓ := 0; γ = deg(b.x)
2: while r̃ := lt(h^ℓ a − qb − r) ≠ 0 do
3:     if x^γ | r̃ then
4:         q := hq + r̃/x^γ; ℓ := ℓ + 1
5:     else
6:         r := r + r̃
7:     end if
8: end while
9: return (q, r, ℓ)
```

Algorithm 4. PSEUDODIVIDE(a,b,x)
$a, b \in \mathbb{D}[x]$, $\deg(b) > 0$, returns $q, r \in \mathbb{D}[x]$ and $\ell \in \mathbb{N}$ such that $h^\ell a = qb + r$, with $\deg(r, x) < \deg(b, x)$.

```
1: q := 0; r := 0; h := lc(b)
2: ε := −1; s := 0
3: k := 1; ℓ := 0; γ := deg(b, x)
4: while ε := heapPeek() > −1 or k ≤ n_a do
5:     if ε < deg(A_k, x) then
6:         r̃ := h^ℓ A_k
7:         η := deg(A_k, x); k := k + 1
8:     else if ε = deg(A_k, x) then
9:         r̃ := h^ℓ A_k − heapExtract()
10:        η := ε; k := k + 1
11:    else
12:        r̃ := −heapExtract()
13:        η := ε
14:    end if
15:    if deg(b, x) ≤ η then
16:        q := hq; ℓ := ℓ + 1; Q_ℓ := r̃/x^γ
17:        heapInsert(Q_ℓ, B_2); q := q + Q_ℓ
18:    else
19:        r := r + r̃
20:    end if
21: end while
22: return (q, r, ℓ)
```

2.2 Pseudo-Division

The pseudo-division algorithm is essentially univariate, and terms here are elements of $\mathbb{D}[x]$ for an arbitrary integral domain $\mathbb{D}$. Pseudo-division is essentially a fraction-free division: rather than dividing a by $h = \text{lc}(b)$ for each term of the quotient q, it multiplies a by h. If the quotient ends up with ℓ terms, then the result must satisfy $h^\ell a = qb + r$.

An important consequence of pseudo-division being univariate is that all of the quotient terms are computed first and then all of the remainder terms are computed. This is because we can always carry out a pseudo-division step provided that $\deg(b, x) \leq \deg(\text{lt}(h^\ell a - qb), x)$, where $\text{lt}(h^\ell a - qb)$ is the equivalent

of $\tilde{r}$ from Algorithm 1 when $r = 0$. Thus, we adopt the same symbol for it in Algorithm 3, which is the extension of Algorithm 1 to pseudo-division. The only difference in these algorithms is that each time we compute a new pseudo-quotient term we do so as $\tilde{r}/x^\gamma$, where $\gamma = \deg(b, x)$ (fraction free division), rather than $\tilde{r}/B_1 = \tilde{r}/(hx^\gamma)$ as before, and because we add a factor of h to a, we must also multiply the previous value of the quotient by h.

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

Proof. Similar to Proposition 1. The two cases here are $Q_\ell = \tilde{r}^{(i)}/x^\gamma$ and $R_k = \tilde{r}^{(i)}$. We consider the first case (the second case is similar and essentially trivial). In the first case $r^{(i)} = 0$, since quotient terms are still being computed, so that $\tilde{r}^{(i)} = \mathrm{lt}(h^\ell a - q^{(i)}b)$. Since $\tilde{r}^{(i)} = Q_\ell x^\gamma$ by assumption, $h\tilde{r}^{(i)} = Q_\ell B_1$, and we have

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

The condition $\deg(r, x) < \deg(b, x)$ is ensured because quotient terms are computed until $x^\gamma \nmid \tilde{r}$ holds, that is, until $\deg(h^\ell a - qb, x) < \deg(b, x)$ holds. □

Heap-optimization of Algorithm 3 proceeds in much the same way as for division. The only additional consideration required for Algorithm 4 is the accounting for factors of h in the computation of $\mathrm{lt}(h^\ell a - qb - r)$. This only requires adding as many factors of h to A_k that have been added to the quotient up to the current iteration. Since ℓ terms have been added to q, we multiply A_k by h^ℓ each time we use one of the terms. Additional factors of h are added when the previous quotient is multiplied by h prior to the computation of the next quotient term. Other than this, the shift from Algorithm 3 to Algorithm 4 follows the analogous shift between Algorithms 1 and 2 exactly. We therefore have the following.

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

Proof. The proof is a straightforward adaptation of the preceding observations and the proofs for Propositions 2 and 3. The key observation is the first main conditional statement in the while loop computes $\tilde{r} = \mathrm{lt}(h^\ell a - qb - r)$, where $r = 0$ until q has been computed, and the second main conditional computes a term of q or r from $\tilde{r}$ accordingly, following the structure of Algorithm 3. □

2.3 Multi-Divisor (Pseudo-)Division

One natural application of division with remainder of multivariate polynomials is the computation of normal forms with respect to Gröbner bases. Moreover, the computation of pseudo-quotient and pseudo-remainder of a polynomial with respect to multiple polynomials (or a triangular set) is also natural. Normal forms can be computed by Algorithms 5 and 7 in Appendix A while pseudo-division

by a triangular set can be computed by Algorithms 6 and 8. Section 4 includes benchmarks of those four algorithms implemented with the BPAS library.

