INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
Johnson, Howard Wayne

THE DESIGN OF DFT ALGORITHMS

Rice University  

University Microfilms International  300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1982
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Other

University Microfilms International
RICE UNIVERSITY

THE DESIGN OF DFT ALGORITHMS

by

HOWIE JOHNSON

A THESIS SUBMITTED
IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

C. Sidney Burrus
Professor of Electrical Engineering
Chairman

Don H. Johnson
Professor of Electrical Engineering

John E. Dennis
Professor of Mathematical Science

HOUSTON, TEXAS

APRIL 1982
THE DESIGN OF DFT ALGORITHMS

Howie Johnson        April, 1982

ABSTRACT

A broad class of efficient discrete Fourier transform algorithms is developed by partitioning short DFT algorithms into factors. The factored short DFT's are combined into longer DFT's using multidimensional index maps. By exploiting a property which allows some of the factors to commute, a large set of possible DFT algorithms is generated which contains both the prime factor algorithm (PFA) and the Winograd Fourier Transform Algorithm (WFTA) as special cases. The problem of finding an algorithm from this class which is optimal with respect to the specific add, multiply, and data transfer characteristics of a particular implementation is posed, and a highly effective dynamic programming algorithm is presented as a solution. Finally, it is demonstrated that the output reordering inherent in a PFA can be accomplished with zero data transfers by modifying the coefficients used in its constituent modules. Such a modification does not alter the signal flow topology of the module algorithms. A concrete procedure is derived to calculate the modified coefficients.
ACKNOWLEDGEMENTS

This work is a reflection of the love and guidance given me by my parents, the faculty at Rice University, and all my friends here in the Electrical Engineering Department. In particular I would like to thank my advisor, Sid Burrus, along with the rest of my thesis committee and express my appreciation for the people at ROLM corporation without whose assistance I would not have endeavored to complete my graduate education.
TABLE OF CONTENTS

1. INTRODUCTION

2. PART I - COMBINING MODULES

TOOLS:  2.1 The Discrete Fourier Transform
        2.2 Discrete Fourier Transform Modules
        2.3 The Prime Factor Algorithm
        2.4 Commutative Linear Transformations
        2.5 Winograd's Fourier Transform Algorithm

RESULTS:  2.6 A Class of Algorithms
          2.7 Comparing Algorithms
          2.8 Finding Optimal Members of the Class
          2.9 Programming the Solution Procedure
          2.10 Examples

3. PART II - INSIDE THE MODULES

TOOLS:  3.1 Euclid's Algorithm
        3.2 Primitive Nth Roots of Unity
        3.3 Cyclotomic Polynomials
        3.4 A Lemma

RESULTS:  3.5 Canonical Forms
          3.6 Permuting Outputs
          3.7 Calculating Coefficients
4. DISCUSSION

5. CONCLUSIONS

6. APPENDIX I — FORTRAN Listings

7. APPENDIX II — FORTRAN Listings
1. INTRODUCTION

When the Cooley–Tukey Fast Fourier Transform (FFT) was first published in 1965 [4] it was the fastest known algorithm for computing the Discrete Fourier Transform (DFT) and hence an excellent choice for performing spectral analysis and linear filtering, operations which are unquestionably of vast importance in digital signal processing [5–9]. The impact which the FFT had on signal processing was a direct result of the advantageous tradeoffs made in that algorithm of control complexity against arithmetic operations count. Since then a number of ways of reformulating the DFT have appeared, all of which are, like the Cooley–Tukey FFT, motivated by a desire to trade off control and arithmetic functions in new and useful ways.

Modern FFT algorithms can be traced in a logical sense, if not a chronological one, to the Common Factor Algorithm proposed by R.C. Singleton in 1969 [22]. The Common Factor Algorithm builds up a large DFT\(^1\) out of many smaller DFT algorithms, introducing in the process a number of so-called "twiddle" operations between the small DFT's. When the desired transform length is a power of two, Singleton's algorithm degenerates into the popular Radix-2 algorithm known as the Cooley–Tukey FFT [4]. The small DFT's used in the Cooley–Tukey algorithm are so trivial that the entire computation is dominated by the twiddle operations. Great efforts have gone into

\(^1\) The DFT is an orthogonal \(NxN\) linear transformation and the size of a DFT refers to its dimension, \(N\).
reducing the twiddle operations in the Cooley Tukey algorithm. A
variation of the Common Factor method uses the prime factor index map
proposed by I.J. Good in 1958 [15]. Use of the prime factor index
map, also called the Ruritarian index map, avoids twiddle operations
entirely but requires many of the constituent small DFT algorithms to
have prime lengths. Prime length DFT's were inherently inefficient
at the time this algorithm was first proposed and for that reason the
prime factor index map was primarily of theoretical interest until
1976 when Winograd [1] introduced a number of very efficient small
prime and prime to a power length DFT algorithms.

At present two distinct prime factor type algorithms are in use, the
Winograd Fourier Transform Algorithm (WFTA) [1] and the Prime Factor
Algorithm with Fast Convolution (PFA) [3]. Both algorithms use the
same short length DFT's proposed by Winograd, and both algorithms
also use the prime factor indexing structure. The two algorithms
differ in their internal orders of operations which ultimately affect
their overall operations counts and program control complexities.
Excellent examples of state of the art WFTA and PFA programs may be
found in [23] and [16], respectively. As a consequence of the use of
the prime factor indexing system both the WFTA and the PFA suffer
from an output scrambling effect which causes their outputs to appear
in scrambled order. This scrambling effect is similar in nature to
the bit-reversed order of outputs in a radix-2 FFT, and is exhibited
clearly in the PFA program [16], where the data is re-ordered by fol-
lowing the main body of the transform program with a data unscram-
bling routine.

Two open problems in this area are how to best order the operations inside a prime factor type algorithm and how to remove the output scrambling effect from a general purpose prime factor type program. Zohar [12] has studied the problem of ordering preweave and postweave sections of a WFTA algorithm and presents some tabulated results, but his data are applicable only to the WFTA case. Nussbaumer [9] presents algorithms which essentially involve alternate orderings of operations, but his algorithms are presented in an obscuring polynomial context and no methods are given for finding a best ordering. The general output permutation problem has been dealt with by Rothweiler [21] who manages to use separate input and output indexing on each small DFT to accomplish an overall data permutation. This technique involves no additional unscrambling data transfers, but does require separate calculation of the output indices. McClellan's WFTA program [23] does not compute the transformation in-place, so unscrambling is not a problem, the data is written back to the original input vector in scrambled order.

The contributions of this thesis may be roughly divided into three categories. First, a broad class of DFT algorithms is derived from the PFA formulation by permuting its internal order of operations. The WFTA is included in this class along with many intermediate algorithms some of which are superior to existing algorithms. Second, the problem of identifying the best algorithm out of the above class for a particular application is posed and solved under an appropriate
definition of "best". Finally, conditions are derived under which any particular in-place DFT algorithm can be trivially modified to permute its outputs. The modification affects only the coefficients used in the algorithm. Such a modification can be used to eliminate the unscrambling problem in a general purpose in-place prime factor type algorithm. An automated procedure is defined for computing the new coefficients required to implement (un)scrambled DFT's. This procedure greatly reduces the complexity of the module design process by eliminating the need for computing Chinese Remainder Theorem polynomial inverses.
2. PART I - COMBINING MODULES

The modules which are at the foundations of Winograd's DFT are themselves prime (or power of a prime) length DFT algorithms which have been constructed with great care and cunning on the basis of an elaborate theory of polynomial rings. The specifics of their construction are not relevant to the study of the macroscopic structure of the PFA, which is the main topic of this section. After some preliminary material covering important characteristics of the DFT itself and also DFT modules a derivation of the PFA is presented which closely follows that given in [3]. Then a Lemma concerning commutative linear transformations is proved, leading to the WFTA as a corollary. In 2.6 a close re-examination of the WFTA derivation reveals a large number of degrees of freedom not exploited in either the PFA or WFTA. Consideration of these extra degrees of freedom, which are related to the commutative nature of the linear transformations used in the PFA, generates a very large class of useful DFT algorithms. In many situations elements of this class can be shown to be superior to both the PFA and WFTA, and can furthermore be tailored to the specific multiply, add, and data transfer characteristics of a particular implementation. Means of comparing algorithms in this class are developed in 2.7 paving the way for the formulation in 2.8 of a well defined optimization problem which loosely translates as "find the best algorithm out of this class for a particular implementation". The proposed solution to this optimization problem is a highly effective dynamic programming procedure. That a
dynamic programming type solution is applicable to this problem is a fortunate consequence of the regular structure of the class of algorithms generated in 2.6 and the particular form of the cost function selected in 2.7. Details of the basic solution procedure are presented in 2.9, along with a number of embellishments which enhance the utility of the procedure. Examples are given.

2.1. The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) actually refers to a collection of invertible linear transformations which operate on complex valued vectors. For each integer \( N \geq 1 \) there exists a unique DFT, called the length \( N \) DFT, which maps the space of all complex \( N \) dimensional vectors into itself. The length \( N \) DFT is defined below in (1).

\[
y(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N} \quad (1)
\]

It is easily verified that this is an orthogonal expansion of the signal \( x(n) \) in terms of the \( N \) complex orthogonal signals \( e^{-j2\pi nk/N} \) for \( 0 \leq k \leq N-1 \). These same \( N \) orthogonal signals also turn out to be eigenvectors of \( N \)-point circular convolution, which is the connection between the DFT and linear filtering that is exploited in order to create linear filtering algorithms with the DFT. Other uses for the DFT, along with many of its interesting properties, are discussed at length in general references [6,7,10].

As expressed in (1), explicit computation of the DFT requires on the order of \( N^2 \) operations. If there were no other ways to compute that
expression, the DFT would not be an efficient computational tool. Fortunately, the DFT contains incredible symmetries and redundancies which may be removed by the clever programmer, reducing drastically the computational work load. This is the essence of all fast DFT algorithms. The fast DFT algorithms considered in this section are derived by a process which is based on the derivation of the PFA, which builds up large DFT's by putting together many smaller ones. The process of pasting together DFT's used in the PFA is an inherently efficient procedure, but beyond that, the small DFT's which are to be pasted are configured to be even more effective. Unfortunately, the processes which collude to create the efficient small DFT modules break down for longer lengths, leaving nothing but the PFA mechanism to efficiently construct large composite length DFT algorithms. Similar phenomena are at work in the power of two length FFT algorithms, but are masked by the overwhelming dominance of the twiddle factor computations.

2.2. Discrete Fourier Transform Modules

The efficient short length DFT algorithms used as building blocks in the PFA and in the WFTA are referred to as DFT modules. All of Winograd's modules are prime (or power of a prime) length DFT's. This does not preclude the use of general composite length modules, if the relevant constraints of the PFA technique are met, namely that the greatest common divisor of any pair of module lengths used in the same algorithm must be unity. Any length N DFT module, as originally presented by Winograd [1], will have the following canonical form:
\[ y = C(Ax \Theta By) \]  \hspace{1cm} (2)

where \( x \) is a length \( N \) complex input vector, and \( y \) is the length \( N \) complex output vector. The matrices \( A \), \( B \) and \( C \) are both real and rational, \( \Theta \) is a component by component dyadic product operator and \( w \) is a vector containing all \( N \)th roots of unity. The operator \( \Theta \) is an unusual vector operation which takes two length \( M \) vectors and produces a third length \( M \) vector according to the rule \( s(i) = t(i) \cdot u(i) \). This operation is equivalent to forming an \( M \) by \( M \) diagonal matrix \( T \) with the entries of \( t(i) \) spread along its diagonal and then computing the matrix product \( T w \). The operator \( \Theta \) is commutative, \( t \Theta u = u \Theta t \).

An \( N \)th root of unity is a complex number which, when raised to the power of \( N \), yields unity. For example, the fourth roots of unity are \( 1, +j, -j \) and \( -1 \). The number of \( N \)th roots of unity is always exactly equal to \( N \). This may be demonstrated by first noting that the equation \( Z^N - 1 = 0 \) can have at most \( N \) roots, and then verifying that all of the \( N \) distinct complex numbers \( e^{\frac{2\pi n}{N}} \) for \( 0 \leq n < N \) are indeed roots of unity. All the roots therefore have been identified, and there are exactly \( N \) of them. Other important algebraic properties of the \( N \)th roots of unity will be introduced as necessary in section 3.2.

The output dimension of \( A \) is typically greater than \( N \) but less than \( N \log_2(N) \) and can always be made less than \( N^2 + 1 \). For a concrete example of a length \( N \) DFT module, let \( A \) be equal to \( N \) identity matrices, each \( N \times N \), stacked one above another. Thus \( A x \) will produce \( N \) copies of \( x \). The matrix \( B \) must then be constructed so that the
product \( Bw \) is a listing of the \( N \) rows of a DFT matrix, one after another. \( B \) "unwraps" the DFT matrix. The reconstruction matrix, \( C \), then only has to collect appropriate terms in order to produce the \( N \) outputs. The resulting algorithm will in general contain \( N^2 \) complex multiplies. This is equivalent to the way one would normally compute a direct implementation of a DFT as a matrix multiplication.

The output dimension of \( A \) controls the number of multiplications used in the algorithm. If all of the entries in \( Bw \) are irrational then the number of multiplications is exactly equal to the output dimension of \( A \). In the length 16 DFT module only ten out of the eighteen entries in \( Bw \) happen to be irrational. Multiplications by simple rational numbers are generally ignored in the complexity analysis of these algorithms because they can be implemented with small numbers of additions. Multiplications by \( 1 \) or \( j \) are particularly easy to implement. The eight rational terms in the middle of the length 16 DFT are all equal to either \( 1 \) or \( j \) and therefore may rightly be ignored. The output dimension of \( A \) therefore is only an upper bound for the number of actual multiplications in the algorithm. The \( Bw \) terms in all of Winograd’s modules are either purely real or purely imaginary. This means that the dyadic operator \( \Theta \) will not have to compute full complex multiplies, but only needs to scale the real and imaginary parts of \( A \), and perhaps reverse their roles.

The ratio of the output dimension of \( A \) to its input dimension is called the "module expansion factor". This is a figure of merit for a module, in that it is related to the number of multiplies per
output point that are required in the computation of that algorithm. This ratio also turns out to be a determining factor in the efficiency of the WFTA, as discussed further in section 2.5. All modules necessarily have an expansion factor greater than or equal to one, because the DFT is an invertible transformation, and a module expansion factor less than one would imply that the matrix $A$ had rank less than $N$, rendering the overall transformation non-invertible. Typical expansion factors for useful modules range from 1 (very good, only possible on modules 2, 3, 4, 6, 12 and 8) to 2.1 (terrible, this is on the enormous length 17 module in [14]). Even with large expansion factors, a well written module can generally represent tremendous savings over a direct implementation of the DFT.

Singleton's Common Factor Algorithm [22] is constructed in a manner similar to the PFA, and also used DFT modules. The modules proposed by Singleton are not as efficient as Winograd's modules, and furthermore they cannot be represented in the form (2). Singleton's modules have been entirely replaced by Winograd's new modules for use in the PFA and WFTA.

The construction in (2) is called a bilinear algorithm because the output $y$ is a linear function of both the input $x$ and the vector $w$. The fact that such algorithms exist for the DFT and are efficient (i.e. the output dimension of $A$ is much less than $N^2$) is not obvious. The derivations of many modules and their uses may be studied further in [1, 3, 5, 9, 11, 12].
In the derivation of Winograd's modules the matrices A, B, and C are restricted to having real rational elements. Furthermore in useful modules the elements of A and C are typically small integers which greatly simplifies the computation of the matrix product Ax and also the matrix product by C. Having rational entries in B, however, is not an advantage because the product By is always precomputed and stored as a vector y. The algorithm which is actually computed at execution time is:

\[ y = C(Ax \otimes y) \]

It is conceivable that the rational restriction on the form of B may have some subtle cost in terms of the efficiency of the class of algorithms which may be so derived. Section 3.5 of this thesis explores the possibility that there might be bilinear DFT algorithms similar to (2), but in which B is not restricted to being a rational matrix. A system of rational equations is developed which must be satisfied by the vector of precomputed constants, By. The striking conclusion of this chapter is that removing the restriction of rationality from B creates no new algorithms.

Many times it will necessary to refer to the actual implementation form of the canonical representation in which the vector By is replaced by the vector of precomputed constants y. The three processing steps represented in this abbreviated form are called the preweave matrix (A), the multiply step (\( \otimes y \)), and the postweave matrix (C). If the terms in y are written out along the diagonal of a
matrix V then the algorithm may also be represented as CVA. It is important to remember that each of the three processing steps is a linear transformation.

