Polynomial Division using Dynamic Arrays, Heaps, and Packed Exponent Vectors

Michael Monagan

Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.

Roman Pearce

Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.

Abstract

A common data structure that is used for multivariate polynomials is a linked list of terms sorted in a term ordering. When dividing polynomials using this data structure, polynomial addition and subtraction is done using merging. This can result in poor performance on some kinds of divisons that commonly arise in practice.

In this paper we consider using an auxilliary heap of pointers to reduce the number of monomial comparisons needed in the worst case where the size of the heap is linear in the number of terms in the quotient(s). We present variations that are designed to improve performance for dense polynomials and divisions with large quotients.

We have implemented the heap algorithms in C with an interface from Maple. We also encode and pack monomials to speed up monomial comparisons. We give some timings demonstrating that a heap of pointers is efficient.

Key words: Polyomial division, Gröbner bases, data structures, heaps, polynomial GCDs.

⋆ This work was supported by NSERC of Canada and the MITACS NCE of Canada

Email addresses: `mmonagan@cecm.sfu.ca` (Michael Monagan), `rpearcea@cecm.sfu.ca` (Roman Pearce).

URLs: `http://www.cecm.sfu.ca/~mmonagan` (Michael Monagan), `http://www.cecm.sfu.ca/~rpearcea` (Roman Pearce).

Preprint submitted to

1. Introduction

In this paper we present and compare algorithms and data structures for polynomial division in the ring $F[x_1, x_2, ..., x_n]$ where F is a field. We are interested in (i) exact division of f by a single polynomial g, that is testing if $g|f$, (ii) exact division of f by a polynomial g modulo a triangular set of polynomials $\{m_1, m_2, ..., m_s\}$, and (iii) division with remainder of f by a set of polynomials $\{g_1, g_2, ..., g_s\}$. Because applications of division use modular algorithms, that is, the compute modulo primes, we will want to divide over characteristic p as well as characteristic 0. Here p will be a machine prime, small enough so that arithmetic in $\mathbb{Z}_p$ can be done directly by the hardware of the computer.

The first case of division is needed for rational function arithmetic over $\mathbb{Q}$ and polynomial GCD computation in $\mathbb{Z}[x_1, ..., x_n]$. The second case is needed for rational function arithmetic and polynomial GCD computation over algebraic number and function fields. The third case of division is important for Gröbner basis computation and computation in quotient rings. In all algorithms for computing Gröbner bases, the most time consuming operation is division.

Modular GCD algorithms compute the gcd G of two polynomials A and B modulo a sequence of primes and evaluation points and then reconstruct G from multiple images. These algorithms use trial division to test if G has been reconstructed correctly and division to compute the cofactors A/G and B/G. In the trial divisions, one wants to stop the division algorithm as soon as it is known that the remainder is non-zero.

The development of efficient modular algorithms for polynomial GCD computation has a long history. Many of the algorithms are described in Geddes et. al. (4). We mention Zippel's sparse modular GCD algorithm in (13) for polynomial GCD computation in $\mathbb{Q}[x_1, ..., x_n]$. Zippel's algorithm is used in Mathematica and in Magma (since version 2.12) and a newer variation on the algorithm is now being used in Maple (since version 11). The algorithm of van Hoeij and Monagan in (6) computes polynomial GCDs over an algebraic function field. It does trial division of f divided g modulo $m(z)$ in characteristic 0 and p. Modular algorithms are also used to compute Gröbner bases. We mention the work of Arnold in (1).

Term orderings and the division algorithm

We consider distributed polynomial representations that assume the terms in the polynomial are ordered in a term ordering. See Cox, Little, O'Shea (3) for background material on term orderings. The term ordering that we are mainly interested in in the paper is graded lexicographical ordering (grlex). In this ordering one first orders by total degree and then by lexicographical order. For example, the polynomial

$$-9x^4 - 7x^3yz + 6x^2y^3z + 8y^2z^2$$

when written with terms in descending grlex order with $x > y > z$ is

$$6x^2y^3z - 7x^3yz + 8y^2z^3 - 9x^4.$$

The data structure used to represent a polynomial will also have a direct impact on the efficiency of the division algorithm. The data structure used for Gröbner basis computation in the Axiom computer algebra system is the SDMP (Sparse Distributed Multivariate Polynomial) data structure (7). This is a linked list of terms where each term is a pair (c, e) where c is a (pointer to) a coefficient and e is a pointer to the exponent

vector, an array of machine integers. Using $\langle a, b, c, ...\rangle$ to denote an array, $[a, b, c, ...]$ to denote a linked list, and (c, e) to denote a pair, the polynomial in example 1 would be stored as

$$[\,(6, \langle 2, 3, 1\rangle),\ (-7, \langle 3, 1, 1\rangle),\ (8, \langle 0, 2, 3\rangle),\ (-9, \langle 4, 0, 0\rangle)\,]$$

The same data structure is used by Bachmann and Schönemann in (2). They show that by packing the exponent vectors, Gröbner basis computations modulo a machine prime p using this data structure a modest speedup is obtained.

