A REVIEW OF THE DISCRETE FOURIER TRANSFORM
Part 1: retarding waves and the powers of two

Gerard M Blair

ABSTRACT

Since fast algorithms for the discrete Fourier transform (DFT) were first introduced thirty years ago, they have had a major impact on signal processing and are now a basic part of every electrical engineer's education. However, some of the options, and particularly the recent advances, are not as widely known as they deserve. Over two articles, we review the fast algorithms for the DFT. In the first article, we consider why the DFT works and look at the various fast algorithms for transforms whose orders are a power of two. In the second article, we will look at techniques for other orders, at adapting algorithms for purely real data, and at the problems of fixed point noise.

I. THE DISCRETE FOURIER TRANSFORM

To begin with, it is worth looking briefly at how the discrete Fourier transform (DFT) actually works. For this we need a little mathematics, but only to show that a few tricks and a few assumptions will transform a sequence of data samples into an equivalent description of underlying frequencies. In brief, the following section says: if you assume that the data samples can be thought of as the sum of periodic waves, then you can find the strengths for each of these waves by a little multiplication and addition, and the justification for this comes from simple trigonometric identities and the idea that certain sums tend to zero over many terms.

Rather than the traditional derivation from the continuous Fourier transform, we start with a real-valued signal and a few basic assumptions.

The first assumption is that any periodic signal $s(t)$ can be represented as the sum of sinusoidal waves:

$$\begin{aligned} s(t) &= \sum_x A_x \sin(2\pi f_x t + \theta_x) \qquad (1) \\ &= \sum_x \Big[B_x \sin(2\pi f_x t) + C_x \cos(2\pi f_x t) \Big] \end{aligned}$$

where t is time, A, B, and C are amplitudes, f is frequency, θ is phase, and $B_x = A_x \cos(\theta_x)$ and $C_x = A_x \sin(\theta_x)$. The second line of this equation eliminates the phase term by splitting the sinusoid into its odd and even components (with respect to some arbitrary time zero). This is convenient since we will be extracting these two components independently.

The second assumption is that for small periods of time, any real world signal can be thought of as periodic and so as a sum of sinusoidal waves. This assumption is valid in that it is useful.

For the Discrete Fourier Transform (DFT), we are not interested in the continuous waveform but rather in samples taken at regular intervals. Thus our signal becomes:

$$s(n) = \sum_x \left[B_x \sin(2\pi \frac{f_x}{f_s} n) + C_x \cos(2\pi \frac{f_x}{f_s} n) \right] \tag{2}$$

where f_s is the frequency of the samples and n is an integer. With a nod to Shannon, we also assume that f_s is more than twice any of the $\{f_x\}$; mathematicians may like to consider what happens if this assumption is invalid and engineers may prefer to design a filter.

So much for the signal, now a little mathematics. We are interested in sinusoids and in the summation of successive samples of a sinusoid. Consider the following:

$$\frac{1}{N} \sum_{n=0}^{N-1} [\ B \sin(2\pi xn) + C \cos(2\pi xn)\ ] \tag{3}$$

If x is an *integer*, then this summation equals C; otherwise it tends to zero as N becomes large. It is this property which we will use to extract the amplitudes of the odd and even components of specific frequencies.

Suppose we are interested in a particular frequency component f_k within a set of signal samples $\{s(n)\}$ and so want to find the amplitudes of its odd and even components. The trick is to transform the samples by multiplying them by samples of the frequency being sought and then to sum these products. In fact, this performs a correlation between the frequency component and the signal samples but think of it as multiply and add. For instance, we can multiply and add equation 2 by samples of a sine wave of frequency f_k:

$$\frac{1}{N} \sum_{n=0}^{N-1} \left[\sum_x \left(B_x \sin(2\pi \frac{f_x}{f_s} n) + C_x \cos(2\pi \frac{f_x}{f_s} n) \right) \right] \sin(2\pi \frac{f_k}{f_s} n) \tag{4}$$

With a little trigonometry, this becomes:

$$\begin{aligned} \frac{1}{2N} \sum_{n=0}^{N-1} \sum_x [B_x & \left(\cos(2\pi \frac{f_x - f_k}{f_s} n) - \cos(2\pi \frac{f_x + f_k}{f_s} n) \right) + \\ & C_x \left(\sin(2\pi \frac{f_x + f_k}{f_s} n) - \sin(2\pi \frac{f_x - f_k}{f_s} n) \right)] \end{aligned} \tag{5}$$

Recalling what we decided for equation 3, we see that for large N, this equation tends to zero *unless* either of $(f_x \pm f_k)/f_s$ is an integer. With the assumption that $f_x, f_k < f_s/2$ (the nod to Shannon), this will only be true when $f_x = f_k$ in which case equation 5 reduces to $B_k/2$ (i.e. half the amplitude of the odd component of f_k). If we had multiplied instead by a cosine (rather than a sine) we would have extracted $C_k/2$: half the amplitude of the even term.