The specification of the preweave and postweave matrices does not uniquely define a DFT module, as their particular realizations as sequences of additions and subtractions may drastically alter the appearance of a module program as well as make the algorithm more or less efficient. For this reason, DFT modules are most often specified either as a signal flow graph, or by means of a system of equations in which every operation is explicitly listed in the order of execution. Figure IA is an example of a flow graph representation. This algorithm is a length 5 module. Listings of module algorithms for lengths 2, 3, 4, 5, 7, 8, 9 and 16 may be found in [9], and modules for lengths 11, 13, 17, 19 and 25 appear in [14].

2.3. The Prime Factor Algorithm

The Prime Factor Algorithm (PFA) is one method used for combining short DFT modules to form long DFT algorithms. The term PFA can also refer to a particular working algorithm which has been constructed using the PFA method. In order to combine DFT modules into larger DFT's some multidimensional terminology will be necessary. In particular, the concepts of 'row' and 'column' as used with respect to a two dimensional data array will need to be expanded to have meaning in the case of multidimensional data. In a two dimensional data array a column of data is indexed by holding the second data index
Figure IA

LENGTH 5 DFT ALGORITHM
fixed and letting the first index range over all its possible values. Such a data structure will be called a 1-column because the first index ranges while the second index is fixed. A row in a 2-D array should likewise be called a 2-column because it is generated by varying the second index. In a general $M$ dimensional array an $i$-column is generated by varying the $i$th index and holding all the other indices fixed. Let $X$ be a data array in $M$ dimensions containing $N$ elements. Let $I$ be a particular integer from 1 to $M$. All the $I$-columns in $X$ will have the same length, and the collection of all $I$-columns completely fills the array. If there are $N_I$ elements in each $I$-column then there must be a total of $N/N_I$ distinct $I$-columns. The $I$-columns form a partition of the data array. For some integer $J$ not equal to $I$ (but between 1 and $M$) the length of an $I$-column is generally not equal to the length of a $J$-column. If $V$ is some $I$-column and $S$ is some $J$-column then $V$ and $S$ may overlap at most on one data point.

Fig II illustrates the general columns of a 2 dimensional array.

A brief sketch of the development of the PFA follows. Let $N_1, N_2, \ldots, N_m$ be a collection of relatively prime integers. This means that the greatest common divisor of any pair $(N_i, N_j)$ is unity. The $N_i$ represent the lengths of the modules which are going to be combined into a PFA. The length of the resulting PFA will be $N = \prod_{i=1}^{m} N_i$.

Typical values used for the lengths $N_i$ are 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17 and 19 because module algorithms have been written for those lengths.
Figure II

General Columns of 2-D Data Array
Define two shorthand symbols:

\[ d_i = \frac{N_i}{N} \quad \text{and also} \quad W_N = e^{-j2\pi/N} \quad (2) \]

Note that \( d_i \) will always be an integer because each the \( N_i \) is a factor of \( N \). Further define the notation \( \Theta_i = W_{N_i}^i \). A useful relationship between these three quantities is:

\[ \Theta_i = \frac{d_i}{W_N} \]

A multidimensional index map will be used to convert an \( m \)-tuple of indices into the one-dimensional index used to access the input and output data arrays. The map used here, known as the Hurwitzian index map [8], is a one-to-one and onto correspondence between the collection of indices \( (n_1, n_2, \ldots, n_m) \) where \( n_i \) is allowed to range from 0 to \( N_i - 1 \), and the integers from 0 to \( N \). Denoting the Hurwitzian map as \( R(n_1, n_2, \ldots, n_m) \) any summation over the simple index \( n \) may replaced by a multiple sum over the variables \( n_1, n_2, \ldots, n_m \) if \( n \) is replaced every where by \( R(n_1, n_2, \ldots, n_m) \). Equivalently, a new multidimensional data array may be defined:

\[ x'(n_1, n_2, \ldots, n_m) = x(R(n_1, n_2, \ldots, n_m)) \]

and \( x' \) is substituted for \( x \) when the simple summation is replaced by a multiply sum. It is sometimes convenient to drop the ' on \( x' \), and since the arguments of \( x \) and \( x' \) are so radically different no confusion should result. The plan in the PFA is to set up a normal length \( N \) DFT and then substitute for both the input and output indices using
the Ruritanian map. The single DFT summation will be replaced by a multiple summation. The output will be recovered from the multidimensional output array by mapping backwards through the Ruritanian map to find \( y(n) \) as a function of \( y'(n_1, n_2, \ldots, n_m) \). The exact equivalence is:

\[
y(n) = y'(R^{-1}(n))
\]

Due to the particular form of the Ruritanian index map and its interaction with the DFT coefficients, the kernel of the resulting multiple summation will be separable, yielding a great savings in computational effort. This separability is the cornerstone of the PFA method.

The Ruritanian correspondence for input and output indices, respectively, is given by:

\[
n = \sum_i d_i n_i \mod N \quad k = \sum_j d_j k_j \mod N
\]

(3)

The \( \mod \) function is discussed at length in [8]. It is a dyadic integer operator which divides the left-hand operand by the right-hand operand (called the modulus) and returns the remainder. Stated differently, given integers \( A \) and \( N \), there always exist integers \( K \) and \( R \) such that two conditions hold:

1) \( A = K \cdot N + R \)
2) \( 0 \leq R < N - 1 \)

The \( \mod \) operation is defined:

\[
A \mod N = R
\]
The mod function is an important concept, so here is another way to define it: A mod \( N \) is the smallest positive integer \( R \) such that \( R=A-KN \) for some (not necessarily positive) integer \( K \).

The inverse Ruritarian mapping is:

\[ n_i = \langle n \cdot d_i^{-1} \rangle \mod N_i \]

The success of this index map rests on the assumption that the module lengths be relatively prime.

Identity (4) will be needed in the PFA derivation.

\[ w_{nk}^N = w_{nj}^N = \frac{1}{N} \sum_{i=1}^{m} \Theta_i (d_in_i k_i) \quad (4) \]

This identity depends on the fact that for \( i \) not equal to \( j \), \( N \) divides \( d_id_j \) evenly, so that \( w_{nk}^N = 1 \). The PFA is derived in (5).

\[ y(k) = \sum_n x(n) w_{nk}^N \]

\[ y(k_1, \ldots, k_m) = \sum_{n_1=1}^{N_1} \Theta_1 d_{n_1 k_1} \cdots \sum_{n_m=1}^{N_m} \Theta_m d_{n_m k_m} x(n_1, \ldots, n_m) \quad (5) \]

Each sum may be treated as a linear transformation of the data array.

The linear transformation indicated by the \( i \)th sum is actually a collection of identical but independent linear transformations, one on each \( i \)-column of \( x \). These independent transformations correspond to length \( N_i \) DFT operations. The PFA formulation says to perform DFT's
on all \( m \)-columns, then DFT's on all \((m-1)\)-columns, and so on ending with DFT's on all 1-columns. If the factors of \( N \) are approximately the same size, then a rough estimate of the complexity of the PFA can be calculated. If there are \( m \) factors then \( N \) is approximately equal to \( \frac{N^m}{N_1} \) and the summations in (5) each represent \( \frac{N}{N_1} \) DFT's of length \( N_1 \). There are \( m \) summations, so the total number of length \( N_1 \) DFT's that must be performed is \( m \frac{N}{N_1} \). This expression may be rewritten as \( \frac{N}{N_1} \log_2 (N) \). If \( N_1 \) is interpreted as an approximate radix for this PFA then the above expression shows that the complexity of the PFA grows with \( N \) in a manner very similar to the complexity of the power of two FFT algorithms. The \( N \log_2 (N) \) growth rate is much slower than \( N^2 \), which is the growth rate of a direct matrix multiplication scheme. The \( N \log_2 (N) \) expression illustrates the fantastic computational savings available with the PFA method.

The linear transformations indicated in (5) are actually not quite DFT's. The difference is that in (5) the weighting coefficient of the summation is \( \theta_{i}^{n_k} \) instead of just \( \theta_{i}^{n_k} \) as in an ordinary length \( N_1 \) DFT. Fortunately, the \( d_i \) term turns out to do nothing other than change a normal DFT into a DFT with permuted outputs. In order to prove this claim, it must be demonstrated that the set of complex numbers \( \{ \frac{n_k}{N} \} \) is mapped bijectively (one-to-one and onto) into itself by the transformation \( x \mapsto x^d \) if \( d \) is relatively prime to \( N \). This will be sufficient because in general if the set \( \{ C(k) \} \) is mapped bijectively into itself by the function \( Z \), then we may replace
everywhere the expression $Z(C(k))$ by the expression $C(P(k))$ where $P(k)$ permutes the indices $k$. In the case at hand we would replace $k_i d_i P(k_i)$ by $W_{N_i}$ which will do nothing but permute the outputs of an otherwise normal DFT. The integers $d_i$ and $N_i$ are guaranteed to be relatively prime by construction. Such a permutation may be computed either by an explicit data scrambling operation after the evaluation of a normal DFT algorithm, or by rewriting that algorithm to inherently include the data scramble. Other means for achieving this end are developed in Part II of this thesis. Now for the proof:

First, we are dealing with a finite set, so if exponentiation is surjective (onto) then it must also automatically be injective (one-to-one). For each value $W_N^k$ an integer $t$ must be found such that $W_N^k = W_N^{t d}$. This is easy to do using a well-known property of greatest common denominators found in any standard text on number theory [18] which states that if $d$ and $N$ are relatively prime (have no common denominator other than unity) then there exist integers $A$ and $B$ such that $Ad + BN = 1$. Using this fact, let $t$ be equal to $Ak$, and check:

$$W_N^{d Ak} = W_N^{(1-BN)k} = W_N^{k-N(Bk)} = W_N^k$$

This is a concrete demonstration of surjectivity, and the proof is finished.

Note that since the order of summations in (5) may clearly be interchanged the linear transformations on $x$ corresponding to each sum must be commutative. This observation will be exploited in section
2.5 in order to develop the WFTA.

2.4. Commutative Linear Transformations

The commutativity of the linear transformations in the PFA is not tied to the DFT structure, but is quite general. Consider two linear transformations \( A \) and \( B \) which operate on a multidimensional data array \( X \). Let \( A \) operate identically and independently on every \( i \)-column of \( X \). Let \( B \) operate identically and independently on every \( j \)-column (\( j \) and \( i \) not equal) of \( X \).

**Lemma**

\( A \) and \( B \) are commutative linear transformations.

**Proof**

Without loss of generality and for notational convenience re-index the data so the \( i \)th and \( j \)th indices appear first, and denote the vector of all other indices by \( \bar{w} \). The actions of \( B \) and \( A \) are defined by:

\[
B(x)(k_i, n_j, \bar{w}) = \sum_{n_i} b(k_i, n_i) x(n_i, n_j, \bar{w})
\]

\[
A(x)(n_i, k_j, \bar{w}) = \sum_{n_j} a(k_j, n_j) x(n_i, n_j, \bar{w})
\]

Check that \( B(A) = A(B) \):
\[ B(A(x))(k_i, k_j, \bar{w}) = \sum_{n_i} b(k_i, n_i) \sum_{n_j} a(k_j, n_j) x(n_i, n_j, \bar{w}) \]

\[ = \sum_{n_j} a(k_j, n_j) \sum_{n_i} b(k_i, n_i) x(n_i, n_j, \bar{w}) \]

\[ = A(B(x))(k_i, k_j, \bar{w}) \]

***

In the PFA the linear transformation represented by the ith sum is composed of a collection of identical linear transformations, DFT modules, which operate on each i-column. This Lemma is therefore applicable to the PFA and states that the multiple summations in the PFA are all commutative. The earlier observation (section 2.3) that the PFA summations are commutative is completely equivalent to the result of this Lemma, however now the messy details of sums and indices have been replaced with a statement about linear transformations. Notice that whereas the input and output dimensions for each of the transformations in the PFA are equal, our Lemma is not restricted to that case. The various transformations are allowed to expand (or contract) the data array without losing their commutative properties. A careful inspection of the above proof will reveal that a transformation of i-columns can change only the dimension of the ith index which, although it may change the total number of j-columns, does not influence the length of any j-column, for j not equal to i. This data expansion and contraction is ultimately
responsible for the interesting add/multiply tradeoffs available in
the algorithms discussed in section 2.6.

The same Lemma may be proved in the restricted case of two dimen-
sional data using the associative property of matrix multiplication.
The following illustrative proof may be found in Parsons [17], but it
is unfortunately not expandable to higher dimensions without intro-
ducing tensor notation.

Let $X$ be a two dimensional data array with the entries of $X$ arranged
so that it looks like a matrix. Define the linear transformation $\bar{A}$
to be the application of the matrix $A$ to every column of $X$. Simi-
larly define $\bar{B}$ to be the matrix $B$ applied to every row of $X$. It is
easy to verify that the action of $\bar{A}$ on $X$ is equivalent to multiplying
$X$ on the left by the matrix $A$. This will transform every column of $X$
by $A$. In the same way, multiplying $X$ on the right by $B^t$ ($B$ trans-
spose) will transform every row of $X$ by $B$. The associative property
of matrix multiplication guarantees that the operations $\bar{A}$ and $\bar{B}$ will
commute, as verified below.

$$\bar{B}(\bar{A}(X)) = (AX)B^t = A(XB^t) = \bar{A}(\bar{B}(X))$$

2.5. Winograd's Fourier Transform Algorithm In this section the
WFTA will be derived as a variation of the PFA through the use of the
Lemma given in section 2.4 on commutative linear transformations.
The derivation given here is essentially the same as the one given by
Kolba and Parks in [3], except that the interchange of summations
used in that reference is replaced here by the commuting of operators. This derivation is readily expandable into the form that will be necessary in section 2.6.

As discussed in section 2.2 any DFT module algorithm may be decomposed into three sections, the preweave, multiply, and postweave steps. If each DFT module can be decomposed into three pieces then the composite action of any one module acting simultaneously on a number of different vectors may also be broken into three pieces:

1) Apply pre-weave to each vector
2) Apply multiply step to each vector
3) Apply post-weave to each vector

Each of the three resulting collective actions is still a valid linear transformation on the collection of vectors.

When this concept is applied to the PFA, the summation in (5) corresponding to the ith DFT algorithm can be decomposed into a succession of three linear transformations, each operating independently and identically on every i-column. These three transformations are formed from the preweave, multiply, and postweave sections of the ith DFT module, and are denoted here by the symbols $P_i$, $M_i$, and $Q_i$, respectively. Similarly break up the jth sum operation (j not equal to i) into $P_j$, $M_j$, and $Q_j$. By the Lemma on commutative linear transformations $P_i$ will (given the opportunity) commute with $P_j$ because those two transformations operate along different indices. Theorem I similarly indicates that any of the i-column operations
will commute with any of the \( j \)-column operations. Using this commutativity the PFA can be modified in the following way:

Start with a valid PFA algorithm.
(Composing functions on the right.)

\[
P_m M Q P_{m} M_2 Q_2 \cdots P_1 M_1 Q_1
\]

Move all the \( P \) terms to the left past all the \( M \) terms.

\[
P_m P_{m} \cdots P_{m} M Q_{m} Q_2 \cdots M_1 Q_1
\]

Move all the \( Q \) terms to the right past all the \( M \) terms.

\[
P_m P_{m} \cdots P_{m} M_2 Q_{m} Q_2 \cdots Q_1
\]

Since the multiply steps from each module are nothing more than scaling operations, each collective operator \( M_1 \) must also be nothing more than a scaling operation. The succession of operators \( M_{m_2} \cdots M_1 \) is therefore a succession of scaling operations. If all the scaling constants for each data location are multiplied out before execution time, then there will be only one composite multiply step left in the executable part of the algorithm (6), which will be denoted \( M_c \).

\[
P_m P_{m} \cdots P_{m} M Q_{m} Q_2 \cdots Q_1
\] (6)

Algorithm (6) is equivalent to the WFTA, the only difference being that Winograd uses slightly different input and output index maps which result in the data output permutation taking place after the
whole computation instead of inside each module.

The important aspect of the WFTA is that by commuting parts of the PFA computation it is possible to arrange for several multiply steps to become adjacent, where they may be collapsed into a single step. This process, which is called nesting multiplies effectively annihilates many of the multiply steps. The drawback becomes apparent when the data size present before the composite multiply step is checked. Due to the data array expansion in the middle of each module, the data size at the composite multiply step can become ridiculously large. This happens because the total data size at that point is the product of the module expansion factors for each module used in the algorithm times the input data size. This may be checked by inspecting the indexing limits for each of the preweave summations.

