MIT/LCS/TM-34

POLYNOMIAL EXPONENTIATION: THE FAST FOURIER TRANSFORM REVISITED

Richard J. Bonneau

June 1973

MAC TECHNICAL MEMORANDUM 34

POLYNOMIAL EXPONENTIATION:
THE FAST FOURIER TRANSFORM REVISITED

Richard J. Bonneau

June 1973

This research is supported in part by the Raytheon Advanced Degree Program and by the National Science Foundation under research grant GJ-34671.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

Table Of Contents

Introduction

The Fast Fourier Transform (FFT) is a method proposed for the computation of powers of symbolic multivariate polynomials over the integers. Despite its acknowledged superiority in performing polynomial exponentiation, the FFT has been labelled as inefficient for practical systems. This report presents concrete evidence to support the claim that the FFT method is a highly efficient algorithm for the practical computation of the powers of polynomials.

The report proceeds by defining the Discrete Fourier Transform (DFT) and its inverse, and detailing the relationship between the DFT and the FFT. The convolution property of the FFT is then stated, along with its applications to univariate polynomial multiplication and exponentiation. These applications are then extended to include multivariate polynomials by considering the computation structures in which the FFT may be performed. Several problems concerned with the implementation of the FFT are discussed and solutions are given for the actual system implementation. Finally, conclusions on the efficiency of the FFT algorithm are drawn from timing results obtained from extensive testing of the FFT and other proposed methods.

Section 1

The Discrete Fourier Transform

Definition: Let R be a commutative ring with unity, written as 1, K an integer > 1, and WK an element of R of order K. Then the DISCRETE FOURIER TRANSFORM (DFT) of the K-sequence (a0,a1,...,a(K-1)) is the K-sequence

$$(\hat{a}0,\hat{a}1,\ldots,\hat{a}(K-1))$$

given by the following equations:

$$\hat{a}_j = \sum_{i=0}^{K-1} a_i W_K^{i\,j} \qquad 0 \le j \le K-1 \qquad (1)$$

Definition: Assuming the same conditions as above, and also, that K possesses an inverse in R (i.e. 1/K), then the INVERSE DISCRETE FOURIER TRANSFORM (IDFT) of the K-sequence (a0,a1,...,a(K-1)) is the K-sequence

$$(\hat{a}0,\hat{a}1,\ldots,\hat{a}(K-1))$$

given by the following equations:

$$\hat{a}_j = (1/K) \sum_{i=0}^{K-1} a_i W_K^{-i\,j} \qquad 0 \le j \le K-1 \qquad (2)$$

If we consider the term ai*WK**(-i*j) to be rewritten as the equivalent term

$$ai*WK**((K-i)*j)$$

then we can rewrite the above equation as

$$\hat{a}_j = (1/K) \sum_{i=0}^{K-1} a_{K-i} W_K^{ij} \qquad (2a)$$

If we now define a(K) = a(0), then the inverse DFT can be computed from the DFT by merely "flipping" the input sequence. Here, flipping consists in replacing the i-th term by the (K-i)-th term. Thus the same computational algorithm (for the DFT) can be used to compute both the DFT and the IDFT. As might be expected from the terminology, under the right conditions (see section 3), the two transforms are inverses of each other, and thus provide different representations of K-sequences. Note, also, that the DFT (and the IDFT) are linear transformations from R**K to R**K, since the quantities WK**(i*j) are all "constants" for each application of the DFT. For more information on this approach to the phenomenon of the FFT, see Nicholson [14].

The particular virtue of the DFT in many applications results from the following:

Definition: Let A = (a0,a1,.. .,a(K-1)) and B = (b0,b1,.. .,b(K-1)) be two K-sequences in R. Then the CONVOLUTION OF A AND B, written A*B, is the K-sequence C = (c0,c1,.. .,c(K-1)) where the cj are defined as follows:

$$c_j = \sum_{i=0}^{K-1} a_i \, b_{j-i} \qquad 0 \le j \le K-1 \qquad (3)$$

where bn for n < 0 is defined as b(n+K).

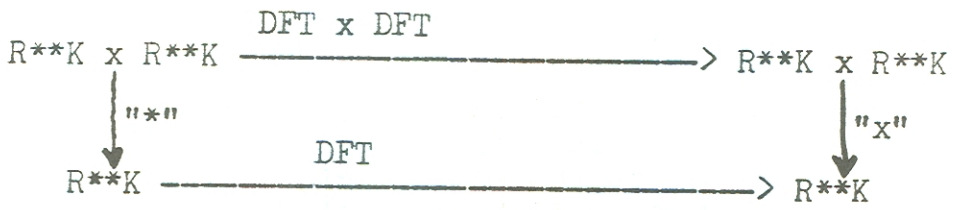
Convolution Property of the DFT:

Let A and B be as in the preceding definition. Let A′ and B′ be the DFT′s of A and B respectively. Then the following equation is true:

$$(A*B)' = A' \times B' \qquad (4)$$

where "x" means component-wise multiplication of K-sequences. In other words, the DFT transforms the convolution operation in R**K

into the component-multiplication in R**K, according to the following commutative diagram:



Proof: $\hat{c}_l = \sum_j c_j \omega_K^{jl} = \sum_j \sum_i a_i b_{j-i} \omega_K^{jl}$

$$= \sum_j \sum_i (a_i \omega_K^{il} \times b_{j-i} \omega_K^{(j-i)l})$$

$$= \sum_i a_i \omega_K^{il} \left[\sum_j b_{j-i} \omega_K^{(j-i)l} \right] = \sum_i a_i \omega_K^{il} \cdot \hat{b}_l$$

$$= \hat{a}_l \cdot \hat{b}_l \quad \text{Q.E.D.}$$

As a result of (4), the convolution of two K-sequences can be computed in the following way:

1) compute the transforms A′ and B′ of A and B respectively.
2) perform componentwise multiplication on A′ and B′ to obtain a K-sequence C′.
3) perform the inverse DFT on C′ to obtain the convolution sequence C.

Thus, the DFT provides a method (though not an intuitive one) for computing the convolution of sequences.