We now state the division algorithm. Our purpose is to understand where the SDMP data structure and the division algorithm are inefficient. Following the notation of Cox, Little, O'Shea in (3), we use $LT(f)$, $LM(f)$ and $LC(f)$ to refer to the leading term, leading monomial and leading coefficient of a polynomial f, respectively, all with respect to the term ordering being used. These satisfy $LT(f) = LC(f)LM(f)$.

The Division Algorithm.
Input: $f, g_1, g_2, ..., g_s \in F[x_1, ..., x_n], F$ a field.
Output: $q_1, q_2, ..., q_s, r \in F[x_1, ...x_n]$ satisfying $f = q_1g_1 + q_2g_2 + ... + q_sg_s + r$.

1: Set $(q_1, q_2, ..., q_s) := (0, 0, ..., 0)$.
2: Set $p := f$.
3: While $p \neq 0$ do
    4: Find the first g_i s.t. $LM(g_i)|LM(p)$.
    5: If no such g_i exists then set $r := r + LT(p)$ and $p := p - LT(p)$
    6: else set $(q_i, p) := (q_i + t, p - t \times g_i)$ where $t = LT(p)/LT(g_i)$.
7: Output $(q_1, q_2, ..., q_s, r)$.

Remark: If one wishes to test if $(g_1, ..., g_s)|f$ with 0 remainder, then Step 5 should be modified to stop execution and output false.

Using a linked list for the terms, assuming the terms of the polynomial are sorted in descending order in the term ordering, accessing the leading term $LT(f)$ takes constant time, the operation $p - LT(p)$ (link to the remaining terms of p) is constant time and also $r + LT(p)$ (append a new term to the remainder r) is constant time. The most expensive step is the subtraction $p - t \times g_i$. This requires is a "merge" – one simultaneously walks down the linked list of terms in p and the linked list of terms in g_i. In the worst case the merge compares $N + M - 1$ monomials where N is the number of terms in p and M is the number of terms in g_i.

The problem of indirect references

There are three sources of inefficiency in this classical division algorithm when we use the SDMP data structure. The first is the many intermediate pieces of storage that need to be allocated when we multiply tg_i, for example, storage for new exponent vectors. The second is the indirect references that occur during the merge when we walk down a linked list, and for each term, link to the exponent vector in order to compare monomials. These indirect references cause a significant loss in efficiency when the polynomials are too large to fit inside the computer's cache which is typically 1 or 2 Megabytes on todays Intel and AMD processors. On an AMD Opteron 150 running at 2.4 GHz with 400 MHz RAM we measured the loss of speed at a factor of 6. We contend in this paper that the SDMP data structure with its many pointers is not suitable for todays computers where

the speed of RAM is significantly slower than the speed of the primary and secondary cache and the size of the RAM is typically 1000 times larger than the size of the cache.

The data structure we shall use to represent polynomials is a single array with coefficients and exponents encoded in the array. For example, we would represent the polynomial above as the array

$$\langle\ 6, 2, 3, 1, -7, 3, 1, 1, 8, 0, 2, 3, -9, 4, 0, 0\ \rangle$$

One computes the difference $p - tg_i$ using merging efficiently as follows. We use two arrays, one, p, that we are copying terms out of and the other, q, that we are forming the difference $p - t \times g_i$ inside. Should it happen that q is not large enough to hold $p - t \times g_i$ one must increase the size of q. To make this efficient one should increase the size of q beyond the minimum required length by, for example, 50%. After $p - t \times g_i$ is computed in q, interchange the roles of p and q for the next iteration of the division algorithm. Thus the arrays p and q are dynamic, they grow in size to accommodate the size of the differences as needed.

To make this strategy very efficient the arrays p and q are reused in the next call to the division algorithm. Over the course of a computation that does many divisions, the size of the global arrays p and q will grow quickly to accommodate the largest polynomials encountered. In this way we eliminate almost all storage management overhead from the division algorithm. Also, since the merging in the division algorithm goes through the terms of the arrays p, g_i and q sequentially, we eliminate the problem of indirect references repeatedly going outside the cache.

But, there is a loss of efficiency – instead of copying pointers (one word) we must now copy the contents of the exponent vectors (n words). However, as we shall see, if we pack the exponents (4 or 8 per 64 bit word), this representation is very good when the number of variables is not too high (say 24 or less).

The problem of too many monomial comparisons.