To illustrate these points, assume there exists a large collection of relatively prime modules of approximately the same length, \( N \), and each having a data expansion factor of \( \alpha \leq 1 \). Let the number of effective multiplies be equal to the preweave output dimension in each case, which will be approximately \( \alpha N \). By examining the expressions for the number of multiplies in the WFTA and PFA implementations as the number of modules used becomes very large, the data expansion limitations of the WFTA will become obvious. If \( m \) modules are combined in the WFTA manner, the nested multiply data size will be equal to the product of all preweave output sizes which is on the order of \((\alpha N)^m\). If the same modules are combined using the PFA method there will be \( m \) stages with \( N^{m-1} \) transforms in each stage each having \( \alpha N \)
multiplies leading to a total of $mN^{m-1}aN$ multiplications. Take the ratio of WFTA multiplies to PFA multiplies:

$$\frac{(aN)^m}{mN^{m-1}aN} = \frac{1}{m^m}$$

This expression grows without bound as $m$ is increased.

Of course, the exact conditions for this argument will not be duplicated in any practical case, but it serves to illustrate the point that as many modules are combined the WFTA is asymptotically less efficient (in terms of its multiply count) than the PFA.

It is desirable to create algorithms which combine the good aspects of multiply nesting while containing the data expansion problem. By commuting about the various blocks it is possible to produce hybrid algorithms that are part PFA and part WFTA. Some of these hybrid algorithms have superior characteristics to both the PFA and WFTA.

In particular, in the case of the length $9009=7 \times 9 \times 11 \times 13$ DFT the number of multiplications in the WFTA exceeds the number of multiplications in an algorithm which nests together only the length 7, 9 and 13 modules. That group is then treated as a length 819 module and combined in the PFA manner with the length 11 module. This example is treated in more detail in section 2.10.

The PFA, WFTA and hybrid algorithms generated by nesting modules irregularly are all part of a much larger class of DFT algorithms developed in section 2.6.
2.6. A Class of Algorithms

According to Figure IB, the length 5 DFT module can be decomposed, or split, into five pieces. This type of splitting is a very general property of Winograd's DFT modules. In fact, any linear transformation that can be written as a multistaged flow graph (similar to Figure IA) can be split into a succession of linear transformations by simply cutting the flow graph with vertical lines as in Figures IA, B. If the length 5 DFT module is used in the context of a PFA as discussed in section 2.3 then the linear transformation on the data array induced by the length 5 module also breaks neatly into five pieces, which will be called A, C, D, E, and F. By the commutation lemma each of these transformations will commute with any transformation induced by a different DFT module. If each module in a multidimensional DFT is broken into many small fragments a large new class of DFT algorithms may be generated by considering all the different ways of commuting the fragments. Specifically, given a particular factorization of a collection of DFT modules the set of all multidimensional DFT algorithms that can be produced by interleaving the factors (using the commutation property) is called the vertically split class of DFT algorithms. It is not immediately clear that any of the commuted algorithms will be better than the old ones, however, since these algorithms have been created by an extension of the process which generated the WFTA, namely fragmenting each module and commuting the pieces, it might be reasonable to expect that at least some of the new ones will have interesting properties. The length
Figure 1B
LENGTH 5 DFT ALGORITHM
35=5x7 algorithm is presented as an example.

2.6.1. Example: 5x7 DFT

In Fig. IB the length 5 DFT algorithm has been split, or factored, into five pieces named a,c,d,e, and f. This particular factoring was generated according to a rule of thumb which suggests that a module should be factored at each point in its signal flow graph where the data dimension changes. In the length 5 example section c changes the data dimension from 5 to 6 and section d is the multiply step which must be separate if it is to be nested with multiply steps from other modules. Section e changes the data size back to 5. This rule of thumb is justified in section 4.

Let the composite operations generated from each piece be given capital letters A,C,D,E, and F respectively. The length 7 DFT module has been split into five fragments in Figure III, labeled t,w,x,y, and z. In the table which follows, a normal PFA is expressed (7), composing functions on the right and using capital letters for composite operations. The WFTA is similarly expressed (8). In the PFA, fragments ACDEF and TWXYZ remain grouped (the modules are considered to be indivisible). In the WFTA the following fragments, representing pre-weave, multiply, and post-weave sections, are grouped: AC, D, EF, TW, X, and YZ. The pre- and post-weave groups in the WFTA have been ordered to minimize the addition count. Add/Multiply counts are indicated for each algorithm. The multiply counts assume that any multiplications that can be 'collapsed' together have been so
Figure III
LENGTH 7 DFT ALGORITHM
treated. Also, multiplies by 1 and -1 are not counted. In (9) a new algorithm has been selected which does not have grouping restrictions (but does have ordering restrictions within each module, i.e. D must follow C which must follow A, etc.).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ordering</th>
<th>Adds</th>
<th>Mpy $\text{a}$</th>
<th>+Mpy $\text{a}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7)</td>
<td>PFA: (ACDEF)(TWXYZ)</td>
<td>299</td>
<td>75</td>
<td>374</td>
</tr>
<tr>
<td>(8)</td>
<td>WFTA: (TW)(AC)(D)(X)(EF)(YZ)</td>
<td>333</td>
<td>53</td>
<td>386</td>
</tr>
<tr>
<td>(9)</td>
<td>New: ATWCDXYEZF</td>
<td>305</td>
<td>53</td>
<td>358</td>
</tr>
</tbody>
</table>

The total number of additions in each algorithm may be accounted for by checking the number of times each factor will be used and multiplying by the number of additions in that factor. The total number of additions due to each factor is then summed. For instance, in algorithm (8) the column operation "c" is used 9 times. This happens because the last row operation, W, had 9 outputs which left 9 columns in the data array. If the operations C and W are commuted, then "c" will only be used 7 times (the output dimension of T), but "w" will then be used 6 times (the output dimension of C) instead of 5, as when it follows A. The act of commuting C and W therefore increases the number of type "w" factors by one and decreases the number of "c" factors by two. Since there are two additions in "w" and only one in "c", the total addition count would remain the same. However, there may be other considerations. In algorithm (9) the operation TWC can be applied with only one change of indexing, between W and C. The operation TCW requires two changes of indexing. Of course if the algorithm is written out in straight line code, the indexing
differences are inconsequential. These and other differences are pursued further in section 2.7.

The multiply counts are computed similarly, except that when two multiply steps (such as D and X) are adjacent they may be nested together. Each non-unity multiplication in the resulting nested multiply step counts as one multiply. Multiplies by unity are not counted.

Algorithm (9) has the same number of multiplies as the WFTA method, but fewer adds. If additions are very expensive, the PFA is a good choice. If multiplications are very expensive, then either the WFTA or (9) will be optimal. If additions and multiplications have equal weight and there are no other considerations then algorithm (9) is the best.

2.6.2. Relationship to Split Prime Factor and Split Nested Algorithms

The vertically split class of algorithms presented here is very similar to H. J. Nussbaumer's collection of split prime factor and split nested DFT algorithms [3], although the algorithms here have been derived from different principles. Algorithm (9), for example, from the previous section can be derived as split nested DFT. Nussbaumer's general approach splits the DFT computations into disjoint (multidimensional) correlations which may then be evaluated using any convolution technique, allowing for detailed individual control of each correlation. The algorithms described here do not
always allow such complete freedom, however their advantage lies in the conceptual regularity of this approach and the relatively easy manner in which they may be programmed.

The class of algorithms introduced by this paper includes at least two groups of algorithms not described by the split PFA and split nested DFT theory. The Real Data WFTA described by T. W. Parsons [10] can be derived using the methods of this paper in two steps. First, rearrange each module so that the real and imaginary parts are not combined together until the very last stage. Then simply commute the last stage of each module in the WFTA to the far end of the whole algorithm. This procedure for creating a Real Data Transform may also be applied to the PFA with equal ease. Secondly, it is not necessary to restrict application of the techniques presented here to Winograd-type convolutional DFT modules. Any module that can be represented as a staged signal flow graph can be used. For example, a Radix-2 module might be combined with a length 5 (convolutional) DFT module by nesting one set of twiddle factors with the multiplications from the length 5 DFT.

2.6.3. A Representation of the Class of Algorithms

This representation is most easily visualized for the two dimensional PFA. For a given factorization of two modules, (assume the factorizations are not internally commutable), a rectangular grid is used with the vertical axis representing one set of factors (in order) while the horizontal axis represents the other set of factors (also
in order). Any positive unit increment path\(^2\) through the grid from the origin to the opposite corner corresponds to a valid DFT Algorithm (Fig. IV). The set of algorithms which can be represented by such a path is exactly the vertically split class. The class is defined by the particular factorization of the DFT modules involved (including the specific order of factorization).

In Figure IV the double bold line represents a PFB where all of the factors of module A are executed before factors of module B. The hashed line represents a WFTA which starts with the module B preweave, then the module A preweave, followed by the two multiply steps in succession so that they can be nested, and finishing with the postweave sections of module B and then A.

The general m-dimensional case may be represented by an m-dimensional grid where the \(i\)th axis is labeled with the names of the factors of the \(i\)th DFT module. As in the two-factor case, the set of positive unit increment paths through the grid from origin to the opposite corner corresponds to the set of vertically split DFT algorithms.

2.7. Comparing Algorithms

Two of the most popular measures of DFT algorithm desirability are the total numbers of additions and multiplications. These may be evaluated for an algorithm in the vertically split class as discussed in the example in section 2.6. Another important parameter may be

\(^2\)A positive increment path moves along only one coordinate at a time and always in a positive direction.
Finding the best vertically split DFT is a least cost path problem that can be solved using dynamic programming.
the total number of data memory accesses.

In a software DFT algorithm with an indexing structure similar to
[16] changes of indexing are important. These happen whenever fac-
tors from different modules are adjacent. The effect hinges on a
software structure where there is only one copy of each factor which
is set up to use indirect addressing. Before each execution of that
factor, an indirect vector of addresses must be set up which points
to the data to be manipulated.

If two successive factor operations are from the same module, then
the indirect indexing work can be shared between them by applying
both factors in succession to each row instead of applying the first
factor to each row and then recalculating the same indices to apply
the second factor to each row. This approach is equivalent to con-
sidering the two factors to be one larger indecomposable factor.

If two successive factors come from different modules, then the first
factor must be completely applied to every row before work with the
second factor can begin. The indexing calculations are different and
cannot be shared.

A number of indexing schemes are possible which surmount the indexing
calculation problem by trading off program space, culminating in the
use of straight-line code where all the addresses are precomputed and
written directly into the program, along with multiple copies of each
factor. Useful measures of the amount of indexing work required might
be the number of times re-indexing is required and the total number
of factors executed. These measures are intended to model in a crude way the control complexity overhead in a DFT algorithm.

A final means of comparing algorithms will be by the largest data array size required by the algorithm. This can be used to distinguish between algorithms which will fit in the data address space of particular processors and those which will not.

These six measures of algorithm performance can be combined into a single figure of merit, given that one knows the characteristics of the intended application. The idea is that for each application some algorithm must have a best figure of merit, and if the figure of merit calculation accurately models the behavior of the potential application then the best algorithm for that application will have been found, or at least an algorithm of close to optimal performance if the modeling is not completely accurate.

The figure of merit calculation used in section 2.3 is a cost function which is computed as a positively weighted sum of the number of additions, multiplies, memory transfers, changes of index and total number of factor executions. The weights will depend on the intended application. This cost function will be supplemented with a rule which throws out all algorithms with maximum data storage requirements exceeding a specified limit. The problem of finding a "best" algorithm for a particular application has now been converted into a three stage process:

1) Select an appropriate set of DFT modules and factor them.
2) Select modeling constants which reflect the constraints of the application.

3) Find an algorithm from the vertically split class with the smallest possible cost.

2.8. Finding the Best Algorithm

Given a collection of DFT modules, a particular factorization of each module, and a set of weights for the figure of merit calculation how can the best algorithm from the vertically split class be located? A direct search of all possibilities is feasible for small problems. However, the number of distinct vertically split algorithms that can be formed from m modules of \( n_i \) factors (1\( \leq i \leq m \)), respectively, is:

\[
\frac{\sum_{i=1}^{m} n_i !}{\prod_{i=1}^{m} (n_i !)}
\]

For the four module problem \( N = 7 \times 9 \times 11 \times 13 \) with 5, 5, 7, ... and 7 factors per module, respectively, there are about \( 1.69 \times 10^{12} \) distinct vertically split algorithms. Obviously, a direct search would be impractical in this case. A very efficient solution is developed below.

In order to clearly develop the solution procedure, only weights corresponding to the number of adds and multiplies will be used in the figure of merit calculation. In section 2.8.3 memory access, memory size limitations and control overhead will be added to the problem.
2.8.1. Basic Solution Procedure

Given a set of factored DFT modules let a realization be defined as an ordered sequence of some subset of the available DFT factors. A valid realization respects the ordering constraints of the factors within each module (the factors of each module must be executed in the given order). Any valid realization which uses all the available modules can be used to compute a DFT. Any realization has a path length defined to be equal to the number of factors used in the realization. A computation is defined to be the overall linear transformation accomplished by a valid realization. By the commutation property it is clear that any particular computation can generally be realized in a number of ways. Furthermore, all valid realizations which result in the same computation will necessarily have identical path lengths, which will by definition be equal to the path length of the computation.

Visualizing the vertically split class of algorithms as the collection of positive unit increment paths through a multidimensional grid, a computation might be represented by two vertices in the grid. A valid realization of that computation would be any particular positive unit increment path from the first vertex to the second vertex. The path length of the computation would be the sum of the differences between the coordinates of the two vertices in the multidimensional space. Not all pairs of vertices represent computations, because there is not in all cases a positive unit increment path from one vertex to the other.
For the remainder of section 2 all realizations will be valid realizations and all computations will have one vertex at the origin. Each such computation can be represented by one vertex, and the path length of the computation is equal to the sum of the coordinates of that vertex.

Let the set of all allowable computations which may be performed by concatenating one more factor onto computation $A$ be called the set of successor computations to $A$, or $\text{Suc}(A)$. This set corresponds to the collection of vertices which may be reached from $A$ in one positive unit increment step. Let $P_n$ be the set of all computations with path length equal to $n$. An optimal realization of a computation is a valid realization whose cost is minimal over all valid realizations which result in that computation.

Let $A$ be a computation and $S$ be a particular (valid) realization of computation $A$. For a moment, assume that the difference between the cost of $x=S+f$ ($x\in\text{Suc}(S)$) and the cost of $S$ depends only upon $A$ and information related to the module factors. Specifically, the cost of $x$ will be equal to the cost of $S$ plus the cost of step $f$ given that it follows $A$. Using the multidimensional cube representation this is equivalent to saying that the incremental cost along any particular path from $A$ to $B$ is independent of the path which led to $A$. This assumption is sufficient to imply R. Bellman's "Principle of Optimality" [24] which, when applied to this problem, states that given any optimal realization $R$, the realization $S=R-f$ formed by stripping off the last factor $f$ from $R$ will also be an optimal realization.
Sufficiency may be demonstrated by noting that if \( S \) had not been optimal, then there would have been another valid realization, say \( S' \), which would perform the same computation as \( S \) but with a lower cost. The cost of the realization \( S'+f \) would then be lower than \( R \) because by assumption the incremental cost of adding \( f \) is not a function of the particular realization of \( R-f \), and \( S' \) had a lower cost than \( S \). But this can never happen if \( R \) is optimal, so \( R-f \) must always also be optimal.

Bellman [24] shows that a problem which satisfies the "Principle of Optimality" can be solved with a staged optimizing scheme commonly known as Dynamic Programming. For the problem at hand the Dynamic Programming approach will first find optimal realizations for all computations of path length \( n \) and use that information to find optimal solutions to all computations of path length \( n+1 \).

Let \( \text{cost}(x, S) \) be the cost of \( x \) given that it is a successor of a particular realization \( S \). Let \( \text{cost}(x, A) \) be the cost of \( x \) given that it is a successor of computation \( A \) and that \( A \) is optimally realized. Given an optimal realization for each computation of path length \( n \), the following procedure (written in the form of a computer program) will then find the minimum cost for each computation of path length \( n+1 \):
For each \( x \in P_n \)

Let \( S \) be an optimal realization of \( A \)
For each \( x \in Suc(S) \)
Compute \( cost(x, S) \)
\( cost(x, A) = cost(x, S) \) \((11)\)
End
End

For each \( x \in P_{n+1} \)

\( cost(x) = \min [cost(x, A)] \) \((12)\)
take the \( \min \) over all \( A \) where \( x \in Suc(A) \)
End

This procedure works because

\[
P_{n+1} = \bigcup_{A \in P_n} Suc(S) ; \ S \text{ an optimal realization of } A
\]

This statement means that any optimal realization in \( P_{n+1} \) must be equivalent to an optimal realization from \( P_n \) concatenated with one more factor. Since the procedure checks all possible successors for a set of optimal realizations which covers \( P_n \) it must hit upon an optimal realization for each computation in \( P_{n+1} \). The optimal realization for \( x \in P_{n+1} \) may be constructed as follows: let \( A_s \) satisfy (12); concatenate an optimal realization for \( A_s \) with the step from \( A_s \) to \( x \).

