Sparse Polynomial Division Using a Heap

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

In 1974, Johnson showed how to multiply and divide sparse polynomials using a binary heap. This paper introduces a new algorithm that uses a heap to divide with the same complexity as multiplication. It is a fraction-free method that also reduces the number of integer operations for divisions of polynomials with integer coefficients over the rationals. Heap-based algorithms use very little memory and do not generate garbage. They can run in the cpu cache and achieve high performance. We compare our C implementation of sparse polynomial multiplication and division with integer coefficients to the routines of existing computer algebra systems.

1. Introduction

In this paper we consider how to multiply and divide sparse multivariate polynomials with integer coefficients. There are two polynomial representations that computer algebra systems mainly use: the *distributed* representation and the *recursive* representation, e.g.

$$f = 9\,xy^3z - 4\,y^3z^2 - 6\,xy^2z + 8\,x^3 + 5\,xy^2$$

$$f = 8\,x^3 + \left(9\,y^3z + (-6\,z + 5)\,y^2\right)x - 4\,y^3z^2$$

In the distributed representation a polynomial is represented as a sum of terms often sorted by a monomial ordering. Users enter polynomials in this format and they write programs for it. Algorithms for distributed polynomial arithmetic have a classical feel – their performance depends on how you sort terms and perform coefficient arithmetic.

⋆ 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).

Computer algebra systems that use the distributed representation by default include Maple, Mathematica, Magma, and Singular.

In the recursive representation polynomials are represented as univariate polynomials in a main variable with coefficients that are univariate polynomials in the next variable, and so on. Polynomial arithmetic uses univariate algorithms with coefficient arithmetic performed recursively. Computer algebra systems that use the recursive representation by default include Maxima, Derive, Reduce, Pari, and Trip.

In 1984, Stoutemyer in (13) compared different polynomial representations. He found that the recursive representation was generally better than the distributed representation, and surprisingly, that the recursive dense representation was clearly the best overall. This finding was confirmed by Fateman in 2003 (3), who tested implementations of sparse polynomial multiplication in Lisp and conducted benchmarks on various computer algebra systems.

This paper reconsiders the distributed representation. Let f and g be polynomials with $\#f = n$ terms $f_1, f_2, \ldots, f_n$ and $\#g = m$ terms $g_1, g_2, \ldots, g_m$ which are sorted by a monomial ordering. The classical algorithm for multiplication computes the product of f and g as a sequence of polynomial additions

$$h = \sum f_i g = ((((f_1 g + f_2 g) + f_3 g) + f_4 g) + \cdots + f_n g)$$

which are done by merging. Similarly, to divide h by g and compute the quotient f, the classical algorithm uses a sequence of subtractions

$$h - \sum f_i g = ((((h - f_1 g) - f_2 g) - f_3 g) - \cdots - f_n g).$$

For dense polynomials in one variable these algorithms perform $\Theta(\#f\,\#g)$ monomial comparisons but for sparse polynomials they can do $\Theta(\#f^2\#g)$ monomial comparisons. An example where this can occur is $f = x + x^2 + \cdots + x^n$ and $g = y + y^2 + \cdots + y^m$. Maple and Singular address this problem by using a divide-and-conquer algorithm for multiplication and by switching to the recursive representation for division.

Our earliest reference for sparse polynomial arithmetic is the paper of Johnson from 1974 (10). He uses a binary heap to multiply f and g using $O(\#f\#g \log(\min(\#f, \#g)))$ monomial comparisons, and to divide f by g with quotient q in $O(\#f + \#q\#g \log(\#q))$ comparisons. These algorithms were implemented in ALTRAN, but as far as we can tell no other system has used them. They were not tested by Stoutemyer in 1984 (13) or by Fateman in 2003 (3).

Notice that Johnson's division algorithm is more expensive than multiplication when $\#q \gg \#g$, that is, when the quotient is much larger than the divisor. This often occurs in the trial divisions of polynomial gcd computations. Our first contribution is a new division algorithm that uses a heap and performs only $O(\#f + \#q\#g \log(\min(\#q, \#g)))$ comparisons, which is the cost to multiply the quotient and the divisor and merge their product with the dividend.

Our second contribution is to modify the division algorithm to make it fraction-free. Using a heap delays all coefficient arithmetic, so we scale each term once to avoid doing arithmetic operations on fractions. Our algorithm adds factors to the denominator only as necessary and for divisions that succeed over the integers it does no additional work. The coefficient arithmetic can still blow up, but this is not something we can avoid in a direct method. Our strategy handles sparse problems with small fractions well.

Our paper is organized as follows. In Section 2 we discuss the work implicit in sparse distributed polynomial division, namely, the sorting of terms and the number of integer operations needed to divide over the rationals. We give a measure of sparsity and show how fraction-free merging can do an order of magnitude too many integer operations if the computation is sparse. In Section 3 we present Johnson's algorithms followed by our new fraction-free division algorithm. In Section 4 we compare our implementation of Johnson's multiplication algorithm and our division algorithm to the routines used by Maple, Magma, Singular, Pari, and Trip. The benchmarks show that systems that use a recursive representation (Pari & Trip) are generally faster than those with a distributed representation, namely Maple, Magma, and Singular. However the heap algorithms are the fastest of all, even for dense problems! In Section 5 we describe our implementation in detail and measure the effects of various optimizations.

We conclude here with a remark on why the heap algorithms perform so well. The main reason is cache locality. Heap algorithms use very little "working storage," that is, the amount of memory that is randomly accessed is small. For problems that multiply n by m terms the product or dividend could be $\Theta(nm)$ space but the working storage for the heap is only $O(\min(n, m))$ space. Also, because we can often pack monomials into 64-bit words, we can avoid the main loss of efficiency in the distributed representation.

2. Sparse Polynomial Division

Consider the problem of dividing $f \in \mathbb{Z}[x_1, \ldots, x_n]$ by $g \in \mathbb{Z}[x_1, \ldots, x_n]$, producing a quotient q and remainder r in $\mathbb{Q}[x_1, \ldots, x_n]$. This is a multivariate division in the sense of Gröbner bases. We assume the polynomials are stored in a sparse distributed format that is sorted with respect to a monomial order $\prec$.

Starting with $q = r = 0$, each step of the division examines the next non-zero term of $f - qg - r$ in descending order by $\prec$. Call this current term t. When t is divisible by g_1 we add a new term t/g_1 to the quotient q, otherwise we move t to the remainder r.

