FAST FOURIER TRANSFORMS: A REVIEW

George Wolberg

Department of Computer Science
Columbia University
New York, NY 10027
wolberg@cs.columbia.edu

September 1988
Technical Report CUCS-388-88

ABSTRACT

The purpose of this paper is to provide a detailed review of the Fast Fourier Transform. Some familiarity with the basic concepts of the Fourier Transform is assumed. The review begins with a definition of the discrete Fourier Transform (DFT) in section 1. Directly evaluating the DFT is demonstrated there to be an $O(N^2)$ process.

The efficient approach for evaluating the DFT is through the use of FFT algorithms. Their existence became generally known in the mid-1960s, stemming from the work of J.W. Cooley and J.W. Tukey. Although they pioneered new FFT algorithms, the original work was actually discovered over 20 years earlier by Danielson and Lanczos. Their formulation, known as the Danielson-Lanczos Lemma, is derived in section 2. Their recursive solution is shown to reduce the computational complexity to $O(N \log_2 N)$.

A modification of that method, the Cooley-Tukey algorithm, is given in section 3. Yet another variation, the Cooley-Sande algorithm, is described in section 4. These last two techniques are also known in the literature as the decimation-in-time and decimation-in-frequency algorithms, respectively. Finally, source code, written in C, is provided in the appendix.

1. DISCRETE FOURIER TRANSFORM

Consider an input function $f(t)$ sampled at discrete time intervals. This yields a list of numbers, f_k, where $0 \le k \le N-1$. For generality, the input samples are taken to be complex numbers, i.e., having real and imaginary components. The discrete Fourier Transform of f is defined as

$$F_n = \sum_{k=0}^{N-1} f_k \, e^{-2\pi i n k / N} \qquad 0 \le n \le N-1 \tag{1a}$$

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} F_k \, e^{2\pi i n k / N} \qquad 0 \le n \le N-1 \tag{1b}$$

Equations (1a) and (1b) define the forward and inverse DFTs, respectively. Since both DFTs share the same cost of computation, we shall confine our discussion to the forward DFT, and refer to it only as the DFT.

The DFT serves to map the N input samples of f into the N frequency terms in F. From Eq. (1a), we see that each of the N frequency terms are computed by a linear combination of the N input samples. Therefore, the total computation requires N^2 complex multiplications and $N(N-1)$ complex additions. The straightforward computation of the DFT thereby gives rise to an $O(N^2)$ process. This can be seen more readily if we rewrite Eq. (1a) as

$$F_n = \sum_{k=0}^{N-1} f_k \, W^{nk} \qquad 0 \le n \le N-1 \tag{2}$$

where

$$W = e^{-2\pi i / N} = \cos(-2\pi/N) + i \sin(-2\pi/N) \tag{3}$$

For reasons described later, we assume that

$$N = 2^r$$

where r is a positive integer. That is, N is a power of 2.

Equation (2) casts the DFT as a matrix multiplication between the input vector f and the two-dimensional array composed of powers of W. The entries in the 2-D array, indexed by n and k, represent the N equally spaced values along a sinusoid at each of the N frequencies. Since straightforward matrix multiplication is an $O(N^2)$ process, the computational complexity of the DFT is bounded from above by this limit.

In the next section, we show how the DFT may be computed in $O(N \log_2 N)$ operations with the Fast Fourier Transform (FFT), as originally derived over forty years ago. By properly decomposing Eq. (1a), the reduction in proportionality from N^2 to $N \log_2 N$ multiply/add operations represents a significant saving in computation effort, particularly when N is large.

2. DANIELSON-LANCZOS LEMMA

In 1942, Danielson and Lanczos derived a recursive solution for the DFT. They showed that a DFT of length $N^{\dagger}$ can be rewritten as the sum of two DFTs, each of length $N/2$, where N is an integer power of 2. The first DFT makes use of the even-numbered points of the original N; the second uses the odd-numbered points. The following proof is offered.

$$F_n = \sum_{k=0}^{N-1} f_k\, e^{-2\pi i n k/N} \tag{4}$$

$$= \sum_{k=0}^{(N/2)-1} f_{2k}\, e^{-2\pi i n(2k)/N} + \sum_{k=0}^{(N/2)-1} f_{2k+1}\, e^{-2\pi i n(2k+1)/N} \tag{5}$$

$$= \sum_{k=0}^{(N/2)-1} f_{2k}\, e^{-2\pi i n k/(N/2)} + W^n \sum_{k=0}^{(N/2)-1} f_{2k+1}\, e^{-2\pi i n k/(N/2)} \tag{6}$$

$$= F_n^e + W^n F_n^o \tag{7}$$

Equation (4) restates the original definition of the DFT. The summation is expressed in Eq. (5) as two smaller summations consisting of the even- and odd-numbered terms, respectively. In order to properly access the data, the index is changed from k to $2k$ and $2k+1$, and the upper limit becomes $(N/2)-1$. These changes to the indexing variable and its upper limit gives rise to Eq. (6), where both sums are expressed in a form equivalent to a DFT of length $N/2$. The notation is simplified further in Eq. (7). There, F_n^e denotes the n^{th} component of the Fourier Transform of length $N/2$ formed from the even components of the original f, while F_n^o is the corresponding transform derived from the odd components.