Section 2

The Fast Fourier Transform

In this section, a method for rapid evaluation of the Discrete Fourier Transform is presented. This method is known as the FAST FOURIER TRANSFORM (FFT) and represents a considerable breakthrough in reducing the computational complexity of many problems. The analysis begins with the assumption that the number of points to be transformed (in our case, K) is a composite number; e.g., K is divisible by 2. Then we have the identity:

$$\hat{a}_l = \sum_{i=0}^{K-1} a_i \, WK^{il} = \sum_{i=0}^{1} WK^{il} \sum_{j=0}^{K/2-1} a_{2j+i} \, WK^{2jl} \qquad 0 \le l \le K-1 \quad (5)$$

If we define two K/2-sequences B(0,l) and B(1,l) by

$$b_{i,l} = \sum_{j=0}^{K/2-1} a_{2j+i} \, WK^{2jl} \qquad 0 \le l \le K-1 \; i=0,1 \quad (6)$$

then we have the final relationship

$$\hat{a}_l = \sum_{i=0}^{1} b_{i,l} \, WK^{il} \qquad 0 \le l \le K-1 \quad (7)$$

If we realize that WK**2 is a K/2-th root of unity in (6), then the computation of a K-point FFT involves the following steps:

1) split the input sequence into 2 sequences consisting of the even-numbered and odd-numbered parts of the sequence, corresponding to i = 0 and 1 in (6).
2) perform K/2-point FFT's on the two sequences, using the square of WK as the K/2-th root of unity, yielding B0 and B1.
3) "splice" the results according to equation (7) above.

As a result of this partitioning of the original sequence, the cost of performing a K-point FFT has been reduced to that of computing two K/2 FFT's plus K multiplications for the splicing operations performed in equation 7 above. Now, if K/2 is also even, then we can perform the same splitting operation on the subsequences. This will reduce the problem of computation of one K-point FFT to that of computing 4 K/4-point FFT's plus the splicing costs. For K a power of 2,(i.e., K = 2**k) the ultimate result is that the K-point FFT can be performed using O(K log K) operations (where an operation is either an addition or a multiplication by a power of the K-th root WK). This represents a tremendous savings over the traditional methods which take something like O(K**2).

It should be noted here, that the Fast Fourier Transform approach can be made to speed up the computation of K-point DFT's for any value of K, but the efficiency of the algorithm is maximized for K a large power of a small prime. Note, also, that the same general algorithm can be used for computation of the inverse DFT with the only changes being a "flip" of the input sequence before applying the FFT and a multiplication by the inverse of K after the FFT. (See section 1.)

Finally, some mention should be made concerning alternative schemes for implementing the FFT. The method described above and attributed to Cooley and Tukey [5] is only one of many ways of performing the computation of the DFT. Nicholson in [14] and his thesis describes algebraically the forms wich will perform the correct computations. Included in this analysis is the Cooley-Tukey method and also one described by Good [7]. This computation relies on the fact that K is factorable into a product of relatively prime numbers. As yet, the numerous other techniques have not been investigated thoroughly.

Section 3

Structures Supporting the FFT

The question naturally arises as to which computation structures are able to support the computation of FFT's. The criteria for the support of a K point FFT reduce to the existence of a K-th root of unity, the inverse of K, and the invertibility of the FFT. In a technical memo by this author [2], it has been shown that the following theorem characterizes a class of rings able to support FFT's:

Theorem: Let R be a commutative ring with unity. Let I(R) be the "Integers of R";i.e., the smallest subring of R containing 1. It is known that I(R) is isomorphic to either the integers Z or the integers modulo M (Z/MZ) where M is the characteristic of the ring R. We assume here that the characteristic is finite and non-zero. Then R supports K-point FFT's if the following holds:

K divides p-1 for every prime p which divides M.

As a result of the above theorem, the following rings will support K-point FFT's:

1) (Z/MZ) where M = p = cK+1 some c such that p is prime.
2) (Z/MZ) where M = p**n for any p as in (1),for any n
3) (Z/MZ) where M = product of p**n for any p's as in (1)
4) (Z/MZ)[x1,x2,...,xn] for any M as in 1,2, or 3.
5) GF(p)[x1,x2,...] for any prime p = cK +1

We will make use of several of these rings in our applications of the FFT. Note that the fifth example above yields an answer to the question of when can we use FFT's over finite fields; i.e., when the number of points of the FFT divides p-1. This fact has applications to schemes involving modular arithmetic and the possible use of the FFT, especially for those algorithms requiring the choice of large but otherwise random primes. (See Horowitz [11] for uses of modular algorithms for polynomial exponentiation and Yun [16] for applications to the gcd problem.)

Section 4

FFT and Algebraic Manipulation

Algebraic manipulation systems provide numerous opportunities for application of the FFT. In particular, the area of rational polynomial computations supplies several instances in which the FFT approach can be of considerable benefit. As described by Pollard [15], Fateman [6], and Horowitz [11], the FFT can be invoked to perform symbolic multiplication and exponentiation of multivariate polynomials over the integers or finite fields of the form GF(p) for some prime p. (Also mentioned in Pollard [15], is the possibility of using FFT's to perform exact polynomial division. This application has yet to be studied with regards to the feasibility of its implementation.) Basically, it is because of the convolutional aspect of the operations of multiplication and exponentiation on the coefficients that the FFT is applicable. More particularly, if f(x) is a univariate polynomial of degree m, then it can be represented by a sequence of m+1 coefficients; e.g., (a0,a1,.. .,am), where ai is the coefficient of x**i in f(x). If we wish to square this polynomial, the answer would contain 2m+1 terms of the form:

$$(c0,c1,c2,\ldots,c(2m))$$

where, in fact, the ci are the terms of the convolution of the sequence A with itself where A has been zero-extended to be of length 2m+1. Therefore, to square f(x), the following could be done:

1) form the extended sequence of 2m+1 terms
   (a0,a1,...,am,0,0,...,0)
2) perform a 2m+1-point FFT on the sequence A, obtaining the sequence
   (b0,b1,...,b(2m))
3) square each of the terms in the above sequence
4) perform the 2m+1-inverse FFT to obtain the answer sequence
   (co,c1,...,c(2m))

In order to avoid the use of complex arithmetic and inaccurate computations, it is desirable that the ring to be used as a base for the FFT computation be a finite computation structure. By such a structure, we mean a ring with finite characteristic. Examples of such rings were given in the preceding section. In particular, our system will be based on modular arithmetic schemes, in which we are dealing with the integers modulo some particular integer (the modulus).