How we compute t has no effect on the result of the algorithm, but it does determine the performance. There are two tasks to perform: we must sort the terms of $f - qg - r$, and we must add up the coefficients of any equal monomials.

2.1. Monomial Comparisons

First consider how to sort the terms of $f - qg - r$ that are generated in the division. There are $\#f + \#q(\#g - 1)$ terms to sort in total, since $-qg_1$ is constructed to cancel the current term and the terms of r come from $f - qg$. We assume a comparison-based sort however a trie-based digital sort could be used instead (4).

The naive approach is to merge the partial products into an ordered linear structure such as a linked list or a dynamic array. The classical algorithm creates an intermediate polynomial $p := f$, and when a new term q_i of the quotient is computed it subtracts $q_i g$ from p using a merge. Unfortunately each merge is $\Theta(\#p + \#g)$, which means an exact division with $\#q = \#g = 1000$ and $\#f = 10^6$ could do $\Theta(10^9)$ comparisons.

For multiplication we can use the divide-and-conquer strategy to limit the number of monomial comparisons. For division we mention the beautiful geobucket data structure of Yan (14), which has a similar complexity but uses less memory (11). They are used in Singular for multiplication and division with remainder so we briefly describe them here.

A polynomial f is represented as a sum of buckets $f = b_1 + b_2 + \cdots + b_k$ with $\#b_i \le 2^i$, that is, bucket b_i has at most 2^i terms. To subtract a polynomial g from f, we subtract g from the bucket b_j with $2^{j-1} < \#g \le 2^j$ using a merge. If the result has more than 2^j terms then it is merged into the next bucket, and so on, until the bucket size constraint is satisfied. The idea is to always merge polynomials of approximately equal size.

Yan showed that geobuckets perform $O(N \log N)$ comparisons for sparse polynomials and $O(N)$ comparisons for dense polynomials, where $N = \#f + \#q(\#g - 1)$ is the total number of terms merged. In 2007 (11) we found that geobuckets used fewer comparisons than heap-based algorithms but ran slower on sparse problems due to cache effects.

Surprisingly, the minimum number of monomial comparisons needed to multiply two sparse polynomials is not known. The literature calls this problem "X+Y sorting" (7), and the fastest methods are $\Theta(nm \log(\min(n, m)))$ where $n = |X|$ and $m = |Y|$. This is the complexity of Johnson's multiplication algorithm, divide-and-conquer merging, and our division algorithm when it multiplies the quotient and the divisor. We also mention Horowitz's algorithm (9), which further exploits structure to reduce the actual number of monomial comparisons performed.

2.2. Measuring Sparsity

When we say a polynomial is sparse or dense we usually refer to its representation in memory. But for the benchmarks it will be important to test both "sparse" and "dense" problems. What does this mean?

The question is closely connected to the cost of sorting monomials. For example, a classical merge uses $\Theta(n^2)$ comparisons to multiply two dense univariate polynomials of degree n. It sorts and merges the $\Theta(n^2)$ products in linear time. This occurs because many of the monomials are equal, so as terms are added together the number of terms is reduced and the sorting task becomes easier. We propose to measure the density of a multiplication by dividing the number of products by the number of distinct monomials.

Definition 1. Let f and g be polynomials. We define the *work per term* of $f \times g$ to be $W(f, g) = (\#f\#g)/|S|$ where $S = \{ab : a \in \mathit{support}(f), b \in \mathit{support}(g)\}$.

The work measures the number of multiplications done to produce each term of the result. For example, let $f = (x + y + z + t + 1)^{20}$ and $g = f + 1$. Then $\#f = \#g = 10626$ terms and $|S| = 135751$ monomials, so the work per term is $W(f, g) = 10626^2/135751 = 831.76$. This large value of W indicates that the problem is very dense.

Now consider $F = (x^2 + y^2 + z^2 + t^2 + 1)^{20}$ and $G = F + 1$. These polynomials are more sparse, their product is more sparse, but the problem of multiplying F and G is equivalent. We multiply and add the same coefficients and sort the same monomials but with each exponent doubled. The work per term is also the same: $W(F, G) = W(f, g)$.

Let f and g be polynomials in n variables of total degree d. Since f and g can have at most $\binom{n+d}{n}$ terms, $1 \le W(f, g) \le B(n, d) = \binom{n+d}{n}^2 / \binom{n+2\,d}{n}$. For large d we have

$$B(n, d) \approx \frac{d^{\,n}}{n!\,2^{\,n}} + o(d^{\,n}).$$

Finally, let $f = a_1X_1 + \cdots + a_mX_m$ and $g = b_1Y_1 + \cdots + b_nY_n$ with $X_1 > \cdots > X_m$ and $Y_1 > \cdots > Y_n$. If we put the monomial products into a matrix $A_{ij} = X_iY_j$ its rows and columns strictly decrease so the number of distinct monomials is at least $n + m - 1$. Then $1 \le W(f, g) \le (nm)/(n + m - 1)$.

2.3. *Coefficient Arithmetic*

Next we count the coefficient operations that are performed in a sparse polynomial division. Multiplying polynomials with n and m terms does nm multiplications, but the number of additions depends on the number of distinct monomials generated, which is at least $n + m - 1$. Then up to $nm - n - m + 1$ additions are performed.

Now consider a division of f by g with quotient q and remainder r. Coefficients of q each require one division to construct, and we do at most $\#f$ additions to merge f with $-q(g - LT(g))$. In total there are $\#q(\#g - 1)$ multiplications, $\#q$ divisions, and up to $\#f + (\#q - 1)(\#g - 2)$ additions, which is comparable to a polynomial multiplication.

In counting the number of operations, we hope to show how different formulations of the division algorithm over a fraction field have different costs. Our main interest is the division of polynomials with small integer coefficients over $\mathbb{Q}$, but our analysis holds for sparse polynomial divisions over any fraction field.

