Adaptive Polynomial Multiplication*

Daniel S. Roche
Symbolic Computation Group
University of Waterloo
www.cs.uwaterloo.ca/~droche

29 February 2008

Abstract

Finding the product of two polynomials is an essential and basic problem in computer algebra. While most previous results have focused on the worst-case complexity, we instead employ the technique of adaptive analysis to give an improvement in many "easy" cases where other algorithms are doing too much work. Three ideas for adaptive polynomial multiplication are given. One method, which we call "chunky" multiplication, is given a more careful analysis, as well as an implementation in NTL. We show that significant improvements can be had over the fastest general-purpose algorithms in many cases.

1 Introduction

Polynomial multiplication has been one of the most well-studied topics in computer algebra and symbolic computation over the last half-century, and has proven to be one of the most crucial primitive operations in a computer algebra system. However, most results have focused on the worst-case analysis, and in doing so overlook many cases where polynomials can be multiplied much more quickly. We develop algorithms which are significantly faster than current methods in many instances, and which are still never (asymptotically) slower.

*Submitted to Milestones in Computer Algebra (MICA 2008), to be held May 1–3 in Stonehaven Bay, Trinidad and Tobago

For univariate polynomials, multiplication algorithms generally fall into one of two classes, depending on which representation is used. Let R be a ring, and $f \in \mathsf{R}[x]$. The *dense* representation of f is by a vector of all coefficients in R, in order. If we denote by n the degree of f, then the length of this vector will be exactly $n+1$. The *sparse* representation is a list of coefficient-exponent pairs in $\mathsf{R} \times \mathbb{N}$ sorted by the exponents, where only the nonzero coefficients are represented. If t is the number of nonzero terms in f, and we assume constant storage for elements in R, then an upper bound on the size of this representation is $O(t \log n)$. Unfortunately, this bound is not always tight, for example when most terms have low degree. A suitable lower bound is $\Omega(t \log t + \log n)$.

Advances in dense polynomial multiplication have usually followed advances in long integer multiplication, starting with the first sub-quadratic algorithm by Karatsuba and Ofman in 1962 [7], followed by the first superlinear algorithm by Schönhage and Strassen in 1971 [11], which is based on the celebrated Fast Fourier Transform (FFT) method [3].

Cantor and Kaltofen completed the important work of extending FFT-based multiplication to polynomials over arbitrary algebras in 1991 [2]. If we denote by $\mathsf{M}(n)$ the number of ring operations needed to multiply two polynomials with degrees less than n over $\mathsf{R}[x]$, they proved $\mathsf{M}(n) \in O(n \log n \operatorname{loglog} n)$. Progress towards eliminating the $\log\log n$ factor continues, with recent work (as usual, for multi-precision integer multiplication) by Martin Fürer in 2007 [4].

Here we will usually assume $\mathsf{M}(n) \in O(n \log n)$.

This is true for example if R contains a 2^k-th primitive root of unity for $2^k \geq n$. Although this is not generally the case, ignoring the loglog factor will greatly simplify our analysis. A lower bound of $\Omega(n \log n)$ has also been proven under a relatively reasonable model [1]. So we will actually assume $\mathsf{M}(n) \in \Theta(n \log n)$.

To multiply two sparse polynomials with t nonzero terms, the naïve algorithm requires $O(t^2)$ ring operations. In fact, this is optimal, since the product could have that many terms. But for sparse polynomials, we must also account for other word operations that arise from the exponent arithmetic. Using "geobuckets", this can be reduced to $O(t^2 \log t \log n)$ [14]; more recent results show how to reduce the space complexity to achieve an even more efficient algorithm [9].

Sparse representations become very useful when polynomials are in many variables, as the dense size grows exponentially in the number of indeterminates. In this case, others have noticed that the best overall approach may be to use a combination of sparse and dense methods in what is called the *recursive dense* representation [13]. Since most multivariate algorithms boil down to univariate algorithms, we restrict ourselves here to polynomials over $\mathsf{R}[x]$. Our algorithms will easily extend to multivariate polynomials, but the details of such adaptations are not presented here.

