ABSTRACT

A METHOD OF COMPUTING THE FAST FOURIER TRANSFORM

by

Randol R. Read

The fast Fourier transform is investigated. It is proved that the number of real (as opposed to complex) multiplications necessary to implement the algorithm for complex input sequence of length \( N = 2^M \) is

\[ 2N(M - 7/2) + 12. \]

Methods which do not avoid the unnecessary multiplications predict \( 2N(M - 1) \) or \( 2NM \). It is shown experimentally that for at least one implementation of the algorithm, it is faster to take advantage of the multiplication savings mentioned above. Some theorems regarding computational savings when transforming real data are presented. A system of subroutines for calculating finite discrete Fourier transforms by the fast Fourier transform method is given. The results of applying this system to two specific problems is presented.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>i</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>PRELIMINARIES</td>
<td>4</td>
</tr>
<tr>
<td>THE FAST FOURIER TRANSFORM THEORY</td>
<td>8</td>
</tr>
<tr>
<td>RESULTS AND EXAMPLES</td>
<td>16</td>
</tr>
<tr>
<td>CONCLUSIONS</td>
<td>20</td>
</tr>
<tr>
<td>APPENDICES</td>
<td></td>
</tr>
<tr>
<td>I. The Relation of the Finite Discrete Fourier Transform to the Fourier Integral Transform</td>
<td>22</td>
</tr>
<tr>
<td>II. User's Guide to FFT, Program Listings</td>
<td>25</td>
</tr>
<tr>
<td>III. The Real Fast Fourier Transform</td>
<td>35</td>
</tr>
<tr>
<td>IV. Algebra of the Fast Fourier Transform when the Dimension of the Transform is $N = 2^M$</td>
<td>40</td>
</tr>
<tr>
<td>V. Counting the Necessary Multiplications when the Dimension of the Transform is $N = 2^M$</td>
<td>42</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>44</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

The author would like to express his appreciation to those from whose work he has borrowed, and to those who have helped in other ways in the preparation of this paper. In particular, I would like to thank my thesis director, Dr. C. S. Burrus and the other members of my committee, Drs. T. W. Parks and F. L. Huband for their guidance. Computer time for this work was provided by NSF grant GU-1153. Personal support was provided by an NDEA Fellowship.
INTRODUCTION

The fast Fourier transform is a very efficient method for machine computation of the finite discrete Fourier transform of a complex input sequence. This finite discrete Fourier transform (DFT) has properties which are closely related to the Fourier integral transform. The relation between the DFT of a sampled time signal and the Fourier integral transform of that signal is given in Appendix I.

In this paper, a method for computing the DFT is examined. In particular, we will examine the fast Fourier transform (FFT) algorithm suggested by Cooley and Tukey [2]. First we will answer such basic questions as why the FFT is fast and why it is appropriate to have the length of the complex input sequence a highly composite number (say of length $N = 2^M$, $M$ an integer). Once we have answered these questions, we take a more detailed look at the FFT algorithm when the length of the input sequence is $N = 2^M$. In that case, we will show that it is possible to accomplish the algorithm with only $2N(M - 7/2) + 12$ real multiplications. Methods of implementing the algorithm which do not avoid all unnecessary multiplications predict $2NM$ or $2N(M-1)$ real multiplications (the exact number depending on
programming details). The important question to be answered is, can we in practice take advantage of these multiplication savings in such a way that we realize a net time savings over a program which does not recognize the unnecessary multiplications? This is a very difficult question to answer exactly because of the many differences in both machines and programs used for computing the algorithm. It is shown experimentally that for the author's program, when the unnecessary multiplications are eliminated, a time savings (6% to 10% depending on the dimension of the transform) can be achieved. The data for this experiment was a complex sequence whose real and imaginary parts were filled with random numbers uniformly distributed on the interval [0,1].

In order to perform the experiment mentioned above, a program that avoids the unnecessary multiplications was developed to compute the FFT. This subroutine, with several other subroutines necessary to use the FFT subroutine, is listed in Appendix II. An explanation of how to use these subroutines is also given there.

When applying the discrete Fourier transform to real data, it is possible to realize further computational savings. To realize these savings, we take advantage of symmetries relating to the input sequence and its transform. It is well known that the integral Fourier transform of a real time function is hermitian symmetric, that is,
f(t) real $\iff F(-\omega) = F^*(\omega)$ where the asterisk denotes conjugation and $F$ is the Fourier transform of $f$. There is an equivalent form for the DFT. Appendix III contains theorems regarding the computational savings that can be achieved when conditions analogous to the above hold, that is, the input sequence is real or the input sequence has certain symmetry which is defined in Appendix III.
The finite discrete Fourier transform (DFT) is a linear transform which can be thought of as a matrix operating on an input vector. In general, calculating the product of $N \times N$ matrix and an $N$-vector involves $N^2$ "operations", that is, there are $N^2$ products to be calculated and $N$ sums of $N$ factors each to be calculated in order to obtain the $N$-vector result. As we shall see, in the case of the DFT, the $N \times N$ matrix has some special properties which will enable us to reduce the number of operations considerably. In particular, for the length input vector $N = 2^M$, $M$ an integer, the number of operations is reduced to the order of $\frac{N}{2^M}$ multiplications and $NM$ additions.

In this paper, the DFT will be defined by *

$$A(k) = \sum_{p=0}^{N-1} X(p) W(pk) \quad k = 0, \ldots, N-1 \quad (1)$$

where $X(p)$ is the $p$th complex input data point of input vector of length $N$, $X = (X(0), \ldots, X(N-1))$, $A(k)$ is the $k$th coefficient of the DFT, and $W(t) = \exp(2\pi \sqrt{-1} t/N)$. Thus equation (1) can be conveniently written $A = WX$ where $A, X$ are vectors and $W$ is a matrix.

* The definition of the DFT is not uniform in the literature. The $A(k)$'s are sometimes defined with scale factors or with a negative exponent for the exponential. The definition given here agrees with that in the Gentleman and Sande paper [5].
As one might expect, the "special properties" mentioned with respect to the matrix \( W \) are direct consequences of the function \( W(\cdot) \). Important properties of \( W(\cdot) \) are given below. Some of them are more or less obvious.

\[
W(kN) = 1, \ \text{k integer}
\]

\[
W(p + k) = W(p) W(k)
\]

Periodicity: \( W(k + N) = W(k) \)

Conjugation: \( W(-k) = W^*(k) \) where \( * \) denotes complex conjugation

Orthogonality: \( \sum_{k=0}^{N-1} W(k(p-\ell)) = N \delta_N(p-\ell) \)

where \( \delta_N \) is the Kronecker delta with argument interpreted modulo \( N \), that is, \( \delta_N(kN) = 1 \) if \( k \) is an integer, \( \delta_N(kN) = 0 \) otherwise. This orthogonality property is the property that is less obvious than the others. The proof of this orthogonality property follows.

\[
W(k(p-\ell)) = [W(p-\ell)]^k
\]

\[
\sum_{k=0}^{N-1} x^k = (1 - x^N)/(1-x) \quad x \neq 1
\]

\[
= \sum_{k=0}^{N-1} W(k(p-\ell)) = 1 - W(N(p-\ell))/1 - W(p-\ell) \text{ for } W(p-\ell) \neq 1
\]

but \( W(n(p-\ell)) = 1 \) for \( p-\ell \) an integer, therefore

\[
1 - W(N(p-\ell))/1 - W(p-\ell) = 0
\]
except where \( W(p-t) = 1 \) (i.e. \( p - t = kN \), \( k \) an integer) then the sum is equal to \( N \).