Thus we have the basis for the DFT. By multiplication and summation we can *sample a frequency* by extracting the amplitudes for its odd and even components. If we consider the significant term $(f_x - f_k)/f_s$ in equation 5, we see that the actual effect of the multiplication by $\sin(2\pi(f_k/f_s)n)$ is to "retard" all the frequencies of all the components of the waveform by the amount f_k; the f_k component becomes zero and so is stationary; and by summing over a large number of samples all the other components are removed, since they are oscillating and their averages tend to zero, and only the desired amplitude remains.

By using complex numbers, we can combine the sine and cosine transforms with the multiplier: $(\cos(2\pi(f_k/f_s)n) - j \sin(2\pi(f_k/f_s)n))$; the amplitudes of the even and odd components of f_k are then transformed into the real and imaginary parts of the sum. In practice, this multiplier is written more compactly as a complex exponential ($e^{-2j\pi(f_k/f_s)n}$) and the multiplication can then be interpreted as a rotation of the complex plane. This use of complex numbers actually helps in the design of fast algorithms and also allows these transforms to be used on data which is itself complex possibly arising, for instance, as the output of other transforms.

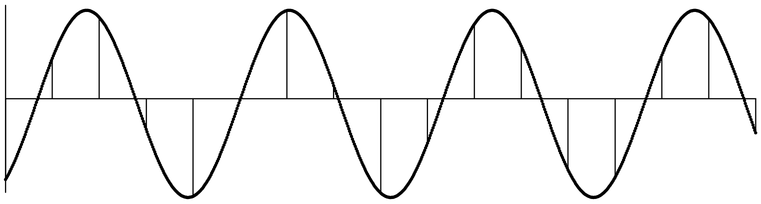


Figure 1: Samples of a periodic wave

In the conventional form of the DFT, we actually seek the frequency samples for integer values of k where $k/N = f_k/f_s$ and N is the number of sample points and so the complex exponential is written as $e^{-2j\pi kn/N}$. Further the summation takes place over a finite number of samples; thus, the effects of frequencies other than the sampled frequency are not totally removed. This can be seen in figure 1 where the retarded sine wave is samples over 16 points. If the sampling were to continue then the average value ($1/N \sum_{n=0}^{N-1} s(n)$) would tend to zero. However, for only these 16 points, the average is clearly not exactly zero and so this component will falsely add to the value of the frequency being extracted (the one retarded to a stationary value). These effects can be minimized by the uses of *windows* which involves the multiplication of the signal samples (or convolution of the frequency samples) by other functions. The reader is advised to read the classic paper by Harris[1] on this topic; for further reading on an intuitive development for the DFT, we recommend a recent article by Kraniauskas[2].

From now on, we will formally define the DFT as:

$$S(k) = \sum_{n=0}^{N-1} W_N^{nk}\ s(n) \quad k = 0 \rightarrow N-1 \quad \text{where} \;\; W_X^x = e^{-j2\pi x/X} \tag{6}$$

(with the 1/N term omitted for convenience).

II. DECOMPOSING THE DFT

Basic Formulation

As with most algorithmic simplifications, the fast algorithms for the DFT are derived by a process of breaking large problems into small ones. We will show this process by manipulating the basic DFT equation and each stage will be illustrated by an accompanying diagram; the reader is encouraged to trust the algebra and to concentrate on making the link between the final equations and the corresponding diagrams.