The third problem occurs in the subtraction $p - tg_i$ when the number of terms in p is much larger than the number in g_i. To see what the problem is, suppose that we are dividing a polynomial f with N terms by a single polynomial $g_1 = x^2 - 3$ with two terms and suppose the quotient q has $M = 1000$ terms and suppose $N = 2M$. The cost of multiplying $q_1 \times g_1$ requires $2M$ coefficient multiplications in F. If we multiply $q_1 \times g_1$ by adding $x^2 q_1$ to $-3q_1$ using merging, we do at most $2M - 1$ comparisons. Dividing p by g_1 the division algorithm subtracts $t \times g_1$ from p. Since the leading terms cancel one simply subtracts $-3t$ from the remaining terms of p. But to find if p has a term with monomial $LM(t)$ requires walking down the linked list to find it, potentially, looking at all 1999 terms in p. This leads to a division algorithm which does $O(N^2)$ comparisons, much worse than multiplication.

One possible solution would be to represent the polynomials as binary search trees. In this representation, all operations, compute $\mathrm{LT}(p)$, $p + t$, $p - t$ can be computed in $O(\log N)$ monomial comparisons where p has N terms. But a binary tree, like a linked list, is based entirely on pointers, and will, for the same reason, be inefficient when p is too big to fit inside the cache.

The "geobucket" data structure of Yan in (12), which is used by Singular computer algebra system (see (5)), can be viewed as a compromise between a linked list and a binary tree. In the "geobucket" data structure the polynomial p is represented as an

array of $O(\log N)$ "buckets" where the i'th bucket p_i is a linked list of at most 2^i terms. To subtract $t \times g_i$ from p one subtracts it from the smallest bucket in which it would fit if there were no cancellation of terms. If no such bucket exists then a new bucket is created. Subtraction is done by merging two linked lists. The idea is that asymptotic efficiency is not lost if when we merge two linked lists the number of terms is similar, within a factor of two say. Yan proves that the geobucket data structure yields a worst case complexity which is comparable to that of a binary tree.

In this paper we consider using a "heap" of pointers. We keep the array representation for the polynomials p, g_i and q_i, because it is compact, though a linked list of terms could be used. When doing a division, we maintain a heap of pointers back into the terms of the divisors $g_1, g_2, ..., g_s$ which indicate which terms have yet to be subtracted from p. We emphasize that we do not maintain a heap of all the terms or a heap of polynomials to be added because this can result in a space blowup. Not counting the space required for the coefficients, the total space that we use is linear in the number of terms in f, the divisors g_i, and the quotients q_i.

The main advantage of using a heap is that the worst case complexity improves to $O(NM \log M)$ comparisons where M is the number of terms in the quotient and N is the number in the divisor. Another advantage is that we can delay coefficient arithmetic until we need to do it. This may result in a significant speedup when we want to test if a polynomial g divides f.

The main disadvantage of using a heap is that for dense polynomials, where the merge algorithm gives an $O(NM)$ algorithm, the heap imposes a significant overhead of a $\log M$ factor. A second disadvantage is that although a heap is implemented inside an array, the entries in the array are not accessed sequentially, and hence, if the heap becomes larger than the computer's cache, we may have a memory access problem.

The idea of using a heap for sparse polynomial arithmetic was first investigated by Johnson in 1974 (8) for sparse univariate polynomial multiplication. Although a heap gives a good worst case performance, heaps have not, as far as we are aware, been used by any of the general purpose computer algebra systems. Heaps were not considered at all by Stoutemyer in (11) which, as far as we are aware, is the only systematic experiment ever done comparing different polynomial data structures on a computer algebra system's test suite.

Our paper is organized as follows. In Section 2 we describe how we encode and pack monomials. In Section 3 we give the main algorithm which uses a heap. We show how to optimize the heap so that it also works well for the dense case and also how to invert the heap so that the size of the heap depends on the number of terms in the divisors $g_1, g_2, ..., g_s$ instead of the number of terms in the quotients $q_1, q_2, ..., q_s$. Although this is not expected to be of much benefit to Groebner basis computation where the number of terms in the divisors $g_1, g_2, ..., g_s$ is usually more than those in the quotients, for polynomial GCD computation, it is usually the case that the gcd G of two polynomials A and B is smaller, often much smaller, than the quotients A/G and B/G.

We have implemented the division algorithm using both merging and a heap in the C programming language. We create polynomials in Maple and call our C code from Maple using Maple's foreign function interface – see Ch 8 of (10). This work is part of a project to design a new distributed polynomial data structure for Maple.

In Section 4 we report on some timings comparing the array representation using merging with packed and unpacked exponent vectors, with the heap representation, again

with packed and unpacked exponent vectors. Our conclusions may be summarized as follows; the simple array representation with merging works well on many problems, but the heap representation is faster for problems with small divisors, big dividends and big quotients and packing exponent vectors really helps with both representations.