It is useful to interpret the arguments of \( A \) and \( X \) to modulo \( N \). Thus we will define \( X(tN + k) = X(k) \), \( t \) an integer.

We will now define the inverse DFT (IDFT) and show how the FFT can be used to compute both the DFT and its inverse. the IDFT is

\[
X(t) = \frac{1}{N} \sum_{k=0}^{N-1} A(k) W(-kt) \quad t = 0, \ldots, N-1 \tag{2}
\]

The DFT is a reversible mapping. This can be seen by substituting (1) into (2)

\[
X(t) = \frac{1}{N} \sum_{k=0}^{N-1} \sum_{p=0}^{N-1} X(p) W(pk) W(-kt) = \sum_{p=0}^{N-1} X(p) \frac{1}{N} \sum_{k=0}^{N-1} W(k(p-t))
\]

\[
= X(t) \text{ by the orthogonality property.}
\]

If we have an algorithm for calculating the DFT, it is easy to see that we can use it to calculate the IDFT of a sequence since

\[
\{ \sum_{k=0}^{N-1} \frac{1}{N} A*(k) W(kt) \}^* = \frac{1}{N} \sum_{k=0}^{N-1} A(k) W(-kt).
\]

Thus in this paper we will discuss only the transform defined by (1).

Now that we have established the properties of the function \( W(\cdot) \), in the next chapter we will see how to take advantage of these properties in order to reduce the
computational effort in calculating the DFT (1).
THE FAST FOURIER TRANSFORM

In the previous chapter we established certain properties of the function $W(\cdot)$, which is basic in the definition of the DFT. We will now look at how these properties can be used in order to make a fast algorithm for the calculation of the DFT. The presentation given here is essentially the same as that in the Cooley-Tukey paper [2]. After we show how computational savings can be achieved when $N$ has two factors, $N = N_1N_2$, an example is given in matrix form for a low order ($N = 4$) transform. The general case of $N = 2^M$ is given in Appendix IV.

We will now show how the periodicity properties of $W(\cdot)$ can be used to factor a transform of length $N = N_1N_2$ into smaller transforms of length $N_1$ and transforms of length $N_2$.

$$A_\ell = \sum_{k=0}^{N-1} X_k W^{\ell k} \quad \ell = 0, \ldots, N-1 \quad (1)$$

if we let $\ell = p + qN_1$ $p = 0,1,\ldots,N_1-1$
$q = 0,1,\ldots,N_2-1$
and $k = r + sN_2$ $r = 0,1,\ldots,N_2-1$
$s = 0,1,\ldots,N_1-1$

the $\ell$ and $k$ still vary over the proper range. We then have
\[ A(p+qN_1) = \sum_{s=0}^{N_1-1} \sum_{r=0}^{N_2-1} X(r + sN_2)W([r+sN_2][p+qN_1]). \]

Expanding the argument of the \(W(\cdot)\) function,
\[ W(rp + sN_2p + qN_1r + sqN_1N_2) = W(rp)W(spN_2)W(qrN_1). \]

Replacing this in the sum and switching summation order,
\[ A(p+qN_1) = \sum_{r=0}^{N_2-1} W(qrN_1) \{W(rp)[\sum_{s=0}^{N_1-1} X(r+sN_2)W(spN_2)]\}. \] \(3\)

Note that for a given \(r\) the sum in the brackets is just a DFT of dimension \(N_1\). Also for a given \(p\), the outside sum is just a DFT of dimension \(N_2\). The sums in (3) represent matrix times vector multiplications of dimension \(N_1\) and \(N_2\) respectively. We have succeeded in reducing the original problem which involved an \(N\)-dimensional matrix times vector multiplication to one involving \(N_2\) matrix times vector multiplications of dimension \(N_1\), multiplying by the \(W(rp)\) terms, then performing \(N_2\) matrix times vector multiplications of dimension \(N_1\). If the \(N \times N\) matrix times an \(N\)-vector involves \(N^2\) "operations", then neglecting the \(W(rp)\) terms*, we have reduced the number of operations from \(N^2\) to \(N_2N_1^2 + N_1N_2^2 = N(N_1 + N_2)\).

Now that we have shown how the term "fast" comes about in the fast Fourier transform, let us turn to an example to see how the properties of \(W(\cdot)\) can be used

* The \(W(rp)\) terms have been called "twiddle factors"[5].
further. We will see in this example (and it will be true in general when \( N = 2^M \)) that it is convenient to incorporate the "twiddle factors" with the \( W(\cdot) \) associated with the outer sum in (3). The example which we will consider here is \( N = 2^2 \). Equation (3) becomes

\[
A(p + 2q) = \sum_{r=0}^{1} W(2qr) \left[ W(rp) \left( \sum_{s=0}^{1} X(r+2s) W(2sp) \right) \right].
\]

This equation, if we followed the brackets and braces, would tell us to calculate two \( 2 \times 2 \) transforms, multiply the resulting array by \( W(rp) \), then calculate 2 more \( 2 \times 2 \) transforms. But since a \( 2 \times 2 \) transform matrix \( W \) has the form

\[
\begin{pmatrix}
1 & 1 \\
1 & -1
\end{pmatrix}
\]

it is convenient for programming purposes to include the "twiddle factors" with the \( W(2qr) \) term. Let us investigate this matter further. In matrix form, the dimension 4 transform is

\[
WX = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 \\
0 & 2 & 0 & 2 \\
0 & 3 & 2 & 1
\end{pmatrix}
\begin{pmatrix}
X(0) \\
X(1) \\
X(2) \\
X(3)
\end{pmatrix} \quad (4)
\]

where in the matrix we have inserted the argument of the \( W(\cdot) \) function in place of the function. This presents
a slight problem when we actually want to multiply by zero but we accomplish this by leaving a blank space in the matrix, thus a matrix will actually look sparse when it is.

We note in (4) that $X(0)$ and $X(2)$ occur only in the combination $W(0)X(0) + W(0)X(2)$ or $W(0)X(0) + W(2)X(2)$ and it would be wasteful to calculate both sums twice as direct multiplication of the matrix times the vector would dictate. The same is true of the pair $X(1)$ and $X(3)$.