In order to guarantee the correctness of the answer sequence, the following restriction is needed on the modulus of the system in which the FFT is to be performed: the modulus M must be greater than twice the absolute value of the maximum coefficient appearing in either a or c.

This restriction is required since operations in modular systems guarantee results only to be congruent to the "actual" answer; i.e., if we compute a form using the integers and then using modular arithmetic, then the latter answer will be only "congruent" to the former. If, however, we can be assured that the "true" answer is in fact less than the modulus, then the modular answer is the "actual answer". The factor of 2 is needed for the bound on M because of the need for a "balanced" representation for the integers modulo M, i.e., we allow positive and negative numbers whose absolute value is less than M/2. We will see more on this subject in section 5 on large coefficients. For now, let us be content with the result that, except for considerations of the modulus, the FFT and inverse FFT can be used to multiply univariate polynomials and raise them to powers. Also, as seen in section 6, the same approach can be utilized to deal with multivariate polynomials.

Thus the FFT's reduce the computation of products of polynomials (or powers of polynomials) to the computation of the products of coefficients (or powers of coefficients).

Section 5

Problem of Large Coefficients

Inasmuch as the modulus to be used for FFT computation must bound the maximum coefficient in either the inputs or outputs (see section 4), we need some estimates on this quantity. Accordingly, we define

Definition: Let f be a multivariate polynomial over the integers. Then the NORM of f (written n(f)) is the sum of the absolute values of all the numerical coefficients in f.

Definition: Let f be a multivariate polynomial over the integers. Then the MAXIMUM COEFFICIENT of f (written maxcoeff(f)) is the largest numerical coefficient of any term of f, regardless of sign.

Note that n(f) $\geq$ maximum coefficient in f. Also, n(f*g) $\leq$ n(f)*n(g).

The following result provides a minimum bound for the choice of moduli for FFT.

Proposition: The maximum coefficient of f**n , where f is a r-variate polynomial with maximal degree m, is less than or equal to the minimum of

$$(n(f))^{**}n \quad \text{and}$$

$$\text{maxcoeff}(f)^{**}n * (\text{maxcoeff}((1+x+\ldots+x^{**}m)^{**}n))\ ^{**}r$$

Inasmuch as the second term is difficult to compute (quickly), the first term has been utilized as a bound for determination of the modulus. However, there may be indications that the second term is significantly smaller than the first for many cases. See section 10 for more details on possible further work on this topic.

As a result of the above proposition, the minimum modulus for a given problem may be quite large and may well exceed the precision of a single computer word. There are two methods which can be used to handle this situation:

1) invoke an arbitrary precision integer arithmetic package, which is found on many systems today.
2) perform the DFT on the inputs for a set of moduli consisting of several single-precision primes pi satisfying the inequality that the product of all the single-precision primes pi is greater than twice the minimum modulus. Once all the answers have been obtained, use the Chinese Remainder Algorithm to re-assemble the final answer from the pi-residues.

The ultimate trade-off between these two methods is the relation of cost of performing a single FFT using multiple-precision arithmetic versus the cost of performing several FFT's in single precision plus the cost of reconstructing numbers from residues. From our results to date, the second approach appears to be much more advantageous for large problems. See section 9 for results actually produced by our experimental system and for conclusions based on these results.

Section 6

Multivariate Polynomials

The application of the FFT to univariate polynomial problems is quite natural as to each polynomial we can assign a sequence consisting of its coefficients. Then the computations to be performed on the polynomials can be performed by suitable manipulation on the coefficient sequences. When one considers the problem of dealing with multivariate polynomials, there are several ways of proceeding. We will discuss here three methods which might be used for utilizing FFT's with multivariate polynomials.

1) Multi-dimensional FFT: Similar to the univariate case, to each multivariate polynomial we can assign an r-dimensional array consisting of the numerical coefficients of the polynomial (where r = number of variables). On this array we can perform an r-dimensional FFT which , intuitively, consists in performing iterative one-dimensional FFT's on consecutive rows, faces, etc. of the coefficient array. Some programming difficulties inherent in this method have prevented the author from implementing it in this study.

2) Non-recursive FFT: In this method, we use polynomial arithmetic with the one-dimensional FFT, where the elements to be transformed are the coefficients of the polynomial in some fixed (but otherwise) arbitrary "main" variable. In other words, we rewrite a polynomial f as follows:

$$f(x,y,z,\ldots) = \sum_i a_i x^i ,$$

where, now, the ai are polynomials in y,z,.. . . Then another method for computing the products or powers of the transformed coefficients is used before taking the inverse FFT, again using polynomial arithmetic. Note that in this system, the only polynomial arithmetic required is that of polynomial addition and multiplication of a polynomial by a constant (plus the "other" method for multiplication or powering). This method may cause some reduction of the time for computation, but is inherently dependent on the characteristics of the other algorithm which is used on the lower level problem. For this reason, this method was not implemented, but rather emphasis was concentrated on the following, "pure" FFT method.

3) Recursive FFT: In this method, we proceed as in the method described above but when it is necessary to perform computations using the transformed (polynomial) coefficients, we now recursively call the FFT method. For example, in the exponentiation of polynomials in two variables, the flow of the computation would be as follows:

1) Perform an FFT on the single-dimension array consisting of the (polynomial) coefficients of the polynomial in one of the two variables. The resulting sequence will have elements which are univariate polynomials.
2) To raise each of these coefficients to a power, call the FFT method which will operate on the numerical coefficients of these polynomials, returning the powers of the polynomials.
3) Perform the inverse FFT on these polynomial coefficients, yielding the (polynomial) coefficients of the "main" variable of the desired power of the given polynomial.

There are some problems inherent in the recursive aspects of this method including the optimal way of reducing the problem to a smaller problem, choice of modulus, use of K-th roots of unity, etc. In the implemented system, some assumptions are made as to the structure of the polynomials being dealt with. (See section 9) However, the method can be used in the general case, with a small amount of added expense.

Section 7

Timing Analysis for the FFT

The purpose of this section is to present a detailed timing analysis of the FFT method for polynomial exponentiation. This timing analysis will be derived in terms of the number of integer operations of multiplication required.