The expression given in Eq. (7) is the central idea of the Danielson-Lanczos Lemma and the decimation-in-time FFT algorithm described later. It presents a divide-and-conquer solution to the problem. In this manner, solving a problem (F_n) is reduced to solving two smaller subproblems (F_n^e and F_n^o). However, a closer look at the two sums, F_n^e and F_n^o, illustrates a potentially troublesome deviation from the original definition of the DFT: $N/2$ points of f are used to generate N points. (Recall that n in F_n^e and F_n^o is still made to vary from 0 to $N-1$). Since each of the subproblems appears to be no smaller than the original problem, this would thereby seem to be a wasteful approach. Fortunately, there exists symmetries which we exploit to reduce the computational complexity.

The first simplification is found by observing that F_n is periodic in the length of the transform. That is, given a DFT of length N, $F_{n+N} = F_n$. The proof is given below.

† This is also known as an N-point DFT.

$$\begin{aligned} F_{n+N} &= \sum_{k=0}^{N-1} f_k \, e^{-2\pi i(n+N)k/N} \\ &= \sum_{k=0}^{N-1} f_k \, e^{-2\pi inkl/N} \, e^{-2\pi iNk/N} \\ &= \sum_{k=0}^{N-1} f_k \, e^{-2\pi ink/N} \\ &= F_n \end{aligned} \tag{8}$$

In the second line of Eq. (8), the last exponential term drops out because the exponent $-2\pi iNk/N$ is simply an integer multiple of 2π and $e^{-2\pi ik} = 1$. Relating this result to Eq. (7), we note that F_n^e and F_n^o have period $N/2$. Thus,

$$\begin{aligned} F_{n+N/2}^e &= F_n^e \qquad 0 \le n < N/2 \\ F_{n+N/2}^o &= F_n^o \qquad 0 \le n < N/2 \end{aligned} \tag{9}$$

This permits the $N/2$ values of F_n^e and F_n^o to trivially generate the N numbers needed for F_n.

A similar simplification exists for the W^n factor in Eq. (7). Since W has period N, the first $N/2$ values can be used to trivially generate the remaining $N/2$ values by the following relation.

$$\begin{aligned} \cos((2\pi/N)(n+N/2)) &= -\cos(2\pi n/N) \qquad 0 \le n < N/2 \\ \sin((2\pi/N)(n+N/2)) &= -\sin(2\pi n/N) \qquad 0 \le n < N/2 \end{aligned} \tag{10}$$

Therefore,

$$W^{n+N/2} = -W^n \qquad 0 \le n < N/2 \tag{11}$$

Summarizing the above results, we have

$$\begin{aligned} F_n &= F_n^e + W^n F_n^e \qquad 0 \le n < N/2 \\ F_{n+N/2} &= F_n^e - W^n F_n^o \qquad 0 \le n < N/2 \end{aligned} \tag{12}$$

where N is an integer power of 2.

2.1. Butterfly Flow Graph

Equation (12) can be represented by the *butterfly* flow graph of Fig. 1a, where the minus sign in $\pm W^n$ arises in the computation of $F_{n+N/2}$. The terms along the branches represent multiplicative factors applied to the input nodes. The intersecting node denotes a summation. For convenience, this flow graph is represented by the simplified diagram of Fig. 1b. Note that a butterfly performs only *one* complex multiplication ($W^n F_n^o$). This product is used in Eq. (12) to yield F_n and $F_{n+N/2}$.

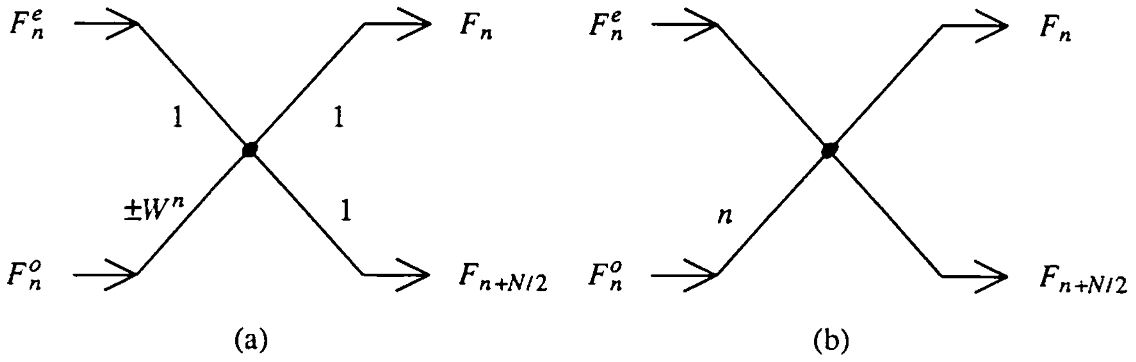

Figure 1: (a) Butterfly flow graph; (b) Simplified diagram

The expansion of a butterfly flow graph in terms of the computed real and imaginary terms is given below. For notational convenience, the real and imaginary components of a complex number are denoted by the subscripts r and i, respectively. We define the following variables.

$$\begin{aligned} g &= F_n^e \\ h &= F_n^o \\ w_r &= \cos(-2\pi n/N) \\ w_i &= \sin(-2\pi n/N) \end{aligned}$$

Expanding F_n, we have

$$\begin{aligned} F_n &= g + W^n h \\ &= [g_r + ig_i] + [w_r + iw_i]\,[h_r + ih_i] \\ &= [g_r + ig_i] + [w_r h_r - w_i h_i + iw_r h_i + iw_i h_r] \\ &= [g_r + w_r h_r - w_i h_i] + i\,[g_i + w_r h_i + w_i h_r] \end{aligned} \tag{13}$$

The real and imaginary components of $W^n h$ are thus $w_r h_r - w_i h_i$ and $w_r h_i + w_i h_r$, respectively. These terms are isolated in the computation so that they may be subtracted from g_r and g_i to yield $F_{n+N/2}$ without any additional transform evaluations.