3 Implementation and Optimizations

With the ever-increasing gap between processor speeds and memory-access time, our implementation techniques focus on memory usage and management. Our implementations effectively traverse memory while making use of memory-efficient data structures with good data locality. In this section we consider polynomial representations and corresponding data structures (Sect. 3.1), addition and multiplication (Sect. 3.2), heap-optimizations (Sect. 3.3), division (Sect. 3.4), and lastly, pseudo-division (Sect. 3.5).

3.1 Polynomial Representations

The simplest scheme to represent a polynomial sparsely would be a linked list where each node in the list is a single term of that polynomial. This representation makes handling and manipulating terms very easy with simple pointer manipulation. However, the indirection created by pointers and (possibly) poor locality of successive nodes in the list makes this scheme inefficient for memory usage. Rather, packing the polynomial terms into an array removes the overhead of linked list pointers and improves locality. We call this array-based representation of a polynomial an *alternating array* following the terminology introduced in 1997 in the `BasicMath` library, part of the European Project FRISCO https://cordis.europa.eu/project/rcn/31471_en.html; see also [2].

The alternating array representation packs terms side-by-side in an array, effectively alternating between coefficients and monomials. A coefficient and its corresponding monomial are thus optimally local in memory with respect to each other. Similar schemes have been used in Maple [18,19]. In the case of Maple, their scheme uses pointers into a parallel array to store the multi-precision integer coefficient, whereas we store the multi-precision coefficients directly in the array. Moreover, for this efficient data structure Maple is limited to integer polynomials while all other polynomials use an old sum-of-products form [18]. In contrast, our alternating array representation in the BPAS library supports both integer and rational number coefficients.

Coefficients are represented easily using GMP multi-precision numbers [7]. As for monomials, we use *exponent packing*. Using bit-masks and shifts, multiple integers, each of small absolute value, can effectively be stored in a single 64-bit machine word. The idea of exponent packing has been employed at least since ALTRAN in the late 60s [8] and more recently in [10,16]. Some systems, such as Maple, also encode the total degree of the monomial in the single 64-bit word. This scheme wastes bits which could be used for additional variables or higher degrees. In particular, monomials are limited to 21 variables each with a maximum degree of 3 [18]. Our representation does not encode total degree, therefore we can encode up to 32 variables, each of maximum degree 3. Moreover,

in polynomial system solving, degrees of lower ordered variables often increase much quicker than those of high ordered variables. Thus, in our implementation, we pack exponents disproportionately within the machine word, giving more bits to lower ordered variables, ensuring all 64 bits are made useful.

It is worth noting that our sparse representations are used for all of our algorithms, including division and pseudo-division, where (pseudo-)quotients and (pseudo-)remainders are often much more dense than the divisor and dividend. However, since we are working with multivariate polynomials, a dense representation would grow exponentially with the number of variables and, therefore, our sparse representation is still worthwhile and efficient.

3.2 Addition and Multiplication

For these two simple operations, we just point out a few implementation tricks. An "in-place" addition (subtraction) can be implemented with our alternating array representation. This is not strictly in-place, as that would involve far too much memory movement and swapping of elements, resulting in poor locality and poor performance. Instead, we can pre-allocate a destination array as with an "out-of-place" addition algorithm, but, rather than copying coefficients, we reuse the underlying GMP data. With modestly-sized coefficients, less than 192 bits each, the savings can reach 20% compared to the out of place implementation.

As for multiplication, we pre-allocate the maximum possible space for the product $(n_a \cdot n_b)$. Assuming that a has fewer terms than b, we pre-allocate space in the heap for exactly n_a elements as that will be the exact number of streams to consider. This minimizes memory movement and reallocation required throughout the computation of appending product terms to the product polynomial. If the product terms were to out-grow some initial conservative pre-allocation the reallocation and memory movement could result in a large overhead.

3.3 Heap-Optimizations

The performance of our code is very dependent on the implementation of its data-structures, and in particular, heaps. Aside from coefficient arithmetic, all of the work for multiplying terms comes from obtaining the ordering of product terms. Hence, the heap, whose purpose is to produce terms in the required order, takes the majority of the effort of our algorithm. Our implementation of heaps includes all the techniques reported in [16], including the technique of *chaining*. We mention an additional trick used in our code. With chaining, the coefficients of the product terms are already not stored directly in the heap, but they still play a role in overall auxiliary memory needed for the algorithm. With our alternating array representation of polynomials it is very easy to directly index the operand polynomials to access the appropriate coefficient. Thus, our heap only stores the *indices* of the operand coefficients which together form the coefficient of the particular product term (Fig. 1). This reduces the memory required for

each coefficient from 32 bytes, in the case of rational number coefficients, down to 8 bytes. Similar schemes using pointers to coefficients have been examined in [16,19] but indices are even more succinct than pointers.

