Practical Fast Polynomial Multiplication

Robert T. Moenck

Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

The "fast" polynomial multiplication algorithms for dense univariate polynomials are those which are asymptotically faster than the classical $O(N^2)$ method. These "fast" algorithms suffer from a common defect that the size of the problem at which they start to be better than the classical method is quite large; so large, in fact that it is impractical to use them in an algebraic manipulation system.

A number of techniques are discussed here for improving these fast algorithms. The combination of the best of these improvements results in a Hybrid Mixed Basis FFT multiplication algorithm which has a cross-over point at degree 25 and is generally faster than a basic FFT algorithm, while retaining the desirable O(N log N) timing function of an FFT approach.

The application of these methods to multivariate polynomials is also discussed. The use is advocated of the Kronecker Trick to speed up a fast algorithm. This results in a method which has a cross-over point at degree 5 for bivariate polynomials. Both theoretical and empirical computing times are presented for all algorithms discussed.

Introduction

One of the most frequently used algorithms in any algebraic manipulation system is multiplication of expressions. For example:

$$(e^{y/2}\sin(x) - \cos(x))*(\sin(x) - e^{y/2}\cos(x)).$$

One way we might compute this product is to view it in the polynomial ring $Q[e^{y/2}\sin(x), \cos(x)]$, i.e. the functions are considered indeterminants over the ring Q. Then we might form the product using a polynomial multiplication algorithm:

$$[-(e^{y/2})^2\sin(x)\cos(x) + (\sin^2(x)+\cos^2(x))e^{y/2} - \sin(x)\cos(x)].$$

Finally, we might simplify the resulting product to:

$$e^{y/2} - 1/2(e^y+1)\sin(2x).$$

This means we can break the general problem of expression multiplication into two subproblems:

a) polynomial multiplication and
b) simplification of expressions.

Ignoring the latter completely, it is the former problem that this paper considers. Since a lot of polynomial multiplication, in one form or another, is done in an algebraic manipulation system, we would like to use the best possible algorithm. The measure of "best" we will use here is the number of steps used in the algorithm.

A second reason for interest in the problem is that over the last several years, a class of algorithms, for operations on polynomials have been developed, whose efficiency (the number of steps they use) is related to the efficiency of polynomial multiplication. In simple terms, if polynomial multiplication can be done quickly, then these algorithms can be performed quickly. A partial list of such algorithms based on an O(N log N) multiplication algorithm is given below.

Polynomial Operation	Operation Count	Reference
Multiplication	$O(N \log N)$	[Pol 71]
Division with Remainder	$O(N \log N)$	[Str 73]
GCD and Resultants	$O(N \log^2 N)$	[Moe 73]
Multipoint Evaluation	$O(N \log^2 N)$	[Bor 74]
Interpolation	$O(N \log^2 N)$	[Bor 74]
Factoring	$O(N \log^4 N)$	[Moe 75]

A list of some fast algebraic algorithms

In the case of dense univariate polynomials of degree n, the classical polynomial multiplication algorithm uses $O(n^2)$ steps to compute a product. Workers in the field of algorithmic complexity have developed several algorithms for polynomial multiplication. These are the so-called "fast" algorithms which use O(f(n)) steps to form a product. They have the property that:

$$\lim_{n\to\infty} \frac{f(n)}{n^2} \to 0$$

i.e. they are asymptotically faster than the classical method. In more concrete terms, for large degrees they are faster than the classical method and this superiority increases as n does.

The problem with these "fast" methods is that they tend to be impractical. The degree at which they start being better than the classical method (the cross-over point) is fairly high. In fact the cross-over points tend to be so high that they are larger than most problems encountered in algebraic manipulation systems. This paper sets out to answer the questions:

1) Can a fast algorithm be made practical?
2) What is the best all-round polynomial multiplication algorithm?

In section 1, we review the classical method and two fast methods and look at their performance. The performance of the algorithms is investigated both from the theoretical point of view and in terms of the run times of Algol-W implementations of the algorithms. Section 2 deals with improvements to one of the fast methods. These lead to an algorithm with the same asymptotic properties as the best of the fast algorithms, but is more practical and in general, faster than such a method. In section 3, we examine methods for multivariate multiplication, and suggest a new approach to this problem. Finally, in section 4 we summarise the results and see what conclusions can be drawn.

Three Basic Methods

In the first two sections we will consider dense univariate polynomials i.e. polynomials $a(x) \in R[x]$

where $a(x) = \sum_{i=0}^{n} a_i x^i$ and $a_i \neq 0$ for $0 \leq i \leq n$.

The base ring R will usually be a finite field Z_p. For some of the algorithms, the base ring could equally well be the integers Z or rationals Q or any other suitable number ring. Coefficients in such rings can be mapped into the finite field Z_p and recovered by means of the Chinese Remainder Algorithm (cf. [Lip71], so there is no loss of generality in considering only finite fields.

1.1) The Classical Method

The Classical Method for polynomial multiplication is the one which is commonly employed in algebraic manipulation systems. If we have two polynomials:

$$a(x) = \sum_{i=0}^{n} a_i x^i \quad , \quad b(x) = \sum_{j=0}^{m} b_i x^j$$

then their product c(x) can be written as:

$$1.1)\ldots\ c(x) = a(x)*b(x) = \sum_{\ell=0}^{m+n} \left(\sum_{\ell=i+j} a_i * b_j \right) x^{\ell}$$

Expression (1.1) can be directly translated into an algorithm for computing this product. This algorithm uses:

(n+1) (m+1) coefficient multiplications and

nm coefficient additions.

For a total of : $2mn + (m+n) + 1$ coefficient operations.

If n = m this is : $2n^2 + O(n)$ operations.

While this function only counts the number of arithmetic operations which are used in the algorithm, all other types of operations (e.g. subscript calculations, store, fetch etc.) will increase proportionally.

1.2) The Fast Fourier Transform Method

In recent years the study of algorithmic complexity has produced a number of multiplication algorithms having operation counts which grow more slowly than $O(n^2)$. The best of these to date is the one based on the Fast Fourier Transform (FFT) (cf. [Pol 71]). Gentleman and Sande [Gen 66] appear to have been the first to propose the use of the FFT for polynomial multiplication.

The FFT provides an efficient method for evaluating a polynomial at M roots of unity $\{w_i\}$ in a finite field. Since evaluation of a polynomial induces a homomorphism from the polynomial ring to the ring of coefficients, the values of the product polynomial can be found by multiplying the values of the multiplicants pairwise.

i.e. if $c(x) = a(x) * b(x)$
then $c(w_i) = a(w_i) * b(w_i)$.

Given the values of the product $\{c(w_i)\}$ at the points $\{w_i\}$, the coefficients of the product can be recovered by interpolation. This interpolation can be performed by means of the inverse FFT.

The advantage to this method is that the FFT can be performed very quickly. If $M = 2^r$ then this algorithm uses:

2 Forward FFTs	$= 2(M + 3/2\ M \log M)$
1 Multiplication of values	$= M$
1 inverse FFT	$= 2M + 3/2\ M \log M)$

For a total of : $9/2 \log M \log M + 5M$ coefficient operations.

Note: all logarithms used in this paper will be base 2 unless otherwise specified.