2.2. Putting It All Together

The recursive formulation of the Danielson-Lanczos Lemma is demonstrated in the following example. Consider list f of 8 complex numbers labeled f_0 through f_7 in Fig. 2. In order to reassign the list entries with the Fourier coefficients F_n, we must evaluate F_n^e and F_n^o. As a result, two new lists are created containing the even and odd components of f. The e and o labels along the branches denote the path of even and odd components, respectively. Applying the same procedure to the newly created lists, successive halving is performed until the lowest level is reached, leaving only one element per list. The result of this recursive subdivision is shown in Fig. 2.

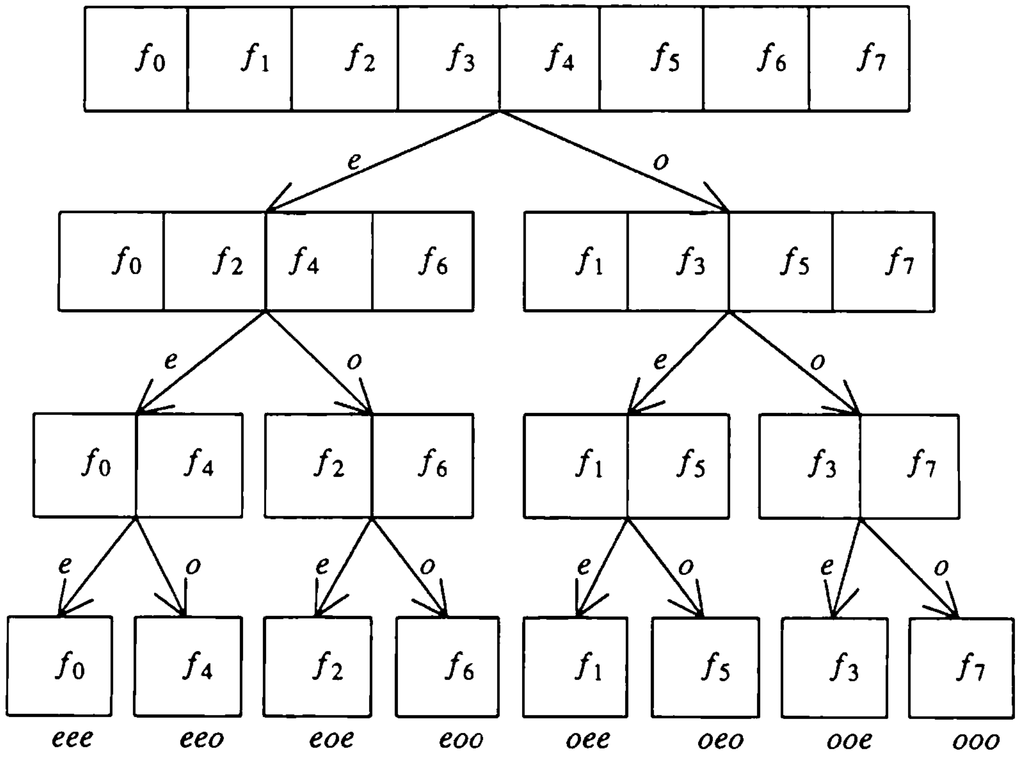

Figure 2: Recursive subdivision into even- and odd-indexed lists.

At this point, we may begin working our way back up the tree, building up the coefficients using the Danielson-Lanczos Lemma given in Eq. (12). Figure 3 depicts this process using butterfly flow graphs to specify the necessary complex additions and multiplications. Note that bold lines are used to delimit lists in the figure. Beginning with the 1-element lists, the 1-point DFTs are evaluated first. Since a 1-point DFT is simply an identity operation that copies its one input number into its one output slot, the 1-element lists remain the same.

The 2-point transforms now make use of the 1-point transform results. Next, the 4-point transforms build upon the 2-point results. In this case, N is 4, and the exponent of W is made to vary from 0 to $(N/2)-1$, or 1. In Fig. 3, all butterfly flow graphs assume an N of 8 for the W factor. Therefore, the listed numbers are normalized accordingly. For the 4-point transform, the exponents of 0 and 1 (assuming an N of 4) become 0 and 2 to compensate for the implied N value of 8. Finally, the last step is the evaluation of an 8-point transform. In general, we combine adjacent pairs of 1-point transforms to get 2-point transforms, then combine adjacent pairs of pairs to get 4-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform.