Suppose that we are dividing polynomials with integer coefficients over the rationals. For an exact division we can apply Gauss' lemma to remove content and divide over the integers, but that does not work for divisions with a remainder. So let each coefficient of q be a fraction and suppose rational arithmetic is used. The $\#q(\#g - 1)$ multiplications are now a fraction times an integer, which we would implement as follows:

```
# compute (a/b) · c
mulqz(integer a, integer b, integer c)
  g := gcd(b, c);
  B := b/g;
  C := c/g;
  return (a · C, B);        # 1 gcd, 2 div, 1 mul
end;
```

To construct a term of the quotient we divide a fraction by an integer, which we can do by calling mulqz with the numerator and denominator swapped. To add fractions we simply use the grade-school method: $a/b + c/d = (ad + bc)/(bd)$ followed by a gcd and two exact divisions. Henrici's algorithm (8) could also be used, however it is not clear if there would be a gain. For word-sized integers, gcd is a program whereas multiplication and division are machine instructions. For multiprecision integers, if asymptotically fast algorithms are used this also favors doing fewer operations. Whichever method is faster, for counting arithmetic operations the grade-school method is preferred. We computed the number of integer operations for a division that uses rational arithmetic below.

Table 1. Integer operations for a sparse polynomial division with rational arithmetic.

multiplications	$3f + 3(q - 1)(g - 1) + qg$
divisions	$2f + 2(q - 1)(g - 1) + 2qg$
gcds	$f + (q - 1)(g - 1) + qg$
additions	$f + (q - 1)(g - 1)$

Counting only multiplications, divisions, and gcds as the most expensive operations, about ten times more work is required to divide polynomials using rational arithmetic ($10qg$ versus qg). When $\#f \approx \#q\#g$ it is sixteen times more. The extra cost is mostly due to adding fractions. The cost of multiplying the coefficients of q and g using mulqz is only qg multiplications, $2qg$ divisions, and qg gcds, or about four times more work.

Fraction-free algorithms were developed to avoid the high cost of rational arithmetic. In a division, fractions can be scaled to a common denominator that is updated as each new term of the quotient is computed. The cost is one extra division and multiplication per product $q_i g_j$, and one multiplication per term of f.

For example, suppose we are computing the next term of $sf - sq(g - LT(g))$, where s is a common denominator. Terms of f are multiplied by s and added into a sum, and for products $q_i g_j$ we compute and subtract $(s/denominator(q_i)) \cdot numerator(q_i) \cdot g_j$.

When all like terms have been added up and the coefficient c/s of the current term is known, if $c \neq 0$ we test if the current monomial is divisible by $LM(g)$. When it is, we divide c/s by $LC(g)$ by calling $(s, q) := mulqz(s, c, LC(g))$. This updates the common denominator s and the coefficient of the new quotient term is q/s. In practice we test if $LC(g) \,|\, c$ first to avoid gcds in polynomial divisions that succeed over the integers.

The cost of this scheme is shown in Table 2. It does three times more multiplications and divisions than a division over the integers, and four times more when $\#f = \#q\#g$. However that is the worst case, where the denominator is updated with every new term of the quotient. In practice this is rare, so we check (using a word comparison) whether the denominator of q_i is equal to s before scaling $q_i g_j$. For divisions that succeed over $\mathbb{Z}$ we use precisely the same operations as the division algorithm that runs over $\mathbb{Z}$.

Table 2. Integer operations for fraction-free sparse polynomial division (worst case).

multiplications	$f + 2q(g-1) + q$
divisions	$q(g-1) + 3q$
gcds	q
additions	$f + (q-1)(g-2)$

A critical feature of the approach is that all coefficient arithmetic is delayed in order to scale each term only once. That is, we do not multiply a term by anything until it is actually needed to compute the next term of $sf - sq(g - LT(g))$. In our algorithm this is accomplished by using a heap. Without delayed arithmetic the fraction-free approach can do an order of magnitude more work. To see why, consider the following algorithm which divides f by g using merging.

```
ffdivide(polynomial f, polynomial g)
  p, q, r, S := f, 0, 0, 1;
  while (p ≠ 0) do
    if (LM(g) | LM(p)) then
      G := gcd(LC(p), LC(g));
      A := LC(p)/G;
      B := LC(g)/G;
      S := S · B;              # update denominator
      t := LM(p)/LM(g);        # monomial quotient
      p := B · p − (A · t) · g; # cancel LT(p)
      q := q + (A/S) · t;
    else
      r := r + LT(p);
      p := p − LT(p);
  end loop;
  return (q, r/S);
end;
```

Consider an exact division $f \div g = q$, where $\#f = \#q\#g$ and the denominators of q_i increase in size. In iteration i of this division, p has $(\#q - i + 1)\#g$ terms and we need $(\#q - i + 2)\#g$ integer multiplications to compute $B \cdot p - (A \cdot t) \cdot g$. The total number of integer multiplications performed in all merge steps is

$$\sum_{i=1}^{\#q}(\#q - i + 2)\#g = \frac{(\#q + 3)\#q\#g}{2} \in \Theta(\#q^2\#g).$$

For sparse divisions, fraction-free merging can be an order of magnitude slower than merging with rational arithmetic. It can be made arbitrarily worse by adding terms to f that are only moved to the remainder at the end of the computation. In general we can perform $\Theta(qf + q^2g)$ integer multiplications. That is a disaster!

Note that for dense univariate polynomials fraction-free merging is not quite as bad. For an exact division with $\#f = \#q + \#g - 1$, each iteration cancels one term of p and the total number of integer multiplications is

$$\sum_{i=1}^{\#q}(\#q + 2\#g - i) = \frac{\#q(\#q + 4\#g - 1)}{2} \in \Theta(\#q^2 + \#q\#g).$$

3. Multiplication and Division Using a Heap

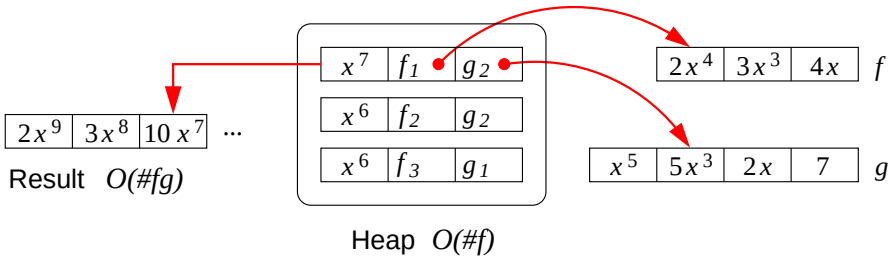
In 1974, Johnson (10) presented the following algorithm to multiply sparse distributed polynomials. Let $f = a_1X_1 + a_2X_2 + \cdots + a_nX_n$ and let $g = b_1Y_1 + b_2Y_2 + \cdots + b_mY_m$ with $X_1 > \cdots > X_n$ and $Y_1 > \cdots > Y_m$. The idea is to compute $\sum_{i=1}^{n} f_ig$ by merging all the partial products simultaneously using a binary heap.