We will postpone the details of computing the FFT until section 2.1. In applying the FFT method to multiply two polynomials of degree m and n the usual approach is to choose $r = \lceil \log (m+n+1) \rceil$. Then the polynomials are considered to be of degree $M-1 = 2^r-1$ by adding a suitable number of zero leading coefficients. This approach does handicap the algorithm by making the polynomials appear larger than necessary, but does not destroy the desired O(M log M) operation count.

Applying this method to two polynomials of degree n, the number of steps in the algorithm is:

$$9N \log N + 19 N$$

where $N = 2^k$, $k = \lceil \log (n+1) \rceil$.

1.3) Comparison of the Methods

Table 1 compares the two methods both in terms of their operational counts and in terms of the actual run times of an Algol-W implementation of the algorithms. It can be seen that the operation count predicts that the FFT method will start being more efficient than the classical method when multiplying polynomials of degree 31.

The empirical run-times bear out this cross-over point. They also illustrate the characteristic quadratic behaviour of the classical method and the slightly faster than linear behavior of the FFT method. This table of performance might be used to advocate the use of the FFT method as the standard algorithm to be used in an algebraic manipulation system. However, several problems arise:

a) For degrees below the cross-over point the FFT method is very much worse than the classical method. This problem might be resolved by combining the two methods and using the FFT method for large problems and the classical for small.

Even if this combined approach is taken there are problems.

b) The cross-over point is quite high. Although products of degree 31 do appear in algebraic manipulation, they do not arise that frequently. Is it worth the trouble of implementing a fast algorithm if it will only be used occasionally?

c) The table does not illustrate the fact that the FFT is performed on sequences of length 2^k. This means that to multiply two polynomials of degree 32, a sequence of length 128 must be used. For this case the implementation of the FFT method takes 2.3 time units and the classical method about 1.1 units. In other words, the FFT method has not one, but three cross-over points with respect to the classical method (at degrees 31, 50 and 70). (see Fig. 1)

It was with these problems in mind that an attempt was made to improve on the existing multiplication algorithms.

1.4) Karatsuba's Algorithm

The first attempt involved looking at an alternative to the FFT method. It was hoped that a different algorithm might be more efficient than the classical method, but not much worse than the FFT method. The method used is due to Karatsuba (see [Kar 62]). Fateman [Fat 72] calls this algorithm SPLIT.

Karatsuba's algorithm applied to two polynomials of degree n-1:

$$a(x) = a1(x)x^{n/2} + a0(x)$$
$$b(x) = b1(x)x^{n/2} + b0(x)$$

where deg(a1), deg(a0), deg(b1), deg(b0) $\leq$ n/2-1.

Then: $c(x) = a(x) * b(x)$

$$= (x^n + x^{n/2})\ a1*b1 + (x^{n/2} + 1)$$
$$a0*b0 - x^{n/2}(a1 - a0)*(b1-b0)$$

The three products of half the degree are formed by applying the algorithms recursively. With care, the addition of the coefficients of the products can be done in 7/2n steps. Therefore, the number of steps used by the algorithm to multiply two polynomials of degree n-1 can be expressed in the following recurrence relation:

$$T(n) = 3T(n/2) + 7/2n$$
$$= 23/2\ n^{\log 3} - 7n$$
$$= 23/2\ n^{1.585} - 7n \text{ coefficient operations.}$$

This means that Karatsuba's algorithm is more efficient than the classical method, but not as efficient as the FFT method, for large inputs. However, it was hoped that the simplicity of the algorithm (in its recursive form it requires about as many line of ALGOL code as the classical method) might give it a lower cross-over point with respect to the classical method.

Table II compares the empirical run times of the classical method with Karasuba's algorithm. The second column gives the time for the first version of the algorithm, where the recursion was carried out down to the level of the coefficients (i.e. degree0). It may be seen that at degree 31 Karatsuba's algorithm takes three times longer than the classical method. Also the method exhibits the same poor performance for small degrees as the FFT method.

On reason for the poor performance of the algorithm was that it had to labour very hard to perform many operations both in the form of procedure calls and arithmetic operations close to the base of its recursion. Therefore, one way to improve the algorithm would be to stop the recursion at some small degree and perform this multiplication classically. The remainder of the columns of Table II show the times when the base of recursion was polynomials of degree 1, 3 or 7.

It can be seen that KPM-7 achieves a cross-over point at degree 15, which is the best of the fast methods. However, at about degree 63 the FFT method is almost as fast as KPM-7. For larger degrees the KPM algorithm will be slower than the FFT method. An advantage of the KPM algorithm shared by the classical method, is that it can be carried out in the original polynomial ring. This means that coefficients do not require any mapping to a residue representation with its attendant overhead in computation.

The FFT Revisited

2.1) Two Views of the FFT

A second attempt involved looking for ways to improve the FFT method. In order to discuss these,

it is first necessary to review the algorithm for computing the FFT.

There are in fact many algorithms for computing FFTs. Each has its advantages, and the choice of which one to use, depends on the nature of the problem. There is a considerable literature on the FFT, Aho et al [Aho 74] give a brief survey. Gentleman and Sande [Gen 66] discuss several such algorithms and compare their merits.

The first view considered here is the "evaluation" form. For this the transform of a polynomial a(x) of degree N is considered as evaluation of a(x) at the points $\{w^i\}$ where $0 < i < N$ and w is a primitive N-th root of unity of the finite field Z_p, i.e. $w^N = 1$, $w^j \neq 1$ for $0 < j < N$.

This can be viewed as:

$$a(w^i) = \sum_{j=0}^{N-1} a_j w^{ij}$$

$$= \sum_{j=0}^{N/2-1} a_{2j} w^{2ij} + w^i \sum_{j=0}^{N/2-1} a_{2j+1} w^{2ij}$$

$$= a_{even}(w^{2i}) + w^i \; a_{odd}(w^{2i})$$

For $0 < i < n$.

Note: $a'(w^{2(N/2+i)}) = a'(w^N w^{2i}) = a'(w^{2i})$

where a'(x) is $a_{even}(x)$ or $a_{odd}(x)$.

Now $a_{even}(x)$ and $a_{odd}(x)$ are two polynomial forms, each with half the number of terms of a(x), which are to be evaluated at a point w^{2i} which is also a root of unity. Thus the same process can be applied. The inverse transform of $\{a(w^i)\}$ can be computed as:

$$\sum_{i=0}^{N-1} a(w^i) w^{-ij} = \sum_{i=0}^{N-1} w^{-ij} \sum_{\ell=0}^{N-1} a_\ell w^{i\ell}$$

$$= \sum_{\ell=0}^{N-1} a_\ell \left(\sum_{i=0}^{N-1} w^{i(\ell-j)} \right) = N a_j$$

since $\sum_{i=0}^{N-1} w^{ik} = \begin{cases} N \text{ if } k = 0 \bmod N \\ 0 \text{ otherwise.} \end{cases}$

$\{w^{-1}\}$ is also a primitive root of unity and so the same algorithm as the forward FFT can be used.

This leads, naturally to a recursive algorithm for computing the transform. However, for an actual implementation, a recursive algorithm is not the most suitable since recursion is usually an expensive operation on many computers. It is possible to express this algorithm in an iterative "bottom-up" form which is much more efficient.