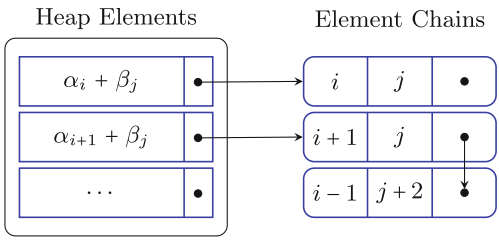

Fig. 1. A heap of product terms, showing element chaining and index-based storing of coefficients. In this case, terms $A_{i+1} \cdot B_j$ and $A_{i-1} \cdot B_{j+2}$ have equal monomials and are chained together.

3.4 Division

Division is essentially a direct application of multiplication. We again use heaps, with all of its optimizations, using the production of product terms in-order to produce the terms of the quotient and remainder in-order. Division varies from multiplication as instead of producing the product terms of the two input operands, we must produce product terms between the divisor and the continually updating quotient. This poses problems for memory management as we do not know ahead of time the sizes of the quotient or remainder. In multiplication we are able to pre-allocate $n_a \cdot n_b$ space for the product as that is the known maximum number of product terms. The indeterminate number of quotient and remainder terms does not allow for such one-time allocation and we must continually check for producing more terms than the number for which we have allocated space. We begin by allocating n_a space for the quotient and remainder, as generally the dividend is larger than the divisor. Then, if more terms are produced than we have currently allocated for, we double the current allocation.

Whenever we reallocate space for the quotient we also reallocate space for the same number of terms in the heap. Recall the maximum number of terms in the heap is equal to the number of quotient terms (as we distribute terms of the quotient over the divisor in the multiplication). So, we are safe in doing this memory allocation for the heap even if it does not make use of it all. This has benefits for performance as we do not need to check for overflow on each insert into the heap; it is guaranteed to have enough space.

3.5 Pseudo-Division

As seen in Sect. 2 the algorithm for division can easily be adapted for pseudo-division. With only the modification of multiplying the dividend and quotient by

the divisor's initial, we obtained an algorithm for pseudo-division that efficiently produces terms in order. However, the implementation between these two algorithms is very different. In essence, pseudo-division is a univariate operation, viewing the input multivariate polynomials recursively. That is, the dividend and divisor are seen as univariate polynomials over some arbitrary (polynomial) integral domain. Therefore, coefficients can be, and indeed are, entire polynomials themselves. Coefficient arithmetic becomes non-trivial. Our distributed multivariate polynomial representation, as seen in Sect. 3.1 would be inefficient to traverse and manipulate in this recursive way. We introduce a new polynomial representation to easily view polynomials in this univariate, recursive way in order to efficiently operate on them within the semantics of pseudo-division.

This recursive polynomial representation uses an in-place, very fast conversion between the normal distributed representation and the recursive one. This amounts to minimal overhead and allows the same polynomials to be easily used as operands to pseudo-division or any other arithmetic operation. Of course, an in-place conversion is beneficial to avoid memory movement and reduce the working memory required for the algorithm.

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

Next, we create a secondary alternating array to store these offsets. This array alternates between an exponent of the main variable and a pointer to the original array which is offset to point to the beginning of the block that corresponds to the preceding main variable exponent. Note that we also store the size of each block. This is convenient when we need to do coefficient arithmetic as those coefficients are themselves polynomials that must know their size to perform arithmetic. In addition, as we traverse the array to determine the blocks, we zero out the degree of the main variable for every monomial. This ensures that the degree of the main variable does not pollute the polynomial coefficient arithmetic. Figure 2 shows this secondary array structure along with the original array, highlighting the conversion process.

These two alternating arrays together exactly and efficiently represent the recursive view of a polynomial, having coefficients from an arbitrary polynomial ring and univariate monomials. The secondary alternating array requires little additional memory. It will have size equal to the number of unique values of degree of the main variable in the distributed polynomial. In practice, with sparse polynomials, this number is quite small. In the absolute worst case, for integer polynomials that are fully dense with respect to the main variable, this secondary array requires $O(\frac{2}{3}n)$ additional space. With multi-precision coefficients and/or

rational number coefficients, this fraction becomes much smaller. This additional space becomes increasingly insignificant as the integers (rational numbers) grow in size, as they always do in pseudo-division calculations.