Let N be non-prime and the product of two factors A and C. We will use index pairs a and b ($0 \rightarrow A-1$) and c and d ($0 \rightarrow C-1$) and replace the original indices n and k ($0 \rightarrow N-1$) by the expressions dA+b and aC+c. Substitution in equation 6 gives:

$$\begin{aligned} S(dA+b) &= \sum_{a=0}^{A-1} \sum_{c=0}^{C-1} W_N^{((dA+b)(aC+c))} s(aC+c) \\ &= \sum_{a=0}^{A-1} \sum_{c=0}^{C-1} W_N^{(adAC)} W_N^{(cdA)} W_N^{(abC)} W_N^{(bc)} s(aC+c) \\ &= \sum_{c=0}^{C-1} W_C^{(cd)} \left[W_N^{(bc)} \sum_{a=0}^{A-1} W_A^{(ab)} s(aC+c) \right] \end{aligned} \tag{7}$$

where $b = 0 \rightarrow A - 1$ and $d = 0 \rightarrow C - 1$

Thus we see that the N-pt DFT can be calculated as C, A-pt DFTs (corresponding to the distinct values of c), the multiplication of their outputs by $W_N^{(bc)}$ (affectionately known as *twiddling*) and their combination through A, C-pt DFTs.

This equation is quite general and may be used to decompose any non-prime value of N; figure 2 illustrates this for a 15-pt DFT. In fact, it explains practically everything.

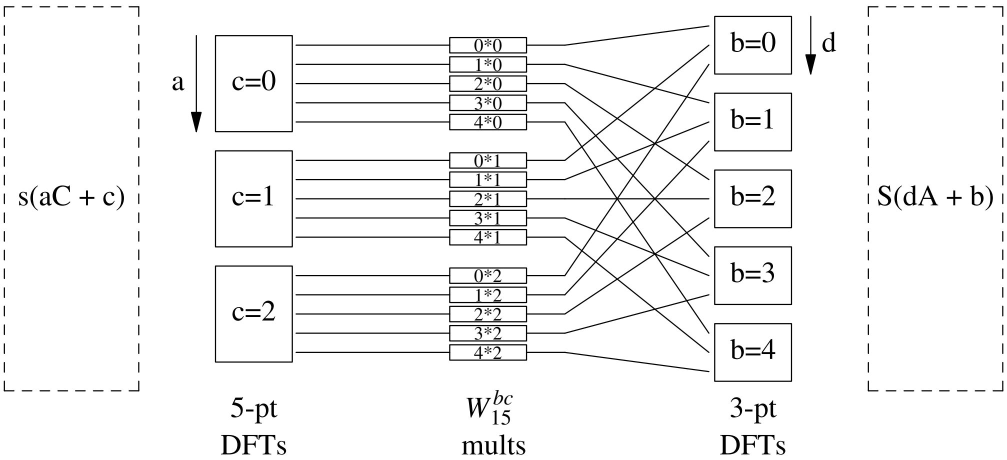


Figure 2: Decomposition of 15-pt DFT

III. RADIX-2

Decimation in Time

Let us consider an N-pt DFT where N is even with the two factors A = N/2 and C = 2. Equation 7 becomes:

$$S(dN/2 + b) = \sum_{c=0}^{1} W_2^{(cd)} \left[W_N^{(bc)} \sum_{a=0}^{N/2-1} W_{N/2}^{(ab)} s(a2 + c) \right] \quad \text{for } b = 0 \rightarrow N/2 - 1,\ d = 0 \rightarrow 1 \quad (8)$$

Using the fact that $W_2^1 = -1$, we see that the first half of the transformed values ($d = 0$) can be expressed as:

$$S(b) = \sum_{a=0}^{N/2-1} W_{N/2}^{(ab)} s(a2) + W_N^{(b)} \sum_{a=0}^{N/2-1} W_{N/2}^{(ab)} s(a2 + 1) \quad \text{for } b = 0 \rightarrow N/2 - 1 \quad (9a)$$

and the second half ($d = 1$) as:

$$S(N/2 + b) = \sum_{a=0}^{N/2-1} W_{N/2}^{(ab)} s(a2) - W_N^{(b)} \sum_{a=0}^{N/2-1} W_{N/2}^{(ab)} s(a2 + 1) \quad \text{for } b = 0 \rightarrow N/2 - 1 \quad (9b)$$

This is illustrated in figure 3 for N=16. The time samples are *decimated* into sets according to whether their indices are odd or even. On each of these sets, an independent N/2-pt DFT is performed. The outputs of the odd-indexed data transform are multiplied by W_N^b (twiddled) where b is the index of the outputs. The results are finally combined such that the first half of the transformed values is the sum of the corresponding outputs from the two DFTs, and the second half is their difference. The final combination is normally referred to as a *butterfly* and depicted as in figure 4 but, of course, this is simply a 2-pt DFT.