We store the first unmerged term from each f_ig in the heap. In each iteration of the algorithm we extract all the maximal terms and add them up to produce the next term of the result. Then we insert the next term from each partial product that was merged, setting up the next iteration of the algorithm.

We use a structure (m, f_i, g_j) to represent the next term of each f_ig, where f_i and g_j are pointers into f and g and $m = X_iY_j$ is the monomial of f_ig_j. The pointer f_i is fixed while the pointer g_j increments through the terms of g.

When (m, f_i, g_j) is extracted from the heap we multiply the coefficients of f_i and g_j and add them to a sum of like terms. When all of the maximal terms have been added, for each term with $j < \#g$ we update its structure to (X_iY_j, f_i, g_{j+1}) and insert it back into the heap. The algorithm is depicted in Figure 1.

Fig. 1. Multiplication Using a Heap of Pointers (Johnson, 1974).



In the example above, we extract $(x^7, f_1, g_2) = 10x^7$ from the heap and it add to the end of the result. We then insert (x^5, f_1, g_3). In the next iteration we extract (x^6, f_2, g_2) and (x^6, f_3, g_1) and compute $15x^6 + 4x^6 = 19x^6$. This is added to the end of the result and we insert (x^4, f_2, g_3) and (x^4, f_3, g_2) for the next iteration.

The advantages of this algorithm are numerous. First, the heap has $\#f$ elements so $O(\log \#f)$ comparisons are used to insert and extract each term, and $O(\#f\#g \log \#f)$ comparisons are performed in total. We can choose $\#f \leq \#g$ to obtain the best known complexity for sparse problems. For dense problems, classical algorithms use $\Theta(\#f\#g)$ comparisons. In Section 5.2 we show how to obtain that complexity for the heap as well.

Second, the algorithm uses only $\Theta(\#f)$ "work space" to sort terms. This is an order magnitude less memory than merging, geobuckets, divide-and-conquer, or a hash table, all of which use memory that is proportional to the size of the result. As a consequence the algorithm can run in the cache and stream the result to main memory.

Third, if multiprecision integers are present, the algorithm needs storage for only one multiprecision integer to add up all the products for the current term. It can then copy this integer to the result and avoid generating garbage from multiprecision integers.

The algorithm also generates terms of the product one at a time in descending order. If we use a heap in the division algorithm to multiply the quotient and the divisor then we can subtract the result from the dividend term by term as the quotient is generated, which avoids storing any intermediate terms. The resulting algorithm uses an order of magnitude less memory to divide.

Whereas Johnson's multiplication algorithm computes $f \cdot g = \sum_{i=1}^{\#f} f_i g$, his division algorithm computes $f - \sum_{i=1}^{\#q} q_i g$. That is, the heap elements multiply by a term of the quotient and increment along the divisor. We call this a "quotient heap" division since the size of the heap is $\#q$. It performs $O(\#f + \#q\#g \log \#q)$ monomial comparisons.

Johnson's algorithm is inefficient when the quotient is large and the divisor is small. The quotient heap imposes a factor of $\log(\#q)$ on the number of comparisons, but a far worse problem is that the heap can outgrow the cache. The $O(\#q\#g \log \#q)$ monomial comparisons that take place in the heap are random memory accesses, all of which may produce cache misses which would slow down the algorithm significantly.

The divisor heap algorithm of Monagan and Pearce (11) solves this problem by using a heap to compute $f - \sum_{i=1}^{\#g} g_i q$. That is, the size of the heap is $\#g$ and heap elements multiply by a term of the divisor and increment along the quotient. An important issue is that we may need to extract $g_i q_j$ from the heap and merge it before computing q_{j+1}. Then we can not immediately insert $g_i q_{j+1}$. Our solution is to set a bit for each g_i that has a term in the heap. After extracting $g_i q_j$ from the heap we insert $g_i q_{j+1}$ if possible, and if g_{i+1} has no term in the heap we insert the next product for g_{i+1} if possible.

We illustrate the quotient and divisor heap algorithms below. In the figure terms are merged in descending order and at most one term from each row is present in the heap at any time. A grey box denotes where the sum of like terms is cancelled by computing a new term of the quotient. We divide $f = 2x^9 + 3x^8 + 10x^7$ by $g = x^5 + 5x^3 + 7$ with quotient $q = 2x^4 + 3x^3 - 15x$ and remainder $r = 61x^4 - 21x^3 + 105x$.

	step	1	2	3	4	5	6	7
	f	$2x^9$	$3x^8$	$10x^7$				
q. heap	$-q_1 g$	$-2x^9$		$-10x^7$		$-14x^4$		
q. heap	$-q_2 g$		$-3x^8$		$-15x^6$		$-21x^3$	
q. heap	$-q_3 g$				$15x^6$	$75x^4$		$105x$
	r					$61x^4$	$-21x^3$	$105x$

	step	1	2	3	4	5	6	7
	f	$2x^9$	$3x^8$	$10x^7$				
d. heap	$-g_1q$	$-2x^9$	$-3x^8$		$15x^6$			
	$-g_2q$			$-10x^7$	$-15x^6$	$75x^4$		
	$-g_3q$					$-14x^4$	$-21x^3$	$105x$
	r					$61x^4$	$-21x^3$	$105x$

3.1. The Minimal Heap Division Algorithm

We now have two algorithms to divide $f \div g = (q, r)$ using a heap with $\#q$ elements or a heap with $\#g - 1$ elements. However we do not know which algorithm to run since the size of the quotient q is a priori unknown. Our new division algorithm starts with a quotient heap and switches to a divisor heap when the number of terms in the quotient equals the number of terms in the divisor. In short, we compute

$$f \quad - \underbrace{\sum_{i=1}^{\#g-1} q_i \cdot g}_{\text{quotient heap}} \quad - \underbrace{\sum_{i=1}^{\#g} g_i \cdot (q_{\#g} + \cdots)}_{\text{divisor heap}}.$$

Our heap contains elements that behave like either a quotient heap or a divisor heap. We say a product q_ig_j "moves along g" if the next term inserted into the heap is q_ig_{j+1}. Likewise, a product q_ig_j "moves along q" if the next term inserted is $q_{i+1}g_j$.