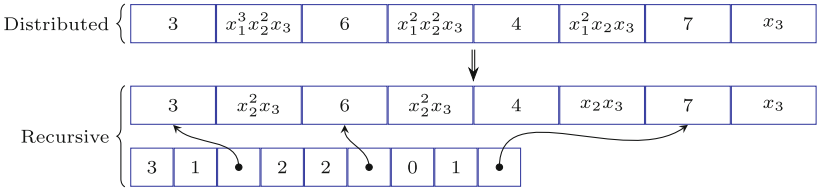

Fig. 2. A distributed polynomial representation converted to the recursive polynomial representation, showing the additional secondary array. The secondary array alternates between: (1) exponent of the main variable, (2) size of the coefficient polynomial, and (3) a pointer to the coefficient polynomial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

With the recursive view of a polynomial efficiently implemented, it is then important to consider efficiency of coefficient arithmetic. As coefficients are now full polynomials there is more overhead in manipulating them and performing arithmetic. One important implementation detail is to perform the addition (and subtraction) of like-terms in-place. 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 the previous sub-section, allows for the re-use of underlying GMP data. Therefore, performance of in-place addition compared to out-of-place becomes increasingly better as coefficients grow throughout the pseudo-division algorithm.

Similarly, the update of the quotient by multiplying by the initial of the divisor, requires a multiplication of full polynomials. If we wish to save on memory movement we should perform this multiplication in place. However, notice that, in our recursive representation, coefficient polynomials are tightly packed in a continuous array. To modify them in place would require shifting all following coefficients down the array to make room for the strictly large product polynomial. To avoid this unnecessary memory movement, we modify the recursive data structure exclusively for the 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. Moreover, we do not require any recursively viewed operations on the remainder. Hence, as terms of the remainder are produced, we store them directly in the normal, distributed representation, avoiding conversion out of the recursive representation and any memory overhead of the additional recursive array.

Lastly, our final optimization is common among other sparse pseudo-division algorithms. We perform a divisibility test between a newly produced quotient term and the initial of the divisor. If division is exact, we avoid one multiplication of the quotient with the divisor's initial, and the newly produced quotient term is replaced by its quotient calculated by the exact division. This divisibility test is little overhead as the test usually fails very early. Often, this divisibility test is instead performed by a GCD calculation in order to always multiply the quotient by the smallest possible polynomial instead of the full initial of the divisor. However, efficient GCD calculation for multivariate polynomials is not trivial. A simple divisibility is often sufficient in practice.

4 Experimentation

For univariate polynomials sparsity is easily defined as the maximum degree difference between successive polynomial terms. Though sparsity is not so easily defined for multivariate polynomials, we propose the following adaptation of the univariate case to the multivariate one, inspired by Kronecker substitution. Let $f \in \mathbb{D}[x_1, \ldots, x_m]$ be non-zero and define $r = \max(\deg(f, x_i), 1 \leq i \leq m) + 1$. Then, every exponent vector $e = (e_1, \ldots, e_m)$ of a term of f can be identified with the radix r representation of the integer $z(e) = e_1 + e_2 r + \cdots + e_m r^{m-1}$. We call *sparsity* of f the smallest integer s which is greater or equal to $z(e) - z(e')$, where e, e' are any two consecutive exponent vectors of f. If f has n terms then we have $r^m \leq n\, s$. For our experiments, sparse polynomials were randomly generated using the following parameters: number of variables m, number of terms n, sparsity s, and maximum number of bits in any coefficient. Then, exponent vectors are generated as radix r representations with m digits and r computed as $\lfloor \sqrt[m]{s \cdot m} \rfloor$.

We compare our implementation against Maple for both integer polynomials and rational number polynomials. Over the past 10 years or so, Maple has become the leader in integer polynomial arithmetic thanks to the extensive work of Monagan and Pearce [15–17,19]. Benchmarks there provide clear indication that their implementation outperforms many other computer algebra systems including: Trip, Magma, Singular, and Pari. Moreover, other common systems like FLINT [9] and NTL [21] provide only univariate polynomial implementations, meaning the comparison against our multivariate implementation would be unfair. Therefore, we compare our implementations against the leading high-performance implementation that is provided by Maple in particular, Maple 2017.

We consider multiplication and division over $\mathbb{Q}$ (Figs. 3 and 4), multiplication and division over $\mathbb{Z}$ (Figs. 5 and 6), pseudo-division over $\mathbb{Q}$ and $\mathbb{Z}$ (Figs. 7 and 8), and multi-divisor normal form and pseudo-division computation over $\mathbb{Q}$ (Fig. 9 and 10). In all cases (except dense integer multiplication) BPAS performs favourably over Maple. We note that random instances of division do not provide smooth results due to varying sizes of resulting quotients and remainders. Our benchmarks were collected using an Intel Xeon X560 processor at 2.67 GHz, 32 KB L1 data cache, 256 KB L2 cache, 12288 KB L3 cache, and 48 GB of RAM.