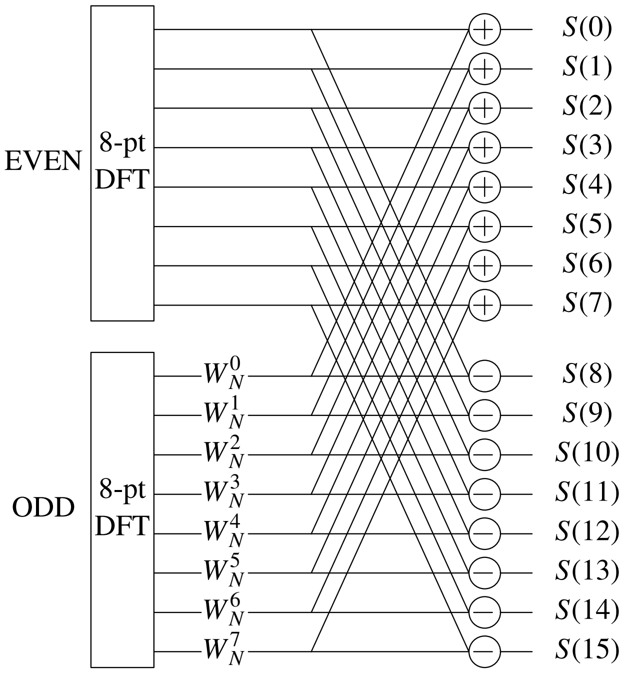


Figure 3: Decimation in time

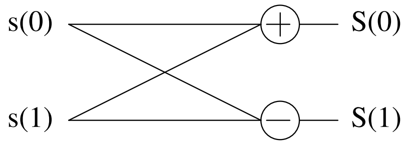


Figure 4: Simple 2-pt DFT – butterfly

Equations (9a) and (9b) were separated to illustrate this structure but they can be combined using the equation:

$$W_N^{N/2+b} = W_N^{N/2} W_N^b = -W_N^b \tag{10}$$

giving the radix-2 decimation-in-time formula as:

$$S(k) = \sum_{a=0}^{N/2-1} W_{N/2}^{(ak)} s(a2) + W_N^{(k)} \sum_{a=0}^{N/2-1} W_{N/2}^{(ak)} s(a2+1) \qquad \text{for } \ k = 0 \rightarrow N-1 \tag{11}$$

Decimation in Frequency

Of course in decomposing the N-pt DFT we could equally well have taken A = 2 and C = N/2 which gives:

$$S(d2+b) = \sum_{c=0}^{N/2-1} W_{N/2}^{(cd)} \left[W_N^{(bc)} \sum_{a=0}^{1} W_2^{(ab)} s(aN/2+c) \right] \ \text{for } \ b = 0 \rightarrow 1,\, d = 0 \rightarrow N/2-1 \tag{12}$$

This approach is known as *decimation in frequency* and can best be interpreted by taking the odd and even values of the transformed values separately. With b = 0 (the even frequency samples), the equation (12) becomes:

$$S(d2) = \sum_{c=0}^{N/2-1} W_{N/2}^{(cd)} [s(c) + s(N/2+c)] \qquad \text{for } \ d = 0 \rightarrow N/2-1 \tag{13a}$$

and for b = 1:

$$S(d2+1) = \sum_{c=0}^{N/2-1} W_{N/2}^{(cd)} W_N^{(c)} [s(c) - s(N/2+c)] \qquad \text{for } d = 0 \rightarrow N/2-1 \tag{13b}$$

This is illustrated in figure 5, again for N=16. The portions in the square brackets are simply the C butterflies (2-pt DFTs) combining the top and bottom halves of the input data to form two N/2-pt sets of data; the lower set (destined to form the odd frequency values) are then twiddled by $W_N^{(c)}$. The two data sets now form the inputs to two independent N/2-pt DFTs (indexed by the same c as the twiddle factors) which produce the even and odd (decimated) frequency samples.