The algorithm starts by adding a product q_ig_2 that moves along g for each new term of the quotient that it computes. The heap is enlarged as necessary during this phase to accommodate the growing quotient. When $\#q = \#g$, the heap is enlarged to $2(\#g - 1)$ elements and products $g_jq_{\#g}$ that move along q are initialized for $2 \le j \le \#g$. Then $g_2q_{\#g}$ is inserted into the heap and the algorithm continues to run with a divisor heap.

The size of the heap is at most $2(\#g - 1)$, which is double the size of a divisor heap when $\#q \ge \#g$. There is one extra level in the heap and therefore one more comparison per heap operation. This does not affect its complexity. We present the algorithm on the next page. Its cost in comparisons and arithmetic operations is given by Theorem 2.

Theorem 2. *Algorithm 1 divides f by g with quotient q and remainder r and performs $O(\#f + \#q(\#g - 1)\log(\min(\#q, \#g - 1)))$ monomial comparisons. If the denominator $s = 1$ it does $\#q(\#g - 1)$ integer multiplications and $\#q$ integer divisions. Otherwise it does at most the number of integer operations given in Table 2.*

Proof. If $\#q < \#g$ the size of the heap is $\#q$, otherwise it is $2(\#g - 1)$. We always use $O(\log(\min(\#q, \#g - 1)))$ comparisons to insert and extract the $\#q(\#g - 1)$ products of $q(g - LT(g))$, and at most $\#f + \#q(\#g - 1)$ comparisons to merge f with those terms. In total we perform $O(\#f + \#q(\#g - 1)\log(\min(\#q, \#g - 1)))$ monomial comparisons.

When $s = 1$ all $\mathit{denominator}(q_i) = s$ and we don't scale terms of f. $LC(g) \mid c$ always succeeds so $\#q(\#g - 1)$ integer multiplications and $\#q$ integer divisions are performed.

When $s > 1$ the strategy is described in Section 2.3. The $\#q(\#g - 1)$ terms extracted from the heap are scaled by at most one division and two multiplications, and terms of f are scaled with one multiplication. There are $\#q$ divisions to test $LC(g) \mid c$ and mulqz performs one gcd, two divisions, and a multiplication for each term of q. □

Algorithm 1: Fraction-Free Minimal Heap Division.

Input: $f, g \in \mathbb{Z}[x_1, ..., x_n]$, $g \neq 0$, a monomial order $\prec$.
Output: $q, r \in \mathbb{Q}[x_1, ...x_n]$ with $f = qg + r$ and no term of r divisible by $LT(g)$.

$(q, r, s, k) := (0, 0, 1, 1)$.
$H :=$ empty heap ordered by $\prec$ with max element H_1.
while ($|H| > 0$ or $k \leq \#f$) do
  if ($k \leq \#f$ and ($|H| = 0$ or $LM(f_k) \geq LM(H_1)$))
    $m := LM(f_k)$.
    if ($s = 1$) $c := LC(f_k)$ else $c := s * LC(f_k)$.
    $k := k + 1$.
  else
    $m := LM(H_1)$.
    $c := 0$.
  while ($|H| > 0$ and $LM(H_1) = m$) do
    extract $H_1 = q_i g_j$ from the heap.
    if ($denominator(q_i) = s$)
      $c := c - LC(numerator(q_i)) * LC(g_j)$.
    else
      $c := c - (s/denominator(q_i)) * LC(numerator(q_i)) * LC(g_j)$.
    if ($q_i g_j$ moves along g and $j < \#g$) // quotient heap product
      insert $q_i g_{j+1}$ into H moving along g.
    else if ($q_i g_j$ moves along q and $i < \#q$) // divisor heap product
      set the index of g_j into q to be $i + 1$.
      insert $q_{i+1} g_j$ into H moving along q.
      if ($j < \#g$ and bit says $g_{j+1} \notin H$)
        $t :=$ the last index of g_{j+1} into q.
        if ($t \leq \#q$)
          set bit for $g_j \in H$.
          insert $q_t g_{j+1}$ into H moving along q.
    else if ($q_i g_j$ moves along q and $i = \#q$) // ran out of quotient
      set the index of g_j into q to be $i + 1$.
      set bit for $g_j \notin H$.
  end loop.
  if ($c \neq 0$ and $LM(g) \,|\, m$)
    if ($LC(g) \,|\, c$)
      $c := c/LC(g)$.
    else
      $(s, c) := mulqz(s, c, LC(g))$.
    $q := q + (c/s)(m/LM(g))$.
    if ($\#q < \#g$) // using quotient heap
      insert $q_{(\#q)} g_2$ into H moving along g.
    else if ($\#q > \#g$) // using divisor heap
      if (bit says $g_2 \notin H$)
        set bit for $g_2 \in H$
        insert $q_{(\#q)} g_2$ into H moving along q.
    else // switching to divisor heap
      set the index of each g_j into q to be $\#q$.
      set bits for $g_2 \in H$ and all other $g_j \notin H$.
      insert $q_{(\#q)} g_2$ into H moving along q.
  else if ($c \neq 0$)
    $r := r + (c/s) * m$.
end loop.
return (q, r)

4. Benchmarks

To benchmark the computer algebra systems we used one core of an Intel Xeon 5160 (Core2) 3.0 GHz with 4 MB of L2 cache, 16 GB of RAM, 64-bit Linux, and GMP 4.2.1. We give two times for our library (sdmp). In the slow time (unpacked) each exponent is stored as a 64 bit integer. For the fast time we pack all of the exponents into one 64 bit integer and use word operations to compare and multiply monomials.

We tested Pari/GP 2.3.3, Magma 2.14-7 (see (2)), Maple 12, Singular 3-0-4 (see (6)), and Trip 0.99 (see (4)). Recall that Maple and Singular use distributed representations to multiply polynomials and recursive representations to divide, Magma and sdmp use distributed representations, Pari uses recursive dense, and Trip uses recursive sparse.

4.1. Fateman's Benchmark

Our first problem is due to Fateman (3). Let $f = (1 + x + y + z + t)^{20}$ and $g = f + 1$. We multiply $p = f \cdot g$ and divide $q = p/f$. The coefficients of f and g are 39 bit integers and the coefficients of p are 83 bit integers.