2. Monomial Representations

Our choice of a monomial representation is motivated by an analysis of sparse polynomial arithmetic. As noted by Johnson (8), the sparse multiplication of two sparse polynomials with N terms and M terms respectively must construct all NM products of terms because the monomials generated may be distinct. Sparse polynomial division is essentially equivalent to a multiplication; to divide a polynomial f by g we construct a quotient q incrementally while computing $f - gq$. If g has N terms and q has M terms this also constructs NM products; the M terms of the quotient plus the $(N-1)M$ terms of $q(g - LT(g))$. To construct a remainder r with $f = gq + r$, the number of monomial divisions required is precisely the number of terms in q plus the number of terms in r.

Our desire to sort the result imposes an additional factor on the number of monomial comparisons. Without this requirement the NM terms could be coalesced in $O(NM)$ time using a hash function, but sorting them requires at least $O(NM(\log N + \log M))$ comparisons. If you know that one of the polynomials is sorted you can exploit this fact to reduce the number of comparisons to $O(NM \log N)$, but in either case the extra logarithmic factor is significant. It means that asymptotically, monomial comparisons can dominate a sparse polynomial computation if there are a large number of variables and the coefficient arithmetic is inexpensive. We suspected that this could be the case for some Gröbner basis computations over small finite fields.

For our implementation we chose to make monomial comparisons and multiplications as fast as possible, accepting an added inconvenience for the less frequent monomial divisions. We store monomials as a sequence of weighted degrees with respect to a positive integer matrix defining the monomial order. For example, under graded lexicographic order (grlex) and graded reverse lexicographic order (grevlex) with $x > y > z$ we would store the monomial $x^1y^3z^2$ as:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 1 \\ 3 \end{bmatrix}_{grlex} \qquad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 1 \end{bmatrix}_{grevlex}$$

Notice that the first component, 6, is the total degree of the monomial $x^1y^3z^2$ (for both orderings). If one uses just the exponent vector $[1, 3, 2]$ to compare two momomials in a graded ordering, one must first compute the total degree of the monomials which would mean looking at all n exponents. In one of the representations considered by Bachmann and Schönemann in (2), the authors explicitly store the total degree as well as the exponent vector, that is, for the monomial $x^1y^3z^2$ they store [6,1,3,2]. Notice also that in this matrix encoding, if we pack the first four components into a single word, then we will have quite a lot of information about the monomial in one word, enough, we hope, so that in most cases, a comparison of two non equal monomials can be decided in just one machine instruction.

Every monomial order with a rational matrix representation has a representation as a positive integer matrix. In order for 1 to be the smallest monomial with respect to the order, the first non–zero entry of each column in the weight matrix must be non–negative. Then we can inductively eliminate negative entries in each row by adding multiples of previous rows, and clear denominators to obtain a positive integer matrix.

Under this scheme all monomial comparisons are reduced to lexicographic comparisons but monomial multiplications can still add exponent vectors because multiplication by the weight matrix is a linear transformation. Monomial division becomes more complicated. Subtracting exponent vectors produces a candidate quotient, but we must check that this candidate is a valid monomial. For lexicographic order we only need to check that each exponent is non–negative. For other monomial orders this condition is also neccesary because the weight matrices are positive, but in general it is not sufficient.

Example 1. Does y^2 divide x^3 in grevlex order with $x > y > z$?

$$x^3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}_{grevlex} \qquad y^2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}_{grevlex}$$

One solution to this problem is to apply the inverse transformation to the quotient and check that the resulting exponents are non–negative. For general matrix orders this is required, but for many important monomial orders it is possible to do a faster check. For example, weighted degree orders (including grevlex) that compare first with respect a vector of positive weights followed by a reverse lexicographic comparison have a positive integer matrix with the following form:

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & 0 \\ w_1 & w_2 & 0 & 0 \\ w_1 & 0 & 0 & 0 \end{bmatrix}$$

Given a vector of transformed exponents $[\, e_0, \ldots, e_n \,]$ we would need to check that the e_i are in descending order, that e_n is a multiple of w_1, and that $e_i - e_{i+1}$ is a multiple of w_{n-i+1} for $1 \leq i \leq n-1$. This is equivalent to testing whether the inverse transformation constructs a monomial with non–negative integer exponents. For grevlex order the divisibility tests are redundant since each $w_i = 1$, so we only need to check that the transformed exponents are in descending order.

Weighted lexicographic orders (including grlex) are somewhat easier to deal with. The weight matrix for these orders consists of a positive vector $[\, w_1, \ldots, w_n \,]$ followed by an $(n-1) \times (n-1)$ identity matrix directly below it, augmented by a column of zeroes. Given a vector $[\, e_1, \ldots, e_n \,]$ of transformed exponents, we need to check that $e_1 - \sum_{i=1}^{n-1} e_{i+1} w_i$ is non–negative and divisible by w_n. In ordinary grlex order $w_i = 1$ for all i, the divisibility test can be skipped, and the dot product becomes a simple sum.