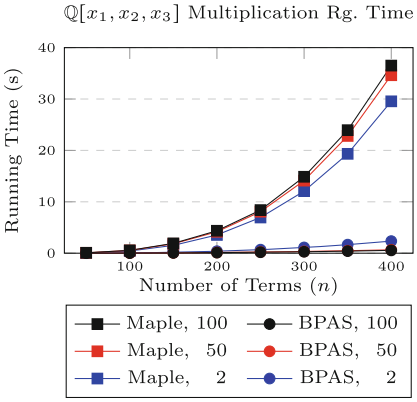

Fig. 3. $\mathbb{Q}$ multiplication. Sparsity varies as noted in the legend, # coefficient bits is 128.

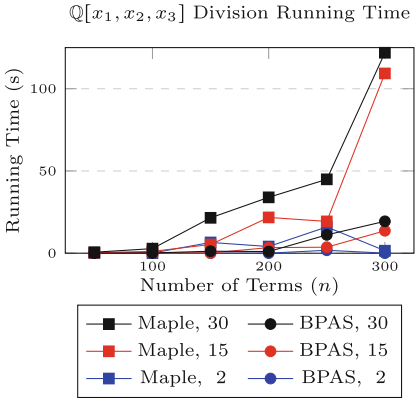

Fig. 4. $\mathbb{Q}$ division. Sparsity varies as noted in the legend, # of divisor terms is $n/2$, # coefficient bits is 128.

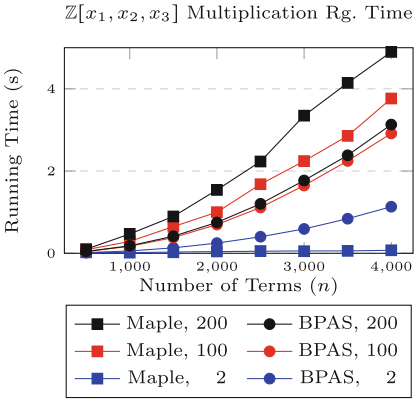

Fig. 5. $\mathbb{Z}$ multiplication. Sparsity varies as noted in the legend, # coefficient bits is 128.

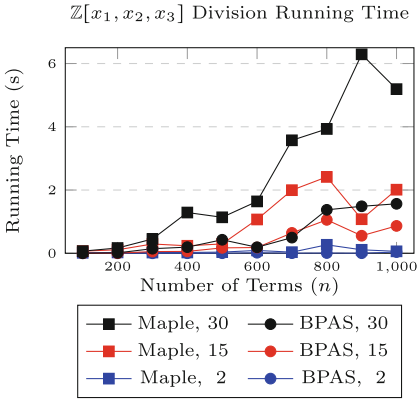

Fig. 6. $\mathbb{Z}$ division. Sparsity varies as noted in the legend, # divisor terms is $n/2$, # coefficient bits is 128.

It is clear from these benchmarks that having optimized data structures and fundamental algorithms is important. For polynomials over the rational numbers, where MAPLE lacks an optimized implementation, our code performs orders of magnitude better. Even for MAPLE's optimized implementation of polynomials over the integers, our code still performs at a fraction of the time. This performance savings is substantial and is very apparent when comparing normal forms (see Fig. 9). With the repeated division required for normal forms, an optimized division algorithm results in extensive performance gains.

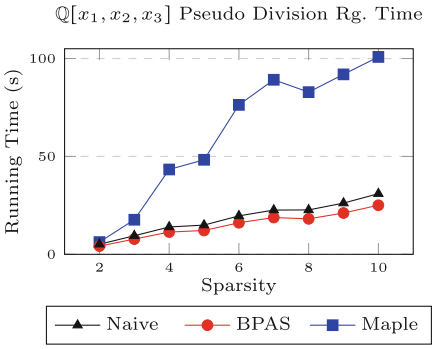

Fig. 7. $\mathbb{Q}$ Pseudo-division. # dividend terms is 175, # divisor terms is 50.

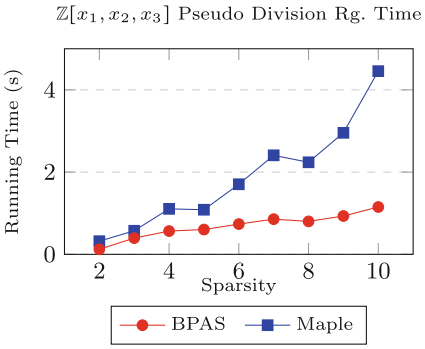

Fig. 8. $\mathbb{Z}$ Pseudo-division. # dividend terms is 175, # divisor terms is 50.

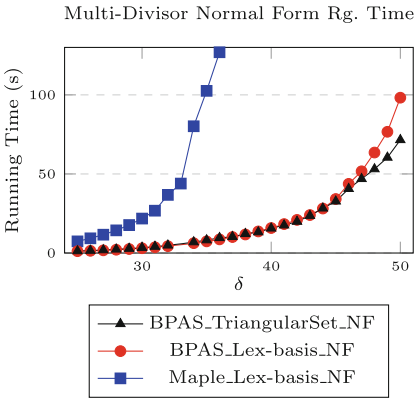

Fig. 9. The divisor set is a random normalized triangular set of $\mathbb{Q}[x_1, x_2, x_3]$ and $deg(a, x_1) - deg(t_3, x_1) = \delta^3$, $deg(a, x_2) - deg(t_2, x_2) = lg(\delta)^3$, $deg(a, x_3) - deg(t_1, x_3) = lg(\delta)^3$ and sparsity 2. BPAS implements Algorithms 5 and 7, see Appendix A.