This is an efficient procedure because at pass \( n \) the algorithm deals only with the successors to every computation of path length \( n \).

After all the passes, every successor to each computation has been examined. If the problem has \( M \) modules with \( n_i \) factors in each module then there are approximately \( M \) successors to each computation and

\[
\text{the total computational effort required is proportional to } M \prod_{i=1}^{M} n_i \text{, or}
\]

if the \( n_i \)'s are about the same size, \( M(n)^M \). This is in contrast to a
brute force search of all paths through the multidimensional grid, which would require computational effort proportional to the expression in (10). In the length 7x9x11x13 example, the dynamic programming method examines 1225 cases, which is a far cry from $1.69 \times 10^{12}$.

The procedure is started with the fact that there is only one computation of zero length (a null operation) and it has zero cost. The final answer appears as the optimal realization to the computation of path length $\sum_{i=1}^{n} x_i$, which represents the complete DFT calculation.

Unfortunately, the cost calculation in (12) is not always well defined. Such a situation will occur if the incremental cost of some particular path from computation A to computation B is influenced by the particular realization which led to A. The solution to this problem is to break up the set of all realizations which compute A into a number of smaller sets, clusters, over which the incremental cost function is sure to be constant. Each cluster will have a path length equal to the path length of A. The overall solution procedure now goes through as before, replacing "computations" everywhere with "clusters". In order to program this technique, there must be a simple way to uniquely name each cluster, and a way to generate $\text{Suc}(C)$ for any cluster C. This process tears the vertices of the multidimensional grid visualization into many pieces, destroying the inherent symmetry of the grid, but does not destroy the properties of the grid important to the success of the dynamic programming method, most importantly that the path lengths of all realizations leading to
the same cluster are identical. The requirements for breaking computations into clusters are intimately related to the form of the cost function. The cluster definitions must be specific enough to insure that all incremental costs subsequent to any cluster are indeed constant over all realizations in the cluster. In practice, more clusters are used than necessary in order to make generation and naming simpler. At the termination of the procedure, if several clusters have maximal path lengths then they each must represent the entire DFT computation and the best one may be selected.

If the incremental cost of a path from A to B is only slightly dependent upon the preceding path, perhaps depending only on the last step in the path leading to A, then the number of clusters may not be too much larger than the number of computations. As the incremental cost from A to B becomes more heavily dependent upon the preceding path, the number of required clusters soon approaches the number of distinct realizations, and the solution procedure reverts to a direct search. Fortunately, the incremental costs considered in this paper depend only upon the immediately preceding step, or upon similar predicates which do not expand the number of required clusters excessively.

2.8.2. Practical Implementation

Three points must be checked before this optimization procedure can be called practical. First, a set of input data requirements must be formulated. There must be a concise way to state the essential pro-
properties of each module and its factorization. Second, a rule for generating successor clusters, and also a cluster naming convention will be needed. Last, it must be demonstrated that the costs are indeed well defined and computable.

2.8.2.1. Input Requirements

The optimization procedure will need to know:

- Number of modules
- Number of factors in each module
- Per add cost (For the specific application)
- Per multiply cost (For the specific application)

For each factor some assumptions will be necessary. Each factor must contain either adds or multiplies but not both. Adjacent multiply factors are not allowed in the same module. If a factor is composed of adds (or multiplies) then the number of adds (or multiplies) must be specified. The number of inputs and outputs (the data size) must be specified for each factor. The factors of each module must be specified in the same order in which they are to be executed.

2.8.2.2. Cluster Names and Successors

Since the factors of each module are assumed to be executed in a fixed order it is not necessary to list all the factors in a computation in order to unambiguously identify it. A list of the index number of the last factor in each module will suffice. Cluster names also associate a flag, called a nesting flag, with each module. The
nesting flags are used to differentiate clusters which represent the same computation. If a realization ends with a succession of multiply-type factors from modules i, j, and k then it belongs to a cluster with nesting flags i, j, and k set to one. If a cluster name has no nesting flags set, then it represents realizations that do not end in multiply-type factors. This cluster definition has been carefully devised so that the incremental cost from some cluster to one of its successors will be constant over all realizations in the cluster.

Successors are easy to generate. Given a cluster name, for each module try to increment the factor index for that module (check that there are more factors available in that module). Incrementing the factor index for a module concatenates another factor onto the cluster of realizations. If the new factor is an add-type factor, reset all nesting flags in the new cluster name. If the new factor is a multiply-type factor, copy the nesting flags from the old cluster name to the new name, and then set the nesting flag in the new name which represents the module corresponding to the new factor.

2.8.2.3. Computing Costs

Starting with cluster A suppose that using the successor generating procedure the next factor from module M is appended to generate cluster B. What is cost(B,A)? This is the cost of attaining cluster B given that it is a direct successor of cluster A, and that A is optimally realized.
Let \( \text{dim}(i) \) be the output dimension of the last factor from the \( i \)th module in \( A \). Let \( \text{adds} \) be the number of additions in the new factor. The incremental number of additions from \( A \) to \( B \) is:

\[
\frac{\text{adds}}{\text{dim}(M)} \times \prod_{i} \text{dim}(i)
\]

Let \( F \) be the set of module indices for which the nesting flag in \( A \) is set to one. Let \( \text{mul}(i) \) be the number of multiplies in the last factor from the \( i \)th module in \( A \) \((\text{mul}(i)=0 \text{ if the last factor in the } i \text{th module is an add-type module})\). The incremental number of multiplications from \( A \) to \( B \) is:

\[
\prod_{i \in F} [\text{dim}(i)-\text{mul}(i)] \times \prod_{i \in F} \text{dim}(i) \times \frac{\text{mul}(M)}{\text{dim}(M)}
\]

This formula takes into account the nesting of multiplications that results when two multiply modules are adjacent.

Next compute the incremental cost by weighing the number of incremental adds and multiplies by their respective per unit costs and adding them together. Finally, \( \text{cost}(B,A)=\text{cost}(A)+\text{incremental cost} \).

2.8.3. Other Embellishments

The basic optimization procedure above will find an element of the vertically split class which has a minimum cost (over the class) where the cost is a weighted linear combination of the number of adds and multiplies in an algorithm. The basic procedure may be embellished by using more complicated cost functions that more accurately model the parameter(s) one actually wants to minimize, such as
program execution time.

2.8.3.1. Data Memory Transfers

In many situations the two algorithms below will have different execution times:

\[
\text{Alg. 1} \quad A = B+C+D
\]

\[
\text{Alg. 2} \quad T = B+C \quad A = T+D
\]

Both algorithms, however, are considered to consist of two additions, and therefore are treated as having equal costs by the solution procedure [20]. This problem may be rectified by actually timing the execution speed of each add-type factor and using that cost, multiplied by the number of times each factor is used, in place of the number of additions. In the authors' implementation, a count of data memory read/write cycles, adds, and multiplies is maintained in a data base for each factor. At the time of optimization weights are assigned to each data memory cycle, add, and multiply and the weighted sum is used to compute algorithm costs.

2.8.3.2. Data Storage Constraint

A maximum data storage constraint may be imposed on the optimization procedure. The proper application of this constraint depends on the implementation of the optimal algorithm. If each factor of the optimal algorithm is to be executed independently and, of course, in
the optimal order then the maximum data storage requirement for that algorithm may be found by checking the data array size after each factor. The appropriate constraining action in this case would be to check the data array size for each cluster in the optimization procedure and assign an infinitely large cost to those clusters which exceed the maximum allowable data array constraint. This approach ignores a simple and effective means of saving data storage. Consider the PFA (which is always an element of the vertically split class). All the factors of module 1 are executed first, then module 2, etc. Since each module typically contains factors which expand the data array size, the above procedure would compute a maximum data array requirement for the PFA somewhat larger than the input data size. But the PFA can be calculated without data expansion. Instead of executing \(N/N_1\) copies of the first factor, storing that data, then executing \(N/N_1\) copies of the second factor, etc., it is possible to first combine all factors of module 1 into one larger factor and then execute \(N/N_1\) copies of the larger “collapsed” factor. Since module 1 is just a DFT, the data storage requirement after execution of the collapsed factor is equal to the input requirement. Each module in the PFA may be collapsed in this manner. In general, the idea is to always collapse together adjacent factors if they are from the same module. With the understanding that algorithms will always be executed in this collapsed form, the constraining procedure only needs to check the data array size between factors from different modules. Any algorithm which violates the data size constraint will be assigned an infinitely large cost. As a practical programming
matter, if the step from cluster A to its successor B violates the
data storage constraint it is easier to simply treat B as if it had
never been a successor of A than it is to carry around "infinite"
cost flags.

Since the cost function now depends somewhat on the ordering of fac-
tors, the clusters must be broken down further. Associate a new
flag, called the direction flag, with each cluster name. If a real-
ization ends with an add-type factor from module M then that realiza-
tion belongs to a cluster with the direction flag set to M. If a
realization ends with a multiply factor, the direction flag points to
the module corresponding to the last add factor in the realization.
This is sufficient to insure that subsequent incremental costs are
constant over all realizations in any cluster, and will also be con-
venient for computing overhead costs in the next section. The fact
that the direction flag is "delayed" across realizations of nested
multiply factors is inconsequential to the data size checking routine
given here because multiply factors do not change the data size any-
way. There must be a null value for the flag, which is used on the
cluster with path length zero.

Here is the cluster generation rule. Suppose an add-type factor from
module M is concatenated to cluster A to generate cluster B. Set the
direction flag in B to point to M. Suppose a multiply-type factor
from module M is concatenated to cluster A to generate cluster B.
Pass forward the direction flag from A, that is, set the direction
flag in B equal to the direction flag in A.
Here is the incremental data size cost calculation rule. Suppose a factor $F$ from module $M$ is concatenated to cluster $A$ to generate cluster $B$. Check the direction flag on $A$. If the direction flag equals $M$, do nothing. If the direction flag is not equal to $M$ then check the data size at $A$.

$$\text{Datasize} = \prod_{i} \text{dim}(i)$$

Do nothing if the data size is below the constraint, otherwise delete $B$ from $\text{Suc}(A)$.

2.8.3.3. Program Overhead Costs

In the PFA, one pass is made through the data array for each DFT module. In the WFTA, two passes must be made for each module. If an algorithm is written out in straight-line code its execution time may be accurately modeled by the number of memory cycles, adds, and multiplies in the algorithm. If, on the other hand, an algorithm is written with loops such as the Prime Factor FFT Program in [9] then the model does not work well because it does not take into account the cost of looping or indexing. If a model for the overhead (indexing and looping) costs can be derived that can be computed based on information available to the optimization program then that cost model can be incorporated into the optimization program.

Here is the authors' overhead model. For each add-type factor assess a fixed indexing cost plus a variable cost proportional to the number of times the factor is used. Treat a single (non-nested) multiply
factor the same as an add-type factor. If two adjacent add-type factors come from the same module, then the two factors may be collapsed together, sharing all indexing calculations. If several adjacent (nested) multiply-type factors appear, assess a fixed cost to the whole nested group plus a variable cost proportional to the number of nested multiplications in the group.

There is enough information in the cluster names as defined in the previous section to compute the overhead incremental costs. The reason for requiring the direction flag on clusters ending in multiply factors to point to the last add factor in the realization is that the overhead cost on a realization ending in a multiply factor cannot be computed until the total number of nested factors has been established. The number of nested multiply factors is not known until an add factor is reached (this presumes that modules do not end with multiply factors). The overhead cost, therefore, is only computed for clusters which end in add factors, and the "delayed" direction pointer is needed to look back over the nested multiply factors to check for collapsibility. The incremental overhead cost calculation rules which implement the authors' overhead cost model are rather involved, and are not included here.

2.9. Programming the Solution Procedure

In the authors' FORTRAN implementation of the optimization procedure, several shortcuts are used. The heart of the procedure works from a structure called the old list, representing cluster in $P_n$, and gen-
creates a new list of clusters in $P_{n+1}$. The old list contains, for each cluster name in $P_n$, a minimum cost, add count, multiply count, and a backward pointer. The add and multiply counts actually have nothing to do with the optimization procedure. They are only a convenience for reading out the number of adds and multiplies in the final algorithm and may be deleted if that information is not desired. The total cost of the solution will be available at the termination of the procedure. Suppose $A$ is in $P_n$ and $B$ is a successor to $A$. Compute cost($B,A$). If $B$ is not in the new list, insert it and set the minimum cost for $B$ to cost($B,A$). Set the backwards pointer to $A$ and fill in the total add and multiply counts. If $B$ is already in the new list, then set cost($B$) to the minimum of the existing cost($B$) and the newly computed cost($B,A$). Appropriately fill in the add and multiply counts for $B$. Update the backwards pointer in $B$ to point to $A$ if the newly computed cost was less than the existing cost($B$). When the old list is exhausted, store it on disk and copy the new list to the old list. Then start generating elements of $P_{n+2}$. At the termination of the procedure (when no more successors exist) the backwards pointers may be traced back from the winning maximal length cluster. By comparing cluster names along the trail, the optimum algorithm may be deciphered.

A copy of the FORTRAN source code for this procedure may be found in Appendix I. This program was debugged and run on a PDP11/55 under UNIX.
2.10. Examples

For simplicity, in the following three examples the cost function has been restricted to be a weighted linear combination of the number of adds and multiplies in each algorithm. All the important aspects of the solution procedure are illustrated in these simple examples. A scatter diagram is presented for each example, where a selection of algorithms from a vertically split class are plotted with the vertical axis representing the number of adds in the algorithm and the horizontal axis representing the number of multiplies. The square symbols indicate algorithms which are in the class generated by factoring each module only down to the pre-weave, multiply, and post-weave factors. These algorithms are therefore equivalent to hybrid WFTA and PFA type algorithms. The cross symbols indicate algorithms which can be formed using the factors stored in the author's data base of modules.

Not all the possible algorithms are plotted. Since the add and multiply per unit costs are both positive it is not necessary to plot any point which is both above and to the right of another point. Also, since the cost function is linear, it is not necessary to consider any points which lie in the interior of the smallest convex polygon containing all the points. This means that no algorithm lying above and to the right of any line segment connecting two other points will ever be optimal. The set of points which may be optimal (depending on the relative costs of adds and multiplies) lie on a convex (upwards) curve which goes from a minimum multiply algorithm
to a minimum add algorithm. The curve will always have a non-positive slope. One graphical method of finding the optimal solution, given a scatter graph, uses contours of constant cost. The contours of constant cost in these examples will be parallel straight lines with a slope equal to the negative of the ratio of the per unit multiply and add costs. Beginning with the contour passing through the origin (zero cost) if the contour is pushed outward, the first point which it intersects will be the minimum cost solution, because all other points will fall above or to the right of the contour passing through the solution point.

Example I is a length 35=5x7 DFT. The PFA and WFTA algorithms are plotted, and they are minimum add and multiply algorithms, respectively. One other algorithm is marked with a cross. Since that point is so close to the "corner" defined by the intersection of the minimum add and multiply limits, it is the optimum choice for a very wide range of relative add/multiply per unit costs. (The WFTA used in this example does have its pre-weave and postweave additions optimally ordered.)

Example II is a length 315=5x7x9 DFT. Again there is one point which lies close to the minimum add and minimum multiply limits.

Example III is a length 9009=7x9x11x13. The modules used had 5, 5, 7, and 7 factors, respectively. There are \(1.69 \times 10^{12}\) ways to construct a length 9009 algorithm using these factors. The optimizing program takes less than a minute to solve this problem running in FORTRAN
under UNIX on a DEC PDP11/55. A total of approximately 5000 clusters are generated during the solution procedure. The source code, including comments, contains 417 lines. To generate the data for this example, two passes were made using the optimizing program in Appendix I. First data was entered for each module's pre-weave, multiply and post-weave sections and the square solutions were found. Then the complete factored form of the modules was entered and the cross solutions were found. Each time, the add cost was first set to zero and a minimal multiply solution found. Then the multiply cost was set to zero and a minimal add solution was found (this will always be the PFA). Then for every two adjacent points already on the possible solution contour, the add/multiply costs were set equal to the slope between the points. This forced the optimization program to find all the intermediate solutions. When no new solutions were found between any adjacent pair of points, the procedure is terminated, and a complete set of optimal solutions has been found. The number of distinct solution possibilities was astonishingly low. In this last example, the WFTA was also plotted (with its additions minimized by pre-weave ordering) because it has the interesting property of not being a minimum multiply algorithm. The minimum multiply algorithm in this case has 39238 multiplies and is constructed by first nesting together the length 7, 9, and 13 algorithms and then combining that length 819 algorithm with the length 11 module in a PFA manner to get the final length 9009 DFT. The fully nested length 9009 WFTA has 43658 multiplications.
EXAMPLE I

MULTIPLIES
LENGTH 35 DFT
EXAMPLE II

MULTIPLIES
LENGTH 316 DFT

1200
EXAMPLE III

MULTIPLIES
LENGTH 8080 DFT

ADDs
3. PART II - INSIDE THE MODULES

It is well known that in the calculation of a PFA one must at some point deal with the issue of output permutations. This may be treated in many ways, two of which are presented in [16]. The problem is rooted in the application of multidimensional index maps to the input and output indices of a composite length DFT. If one chooses to use the Rudinian index map for the input index, then there exists an output index map under which the DFT calculation is separable into the familiar stages of a PFA where each stage represents the application of a number of identical short length DFT algorithms to the (multidimensional) rows of the data array. Unfortunately, in this circumstance the output map is not identical to the input map, so that if the algorithm is applied in-place the outputs will appear in permuted order. The remedy to this problem is to unscramble the outputs at the end of the computation, a tactic which is used in [16]. If, in order to force the output to appear in order, the Rudinian index map is applied to both the inputs and outputs of a composite length DFT then the calculation is still separable into stages as in the PFA, but the calculations at each stage are no longer DFT's, they are permuted DFT calculations. This is the derivation given in section 2.3. The permutation problem has been transferred from the overall algorithm output back into each individual DFT module. This method is also discussed in [16], where it is pointed out that the DFT module algorithms may be re-written to take into account this permutation, however the modules then lose
their generality, as the permutations required in each PFA are different.