Section 2 outlines the general idea behind adaptive analysis, and how we will make use of this analysis for polynomial multiplication. Next we present one idea for adaptive multiplication, where the input polynomials are split up into dense "chunks". In Section 4, we cover an implementation of this idea in the C++ library NTL. Two other ideas for adaptive multiplication are put forth in Section 5. Finally, we discuss the practical usefulness of our algorithms and future directions for research.

2 Adaptive Analysis

By "adaptive", we mean algorithms whose complexity depends not only on the size of the input, but also on some other measure of difficulty. This terminology comes from the world of sorting algorithms, and its first use is usually credited to Mehlhorn [8]. Adaptive algorithms for sorting will have complexity dependent not only on the length of the list to be sorted, but also to what extent the list is already sorted. The results hold both theoretical interest and practical importance (for a good overview of adaptive sorting, see [10]).

In some sense, these algorithms identify "easy" cases, and solve them more quickly than the general, "difficult", cases. Really, we are giving a finer partition of the problem space, according to some measure of difficulty in addition to the usual size of the input. We require that our algorithms never behave worse than the usual ones, so that a normal worst-case analysis would give the same results. However, we also guarantee that easier cases be handled more quickly (with "easiness" being defined according to our chosen difficulty measure).

Adaptive analysis is not really new to computer algebra, but it usually goes by some other name. For instance, "early termination" strategies have proven very useful in some linear algebra and polynomial computations where bounds are not tight or not known a-priori (see e.g. [6]). These algorithms essentially recognize an intrinsic measure of difficulty and perform better than the worst case when the problem is easier to solve.

Some have observed at least a historical connection between polynomial multiplication and sorting [5], so it makes sense that our motivation comes from this area. Of course, multiplying polynomials is not the same as sorting, and we see right away that a difficulty measure as intrinsic as "the presortedness of the input" will probably not be possible.

From the discussion above, however, an obvious difficulty measure is the sparsity of the input polynomials. This leads to a trivial adaptive algorithm: (1) find the number of nonzero terms and determine whether sparse or dense algorithms will be best, and then (2) convert to that representation and perform the multiplication. In fact, such an approach has been suggested already to handle varying sparsity in the intermediate computations of triangular decompositions.

2.1 Our Approach

The algorithms we present will always proceed in three stages. First, the polynomials are read in and converted to a different representation which effectively captures the relevant measure of difficulty. Second, we multiply the two polynomials in the alternate representation. Finally, the product is converted back to the original representation.

The reader will immediately notice that this is the same as the general outline of FFT-based multiplication. However, our aim is somewhat opposite. For FFT-based multiplication, computing the product in the alternate representation is fast (linear time), and the dominating cost comes from the cost of steps (1) and (3) to convert to and from this representation. But for our purposes, only step (2) will have complexity dependent on the difficulty measure, and so we want steps (1) and (3) to be as fast as possible, which will usually mean linear time in the size of the input.

For the methods we put forth, the second step is relatively straightforward given the chosen representation. The final step will be even simpler, and it will usually be possible to combine it with step (2) for greater efficiency. The most challenging aspect is designing an algorithm for the first step which is linear time, as we are somehow trying to recognize structure from chaos. This is also possible, but we do sometimes sacrifice some efficiency of step (2) in order to guarantee linear time for the first step.

Our adaptive algorithms will rely on fast dense polynomial arithmetic, for which we require the following simple observation. If $f, g \in \mathsf{R}[x]$ with degrees less than n and m respectively, and $m \leq n$, then the product fg can be computed with $O(\frac{n}{m}\mathsf{M}(m))$ ring operations. (This is achieved by partitioning the coefficient list of f into blocks of length m.) Using our assumption that $\mathsf{M}(n) \in \Theta(n \log n)$, this becomes $O(n \log m)$. To be even more concrete, we will say the cost is bounded by $cn \log(m+1)$ for some constant $c > 0$. The $(m+1)$ adjustment is necessary to reflect the fact that multiplication when $m = 1$ (i.e. multiplication by a scalar) is not free.