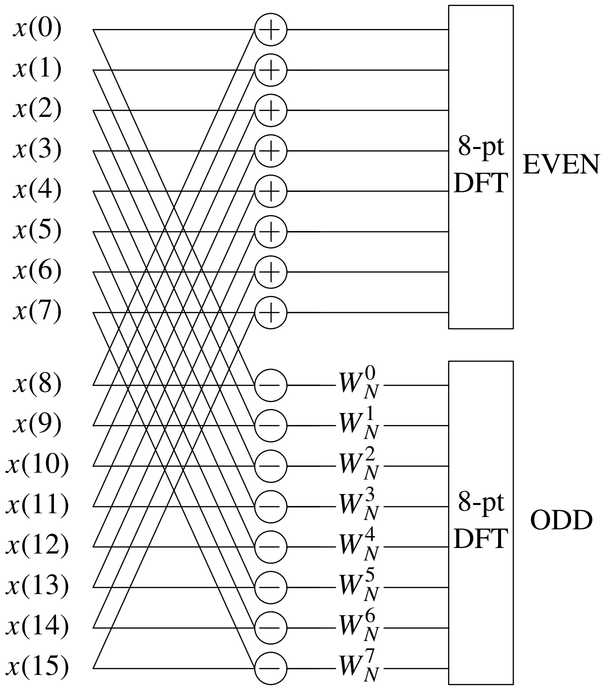


Figure 5: Decimation in frequency

Encoding

With both of these approaches, the true saving comes from applying the same principle repeatedly to each of the smaller DFT blocks; if N is simply a power of 2, then the smallest block is itself a final 2-pt DFT (or butterfly).

This simple, recursive approach is illustrated in figure 6: this is C-like code for the decimation-in-frequency algorithm based directly on figure 5. It assumes that the complex data is held in two arrays of length N which can be indexed by an array pointer: *arrayPtr* and computes the DFT of order N = 2^n for any integer n > 0.

However, there is no reason why the smaller blocks must be done in the same manner. The smaller DFT blocks are totally independent and could be done with different algorithms, they might even be odd in size. For instance, a 20-pt DFT can be achieved with two layers of radix-2 and a final layer of 5-pt DFTs; figure 7 shows an alternative segment of the code.

```
/* Recursive definition of Decimation in Frequency */
decInFre(arrayPtr, blockSize)
int arrayPtr, blockSize;
{
	int i;
	int halfB = blockSize/2;

	/* end condition */
	if (blockSize == 2)
		butterfly(arrayPtr, arrayPtr + 1);
	/* if larger than 2 .... */
	else {
		for (i=0; i<halfB; i++) {
			/* .... perform butterfly operations .... */
			butterfly(arrayPtr + i, arrayPtr + i + halfB);
			/* .... twiddle the lower outputs .... */
			multByCoef(arrayPtr + i + halfB, W^i_blockSize);
			}
		/* .... and call two DFTs of half size */
		decInFre(arrayPtr, halfB);
		decInFre(arrayPtr + halfB, halfB);
		}
}
```

Figure 6: Recursive coding of decimation-in-frequency algorithm

```
	/* end condition */
	if (blockSize == 5)
		five-pt-DFT(/* block of five numbers from */ arrayPtr);
```

Figure 7: Adaptation for single odd factor

Intuitive explanation

While it is possible to apply these formulae without further effort, let us consider how they may be interpreted according to our discussion of the DFT in part I. Consider the decimation in frequency algorithm from equations (13a) and (13b). There are two principles at work here.

The first is the same as for the DFT in general: by applying the transform $\sum_n x_n W_N^{nk}$ to the sampled data set $\{x_n\}$, we simply retard the constituent sinusoidal waves by f_k ($= f_s * k/N$), and the magnitude of the sinusoid whose effective frequency is (the integer) zero is found by adding the sample points.

The second is that if you add two sinusoids of the same frequency, they double in amplitude if they are in phase and sum to zero if they are directly out of phase.

Consider the two waves in figure 8 whose frequencies are 3/16 and 6/16 of the sampling frequency as represented by the notches in the x-axis. Our intention is to extract the amplitudes of these two waves using the 16 sample points.