A second consideration is the storage used by the algorithm. It would seem that extra storage might be needed to store the $a_{even}(x)$ and $a_{odd}(x)$ results at each level, so that the coefficients of a(x) are not overwritten while they are still needed. In fact the algorithm can be coded in such a fashion that it works "in-space" and the values which are overwritten are those which are no longer required.

A peculiarity of the algorithm is the order in which it selects coefficients. Selecting even and odd parts means that the lowest order bit of the index of a coefficient is used as a selector. If this is carried through several levels, it is apparent that the indices of the coefficients are being read as binary numbers from right to left, rather than left to right as is conventional. This is the digit reverse form discussed by Cooley and Tukey [Coo 65]. One problem with this method of indexing is that it is necessary to perform "random" accessing of coefficients to apply the algorithm. This is easy if the coefficients are stored in an array, but in the realm of symbolic mathematics, polynomials are usually represented by linked lists ordered by exponent and this mode of accessing is quite unacceptable.

An alternative algorithm can be obtained by viewing the computation as being carried out modulo $x^N - w^N$.

Hence: $a(x) = \sum_{i=0}^{N-1} a_i x^i \bmod x^N - w^N$

$$= \left(\sum_{i=N/2}^{N-1} a_i x^{i-N/2} \right) x^{N/2} + \sum_{i=0}^{N/2-1} a_i x^i$$

$$= a_{high}(x) \; x^{N/2} + a_{low}(x)$$

Then $a(x) \equiv a(x) \bmod x^{N/2} - w^{N/2}$

$$= a_{low}(x) + a_{high}(x) \; w^{N/2} = a'(x)$$

and $a(x) \equiv a(x) \bmod x^{N/2} + w^{N/2}$

$$= a_{low}(x) - a_{high}(x) \; w^{N/2} = a''(x)$$

Note that $x^{N/2} + w^{N/2} = x^{N/2} - w^N$ and so the process can be recursively or iteratively applied to a'(x) and a"(x) until the values of $a(x) \bmod x - w^i$ are reached. It is apparent that a'(x) and a"(x) can be generated simultaneously by scanning the top and bottom halves of a(x) sequentially. This algorithm can also be performed "in-space". Perhaps less obvious, is the fact that, the powers of w used by the algorithm are accessed in digit reverse order and that the transformed representation of the polynomial is also permuted in this fashion.

This is less serious than the previous case since the powers of w can be generated in this order in linear time. Secondly, since the products of two transformed polynomials are formed pairwise, any permutation of the transformed

representation is irrelevant, and the inverse transform unscrambles the results. As Singleton [Sin 67] points out, this permutation must appear somewhere in an FFT algorithm. We will call this form of the FFT the sequential access method.

In the case of the second form of the FFT the inverse transform may be computed by forming a(x) out of a'(x) and a"(x) as follows:

$$a(x) = 1/2\ [a'(x) + a''(x)]x^{N/2} + \frac{W^{-N/2}}{2}\ [a'(x) - a''(x)]$$

$$= a_{high}(x)\ x^{N/2} + a_{low}(x)$$

For either algorithm it may be seen that each level of the FFT may be accomplished in 3/2N steps. Each level divides the problem into two sub-problems of half the degree, each of which is to be evaluated at half the number of points. Therefore, the number of steps used by the algorithms exhibits the characteristic recurrence relation:

$$T(N) = 2T(N/2) + 3/2N$$

$$= 3/2\ N \log N + N \text{ steps.}$$

During the inverse transform the factor of 1/2 is not usually divided out at each stage but accumulated and divided out in a final step as $1/2^{\log N} = 1/N$. Thus the inverse FFT takes: 3/2N log N + 2N steps.

2.2) A Hybrid Method

The basic strategy for the FFT polynomial multiplication algorithm applied to two polynomials a(x) and b(x) of degree n is as follows:

Basic Radix-2 FFT Multiplication Algorithm

1) Choose $N > 2n+1$ such that $N=2^{\lceil \log(2n+1) \rceil}$.

2) Perform the N point FFT on a(x) and b(x) to give the sequences $\{A_i\}$ and $\{B_i\}$.

3) Multiply the sequences pairwise to give a sequence $\{C_i\}$.

4) Perform the N point FFT on $\{C_i\}$ to give a polynomial c(x). This polynomial is the product of a(x) and b(x).

The problem with this approach is that it chooses a gross maximum degree N for the product. All products in the range $N/2 < 2n+1 < N$ are essentially treated the same way. One consequence of this is that for large problems within such a range the time for the multiplication is constant. For example, it takes no more time for n=240 than it does for n=130.

On the other hand, at the end of such a range there is a large jump in time from one interval to the next. Clearly it is not a great deal more dfffficult to multiply polynomials of degree 32 than it is for degree 31. However, the basic FFT strategy does not reflect this fact.

One way we can improve on this situation is to notice that the FFT method is a powerful engine for multiplying polynomials. We can use it to best effect by giving it the "difficult" problems and solving the "easy" ones some other way. In polynomial multiplication the "easy" problems are the leading and trailing coefficients of the product. These involve only a few terms and operations and are best done using a classical algorithm. The "difficult" problems are the intermediate coefficients of the product. These involve most of the coefficients of the multiplicands and many operations.

To see how this might be done we can examine the inverse transform for the j-th coefficient of the product.

$$c_j = \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{-ij}$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{N/2 i} w^{(-N/2-j)i}$$

$$= \frac{1}{N} \sum C_i (-1)^i w^{-(N/2+j)i}$$

and

$$c_{N/2+j} = \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{-(N/2+j)i}$$

Then consider extracting the even order terms from each sum:

$$p_j = c_j + c_{N/2+j}$$

$$= \frac{2}{N} \sum_{i=0}^{N/2-1} C_{2i}\ w^{-(N/2+j)2i}$$

Note that the $\{p_j\}$ are just the result of an N/2 point inverse FFT and that

$$\left.\begin{aligned} c_j &= p_j - c_{N/2+j} \\ c_{N/2+j} &= p_j - c_j \end{aligned}\right\} \text{for } 0 < j < N/2$$

This means we could use an N/2 point FFT to compute the $\{p_j\}$ (i.e. the "difficult" terms) and and use a classical method to compute the leading and trailing terms. The two sets of terms can be combined to find the coefficients of the product.

This leads to an alternate strategy for a Hybrid Method computing a product of degree m.

The Hybrid Multiplication Algorithm

1) Choose $N = 2^{\lfloor \log m \rfloor}$.

2) Apply the FFT method to find the $\{p_j\}$.

3) Form the coefficients C_i classically for $0 \le i \le \lfloor m/2 \rfloor + N/2$ and $N/2+\lfloor m/2 \rfloor+1 \le i \le m$. i.e. the leading and trailing terms.

4) Find the remaining coefficients of the product using the relations:

$$c_j = p_j - c_{N/2+j} \text{ , for } \lfloor m/2 \rfloor + N/2 + 1 \le j \le N/2 + m$$

$$c_{N/2+j} = p_j - c_j \text{ , for } \lfloor m/2 \rfloor + N/2 + 1 \le j \le m$$

$$c_j = p_j \text{ , for the rest of the coefficients.}$$

Examining the algorithm, we see that step 2 takes: 9/2 N log N + 5N operations. Step 3 will take the same number of steps as multiplying two polynomials of degree m/2 - N/2 i.e. $\frac{1}{2}(m-N-1)^2 + (m-N-1)$ operations. There are (m-N) subtractions used in step 4. In summary we can state:

Theorem: The Hybrid multiplication algorithm takes:

2.1)... $(9/2\ N \log N + 5N) + ((\frac{m-N}{2})^2 + (m-N))$

operations, where $N = 2^{\lfloor \log m \rfloor}$. □

This can be compared to the Basic FFT method which takes:

2.2)... 9 N log N + 19N basic steps.