In order to make the derivation tractable and readable, we assume that the polynomial we are dealing with, f(x1,.. .,xr), is dense in all of its r variables and of degree d in each variable. As a result, f has (d+1)**r terms. Letting K = nd+1, we also assume that the cost of performing a K-point FFT is given by K log K "coefficient operations." By a coefficient operation, we mean either an addition of sequence elements, or multiplication of a sequence element by a "constant"; i.e., an integer. Because of the recursive nature of the application of the FFT, the sequence elements can be integers or polynomials. In either case, however, the cost of the coefficient operation is proportional to the number of terms in the coefficient. Thus, for example, the cost of performing a K-point FFT on a sequence of r-1-variate polynomials, all having common degree d in each variable, is given by K log K * (d+1)**(r-1) integer operations. Also, we make the additional assumption that the Chinese Remainder Algorithm is not invoked and thus only one prime is needed.

Having made the above assumptions, we now proceed to compute the costs of the algorithm as given by the steps below:

Inputs: f: an r-variate, dense polynomial of common degree d
n: an integer > 1.
Output: f**n.

1) Writing f as
form a sequence of length K from the r-1-variate polynomial coefficients of f by extending the coefficients with zeros.
2) Apply a K-point FFT to this sequence, yielding the sequence
3) Raise each transformed element to the n-th power by calling this routine recursively.
4) Apply the inverse FFT to this new sequence of r-1-variate polynomials, all of common degree nd in each variable.
5) The coefficients of f**n are precisely the elements in the sequence computed in step 4.

In order to terminate the recursion, we insert the following step:

0) If r = 0, compute f**n by repeated squaring and return.

Let M(r) = cost of performing polynomial exponentiation on r-variate polynomials of common degree d. Then, we can write

$$M(r) = K \log K * (d+1)^{**}(r-1) + k * M(r-1) + K \log K * (nd+1)^{**}(r-1)$$

Recalling that K was chosen as nd+1, the cost now becomes

$$M(r) \leq 2\ K^{**}r \log K + K * M(r-1)$$

At the i-th level of recursion, we can show that

$$M(r) \leq 2\ i\ K^{**}r \log K + K^{**}i * M(r-i)$$

Finally, at the last level of recursion (i=r), we execute step 0, which costs O(log n) operations. Thus, we obtain

$$M(r) \leq 2\ r\ K^{**}r \log K + K^{**}r * \log n,$$

which can be rewritten as

$$M(r) = O(K^{**}r \log (n\ K^{**}r)).$$

Finally, replacing K by nd+1 and simplifying the expression, we obtain the final result for the cost of the FFT method:

$$M(r) = O(\ (nd)^{**}r * \log (n * (nd)^{**}r)).$$

This compares quite favorably with the other methods proposed such as repeated multiplication, repeated squaring, binomial expansion, an evaluation-interpolation method. (See section 9 for a brief description of these methods and consult Fateman [6] and Horowitz [11] for more complete details on the computing time for all the methods). It is the contention of the author that the "difficulty of programming" aspects of the FFT method mentioned by both authors cited above, are not as difficult as first anticipated. In fact, a very reasonable implementation of the FFT approach has been developed on the MACSYMA system at MIT [1] and has been used to produce data by which comparisons can be made with the other proposed methods of computing the powers of polynomials. The results of section 9 will demonstrate the efficiency of the FFT over the other methods and that the added costs incurred by the FFT are well worth it in the long run.

Section 8

Implementation Problems

The implementation problems associated with the Fast Fourier Transform are well known (as mentioned by Fateman [5], Horowitz [11] and Borodin & Moenck [3]), but they are not insurmountable as might be inferred from their gloomy statements. Basically, the problems break down into the following categories:

1) finding moduli supporting the FFT
2) given a modulus, determining a primitive root
3) given a primitive root, computing a K-th root of unity
4) computing the inverse of K
5) in the case of the Chinese Remainder Algorithm, precomputing various constants

For ease of programming and demonstration, FFT's were implemented for the cases of K a power of 2 or a power of 3. Another reason for doing so is mentioned in the classic paper of Cooley-Tukey [5], wherein the statement is made that the optimal choice for a basis is 3 with 2 (and 4) next optimal.

The problems mentioned above will be considered one by one.

1. As has been noted in an earlier section, the choice of the modulus is dependent on K, the size of the transform to be performed. Inasmuch as we have only considered K a power of 2 or a power of 3, our problem reduces to the following:

find primes p such that K divides p-1.

Once we have a set of such primes, then (theoretically) we can use as a modulus for the K-point FFT any number whose prime factors lie in the given set of primes. Another consideration in this regard was that of the Chinese Remainder Algorithm and the single-precision/multiple-precision arithmetic trade-off. In the actual system, the six largest single-precision primes were computed for each K (where K went up to 2**8 and 3**8). (See Appendix 1). Actually, in the system, use was made of the fact that if K divides p-1 for some prime p and hence p can be used as a modulus, then it also serves as a modulus for K'-point FFT's for all K' dividing K. In other words, if we choose large enough K and then choose the p as a function of this large K, we only need one prime. This is precisely what has been done in the system. The six largest single precision primes were chosen such that they could support both 512 and 729-point FFT's. As a result, with these primes, FFT's can be performed for all K

dividing 512 or 729; i.e., powers of 2 or powers of 3. This use of a single list of primes (and hence of Chinese Remainder Algorithm constants) reduces the storage required and allows use of FFT's for all practical problems. Using these primes, we could perform FFT's for arbitrary minimum modulus by taking powers of the given primes and working in multiple precision arithmetic or for minimum modulus less than sextuple-precision using the 6 primes and the Chinese Remainder Algorithm. In the experience of the implemented system, these bounds are quite practical.

2. Once a modulus has been computed, a K-th root must be computed for this modulus. Inasmuch as we have produced moduli which are primes, the problem reduces to that of finding a primitive root in the modulus. In order to facilitate the use of this primitive root for powers of the prime as modulus, it is necessary to find a special primitive root. The criterion for this root is:

Definition: A primitive root g modulo p is a SPECIAL PRIMITIVE ROOT iff g is a primitive root modulo p**n for all n iff

$$g^{**}(p-1) \text{ not congruent to } 1 \text{ modulo } p^{**}2.$$

(See Grosswald, Chapter 4, Exercise 32 [8].) Based on empirical evidence obtained from many test cases, it is the author's conjecture that the smallest primitive root modulo any prime is also a special primitive root modulo that prime. The author has learned of several counterexamples to the conjecture above, but has verified that all of the primes used in the system satisfied the conjecture. Thus, in the system implementation, for each prime modulus computed from problem 1 discussed above, the smallest primitive root was also computed.