Rothweiler [21] uses separate input and output indices to permute the outputs of each short DFT module in order to accomplish an overall data unscrambling. His technique involves no additional unscrambling data transfers but does require separate calculation of the output indices at execution time.

This section studies the abstract structure of DFT modules apart from the details of their derivation in order to shed new light on the nature of module algorithms, and the permutation problem in particular. Two main results are uncovered along the way. The first states that any permutation required of a DFT module in the PFA can always be implemented without altering the signal flow topology of that module by modifying the scaling coefficients used in the module. Thus in the context of a software module, the software algorithm does not need to be changed, only the data used by that algorithm. This comprehensive result follows from a Lemma concerning the factorization of rational polynomials. The second main result presents on the surface a method for finding the new coefficient sets to perform output permutations in DFT modules, but goes beyond that to provide a useful shortcut in the DFT modules design procedure.

3.1. Euclid’s Algorithm

This section is intended to familiarize the reader with certain algebraic concepts which will be used in the Lemma of section 3.4.
3.1.1. Fields

An algebraic field $H$ is a set of elements, together with two dyadic operators, generally denoted $+$ and $\cdot$ which satisfy several defining relationships, known as the field axioms. The only fields used here will be the familiar fields of complex numbers $\mathbb{C}$, real numbers $\mathbb{R}$ and rational numbers $\mathbb{Q}$. In those fields the $+$ and $\cdot$ (add and multiply) operators assume their normal significance. Henceforth $H$ will refer to one of these three fields. One interesting algebraic object which is closely related to a field is the ring of polynomials $H(z)$. The set $H(z)$ is composed of all polynomials of finite degree in the indeterminate $z$ with coefficients in $H$.

$$H(z)=\left\{ \sum_{i=0}^{N} p_i z^i \mid \begin{array}{c} p_i \in H \\ p_N \neq 0 \end{array} \right\}$$

An element of $H(z)$, $\sum_{i=0}^{N} p_i z^i$, is referred to as $p(z)$, or simply $p$, a polynomial in the variable $z$ where $N$ is the degree of $p(z)$, written $\deg(p)$. The degree of the zero polynomial is undefined. A monic polynomial has a leading coefficient, $p_N$, of unity. The $+$ operation for $H(z)$ is normal polynomial addition, adding the coefficients of various powers of $z$ on a term by term basis. The $\cdot$ operation is normal polynomial multiplication, where all the terms are first expanded using the distributive law of $\cdot$ over $+$ and the powers of $z$ are collapsed according to the rule $z^i \cdot z^j = z^{i+j}$. Finally terms with like exponents are collected. Polynomial multiplication with
H equal to C, R or Q is always commutative. H(z) satisfies almost all of the field axioms except that there is no inverse in H(z) for multiplication by a polynomial of degree greater than zero. This is because the degrees of polynomials add when they are multiplied together, and since no polynomial can have a negative degree (by definition) once the degree is shifted above zero it can never be brought back down.

2.1.2. Dividing

If for polynomials p, s, t ∈ H(z) the relationship p = s • t holds then s is said to divide p, written s | p. Also t | p. Euclid’s algorithm, which is equivalent to the well known process of long division, is a means of ascertaining whether or not s | p for any s and p. Given p and s in H(z) the Euclidean algorithm will always find q and r also in H(z) with \( \deg(r) < \deg(s) \) such that

\[ p = q • s + r \]

If r = 0 then it may be concluded that s | p. The polynomial q is called the quotient and r the remainder of the division process. For example, \( z^4 - 1 = (z^3 - z^2 + z - 1) • (z + 1) \) so clearly \( (z + 1) | (z^4 - 1) \). On the other hand, \( z^4 + 1 = (z^3 - z^2 + z - 1) • (z + 1) + 2 \) so that \( (z + 1) \) does not divide \( z^4 + 1 \). That is not surprising, as it may easily be checked that the only root of \( (z + 1) \), namely \(-1\), is not a root of \( z^4 + 1 \).
3.1.3. **Greatest Common Divisors**

For two polynomials \( p \) and \( s \) in \( H(z) \) there always exists a unique monic polynomial called the **greatest common divisor (GCD)** of \( p \) and \( s \), also denoted \( (p,s) \). The polynomial \( (p,s) \) is defined by three relationships:

1) \( (p,s) \mid p \)
2) \( (p,s) \mid s \)
3) Whenever \( g \mid p \) and \( g \mid s \) then \( g \mid (p,s) \) also.

The last property explains the notion of a **greatest common divisor**. If \( p,s \in H(z) \) then \( (p,s) \) is also in \( H(z) \) and may always be generated as a combination of \( p \) and \( s \):

\[
(p,s) = r*p + t*s
\]

for some polynomials \( r \) and \( t \) in \( H(z) \). This property is a consequence of the Euclidean algorithm which may be used to calculate the GCD, and is directly related to the fact that in the division process the remainder and quotient polynomials are always members of the same polynomial ring as the divisor and dividend polynomials. For \( p \) and \( s \) in \( H(z) \) to be relatively prime means that their GCD is 1. Equivalently, \( p \) and \( s \) are relatively prime if and only if there exist polynomials \( r \) and \( t \) also in \( H(z) \) such that:

\[
r*p + t*s = 1
\]

**Examples**

\[(z^4-1,z+1) = z+1 \quad ; \quad r=0 \quad t=1\]

\[(z^4-1,z^3-1) = z-1 \quad ; \quad r=1 \quad t=-z\]
\[(z^4 + 1, z + 1) = 1 \quad ; \quad x = 1/2 \quad t = -(z^3 - z^2 + z - 1)/2\]

3.2. **Primitive Nth Roots of Unity**

An Nth root of unity is defined (for integer \(N\)) to be a root of the equation \(z^N - 1 = 0\). Consequently, if \(w\) is an Nth root of unity then \(w^N = 1\). All Nth roots of unity may be found in \(\mathbb{C}\), the field of complex numbers. Their magnitudes are all precisely 1 and their angles must be integral multiples of \(2\pi/N\). The number of Nth roots of unity is always exactly \(N\), and their values are given by \(e^{i2\pi k/N}\) for \(0 \leq k < N\). The only Nth roots of unity to be found in \(\mathbb{Q}\) or \(\mathbb{R}\) are the roots 1 and \(-1\). Note that 1 is an Nth root of unity for any \(N\).

For some particular \(N\), the set of all Nth roots of unity form an algebraic group under the operation of (complex) multiplication. This means, among other things, that for any two Nth roots of unity, \(w_1\) and \(w_2\), the product \(w_1w_2\) must also be an Nth root of unity. This property may be checked by noting that \((w_1w_2)^N = w_1^Nw_2^N = 1^N1 = 1\).

For \(w\) an Nth root of unity one may study the **orbit** of \(w\). That is the collection of all numbers \(w^k\) for (possibly negative) integer values of \(k\). The elements of the orbit of \(w\) constitute a subset of all Nth roots of unity also happen to form a subgroup of the algebraic group of Nth roots of unity. The number of distinct elements in the orbit of \(w\) is called the **order** of \(w\). A **primitive** Nth root of unity is a root whose order is precisely \(N\).

Four properties of the order of \(w\), denoted \(O(w)\), will be used in
later sections:

Let \( w \) be an \( N \)th root of unity.

Let \( M=O(w) \) be the order of \( w \).

1) \( M = 1 \)

2) \( M \mid N \)

3) if \( (k,M)=1 \) then \( w^k \) also has order \( M \)

4) There are \( \phi(M) \) distinct numbers of order \( M \), where \( \phi(M) \) is Euler's totient function, and is equal to the number of integers \( k \) less than \( M \) for which \( (k,M)=1 \).

Proofs

1) Let \( u \) be the smallest integer such that \( w^u = 1 \). Since \( w^N = 1 \), \( u \) exists and is known to be less than (or equal to) \( N \). The orbit of \( w \) is periodic with period \( u \) \( (w^{k+u} = w^k w^u = w^k 1 = w^k) \) so the order of \( w \), equal to the number of elements in the orbit of \( w \), must be less than or equal to \( u \). On the other hand, if the powers of \( w \) up through \( u \) were distinct then there would be at least \( u \) elements in the orbit of \( w \) and it could be concluded that \( u=M \), and finally that \( w^M = 1 \). Suppose that the positive powers of \( w \) up through \( u \) were not distinct, but rather that for some \( i \) and \( k \) both less than \( u \) (assume \( i>k \)) that \( w^i = w^k \). Then the calculation \( w^{i-k} = 1 \) would contradict the definition of \( u \) because \( (i-k) \) is less than \( u \). Therefore there are at least \( u \) distinct elements in the orbit of \( w \) and the proof is finished.

***

The proof of (1) provides an alternate definition of the order of a
root of unity as the smallest positive integer \( k \) for which \( w^k = 1 \).

2) Compute the GCD of \( M \) and \( N \). It must (see section 3.1.3) be equal to \( rN + sM \) for some integers \( r \) and \( s \), which implies that:

\[
\underbrace{w^k}_{(N,M)} = \underbrace{w^{rN+sM}}_{w^{rN}} \underbrace{w^{sM}}_{w^s} = 1
\]

Since \( M \) is the smallest integer which annihilates \( w \), it must be true that \( (N,M) \geq M \) while on the other hand the definition of GCD demands that \( (N,M) \leq M \) so it may be concluded that \( (N,M) = M \). The GCD always divides each of its arguments, so \( M \) necessarily divides \( N \).

***

3) First check \((w^k)^{M} = (w^M)^k = 1\) so that \( \theta(w^k) \leq M \). On the other hand, by the properties of the GCD there exist integers \( r \) and \( s \) such that \( rk + sM = 1 \), or equivalently \( rk = 1 - sM \). Denote \( \theta(w^k) \) by \( L \) and substitute for \( rk \) in the equation:

\[
l = 1 = \underbrace{(w^L)_{w}^r}_{w} \underbrace{w^{rkL}}_{w} \underbrace{(1-sM)L}_{w} \underbrace{L}_{w} - \underbrace{sM}_{w} \underbrace{L}_{w}
\]

By the definition of \( M \), \( L \leq M \) so it may be concluded that \( M = L \).

***

4) If \( \theta(x) = M \) then \( x^M = 1 \) and therefore \( x = e^{i2\pi k/M} \) for some positive \( k \) less than \( M \). Let \( w = e^{i2\pi / M} \). Some subset of the orbit of \( w \) will be

---

\(^3\) For an integer \( k \) to annihilate an element \( w \) of a multiplicative group means that \( w^k = 1 \).
the set of all numbers of order $M$. All that needs to be checked is the 
order of all the elements in the orbit of $w$.

Let $L = O(w^k)$. By the properties of the GCD, $rk + sM = (k, M)$ for some 
integers $r$ and $s$. Calculate:

$$1 = w^{kL} = w^{rkL} = w^{L((k, M) - sM)} = w^{L(k, M)}$$

and since $w^n \neq 1$ for $n$ less than $M$ we must have $L \leq \frac{M}{(k, M)}$. On the other 
hand,

$$(w^k)^{(k, M)} = (w^{(k, M)})^k = 1$$

so $L$ must be less than or equal to $M/(k, M)$ from which it is concluded 
that $L = M/(k, M)$.

Therefore if $(k, M) = 1$ then $L = M$ and if $(k, M) \neq 1$ then $L$ is less than $M$.
The number of integers $k$ for which $O(w^k) = M$ is equal to the number of 
integers $k$ less than $M$ for which $(k, M) = 1$, defined to be $\varphi(M)$.

***

3.3. Cyclotomic Polynomials

Cyclotomic polynomials are crucial to the Lemma in section 3.4.

More material on cyclotomic polynomials and their properties is 
available in [19] and [8].

Cyclotomic polynomials are actually a family of polynomials indexed 
by a single integer value. The $N$th cyclotomic polynomial is defined 
to be:
\[ C_N(z) = \frac{\phi(N)}{\prod_{i=1}^{\phi(N)} (z-w_i)} \]

where \( \phi(N) \) is Euler's totient function and \( w_1, w_2, \ldots, w_{\phi(N)} \) are the primitive \( N \)th roots of unity. Since each primitive \( N \)th root of unity is by definition also a root of the equation \( z^N - 1 = 0 \) the whole polynomial \( C_N \) must be a factor of \( z^N - 1 \). Two facts from [19] about cyclotomic polynomials which will not be proved here are:

1) \( C_N \) has integer coefficients for all \( N \)
2) \( C_N \) is always irreducible over the rationals

The second statement means that there is no rational monic polynomial \( p \) (\( p \) is in \( \mathbb{Q}(z) \)) such that \( p | C_N \) except for 1 and \( C_N \) itself. Obviously \( C_N \) is factorable over \( \mathbb{C}(z) \), the factors are explicitly listed in the definition.

3.4. A Lemma

This section is devoted to proving the following Lemma:

Let \( p \) be a rational polynomial.

Let \( w_N \) be some primitive \( N \)th root of unity.

IF \( p(w_N) = 0 \) THEN

for every integer \( k \) such that \( (k,N) = 1 \)

the condition \( p(w_N^k) = 0 \) holds.

From section 3.2, the collection of all \( w_N^k \) for \( (k,N)=1 \) is equivalent to the set of all primitive \( N \)th roots of unity. If \( p(a_1) = 0 \) for \( a_1, a_2, \ldots, a_{\phi(N)} \) equal to the primitive \( N \)th roots of unity then it must
be true that $C_N | p$. Conversely if $C_N | p$ then $p$ is necessarily zero at each of the primitive $N$th roots of unity. Therefore it suffices to prove:

If $p(w_N) = 0$ then $C_N | p$.

Consider the GCD of $p$ and $C_N$. The GCD will be a member of $Q(z)$ because both $p$ and $C_N$ are in $Q(z)$. Combining the facts that $C_N$ is irreducible over $Q(z)$ and that $(p, C_N)$ must divide $C_N$ it is apparent that $(p, C_N)$ must equal either $1$ or $C_N$. Because $p(w_N) = 0$ it follows that $(z - w_N) | p$ and by definition $(z - w_N) | C_N$. The properties of the GCD then imply that $(z - w_N)$ must also divide $(p, N)$, so that $(p, N)$ cannot be $1$. This forces $(p, N) = C_N$ and finally the result $C_N | p$ follows as a consequence of the fact that the GCD always divides both of its arguments.

***

In order to fully treat power of two length modules in section 3.7 the following corollary will be proved here:

Let $p$ be a polynomial of finite degree whose coefficients

are complex rationals (i.e. $x + jy$ where $x, y \in Q$).

Let $N$ equal $2^b$ for some integer $b > 1$.

Let $w_N$ equal $e^{-j 2\pi/N}$.

IF $p(w_N) = 0$ THEN

for every integer $k$ such that $\langle k \rangle_4 = 1$

the condition $p(w_N^k) = 0$ holds.

For notational convenience the set of points $w_N^k$ for $\langle k \rangle_4 = 1$ will be
denoted by $S$.

This corollary may be proved by constructing an auxiliary polynomial $\overline{p}$. It is first shown that $p$ equals $\overline{p}$ at every point in $S$. In particular, $\overline{p}$ will have a zero at $w_N^{-1}$. By construction, $\overline{p}$ will be a real rational polynomial and since it has a zero at $w_N^{-1}$, the Lemma may be invoked to show that $\overline{p}$ is zero at every point in $S$, establishing the fact that $p$ also has zeroes at every point in $S$.