3 Chunky Multiplication

The idea here is simple, and provides a natural gradient between the well-studied dense and sparse algorithms for univariate polynomial arithmetic. For $f \in \mathsf{R}[x]$ of degree n, we represent f as a sparse polynomial with dense "chunks" as coefficients:

$$f = f_1 x^{e_1} + f_2 x^{e_2} + \cdots + f_t x^{e_t}, \tag{1}$$

with each $f_i \in \mathsf{R}[x]$ and $e_i \in \mathbb{N}$. Let $d_1, d_2, \ldots, d_t \in \mathbb{N}$ be such that the degree of each f_i is less than d_i. Then we require $e_{i+1} > e_i + d_i$ for $i = 1, 2, \ldots, t-1$, so that there is some "gap" between each dense "chunk". We do *not* insist that each f_i be completely dense, but require only that the leading and constant coefficients be nonzero. In fact, deciding how much space to allow in each chunk is the challenge of converting to this representation, as we will see.

Multiplying polynomials in the chunky representation uses sparse multiplication on the outer loop, treating the f_i's as coefficients, and dense multiplication to find each product $f_i g_i$. If we use a heap to store pointers into the divisor as in [9], then chunks of the result will be computed in order, so we can combine chunks of the product that are close together, and then convert back to the original representation in linear time, as required.

3.1 Analysis

To analyze the complexity of the multiplication step, we consider a somewhat degenerative case, where a chunky polynomial f is multiplied by a completely dense polynomial g (i.e. g has only one chunk). Clearly multiplying by a chunky polynomial g instead will just simplify to multiplying f by each of the dense chunks, so we do not really lose any generality in this assumption.

Theorem 3.1. *Let $f, g \in \mathsf{R}[x]$ with f as in (1) and g dense of degree m. Then the product fg can be computed with*

$$O\left(m \log \prod_{d_i \leq m} (d_i + 1) + (\log m) \sum_{d_i > m} d_i\right)$$

ring operations.

The proof is from our assumption that $\mathsf{M}(n) \in O(n \log n)$, and therefore the cost of multiplying a chunk f_i of f by g is $O(m \log d_i)$ if $m \geq d_i$ and $O(d_i \log m)$ otherwise.

The following two lemmas indicate what we must minimize in order to be competitive with known techniques for dense and sparse multiplication.

Lemma 3.2. *If $\prod(d_i + 1) \in O(n)$, then the cost of chunky multiplication is never asymptotically greater than the cost of dense multiplication.*

Proof. First, notice that $\sum d_i \leq n$ (otherwise we would have overlap in the chunks). And assume $\prod(d_i + 1) \in O(n)$. From Theorem 3.1, the cost of chunky multiplication is thus $O(m \log n + n \log m)$. But this is exactly the cost of dense multiplication, from the assumption that $\mathsf{M}(n) \in \Omega(n \log n)$. □

Lemma 3.3. *Let s be the number of nonzero terms in f. If $\sum d_i \in O(s)$, then the cost of chunky multiplication is never asymptotically greater than the cost of sparse multiplication.*

Proof. Assume g is totally dense. Then sparse multiplication of f times g costs $O(sm)$ ring operations. Now clearly $t \leq s$, and note that

$$\log \prod(d_i + 1) = \sum \log(d_i + 1) \leq t + \sum d_i \in O(s).$$

Since $\log m \in O(m)$, this gives a total cost of $O(sm)$ ring operations from Theorem 3.1. The cost of exponent arithmetic will be $O(mt \log t \log n)$, which is less than the $O(ms \log s \log n)$ for the sparse algorithm as well. □

It is easy to generate examples showing that these bounds are tight. Unfortunately, this means that there are instances where a single chunky representation will not always result in better performance than the dense *and* sparse algorithms. One such example is when f has $\sqrt{n}$ nonzero terms spaced equally apart. Therefore we consider two separate cases for converting to the chunky representation, depending on the representation of the input. When the input is dense, we seek to minimize $\prod(d_i + 1)$, and when it is sparse, we seek to minimize $\sum d_i$ (to some extent).

3.2 Conversion from Sparse

Converting from the sparse representation is somewhat simpler, so we consider this case first. Lemma 3.3 indicates that minimizing the sum of the degrees of the chunks will guarantee competitive performance with the sparse algorithm. But the minimal value of $\sum d_i$ is actually achieved when we make every chunk completely dense, with no spaces within any dense chunk. While this approach will always be at least as fast as sparse multiplication, it will usually be more efficient to allow some spaces in the chunks if we are multiplying f by a dense polynomial g of any degree larger than 1.