3. Now that we have computed a special primitive root for each modulus which might be used in the system, what is needed is the K-th root of unity in that modulus. This is obtained from the special primitive root g modulo p by computing:

$$WK = g^{**}(p-1)/K \quad \text{modulo } p.$$

If we desire the K-th root modulo a power of p (p**n), then the formula becomes:

$$WK = g^{**}((p-1)p^{**}(n-1))/K \quad \text{modulo } p^{**}n.$$

Again, there has been much empirical evidence pointing to the conjecture that the K-th root modulo p**n can be computed directly from a K-th root modulo p by the following:

$$WK^{**}(p^{**}(n-1)) \quad (\text{modulo } p^{**}n)$$

where WK is the K-th root modulo p. Again, it can be shown that if the underlying prime p satisfies the condition given in the above discussion, then for $n<p$, the above conjecture concerning the computation of K-th roots modulo p**n is valid. Inasmuch as all the primes used by the system did indeed satisfy the original conjecture, the computation of K-th roots was accurate. As a result, it was only necessary to compute K-th roots modulo the primes in the system. A list of such K-th roots were computed for the moduli used in the system and were made available to the FFT's.

4. The inverse of K is needed for computation of the inverse FFT, and hence was computed for the moduli and the K's used in the system. The value of the inverse of K modulo p is computed from

$$\text{"1/K"} = K^{**}(p-2) \quad \text{modulo } p.$$

For a power of p (e.g. p**n), the following formula is used

$$\text{"1/K"} = K^{**}((p-1)*p^{**}(n-1)-1) \quad \text{modulo } p^{**}n.$$

Again, these values were computed for all the cases considered above. This method for computing inverses is included in the paper by Collins [4].

5. In order to use the Chinese Remainder Algorithm (CRA), it is convenient to compute certain constants beforehand. In particular, the CRA algorithm utilizes the following computational formula:

$$a = a1 + (a2 - a1) * (p^{**}(-1) \bmod q) * p \quad (\text{mod } p*q)$$

for finding the unique (up to modulus p*q) value of a satisfying $a = a1 \pmod p$ and $a = a2 \pmod q$. Thus, in order to apply the CRA to our circumstances, it was advantageous to compute the constants (p**(-1) mod q) * p (mod p*q) for the possible cases in which the CRA was to be applied. In fact, this was done for the case of the primes computed above for the support of the FFT. A table of such values is in Appendix 1.

Section 9

Testing Results and Conclusions

The goal of this section is to present the results of exhaustive testing of the FFT methods of exponentiation and to provide conclusions based on these results. Before these tasks can be performed, however, some mention should be made about certain system assumptions, the testing environment, and some of the other methods used in the tests. The following paragraphs will be concerned with these topics.

SYSTEM ASSUMPTIONS: The system assumptions can be broken down into two classes: those concerned with the implementation of the FFT itself, and those concerned with the overall system in which the tests were performed. Two of the assumptions from the former class have already been mentioned in section 8 on implementation problems. In particular, the methods used for computing primitive roots of unity and K-th roots of unity are based on conjectures put forth in parts 2 and 3 of that section. Next, it should be mentioned that, although it is theoretically possible to program the FFT for the number of points being the power of any prime, it was only feasible in the testing to program two versions of the FFT; one for the number of points being a power of 2 and one for powers of 3. As mentioned previously, these prime numbers are considered to be optimal by Cooley and Tukey [5]. The result of this decision is that the running times for the FFT's follow a pattern which is very much like a step-function. Finally, two versions of each of these FFT's were programmed; one using multiple-precision arithmetic for large coefficients, and the other using a Chinese Remainder Algorithm approach. See section 5 for more details on this topic.

The system assumptions which were independent of the FFT itself included the following: all the methods tested were timed on the basis of CPU time elapsed and did not include the time spent in "garbage-collection", all the methods were given exactly the same inputs and were required to produce identical outputs (i.e.; time for "translation" from one polynomial representation to another was included in the timing), the choice of moduli for the modular algorithms was based on the largest prime numbers which could fit into a single computer word, which is 35 bits plus sign bit on the DEC PDP-10.

TEST ENVIRONMENT: The test environment was the MACSYMA system of Project MAC located at MIT. See Bogen [1] for more detail on the actual system. The test code was written in the LISP programming

language and was compiled prior to its execution in the tests. The rational polynomial package, developed primarily by R. Fateman, provided the author with much of the software necessary to deal easily with multivariate polynomials. This fact made the efficient programming of the FFT much simpler.

METHODS TESTED: The actual tests performed for polynomial exponentiation included nine different algorithms. The FFT methods were the power of 2, multiple precision (FFT2); the power of 2, Chinese Remainder Algorithm (CRAFFT2); the power of 3, multiple-precision (FFT3); and the power of 3, Chinese Remainder Algorithm (CRAFFT3). The remaining 5 algorithms were devised by several people including Horowitz and Fateman, and the code for all these algorithms was coded by Fateman for the MACSYMA system. It should be noted that all of these algorithms have been closely studied by Fateman [6] with respect to asymptotic behavior and running "cost"; i.e., number of numerical coefficient operations performed during the exectution of each algorithm. On this basis, the FFT is the best in theory for the general problem of polynomial exponentiation, but it is of interest to see at what point the FFT algorithms take over in practice.

At this point, it would be beneficial to briefly review the other methods tested. The reader is warned that these descriptions have been made with respect to the salient aspects of each algorithm, and do not mention many details which may have significant bearing on the actual running times. For more details, the reader is directed to the references mentioned for each method.

Method 1: RMUL This algorithm computes a power of a polynomial by repeated multiplication by the polynomial. [6, 8, 10]

Method 2: RSQ This algorithm uses the binary decomposition of the exponent to form the correct sequence of multiplications and squarings to obtain the power of the polynomial. [6, 8, 10]

Method 3: BINOM This method invokes a property of the binomial expansion to compute powers of polynomials. [6]

Method 4: SUMS This method computes powers of a polynomial by computing successive coefficients using recurrence relations involving previously computed coefficients. [13, p. 445]

Method 5: MOD This method utilizes a scheme involving modular arithmetic, polynomial evaluation and polynomial interpolation in order to find the powers of a polynomial. [11]

The testing itself proceeded in the following fashion: the user chose the number of variables, the degree of the polynomial in each variable, the maximum size of the coefficients and a range of powers to be computed. The system then produces a random polynomial of the correct specifications, applies each algorithm to this polynomial, computes the run time for each algorithm, and then prints the results. The results were given in the following form: the minimum time was determined, then divided into all the run times yielding normalized run times. These normalized times were printed in tabular form, along with the best time in milliseconds. Many different tests were performed in order to judge the performance of the various methods under different circumstances such as large polynomials, high powers, multivariate polynomials, and coefficient size. The results of this testing are presented in Appendix 2. In the discussion below, we will refer to the numbered charts in the Appendix.