The algorithm is simply one of progressive consolidation of the data. The first stage is to take all the samples from the right half of the graph and to add them (one for one) to the samples in the left half (as in equation 13a). By looking at the graph, we can see that one wave (whose frequency is an even multiple of $f_s/16$) will reinforce itself, and that the other wave (the "odd"

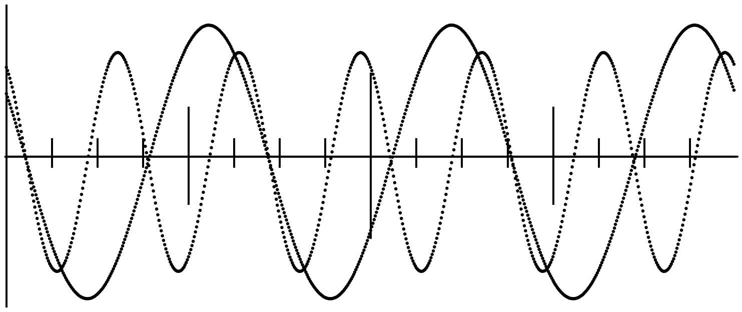


Figure 8: Sampling of two sinusoids

one) will cancel itself. Thus we consolidate the data for the even sample frequencies and remove the data associated with odd sample frequencies.

To cater for the odd waves, we need to transform the data so that the odd waves can also reinforce themselves. This we do by retarding them by $f_s/16$ (i.e. multiplying by W_{16}^{n}) which turns all the odd sample frequencies into even sample frequencies and allows us to consolidate these in the same way. This is equivalent to equation 13b since the inner term can be rewritten using $W_N^{N/2} = -1$:

$$W_N^{(c)}(s(c) - s(N/2 + c)) = (W_N^{(c)} s(c)) + (W_N^{(N/2+c)} s(N/2 + c)) \tag{14}$$

Thus the decimation-in-frequency algorithm consolidates the even-frequency data (to half the number of sample point), and retards and consolidates the odd-frequency data (again to half the number of points). In the radix-2 decimation-in-frequency algorithm, this is repeated on the two halves recursively and terminates when there are N single "consolidated" values which correspond to the frequency samples.