One way to balance these concerns would be to look at both f and g (the two operands to the multiplication), and choose the size of the chunks of f and g simultaneously (somehow). However, we prefer to convert polynomials to the chunky representation independently, for a few reasons: it will simplify the algorithms considerably, allowing for more efficiency in the conversion step, and it will make the computation of some chain of multiplications (rather than just one at a time) much faster, since we will avoid converting between representations except at the beginning and the end of the computation.

Our approach to balancing the need to minimize $\sum d_i$ and to allow some spaces into the chunks will be the use of a *slack variable*, which we call ω. Really this is just the constant hidden in the big-O notation when we say $\sum d_i$ should be $O(s)$ as in Lemma 3.3.

The algorithm to convert a single polynomial from the sparse to the chunky representation is given below.

We start by inserting every possible gap between totally dense chunks (in order) into a *doubly-linked heap*. This is a doubly-linked list embedded in an array-based max-heap, so that each gap in the heap has a pointer to the locations of adjacent gaps.

The key for the max-heap will be a score we assign to each gap. This score will be the ratio between the value of $\prod(d_i+1)$ with and without the gap included, raised to the power $(1/r)$, where r is the length of the gap. So high "scores" indicate an improvement in the value of $\prod(d_i + 1)$ will be achieved if the gap is included, and not too much extra space will be

Algorithm `SparseToChunky`

Input: $f \in \mathsf{R}[x]$ in the sparse representation, and slack variable $\omega \geq 1$

Output: Chunky representation of f with $\sum d_i \leq \omega s$.

1: $r \leftarrow s$
2: $H \leftarrow$ doubly-linked heap with all possible gaps from f and corresponding scores
3: **while** $r \leq \omega s$ **do**
4: Extract gap with highest score from heap
5: Remove gap from chunky representation, update neighboring scores, and add size of gap to r
6: **end while**
7: Put back in the most recently removed gap
8: **return** Chunky representation with all gaps which still appear in H

introduced.

We then continually remove the gap with the highest score from the top of the heap, "fill in" that gap in our representation (by combining the chunks surrounding it into a single chunk), and update the scores of the adjacent gaps. Since we have a doubly-linked heap, and since there can't possibly be more gaps than the number of terms, all this can be accomplished with $O(s)$ word operations at each step. There can be at most s steps, for a total cost of $O(s \log s)$, which is linear in the size of the input from the lower bound on the size of the sparse representation. So we have the following:

Theorem 3.4. *Algorithm* `SparseToChunky` *returns a chunky representation satisfying* $\sum d_i \leq \omega s$ *and runs in* $O(s \log s)$ *time, where* s *is the number of nonzero terms in the input polynomial.*

3.3 Conversion from Dense

Converting from the dense to the chunky representation is more tricky. This is due in part to that fact that, unlike with the previous case, the trivial conversion does not give a minimum value for the function we want to minimize, which in this case is $\prod(d_i + 1)$. As a result, the algorithm here is a bit more complicated, and we do not give a complete proof.

Let $S_1, S_2, \ldots, S_k$ denote gaps of zeroes between dense chunks in the target representation, ordered from left to right. The algorithm is based on the predicate function $P(S_1, S_2, \ldots, S_k)$, which we define to be true iff inserting all gaps $S_1, \ldots, S_k$ into the chunky representation gives a smaller value for $\prod(d_i+1)$ than just inserting the single gap S_k. Since these gaps are in order, we can evaluate this predicate by simply comparing the products of the sizes of the chunks formed between $S_1, \ldots, S_k$ and the length of the single chunk formed to the left of S_k.

Our algorithm is given below. We maintain a stack of gaps $S_1, \ldots, S_k$ satisfying $P(S_1, \ldots, S_i)$ is true for all $2 \leq i \leq k$. This stack is updated as we move through the array from left to right in a single pass; those gaps remaining at the end of the algorithm are exactly the ones returned in the representation.