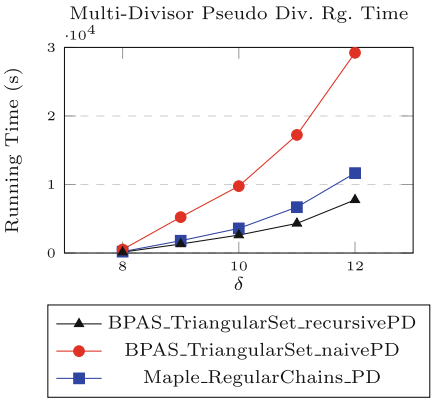

Fig. 10. The divisor set is a random triangular set of $\mathbb{Q}[x_1, x_2, x_3]$ with non-constant initials, sparsity 2 and $deg(a, x_1) - deg(t_3, x_1) = \delta^3$, $deg(a, x_2) - deg(t_2, x_2) = lg(\delta)^3$, $deg(a, x_3) - deg(t_1, x_3) = lg(\delta)^3$. BPAS uses Algorithms 6 and 8.

5 Conclusion

The open-source library Basic Polynomial Algebra Subprograms (BPAS) provides high performance implementations of sparse multivariate polynomial arithmetic, over $\mathbb{Z}$ and $\mathbb{Q}$, including addition, multiplication, division, and pseudo-division, using highly efficient data structures and algorithms. These fundamental operations were extended to the mid-level algorithms of multi-divisor division (normal form) and multi-divisor pseudo-division. Their performance against the leader in polynomial arithmetic, MAPLE, was shown to be a 2–3 times (or order

of magnitude for $\mathbb{Q}$) better. The optimization of these fundamental operations will become the basis for efficient computations with regular chains.

Acknowledgments. The authors would like to thank IBM Canada Ltd (CAS project 880) and NSERC of Canada (CRD grant CRDPJ500717-16).

A Appendix

Let $\mathbb{K}$ be a field. If B is a Gröbner basis of $\mathbb{K}[x_1, \ldots, x_m]$ Algorithm 5 computes the normal form of a polynomial $a \in \mathbb{K}[x_1, \ldots, x_m]$ (together with the quotients) w.r.t. B; the principle is direct (or naïve). Alternatively, when B is a zero-dimensional normalized (thus so-called Lazard) triangular set, one can use Algorithm 7, the recursive principle of which is taken from [14]. Some details are given here. For computing the normal form of polynomial $a \in \mathbb{K}[x_1, \ldots, x_m]$ with respect to a Lazard triangular set $T = \{t_1, \ldots, t_m\} \subset \mathbb{K}[x_1, \ldots, x_m]$, Algorithm 7 uses the recursive representation of polynomials. If $m = 1$, the result is obtained by applying Algorithm 2. Otherwise, the coefficients of a with respect to x_m are reduced w.r.t. to $\{t_1, \ldots, t_{m-1}\}$ by means of a recursive call (Lines 4–11 of the pseudo-code), yielding a polynomial r. Then, r is divided by t_m by applying Algorithm 2, see Line 12, yielding a new polynomial r. Finally, the coefficients w.r.t. x_m of this new polynomial r are reduced w.r.t. to $\{t_1, \ldots, t_{m-1}\}$, by means of a second recursive call, see Lines 13–16.

Algorithm 5. NormalForm (a,B)
Given $a, b_1, \ldots, b_N \in \mathbb{K}[x_1, \ldots, x_m]$, $B = \{b_1, \ldots, b_N\}$ a Gröbner basis, returns $q_1, \ldots, q_N, r \in \mathbb{K}[x_1, \ldots, x_m]$ such that $a = q_1 b_1 + \cdots + q_N b_N + r$ where r is reduced with respect to B.