Table 3. Dense multiplication and division over $\mathbb{Z}$, $W(f, g) = 831.76$.

$10626 \times 10626 = 135751$ terms	$p = f \cdot g$		$q = p/f$	
sdmp (1 word monomial)	2.26 s	(4.9 MB)	2.77 s	(1.7 MB)
sdmp (4 word monomial)	5.18 s	(8.4 MB)	5.44 s	(2.6 MB)
Trip v0.99 (floating point)	2.50 s	(8.0 MB)	-	
Trip v0.99 (rationals)	5.93 s	(15.3 MB)	-	
Pari/GP 2.3.3	32.43 s		14.76 s	
Magma V2.14-7	23.02 s	(32.2 MB)	22.76 s	(12.7 MB)
Singular 3-0-4	62.00 s	(21.0 MB)	20.00 s	(15.5 MB)
Maple 12	289.23 s	(200.4 MB)	187.72 s	(48.2 MB)

Fateman's benchmark is a dense computation. The $\binom{n+d}{d}$ monomials in n variables of degree $\leq d$ are multiplied to produce $\binom{n+2d}{2d}$ terms. Maple and Singular can both divide faster than they can multiply because they use recursive algorithms. In sdmp division is slightly slower than multiplication because the code is more complex.

4.2. Sparse 10 Variables

Now let $f = \left(\sum_{i=1}^{9} x_i(x_{i+1} + 1) + x_{10}(x_1 + 1) + 1\right)^4$ and $g = \left(\sum_{i=1}^{10}(x_i^2 + x_i) + 1\right)^4$. We multiply $p = f \cdot g$ and divide $q = p/f$. All coefficients are less than 20 bits long.

Table 4. Sparse multiplication and division over $\mathbb{Z}$, $W(f, g) = 17.86$.

$6746 \times 8361 = 3157883$ terms	$p = f \cdot g$		$q = p/f$	
sdmp (1 word monomial)	2.46 s	(54.8 MB)	2.61 s	(1.0 MB)
sdmp (10 word monomial)	11.12 s	(300.8 MB)	10.37 s	(3.6 MB)
Trip v0.99 (floating point)	4.10 s	(206.3 MB)	-	
Trip v0.99 (rationals)	8.13 s	(351.9 MB)	-	
Pari/GP 2.3.3	7.06 s		7.05 s	
Magma V2.14-7	17.43 s	(413.2 MB)	197.72 s	(94.5 MB)
Singular 3-0-4	31.00 s	(200.1 MB)	18.00 s	(236.0 MB)
Maple 12	305.76 s	(973.9 MB)	280.65 s	(485.0 MB)

This benchmark clearly demonstrates the value of packing exponents. Pari's recursive dense algorithms perform well on many variables of low degree, while Magma's division (ExactQuotient) seems to sort terms inefficiently. It may be using repeated subtraction.

4.3. Very Sparse 5 Variables

We multiply $f = (1 + x + y^2 + z^3 + t^5 + u^7)^{12}$ and $g = (1 + u + t^2 + z^3 + y^5 + x^7)^{12}$ and divide the product by f. The coefficients of f, g, and p are 23, 23, and 47 bits long.

Table 5. Very sparse multiplication and division over $\mathbb{Z}$, $W(f, g) = 2.9$.

$6188 \times 6188 = 13209653$ terms	$p = f \cdot g$		$q = p/f$	
sdmp (1 word monomial)	2.12 s	(202.2 MB)	2.60 s	(1.0 MB)
sdmp (5 word monomial)	5.74 s	(606.6 MB)	7.41 s	(2.0 MB)
Trip v0.99 (floating point)	2.54 s	(648.4 MB)	-	
Trip v0.99 (rationals)	5.63 s	(1255.1 MB)	-	
Pari/GP 2.3.3	284.19 s		134.87 s	
Magma V2.14-7	36.30 s	(4155.5 MB)	276.45 s	(405.2 MB)
Singular 3-0-4	44.00 s	(1197.2 MB)	84.00 s	(1013.3 MB)
Maple 12	250.24 s	(2350.2 MB)	277.65 s	(1522.0 MB)

On sparse problems the advantages of heaps are substantial. The product $p = f \cdot g$ may be hundreds of megabytes but that data is written to memory sequentially. The heap is less than half a megabyte. For division the heap is larger but the total memory required is still only one or two megabytes which easily fits in the cache.

4.4. Sparse Unbalanced Divisions

We multiply $f = (1 + x + y^2 + z^3 + t^5 + u^7)^n$ and $g = (1 + u + t^2 + z^3 + y^5 + x^7)^m$ and divide the result by f, using n and m to vary the sizes of the quotient and divisor. The work $W(f, g) < 3$ implies that these problems are very sparse.

The column H_{real} is the actual size of the heap during the division, which is less than $H_{max} = \min(\#q, 2(\#f - 1))$ due to the chaining optimization discussed in Section 5.2. Chaining is also used for multiplication but the size of the heap remained $\min(\#f, \#g)$. We conclude that our division algorithm sorts terms as efficiently as a multiplication.

Table 6. Varying the quotient and the divisor.

n	m	$\#f$	$\#g$, $\#q$	$\#p$	$W(f, g)$	$p = f \cdot g$	$q = p/f$	H_{max}	H_{real}
30	4	324632	126	17691345	2.31	2.48 s	2.60 s	126	126
18	8	33649	1287	15143968	2.86	2.86 s	2.93 s	1287	1035
12	12	6188	6188	13209665	2.90	2.12 s	2.60 s	12374	2366
8	18	1287	33649	15143968	2.86	2.30 s	2.27 s	2572	1035
4	30	126	324632	17691345	2.31	2.10 s	2.10 s	250	70

n	m	Pari/GP 2.3.3		Magma V2.14-7		Singular 3-0-4		Maple 12	
30	4	201.34 s	65.94 s	41.27 s	27.46 s	28.00 s	124.00 s	329.89 s	357.18 s
18	8	511.26 s	157.00 s	45.26 s	108.86 s	46.00 s	99.00 s	302.30 s	349.15 s
12	12	284.19 s	134.87 s	36.30 s	276.45 s	44.00 s	84.00 s	250.24 s	277.65 s
8	18	97.17 s	94.08 s	42.06 s	1462.25 s	44.00 s	107.00 s	293.23 s	387.99 s
4	30	19.25 s	17.48 s	39.49 s	15690.64 s	26.00 s	2990.00 s	308.30 s	479.35 s

4.5. Division with Remainder

Finally, to test the effectiveness of our fraction-free strategy, we divide $f = (xyztu)^{36}$ by $g = \big((x^9 - y - 1)(2y^9 - z - 2)(3z^9 - t - 3)(4t^9 - u - 4)(5u^9 - x - 5)\big)^2$, computing a quotient q and remainder r using graded lexicographical order with $x > y > z > t > u$.