If $9/2\ N \log N + 14N > (\frac{m-N}{2})^2 + (m-N)$ then this method is faster than the basic FFT method. This inequality predicts that if: $N < m < N+120$ then the hybrid strategy is better than the basic FFT strategy. In fact, this cross-over point of 120 is borne out in empirical tests. In order not to make an $O(N^2)$ algorithm out of an O(N log N) algorithm, the hybrid method is used up to the point where 120 leading and trailing terms are computed classically. Above this point, the basic FFT method is used.

The performance of the hybrid strategy is shown in figures 1,2 and 3 and in table III. It leads to a much more smoothly growing timing function, which for large intervals, is faster than the basic FFT method. This Hybrid method crosses over with respect to the classical method at degree 25, a significant improvement over the degree 70 of the basic FFT method.

2.3) Mixed Basis FFTs

The FFT described in section 2.1 is called the radix 2 form since $N=2^k$. In fact the FFT need not be restricted to radix 2. N can be any highly composite number. For example radix 3 with $N = 3^j$ or $N = 2^k 3^j$, etc.

It would seem that another way to improve the efficiency of the FFT multiplication algorithm is to choose a value for N which more closely matches the size of the problem. If we allow mixed radix FFTs then we have a larger selection of Ns to choose from. However, the reason that the radix 2 FFT is the one most frequently used is that it is easiest to implement. This partly is due to the digit reverse permutation which is required for an iterative version of the algorithm.

The digit reverse permutation in radix 3 looks a little strange, but the permutation in a 2-3 radix FFT is stranger still. A number in the (2,2,...,2,3,...,3) number system has as its reverse form a number in the (3,...,3,2,...,2) number system (cf. Knuth [Knu 69]). This means that the permutation is not so easily computed.

An alternative is to use $N = \ell * 2^k$, where ℓ is some small prime (e.g. 3,5 or 7). This choice together with the sequential access algorithm allows the following variant of the FFT to be applied.

The Mixed Basis FFT Multiplication Algorithm

1) For a product of degree m, choose $N = 2^k \cdot \ell > m$ and a 2^k-th primitive root of unity w.

2) Perform the Sequential Access FFT algorithm, carrying out the computation modulo $x^N - w^{2^k}$. This means that the final step of the FFT yields 2^k pairs of polynomials $\{A_i(x)\}$ and $\{B_j(x)\}$ of degree ℓ-1 of the form:

$$A_i(x) \equiv a(x) \bmod x^\ell - w^i \text{ , } 0 \le i < 2^k$$

$$B_i(x) \equiv b(x) \bmod x^\ell - w^i \text{ , } 0 \le i < 2^k.$$

3) The products $C_i(x)$ and $A_i(x) * B_i(x) \bmod x^\ell - w^i$ can now be formed using the classical algorithm.

4) Finally the polynomials $C_i(x)$ have the inverse FFT applied to them, to yield the product c(x) in coefficient form.

Note that the results of the inverse FFT are divided by 2^k and not N in this case. Counting the cost for each of steps 2, 3 and 4 we have:

$$2(N + 3/2\ Nk) + (2^k * 2\ell^2) + (2N + 3/2\ Nk)$$

$$= 9/2\ Nk + N(4 + 2\ell) \text{ where } N = \ell \cdot 2^k > m.$$

We can summarise this in:

Theorem: The total number of steps for the Mixed Basis FFT Multiplication Algorithm is:

2.3)... 9/2 Nk + N(4 + 2ℓ), basic operations. □

An implementation of this algorith with ℓ = 1,3,5,7 was made. The performance curve of this code is shown in figures 1, 2 and 3 and in table III. This curve is much smoother than that given by either of the two previous methods. This is because the curve has many small plateaus,

where the Basic radix 2 FFT has only one. This technique is at least as fast as the Basic FFT strategy almost everywhere and in some cases faster than the Hybrid method. However, in general, the improvement is not as substantial as the Hybrid method.

The two improved methods can be combined together to give a Hybrid-Mixed-Basis FFT algorithm which is the best of the four methods (cf. figures 1, 2, 3 and table III). However, the improvements of the Hybrid and Mixed Basis algorithms do not combine together strictly additively and so the improvement of the combined method is not as large as might be expected.

This Hybrid-Mixed-Basis method can be tuned by choosing optimum values for ℓ and k and hence N in:

$$m/2 < N = \ell \cdot 2^k < m$$

for a product of degree m. How to find this algorithm is an object for further study. At the moment a "good" value is chosen by:

a) finding $N = 2^k > m/2$

b) while $m-N > 100$, increment N to a form $N = 2^k \cdot \ell$.

At the moment ℓ is always chosen as $\ell = 1,3,5$ or 7, but other choices might be better. In particular, if ℓ is a small multiple of 2, equation (2.3) predicts that the Mixed Basis method will be marginally faster than the Basic FFT method.

Multivariate Polynomials

So far we have only considered univariate polynomial multiplication. We have seen that the methods based on the FFT have a cross-over point about degree 25, with respect to classical multiplication. While problems of this size or larger do arise in algebraic manipulation, generally the problems are smaller. If we look at the multivariate case we see polynomials with similar low degrees but they tend to have a large number of coefficients. This raises the question, "Can the FFT methods be applied to the multivariate case?".

3.1) Recursive Classical Method

The standard approach to multiplication of polynomials:

$$a(x_1,\ldots,x_v), b(x_1,\ldots,x_v) \in R[x_1,\ldots,x_v]$$

is to view the polynomials as being represented recursively in:

$$a(X) \in R[x_1,\ldots,x_{v-1}][x_v]$$

i.e. as polynomials in the indeterminant x_v whose coeffients are polynomials in the remaining v-1 variables. Then the classical algorithm can be applied as before. Whenever the product of two coefficients is required, the polynomial multiplication algorithm calls itself recursively, until coefficients in the base ring R are found.

Let MCPM(n,v) be the number of steps required to multiply dense polynomials of degree n in each of v variables, using this method. Then:

$$\begin{aligned} MCPM(n,v) &= (n+1)^2 \text{ coefficient multiplications} \\ &\quad + n^2 \text{ additions to these products} \\ &= (n+1)^2\, MCPM(n,v-1) + n^2(2n+1)^{v-1} \end{aligned}$$

where MCPM(n,0)=1. The asymptotic form of this function is $MCPM(n,v) = O(n^{2v})$.

3.2 The Recursive FFT Method

One proposal for applying the FFT methods is to also apply them recursively. Viewing the FFT as a quick method to evaluate the polynomial at a set of points, we could use the following process.

1) Evaluate the $(n+1)^{v-1}$ coefficient polynomials in x_1 of a(X) and b(X) at m points. This yields m pairwise products of polynomials in v-1 variables to be formed. These products can be computed by applying this method recursively.

2) The coefficients of these products can then be interpolated using the inverse FFT to give the product of a(X) and b(X). Again the recursion stops at the pairwaise multiplication of base ring elements.