Here is the construction for $\overline{p}$.

Let $p(z) = \sum_{i=0}^{M} (x_1 + iy_1)^i z^i$

Construct $\overline{p}$:

$\overline{p}(z) = \sum_{i=0}^{M} (x_1 - z^{-N/4} y_1) z^i$

When $<k>_4=1$ the quantity $(w_N^{-1})^{N/4}$ will equal $-j$, forcing $\overline{p}$ equal to $p$ over all points in $S$. Furthermore, $\overline{p}$ is a real rational polynomial.

***

An alternate proof is based on the irreducibility of the factorization $G_N = (z^{N/4} + j)(z^{N/4} - j)$ over the complex rationals (remember that $N=2^b$). Using a process which parallels the proof of the Lemma above the GCD of $p$ and $(z^{N/4} + j)$ is formed. Combining the facts that these two polynomials share a root $(w_N^{-1})$ and that $(z^{N/4} + j)$ is irreducible over the complex rationals it may be concluded that $p$ is divided by $(z^{N/4} + j)$. Since the roots of $(z^{N/4} + j)$ are exactly the set $S$, the result follows.
3.5. **Canonical Forms**

The canonical form for Winograd's length N DFT module as presented in [1] is:

\[ y = C(Ax \Theta Bw) \]  \hspace{1cm} (1)

where \( x \) is the length N input vector, \( \Theta \) is a low precedence component by component product operator, and \( w \) is a length N vector containing all the Nth roots of unity. In this case \( \Theta \) multiplies together the components of the vectors \( Ax \) and \( Bw \) to produce a new vector which is then transformed by \( C \). The matrices \( A, B \) and \( C \) are all real and rational. In any reasonable implementation of such a module the matrix product \( Bw \) is precomputed and stored as a vector of coefficients \( y \). The algorithm actually executed is then:

\[ y = C(Ax \Theta y) \]  \hspace{1cm} (2)

If one considers the set of all possible DFT algorithms of form (2) with \( A \) and \( C \) rational matrices it is clear that this set contains all algorithms of type (1). It is possible that there might be algorithms of form (2), however, which cannot be represented in form (1). That is, given some general algorithm of type (2), what reason is there to believe that there exists any rational \( B \) with \( y = Bw \)? The balance of this section is devoted to proving that for any length N DFT algorithm of form (2) with rational \( A \) and \( C \) matrices there always exists a rational matrix \( B \) such that \( C(Ax \Theta Bw) \) is also a valid length N DFT algorithm. In this sense, the formulations (1) and (2) are equivalent. This is a convenient and satisfying result. The
permutation properties of DFT modules which are derived in section 3.6 depend implicitly on the existence of a matrix \( B \) such that \( y = Bw \), but the relationship between forms (1) and (2) investigated here indicate that those properties are actually a consequence of the structure of \( A \) and \( C \), in that if the module works at all, the existence of an appropriate \( B \) is guaranteed. In this light the significance of the \( B \) matrix is considerably reduced, and in section 3.7 the \( B \) matrix is completely removed from the module design process.

To lend more meaning to the equivalence argument it should first be pointed out that the set of all values \( Bw \) for \( B \) rational does not contain the set of all possible complex vectors \( y \). This fact may be intuitively understood by observing that the set of all rational \( B \) matrices, and hence the set of all vectors \( Bw \), is countable whereas the set of all complex vectors \( y \) is not. It is indeed a rare coincidence that for any DFT algorithm, each element of \( y \) can be generated as a rational combination of \( N \)th roots of unity.

To begin, express the system (2) as a nested summation:

\[
y(k) = \sum_{j=1}^{N} c_j \sum_{i=0}^{N-1} a_{ji} x(i)\]

Where \( N \) is the output dimension of the matrix \( A \), also equal to the length of the vector \( y \). The matrix product \( Ax \) is bracketed, followed by the componentwise multiplication with \( y \) which takes place across the index \( j \) and then the product with \( C \) is implemented, summing across \( j \) and transforming to the output index \( k \).
Interchanging the order of summations:

\[ y(k) = \sum_{i=0}^{N-1} x(i) \left( \sum_{j=1}^{M} v_j \sum_{j=1}^{a} k_j a_{ji} \right) \]

Now the quantity \( \sum_{j=1}^{M} v_j c_{kj} a_{ji} \) behaves like an \( N \times N \) matrix acting on \( x(i) \). In order for the overall transformation to be equivalent to a length \( N \) DFT it must be true for all \( k \) and \( i \) that:

\[ \sum_{j=1}^{M} v_j c_{kj} a_{ji} = w_{N}^k \]

where \( w_{N} = e^{-j2\pi/N} \). This expression is called the fundamental equation of the DFT module and represents an \( N^2 \) dimensional linear approximation problem involving the \( M \) signal vectors \( c_{kj} a_{ji} \) for \( j=1,2,\ldots,M \) and the \( M \) weights \( v_j \). The object is to make the weighted signal (the matrix of the net transformation) look like \( w_{N}^k \), the DFT. The problem is an overdetermined set of \( N^2 \) linear equations in the \( M \) unknowns \( v_j \) and may be abstractly represented by:

\[ Qv = D \]

where \( Q \) is a rational \( N^2 \times M \) matrix and \( D \) is the \( N^2 \) length vector corresponding to an "unwrapped" DFT matrix. Each entry of the vector \( D \) is a power of \( w_{N} \). The underbar has been dropped on the vector \( v \).

Given that the original system was a valid DFT, the above equation is known to be soluble. We want to find a solution to this equation where \( v \) is the product of a rational matrix times \( w \). The matrix \( Q \) is \( N^2 \times M \) which is certainly not square or necessarily invertible. In
order to solve this system first throw out any columns for which the corresponding element of \( \mathbf{v} \) in the initial algorithm is zero, and also throw away the zero elements of \( \mathbf{v} \). This sounds silly (why should there be zero values of \( \mathbf{v} \)?) but it will be important later. Similarly throw out all linearly dependent columns of \( \mathbf{Q} \), and remove the corresponding elements of \( \mathbf{v} \). This process leads to a reduced system of full column rank:

\[
\mathbf{Q}\mathbf{v} = \mathbf{D}
\]

which is guaranteed to have some solution (\( \mathbf{D} \) is in the span of \( \mathbf{Q} \)). This happens because the effective span of \( \mathbf{Q} \) was not changed by throwing out dependent columns and columns that weren't used. Not only is the solution guaranteed to exist, it will also be unique (\( \mathbf{Q} \) has full column rank). The matrix \( \mathbf{Q} \) will have a left inverse, \( \mathbf{Q}^{-1} \), which can used to premultiply each side of the previous equation, resulting in:

\[
\bar{\mathbf{v}} = \mathbf{Q}^{-1}\mathbf{D}
\]

Since it was known that \( \mathbf{D} \) is in the span of \( \mathbf{Q} \), this will yield the proper solution. Additionally, the matrix \( \mathbf{Q}^{-1} \) will be rational, and it is almost the \( \mathbf{B} \) matrix that is being sought. The vector \( \mathbf{D} \) is a listing of powers of \( \mathbf{w}_N \) and can be represented \( \mathbf{D} = \mathbf{P}\mathbf{w} \) for some "permutation" matrix \( \mathbf{P} \). Substituting for \( \mathbf{D} \) above we get:

\[
\bar{\mathbf{v}} = (\mathbf{Q}^{-1}\mathbf{P})\mathbf{w}
\]
The B matrix may now be identified as \((Q^{-1}P)\). Extra rows of zeroes may need to be added to B to produce the coefficients of \(v\) in their original format.

The whole padding with zeroes phenomenon is actually quite rare in that it implies that some of the original coefficients in \(v\) were superfluous. In a well designed algorithm this would not happen, but the concept is useful in the following context.

In the majority of DFT modules the coefficients \(v\) are not general complex numbers, but are each either purely real or purely imaginary. The same construction used above can be used to find a \(B\) for which \(C(Ax \Theta Bw)\) is a DFT and where additionally each new coefficient is purely real or imaginary according to the disposition of its corresponding original coefficient. First partition the system into real and imaginary subsystems, separating the real and imaginary coefficients into two distinct problems. The real problem will be to approximate a cosine transform using the real coefficients and the imaginary problem will be to approximate a sine transform using the imaginary coefficients. The real coefficients will turn out to be rationally weighted combinations of cosines and the imaginary coefficients will be rationally weighted sums of sines. As in the original formulation the cosines and sines can be represented as rational combinations of the Nth roots of unity and the complete vector \(v\) will be some rational matrix \(B\) times the vector \(w\). Because the various real and imaginary parts of \(v\) were separated out at the beginning of the
procedure and then recombined at the end the "new" coefficients will be correspondingly real or imaginary.

It should be pointed out that in a well constructed algorithm, there will be no wasted columns of \( Q \) and the "new" coefficients will by the uniqueness of the solution be required to be precisely the same as the "original" coefficients. In that case, what we have shown is that the original coefficients must have been rational combinations of the \( N \)th roots of unity in the first place. This is a subtle point, but it becomes important when one is given an algorithm with no derivation, and wishes to apply the results of section 3.6. From an aesthetic viewpoint it is interesting to know that the only useful way to construct an algorithm of type (2) is to use coefficients in \( Q(\omega_N) \), the set of all rationally weighted sums of powers of \( \omega_N \).

3.6. Permuting Outputs

The PFA formulation in section 2.3 indicated that the computations needed to piece together a large in-place PFA are not DFT modules, but are instead DFT modules with permuted outputs. The desired module calculation is:

\[
y'(k) = \sum_{i=0}^{N-1} x(i) w_N^{Di k}
\]

where \( w_N = e^{-j2\pi/N} \) and \( D \) is an integer such that \( (D,N)=1 \). This computation is rightly called a permuted DFT because it could be calculated by first evaluating a normal DFT, storing the result in \( y(k) \) and then transferring that data to a final output vector according to
the rule \( y'(k) = y(Dk) \) where the index of \( y \) is evaluated modulo \( N \). The index map \( k \rightarrow Dk \) is a permutation if and only if \( (D, N) = 1 \).

This section asserts the claim that given any length \( N \) DFT module of the form \( C(Ax@v) \), and any integer \( D \) with \( (D, N) = 1 \) there exists a vector of new coefficients \( v' \) such that the algorithm \( C(Ax@v') \) will directly compute \( y'(k) = y(Dk) \). No modifications to the control structure of the algorithm are necessary. The new coefficients entirely eliminate the need for "unscrambling" outputs in the PFA. The proof of this claim rests on the theory of cyclotomic polynomials.

An example will be given to enhance the meaning of this proof. Consider a length \( N \) DFT module with purely real coefficients \( v_r \) and purely imaginary coefficients \( v_i \). For any \( N, N-1 \) will be relatively prime to \( N \). Let the desired output permutation be \( y'(k) = y((N-1)k) \), where the \( y \) index is evaluated modulo \( N \). This is equivalent to \( y'(k) = y(-k) \), a reversal of the output sequence. By the properties of the DFT (for real input sequences) \( y(-k) = y^*(k) \) where \( ^* \) denotes complex conjugation. Conjugating the outputs is a simple operation which may be accomplished by negating all the values \( v_i \). The new algorithm with all its imaginary coefficients negated will have a reversed output. For a length 9 DFT module it is not so simple to see how to change the coefficients so as to produce an output permutation of say \( y'(k) = y(4k) \). The proof below, however, asserts that coefficients to perform such a permutation do indeed exist.

The proof starts with the fundamental equation of the DFT module
presented in section 3.5.

\[
\sum_{j=1}^{M} v_j c_{kj} a_{ji} = \frac{k_i}{w_N}
\]

Given that the module works (this equation can be satisfied by some vector \( v \)) we know from section 3.5 that there exists a rational matrix \( B \) with \( v = Bw \) such that \( C(Ax@v) \) is a valid DFT. Substituting for \( v_j \) in the fundamental equation gives

\[
\sum_{j=1}^{M} c_{kj} a_{ji} \sum_{h=0}^{N-1} b_j h w_N = -\frac{k_i}{w_N} = 0
\]

This is a set of \( N^2 \) equations, for \( k \) and \( i \) each ranging from 0 to \( N-1 \). Each expression may be represented as a rational polynomial:

\[
p_{ik}(w_N) = 0 \quad \text{for all } i, k
\]

By the Lemma of section 3.4 it must also be true that for any integer \( D \) with \((D,N)=1\):

\[
p_{ik}(w_D^N) = 0 \quad \text{for all } i, k
\]

which implies that

\[
\sum_{j=1}^{M} c_{kj} a_{ji} \sum_{h=0}^{N-1} b_j h w_N = \frac{Dk_i}{w_N}
\]

so that the system \( C(Ax@v') \) using the coefficients \( v'(j) = \sum_{h=0}^{N-1} b_j h w_n \)

will compute
\[ \sum_{i=0}^{N-1} x(i) \sum_{j=1}^{M} c_{kj} a_{ji} \sum_{h=0}^{N-1} b_{jh} w_N^h \]

\[ = \sum_{i=0}^{N-1} x(i) w_{Dki}^N \]

as desired.

Section 3.7 discusses means of actually computing these new coefficients.

3.7. Calculating Coefficients

This is the practical section of Part II. Summarizing section 3.1-3.6 it has been shown that any length N DFT module of type 2 can be reconfigured to have permuted outputs of the form \( y'(k) = y(kD) \) for \((D,N) = 1\) by changing the coefficient vector \( y \). In the proof of that claim, a matrix \( B \) was constructed with the original coefficient vector \( y \) equal to \( By \). Then the new coefficients \( y' \) could be computed as \( y' = Bw' \) where \( w'(n) = e^{-j2\pi nD/N} \).

This section presents a method for computing the coefficients \( y' \) directly from the A and C matrices without using the original coefficient set. The approach taken here is to start with the fundamental equation of the DFT module (from section 3.5)

\[ \sum_{j=1}^{M} v_j c_{kj} a_{ji} = w_N^i \]

(where \( w_N = e^{-j2\pi/N} \)) and attempt to directly solve
\[ \sum_{j=1}^{M} v' j c_{k} j^a j = w_{N}^{D^{ki}} \]

for some integer \( D \) with \( (D, N) = 1 \). One obvious solution method is to select \( M \) linearly independent equations out of the \( N^2 \) possibilities and solve for the unique \( v' \) which satisfies those equations. That unique \( v' \) will also satisfy the rest of the \( N^2 \) equations, because existence of a solution is guaranteed in this case.

In order to avoid the problem of finding \( M \) linearly independent equations we may instead solve the equivalent least squares problem:

\[ \text{minimize: } E = \sum_{k^{i}j} \left( \sum_{j=1}^{M} v' j c_{k} j^a j - w_{N}^{D^{ki}} \right)^2 \]

The error \( E \) is expected to go to exactly zero. The advantages to this method are twofold:

1) All \( N^2 \) equations are simultaneously solved without worrying about which ones are linearly independent.

2) There exists a LINPACK Q-R subroutine [25] for effectively solving this problem.

If the LINPACK Q-R routine detects fewer than \( M \) linearly independent equations it will force some of the coefficients to zero until it gets a unique solution. Another popular possibility for forcing a unique solution in this type of problem is to find the solution vector with the smallest norm. That would be a fine solution, but it is better in this case to set the some of the coefficients to zero.
because that reduces the execution time workload of the module algorithm.

This method has been successfully applied to a real design problem encountered by the author in the construction of a length 19 DFT module [14]. A FORTRAN routine was first generated which implemented the preweave reduction procedure A in several passes, performing the required polynomial reductions in stages [8]. The technique of "transposing the tensor" [26] was used to derive the output algorithm C, which is the transpose of the A procedure. This can be accomplished working directly from the listing of the preweave algorithm or from flowcharts of its operation. The matrices A and C were never actually computed, and the derivation of the reconstruction procedure, which was "transposed" into position B, was never attempted. It did not have to be, because the coefficients v could be found directly from A and C. Given the working subroutines for A and C, a prototype module was constructed which used a coefficient vector v that could be specified at execution time. By testing the algorithm with various inputs and coefficient vectors all the terms $c_{kj}a_{ji}$ could be measured from the prototype module without ever seeing A or C explicitly. To measure $c_{Kj}a_{jI}$ set the coefficient vector v to all zeroes except the Jth position ($v(j)=\delta(j-J)$) and set $x(i)=\delta(i-I)$ [where $\delta$ is the unit impulse, $\delta(0)=1$; otherwise $\delta(n\neq0)=0$]. The Kth output will equal $c_{Kj}a_{jI}$

$$\sum_{j=1}^{N} c_{kj} \delta(j-J) \sum_{i=0}^{N-1} a_{ji} \delta(i-I)$$
\[ \sum_{i=0}^{N-1} a_{ji} \delta(i-I) \]

and the Kth output is \( c_{kj} a_{ji} \) as desired.