In our implementation these checks slowed down successful grevlex divisions by almost a factor of four, although when division failed this was usually detected early by a check for non–negative exponents. Exact polynomial division, where every monomial division is successful, ran about two percent slower in grevlex order compared to ordinary

lexicographic order. The results for graded lexicographic order showed a similar performance hit for successful divisions, but failures were much more likely to be detected early because all but one of the transformed exponents are independent.

Finally we wanted to test the idea of packing multiple exponents into a single machine word. This idea, first used by Bachmann and Schönemann (2) in the Singular computer algebra system ((5)), speeds up all monomial operations by reducing the size of monomials and the number of instructions needed to operate on them. Our exponents are always positive, so we pack $[\, e_1, \ldots, e_k \,]$ using a base $B = 2^w$ and the formula $e = \sum_{i=1}^{k} e_i B^{k-i}$. One bit is used for each exponent to detect overflow, so that the largest representable exponent is $2^{w-1} - 1$. We do not require w to divide the wordsize of the machine, so we can, for example, pack three 20 bit exponents into a 64 bit word. Our implementation also allows us to distribute the bits unevenly among the exponents, however this feature was not used in our tests.

For graded orderings, the total degree of the largest monomial that can appear in the division algorithm cannot be larger than the maximum of the total degrees of $LT(f), LT(g_1), ..., LT(g_s)$. Thus we can dynamically pack the exponent vectors at start of the division algorithm.

Overall, this monomial representation is not as good as we had hoped. The added cost and complexity of monomial divisions probably cancelled any gains from using a simple and fast comparison function. This is especially true when dividing by a large set of polynomials, since monomial division must loop over the entire set of the divisors to construct each term of the remainder.

Another problem with this representation is the storage required for weighted degree orders, which compare first using a vector of weights followed by a reverse lexicographic comparison. In our encoding every exponent is proportional to the weighted degree of the entire monomial, which can be quite large if large weights are used. These large exponents can not be packed efficiently. The representation of Bachmann and Schönemann (2) stores the weighted degree followed by the original exponents, which are easily packed, and so is much more efficient in this case.

It should be pointed out that wasted storage is not a problem for the grlex and grevlex orders which we originally designed for. In this case one typically wants to represent all monomials of degree less than or equal to d, and doing so requires that we represent x_i^d for $1 \leq i \leq n$. Thus any exponent is potentially equal to the maximum total degree, and there is no wasted storage versus storing the exponents themselves.

3. Heap Algorithms for Division

In this section we describe multivariate division using a heap, which has $O(NM \log N)$ complexity and uses $O(N)$ temporary storage. N and M are the total number of terms in the quotient(s) and divisor(s), respectively. The algorithm performs an N–ary merge using a pointer into the dividend f, pointers into the divisors $\{g_1, \ldots, g_s\}$, and pointers into the quotient $Q = \sum_{j=1}^{N} q_j$.

The Division Algorithm (Quotient Heap).
Input: $f, g_1, g_2, ..., g_s \in F[x_1, ..., x_n], F$ a field.
Output: $q_1, q_2, ..., q_s, r \in F[x_1, ...x_n]$ satisfying $f = q_1 g_1 + q_2 g_2 + ... + q_s g_s + r$.

1: Set $Q := \{\}, r := 0$.
2: Initialize a heap H with $H_1 := (t, p, q)$ where:
  p points to the first term of f,
  q points to a multiplier equal to -1
  t is the product $p_{term} \cdot q_{term}$

In general:
p will point to a term of f or one of the g_i's
q will point a term of the quotient or to -1 if p points to a term of f
t is the product $p_{term} \cdot q_{term}$

3: While $H \neq \emptyset$ do
  4: $(t, p, q) := H_1$ the maximum element of the heap
  5: remove H_1 from the heap and initialize $T := -t$
  6: if p does not point at the last term of its polynomial
    7: increment $p \leftarrow p_{next}$ and update $t := p_{term} \cdot q_{term}$
    8: insert the new (t, p, q) into the heap
  9: While $H \neq \emptyset$ and $LM(H_1) = LM(T)$ do
    10: $(t, p, q) := H_1$, remove H_1 from the heap
    11: $T := T - t$
    12: if p does not point at the last term of its polynomial
      13: increment $p \leftarrow p_{next}$ and update $t := p_{term} \cdot q_{term}$
      14: insert the new (t, p, q) into the heap
  15: if $T \neq 0$ then
    16: Find the first g_i with $LT(g_i)|T$.
    17: If no such g_i exists set $r := r + T$
    18: else:
      19: let p point to the first term of g_i
      20: append a new term to the quotient $Q = Q + (T/LT(g_i), i)$
      21: let q point to this new term of Q
      22: assign $t := p_{term} \cdot q_{term}$ and insert (t, p, q) into the heap
23: do a linear pass of Q to construct each quotient q_i
24: Output $(q_1, q_2, ..., q_s, r)$.