Let MFPM(n,v) be the number of steps required to multiply two polynomials of degree n in each of v variables. Then:

$$\begin{aligned} MFPM(n,v) &= 2(N+1)^{v-1}(m + 3/2m\log m) \\ &\quad + m\, MFPM(n,v-1) \\ &\quad + m^{v-1}(2m + 3/2m\log m) \end{aligned}$$

where $MFPM(n,0) = 1$ and $m = 2^r, \lceil r = \log(2n+1) \rceil$. The function is asymptotically $MFPM(n,v) = Q(v\, n^v \log n)$.

One problem with this method is that FFT methods are worthwhile only for large degree polynomials. Even with the modifications suggested in the previous section it appears intuitively that the polynomial must be of degree 25 in each variable before this method will pay dividends. This does not require many variables before the size of the problem is compounded out of any useful range.

3.3) The Kronecker Trick

A third possibility advocated here and in [Moe 76] is to map the multivariate problem into a univariate one and then apply an FFT method to the univariate case. Consider the multiplication of a(x,y) by b(x,y) where:

$$a(x.y) = (2y+1)x + (-y+2)$$

$$b(x,y) = (y+3)x + (4y-3)$$

Since the product will be of degree two in each variable we might try substituting $x = y^3$ in each of the polynomials, to obtain a'(y) and b'(y):

$$a'(y) = 2y^4 + y^3 - y + 2$$

$$b'(y) = y^4 + 3y^3 + 4y - 3$$

and $a'(y)*b'(y) =$

$$2y^8 + 7y^7 + 3y^6 - 7y^5 - 3y^4 + 3y^3 - 4y^2 + 11y - 6.$$

Taking the remainder of successive quotients with respect to $\{y^{3i}\}$, $2 > i > 0$, we invert the substitution to get:

$$(2y^2 + 7y + 3)x^2 + (7y^2 - 3y + 3)x +$$

$$(-4y^2 + 11y - 6).$$

This is the product of a(x,y) and b(x,y) as can be verified by carrying out the multiplication in some more conventional way. This method is known in the literature as "Kronecker's Trick".

For the general case the validity of the method is established by the following theorems shown in [Moe 76].

Theorem:

In the polynomial ring $R[x_1,\ldots,x_v]$, $v > 2$.

The mapping: $\phi : R[X] \longrightarrow R[x_1]$

$$\phi : x_i \longmapsto x_1^{n_i},\ 1 \le i \le v$$

where $n_v > \ldots > n_1 = 1$ is a homomorphism of rings.

□

Let $d_i(p)$ be the maximum degree of the polynomial p in variable x_i. The following theorem relates the $\{n_i\}$ of the mapping to d_i and estabilishes the validity of the inverse mapping.

Theorem:

Let ψ be the homomorphism of free R-modules defined by:

$$\psi : R[x_1] \longrightarrow R[x_1,\ldots,x_v]$$

$$\psi : x_1^k \longmapsto \begin{cases} 1 \text{ if } k = 0, \\ \psi(x_1^r) * x_i^q, \text{ otherwise} \end{cases}$$

where $n_{i+1} > k > n_i$, $k = q*n_i + r$, $0 \le r < n_i$

and $n_v > \ldots > n_1 = 1$.

Then for all $p(X) \in R[X]$, $\psi(\phi(p)) = p$

3.1... iff $\forall$ i, $1 \le i < v$,

$$\sum_{j=1}^{i} d_j(p) n_j < n_{i+1}.$$

□

Note that this result in effect shows that the set of polynomials, for which relation (3.1) holds, are isomorphic to their univariate images. Thus any polynomial ring operation on elements of this set, giving results in the set, will be preserved by the isomorphism. In this sense ϕ behaves like a ring isomorphism on the set of polynomials. Another way to view the mapping given in the theorems is:

$$\phi : x_i \longmapsto x_{i-1}^{m_i},\ \text{for } 1 < i \le v.$$

3.4) The Kronecker Trick Algorithm

We can apply these results in the following:

Kronecker Trick Algorithm

1) For two polynomials a(X) and b(X), find a bound on the maximum degree in each variable of their product:

$$m_i = d_i(a) + d_i(b) + 1.$$

2) Apply the Kronecker Trick mapping:

$$\phi : x_i \longmapsto x_{i-1}^{m_i},\ \text{for } 1=v,v-1,\ldots,2$$

to a(X) and b(X) to give two univariage polynomials $a'(x_1)$ and $b'(x_i)$.

3) Multiply the polynomials $a'(x_1)$ and $b'(x_1)$ to give a product of $c'(x_1)$. Any fast multiplication algorithm may be used.

4) Invert the Kronecker Trick mapping on $c'(x_1)$ to give a multivariate product c(X).

□

From the theorems given above, the result c(X) will be the product of a(X) and b(X). The Kronecker Trick mapping and its inverse can be accomplished by a linear scan over the terms of the polynomial. This means we can count the steps used in the algorithm as follows. Let MKPM(n,v) be the number of steps required to multiply two polynomials of degree n in each of v variables, using the Kronecker Trick and a univariate FFT method. Then:

$$MKPM(n,v) = (9/2m \log m + 5m)$$

where $m = 2^r$ and $r = \lceil \log (2n+1)^v \rceil$. Again the asymptotic form of the function is $MKPM(n,v) = O(v\, n^v \log n)$. However, the constants of proportionality are smaller in this case than in the recursive FFT algorithm

The three functions MCPM, MFPM and MKPM can be evaluated using their recursive definitions and an estimate of the cross-over points can be made. Some sample values for small n and v are displayed in table IV. The table illustrates the characteristic jumps in the timing function of the FFT methods. One effect of the Kronecker Trick is to smooth out these jumps to give a more uniform function. While the recursive FFT method is better than the Kronecker method at a few points,

generally the latter is faster. The cross-over points for the Kronecker method with respect to the classical method are shown in the following table.

variables	theoretical degree	empirical degree
2	10	6
3	7	3
4	4	
5	4	
6	3	

Some empirical tests using the Hybrid-Mixed-Basis FFT algorithm together with the Kronecker Trick were carried out. The results of these are shown in figure 4 and table IV. It can be seen that for bivariate polynomials a cross-over point at degree 5 and for trivariate polynomials a cross-over point at degree 3 is obtained.

Summary and Conclusions

We have surveyed some of the existing fast methods for computing the product of two polynomials. When contrasted with the classical algorithm for computing products, these methods tend to suffer from the common problem that they do not perform well for small degrees. A number of techniques were proposed to improve these methods. These techniques include:

1) Performing the basis step of Karatsuba's algorithm classically. If the degree of the basis was increased to 7, the algorithm attains a cross-over point at degree 15, with respect to classical multiplication.

2) Another technique used was the Hybrid FFT algorithm which computed the leading and trailing coefficients of the product classically, and left the intermediate terms to an FFT algorithm.

3) A Mixed-Basis FFT algorithm reduced the computation of one large product to that of a large number of small products, which could be computed classically.

4) A Hybrid-Mixed-Basis FFT algorithm combined the previous two methods. This last method was the best of the FFT methods considered here, attaining a cross-over point at degree 25 and having a much smoother performance curve.