Algorithm `DenseToChunky`

Input: $f \in \mathsf{R}[x]$ in the dense representation

Output: A chunky representation for f satisfying $\prod(d_i + 1) \in O(n)$

1: $G \leftarrow$ stack of gaps, initially empty
2: $i \leftarrow 0$
3: **for each** gap S in f, moving left to right **do**
4: $k \leftarrow |S|$
5: **while** $P(S_1, \ldots, S_k, S) \neq$ **true do**
6: Pop S_k from G and decrement k
7: **end while**
8: Push S onto G
9: **end for**
10: **return** Chunky representation only with gaps remaining in G

Theorem 3.5. *Algorithm* `DenseToChunky` *always returns a representation containing the maximal number of gaps and satisfying* $\prod(d_i + 1) \leq n$ *and runs in* $O(n)$ *time, where the degree of the input is less than* n.

Proof. For the correctness, we first observe that $P(S_1, \ldots, S_k, S_\ell)$ is true only if $P(S_1, \ldots, S_k, S_{\ell'})$ is true for all $\ell' \leq \ell$. Then a simple inductive argument tells us that, the first time we encounter the

gap S_i and add it to the stack, the stack is trimmed to contain the maximal number of gaps seen so far which do not increase $\prod(d_i+1)$. When we return, we have encountered the last gap S_t which of course is required to exist in the returned representation since no nonzero terms come after it. Therefore, from the definition of P, inserting all the gaps we return at the end gives a smaller value for $\prod(d_i+1)$ than using no gaps.

The complexity comes from the fact that we push or pop onto G at every iteration through either while loop. Since we only make one pass through the polynomial, each gap can only be pushed and popped onto the stack at most once. Therefore the total number if iterations is linear in the number of gaps, which can never be more than $n/2$.

To make each calculation of $P(S_1, \ldots, S_k, S)$ run in constant time, we will just need to save the calculated value of $\prod(d_i + 1)$ at each time a new gap is pushed onto the stack. This means the next product can be calculated with a single multiplication rather than k of them. Also note that the product of degrees stays bounded by n, so intermediate products do not grow too large. □

The only component missing here is a slack variable ω. For a practical implementation, the requirement that $\prod(d_i + 1) \leq n$ is too strict, resulting in slower performance. So, as in the previous section, we will only require $\sum \log(d_i + 1) \leq \omega \log n$, which means that $\prod(d_i + 1) \leq n^\omega$, for some positive constant ω. This changes the definition and computation of the predicate function P slightly, but otherwise does not affect the algorithm.

4 Implementation

A complete implementation of adaptive chunky multiplication of dense polynomials has been produced using Victor Shoup's C++ library NTL [12], and is available for download from the author's website. This is an ideal medium for implementation, as our algorithms rely heavily on dense polynomial arithmetic being as fast as possible, and NTL implements asymptotically fast algorithms for dense univariate polynomial arithmetic, and in fact is often cited as containing some of the fastest such implementations.

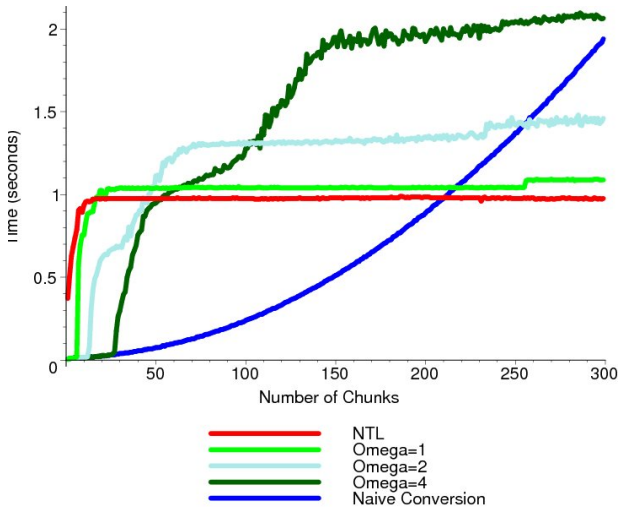
Although the algorithms we have described work over an arbitrary ring R, recall that in our analysis we have made a number of assumptions about R: Ring elements use constant storage, ring operations have unit cost, and the multiplication of degree-n polynomials over R can be performed with $O(n \log n)$ ring operations. To give the best reflection of our analysis, our tests were performed in a ring which makes all these assumptions true: $\mathbb{Z}_p$, where p is a word-sized "FFT prime" which has a high power of 2 dividing $p - 1$.