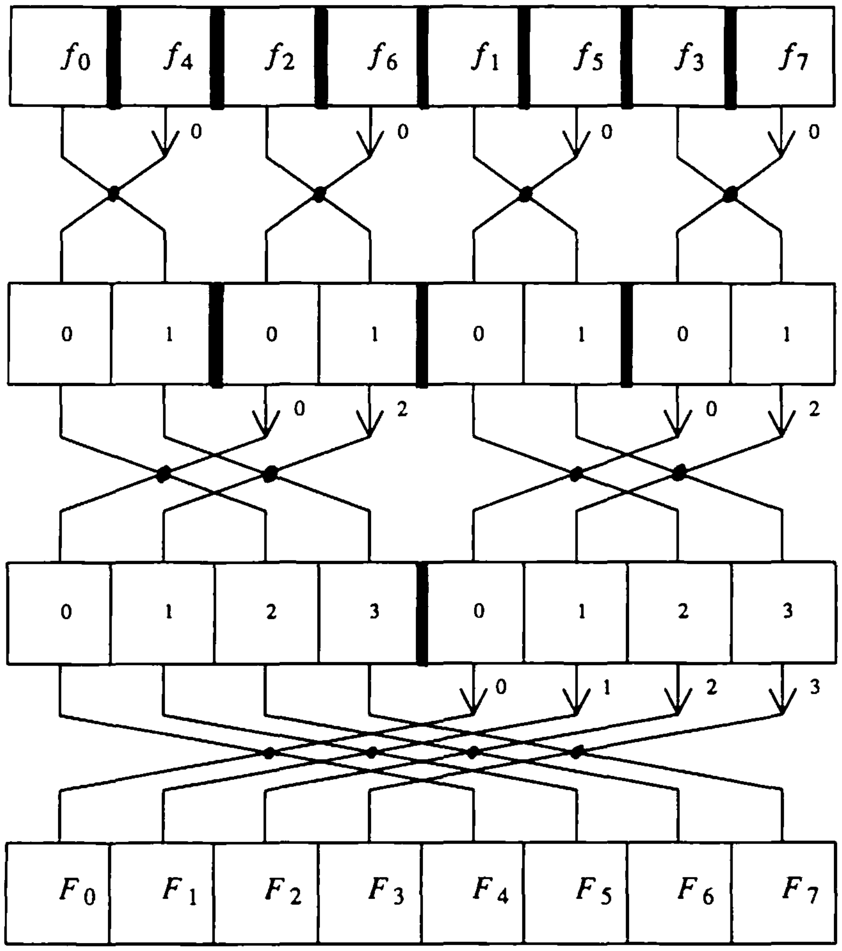

Figure 3: Application of the Danielson-Lanczos Lemma.

2.3. Recursive FFT Algorithm

The Danielson-Lanczos Lemma provides an easily programmable method for the DFT computation. It is encapsulated in Eq. (12) and presented in the FFT procedure given below.

```
Procedure FFT(N,f)
1.    If N equals 2, then do
              Begin
2.            Replace $f_0$ by $f_0+f_1$ and $f_1$ by $f_0-f_1$.
3.            Return
              End
4.    Else do:
              Begin
5.            Define $g$ as a list consisting of all points of $f$ which have an even index
              and $h$ as a list containing the remaining odd points.
6.            Call FFT(N/2, $g$)
7.            Call FFT(N/2, $h$)
8.            Replace $f_n$ by $g_n + W^n h_n$ for n=0 to $N-1$.
              End
      End
```

The above procedure is invoked with two arguments: N and f. N is the number of points being passed in array f. As long as N is greater than 2, f is split into two halves g and h. Array g stores those points of f having an even index, while h stores the odd-indexed points. The Fourier Transforms of these two lists are then computed by invoking the FFT procedure on g and h with length $N/2$. The FFT program will overwrite the contents of the lists with their DFT results. They are then combined in line 8 according to Eq. (7).

The successive halving proceeds until N is equal to 2. At that point, as observed in Fig. 3, the exponent of W is fixed at 0. Since W^0 is 1, there is no need to perform the multiplication and the results may be determined directly (line 2).

Returning to line 8, the timesavings there arises from using the $N/2$ available elements in g and h to generate the N numbers required. This is a realization of Eq. (12), with the real and imaginary terms given in Eq. (13). The following segment of C code implements line 8 in the above algorithm. Note that all variables are of type *double*.

```
ang  = 0;                      /* initialize angle */
inc  = -6.2831853 / N;         /* angle increment: $2\pi/N$ */
N2   = N / 2;
for(n=0; n<N2; n++) {
        $w_r$ = cos(ang);              /* real part of $W^n$ */
        $w_i$ = sin(ang);              /* imaginary part of $W^n$ */
        ang += inc;                /* next angle in $W^n$ */

        a = $w_r$*$h_r$[n] – $w_n$*$h_n$[n];  /* real part of $W^n h$ (Eq. 13) */
        $f_r$[n]      = $g_r$[n] + a;      /* Danielson-Lanczos Lemma (Eq. 12) */
        $f_r$[n+N2] = $g_r$[n] – a;

        a = $w_i$*$h_r$[n] + $w_r$*$h_i$[n];  /* imaginary part of $W^n h$ (Eq. 13) */
        $f_i$[n]      = $g_i$[n] + a;      /* Danielson-Lanczos Lemma (Eq. 12) */
        $f_i$[n+N2] = $g_i$[n] – a;
}
```

2.4. Cost of Computation

The Danielson-Lanczos Lemma, as given in Eq. (12), can be used to calculate the cost of the computation. Let $C(N)$ be the cost for evaluating the transform of N points. Combining the transforms of N points in Eq. (12) requires effort proportional to N because of the multiplication of the terms by W^n and the subsequent addition. If c is a constant reflecting the cost of such operations, then we have the following result for $C(N)$.

$$C(N) = 2C(\frac{N}{2}) + cN \tag{14}$$

This yields a recurrence relation which is known to result into an $O(N \log N)$ process. Viewed another way, since there are $\log_2 N$ levels to the recursion, and cost $O(N)$ at each level, the total cost is $O(N \log_2 N)$.