Suppose that we factor out of the matrix in (4) a sparse matrix to calculate the appropriate sums of the pairs $X(0)$ and $X(2)$ then $X(1)$ and $X(3)$,

$$
\begin{bmatrix}
0 & 0 & 0 & 1 \\
0 & 2 & 0 & 3
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 2
\end{bmatrix}
\begin{bmatrix}
X(0) \\
X(1) \\
X(2) \\
X(3)
\end{bmatrix}
$$

These two matrices are really just operations on pairs of data points. In this example $M = 2$ and we can factor into 2 sparse matrices. It is true that for $N = 2^M$, we can factor the $W$ matrix into $M$ sparse matrices. Each sparse matrix operates on the data array by combining them in pairs. This is one of the main reasons why the programming is simple and high efficiency can be achieved when $N = 2^M$.

In Appendix IV the form analogous (3) is derived when $N = 2^M$. Again the argument there is essentially
that of the original Cooley-Tukey paper [2]. The result is stated here. If in (1) we let

\[ l = \sum_{i=0}^{M-1} l_i 2^i = L(M-1), \ l_i \in \{0,1\} \ \forall i \]

similarly for \( k = \sum_{i=0}^{M-1} k_i 2^i = K(M-1) \), then (1) can be factored

\[ A(L(M-1)) = \sum_{k_0} W(k_0 l M-1 \ 2^{M-1}) [W(k_0 L(M-2))] \sum_{k_1} W(k_1 l M-2 \ 2^{M-1}) \]

\[ \sum_{k_2} \ldots \sum_{k_{M-2}} W(k_{M-2} l_1 2^{M-2}) \sum_{k_{M-1}} X(K(M-1)) W(k_{M-1} l_0 2^{M-1}) \]

(5)

Here the limits on the sums are \( k_i = 0 \) to \( k_i = 1 \), that is, we only operate on pairs of data points. This expression is rather complicated algebraically but once it is understood, the fast Fourier transform algorithm for \( N = 2^M \) becomes very clear.

Each sum in (5) represents a pass through the data array. On these passes through the array the data points are multiplied by a complex exponential \( W(\cdot) \) (the argument of which depends on the array position), then two of the data points are added. We start with an \( N \)-length array indexed on \( k_0, \ldots, k_{M-1} \) (i.e. \( k \)). After the first pass (i.e. the sum on \( k_{M-1} = 0 \) to 1 for each value \( k_i \ i = 0 \),
... M-2 and \( l_0 = 0 \) or 1) we have an N length array indexed on \( k \), \( i = 0, \ldots , M-2 \) and \( l_0 \). We then pass through this array (i.e. sum on \( k_{M-2} \)), multiplying by the appropriate \( W(\cdot) \) and adding. At the end we will have an array indexed only on \( l_0 \), \( i = 0, \ldots , M-1 \), which will be \( A \).

Now that we have the FFT algorithm, let us look at a table of the complex exponentials which are used in (5). We do this with a view toward eliminating all the unnecessary multiplications in (5). For example, when we are multiplying an array point by \( W(0) = 1 \) we would like to know it and avoid doing so, thereby saving computational effort.

Consider the sum on \( k_{M-1} \) in (5), that is, the first pass through the data array. The complex exponentials \( W(\cdot) \) that we use in this pass are \( W(0) \) or \( W(2^{M-1}) \). But \( W(0) = 1 \) and \( W(2^{M-1}) = -1 \) so there are no multiplications involved in the first pass through the array. On the second pass, the argument of \( W(\cdot) \) depends on \( k_{M-2} \), \( l_0 \) and \( l_1 \). The \( W(\cdot) \)'s that are used in this pass are \( W(0) \), \( W(2^{M-1}) \), \( W(2^{M-2}) \), \( W(3 \cdot 2^{M-2}) \) which are 1, -1, j, -j*, respectively. There are no machine multiplications involved on this pass either, since multiplication by j only involves a change of sign and a swap of data in the real and imaginary parts. It becomes clear that a large number of the "multiplications" in the algorithm are not really multiplications at all

* Here j is the complex exponential \( \exp(j\pi/2) \).
in the sense that the computer must actually perform a multiplication. We will now investigate this matter further and show exactly how many multiplications there are.

Half of the complex exponentials in (5) are easily eliminated since a $k_i$ can be factored out of the argument of each $W(\cdot)$ which follows each $\sum_{k_i}$. Since $k_i = 0$ or 1, half of the arguments are zero which means $W(\cdot) = 1$. Another half of the multiplications can be eliminated since at each summation there is one "free" index. That is, on the $i$th pass through the array (starting from the "zeroth") the $l_i$ is "free" in the sense that the data is not indexed on $l_i$ but will be after the summation. Note that in (5) these $l_i$'s appear only in the $W(\cdot)$'s just after the $i$th summation, this means that as this free $l_i$ varies from 0 to 1, the argument of the exponential multiplier shifts by $2^{M-1}$. This simply means it changes sign. Since changing the $l_i$ doesn't affect the position in the array, this implies that we will use the product of some $W(\cdot)$ times the appropriate array location, then use the same product with its sign reversed when we change $l_i$ from 0 to 1. Therefore, for the purpose of counting multiplications, we can let $l_i = 0$.

While it would seem complicated, it is really an easy task to write a table of the remaining exponentials
for the program of dimension $N = 2^M$. 

$k_{M-1} = 1 \ W(0) \ W(0) \ W(0) \ W(0) \ ... \ W(0) \ W(0) \ N/2$ elements per row

$k_{M-2} = 1 \ W(0) \ W(2^{M-2})W(0) \ W(2^{M-2})...W(0) \ W(2^{M-2})$

\[ \vdots \]

$k_o = 1 \ W(0) \ W(1) \ W(2) \ \ldots \ \ldots \ W(2^{M-1}-1)$

We have already shown that $W(2^{M-1}) = -1$ and $W(2^{M-2}) = j$. There are two more times when machine multiplications can be saved, these are when the argument of the exponential function are a multiple of $\pi/4$ radians other than those already given. If we take these multiplication savings into account, it is shown in Appendix V that the number of real (machine) multiplications necessary in order to calculate (5) is

$$2N(M - 7/2) + 12.$$ 

It is easy to see if we used every number in the table of complex exponentials, the number of real multiplications (4 real multiplications per complex multiplication) would be $2NM$. If we implement a program that will skip the unnecessary multiplications, we should be able to realize a time savings provided that the time required to test for the conditions where the multiplications can be saved does not take more time than actually carrying out the multiplications.
RESULTS AND EXAMPLES

Results

An algorithm for computing the FFT has been programmed in Fortran IV. The subroutine was tested and timed, then modified by adding statements to recognize when the conditions occurred and when a multiplication savings pointed out in the previous chapter occurred.

Table I shows the results of this test. Figure I shows a curve of the percentage time savings and Figure II shows a curve of the approximate percentage of multiplication savings. Note that in all cases the version which did fewer multiplications was faster. Note also that the curvature of the percentage time saving curve is in the right direction to agree with Figure II.