1: $h := a$; $r := 0$
2: **while** $h \neq 0$ **do**
3: $\quad i = 1$;
4: $\quad$ **while** $i \leq N$ **do**
5: $\quad\quad$ **if** $\mathrm{lm}(b_i) \mid \mathrm{lm}(h)$ **then**
6: $\quad\quad\quad q_i := q_i + \frac{\mathrm{lt}(h)}{\mathrm{lt}(b_i)}$
7: $\quad\quad\quad h := h - \frac{\mathrm{lt}(h)}{\mathrm{lt}(b_i)} b_i$
8: $\quad\quad\quad i := 1$
9: $\quad\quad$ **else**
10: $\quad\quad\quad i := i + 1$
11: $\quad\quad$ **end if**
12: $\quad$ **end while**
13: $\quad r := r + \mathrm{lt}(h)$
14: $\quad h := h - \mathrm{lt}(h)$
15: **end while**
16: **return** $(q_1, \ldots, q_N, r)$

Algorithm 6. naïveTSPD (a,T)
Given $a, t_1, \ldots, t_N \in \mathbb{K}[x_1, \ldots, x_m]$, $T = \{t_1, \ldots, t_N\}$, with $\mathrm{mvar}(t_1) < \cdots < \mathrm{mvar}(t_N)$, returns $q_1, \ldots, q_N, r, h \in \mathbb{K}[x_1, \ldots, x_m]$ such that $ha = q_1 t_1 + \cdots + q_N t_N + r$ where r is reduced with respect to the triangular set T (in the sense that $r = 0$ or $\deg(r, \mathrm{mvar}(t_j)) < \deg(t_j, \mathrm{mvar}(t_j)), 1 \leq j \leq N$) and h is a product of powers of the initials of the polynomials of T.

1: $r := a$; $h := 1$
2: **for** $i = 1, \ldots, N$ **do**
3: $\quad v := \mathrm{mvar}(T_{N-i+1})$
4: $\quad (Q, r, e) :=$ pseudoDivide(r, T_{N-i+1}, v)
5: $\quad H := \mathrm{init}(T_{N-i+1})^e$
6: $\quad h := H\,h$
7: $\quad$ **for** $j = 1, \ldots, N$ **do**
8: $\quad\quad q_j := q_j H$
9: $\quad$ **end for**
10: $\quad q_i := q_i + Q$;
11: **end for**
12: **return** $(q_1, \ldots, q_N, r, h)$

Algorithm 7. TSNF (a, T)
Given $a \in \mathbb{K}[x_1, \ldots, x_m]$, $T = \{t_1, \ldots, t_m\} \subset \mathbb{K}[x_1, \ldots, x_m]$, with $\mathrm{mvar}(t_1) = x_1 < \cdots < \mathrm{mvar}(t_m) = x_m$ and $\mathrm{init}(t_1), \ldots, \mathrm{init}(t_m) \in \mathbb{K}$, returns $q_1, \ldots, q_m, r \in \mathbb{K}[x_1, \ldots, x_m]$ such that $a = q_1 t_1 + \cdots + q_m t_m + r$ where r is reduced (in the sense of Gröbner bases) with respect to the Lazard triangular set T.

1: **if** $m = 1$ **then**
2: $(q_1, r) :=$ DIVIDE(a, t_1)
3: **else**
4: **for** $i = 0, \ldots, \deg(a, x_m)$ **do**
5: $(\{Q_j[i]\}_{j=1}^{m-1}, R[i]) := \mathrm{TSNF}(\mathrm{coef}(a, x_m, i), \{t_j\}_{j=1}^{m-1})$
6: **end for**
7: $q_1 := 0; \ldots; q_m := 0$
8: $r := \sum_i R[i] x_m{}^i$
9: **for** $j = 1, \ldots, m-1$ **do**
10: $q_j := q_j + \sum_i Q_j[i] (x_m)^i$
11: **end for**
12: $(\tilde{q}, r) :=$ DIVIDE(r, t_m); $q_m := q_m + \tilde{q}$
13: **for** $i = 0, \ldots, \deg(r, x_m)$ **do**
14: $(\{Q_j[i]\}_{j=1}^{m-1}, R[i]) := \mathrm{TSNF}(\mathrm{coef}(r, x_m, i), \{t_j\}_{j=1}^{m-1})$
15: **end for**
16: **execute** Lines 8-11
17: **end if**
18: **return** $(q_1, \ldots, q_m, r)$

Algorithm 8. recTSPD (a,T)
Same input and output specifications as Algorithm 6.