3. COOLEY-TUKEY ALGORITHM

The Danielson-Lanczos Lemma presented a recursive solution to computing the Fourier Transform. The role of the recursion is to subdivide the original input into smaller lists which are eventually combined according to the lemma. The starting point of the computation thus begins with the adjacent pairing of 1-point DFTs. In the preceding discussion, their order was determined by the recursive subdivision. An alternate method is available to determine their order directly, without the need for the recursive algorithm given above. This result is known as the *Cooley–Tukey*, or *decimation-in-time* algorithm.

To describe the method, we define the following notation. Let F^{ee} be the list of even-indexed terms taken from F^e. Similarly, F^{eo} is the list of odd-indexed terms taken from F^e. In general, the string of symbols in the superscript specifies the path traversed in the tree representing the recursive subdivision of the input data (Fig. 2). Note that the height of the tree is $\log_2 N$ and that all leaves denote 1-point DFTs which are actually elements from the input numbers. Thus, for every pattern of e's and o's, numbering $\log_2 N$ in all,

$$F^{eoeeoeo...oee} = f_n \qquad \text{for some } n \tag{14}$$

The problem is now to directly find which value of n corresponds to which pattern of e's and o's in Eq. (14). The solution is surprisingly simple: reverse the pattern of e's and o's, then let $e = 0$ and $o = 1$, and the resulting binary string denotes the value of n. This works because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits of n. Examining Fig. 2, we observe that traversing successive levels of the tree along the e and o branches corresponds to successively scanning the binary value of index n from the least significant to the most significant bit. The strings appearing under the bottom row designates the traversed path.

The procedure is summarized in Table 1 for $N = 8$. There we see the binary indices listed next to the corresponding array elements. The first subdivision of the data into even- and odd-indexed elements amounts to testing the least significant (rightmost) bit. If that bit is 0, an even index is implied; a 1 bit designates an odd index. Subsequent subdivisions apply the same bit tests to successive index bits of higher significance. Observe that in Fig. 2, even-indexed lists move down the left branches of the tree. Therefore, the order in which the leaves appear from left to right indicate the sequence of 1s and 0s seen in the index while scanning in *reverse order*, from least to most significant bits.

Original Index	Original Array	Bit-reversed Index	Reordered Array
0 0 0	f_0	0 0 0	f_0
0 0 1	f_1	1 0 0	f_4
0 1 0	f_2	0 1 0	f_2
0 1 1	f_3	1 1 0	f_6
1 0 0	f_4	0 0 1	f_1
1 0 1	f_5	1 0 1	f_5
1 1 0	f_6	0 1 1	f_3
1 1 1	f_7	1 1 1	f_7

Table 1: Bit-reversal and array reordering for input into FFT algorithm.

The distinction between the Cooley-Tukey algorithm and the Danielson-Lanczos Lemma is subtle. In the latter, a recursive procedure is introduced in which to compute the DFT. This procedure is responsible for decimating the input signal into a sequence which is then combined, during the traversal back up the tree, to yield the transform output. In the Cooley-Tukey algorithm, though, the recursion is unnecessary since a clever bit-reversal trick is introduced to achieve the same disordered input. Furthermore, directly reordering the input in this way simplifies the bookkeeping necessary in recombining terms. Source code for the Cooley-Tukey FFT algorithm, written in C, is provided in the appendix.

3.1. Computational Cost

The computation effort for evaluating the FFT is easily determined from this formulation. First, we observe that there are $\log_2 N$ levels of recursion necessary in computing F_n. At each level, there are $N/2$ butterflies to compute the F_n^e and F_n^o terms (see Fig. 3). Since each butterfly requires one complex multiplication and two complex additions, the total number of multiplications and additions is $(N/2)\log_2 N$ and $N \log_2 N$, respectively. This $O(N \log_2 N)$ process represents a considerable saving in computation over the $O(N^2)$ approach of direct evaluation. For example if $N \geq 512$, the number of multiplications is reduced to a fraction of 1 percent of that required by direct evaluation.

4. COOLEY-SANDE ALGORITHM

In the Cooley-Tukey algorithm, the given data sequence is reordered according to a bit-reversal scheme before it is recombined to yield the transform output. The reordering is a consequence of the Danielson-Lanczos Lemma that calls for a recursive subdivision into a sequence of even- and odd-indexed elements.