ACTUAL PERFORMANCE ON IBM 7040*

<table>
<thead>
<tr>
<th>Length</th>
<th>1st program All multiplications (FFT_{II})</th>
<th>2nd program Sav.Mult.FFT_{III} #times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>165 sec</td>
<td>156 sec</td>
</tr>
<tr>
<td>128</td>
<td>191 sec</td>
<td>178 sec</td>
</tr>
<tr>
<td>32</td>
<td>34 sec</td>
<td>31 sec</td>
</tr>
<tr>
<td>8</td>
<td>30.5 sec</td>
<td>27.5 sec</td>
</tr>
</tbody>
</table>

* Times include bit reversal [2] of the data. They were measured with a stop watch. The input was random numbers, distributed uniformly on [0,1].
Examples

One way to test an FFT algorithm for accuracy is to generate a random input, transform the data, inverse transform it, then compare the output to the input.

This technique was used to test the FFT program listed in Appendix II. The input for the test was a sequence of random numbers uniformly distributed on $[0,1]$. These numbers were used to fill the real and imaginary positions of the complex input sequence.

The error criterion used was the sum of the modulae of the input sequence minus the final sequence divided by the length of the sequence. Note that this error is not the "norm" of the error, it is the average of the complex errors

$$\text{ERROR} = \frac{1}{N} \sum_{k=0}^{N-1} |X_k - A_k|$$

where $A_k$ is the result of transforming, then inverse-transforming $X_k$. Figure III shows a semi-logarithmic
plot of the result of this test for different length inputs.

As another example we might consider the Gaussian function \( g(t) = e^{-\pi t^2} e^{-\pi f^2} \). Note that this function decreases very rapidly with the argument in both the time and frequency domain. Therefore, we can predict that the aliasing (see Appendix I) will have negligible effect if we choose the sampling rate and the number of samples correctly. The following simple experiment proves that this is true:

1. Approximate \( g_p(t) \) by \( g(t) + g(t - T) \)
2. Sample \( g(t) + g(t - T) \) at \( t = jT/N \) \( \Rightarrow g_j \)
   \( j = 0, \ldots, N-1 \) with \( N = 256, T = 16 \).
3. Transform \( g_j/16 \Rightarrow A_j \) \( j = 0, \ldots, N-1 \)
4. Compare the output, \( A_j \), with the \( g_j \), \( j = 0, \ldots, N-1 \).

This experiment was performed. The output of the transform,
$A_j$, was indeed $g_j$, within the accuracy of the machine, that is,

$$\max_j |A_j - g_j| \leq 4.0 \times 10^{-8}$$

(The IBM 7040 carries 8 digits).
CONCLUSIONS

We have shown that it is possible to modify the fast Fourier transform algorithm when the dimension is $N = 2^M$ so that we eliminate certain unnecessary machine multiplications by complex exponentials, that is to say, we have recognized the cases where the complex exponential multiplier in the DFT has as its argument a multiple of $\pi/4$ radians. We have shown experimentally in the case of the author's program, that it is possible to realize a time savings by using this technique. The reduction in time in this case is certainly not the step from a time proportional to $N^2$ operation to a time proportional to $N \log_2 N$ operations, but the question of whether the modification is worthwhile and must be decided by individual needs. The 6% to 10% time savings cost only a number of additional statements in the FFT subroutine, so it seems one should answer why not enjoy the time savings? In some cases additional length in the program may be a completely negligible factor compared to program execution time.

This research has concerned itself with only a small part of the fast Fourier transform algorithm, namely eliminating certain unnecessary machine multiplications in the algorithm. Gentlemen and Sande[5] have compiled a number of uses for the FFT and suggest that
we may expect surprising and beneficial applications of the finite discrete Fourier transform to arise in the future.

Perhaps one area which might be of interest would be a comparison of the error between the program given in this paper and a program which does not eliminate the suggested multiplications. Intuitively one might expect a slight gain in accuracy; after all, we do eliminate some of the arithmetic when we eliminate the suggested multiplications. This matter has not yet been investigated.
APPENDIX I

THE RELATION OF THE FINITE DISCRETE FOURIER TRANSFORM TO
THE FOURIER INTEGRAL TRANSFORM

In what follows we will discuss the relation between
the DFT and the Fourier integral [4]. Consider the
Fourier transform pair

\[
a(f) = \int_{-\infty}^{\infty} X(t) e^{2\pi jft} dt \quad X(t) = \int_{-\infty}^{\infty} a(f) e^{-2\pi jft} df \quad j = -1
\]

Evaluated at points \( f = nA_f \) and \( t = nA_t \) respectively,
these become

\[
a(nA_f) = \int_{-\infty}^{\infty} X(t) e^{2\pi jnA_ft} dt \quad X(nA_t) = \int_{-\infty}^{\infty} a(f) e^{-2\pi jnA_ft} df
\]

Now let \( T_Af = 1, \quad F_At = 1 \)

\[
a(nA_f) = \sum_{K=-\infty}^{\infty} \int_{K}^{(K+1)T} X(t) e^{2\pi jnA_ft} dt,
X(nA_t) = \sum_{K=-\infty}^{\infty} a(f) e^{-2\pi jnA_ft} df
\]

Make the substitutions \( t = \tilde{t} + kT, \quad f = \tilde{f} + kF \) and we have

\[
a(nA_f) = \sum_{K=-\infty}^{\infty} \int_{0}^{T} X(\tilde{t} + kT) e^{2\pi jn\tilde{t}/T} d\tilde{t},
X(nA_t) = \sum_{K=-\infty}^{\infty} a(\tilde{f} + kF) e^{-2\pi jn\tilde{f}/F} d\tilde{f}
\]

or

\[
a(nA_f) = \int_{0}^{T} X_p(\tilde{t}) e^{2\pi jn\tilde{t}/T} d\tilde{t}, \quad X(nA_t) = \int_{0}^{\infty} a(\tilde{f}) e^{-2\pi jn\tilde{f}/F} d\tilde{f}
\]
where

\[ X_p(t) = \sum_{K=-\infty}^{\infty} X(t + KT), \quad a_p(f) = \sum_{K=-\infty}^{\infty} a(f + KF) \]

that is, \( X_p \) is a "aliased" version of \( X \), the shifting being around \( T \). Similarly for \( a_p \), with the shifting around \( F \). Since \( X_p \) and \( a_p \) are periodic, they can be expressed as Fourier series.

\[ X_p(t) = \frac{1}{T} \sum_{\ell=-\infty}^{\infty} a(\ell \Delta f) e^{-2\pi j n \Delta t / T}, \]

\[ a_p(n \Delta f) = \frac{1}{F} \sum_{\ell=-\infty}^{\infty} X(\ell \Delta t) e^{2\pi j n \Delta t / F} \]

Define \( \Delta t / T = \Delta f / F = 1/N \). Now the exponential is periodic in \( N \), thus

\[ X_p(n \Delta t) = \frac{1}{T} \sum_{\ell=0}^{N-1} \left[ \sum_{K=-\infty}^{\infty} a(\ell \Delta f + KF) \right] e^{-2\pi j \frac{n \ell}{N}}, \]

\[ a_p(j \Delta t) = \frac{1}{F} \sum_{\ell=0}^{N-1} \left[ \sum_{K=-\infty}^{\infty} X(\ell \Delta t + KT) \right] e^{2\pi j \frac{n \ell}{N}} \]

but remembering the definition of \( X_p \) and \( a_p \), this is just the DFT of \( x_p(\ell \Delta t) \) and the IDFT of \( a_p(\ell \Delta f) \), provided we take \( F = 1, \; T = N \).