CHARTS 1,2: We can see from Charts 1 and 2 for polynomials of degree 30 and 50 respectively that the FFT methods quickly become the most efficient methods for exponentiation, at least for the univariate case. Also, it should be noted here that similar tests with use of the multiple precision version of the FFT revealed it to be much less efficient than the Chinese Remainder Algorithm version. For this reason, the timing results for this version is not included in this appendix.

CHART 3: We now turn to the problem of coefficient growth. In Chart 3 we have tested the various methods on polynomials of the same size but with the numerical coefficients growing as indicated by the first column "COEFF". Here we see that the FFT routines are best when the coefficients are small but as they grow, the maximum possible coefficient in the answer also grows, requiring either large multiple-precision arithmetic or repeated applications of the FFT followed by applications of the Chinese Remainder Algorithm. Thus, by the time the coefficients are of the size of 10^9, the 2 FFT methods are losing quite badly to all of the other algorithms. Thus, we conclude that the presence of large coefficients in the input polynomial degrades the overall performance of the FFT.

CHARTS 4,5: The next charts are devoted to the results of testing the algorithms on multivariate polynomials. The rapid growth of the size of powers of multivariate polynomials places limitations on the practical aspects of testing. As we can see from Chart 4, the bivariate polynomials of degree 5 exhibit the efficiency of the FFT methods. This is due primarily to the fact that for a bivariate polynomial of degree 5 in both variables, the transformed sequence of coefficients has univariate polynomials of degree 6, each of which must be raised to a power. Reverting back to charts 1 and 2, it is seen that the larger

these polynomials are, the more efficient is the FFT algorithms. We can thus conclude that for "larger" problems in the bivariate cases, the FFT is the best method. The case for tri-variate polynomials is not as easy to assess, as the testing was not very extensive. We can see from Chart 5 that the FFT appears to be coming to the fore as far as the trivariate polynomials are concerned. However, the lack of more evidence leaves the conclusions concerning trivariate polynomials in doubt.

CHART 6: For the case of high powers of a polynomial we once again run into the problem of large coefficients and hence the problems of multiple-precision versus CRA FFT's. In Chart 6, we see that the FFT methods not faring very well. One reason is the overhead incurred by multiple-precision arithmetic or Chinese Remainder Algorithm. Another reason is that the SUMS algorithm has been shown to be a linear time algorithm in the power of the polynomial for univariate polynomials.(See Fateman [6].) As a result, we see that the FFT does not appear to be a winner for this particular problem. Again, practical constraints prevented any testing of the case of multivariate polynomials for high powers. Note, however, that most of the other algorithms are performing quite poorly in relation to SUMS. The next aspect of the problem to be considered is that of the relative densenes of the input polynomials. It can easily be shown that the FFT method does not take advantage of the density of the polynomial, as for example, the same FFT cost are incurred for the polynomial x**7 as for any dense polynomial of degree 7. Since most of the other methods rely somewhat on the actual non-zero coefficients of the polynomial, it is safe to conclude that they will run more quickly on sparse polynomials. There are some "tricks" which can be added into the FFT algorithm to take advantage of zero coefficients, but the effect of these tests on the overall run time has not yet been fully analyzed. Thus, it may be concluded that the FFT methods profit more from the greater density of polynomials than the other methods.

From the above analysis and the data referred to in the appendix, we can draw several conclusions concerning the types of problems on which the FFT methods appear to perform the best:

1) High degree univariate polynomials
2) Bivariate and trivariate polynomials.
3) Polynomials with small numerical coefficients.
4) Dense polynomials.

The author has performed extensive tests on smaller univariate polynomials, bivariate and trivariate polynomials and has found that the above conclusions apply to these test results as well as those presented in Appendix 2.

Section 10

Future Work

There is still room for much work in this area in that many alternative solutions might be used in place of those used in the system described in this paper. Most prominently, the use of arrays instead of lists for FFT computations might permit an implementation of multi-dimensional FFT's for multivariable polynomials. This method would avoid the "recursive" FFT method. However, there are numerous programming and theoretical obstacles to overcome in this approach to the problem.

Another possible area for future work includes the possibility of using alternative schemes for the actual programming of the FFT. The most well-known alternate is the Good algorithm which enables the efficient programming of a K-point DFT where K factors into a product of relatively prime factors(see [7]). This method is most efficient for those values of K where the Cooley-Tukey method is least efficient. It is my hypothesis that some combination of the two would be optimal for all computations.

Inasmuch as the main failing of the FFT occurs due to the problem of coefficient growth and use of the Chinese Remainder Algorithm, it would seem most advantageous to obtain tighter bounds on the maximum coefficient as discussed in section 5. Thus, some work could profitably be done on the problem of determining more closely the minimum modulus required for the correct operation of the FFT.

The results obtained thus far in the study once again renew the original optimism vis-a-vis the computational efficiency of the FFT. As a result, it may now be feasible to use the FFT for the multiplication of multivariate polynomials, as suggested in Pollard [15]. Again, the main reason it was rejected there was its implementation problems. However, it is my belief that the FFT can be effectively used for polynomial multiplication for cases when the polynomials are large. Work has begun on verifying this conjecture.

Finally, it is conceivable that there are applications of the FFT to other computation structures currently under investigation in the area of symbolic and algebraic manipulation. Such structures include polynomial rings, Galois fields, algebraic number fields. Inasmuch as these types of structures may very well support FFT computations, it might be desirable to investigate the ramifications of the efficiency of the FFT on computations to be done in these structures.