1: **if** $N = 1$ **then**
2: $(q_1, r, e) :=$ PSEUDODIVIDE$(a, t_1, \mathrm{mvar}(t_1))$; $h = \mathrm{init}(t_1)^e$
3: **else**
4: $v := \mathrm{mvar}(t_N)$
5: **for** $i = 0, \ldots, \deg(a, v)$ **do**
6: $(\{Q_j[i]\}_{j=1}^{N-1}, R[i], H[i]) :=$ recTSPD$(\mathrm{coef}(a, v, i), \{t_j\}_{j=1}^{N-1})$
7: **end for**
8: $q_1 := 0; \ldots; q_N := 0$
9: $H_1 := \mathrm{lcm}(H[i], 0 \le i \le \deg(a, v))$
10: $r := \sum_i \frac{H_1}{H[i]} R[i] v^i$
11: **for** $j = 1, \ldots, N-1$ **do**
12: $q_j := q_j + \sum_i \frac{H_1}{H[i]} Q_j[i] v^i$
13: **end for**
14: $(\tilde{q}, r, \tilde{e}) :=$ PSEUDODIVIDE(r, t_N, v); $\tilde{h} = \mathrm{init}(t_N)^{\tilde{e}}$
15: **for** $j = 1, \ldots, N-1$ **do**
16: $q_j := q_j \tilde{h}$
17: **end for**
18: $q_N := q_N + \tilde{q}$
19: **for** $i = 0, \ldots, \deg(r, v)$ **do**
20: $(\{Q_j[i]\}_{j=1}^{N-1}, R[i], H[i]) :=$ recTSPD$(\mathrm{coef}(r, v, i), \{t_j\}_{j=1}^{N-1})$
21: **end for**
22: $H_2 := \mathrm{lcm}(H[i], 0 \le i \le \deg(r, v))$
23: **for** $j = 1, \ldots, N$ **do**
24: $q_j := q_j H_2$
25: **end for**
26: **execute** Lines 10-13 **with** H_2 replacing H_1
27: $h := H_1 \tilde{h} H_2$
28: **end if**
29: **return** $(q_1, \ldots, q_N, r, h)$

Algorithm 6 is a direct (or naïve) procedure for computing the pseudo-remainder and the pseudo-quotients of a polynomial $a \in \mathbb{K}[x_1, \ldots, x_m]$ by a

triangular set $T = \{t_1, \ldots, t_N\}$. Note that T may not be zero-dimensional, that is, $N < m$ may hold. Moreover, T may not be normalized; in particular its initials may not be constant. Algorithm 8 is a recursive version of Algorithm 6 following the same principles as Algorithm 7 and calling Algorithm 4 at Line 14.

References

1. Arnold, A., Roche, D.S.: Output-sensitive algorithms for sumset and sparse polynomial multiplication. In: Proceedings of ISSAC 2015, pp. 29–36 (2015). http://doi.acm.org/10.1145/2755996.2756653
2. Bronstein, M., Moreno Maza, M., Watt, S.: Generic programming techniques in ALDOR. In: Proceedings of AWFS 2007, pp. 72–77 (2007)
3. Chen, C., Covanov, S., Mansouri, F., Maza, M.M., Xie, N., Xie, Y.: Parallel integer polynomial multiplication. CoRR abs/1612.05778 (2016). http://arxiv.org/abs/1612.05778
4. Chen, C., Moreno Maza, M.: Algorithms for computing triangular decomposition of polynomial systems. J. Symb. Comput. **47**(6), 610–642 (2012). https://doi.org/10.1016/j.jsc.2011.12.023
5. Gastineau, M., Laskar, J.: Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2013. LNCS, vol. 8136, pp. 100–115. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02297-0_8
6. Gastineau, M., Laskar, J.: Parallel sparse multivariate polynomial division. In: Proceedings of PASCO 2015, pp. 25–33 (2015). http://doi.acm.org/10.1145/2790282.2790285
7. Granlund, T., et al.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited (2015)
8. Hall Jr., A.D.: The ALTRAN system for rational function manipulation-a survey. In: Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, pp. 153–157. ACM (1971)
9. Hart, W., Johansson, F., Pancratz, S.: FLINT: Fast Library for Number Theory, v. 2.4.3. http://flintlib.org
10. van der Hoeven, J., Lecerf, G.: On the bit-complexity of sparse polynomial and series multiplication. J. Symb. Comput. **50**, 227–254 (2013). https://doi.org/10.1016/j.jsc.2012.06.004
11. Johnson, S.C.: Sparse polynomial arithmetic. ACM SIGSAM Bull. **8**(3), 63–71 (1974)
12. Leiserson, C.E.: Cilk. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 273–288. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4_289
13. Lemaire, F., Maza, M.M., Xie, Y.: The regularchains library in MAPLE. ACM SIGSAM Bull. **39**(3), 96–97 (2005). https://doi.org/10.1145/1113439.1113456
14. Li, X., Maza, M.M., Schost, É.: Fast arithmetic for triangular sets: from theory to practice. J. Symb. Comput. **44**(7), 891–907 (2009). https://doi.org/10.1016/j.jsc.2008.04.019
15. Monagan, M.B., Pearce, R.: Parallel sparse polynomial multiplication using heaps. In: ISSAC, pp. 263–270 (2009)
16. Monagan, M., Pearce, R.: Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2007. LNCS, vol. 4770, pp. 295–315. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75187-8_23

17. Monagan, M., Pearce, R.: Parallel sparse polynomial division using heaps. In: Proceedings of PASCO 2010, pp. 105–111. ACM (2010)
18. Monagan, M., Pearce, R.: The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. ACM Commun. Comput. Algebra **48**(3/4), 166–186 (2015)
19. Monagan, M.B., Pearce, R.: Sparse polynomial division using a heap. J. Symb. Comput. **46**(7), 807–822 (2011). https://doi.org/10.1016/j.jsc.2010.08.014
20. Moreno Maza, M., Xie, Y.: Balanced dense polynomial multiplication on multicores. Int. J. Found. Comput. Sci. **22**(5), 1035–1055 (2011)
21. Shoup, V., et al.: NTL: A library for doing number theory. www.shoup.net/ntl/