The *Cooley-Sande* FFT algorithm, also known as the *decimation-in-frequency* algorithm, calls for recursively splitting the given sequence about its midpoint, $N/2$.

$$\begin{aligned} F_n &= \sum_{k=0}^{N-1} f_k e^{-2\pi inkl/N} \qquad (15) \\ &= \sum_{k=0}^{(N/2)-1} f_k e^{-2\pi inkl/N} + \sum_{k=N/2}^{N-1} f_k e^{-2\pi inkl/N} \\ &= \sum_{k=0}^{(N/2)-1} f_k e^{-2\pi inkl/N} + \sum_{k=0}^{(N/2)-1} f_{k+N/2}\, e^{-2\pi in(k+N/2)/N} \\ &= \sum_{k=0}^{(N/2)-1} \left[f_k + f_{k+N/2}\, e^{-\pi in} \right] e^{-2\pi inkl/N} \end{aligned}$$

Noticing that the $e^{-\pi in}$ factor reduces to $+1$ and -1 for even and odd values of n, respectively, we isolate the even and odd terms by changing n to $2n$ and $2n+1$.

$$\begin{aligned} F_{2n} &= \sum_{k=0}^{(N/2)-1} \left[f_k + f_{k+N/2} \right] e^{-2\pi i(2n)k/N} \qquad 0 \le n < N/2 \qquad (16) \\ &= \sum_{k=0}^{(N/2)-1} \left[f_k + f_{k+N/2} \right] e^{-2\pi inkl(N/2)} \\ F_{2n+1} &= \sum_{k=0}^{(N/2)-1} \left[f_k - f_{k+N/2} \right] e^{-2\pi i(2n+1)k/N} \qquad 0 \le n < N/2 \qquad (17) \\ &= \sum_{k=0}^{(N/2)-1} \left[f_k - f_{k+N/2} \right] e^{-2\pi ik/N}\, e^{-2\pi inkl(N/2)} \end{aligned}$$

Thus, the even- and odd-indexed values of F are given by the DFTs of f_k^e and f_k^o where

$$f_k^e = f_k + f_{k+N/2} \qquad (18)$$

$$f_k^o = \left[f_k - f_{k+N/2} \right] W^k \qquad (19)$$

The same procedure can now be applied to f_k^e and f_k^o. This sequence is depicted in Fig. 4. The top row represents input list f containing 8 elements. Again, note that lists are delimited by bold lines. Regarding the butterfly notation, the lower left branches denote Eq. (18) and the lower right branches denote Eq. (19).

Since all the even-indexed values of F need f_k^e, a new list is created for that purpose. This is shown as the left list of the second row. Similarly, the f_k^o list is generated, appearing as the second list on that row. Of course, the list sizes diminish by a factor of two with each level since generating them makes use of f_k and $f_{k+N/2}$ to yield one element in the new list. This process of computing Eqs. (18) and (19) to generate new lists terminates when $N=1$, leaving us F, the transform output, in the last row.

In contrast to the decimation-in-time FFT algorithm, in which the input is disordered but the output is ordered, the opposite is true of the decimation-in-frequency FFT algorithm. However, reordering can be easily accomplished by reversing the binary representation of the location index at the end of computation. The advantage of this algorithm is that the values of f are entered in the input array sequentially.

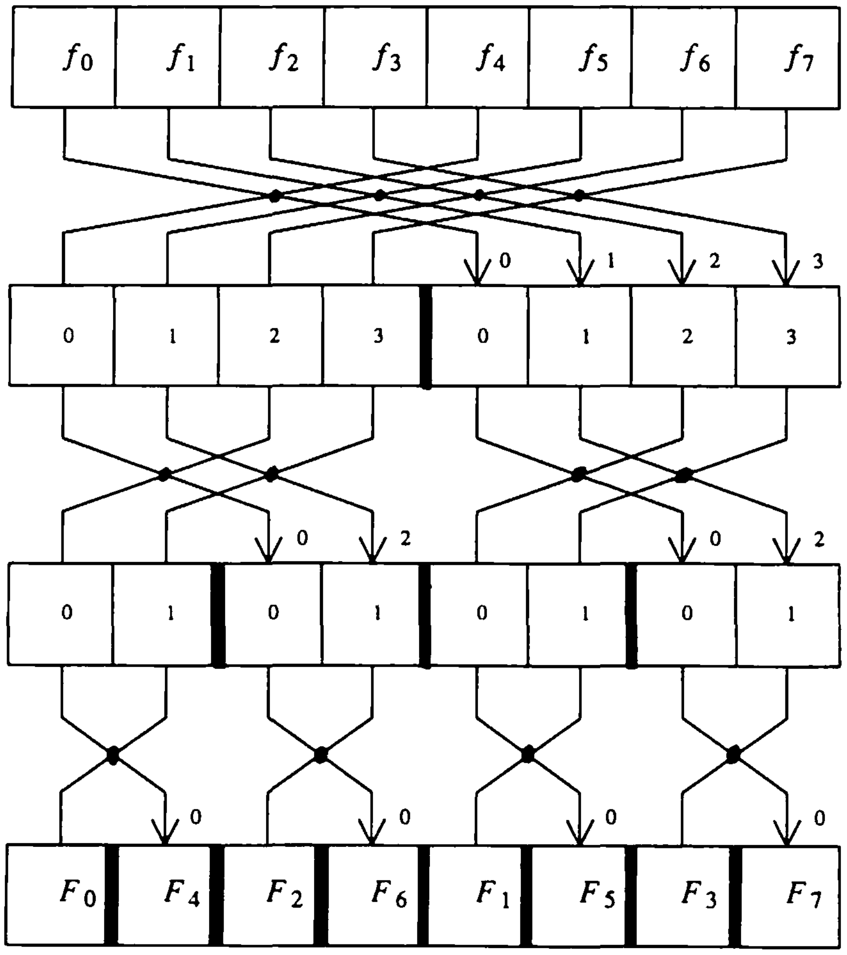
f_0 f_1 f_2 f_3 f_4 f_5 f_6 f_7
0 1 2 3
0 1 2 3 0 1 2 3
0 2 0 2
0 1 0 1 0 1 0 1
0 0 0 0
F_0 F_4 F_2 F_6 F_1 F_5 F_3 F_7