The problem of multivariate multiplication was also considered. The Kronecker Trick was proposed as an alternative to the standard recursive approach to multivariate problems. Evidence was presented showing that the Kronecker Trick is superior to the standard methods. A cross-over point at degree 5 for bivariate polynomials was obtained in empirical tests.

When this research was begun, it was hoped that a definitive answer could be found for the question, "What is the best all-round polynomial multiplication algorithm(at least in the dense case)?". Instead of a definitive answer, more questions were found. Some of these were:

1) The techniques used to speed up the basic algorithms suggest that there are classes of methods for polynomial multiplication that are yet to be explored, e.g.

   a) The Hybrid method uses the classical algorithm to compute up to 60 of the leading and trailing terms of a product. Could some other algorithm (e.g. Karatsuba's) do this computation more efficiently?

   b) The Hybrid method uses the FFT to compute half of the terms of the product. Could the FFT be used to compute a quarter of the terms, leaving the rest to be formed classically? This might reduce the cross-over point for these methods.

   c) What is the optimum choice for the division of labour in a Hybrid-Mixed-Basis FFT method?

2) The FFTs used in the empirical tests were the standard finite field methods (cf. [Pol 71]). Recently, Mersenne and Fermat Transforms which replace multiplication by powers of w with clever shifting, have been proposed. Agrarwal and Burrus [Agr 74] claim that such methods are faster than FFT method by a factor of 3. The techniques used in the Hybrid and Mixed Basis methods are not senstitive to the kind of transform employed, and so could be used in conjunction with such methods. Would this yield a better algorithm?

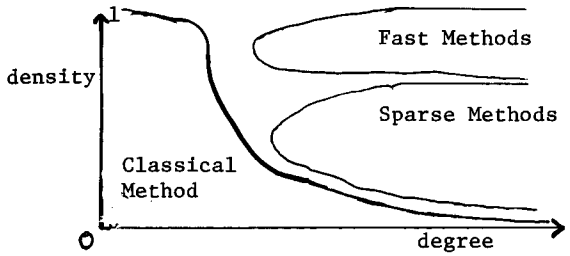
3) The products considered in this paper have all been balanced in the following sense: for a product of polynomials of degrees n and m: $n < m < 2n$. If they are unbalanced i.e. $n << m$, then usually one segments the polynomial of degree m into a number of "chunks" each of approximate degree n, and forms the product of these with the polynomials of degree n. What is the best way to do this?

4) Last, but not least, comes the question of sparse polynomials. These have been completely ignored in the present work because the dense case has enough problems of its own. The classical algorithm can perform as $O(t^3)$ when multiplying polynomials with t terms. Johnson [Joh 74] and Horowitz [Hor 74] have proposed $O(t^2 \log t)$ algorithms and Gustavson and Yun have an $O(t^2)$ algorithm for performing unordered sparse polynomial multiplication.

Current opinion holds that fast techniques are hopeless for sparse polynomials. Is this in fact true? The Kronecker Trick maps a dense multivariate problem onto a sparse univariate one. For example the bivariate case where half the terms are zero. In a sense, the classical algorithm knows that the problem is sparse, and yet it still looses to the FFT techniques.

One might define the density of a polynomial

as the fraction of non-zero coefficients, when the polynomial is viewed in its dense form. The best multiplication algorithm to use, is a function of the density and the degree of the result. The optimum regions for the various methods are probably as shown below.



At the moment a best all-round algorithm for polynomial multiplication is probably a poly-algorithm which would examine the density and degree of the result and choose an appropriate method to compute the product. In order to generate such a poly-algorithm, the boundaries between these optimum regions must be known. Exactly where the boundaries between the methods lie is another question for further research.

Acknowledgement

The author would like to thank Professor J. Lipson for introducing him to FFT techniques.

References

Agr 74 Agarwal R.C. and Burrus C.S.: Number Theoretic Transforms to Implement Fast Digital Convolution; IBM Research Report RC5025, 40 pp, (1974).

Aho 74 Aho A., Hopcroft, J., Ullman J.: The Design and Analysis of Computer Algorithms; Addison-Wesley, Reading, Mass (1974).

Bor 74 Borodin A. and Moenck R.: Fast Modular Transforms; J. Computer and System Sciences, Vol 18, pp 366-387 (1974).

Coo 65 Cooley J.W. and Tukey J.W.: An Algorithm for Machine Calculation of Complex Fourier Series; Math. Comp., Vol 19, pp 297-301 (1965).

Fat 74 Fateman R.J.: Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments; SIAM J. of Computing, Vol 3, pp 196-213 (1974).

Gen 66 Gentleman W.M. and Sande G.: Fast Fourier Transforms - For Fun and Profit; Proc AFIPS 1966 FJCC, Vol 29, pp 563-578.

Gus 76 Gustavson F. and Yun D.: Arithmetic Complexity of Unordered Sparse Polynomials; preprint.

Hal 71 Hall A.D.: The Altran System for Rational Function Manipulation - A Survey; CACM, Vol 14, pp 517-521 (1971)

Hor 75 Horowitz E.: A Sorting Algorithm for Polynomial Multiplication; JACM, Vol 22, No. 4, pp 450-462 (Oct 1975).

Joh 74 Johnson S.C.: Sparse Polynomial Arithmetic; Proc EuroSam 74 Conf, pp 63-71 (1974).

Kanada E. and Goto E.: A Hashing Method for Fast Polynomial Multiplication.

Kar 62 Karatsuba A.: Doklady Akademia Nauk SSSR, Vol 145, pp 293-294 (1962).

Knu 69 Knuth D.: Seminumerical Algorithms; Addison-Wesley, Reading, Mass (1962).

Lip 71 Lipson J.D.: Chinese Remainder and Interpolation-Algorithms; In: Petrick S.R. (ed.) Proc. 2nd Symp. on Symbolic and Algebraic Manipulation, New York, ACM, pp 188-194 (1971).

Moe 73 Moenck R.: Studies in Fast Algebraic Algorithms; University of Toronto, Ph.D. Thesis, Sept 1973.

Moe 74 Moenck R.: On the Efficiency of Algorithm for Polynomial Factoring; to appear in Math. Comp.

Moe 75 Moenck R.: Another Polynomial Homomorphism; to appear in Acta Informatica.

Pol 71 Pollard J.M.: The Fast Fourier Transform in a Finite Field; Math. Comp., Vol 25, No. 114, pp 365-374 (April 1971).

Sin 67 Singleton R.G.: On Computing the Fast Fourier Transform; CACM, Vol 10, No. 10, pp 647-654 (Oct 1967).

Str 73 Strassen V.: Die Berechnungs Komplexitaete elementar symetrischen Funktionen und von interpolations Koeffizienten; Numerische Math., Vol. 20, pp 238-251 (1973).

	Classical Method		FFT Method	
N Deg + 1	2N - 2N + 1	Empirical Secs./60	Theoretical 9N log N + 19N	Empirical Secs./60
2		0.01		0.12
4		0.03		0.16
8		0.08		0.3
16	481	0.29	880	0.55
32	1985	1.10	2048	1.05
64	8055	4.25	4672	2.3
128		17		5.0
256		66		11

Table 1

A comparison of the classical vs. the FFT method of univariate polynomial multiplication. Times are taken for an Algol-W code running on an IBM 360-75 for multiplication of polynomials with coefficients in the finite field GF(40961). Note that both the theoretical and empirical times indicate a cross-over point at degree 31.