Next the Q-R algorithm was used to solve for the coefficient vector which minimized \( E \). That vector was then written into the prototype module source listing as constants and the variable coefficient array reference removed from the calling sequence, completing the module.

This basic procedure must be modified slightly. In the construction of a DFT module it is known that some of the constants in \( y \) will be small integers and those multiplies may be replaced with additions, as is frequently done. The coefficients associated with those additions are no longer controllable and it also is no longer possible to directly examine \( c_{kj} a_{ji} \). Multiplications by controllable constants could be placed on those links and the resulting system solved but that might necessitate changing some of the module program code if those constants must be modified to attain some desired permutation.

If it turns out to be true that the uncontrollable rational constants do not have to be changed as a function of the desired permutation, then the effect of the uncontrollable constants can be subtracted from the desired permuted DFT and the remaining constants used to fit that residual. The residual fitting problem will otherwise be identical to the full coefficient solution procedure. Assuming for the
moment that the uncontrollable constants will not require modification, the following procedure describes how to compute the residual and how to solve the remaining system.

Separating the coefficient vector into fixed predetermined (also rational) constants and other controllable constants the fundamental equation decomposes into:

\[ \sum_{j_{\text{fixed}}} v_{j} c_{kj} a_{ji} + \sum_{j_{\text{variable}}} v_{j} c_{kj} a_{ji} = w_{N} \]

and the system to be solved is:

\[ \sum_{j_{\text{variable}}} v_{j} c_{kj} a_{ji} = w_{N} - \sum_{j_{\text{fixed}}} v_{j} c_{kj} a_{ji} \]

The expression \( \sum_{j_{\text{fixed}}} v_{j} c_{kj} a_{ji} \) is just the non-variable part of the transformation and may be measured by setting all controllable constants to zero and applying test inputs \( \delta(i-I) \) for \( I=0,1,\ldots,N-1 \). Let the notation \( T_{ki}^{(y)} \) represent the \( K \)th output with the input set to \( \delta(i-I) \) and the vector of controllable constants set to \( y \). The uncontrollable part of the transformation, as a function of \( K \) and \( I \), is equal to \( T_{ki}^{(0)} \) The quantities \( c_{kj} a_{ji} \) may be measured using the identity \( c_{kj} a_{ji} = T_{ki}^{(\delta(j-I))} - T_{ki}^{(0)} \). This expression subtracts out the effect of the uncontrollable constants.

The point about non-controllable rational constants not needing to be changed as a function of the desired output permutation needs to be checked carefully. On the surface, there is absolutely no guarantee that, say, the nice coefficients of 1 and 5/4 in the length 5 DFT
module will not have to be modified to effect some output permutation. Fortunately, it can be proved that real rational coefficients in DFT modules never have to be changed to effect any allowable output permutation. The case of power of two modules is complicated by the fact that there are coefficient equal to j and -j which are rational, but not real. These coefficients are frequently implemented by specialized code which is not controllable. It can be proved that two versions of each power of two module are sufficient for reproducing any output permutation. One module can generate all permutations with D equal to <1> mod 4 and the other generates permutations with D equal to <3> mod 4. The two versions differ in that the outputs (or inputs) of the second module are reversed, so that if for some set of coefficients the first module computes a DFT permutation of D the other will compute a permutation of -D using the same coefficients. It can also be shown that purely real coefficients remain real and imaginary coefficients remain purely imaginary after the permuting transformation.

To prove these results, recall from earlier sections that the original coefficient vector is given by \( y = B w \) and that the new coefficients for permutation D are given by \( y' = B w' \) where \( w'(n) = w_{\frac{nD}{N}} \) and D is relatively prime to N. For each coefficient, the corresponding row of B may be viewed as a polynomial with the coefficient satisfying the relation \( v_i = p_i(w) \). This is true only due to the very special nature of the vector \( w \). Because the entries of B are real and rational and by assumption the coefficient \( v_i \) is also real and rational the
expression $v_i - p_i$ (a scaler minus a polynomial) is a real rational polynomial with a root at $w_N$. The Lemma of section 3.4 indicates that the expression will therefore have roots at all primitive $N$th roots of unity, so that for any $D$ relatively prime to $N$, $v_i = p_i(w_N^D)$. But the righthand side of this equation is just $v_i'$ so that the result $v_i' = v_i$ follows.

Now if $v_i$ is complex, but still rational, and $N$ is a power of two, then the argument is similar except that the Corollary from section 3.4 must be used leading to the equality of $v_i'$ and $v_i$ only in the restricted cases where $<D> \text{ mod } 4 = 1$. It should be pointed out that there are at least three ways to generate the complimentary power of two modules with reversed outputs from regular modules.

1) Reverse the outputs by changing the program structure
2) Reverse the inputs
3) Conjugate all the uncontrollable coefficients

Next check what the permuting transformation does to a purely real coefficient. Starting with the relation $v_i = v_i^*$ and substituting in the polynomial expression for $v_i$ yields $p_i(w_N) = [p_i(w_N)]^*$ and since $p_i$ is a real polynomial this is also equal to $p_i(w_N^{-1})$. By the Lemma this last equality implies that $p_i(w_N^k) = p_i(w_N^{-k})$ which establishes the final equality $v_i' = v_i^*$. The relationship $v_i = -v_i^*$ can be used to show a purely imaginary coefficient is always transformed into another imaginary coefficient.
In conclusion, the module program storage requirements for a general purpose PFA routine can be reduced to a single copy of each odd-length module, two copies of each power of two module and a coefficient vector corresponding to each possible permutation for each module.

A program GEN is included in Appendix II which was compiled with DFT prototype modules and then run to generate the data \( c_{kj}a_{ji} \) for each module. This data was then run through SOLVE which solved for the coefficients \( v_j \) that satisfied the fundamental equations of the DFT modules. The variable \( D \) was read by the second program and the resulting coefficients produced a module that computed \( \sum_{i=1}^{N} x(i) v_k^{Dki} \).

These programs can be used for several functions:

1) Initial algorithm design; eliminates the need for deriving the matrix \( B \), which in "transposed tensor" designs is the most complicated part of the design process.\(^4\)

2) Updating a given algorithm to permute its outputs.

3) Computing the coefficients of a given module to higher precision.

Once a module is generated the \( B \) matrix is not generally included with the module listing. This program permits recomputation of the coefficients without prior knowledge of \( B \).

---

\(^4\) The matrix \( B \) is the only place where Chinese Remainder Theorem polynomial reconstruction calculations take place. Using this method, no reconstructing polynomials ever have to be calculated.
4. Discussion

The problem studied in Part I is of a very specific nature, but can be applied to a wide variety of circumstances. The optimization procedure is suitable for finding a minimum cost algorithm from some vertically split class, where the cost is a function of the number of adds, multiplies, data memory read/write cycles, the data storage size, and to some degree the amount of indexing overhead. A vertically split class can represent algorithms other than DFT's. If the modules are pre-weave sections of short-length DFT algorithms and each module consists of only one factor, then the problem of ordering the additions in a conventional WFTA can be solved. If each pre-weave module is further factored, but the multiply and post-weave sections are still not included, then the optimization procedure will generate split nested DFT algorithms. If each module is a complete DFT factored into exactly three pieces, pre-weave, multiply, and post-weave, then the conventional WFTA-PFA hybrid tradeoffs will be solved. If each module is a complete DFT but further factored then the procedure examines many hybrid possibilities between the PFA and split nested algorithms. Unusual DFT modules containing twiddle factors can be used with this optimization procedure. If each module represents a DFT without the real and imaginary data combination step at the end then REAL DATA algorithms can be optimized. The pre-weave addition operations on a multidimensional convolution may be optimized in a manner identical to the pre-weave addition problem for the WFTA or the split nested DFT.
Backing up to a more global viewpoint, given an appropriate cost function the dynamic programming algorithm will indeed seek out the best algorithm from the vertically split class of possible algorithms. The class itself is restricted in form by both the modules used and the particular factors selected for each module. Two questions arise concerning the effectiveness of the optimizing procedure in a real problem. First, given that one is using modules which are well suited for the application, for any reasonable factoring will some member of the class approach the best possible algorithm over all possible factorings? Also, does the cost model genuinely reflect the workings of typical processors and will low cost correspond to desirability in an algorithm?

The first question may only be answered empirically. By a reasonable factoring it is meant that the module is factored at each stage where the polynomial reductions (or reconstructions) change the dimension of the data array. Using this principle the example length 35,315 and 9009 algorithms were worked out. In each case, optimal algorithms were found near the corner of the minimum multiply and minimum addition restrictions on the scatter plots. The minimum multiply restriction may be found by setting the multiply cost to one and all other costs to zero and running the optimizing routine; the number of multiplies thus obtained is invariant to the particular factoring used, as long as the multiply steps appear as individual factors. The minimum number of additions will always occur in the PFA. Since over all possible factorings one cannot expect to do better than the
minimum multiply and add restrictions, the occurrence of algorithms near the corner of both of these limits indicates that "reasonable" factoring yields algorithms almost as good as could be expected with the vertically split technique using any factoring. This result assumes one is looking only at add and multiply costs.

The cost function, while it may not be all encompassing, does include what are generally considered to be the most important aspects of algorithm design, namely the number of additions and multiplies and memory cycles. Note that the memory cycle figure may be adjusted to reflect not only data memory cycles but also program instruction fetches. These costs plus the crude attempt at indexing and overhead estimation are certainly more refined than simply counting multiplies [1] or multiplies and adds [20]. In that sense this is at least a better cost model if not a best one.

The last point with respect to optimal DFT finding is that, while the gains may be small, it does indeed find the very best solution, so that one may make an intelligent algorithm selection. Having the best solution at hand is not a trivial matter. In the case of the length 9009 DFT even the check to find out that the WFTA has a non-minimum number of multiplies is tedious by hand. Over 5000 possible length 9009 vertically split algorithms were checked to find the optimal algorithms in the scatter plot of Example III. Generation of that scatter plot by hand calculations would have been prohibitively difficult.
The material in Part II is targeted simultaneously at two problems: permuting outputs by changing coefficients and finding coefficients as part of the original design process. The first problem arises when one has on hand standard modules (perhaps in VLSI) and wants to create a PFA without modifying the module structure, unscrambling the outputs, or using separate input/output index maps. The results here indicate that the PFA method is actually simpler than originally suspected, and that the unscrambling problem is not really a problem at all. The applications of coefficient finding to the design process are dramatic. The author's length 17 and 19 DFT modules would not have been attempted without such a procedure. Hand generation of the B matrix, even using an interactive polynomial manipulation program such as SCRATCHPAD, would have been inordinately complicated.

As an outgrowth of this research it would be interesting to characterize the class of A matrices such that $A^t(Ax@y)$ is a valid DFT for some vector $y$. This class may include some algorithms not derivable through a CRT polynomial reconstruction point of view. It is conjectured that if $C(Ax@y)$ is a DFT algorithm then $A^t(Ax@y')$ can also be made a DFT with the proper choice of $y'$, which would imply that all the important information in a DFT module is actually contained in the A matrix, hence the desire to directly characterize that class of possible A matrices.
5. **Conclusions**

The general vertical splitting approach has been used to develop DFT algorithms which are related to the PFA, WFTA, split nested DFT, and Real Data DFT, and furthermore provide many interesting add-multiply-control tradeoffs not available in the above algorithms. A flexible optimization procedure has been developed for finding the "best" algorithm over the vertically split class judged by a variety of costs.

The theory of cyclotomic polynomials has been applied to the study of DFT module structures and revealed that certain explicit permutations required in the PFA can be eliminated. The need for generating the polynomial reconstruction matrix in a DFT module design has been eliminated by directly solving for module coefficients without using the matrix B.
References


General References on Signal Processing and DFT Algorithms


5.1. APPENDIX I

This program will solve a system of partially commutative operations, finding the least cost ordering. The program contains special features for use in PFA DFT problems.

This version, dyn.DO.f, is specially equipped to take into account the cost of extra DO LOOPS induced by the new splintered DFT structures.

parameter (mxdm=5,mxet=12,wwln=500)

mxdm = max number of operation strings
mxet = max length of each string
wwln = max number of nodes with identical pathlengths

real memcost,addcost,mpycost
per unit costs for memorycycles, adds, mpy's, and do loops
The mpycost must include the cost of memory cycles associated with a scaling multiply.

integer dim,mem,add,mpy,dtasze

dim = actual no. of operation strings in the input data
mem,add,mpy = actual no. of memorycycles, adds and mpy's per factor
as read in

dtasze = data size at an old cluster

integer*2 outdim(mxdm,0:mxet),extent(mxdm)

outdim = output data size of each operation
extent = no. of operations in each input string

integer memcy(mxdm,mxet),adds(mxdm,mxet),mpys(mxdm,mxet)

actual no. of memorycycles, adds and mpy's per factor

integer oname(wwln),nname(wwln),sname

old pathname array, new pathname array, temp pathname

integer*2 nptr(wwln),sptr

backwards solution pointers

real ocost(wwln),ncost(wwln),tcoast

the solution cost at old and new nodes, var. cost variables

integer oadd(wwln),nadd(wwln),ompy(wwln),mpy(wwln)

old and new add and multiply counts

integer tadd,tmpy,tmem,tdim

temp. storage for add, mpy and memory cycle counts

tapp = number of applications of some new factor

integer oend,neend,mainl,inx1,inx2,pathin

EOF pointers for old and new node lists, max path length,

index variables

character*1 ast(0:1),alpha(0:9)
character*6 fsize
character*8 ffsize

integer oindex,ish,imid,dim

index to old node list (used while generating new nodes)

ish used to find matching nodes in new node list

imid steps through all successor nodes
dir points to the direction of change between node names
integer*2 pname(mxdm), pflag(mxdm), tname(mxdm), tflag(mxdm)
temp, sto for pathname and special pathname flags
pname, pflag always used with old clusters
tname, tflag always used with new clusters
integer nstflg, numnst, dirflg
used in overhead calculations
integer maxdat
maximum data size constraint
real fad, vad, fmp, vmp
fixed and variable add-type and multiply-type overhead costs
data alpha /'0','1','2','3','4','5','6','7','8','9'/
data ast /' ','*'/
data fnme /'txfile'/

DATA INPUT FORMAT
memcost, addcost, mpycost, docost
maxdatasize (ignore if zero)
FAD, VAD, FMP, VMP
no. operation strings
input datasize                   (string 1)
no. of blocks
output datasize, memcycles, adds, mpyys (actual count)
.
.
input datasize                   (string 2)
.
.
read(5,*) memcost, addcost, mpycost
read(5,*) maxdat
read(5,*) fad, vad, fmp, vmp
read(5,*) dim
if (dim.gt.mxmd) then
   write(6,*) 'MXDM =', mxmd, 'MAX. # OPERATION STRINGS EXCEEDED'
   stop
endif
write(6,*) dim
do 3000 i7=1,dim
   read(5,*) outdim(i7,0)
   read(5,*) extent(i7)
   if (extent(i7).gt.mxet) then
      write(6,*) 'MXET =', mxet, 'MAX. OP. STRING LENGTH EXCEEDED'
      stop
   endif
write(6,*) outdim(i7,0)
write(6,*) extent(i7)
do 3000 i8=1,extent(i7)
   read(5,*) outdim(i7,i8), mem, add, mpy
   write(6,*) outdim(i7,i8), mem, add, mpy
```c
memory(i7,i8)=mem
adds(i7,i8) =add
mpys(i7,i8) =mpy
write(6,3001) memost,addcost,mpycst
if(maxdat.ne.0) write(6,3002) maxdat
write(6,3003) fad,vad,ftp,vmp

3001 format('memost=',e10.4,' addcost=',e10.4,' mpycst=',e10.4)
3002 format('maxdat=',i10)
3003 format('fad=',e10.4,' vad=',e10.4,' ftp=',e10.4,' vmp=',e10.4)

Input data is stored.

Setup the node lists for the first solution pass
oname(i)=0
ocost(i)=0
oadd(i)=0
ompy(i)=0
oend=1
nend=0
maxln=0

do 1170 i6=1,dim
1170 maxln=maxln+extent(i6)
ix1=0
ix2=0
fname=fname // '00'
open(7,file=fname)
rewind 7
write(7,*) 1
write(7,*) 0,0
close(7)

for each possible pathlength
do 1160 pathln=1,maxln
  for each old node
do 1150 oindex=1,oend
    call unpack(pname,pflag,oname(oindex),dim)
dtazse=1
    do 1300 i4=1,dim
1300 dtazse=dtazse*outdim(i4,pname(i4))
  compute total data size at current pathname
  idim=1
  next try to generate a successor pathname
1100 if(idim.gt.dim) go to 1150
  no more successors exist for this cluster
  if(pname(idim).eq.extent(idim)) then
    idim=idim+1
    go to 1100
  else
    try incrementing the factor index for the next module
    we did get a successor
  do the datasize constraint check
```
if(pflag(idim)/2.eq.0) then
  if(atasze.gt.maxdat.and.maxdat.gt.0) then
    max datsize exceeded
    idim=idim+1
    go to 1100
  endif
endif