The divisor g has 19 bit coefficients and its leading coefficient is 14400. The quotient q and remainder r have 22 and 39 bit numerators over a 26 bit common denominator. For Magma we timed the NormalForm command and in Singular we timed reduce(f, g). Pari and Trip lack this functionality and Maple's interpreted algorithm did not finish in reasonable time. We also performed the division over $\mathbb{Z}_{32003}$ for comparison.

Table 7. Sparse division with remainder, $W(q, g) = 79.6$.

$1 \div 7776 = (7776, 99999)$ terms	$f \div g = (q, r)$ over $\mathbb{Q}$		$f \div g = (q, r)$ over $\mathbb{Z}_p$	
sdmp (1 word monomial)	3.02 s	(4.8 MB)	2.69 s	(2.5 MB)
sdmp (6 word monomial)	8.35 s	(14.1 MB)	7.84 s	(13.3 MB)
Magma V2.14-7	120.90 s	(62.7 MB)	112.60 s	(38.5 MB)
Singular 3-0-4	447.00 s	(120.1 MB)	249.00 s	(29.5 MB)

The fraction-free strategy of Section 2.3 holds up very well. Provided the coefficients remain small, it imposes only a small cost in overhead versus a computation modulo p. Magma also has a good strategy but we do not know what it is. Singular uses too much coefficient arithmetic to do the division over the rationals.

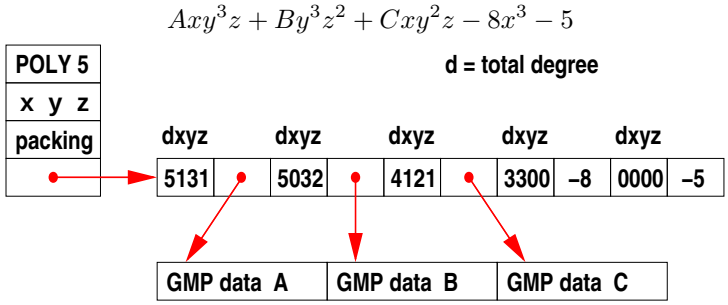
5. Optimizations

5.1. Polynomial Data Structure

In selecting a data structure, we assumed that polynomials could become very large. Because the heap algorithms access terms sequentially, we sought to minimize the cost in machine cycles of accessing the next term.

The natural approach is to store the terms in an array, but multiprecision coefficients posed a problem. We wanted to be able to seek to arbitrary terms of the polynomial in constant time. Our solution was to adopt Maple's integer format. An integer coefficient x is stored in place as $2x + 1$ when $|x| < 2^{B-2}$ where $B = 64$ is the base of the machine. For multiprecision coefficients we store pointers, which we distinguish by the lowest bit. We store all multiprecision coefficients in a second array as shown in Figure 2.

Fig. 2. Packed representation for graded lex order with $x > y > z$.



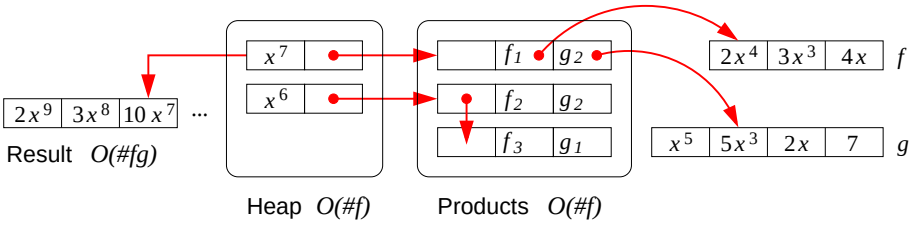
The benchmarks show the significant benefit of packing exponents. We pack multiple exponents into a word and store extra information so that monomials can be compared using word operations (see (1; 11)). For example in graded lexicographical order $x^2y^3z^4$ is stored as $[9, 2, 3, 4]$, where 9 is the total degree. This can be packed into the machine integer $9 \cdot 2^{3B/4} + 2 \cdot 2^{B/2} + 3 \cdot 2^{B/4} + 4$ and compared with one instruction.

To multiply or divide monomials we add or subtract unsigned machine integers. The detection of exponent underflow in division is done using a bit mask, which requires the top bit of each exponent to be zero. For example, the trial division of $x^2y^3z^4$ by xy^4z^4 in 32 bits computes $[9, 2, 3, 4] - [9, 1, 4, 4] =$ 00000000 00000000 11111111 00000000, and then a bitwise OR with 10000000 10000000 10000000 10000000 detects underflow in the third exponent. The top bit of every exponent in every monomial must be zero in order to run the division algorithm. A big advantage of the heap method is that this test can be performed once per distinct monomial at the beginning of the main loop, instead of once per product. The $\#q(\#g - 1)$ monomial multiplications do not test for overflow.

5.2. Heap Chaining

We developed a number of optimizations that dramatically improve the performance of the heap. The most important is chaining. When a product is inserted into the heap we first compare it to the top element and if they are not equal we compare bottom-up to find its position in the heap. If a comparison returns equality we insert the product at the head of a linked list to form a chain of like terms, as shown below for x^6.

Fig. 3. Chaining Products in the Heap.



Notice that by storing $\{f_i, g_j\}$ outside of the heap, we allow the heap comparisons to access as little memory as possible (two words). The pointers for $\{f_i, g_j\}$ and chains are needed less often, to add up products. This organization of data by frequency of access further improves cache performance.