As a final note, it should be noted that while the FFT methods may not be the best for all the types of problems tested, it is usually second or third, and that the algorithms beating them are not the same. For example, SUMS is best for high powers of polynomials, but far from best for bivariate polynomials, while BINOM is best for trivariate polynomials, but extremely poor for high powers. Note that in the cases mentioned above, FFT methods were very close to best. In this sense, it can be stated that the FFT method for multivariate polynomial exponentiation provides the optimal single algorithm for the widest class of polynomials. However, it should be noted that the optimal method would be to provide several algorithms and a scheme for choosing the appropriate method depending on the characterization of the input polynomial and the power to which it is to be raised.

Appendix 1

Numerical Tables

This appendix contains tables used by the FFT routines while testing of the various methods of polynomial exponentiation was performed. These tables include values for prime moduli which will support the computation of K-point FFT's for K being a power of 2 or of 3, values of K-th roots of unity, the inverse of K used by the inverse FFT, and constants needed by the Chinese Remainder Algorithm. For more information on the computation of these tables see sections 6 and 8.

Implementation Constants for K = k-th Power of 2

k	Prime	K-th Root	Inverse of K
1	34357478401	-1	-17178739200
	34355238913	-1	-17177619456
	34353372673	-1	-17176686336
	34348893697	-1	-17174446848
	34347773953	-1	-17173886976
2	34357478401	9891852171	25768108801
	34355985409	19811282838	25766989057
	34355238913	4927062827	25766429185
	34353372673	4951275475	25765029505
	34348893697	5709113108	25761670273
	34347773953	21761134509	25760830465
3	34357478401	32682092207	30062793601
	34355985409	15131840309	30061487233
	34355238913	33303674156	30060834049
	34353372673	29927889522	30059201089
	34348893697	18569911987	30055281985
	34347773953	24411524499	30054302209
4	34357478401	34308022287	32210136001
	34355985409	3547587927	32208736321
	34355238913	16618834046	32208036481
	34353372673	2310845740	32206286881
	34348893697	9215285218	32202087841
	34347773953	17539867326	32201038081
5	34357478401	25761471905	33283807201
	34355238913	9120821392	33281637697
	34353372673	12655411921	33279829777
	34348893697	259396819	33275490769
	34347773953	7440474820	33274406017
6	34357478401	6646977762	33820642801
	34355985409	34277845374	33819173137
	34355238913	4601325225	33818438305
	34353372673	6024103	33816601225
	34348893697	8823147390	33812192233
	34347773953	31586785258	33811089985

k	Prime	K-th Root	Inverse of K
7	34357478401	22587462373	34089060601
	34355985409	25301099943	34087579273
	34355238913	7616143182	34086838609
	34353372673	31417233872	34084986949
	34348893697	19801476384	34080542965
	34347773953	26520592997	34079431969
8	34357478401	24034500045	34223269501
	34355985409	3243609365	34221782341
	34355238913	5540034754	34221038761
	34353372673	32127020682	34219179811
	34348893697	12027032227	34214718331
	34347773953	30116982508	34213602961
9	34357478401	27916816253	34290373951
	34355985409	14736327539	34288883875
	34355238913	33132971988	34288138837
	34353372673	16000368730	34286276242
	34348893697	25770368986	34281806014
	34347773953	15051603591	34280688457

Implementation Constants for K = k-th Power of 3

k	Prime	K-th Root	Inverse of K
1	34357478401	11038728257	-11452492800
	34355985409	-11530751209	-11451995136
	34355238913	7592415340	-11451746304
	34353372673	-6446882571	-11451124224
	34348893697	-16448427828	-11449631232
	34347773953	-6746631931	-11449257984
2	34357478401	32978710811	30539980801
	34355985409	20601174901	30538653697
	34355238913	23497694187	30537990145
	34353372673	10647751358	30536331265
	34348893697	27497752985	30532349953
	34347773953	14101174432	30531354625
3	34357478401	14476663079	33084979201
	34355985409	10394425823	33083541505
	34355238913	12290146715	33082822657
	34353372673	5927656286	33081025537
	34348893697	31973225865	33076712449
	34347773953	11448793429	33075634177
4	34357478401	5089336041	33933312001
	34355985409	33024784276	33931837441
	34355238913	20939201693	33931100161
	34353372673	14083970947	33929256961
	34348893697	14803753625	33924833281
	34347773953	30762734906	33923727361
5	34357478401	2624266140	34216089601
	34355985409	7987108064	34214602753
	34355238913	11015976334	34213859329
	34353372673	3390565065	34212000769
	34348893697	18664094272	34207540225
	34347773953	12869631580	34206425089
6	34357478401	10402372247	34310348801
	34355985409	15090413300	34308857857
	34355238913	15473845574	34308112385
	34353372673	23779923705	34306248705
	34348893697	770594219	34301775873
	34347773953	15091683908	34300657665

Constants for Chinese Remainder Algorithm

Primes	CRA Constant
34357478401	NIL
34355985409	590191722700280100893
34355238913	-126841084041355193378909267362 87
34353372673	3153799607494996286522316597950528747476 88
34348893697	-139597802660382165286688443662492055965336258690435 91
34347773953	607740804817197067441373401894506068454545491803651359897317521

Appendix 2

Timing Charts

This section contains several charts of timings computed during extensive tests on various methods for polynomial exponentiation. Included in this section are results for large univariate polynomials, bivariate and trivariate polynomials, large numerical coefficients and high powers of polynomials. The charts are arranged so that all the running times are presented as a ratio with respect to the best running time. This best time is then included in order to obtain actual running times, if desired. Note that these times do not include "garbage collection" time, but include all other overhead. Timing was done on a PDP-10 computer with 2 microsecond memory cycle time. Average instruction execution time is about 3 microseconds. See section 9 for discussion of the various methods tested and for conclusions following from these results.