Figure 4: Decimation-in-frequency FFT algorithm.

5. APPENDIX

This appendix provides source code for the recursive FFT procedure given in section 2, as well as code for the Cooley-Tukey algorithm described in section 3. The programs are written in C and make use of IMPROC library routines [Wolberg 88]. The following brief remarks should clarify some of the IMPROC library functions.

The data is passed to the functions in *quads*. A quad is an image control block, containing information about the image. Such data includes the image dimensions (*height* and *width*), pointers to the uninterleaved image channels (*buf* [0] ... *buf* [15]), and other necessary information. Since the complex numbers have real and imaginary components, they occupy 2 channels in the input and output quads (channels 0 and 1). A brief description of the library routines included in the listing is given below.

1) *cpqd* ($q1,q2$) simply copies quad $q1$ into $q2$.

2) *cpqdinfo* ($q1,q2$) copies the header information of $q1$ into $q2$.

3) *NEWQD* allocates a quad header. The image memory is allocated later when the dimensions are known.

4) *getqd* ($h,w,type$) returns a quad containing sufficient memory for an image with dimensions $h \times w$ and channel datatypes *type*. Note that *FFT_TYPE* is defined as 2 channels of type *float*.

5) *freeqd* (q) frees quad q, leaving it available for any subsequent *getqd* call.

6) *divconst* ($q1,num,q2$) divides the data in $q1$ by *num* and puts the result in $q2$. Note that *num* is an array of numbers used to divide the corresponding channels in $q1$.

7) Finally, PI2 is defined to be 2π, or 6.28318531.

5.1. Recursive FFT Algorithm

```
fft1D(q1,dir,q2)        /* Fast Fourier Transform (1D)      */
int dir;                /* dir=0: forward;  dir=1: inverse  */
qdP q1, q2;
{
        int i, N, N2;
        float *r1, *i1, *r2, *i2, *ra, *ia, *rb, *ib;
        double FCTR, fctr, a, b, c, s, num[2];
        qdP qa, qb;

        cpqdinfo(q1, q2);
        N  = q1->width;
        r1 = (float *) q1->buf[0];
        i1 = (float *) q1->buf[1];
        r2 = (float *) q2->buf[0];
        i2 = (float *) q2->buf[1];

        if(N == 2) {                    /* F(0)=f(0)+f(1); F(1)=f(0)-f(1) */
                a = r1[0] + r1[1];      /* a,b needed when r1=r2 */
                b = i1[0] + i1[1];
                r2[1] = r1[0] - r1[1];
                i2[1] = i1[0] - i1[1];
                r2[0] = a;
                i2[0] = b;
        } else {
                N2 = N / 2;
                qa = getqd(1, N2, FFT_TYPE);
                qb = getqd(1, N2, FFT_TYPE);
                ra = (float *) qa->buf[0];     ia = (float *) qa->buf[1];
                rb = (float *) qb->buf[0];     ib = (float *) qb->buf[1];

                /* split list into 2 halves: even and odd */
                for(i=0; i<N2; i++) {
                        ra[i] = *r1++;          ia[i] = *i1++;
                        rb[i] = *r1++;          ib[i] = *i1++;
                }

                /* compute fft on both lists */
                fft1D(qa, dir, qa);
                fft1D(qb, dir, qb);

                /* build up coefficients */
                if(!dir)          /* forward */
                        FCTR = -PI2 / N;
                else    FCTR =  PI2 / N;
                for(fctr=i=0; i<N2; i++,fctr+=FCTR) {
                        c = cos(fctr);
                        s = sin(fctr);
                        a = c*rb[i] - s*ib[i];
                        r2[i]     = ra[i] + a;
                        r2[i+N2] = ra[i] - a;

                        a = s*rb[i] + c*ib[i];
                        i2[i]     = ia[i] + a;
                        i2[i+N2] = ia[i] - a;
                }
                freeqd(qa);
                freeqd(qb);
        }
        if(dir) {        /* inverse: divide by log N */
                num[0] = num[1] = 2;
                divconst(q2, num, q2);
        }
}
```