if(mpyas(idim,pname(idim)+1).ne.0) then
  numnst=0
  (we have a multiply cluster)
  create the successor cluster name
  do 1200 i3=1,dim
    tname(i3)=pname(i3)
  enddo
  copy the factor indices
  if(mod(pflag(i3),2).ne.0) numnst=numnst+1
  count the number of nesting flags
  1200 tflag(i3)=pflag(i3)
  copy the nesting and direction flags
  tname(idim)=pname(idim)+1
  set the nesting flag for module "idim"
  tflag(idim)=pflag(idim)+1
  Now check to see if we are nesting two multiply factors. If so, then we don’t need any direction flags.
  if(numnst.gt.0) then
    do 1215 i3=1,dim
    1215 tflag(i3)=mod(tflag(i3),2)
  endif
  reset all direction flags
endif

calculate the link cost
tmpy=1
  do 1400 i5=1,dim
    if(i5.eq.idim) then
      tmpy=tmpy*mpys(i5,tname(i5))
    elseif(mod(tflag(i5),2).ne.0) then
      tmpy=tmpy*(outdim(i5,tname(i5))-mpys(i5,tname(i5)))
    else
      tmpy=tmpy*outdim(i5,tname(i5))
    endif
  enddo
  continue
  toocst=mpyocst*float(tmpy)
tadd=0
else
  We have an Add-type cluster
  Create the new cluster name
  do 1210 i3=1,dim
c tname(i3)=pname(i3)
c    copy the factor indices
tflag(i3)=0
1210 c reset all direction and nesting flags
tflag(idim)=2
c set the direction flag for module "idim"
tname(idim)=pname(idim)+1
c increment the factor index for module "idim"
c compute tapp
tapp=1
do 1420 i5=1,dim
   if(i5.ne.idim) tapp=tapp*outdim(i5,tname(i5))
1420 continue
tmec=mecy(idim,tname(idim))*tapp
tadd=adds(idim,tname(idim))*tapp
tmpy=0
tcost=addcost*float(tadd)
tcost=tcost+mecost*float(tmec)
c count the number of nesting flags on cluster A
nstflg=0
numnst=0
dirflg=0
do 1430 i5=1,dim
   if(mod(pflag(i5),2).eq.1) then
      numnst=numnst+1
      nstflg=i5
   endif
   if(pflag(i5)/2.eq.1) dirflg=i5
1430 continue
if(numnst.eq.0) then
   c Both A and B end with add-type factors. Check for
c collapsibility.
   if(dirflg.ne.idim) tcost=tcost+fadd+vadd*float(tapp)
elseif(numnst.eq.1) then
c We have one multiply step sandwiched between two
   add factors. Check for collapsibility in both places.
   if(nstflg.ne.idim) tcost=tcost+fadd+vadd*float(tapp)
   if(dirflg.ne.nstflg) then
   c compute tapp for the multiply factor
tapp=1
do 1440 i5=1,dim
   if(i5.ne.nstflg) tapp=tapp*outdim(i5,pname(i5))
1440 continue
tcost=tcost+fadd+vadd*float(tapp)
endif
else
   c We have several nested multiplies sandwiched between
two add factors.
tcost=tcost+fad+vad*float(tapp)

Compute the total number of multiplies in the nested bunch of factors.
tmpy=1
do 1450 i5=1,dim
if(mod(pflag(i5),2).ne.0) then
  tmpy=tmpy*(outdim(i5,pname(i5))-mpys(i5,pname(i5)))
else
  tmpy=tmpy*outdim(i5,pname(i5))
endif
continue 1450
end do

continue

tmpy=dtasze*tmpy

tcost=tcost+fmp+vmp*float(tmpy)

tcost=tcost+ocost(oindex)
tmpy=tmpy+ompy(oindex)
tadd=tadd+oadd(oindex)
idim=idim+1
endi
endi

call pack(tname,tflag,sname,dim)

ish=1

Search for a match in newlist for the fresh pathname, sname

1180 if(ish.gt.mend) then

Sname was not found

  if(nend.eq.wvln) then
      write(6,*),'WVLN =',wvln,' INSUFFICIENT STORAGE ALLOCATED'
      write(6,*),'PATHLN =',pathln
      stop
  endif

  nend=nend+1

  nptr(nend)=oindex

  ncost(nend)=tcost

  nname(nend)=sname

  nadd(nend)=tadd

  nmpy(nend)=tmpy
else

  if(nname(ish).ne.sname) then
      ish=ish+1
      go to 1180
  else

    if(tcost.lt.ncost(ish)) then
      nptr(ish)=oindex
      ncost(ish)=tcost
      nadd(ish)=tadd
      nmpy(ish)=tmpy
endi
endif
endif
go to 1100

(try another successor)

1150 continue

(try another old cluster)

c
At this point all old clusters have been exhausted
Store the old cluster information needed to reconstruct
a valid least cost path.

if(mend.eq.0) then
  write(6,*), 'PATHLN=',pathln,' NO SUCCESSOR CLUSTERS'
  stop
dendif

if(inx1.eq.9) then
  inx1=0
  inx2=inx2+1
else
  inx1=inx1+1
dendif
ffmme=ffmm // alpha(inx2) // alpha(inx1)
open(7,fil=ffmm)
rewind 7
write(7,*) mend
do 3200 i10=1,mend
write(7,*) nnname(i10),npntr(i10)
oname(i10)=nnname(i10)
oadd(i10)=nadd(i10)
ompyn(i10)=nmpyn(i10)
3200 ocost(i10)=ncost(i10)
close(7)
0end=mend
mend=0
1160 continue

(do all clusters in the next pathlength group)

c
Retrieve the least cost path from the stored data

The direction flags split the last cluster into several pieces.
It is therefore necessary to first search the ncost list
for the winning cluster, and start the path reconstruction
from that cluster.
tcost=ncost(1)
sprt=1
do 3305 illa=1,mend
  if(ncost(illa).lt.tcost) then
tcost=ncost(illa)
sprt=illa
dendif
3305 continue
Now we are ready to start the path reconstruction procedure.

sptr points to the winning terminal cluster.

The path information will be stored in the nname, ncost array.

ncost(1) = ncost(sptr)
nptr(1) = nptr(sptr)
nname(1) = nname(sptr)

mpys = 'mp', mpy(sptr)

mpys = 'mp', mpy(sptr)

sptr = nptr(1)

do 3300 i11 = 1, maxln

if (inx1 .eq. 0) then
  inx1 = 9
  inx2 = inx2 - 1
else
  inx1 = inx1 - 1
endif

ffname = fname // alpha(inx2) // alpha(inx1)

open (7, file = ffname)

rewind 7

read (7, *) end

do 3310 i12 = 1, sptr

read (7, *) nname(i11 + 1), nptr(i11 + 1)

3310

sptr = nptr(i11 + 1)

3300

doi32 = 1, maxln

i14 = maxln + 1 - i13

sname = nname(i14)

call unpack(pname, pflag, sname, dim)

call unpack(tname, tflag, nname(i14 + 1), dim)

do 3330 i15 = 1, dim

if (tname(i15).ne. pname(i15)) dir = i15
call unpack(tname, tflag, nname(i15), dim)

call unpack(pname, pflag, sname, dim)

do 3320 i16 = 1, dim

write (6,3340) dir, (pname(i16), ast (mod (pflag(i16), 2)), i16 = 1, dim)

3330

continue

3340

format ('dir=', i13, 5x, 5(i13, a1, 3x))

stop

end

The format of the packed cluster names is irrelevant.

The format of the unpacked names as stored in nmeary, flgary is important.

nmeary - contains the factor indices for each module

flgary - contains direction and nesting flags. The

1st is the nesting flag, and the next bit is

the direction flag. There is one nesting

and direction flag for each module. Flgary has

the same dimension as nmeary.

subroutine pack(nmeary, flgary, name, dim)

integer*2 nmeary(mxdm), flgary(mxmd)

integer name, dim

name = 0

do 2100 ipack = 1, dim
i2pack = dim - ipack + 1
name = 16 * (4 * name + fgary(i2pack)) + nmeary(i2pack)
return
end

subroutine unpack(nmeary, fgary, name, dim)
   integer*2  nmeary(mxdm), fgary(mxdm)
   integer    name, dim
   do 2200 ipack = 1, dim
      nmeary(ipack) = mod(name, 16)
      name = name / 16
      fgary(ipack) = mod(name, 4)
   enddo
   name = name / 4
end
5.2. **APPENDIX II**

This is the Gen program
This program must be compiled with two subroutines

**INIT(M,N)** returns $M=$number of coefficients+1
$N=$length of DFT

**TRANS(X,Y,COEFF)** is the DFT prototype module which transforms
the array $X,Y$ of length $N$ where $X$ is the real part and
$Y$ is the imag. part. TRANS uses coefficients from
the length $M$ array COEFF starting with the second
coefficient. The first coefficient corresponds to
fixed rational constant multipliers and will always
be 1.

```fortran
parameter (nize=39,ndim=60)
c nize ---- max. number of coefficients
c ndim ---- max. DFT length
integer n,m,x(ndim),y(ndim)
character*(80) form,form2

call init(m,n)
n2=(n/2)+1

form= ',
  do 50 i1=1,(n-n2)
    form2=form
  50   form= ',i4' // form2
    form2=form
    form= ',2x' // form2
    do 51 i1=(n-n2+1),n-1
      form2=form
      51   form= ',i4' // form2
      form2=form
      form= '(i4' // form2

write(6,101) n,m

format(2i4)
do 100 i1=1,m
do 110 i2=1,n

  call t(i1,i2,x,y,n,m)
do 200 i3=1,(n-n2)
x(i3+n2)=y(i3+1)
write(6,form) (x(i4),i4=1,n)
```
110 continue
write(6,*), '-------------------------------'
100 continue
stop
end

Here are some subroutines
subroutine t(i,j,x,y,m,n)
parameter (nize=39, ndim=60)
integer xt(ndim),yt(ndim),x(ndim),y(ndim)
integer coef(nize)
if(i,ne.1) then
500 coef(it)=0
cof(1)=1.
coef(i)=1.
end if

Set both xt and x equal to the Jth basis vector
510 do 510 it=1,n
xt(it)=0.
x(it)=0.
yt(it)=0.
510 yt(it)=0.
xt(j)=1.
x(j)=1.
call trans(xt,yt,coef(2))
cof(1)=0.
call trans(x,y,coef(2))
do 520 it=1,n
x(it)=xt(it)-x(it)
y(it)=yt(it)-y(it)
520 else
515 do 515 it=1,m
515 coef(it)=0.
515 coef(1)=1.
do 525 it=1,n
525 x(it)=0
y(it)=0
x(j)=1.
call trans(x,y,coef(2))
endif

return
end

THIS IS THE Q-R SOLUTION PROGRAM
THE FOUR ARGUMENTS ARE:
ROWFILE — TELLS WHICH ROWS AND COLUMNS OF GEN DATA TO USE
THIS IS USEFUL FOR PARING DOWN LARGE SYSTEMS
FORMAT: N -- DFT SIZE
        D -- OUTPUT PERMUTATION CONSTANT
          X X X X Y Y ONE'S AND ZERO'S TO INDICATE
          WHICH COLUMNS OF DFT TO USE. FIRST N/2
          PLACES REFER TO REAL PART AND SECOND N/2
          REFER TO IMAG. PART
          X X X X X X ONE'S AND ZERO'S FOR ROWS

GENFILE -- FILE WITH GEN OUTPUT DATA
OUTFILE -- OUTPUT FILE FOR COEFFICIENT DATA

PARAMETER (NSQR=125,NDIM=19,MAXM=39)
DOUBLE PRECISION X(NSQR,MAXM),R(NSQR),W,CRI
INTEGER D,ROW(NDIM),COL(NDIM),NUC(NDIM)
DOUBLE PRECISION B(MAXM),RSD(NSQR),QRAUX(MAXM),WORK(MAXM)
INTEGER JPVX(MAXM)
CHARACTER*80 ROWFIL,GENFIL,OUTFIL
CALL GETARG(1,ROWFIL)
CALL GETARG(2,GENFIL)
CALL GETARG(3,OUTFIL)
OPEN(4,FILE=ROWFIL)
REWIND 4
READ(4,*) N
READ(4,*) D

IF(N.GT.NDIM) THEN
WRITE(6,*) 'NDIM=',NDIM,'IS TOO SMALL'
STOP
ENDIF

N2=(N/2)+1
W=.8*D0*DFLOAT(D)*DATA(1.D0)/DFLOAT(N)
READ(4,*) (ROW(I),I=1,N)
READ(4,*) (COL(I),I=1,N)
CLOSE(4)

C CHECK TO SEE IF WE WILL OVERFLOW ARRAY BOUNDS
N1B=0
N2B=0
DO 217 I=1,N
N1B=N1B+ROW(I)
217 N2B=N2B+COL(I)
NIB=N1B-N2B-(N1B*COL(N))
IF(NIB.GT.NSQR) THEN
WRITE(6,*) 'NSQR=',NSQR,'IS TOO SMALL'
WRITE(6,*) 'REQUIRE NSQR=',NIB
STOP
ENDIF
open(4, file=genfil)
rewind 4
read(4, *) n1, m

if(n1.ne.n) then
write(6, *) 'incompatible data'
stop
endif

if(m.gt.maxm) then
write(6, *) 'maxm=', maxm, 'is too small'
stop
endif

do 220 i1=1, m
   inx=1
   do 210 i2=1, n
      read(4, *) (x(i3, i1), i3=inx, (inx+n-1))
      do 214 iyu=1, n
         nyuc(iyu)=int(x(iyu+inx-1, i1))
      enddo
      write(6, 211) (nyuc(iyu), iyu=1, n)
   enddo
   format(2014)
   iscan=inx-1
   do 210 i4=1, n
      if((col(i2).ne.0).and.(row(i4).ne.0)) then
         x(inx, i1)=x(iscan+i4, i1)
         inx=inx+1
      endif
   enddo
210 continue
if(i1.ne.m) read(4, 201)
201 format(ix)
if(i1.ne.m) write(6, 201)
220 continue
close(4)
inx=1
write(6, *) (row(i), i=1, n)
write(6, *) (col(i), i=1, n)
do 230 i2=1, n
do 230 i4=1, n
   if((col(i2).ne.0).and.(row(i4).ne.0)) then
      if(i4.1e.n2) then
         r(inx)=dcos(w*dfloat(i2-1)*dfloat(i4-1))
      else
         r(inx)=dsin(w*dfloat(i2-1)*dfloat(i4-n2))
      endif
      inx=inx+1
   endif
230 continue
nrows=inx-1
write(6, *) 'nrows= ', nrows
call sqrst(x, nsq r, n rows, m, r, 0, b, rsd, k, j pvt, q raux, work)
write(6, 232) (rsd(i), i=1, n rows)
232
format(7e10.3)

open(7, file='outfil')
rewind 7
write(7, 805) n
805 format(1c1, DATA FOR LENGTH ', i2, ' DFT')
do 660 i=1, m
660 c r(i) = b(i)
ii = i - 1
write(7, 601) ii, c r(i)
601 format(6x, 'data c', i4, ' /', f23.16, ' d0 /')
stop
close(7)
end

'subroutine sqrst(x, ldx, n, p, y, to l, b, rsd, k, j pvt, q raux, work)
 integer ldx, n, p, k, j pvt(1)
 double precision x(ldx, 1), y(1), b(1), rsd(1), q raux(1), work(1)
 integer info, j, kk, m
 double precision t, to l

do 10 j=1, p
    j pvt(j) = 0
10 continue

call dqrdc(x, ldx, n, p, q raux, j pvt, work, 1)

k = 0
m = min(n, p)
do 20 kk = 1, m
    if(abs(x(kk, kk)).le.tol*abs(x(1, 1)) go to 30
k = kk
20 continue
30 continue
if(k.ne.0)
1 call dqrsl(x, ldx, n, k, q raux, y, rsd, rsd, b, rsd, rsd, 110, info)
do 40 j=1, p
    j pvt(j) = -j pvt(j)
    if(j.gt.k) b(j) = 0
40 continue

do 70 j=1, p
    if(j pvt(j).gt.0) go to 70
k = j pvt(j)
jpvt(j) = k
70 continue
if(k.eq.j) go to 60
t = b(j)
b(j) = b(k)
b(k)=t
jpv(k)=jpv(k)
k=jpv(k)
go to 50
60 continue
70 continue
return
end