As in any practical implementation, especially one that hopes to ever be competitive with a highly-tuned library such as NTL, we employed a number of subtle "tricks", mostly involving attempts to keep memory access low by performing computations in-place whenever possible. We also had to implement the obvious algorithm to multiply a high-degree polynomial by a low-degree one as discussed at the end of Section 2, since (surprisingly) NTL apparently does not have this built-in. However, the relatively low crossover points of our algorithms, as shown below, indicates that more fine tuning is probably necessary to make them practical.

For the tests, we fixed the degree at 10 000 and randomly inserted small chunks into the polynomial. Each chunk has degree around 10 and tests were performed with 1 to 300 such chunks. We compared NTL's default multiplication method, which will always use FFT multiplication when the degree is this high, to our chunky multiplication method as described above. We also compared different strategies for converting the input: the naïve method of simply choosing every possible gap, and Algorithm `DenseToChunky`, with a few different choices for the slack variable ω. The results are presented in Figure 1.

We observe that the naïve conversion is best for very sparse and "friendly" polynomials, but the cost is growing very quickly. Also, increasing the slack variable makes algorithm `DenseToChunky` run faster for a while, but the ultimate ratio between it and the dense method will be greater.

Figure 1: Timing comparisons for Chunky Multiplication



5 Other Ideas for Adaptive Multiplication

5.1 Equally-Spaced Terms

Suppose many of the terms on a polynomial $f \in \mathsf{R}[x]$ are spaced equally apart. If the length of this common distance is k, then we can write $f(x)$ as $f_D(x^k)$, where $f_D \in \mathsf{R}[x]$ is dense with degree less than n/k. Now say we want to multiply f by another polynomial $g \in \mathsf{R}[x]$, where $\deg g < m$, and without loss of generality assume $m \leq n$. Then similarly write $g(x) = g_D(x^\ell)$. If $k = \ell$, then to find the product of f and g, we just compute $h_D = f_D g_D$ and write $f \cdot g = h_D(x^k)$. The total cost is only $O((n/m)\mathsf{M}(m/k))$, which by our assumption is $O((n/k)\log(m/k))$.

If $k \neq \ell$, the algorithm is a bit more complicated, but we still get a significant improvement. Let r and s be the greatest common divisor and least common multiple of k and ℓ, respectively. Split f into ℓ/r polynomials, each with degree less than n/s, as follows:

$$f(x) = f_0(x^s) + f_1(x^s) \cdot x^k + \cdots + f_{l/r-1}(x^s) \cdot x^{s-k}.$$

Similarly, split g into k/r polynomials $g_0, g_1, \ldots, g_{s/k-1}$, each with degree less than m/s. Then to multiply f by g, we compute all products $f_i g_j$, then multiply by powers of x and sum to obtain the final result. The total complexity in this more general case, assuming again that $\mathsf{M}(n) \in O(\log n)$, is $O((n/r)\log(m/s))$. So even when k and ℓ are relatively prime, we still perform the multiplication faster than any dense method.

As usual, identifying the best way to convert an arbitrary polynomial into this representation will be the most challenging step algorithmically. We will actually want to write f as $f_D(x^k) + f_S$, where $f_S \in \mathsf{R}[x]$ is sparse with very few nonzero terms, representing the "noise" in the input. To determine k, we must find the gcd of "most" of the exponents of nonzero coefficients in f, which is a nontrivial problem when we are restricted by the requirement of linear-time complexity. We will not go into further detail here.

5.2 Coefficients in Sequence

This technique is best explained by first considering an example. Let $\mathsf{R} = \mathbb{Z}$, $f = 1 + 2x + 3x^2 + \cdots + nx^{n-1}$, and $g = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$, where $b_0, b_1 \ldots, b_{m-1}$ are arbitrary integers. Let $h = fg$ be the product that we want to compute. Then the first n coefficients of h (starting with the constant coefficient) are $b_0, (2b_0 + b_1), (3b_0 + 2b_1 + b_2), \ldots$. We can compute these in linear time by initializing an accumulator with b_0, and for $i = 1, 2, \ldots, n-1$, we compute the coefficient of x^i by adding b_i to the accumulator, and adding the accumulator to the value of the previous coefficient. The high-order terms can be constructed in the same way.