Deg	Classical	KPM-0	KPM-1	KPM-3	KPM-7	FFT
3	0.7	3	1.1	0.8		-
7	2	8	4	3	2	8
15	8	30	15	10	8	20
31	38	91	50	30	27	42
63	113	-	150	100	90	97

Table II

A table comparing empirical times in milli-seconds for classical multiplication with a class of Karatsuba algorithms. The times were obtained from an Algol-W code running on an IBM 360-75. KPM-n indicates the basis of degree n multiplication was performed classically.

The times for an FFT algorithm operating on polynomials of the same degree are included for contrast. Note that the classical method and Karatsuba's algorithm were carried out over the integers. This accounts for the discrepancy with Table I. The FFT times are for computation in a finite field. Each finite field operation involves an extra division to compute the residue of the result of an integer operation.

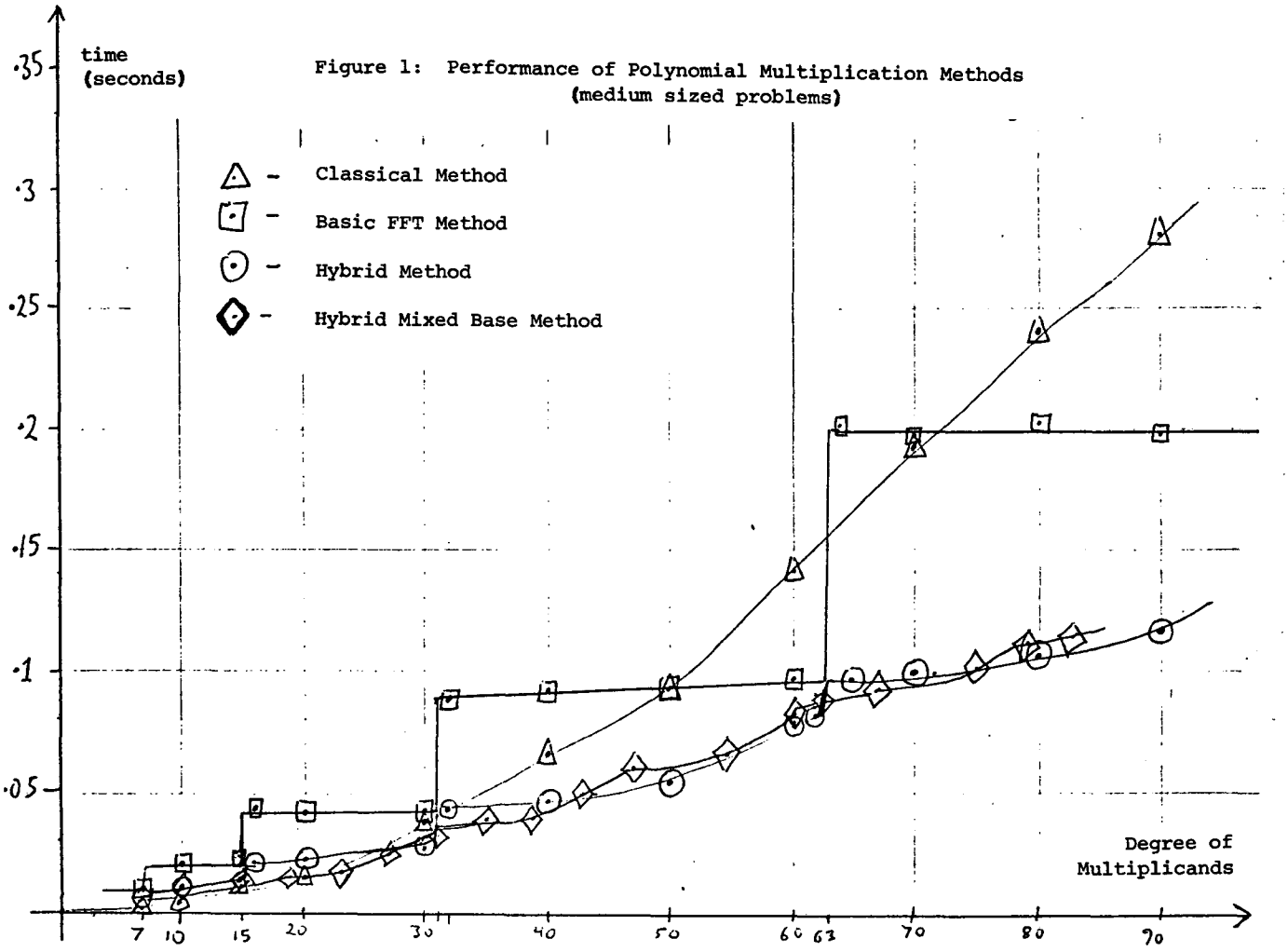


Figure 1: Performance of Polynomial Multiplication Methods (medium sized problems)