We list some details which are important for an implementation. First the heap H has to grow, potentially to size $|Q| + 1$, where $|Q| = \sum_{i=1}^{s} |q_i|$. In practice the heap is implemented inside an array, so enlarging it involves copying its contents to a new location. We suggest enlarging the heap by 50% beyond the required size each time to avoid enlarging it frequently. Also, the heap H should be reused in the next division so that in a computation which does many divisions, the heap grows quickly to the maximum size needed and no further allocations occur.

Second, inserting new terms immediately after extracting terms (steps 5 and 8) is slow. It is faster to move the terms to a queue and to insert the entire queue at the end of the main loop. On dense problems where the heap extracts a large number of terms in each iteration, a shrinking heap makes subsequent extractions faster. There are also caching

advantages to be gained from inserting multiple elements at a time. Each pair of elements that are inserted consecutively have a 50% chance of having the same parent and a 75% chance of having the same grandparent (9), so the monomials that we compare against are likely to be in the L1 cache.

The complexity of this algorithm is $O(NM \log N)$, where N is the number of quotient terms and M is the number of terms in the divisor(s). A total of $\sum_{i=1}^{s} |q_i||g_i| \in O(NM)$ terms are inserted into and extracted from the heap, and each of these operations requires $O(\log N)$ comparisons. In our implementation the average number of comparisons for insertion is $O(1)$ and extracting the largest element uses $\log_2 N + O(1)$ comparisons. The temporary storage required is $O(3N)$, consisting of the quotient, the heap, and the current term of each $q_j g_i$.

The main disadvantage with the heap algorithm is that when the polynomials are dense it is asymptotically slower than algorithms based on merging. Both a repeated merge and a divide–and–conquer merge will do $O(NM)$ in the dense case, and divide–and–conquer is $O(NM \log N)$ for the sparse case as well, although it has an $O(NM)$ temporary storage requirement. We wanted to fix this problem with the heap algorithm and retain $O(N)$ temporary storage.

The solution we came up with is similar to Johnson's (8), however we will prove that only $O(NM)$ comparisons are needed for a totally dense problem. The idea is to allocate an additional buffer whose size is equal to the heap, and to chain equal elements using pointers into this buffer. Chains are created only when terms are inserted into the heap and the comparison function detects equal monomials. We make no attempt to chain elements as terms are extracted from the heap. If the largest element of the heap is a chain, all of its terms are extracted using one comparison and the other elements of the heap are promoted as usual.

Lemma 2. *Let f and g be dense polynomials with N terms and M terms respectively. Then the division algorithm with a chained heap and a reinsertion queue computes a quotient q and remainder r with $f = qg + r$ using $O(NM)$ monomial comparisons.*

Proof. We prove the following loop invariant: at the beginning of the main loop, the heap contains exactly one element or chain. This is certainly true initially since the only element in the heap is $LT(f)$. In each iteration of the algorithm this element or chain of elements is extracted using one comparison producing an empty heap. The polynomials f, and $q_j g$ for $1 \leq j \leq k$ are moved to the reinsertion queue, and their next terms are inserted into the heap. However these terms are all equal, because polynomials are dense and their previous terms were equal. If a new divisor $q_{k+1} g$ is added, its first term cancels the term extracted from the heap, and its next term equals all the other next terms because all the polynomials are dense. Thus in each iteration of the main loop, we add some equal terms to the heap, and these terms are chained to the top element using one comparison. The total number of terms is bounded by NM. $\square$

Remark: In our statement of the lemma, N is equal to the number of terms in f, not the number of terms in the quotient. It actually doesn't matter whether the quotient is dense or not, because each $q_j g$ is dense. In the univariate case our upper bound can be improved to $NM + M$ comparisons where $|q| = N$, $|g| = M$, and $|f| = N + M$.

Finally, we mention a unique approach to division that is asymptotically faster when the quotient is larger than the divisor. One situation where this occurs is in dividing large

polynomials by a (set of) small field extensions. Many terms may be reducible, producing a large quotient. Another case was discussed in the introduction, where we divide two polynomials A and B by a candidate for $\gcd(A, B)$ in order to obtain the cofactors and check whether the gcd is correct.

Recall that sparse division essentially computes qg and subtracts it from f. In the algorithm above, we use a heap of size $|q| + 1$ to do an n–ary merge of f and the polynomials $q_j g$. What if instead we merged f with the products $g_i q$? This option is not available in merging algorithms because the terms of q are not all known, but with the heap algorithm it can be made to work. The key observation is that each term q_j of the quotient is constructed using $LT(g)$ before any of the terms of $(g - LT(g))q_j$ need to be merged with f. If $f = qg + r$ where $|g| = M$ and $|q| = N$, this algorithm uses heap of size M to do the merge, with a pointer into f and into each $g_i q$. The complexity of this approach is $O(NM \log M)$, which is linear in the number of division steps. We state the algorithm using a single divisor g, although it is easily adapted to handle multiple divisors.