So we have a method to compute fg in *linear time* for any $g \in \mathsf{R}[x]$. In fact, this can be generalized to the case where the coefficients of f form any arithmetic-geometric sequence. That is, $f = a_0 + a_1 x + \cdots$ and there exist constants $c_1, c_2, c_3, c_4 \in \mathsf{R}$ such that $a_i = c_1 + c_2 i + c_3 c_4^i$ for all i. The number of such sequences will be exponential in the size of the ring R, so many polynomials will fit into this category.

Now note that, if we wish to multiply $f, g \in \mathsf{R}[x]$, only one of the two input polynomials needs to have

sequential coefficients in order to compute the product in linear time. To recognize whether this is the case, we start with the list of coefficients, which will be of the form $(c_1 + c_2 i + c_3 c_4^i)_{i \geq 0}$ if the polynomial satisfies our desired property. We compute successive differences to obtain the list $(c_2 + c_3(c_4 - 1)c_4^i)_{i \geq 0}$. Computing successive differences once more and then successive quotients will produce a list of all c_4's if the coefficients form an arithmetic-geometric sequence as above. We can then easily find c_1, c_2, c_3 as well.

In practice, we will again want to allow for some "noise", so we will actually write $f = \sum (c_1 + c_2 i + c_3 c_4^i) x^i + f_S$, for some very sparse polynomial $f_S \in \mathsf{R}[x]$. The resulting computational cost for multiplication will be only $O(n)$ plus a term depending on the size of f_S.

6 Conclusions

We have seen some approaches to multiplying polynomials in such a way that we handle "easier" cases more efficiently, for various notions of easiness. These algorithms have the same worst-case complexity as the best known methods, but will be much faster if the input has certain structure.

However, our preliminary implementation seems to indicate that the input polynomials must be very structured in order to obtain a practical benefit. Adaptive sorting algorithms have encountered the same difficulty, and those algorithms have come into wide use only because almost-sorted input arises naturally in many situations. To bring what may currently be interesting theoretical results to very practical importance, we will need to more carefully investigate the properties of high-degree polynomials which people actually want to multiply, and see if they often contain any structure which we could exploit. Perhaps concentrating only on certain domains, such as very small finite fields, could also provide more easy cases for adaptive algorithms.

There is much other future work as well in working out the details of all the approaches put forth here. In fact, some combination of the three approaches could lead to better results on real input. In addition, it would be interesting to compare adaptive multiplication performance in the case of sparse polynomials, where the contrast between the fast dense methods we use here and the standard sparse methods might be more striking.

References

[1] Peter Bürgisser and Martin Lotz. Lower bounds on the bounded coefficient complexity of bilinear maps. *J. ACM*, 51(3):464–482 (electronic), 2004.

[2] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.*, 28(7):693–701, 1991.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[4] Martin Fürer. Faster integer multiplication. pages 57–66, 2007.

[5] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, second edition, 2003.

[6] Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *J. Symbolic Comput.*, 36(3-4):365–400, 2003. International Symposium on Symbolic and Algebraic Computation (ISSAC'2002) (Lille).

[7] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Dokl. Akad. Nauk SSSR*, 7:595–596, 1963.

[8] Kurt Mehlhorn. *Data structures and algorithms. 1.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1984. Sorting and searching.

[9] Michael B. Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In *CASC*, pages 295–315, 2007.

[10] Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Appl. Math.*, 59(2):153–179, 1995.

[11] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

[12] Victor Shoup. NTL: A Library for doing Number Theory. Online, http://www.shoup.net/ntl/, 2007.

[13] David R. Stoutemeyer. Which polynomial representation is best? In *Proc. 1984 MACSYMA Users' Conference*, pages 221–244, Schenectady, NY, 1984.

[14] Thomas Yan. The geobucket data structure for polynomials. *J. Symbolic Comput.*, 25(3):285–293, 1998.