CHART 1: LARGE UNIVARIATE POLYNOMIAL

DEGREE = 30 MAXIMUM COEFFICIENT = 10

POWER	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
2	4.108	1.776	1.0	1.812	1.815	1.295	1.723	690.196
3	2.506	1.964	1.682	2.026	2.028	1.0	2.134	1837.72
4	3.438	3.908	3.626	4.026	3.352	1.0	2.14	1856.856
5	1.062	3.04	1.444	1.555	1.395	1.0	1.024	8033.314
6	1.737	4.532	2.183	2.327	2.006	1.0	1.018	8056.669
7	2.154	5.913	2.936	3.093	2.839	1.0	1.015	8671.838

CHART 2: LARGE UNIVARIATE POLYNOMIAL

DEGREE = 50 MAXIMUM COEFFICIENT = 20

POWER	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
2	4.493	1.932	1.0	1.991	1.992	1.16	2.486	1710.302
3	3.09	2.434	2.063	2.509	2.514	1.0	1.046	4067.266
4	2.3	4.771	2.206	2.455	2.036	1.0	1.041	8487.418
5	4.186	7.96	3.806	4.081	3.668	1.0	3.349	8487.585
6	2.362	5.352	2.807	2.964	2.628	1.0	1.567	19014.763
7	2.693	7.222	4.308	4.487	4.583	1.0	1.545	19756.631
8	1.866	8.751	3.763	3.905	3.339	1.0	1.504	32596.178

CHART 3: COEFFICIENT GROWTH FOR UNIVARIATE POLYNOMIAL

DEGREE = 10 POWER = 10

COEFF	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
10	1.304	1.848	1.728	1.901	1.642	1.0	1.379	760.923
10^2	1.097	2.019	1.0	1.124	1.023	1.242	1.647	1494.588
10^3	1.0	1.877	1.171	1.277	1.232	1.176	1.549	1621.639
10^4	1.0	2.583	1.145	1.261	1.139	1.786	2.29	1804.3
10^5	1.0	2.538	1.147	1.261	1.141	1.739	2.228	1871.39
10^6	1.0	3.209	1.133	1.277	1.167	2.355	2.971	1985.628
10^7	1.0	3.056	1.102	1.233	1.132	2.235	2.814	2118.297
10^8	1.0	3.757	1.12	1.252	1.132	2.916	3.633	2151.532
10^9	1.0	3.484	1.075	1.198	1.094	2.691	3.344	2359.455
10^10	1.0	4.054	1.036	1.155	1.053	3.328	4.091	2432.368

CHART 4: BIVARIATE POLYNOMIAL

DEGREE = 5 MAXIMUM COEFFICIENT = 3

POWER	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
2	4.014	1.626	1.0	1.652	1.651	3.111	9.994	1007.224
3	3.693	1.979	1.586	2.042	2.042	1.0	3.196	3423.502
4	2.403	1.548	1.329	1.6	1.561	1.08	1.0	11836.132
5	4.201	3.023	2.694	3.139	3.093	1.06	1.0	12746.566
6	3.04	4.716	2.151	2.423	2.837	1.0	6.66	29859.483

CHART 5: TRIVARIATE POLYNOMIAL

DEGREE = 2 MAXIMUM COEFFICIENT = 3

POWER	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
2	2.177	1.56	1.0	1.552	1.558	8.976	15.265	632.006
3	1.645	1.316	1.0	1.356	1.357	1.752	3.026	3750.7
4	1.392	1.263	1.0	1.308	1.413	3.341	1.04	12970.906

CHART 6: HIGH POWERS OF UNIVARIATE POLYNOMIAL

DEGREE = 2 MAXIMUM COEFFICIENT = 3

POWER	SUMS	MOD	BINOM	RMUL	RSQ	CRAFFT2	CRAFFT3	BEST(MS)
20	1.0	7.164	2.828	3.773	2.527	4.007	5.394	371.495
21	1.0	7.365	2.942	3.976	2.833	3.784	5.062	400.354
22	1.0	7.548	3.012	4.177	3.061	3.594	4.785	429.54
23	1.0	7.853	3.168	4.465	3.34	3.492	4.633	452.253
24	1.0	8.176	3.305	4.769	3.103	3.379	4.479	473.497
25	1.0	8.45	3.446	5.039	3.69	3.273	4.302	497.122
26	1.0	8.767	3.593	5.349	3.923	3.189	4.181	518.752
27	1.0	9.114	3.751	5.678	4.119	5.219	6.675	538.26
28	1.0	14.213	3.906	5.979	4.378	5.076	6.485	558.601
29	1.0	14.656	4.043	6.271	4.6	4.941	6.283	580.773
30	1.0	15.063	4.204	6.543	4.273	4.856	6.163	604.218

Bibliography

[1] Bogen et al. MACSYMA User's Manual. Project MAC. Massachusetts Institute of Technology, Cambridge, Mass. September 1972.

[2] Bonneau. "A Class of Finite Computation Structures Supporting the Fast Fourier Transform." Technical Memo No. 31, Project MAC, Mass. Institute of Technology, Cambridge, Mass. 1973.

[3] Borodin & Moenck. "Fast Modular Transforms Via Division." Proceedings of the 13th Conference on Switching and Automata Theory, 1972. pp. 90- 96.

[4] Collins. "On Computing Multiplicative Inverses in GF(p)." Mathematics of Computation, vol. 23, January 1969, pp. 197-200.

[5] Cooley & Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." Mathematics of Computation, vol. 19, April 1965. pp. 297-301.

[6] Fateman. "On the Computation of Powers of Polynomials." Department of Mathematics, Mass. Institute of Technology, Cambridge, Mass. 1972. (Unpublished)

[7] Good. "The Interaction Algorithm and Practical Fourier Analysis," Journal Royal Statistical Society. Series B. vol. 20, pp. 361-372.

[8] Grosswald. Topics from the Theory of Numbers. Macmillan Company, New York. 1966. Chapter 4.

[9] Heindel. "Computation of Powers of Multivariate Polynomials Over the Integers." Journal of Computer and System Sciences, vol 6, 1972. pp. 1 - 8.

[10] Heindel & Horowitz. "On Decreasing the Computing Time for Modular Arithmetic." Proceedings of the 12th Conference on Switching and Automata Theory, 1971. pp. 126-128.

[11] Horowitz. "The Efficient Calculation of Powers of Polynomials." Proceedings of the 13th Conference on Switching and Automata Theory,1972. pp. 97 - 104.

[12] Horowitz & Sahni. "On the Computation of Powers of a Class of Polynomials." Department of Computer Science, Cornell University, Ithaca, New York, tr 72-143.

[13] Knuth. The Art of Computer Programming, Vol. 2, "Semi-numerical Algorithms." Addison-Wesley, Reading, Mass. 1969.

[14] Nicholson. "Algebraic Theory of Finite Fourier Transforms." Journal of Computer and System Sciences, vol. 5, 1971. pp. 524 - 547.

[15] Pollard. "The Fast Fourier Transform in a Finite Field." Mathematics of Computation, vol. 25, number 114, April, 1971. pp. 365 - 374.