The last two equations show very clearly how to choose the number of samples and how to choose the sampling interval for a given situation. Consider a specific example. Suppose that \( X(t) \) is the output of a microphone which represents a sound of duration \( T_0 \). Suppose that we are interested in obtaining the Fourier transform of \( X \). Obviously we
choose $T = T_0$ and choose the sampling interval $\Delta t$ by deciding what $F$ should be to keep the aliasing distortion small. In this particular case, we might choose $F = 20\text{kHz}$ and if $T = 0.1 \text{sec}$, then $N = 2000$, $\Delta t = 50 \mu s$.

The DFT of the values $X(l\Delta t)$, $l = 0, \ldots, 2000$ will be "almost" equal to the Fourier transform of $X$ evaluated at points $f = j\Delta f$, $j = 0, \ldots, 2000$. Any error will be due to the aliasing effect if the transform of $X$ has spectral components above $10 \text{kHz}$. 
APPENDIX II
USER'S GUIDE TO FFT

The following is a list of subroutines developed for computing DFT's on the Rice University IBM 7040. They are written in Fortran IV. Only the minor modification of eliminating the complex arithmetic and logical IF statements is necessary to make the programs work with Fortran II compilers.

### Fast Fourier Transform Subroutines

**FFT** -- Appropriately enough this is the main subroutine of the group of subroutines used to calculate DFT's. This subroutine performs the discrete Fourier transformation on the data. The length of the data sequence must be a power of two, that is the length must be $N = 2^M$, $M$ an integer $\geq 3$. Before FFT is called, the data must be put in bit reversed order [3]. To do this, subroutine REVBIN must be called. Subroutine FFT operates on the data in place, thus if the data was originally stored in complex array $X$, after calling FFT, $X$ will contain the complex coefficients of the DFT of $X$.

Since FFT is a subroutine written in FORTRAN IV, storage allocation must be taken care of in the main program. This is very easy to do. If the length of the sequence to be transformed is $N = 2^M$, the main program must contain the following statements at the first of
the program:

```
COMPLEX X(N)
DIMENSION SSIN (N/4 + 1)
INTEGER I(M), J(M)
```

X is, of course, the complex vector to be transformed.

SSIN is the name of a vector which contains the necessary sine values for the transformation. I and J are integer vectors used by FFT and REVBIN.

REVBIN -- This subroutine reverses the binary bit order of the data. Calling REVBIN twice produces data in original order.

SEPAR -- This subroutine is used when real data is being transformed. If we want to transform two real arrays of length N, say Y and Z, we store Y in the real part of X, Z in the imaginary part of X, call REVBIN, then FFT, then SEPAR. X will then contain the coefficients of the DFT of A and B, stored in the following way:

<table>
<thead>
<tr>
<th>Re(X)</th>
<th>Im(X)</th>
<th>Y→A, Z→B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_0</td>
<td>B_0</td>
<td></td>
</tr>
<tr>
<td>Re(A_1)</td>
<td>Im(A_1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Re(A_{N/2-1})</td>
<td>Im(A_{N/2-1})</td>
<td></td>
</tr>
<tr>
<td>A_{N/2}</td>
<td>B_{N/2}</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Re}(B_{N/2-1}) & & \text{Im}(B_{N/2-1}) \\
\vdots & & \vdots \\
\vdots & & \vdots \\
\text{Re}(B_1) & & \text{Im}(B_1)
\end{align*}
\]

where \( B_0 \) is stored in the imaginary part of \( X_0 \) and \( B_{N/2} \) is stored in the imaginary part of \( X_{N/2} \). It was shown in Appendix 3 that \( A_o, B_o, A_{N/2}, B_{N/2} \) are real for real input \( Y \) and \( Z \), and that \( A \) and \( B \) have complex conjugate symmetry around \( N/2 \). This subroutine uses Theorems Ia (see Appendix III).

**RELSAV** -- This subroutine makes use of Theorem IIa.

Subroutine SEPAR can be used when we have two real sequences to be transformed, but if we have only one real sequence we can still save computational effort by using Theorem IIa. If we have a real input vector of length \( 2N \), we store it alternately in \( X \) as in Theorem IIa, then call in order \( \text{REVBIN}, \text{FFT}, \text{SEPAR}, \text{RELSAV} \). If \( A \) is the transform of \( X \), \( A \) will be stored as

\[
\begin{align*}
\text{Re}(X) & & \text{Im}(X) \\
A_o & & A_N \\
\text{Re}(A_1) & & \text{Im}(A_1) \\
\vdots & & \vdots \\
\text{Re}(A_{N-1}) & & \text{Im}(A_{N-1})
\end{align*}
\]

since \( A_o, A_N \) real.
Subroutines to take advantage of the savings shown in Theorems Ib and IIb in Appendix III are not yet available.

EXAMPLE:

Suppose we wish to find the DFT of a real input sequence \( B = (B_0, \ldots, B_{2N-1}), N = 256. \)

We could accomplish this in the following way:

**Heading**

```
COMPLEX X(256)
DIMENSION SSIN(65)
INTEGER I(8), J(8)
M = 8
```

```
X(1) = B_0 + jB_1
X(2) = B_2 + jB_3
X(256) = B_{510} + jB_{511}
```

CALL REVBIN (X,M,I,J)
CALL FFT (X,M,I,SSIN)
CALL SEPAR (X,M,I)
CALL RELSAV (X,M,I)

After this last statement, \( X \) will contain the coefficients of \( B \) as explained.
Subroutine REVBIN (X,M,I,J)

Integer I(1), J(1)
Complex X(1), T

I(1) = 1
J(1) = 0

DO 1 L = 2, M
1 I(L) = 2*I(M)

J(L) = 0
ISTP = 2*I(M)

JFOR = 0

IF (JFOR .EQ. ISTM) Go to 5
JSUM = 0

J(M) = J(M) + I(M)