Two more optimizations increase the effectiveness of chaining. In the main loop of the algorithm we extract all equal terms before inserting their successors to ensure that the largest new terms are chained together at the top of the heap. Second, at the beginning of a multiplication $f \cdot g$ we insert only f_1g_1 into the heap, and after extracting each f_ig_1 we insert both f_ig_2 and $f_{i+1}g_1$ if they exist. This approach is also used in Algorithm 1. In (11) we show that for dense univariate polynomials the size of the heap is always one so a multiplication $f \cdot g$ does $\Theta(\#f\#g)$ monomial comparisons and a division $f \div g = q$ does $\Theta(\#f + \#g\#g)$ monomial comparisons.

Table 8 measures the effectiveness of chaining. In an $n \times m$ multiplication, nm terms are inserted and removed from the heap, but we do only one heap extraction to remove all the elements of a chain. In Table 8 we report the total number of heap extractions, the number of products nm, and the percentage of heap extractions saved by chaining.

On Fateman's problem a typical term of the result came from one large chain with 2690 terms, 29 chains of length 1, and additional chains of length 9, 12, 15, 20, 27, 27, 29, 32, 33, 37, 38, and 40.

Table 8. The percentage of heap extractions saved by chaining.

	$W(f,g)$	extractions	nm	% saved
Fateman	831.76	3194958	112911876	97.2%
sparse10	17.86	17087173	56403306	69.7%
vsparse5	2.90	14720559	38291344	61.5%

5.3. Immediate Monomials and Assembly Code

Two more optimizations warrant brief discussion. We store the monomials directly in the heap if they are only one word long to improve the locality and performance of heap operations. We call these immediate monomials. Multi-word monomials are stored in a separate array so that heap operations do not have to move the monomial data.

We use two different algorithms to extract the maximal element from the heap. For one word monomials we insert the last element into the empty spot at the root and sift it down towards the bottom using two comparisons per level. This method best utilizes the cache. For multi-word monomials we promote the largest child into the empty spot using one comparison per level until the empty spot reaches the bottom, then we insert the last heap element into this spot and promote it if necessary. This method uses fewer monomial comparisons on average. For more details see (11).

We also implemented coefficient arithmetic for word-sized integers in assembly code. When the coefficients of f_i and g_j are both stored in place (see Section 5.1) we multiply them in assembly and add the double word result to a three word array. Other products involve multiprecision integers and are computed using GMP (5). Finally, when all the products for the current term have been added, we add the three word array to the sum for general multiprecision products to obtain the next coefficient. This integer is copied to the result and we reuse the storage in the next iteration of the algorithm.

5.4. Performance Assessment

Table 9 compares our times for packed and unpacked multiplication (first line) to the cost of sorting products without any coefficient arithmetic (second line). Thus, the time spent doing arithmetic on Fateman's benchmark is roughly $2.26 - 1.60 = 0.66$ seconds. Heap chaining (third line) is by far the most important optimization. This is followed by assembly code on dense problems or immediate monomials on sparse problems.

Table 9. The cost of sorting terms and the effect of disabling optimizations.

$p = f \cdot g$	Fateman		sparse10		vsparse5	
sdmp multiplication	2.26 s	5.18 s	2.46 s	11.12 s	2.12 s	5.74 s
no coefficient arithmetic	1.60 s	4.47 s	2.10 s	10.39 s	1.65 s	5.15 s
no heap chaining	9.76 s	31.43 s	4.56 s	22.98 s	3.92 s	11.00 s
no immediate monomials	3.09 s	-	3.97 s	-	2.96 s	-
alternate heap extract	2.28 s	5.23 s	2.68 s	11.30 s	2.36 s	5.52 s
no assembly (uses GMP)	4.12 s	7.09 s	3.34 s	11.87 s	2.49 s	6.01 s

6. Conclusion

We analyzed sparse polynomial division and presented a new algorithm which uses a chained heap of pointers to divide. It sorts terms as efficiently as a multiplication and it uses few extra arithmetic operations to divide polynomials over the rationals. Its space requirement is linear in the size of the input and the result so it can often run in cache. Our implementation of Johnson's algorithm for multiplication and our division algorithm achieves high performance. This shows that the distributed representation is competitive with recursive representations when good algorithms are used.

We have integrated our routines into the `expand` and `divide` commands for the next release of the Maple computer algebra system, but in doing so we noticed a problem. The conversions from our data structure to Maple's involve a lot of overhead. New objects are created for every monomial in the polynomial, and these are all simplified, hashed, and sorted by Maple's kernel. We plan to address this issue in the future by modifying Maple's kernel to use our data structure by default for polynomials with sufficiently few variables and low enough total degree.

Acknowledgment

We thank the referees for their suggestions which have improved this paper.

References

[1] Bachmann, O., Schönemann, H., 1998. Monomial representations for Gröbner bases computations. Proc. ISSAC '98, pp. 309–316.

[2] Bosma, W., Cannon, J., Playoust, C., 1997. The Magma algebra system I: The user language. J. Symb. Comput. **24** (3-4), 235–265.

[3] Fateman, R., 2003. Comparing the speed of programs for sparse polynomial multiplication. ACM SIGSAM Bulletin **37** (1), 4–15.

[4] Gastineau, M., Laskar, J., 2006. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. In: Proc. ICCS 2006, Springer LNCS **3992**, 446–453. http://www.imcce.fr/Equipes/ASD/trip

[5] Granlund, T., 2008. The GNU Multiple Precision Arithmetic Library, version 4.2.2. http://www.gmplib.org/

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

[7] Harper, L.H., Payne, T.H., Savage, J.E., Straus, E., 1975. Sorting X+Y. Comm. ACM **18** (6) 347–349.

[8] Henrici, P., 1956. A Subroutine for Computations with Rational Numbers. J. ACM **3** (1), 6–9.

[9] Horowitz, E., 1975. A Sorting Algorithm for Polynomial Multiplication. J. ACM **22** (4), 450–462.

[10] Johnson, S.C., 1974. Sparse polynomial arithmetic. ACM SIGSAM Bulletin **8** (3), 63–71.

[11] Monagan, M., Pearce, R., 2007. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. Proc. of CASC 2007, pp. 295–315.

[12] PARI/GP, version 2.3.4, Bordeaux, 2008. http://pari.math.u-bordeaux.fr/

[13] Stoutemyer, D., 1984. Which Polynomial Representation is Best? Proc. of the 1984 Macsyma Users Conference, Schenectedy, N.Y., pp. 221–244.

[14] Yan, T., 1998. The geobucket data structure for polynomials. J. Symb. Comput. **25** 285–293.