The Division Algorithm (Divisor Heap).
Input: $f, g \in F[x_1, ..., x_n], F$ a field.
Output: $q, r \in F[x_1, ...x_n]$ satisfying $f = qg + r$.

1: Set $Q := \{\}, r := 0$.
2: Initialize a heap H with $H_1 := (t, p, q)$ where:
  p points to a multiplier -1
  q points to the first term of f
  t is the product $p_{term} \cdot q_{term}$

3: Initialize a "missing terms" queue M with $M_i = (t_i, p_i, q_1)$ for $1 \leq i \leq |g| - 1$ where:
  p_i points to term $i + 1$ of g,
  q_i points to the first term of the quotient
  t is empty storage

3: While $H \neq \emptyset$ do
  4: $(t, p, q) := H_1$ the maximum element of the heap
  5: remove H_1 from the heap and initialize $T := -t$
  6: if q points to the last term of the quotient
    7: add (t, p, q) to M
  8: else if q does not point to the last term of f
    9: increment $q \leftarrow q_{next}$ and update $t := p_{term} \cdot q_{term}$
    10: insert the new (t, p, q) into the heap
  11: While $H \neq \emptyset$ and $LM(H_1) = LM(T)$ do
    12: $(t, p, q) := H_1$, remove H_1 from the heap
    13: $T := T - t$
    14: if q points to the last term of the quotient
      15: add (t, p, q) to M
    16: else if q does not point to the last term of f
      17: increment $q \leftarrow q_{next}$ and update $t := p_{term} \cdot q_{term}$
      18: insert the new (t, p, q) into the heap
  19: if $T \neq 0$ then

20: if $LT(g)|T$ then
21: make a new term of the quotient $Q = Q + (T/LT(g), i)$
22: for all $(t, p, q) \in M$ do
23: increment $q \leftarrow q_{next}$
24: update $t := p_{term} \cdot q_{term}$
25: move (t, p, q) to the heap
26: else set $r := r + T$
27: Output (Q, r).

The algorithm extracts products $g_i q_j$ from the heap and checks whether q_{j+1} has been computed. If it has, then the next product $g_i q_{j+1}$ is constructed and inserted into the heap. Otherwise we move $g_i q_{j+1}$ to a queue for polynomials that are "missing terms" until q_{j+1} is computed. If q_{j+1} is ever constructed we compute $g_i q_{j+1}$ and insert it into the heap. The algorithm terminates when all the polynomials are "missing terms" and the heap is empty, which implies that no more terms will be constructed.

Our comments about using a reinsertion queue and chaining heap elements apply equally to this algorithm. One nice feature of both heap algorithms is that they delay all coefficient arithmetic until the last possible moment. This means that if we are testing whether $g|f$ and the answer is no, we will do the minimum amount of arithmetic required to find that out. That is, we will compute all terms up to and including the first remainder term and nothing more.

We can also exploit the lazy nature of the algorithm to do the coefficient arithmetic more efficiently. For example, with coefficients modulo a small prime p we can predict how many products can be added up before an overflow can occur, and reduce the sum mod p only when it is required. We implemented this technique, however the savings are relatively small and only noticable on dense problems. This feature may be much more important if the coefficients are sparse polynomials however, because as this paper endeavors to show, repeatedly adding sparse polynomials to an accumulating sum can be a bad idea.

Finally, the heap algorithms can be expected to perform well on very large problems where the intermediate products accumulating in the merge algorithm might not fit in main memory. The heap algorithms, by comparison, use an amount of working memory that is proportional to the size of the input and to the size of the output.

4. Benchmarks

We conducted benchmarks to test the performance of the merge and heap algorithms as well as the effectiveness of packing exponents. The tests were performed on a 2.4 GHz Intel Core 2 Duo with 2 GB of RAM and 4 MB of L2 cache, running 32–bit Linux. In addition to the times, we report the number of monomial comparions, the number of times a monomial was copied, and the average number of exponents compared using the unpacked representation. These statistics were gathered by repeating the computations with debugging code enabled.