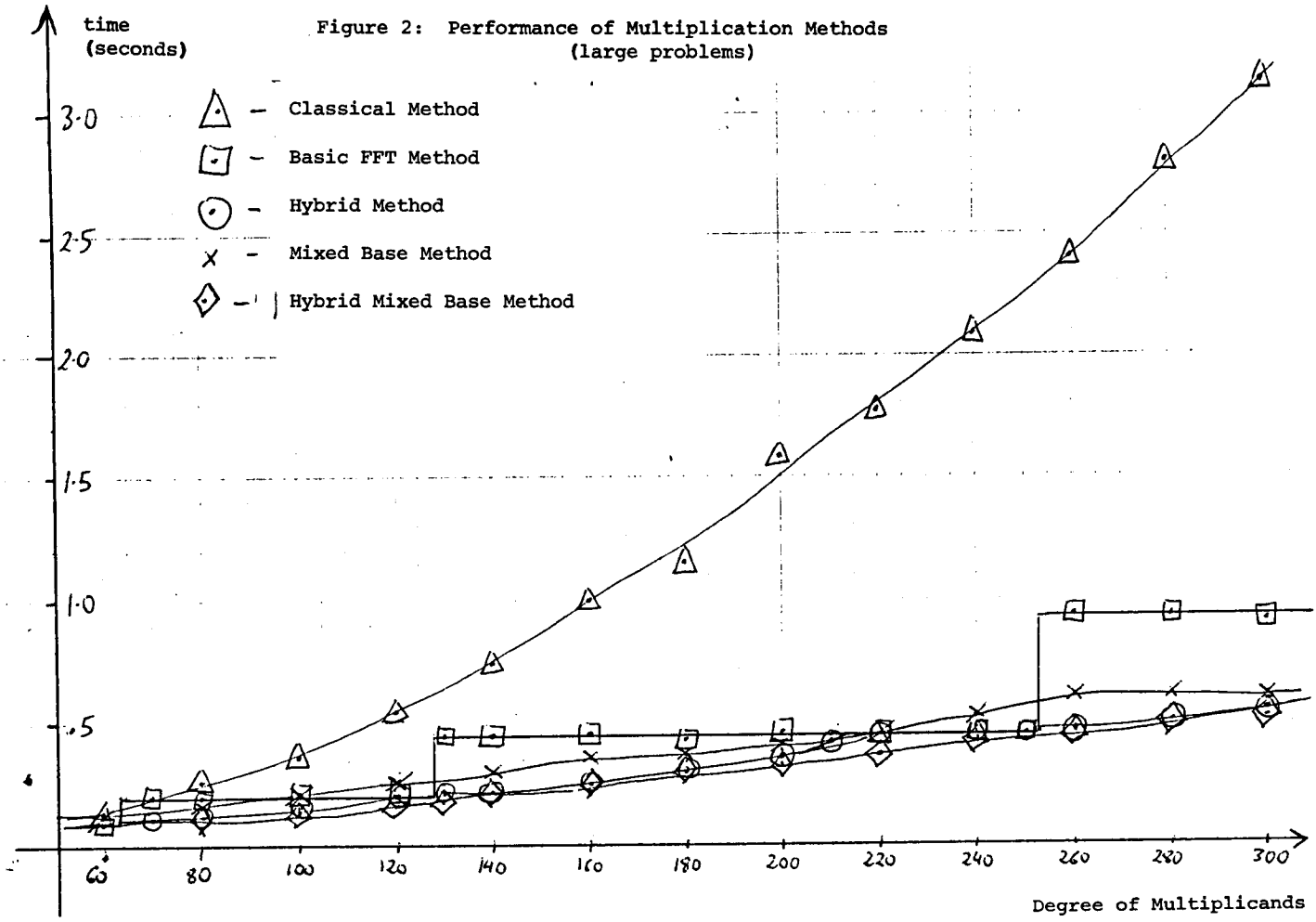


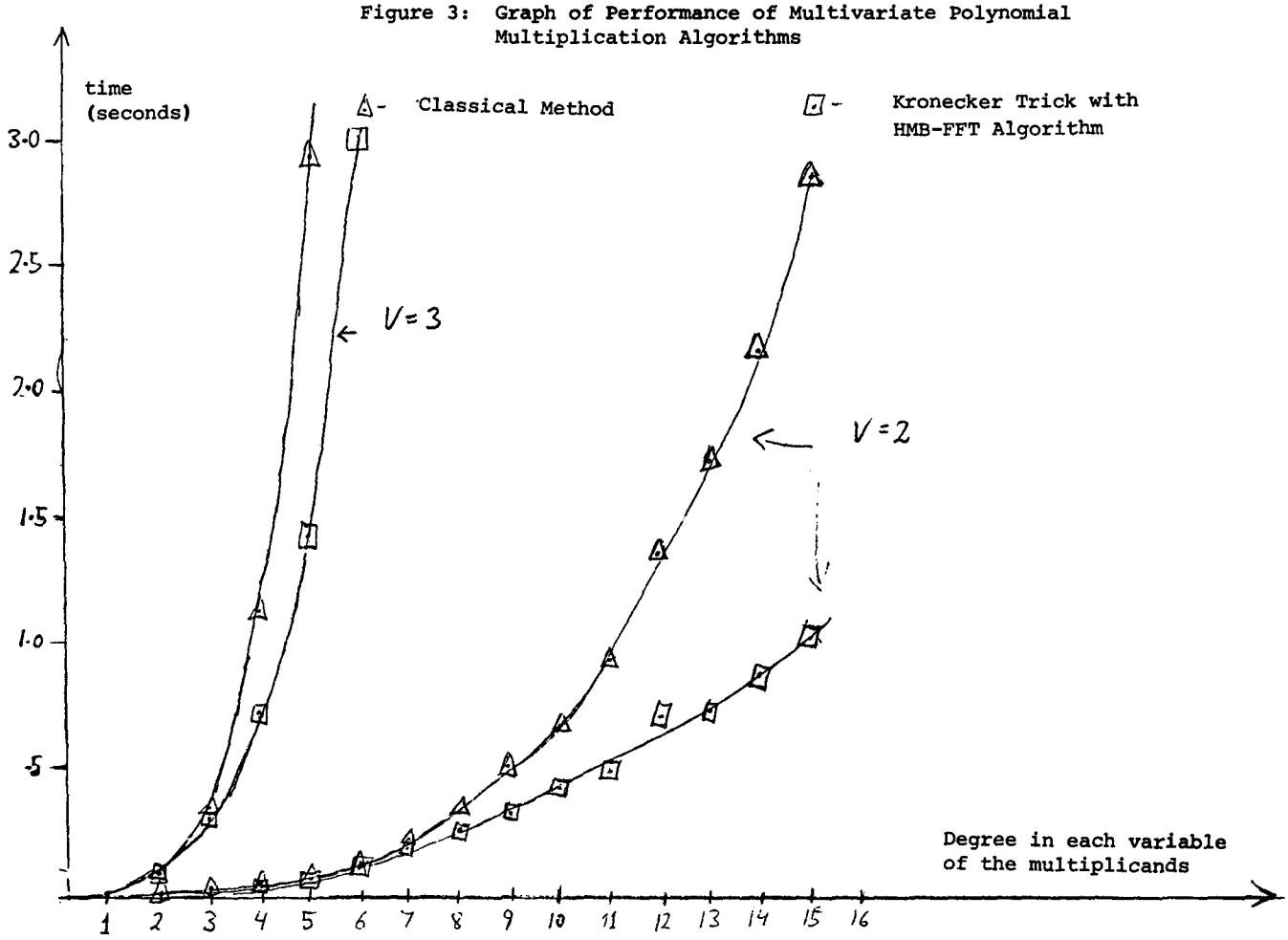
Figure 2: Performance of Multiplication Methods (large problems)

N	Classical	Basic FFT	Hybrid	Mixed Basis	HMB-FFT
10	5.5	22	15	16	11
20	16.6	44	28.3	33	22
30	33	44	35	55	33
40	66	94	62	72	50
50	94	88	68	88	61
60	139	94	88	111	83
70	194	205	117	133	94
80	250	200	125	166	111
90	294	206	137	166	139
100	377	206	163	200	150
150	866	450	255	294	250
200	1600	455	412	444	350
250	2238	433	530	511	472
300	3128	906	583	606	572

Table III

This table contrasts the run-times for the Classical, Basic FFT, Hybrid, Mixed Basis and Hybrid Mixed Basis univariate polynomial multiplication algorithms. The times (in milli-seconds) are taken from an Algol-W code running on an IBM 360-75.

Figure 3: Graph of Performance of Multivariate Polynomial Multiplication Algorithms



var v	de N	Classical steps	time	Kronecker steps	time	FFT steps
2	2	137	8.33	880	13.3	1192
	4	1169	43.3	4670	50	7264
	6	4633	135	10496	130	7712
	8	12833	345	23296	257	47392
	10	28841	696	23296	405	48480
	12	56497	1333	51200	650	49568
	14	100409	2200	51200	850	50656
	16	165953		111616		328960
	18	259273		111616		331520
	20	387281		111616		334080
3	2	1333	80	4672	85	10328
	4	30521	1166	51200	727	121824
	6	233101	6780	241664	3012	134368
	8	1057969		520192		1560608
	10	35333861		11114112		1617184
4	2	12497		51200		85000
	4	774689		520192		1977184
	6	11501041		2375680		2226720
5	2	114937		241664		687128
	4	19472201		5046272		31774944
6	2	1047257		1114112		5518408
	4	487749809		9961472		509099104

Table IV

A table comparing the number of steps and the run times (in milli-seconds) of the Recursive Classical, Kronecker Trick and Recursive FFT algorithms for multivariate polynomial multiplication.