5.2. Cooley-Tukey FFT Algorithm

```
fft1D(q1, dir, q2)          /* Fast Fourier Transform (1D)     */
int dir;                    /* dir=1: forward;  dir= -1: inverse */
qdP q1, q2;                 /* Uses bit reversal to avoid recursion */
{                           /* and trig recurrence for sin and cos    */
        int i, j, logN, N, N1, NN, NN2, itr, offst;
        unsigned int a, b, msb;
        float *r1, *r2, *i1, *i2;
        double wr, wi, wpr, wpi, wtemp, theta, tempr, tempi, num[2];
        qdP qsrc;

        if(q1 == q2) {
                qsrc = NEWQD;
                cpqd(q1, qsrc);
        } else  qsrc = q1;

        cpqdinfo(q1, q2);
        r1 = (float *) qsrc->buf[0];
        i1 = (float *) qsrc->buf[1];
        r2 = (float *) q2->buf[0];
        i2 = (float *) q2->buf[1];

        N  = q1->width;
        N1 = N - 1;
        for(logN=0,i=N/2; i; logN++,i/=2);  /* # of bits sig digits in N */
        msb = LSB << (logN-1);
        for(i=1; i<N1; i++) {           /* swap all nums; ends remain fixed */
                a = i;
                b = 0;
                for(j=0; a && j<logN; j++) {
                        if(a & LSB) b |= (msb>>j);
                        a >>= 1;
                }
                /* swap complex numbers: [i] <--> [b] */
                r2[i] = r1[b];          i2[i] = i1[b];
                r2[b] = r1[i];          i2[b] = i1[i];
        }
        /* copy elements 0 and N1 since they don't swap */
        r2[0]  = r1[0];                 i2[0]  = i1[0];
        r2[N1] = r1[N1];                i2[N1] = i1[N1];

        /* NN denotes the number of points in the transform.
          It grows by a power of 2 with each iteration.
          NN2 denotes NN/2 which is used to trivially generate
          NN points from NN2 complex numbers.
```

```
            Computation of the sines and cosines of multiple
            angles is made through recurrence relations.
            wr is the cosine for the real terms; wi is sine for
            the imaginary terms.
          */
          NN = 1;
          for(itr=0; itr<logN; itr++) {
                  NN2 = NN;
                  NN  <<= 1;    /* NN *= 2 */

                  theta = -PI2 / NN * dir;
                  wtemp = sin(.5*theta);
                  wpr = -2 * wtemp * wtemp;
                  wpi = sin(theta);
                  wr  = 1.;
                  wi  = 0.;

                  for(offst=0; offst<NN2; offst++) {
                          for(i=offst; i<N; i+=NN) {
                                  j = i + NN2;
                                  tempr = wr*r2[j] - wi*i2[j];
                                  tempi = wi*r2[j] + wr*i2[j];
                                  r2[j] = r2[i] - tempr;
                                  r2[i] = r2[i] + tempr;
                                  i2[j] = i2[i] - tempi;
                                  i2[i] = i2[i] + tempi;
                          }
                          /* trigonometric recurrence */
                          wr = (wtemp=wr)*wpr - wi*wpi + wr;
                          wi = wi*wpr + wtemp*wpi + wi;
                  }
          }
          if(dir == -1) {           /* inverse transform: divide by N */
                  num[0] = num[1] = N;
                  divconst(q2, num, q2);
          }
          if(qsrc != q1) freeqd(qsrc);
  }
```

1. REFERENCES AND SUGGESTED READING

[Antoniou 79] Antoniou, Andreas, *Digital Filters*: *Analysis* and *Design*, McGraw-Hill, New York, 1979.

[Bergland 69] Bergland, G.D., "A Guided Tour of the Fast Fourier Transform," *IEEE Spectrum*, vol. 6, pp. 41-52, July 1969.

[Brigham 74] Brigham, E. Oran, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[Cochran 67] Cochran, W.T., Cooley, J.W., *et al.*, "What is the Fast Fourier Transform?," *IEEE Trans. Audio* and *Electroacoustics*, vol. AU-15, no. 2, pp. 45-55, 1967.

[Cooley 65] Cooley, J.W., and Tukey, J.W., "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comp.*, vol. 19, pp. 297-301, April 1965.

[Cooley 67a] Cooley, J.W., Lewis, P.A.W., and Welch P.D., "Historical Notes on the Fast Fourier Transform," *IEEE Trans. Audio* and *Electroacoustics*, vol. AU-15, no. 2, pp. 76-79, 1967.

[Cooley 67b] Cooley, J.W., Lewis, P.A.W., and Welch P.D., "Application of the Fast Fourier Transform to Computation of Fourier Integrals," *IEEE Trans. Audio* and *Electroacoustics*, vol. AU-15, no. 2, pp. 79-84, 1967.

[Cooley 69] Cooley, J.W., Lewis, P.A.W., and Welch P.D., "The Fast Fourier Transform and Its Applications," *IEEE Trans. Educ.*, vol. E-12, no. 1, pp. 27-34, 1969.

[Danielson 42] Danielson, G.C. and Lanczos, C., "Some Improvements In Practical Fourier Analysis and Their Application to X-Ray Scattering from Liquids," *J. Franklin Institute*, vol. 233, pp. 365-380 and 435-452, 1942.

[Pavlidis 82] Pavlidis, Theo, *Algorithms for Graphics and Image Processing*, Springer-Verlag, Berlin, 1982.

[Press 88] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1988.

[Wolberg 88] Wolberg, G., "IMPROC: An Interactive Image Processing Software Package," Columbia University Computer Science Technical Report 330-88, April 1988.