Our first example simulates a GCD problem, where a large polynomial is divided by one of its factors to compute the cofactor and check the GCD. We constructed three random polynomials $\{f_1, f_2, f_3\}$ and computed their product p in $\mathbb{Z}_{32003}[x, y, z, w]$. Then we divided p by f_1 and $f_1 f_2$ to test exact division with both a small factor and a large

factor. The polynomials f_i have 96, 93, and 93 terms respectively, with random exponents from 0 to 30. The product $p = f_1 f_2 f_3$ has 795357 terms and $f_1 f_2$ has 8922 terms. The division used lexicographic order with $x > y > z > w$. All times are in seconds, and the column headings indicate the number of exponents packed into each 32–bit word.

p/f_1	comparisons	copies	depth	1 exp/word	2 exp/word	4 exp/word
div. heap	19392137	804191	3.08	.416 s	.324 s	.255 s
quo. heap	32564804	812829	2.70	.637 s	.470 s	.394 s
merge	1971092135	3463979612	1.60	45.592 s	37.368 s	25.102 s
$p/(f_1 f_2)$	comparisons	copies	depth	1 exp/word	2 exp/word	4 exp/word
div. heap	34592866	795737	2.67	.709 s	.550 s	.451 s
quo. heap	18874977	795829	3.13	.401 s	.291 s	.219 s
merge	34740676	37789852	2.51	.592 s	.428 s	.312 s

The results above suggest that heap algorithms would make a good foundation for sparse polynomial arithmetic. The times also clearly show the advantage of packing exponents in a large computation. The merging algorithm benefits the most from this (as a percentage of total time), because of the number of copies made during a merge.

Next we tried modifying the Maple 11 implementation of the Buchberger algorithm to use our routines to compute normal forms. We had to be careful not to include the overhead of Maple's external calling mechanism in our timings, because the time required to call our routine was usually longer than each normal form computation. We report the total time for all normal form computations, and the size of the merge buffer (in terms). The Gröbner basis computations used graded reverse lexicographic order.

		merge (exponents/word)				heap (exponents/word)		
system	n	1	2	4	buffer	1	2	4
cohn3	4	.600 s	.600 s	.500 s	2135	1.990 s	1.960 s	1.940 s
cyclic6	6	.100 s	.080 s	.050 s	912	.230 s	.160 s	.130 s
katsura7	8	.660 s	.610 s	.550 s	1770	1.300 s	1.240 s	1.200 s
virasoro	8	2.250 s	2.180 s	2.150 s	4416	4.860 s	5.100 s	5.050 s

This task showed less of a benefit for packing exponents. We believe that this is because our monomial representation requires us to unpack quotients to check if they are valid. Monomial divisions were a bottleneck, so it should be possible to improve these times by switching to a different monomial representation. Aside from this issue, which affects both algorithms equally, the merge algorithm greatly outperformed the heap algorithm on all of the examples. This is partly due to storage allocation, since our implementation of heap division does not yet recycle all of its storage, but it may also be the nature of Gröbner basis computations, it is just too early to say. We plan to do additional experiments in the near future.

5. Conclusion

We discussed the problems inherent in multivariate polynomial division and offered solutions in the form of good algorithms. The traditional merge algorithm for division was improved to efficiently recycle storage and utilize the cache on modern processors more effectively. An old and possibly forgotten algorithm based a heap was implemented and shown to be of great value. This algorithm is asymtotically fast and its temporary storage requirements are linear in the size of the input and the output. The heap algorithm was also improved in two critical ways. We described how to modify it to run in $O(NM)$ time on dense problems, and we presented a variation that runs in linear time in the size of the quotient, using a heap that is bounded by the size of the divisor.

References

[1] Elizabeth Arnold. Modular algorithms for computing Gröbner bases. *J. Symb. Cmpt.* **35** pp. 403–419, 2003.

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

[3] David Cox, John Little, Donal O'Shea. Ideals, Varieties and Algorithms: *An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer Verlag, 1992.

[4] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*, Kluwer Academic, 1992.

[5] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.0. A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra, University of Kaiserslautern (2005). `http://www.singular.uni-kl.de`.

[6] M. van Hoeij and M. Monagan. Algorithms for Polynomial GCD Computation over Algebraic Function Fields. *Proceedings of ISSAC '2004*, ACM Press, pp. 297–304, 2004.

[7] Richard Jenks, Robert Sutor and Scott Morrison. *AXIOM: The Scientific Computation System* Springer-Verlag, 1992.

[8] Stephen C. Johnson. Sparse polynomial arithmetic. ACM SIGSAM Bulletin, Volume 8, Issue 3, 63-71, 1974.

[9] Anthony LaMarca, Richard Ladner. The Influence of Caches on the Performance of Heaps. *J. Experimental Algorithms* **1**, Article 4, 1996.

[10] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Introductory Programming Guide* Maplesoft, 2005. ISBN 1-894511-76.

[11] David Stoutemyer. Which Polynomial Representation is Best? *Proceedings of the 1984 Macsyma Users Conference* Schenectedy, N.Y., pp. 221–244, 1984.

[12] Thomas Yan. The Geobucket Data Structure for Polynomials. *J. Symb. Comput.* **25**(3): 285-293, 1998.

[13] Richard Zippel. Probabilistic algorithms for sparse polynomials. *Proceedings of EUROSAM '79*, Springer-Verlag, pp. 216–226, 1979.