DO 3, LREV = 1, M
L = M - LREV + 1

IF(J(L) .GT. I(L) Go to 6
JSUM = JSUM + J(L)
Go to 3

J(L-1) = J(L-1) + I(L-1)
J(L) = 0
CONTINUE

IF(JSUM .GT. JFOR) Go to 4
Go to 2

T = X(JFOR + 1)
X(JFOR + 1) = X(JSUM + 1)
X(JSUM + 1) = T
Go to 2
CONTINUE

END
SUBROUTINE FFT(X ,M, I, SSIN)
COMPLEX X(L), W, B,
INTEGER I(1), J(1)
DIMENSION SSIN(1)
N=2*I(M)
K7=I(M-1)+1
A=0.70710678
DO 16 K=1,M
KSPAN=I(K)
KJUMP=2*KSPAN
K22=M-K+2
DO 16 KKK=1,KSPAN
IF(KKK.EQ.1) Go to 10
KEXP=I(K22-1)*(KKK-1)
IF(KEXP.EQ.I(M-1)) Go to 11
IF(KEXP.EQ.I(M-2)) Go to 12
IF(KEXP.EQ.#*I(M-2)) Go to 13
Go to 23
C Type 1 Multiply
DO 14 L=1,N,KJUMP
K8=KSPAN+L
B=X(K8)
X(K8)=X(1)-B
X(L)=X(L)+B
Go to 16
C Type J Multiply
DO 15 L=KKK,N,KJUMP
K8=KSPAN+L
B=CMPLX(-AIMAG(X(K8)),REAL(X(K8)))
X(K8)=X(L)-B
X(L)=X(L)+B
Go to 16
C Type 1+J Multiply
DO 20 L=KKK,N,KJUMP
K8=KSPAN+L
B=CMPLX(A*(REAL(X(K8))-AIMAG(X(K8))),
1A*(REAL(X(K8))+AIMAG(X(K8))))
X(K8)=X(L)-B
X(L)=X(L)+B
Go to 16
C Type -1+J Multiply
DO 20 L=KKK,N,KJUMP
K8=KSPAN+L
B=CMPLX(-(A*REAL(X(K8))+AIMAG(X(K8))),
1A*(REAL(X(K8))-AIMAG(X(K8))))
X(K8)=X(L)-B
X(L)=X(L)+B
Go to 16
IF(KEXP.LE.I(M-1)) Go to 22
K50=KEXP-I(M-1)
K51=I(M)-KEXP
W=CMPLX(SSIN(K50+1),SSIN(K51+1))
Go to 19
K52=I(M-1)-KEXP
W=CMPLX(SSIN(K52+1),SSIN(KEXP+1))
DO 16 L=KKK,N,KJUMP
K8=KSPAN+L
B=W*X(K8)
X(K8)=X(L) - B
X(L)=X(L)+B
CONTINUE
RETURN
END
SUBROUTINE SEPAR (X,M,I)
COMPLEX X(1), C,D
INTEGER I(1)
NP2=2*I(M)+2
NEND=I(M)
DO 1 L=2, NEND
LREV=NP2-L
D=CONJG(X(LREV))
C=D-X(L)
X(LREV)=0.5*CMPLX(-AIMAG(C), REAL(C))
1 X(L)=0.5*(D+X(L))
RETURN
END
SUBROUTINE RELSAV(X,M,I,SSIN)
  DIMENSION SSIN(1)
  COMPLEX X(1),B
  INTEGER I(1)
  N=2*I(M)
  SCOS=2.0*COS(3.1415927/FLOAT(N))
  X(1)=CMPLX((REAL(X(1))+AIMAG(X(1))),
  1(REAL(X(1))-AIMAG(X(1))))
  KSTOP=I(M-1)
  KSTP=KSTOP+1
  DO 14 K=1,KSTOP,2
   J=((K-1)/2)+1
   KL=N-K+1
   KU=1+K
   WI=(SSIN(J)+SSIN(J+1))/SCOS
   KL=I(M-1)-J+1
   WR=(SSIN(K1+1)+SSIN(K1))/SCOS
   L=1
   Go to 10
 10   R=WI
   WI=WR
   WR=R
   KL=I(M)+1+K
   KU=I(M)+1-K
   L=2
   Go to 10
 12   WR=SSIN(K1)
   WI=SSIN(J+1)
   KL=N-K
   KU=K+2
   L=3
   Go to 10
 13   IF(KU.EQ.KSTP) Go to 15
   R=WI
   WI=WR
   WR=R
   KL=I(M)+2+K
   KU=I(M)-K
   L=4
   Go to 10
 14   CONTINUE
 15   CONTINUE
 16   Go to 15
 10   R=REAL(X(KL))
   A=AIMAG(X(KL))
   B=CMPLX((WR*B-WI*A),(WI*B+WR*A))
   X(KL)=CONJM(X(KU)-8)
   X(KU)=X(KU)+B
   Go to (11, 12, 13, 14),L
 15   CONTINUE
 16   RETURN
 17   END
SUBROUTINE GENSIN(M, I, SSIN)
INTEGER I(1)
DIMENSION SSIN(1)
K7 = I(M-1) + 1
SSIN(1) = 0.0
SSIN(K7) = 1.0
DO 2 K = 3, M
K22 = M - K + 2
K1 = I(K-2)
SCOS = 2.0 * COS(3.1415927 / FLOAT(I(K)))
DO 2 NN = 1, K1
K3 = I(K22) * (NN - 1) + 1
K4 = K3 + I(K22 - 1)
K5 = K3 + I(K22)
2 SSIN(K4) = (SSIN(K3) + SSIN(K5)) / SCOS
RETURN
END
APPENDIX III
THE REAL FAST FOURIER TRANSFORM

Two special cases of the FFT arise quite often. The first case is when real data is to be transformed. The second case is when the data to be transformed has complex conjugate symmetry around \( X(N) \) i.e., if the input vector is \( X = (X(0), \ldots, X(2N-1)) \) then \( X(j)^* = X(2N-k) \). This second case follows quite naturally from the first since this symmetry is always present when real data is transformed, just as the transform of a real continuous time signal has complex conjugate symmetry around the origin, i.e. \( f(t) \) real \( \Leftrightarrow F(-w) = F^*(w) \), where \( f \Leftrightarrow F \). Thus the two cases are complimentary so that it is quite reasonable to approach both problems at the same time. In the proofs which follow, this dual nature will be emphasized by the method of designation of the lemmas and theorems.

**Lemma Ia.** The DFT, \( A = (A(0), \ldots, A(2N-1)) \), of a real input sequence \( X = (X(0), \ldots, X(2N-1)) \) has a complex conjugate symmetry around \( A(N) \).

**PROOF:**

\[
A(\ell) = \sum_{K=0}^{2N-1} X(K) W(K\ell/2) = \sum_{K=0}^{2N-1} X(K) W(K(2N-\ell)/2) = A(2N-\ell)
\]

Q.E.D.

*Most of these theorems are generally known, except perhaps, with the exception of Theorem IIb.*
Lemma Ib. The DFT, \( A = (A(0), \ldots, A(2N-1)) \) of a complex input sequence \( X = (X(0), \ldots, X(2N-1)) \) which has complex conjugate symmetry around \( X(N) \), i.e. \( X(K) = X^{\ast}(2N-K) \), is real.

**Proof:**

\[
A(\ell) = \sum_{K=0}^{2N-1} X(K) W(K\ell/2) = \sum_{K=0}^{2N-1} X^{\ast}(2N-K) W(K\ell/2)
\]

\[
= \sum_{p=2N}^{1} X^{\ast}(p) W(\ell(2N-p)/2) = \sum_{p=1}^{2N} X^{\ast}(p) W(-p\ell) = A^{\ast}(\ell)
\]

since \( X(0) = X(2N) \).
**Theorem Ia.**

The transform of two real sequences of length $N$ can be calculated, with only minor additional calculations, by performing a single complex FFT of dimension $N$.

**PROOF:**

Let $X = (X(0), \ldots, X(N-1))$, $Y = (Y(0), \ldots, Y(N-1))$ be the two real sequences. Form $Z(K) = X(K) + jY(K)$ for $K = 0, \ldots, N-1$. Let $A$ be the DFT of $X$ and $B$ be the DFT of $Y$. By Lemma Ia, $A$ and $B$ have complex conjugate symmetry around $N/2$. Let $C$ be the DFT of $Z$. Then $C(K) = A(K) + jB(K)$. By the symmetry of $A$ and $B$ we have

$$2A(K) = C(K) + C^*(N-K) \quad K = 0, \ldots, N/2$$
$$2jB(K) = C(K) - C^*(N-K) \quad K = 0, \ldots, N/2$$

Of course $A(0), A(N/2), B(0), B(N/2)$ are real since we interpret $C(N) = C(0)$.

**Theorem Ib.**

Two complex sequences of length $N$ with complex conjugate symmetry around $N/2$ can be transformed, with only minor additional calculation, by using the FFT of length $N$.

**PROOF:**

Let $X = (X(0), \ldots, X(N-1))$, $Y = (Y(0), \ldots, Y(N-1))$ be the two input sequences. Generate
\[ Z(K) = X(K) + jY(K) \quad K = 0, \ldots, N/2-1 \]
\[ Z(N-K) = X^*(K) + jY^*(K) \quad K = 1, \ldots, N/2 \]

This defines \( Z = (Z(0), \ldots, Z(N-1)) \). Note \( Z(K) + Z^*(N-K) = 2X(K) \) and \( Z(K) - Z^*(N-K) = 2jY(K) \). Let

the DFT of \( Z \) be \( A \) and observe

\[
2 \text{RE}(A(\ell)) = A(\ell) + A^*(\ell) = \sum_{K=0}^{N-1} Z(K) W(K\ell) + \sum_{K=0}^{N-1} Z^*(K) W(-K\ell) \\
= \sum_{K=0}^{N-1} Z(K) W(K\ell) + \sum_{M=N}^{N-1} Z^*(N-M) W(-(N-M)\ell) \\
= \sum_{K=0}^{N-1} Z(K) W(K\ell) + \sum_{M=1}^{N} Z^*(N-M) W(K\ell) = 2 \sum_{K=0}^{N-1} X(K) W(K\ell) \\
2j \text{IM}(A(\ell)) = A(\ell) - A^*(\ell) = \sum_{K=0}^{N-1} Z(K) W(K\ell) - \sum_{K=0}^{N-1} Z^*(K) W(-K\ell) \\
= 2j \sum_{K=0}^{N-1} Y(K) W(K\ell) \quad (\text{similar to above})
\]

Thus the transform of \( X \) is the real part of \( A \) and the transform of \( Y \) is the imaginary part. \( \text{Q.E.D.} \)

Theorem IIa.

The DFT of a real input vector of length \( 2N \) can be calculated, with only minor additional calculations, by using the FFT of dimension \( N \).

PROOF:

Let \( X = (X(0), \ldots, X(2N-1)) \)
form \( Y(\ell) = X(2\ell) + jX(2\ell + 1) \).
Transform and separate $Y$, as in Theorem Ia, calling $A$ the transform of $X(2\ell)$, $\ell = 0, \ldots, N-1$, $B$ the transform of $X(2\ell+1)$, $\ell = 0, \ldots, N-1$. We seek the transform $C$,

$$C(\ell) = \sum_{K=0}^{2N-1} X(K) W(K\ell/2) = \sum_{K=0}^{2N-2} X(K) W(K\ell/2) + \sum_{K=1}^{2N-1} X(K) W(K\ell/2)$$

$$= \sum_{m=0}^{N-1} X(2m) W(m\ell) + \sum_{m=0}^{N-1} X(2m+1) W(m\ell) W(\ell/2)$$

$$= A(\ell) + W(\ell/2) B(\ell) \quad \ell = 0, \ldots, M \quad \text{Q.E.D.}$$

Note that in this case the "minor additional calculations" are slightly more complicated than in the case of two real sequences since the additional trigonometric values necessary to evaluate $W(\ell/2)$ must be generated.

**Theorem IIb**

The DFT of a complex input sequence $X = (X(0), \ldots, X(\ell N-1))$ with complex conjugate symmetry around $X(N)$, can be calculated, with only minor additional computing effort, by using the FFT of dimension $N$.

**PROOF:**

Let $A(K) = X(K) + X^*(N-K)$ \quad $K = 0, \ldots, N/2, \ldots N$

$$B(K) = W(K/2) (X(K) - X^*(N-K))$$

Since $X(N)$ is real $A(0) = A(N)$, $B(0) = B(N)$ real.

$$A^*(K) = X(K) + X(N-K) = X^*(N-(N-K)) + X^*(N-K) = A(N-K)$$

$$B^*(K) = W(-K/2) (X^*(K) - X(N-K)) = W(N/2) W(-K/2) (X(N-K) - X^*(N-(N-K))) = B(N-K)$$
Thus $A$ and $B$ satisfy the hypotheses of Theorem Ib.

Transform them according to that theorem, we know

$$\text{Re}(A(\ell)) = \sum_{K=0}^{N-1} A(K)W(K\ell) = \sum_{K=0}^{N-1} (X(j) + X^*(N-j)W(K\ell))$$

$$= \sum_{r=0}^{N-1} X(r)W(\ell r) + \sum_{r=0}^{N-1} X^*(N-r)W(\ell(N-r)) \text{ let } q=N-r$$

$$= \sum_{r=0}^{N-1} X(r)W(\ell r) + \sum_{q=0}^{N-1} X^*(q)W(-\ell q)$$

$$= \sum_{r=0}^{N-1} X(r)W(\ell r) + \sum_{r=0}^{N-1} X(2N-q)W(\ell(2N-q)) \text{ let } r=2N-q$$

$$= \sum_{r=0}^{N-1} X(r)W(\ell r) + \sum_{r=N}^{2N-1} X(r)W(\ell r)$$

$$= \sum_{r=0}^{2N-1} X(r)W(\ell r) = Z(2\ell)$$

where $Z$ is the DFT of $X$. Quite similarly,

$$\text{Im}(A(\ell)) = Z(2\ell +1) \quad \text{Q.E.D.}$$
APPENDIX IV

ALGEBRA OF THE FAST FOURIER TRANSFORM WHEN $N = 2^M$

We now look at the general power of $2$ program where $N = 2^M$.

Considering the definition (1), we can let

$$\ell = \ell_0 2^0 + \ell_1 2^1 + \ldots + \ell_{M-1} 2^{M-1} \quad \ell_i = 0 \text{ or } 1 \quad i = 0, \ldots, M-1$$

$$k = k_0 2^0 + k_1 2^1 + \ldots + k_{M-1} 2^{M-1} \quad k_i = 0 \text{ or } 1 \quad i = 0, \ldots, M-1$$

Here we have switched to binary representation of $k$ and $\ell$, thus (1) becomes

$$A(\ell_0 2^0 + \ell_1 2^1 + \ldots + \ell_{M-1} 2^{M-1}) = \sum_{k=0}^{1} \sum_{\ell_0=0}^{1} \sum_{k_{M-1}=0}^{1} X(k_0 2^0 + k_1 2^1 + \ldots + k_{M-1} 2^{M-1}) W^{\ell_0 2^0 + \ell_1 2^1 + \ldots + \ell_{M-1} 2^{M-1}} W^{(\ell k)} \quad (3)$$

$$\ell k = (\ell_0 + \ell_1 2^1 + \ldots + \ell_{M-1} 2^{M-1})(k_0 + k_1 2^1 + \ldots + k_{M-1} 2^{M-1})$$

$$= \ell_0 k_0 + (\ell_1 k_0 + k_1 \ell_0) 2^1 + (\ell_1 k_1 + \ell_2 k_0 + k_2 \ell_0) 2^2 + \ldots + k_{M-1} \ell_{M-1} 2^{2(M-1)}$$

We now use the property $W^{(k+\ell)} = W^{(k)} W^{(\ell)}$ to factor the expression (3) in a particular way. We move as many of the $W$ terms in equation (3) as far to the left as possible.

Thus terms involving $k_1$ will be immediately to the right of the sum on $k_1$, etc. This is the Cooley-Tukey factorization. It is the same as collecting terms in the expansion for $\ell k$ on the $k_1$'s. Therefore, equation (3) becomes
\[ A(t_0^{2^0} + t_1^{2^1} + \ldots + t_{M-1}^{2^{M-1}}) = \sum_{k_0=0}^1 W(k_0 t_{M-1}^{2^{M-1}}) \]

\[
\left[ W(k_0(t_0^{2^1} + \ldots + t_{M-2}^{2^{M-2}})) \right] \sum_{k_1=0}^{1} W(k_1 t_{M-2}^{2^{M-2}})
\]

\[
\left[ W(k_1(t_0^{2^1} + \ldots + t_{M-3}^{2^{M-3}})) \right] \sum_{k_2=0}^{1} \ldots \sum_{k_{M-2}=0}^{1} W(k_{M-2} t_{1}^{2^{M-1}})
\]

\[
\left[ W(t_0^{k_{M-2}^{2^{M-2}}}) \right] \sum_{k_{M-1}=0}^{1} X(k_0^{2} + 2k_1^{2} + \ldots + k_{M-1}^{2} t_{M-1}^{2^{M-1}}) W(k_{M-1} t_{0}^{2^{M-1}})
\]

Here all of the "twiddle factors" have been put in brackets.

Now considering

\[
\sum_{k_{M-1}=0}^{1} X(k_0^{2} + 2k_1^{2} + \ldots + k_{M-1}^{2} t_{M-1}^{2^{M-1}}) W(k_{M-1} t_{0}^{2^{M-1}})
\]

this is a \(2 \times 2\) DFT for each selection of \(k_i's, k_0, \ldots, k_{M-2}\), so there are \(2^{M-1}\) of them and the result is a \(2 \times 2^{M-1}\) array indexed on \(t_0, t_1, \ldots, k_{M-1}\).

We multiply by the \(N\) twiddle factors \(W_0^{k_0^{2^{M-2}}}\) (either 1 or \(j\)), and the array is still indexed on \(t_0, t_1, \ldots, k_{M-1}\). At the end of the second iteration we have a \(2^2 \times 2^{M-2}\) array indexed on \(t_0, t_1, \ldots, k_{M-3}\). Then we multiply this array by the twiddle factor and continue this procedure to the last sum.
APPENDIX V
COUNTING THE NECESSARY MULTIPLICATIONS WHEN N = 2^M

Referring to the factorization given in Equation (5) of the text, in the first step the "twiddle factors" are

\[ W(\ell_o k_{M-2} 2^{M-2}) \]

for data indexed on \( \ell_o \) and \( (k_o, \ldots, k_{M-2}) \); after the second step, the twiddle factors are

\[ W(k_{M-3}(\ell_o 2^{M-3} + \ell_1 2^{M-2})) \]

and after the next to last step, the twiddle factors are

\[ W(k_o (\ell_o + \ell_1 2^1 + \ldots + \ell_{M-1} 2^{M-1})) \]

Now we can write out a table which will be called the "fundamental table of exponentials". We can ignore the half of the table for which the \( k_i \)'s, \( i = 0, \ldots, M-2 \) are zero since \( W^0 = 1 \). Assuming that the \( k_i \)'s are 1 then, we can write the table for the program of dimension \( N = 2^M \).

\[
\begin{align*}
k_{M-2} &= 1 & W(0) W(2^{M-2}) W(0) W(2^{M-2}) & \ldots & \ldots & W(0) W(2^{M-2}) \\
k_{M-3} &= 1 & W(0) W(2^{M-3}) W(2 \cdot 2^{M-3}) W(3 \cdot 2^{M-3}) & \ldots & \ldots & W(3 \cdot 2^{M-3}) \\
& \vdots & & & & \vdots \\
k_o &= 1 & W(0) W(1) W(2) & \ldots & \ldots & \ldots & \ldots & W(2^{M-1}-1)
\end{align*}
\]
Now it is obvious that $W(0) = 1$, $W(2^{M-2}) = W(N/4) = j$ and $W(2^{M-3}) = .707(1+j)$ and $W(3 \cdot 2^{M-3}) = .707(-1+j)$ are special cases where there are either no complex multiplications or $1/2$ complex multiplications ($2$ real multiplications). It is very easy to see from the table how often these special cases appear, indeed, we can even count the multiplications required. The row where $K_{M-2} = 1$ has only $1$'s or $j$'s; the row where $K_{M-3} = 1$ has only $1$'s, $j$'s, $0.707(1+j)$ or $0.707(-1+j)$. If we carry out this analysis, we find that the number of real multiplications is

$$4(M-1) \frac{N}{2} - 4 \sum_{\ell=2}^{M-2} 2^{M-\ell} - 4 \sum_{\ell=2}^{M-3} 2^{M-\ell} - 4 \sum_{\ell=3}^{M} 2^{M-\ell}$$

(4)

where the first sum represents the number of $W^0$'s, the second sum is the number of $W^N/4$'s, the third sum is the combined savings from the number of $W^N/8$'s and $W^3N/8$'s. Since $\sum_{\ell=2}^{M-2} 2^{M-\ell} = \sum_{\ell=0}^{M-1} 2^{M-\ell} = 2^{M-1} - 1$, (4) becomes $2N(M - 7/2) + 12$. 

REFERENCES