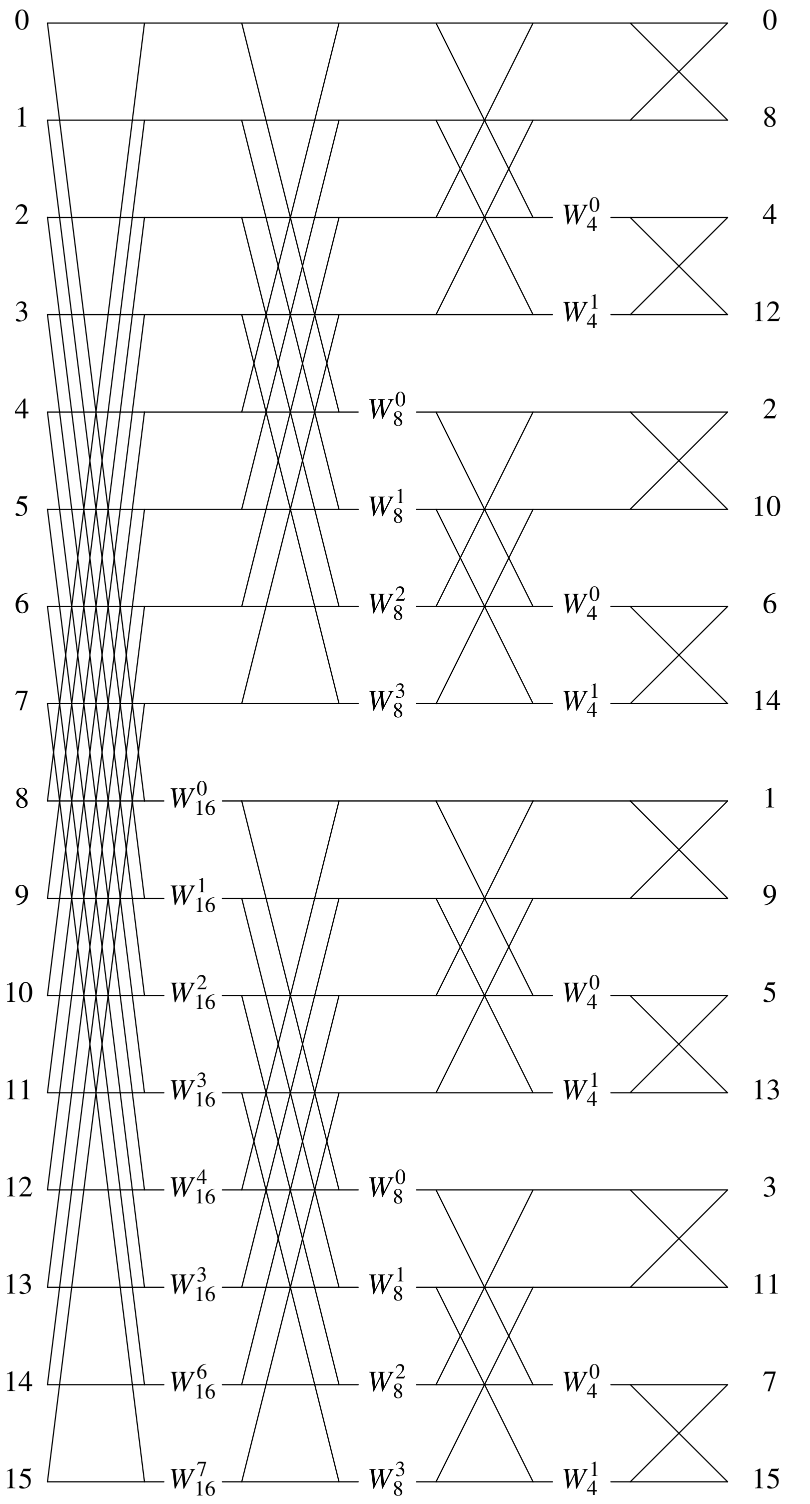
IV. DATA FLOW DIAGRAM

Let us do one example to illustrate the basic similarity of these approaches using a data flow diagram as shown in figure 9 for N=16. This is a common representation but let us dwell on it for a while.

Consider the indexing. With both decimation in frequency and decimation in time, the first half and second half are transformed onto the even and odd halves or visca versa. If we consider the indices as binary numbers, this is equivalent to transforming the most significant bit to become the least significant bit. If this is repeated for each of the smaller DFTs, the complete algorithm leads to a *bit reversal* whereby the data index $\{a_n \ldots . a_1 a_0\}$ is transformed onto $\{a_0 a_1 \ldots . a_n\}$ (e.g. the 4-bit binary number $13_{10} = 1101_2$ becomes $11_{10} = 1011_2$ This is a common feature of radix based algorithms.

Figure 9 may be interpreted as a decimation-in-frequency diagram with the time domain on the left and the (decimated) frequency domain on the right. As such the data flows from left to right. Comparison with figure 5 should confirm this.

However it is equally valid to interpret this as a decimation in time diagram with data flowing from the (decimated) time domain on the right to the frequency domain on the left. Comparison with figure 3 should confirm this. Thus the two algorithms lead to the same diagram with the data flowing in opposite directions but with the same twiddle factors.



Left to right: Decimation in Frequency
Right to left: Decimation in Time

Figure 9: Radix-2 – order-16 flow diagram

Let us continue. If figure 9 is thought of as four columns of butterflies separated by three columns of multipliers, we can construct a 3*16 twiddle-factor table as shown in table 1 giving the multiplications for the corresponding locations in the diagram.

–	–	–
–	–	–
–	–	W_4^0
–	–	W_4^1
–	W_8^0	–
–	W_8^1	–
–	W_8^2	W_4^0
–	W_8^3	W_4^1
W_{16}^0	–	–
W_{16}^1	–	–
W_{16}^2	–	W_4^0
W_{16}^3	–	W_4^1
W_{16}^4	W_8^0	–
W_{16}^5	W_8^1	–
W_{16}^6	W_8^2	W_4^0
W_{16}^7	W_8^3	W_4^1

Table 1: Twiddle-factors for radix-2 algorithms

If the flow diagram itself is re-ordered so that the left side is decimated and the right side is in ascending order, the twiddle factors will be re-ordered and the flow lines will appear reflected in the vertical axis. If we then perform such a reflection to map it back onto the original diagram, the twiddle-factor table becomes table 2. Of course now the "decimation-in-time" algorithm flows from left to right and the "decimation-in-frequency" algorithm from right to left. In effect this shows that the same flow diagram can take ordered (or bit reversed) data and transform it to bit reversed (or ordered) data through the same network with different sets of twiddle factors. And as we will see below, there are further (and simpler) sets of twiddle factors to achieve the same transform.

V. RADIX-4

Equation 7 can also be used to derive the radix-4 algorithms (for N divisible by 4) by taking the two factors A = N/4 and C = 4. Equation 7 thus becomes:

$$S(dN/4+b) = \sum_{c=0}^{3} W_4^{(cd)} \left[W_N^{(bc)} \sum_{a=0}^{N/4-1} W_{N/4}^{(ab)} s(a4+c) \right] \text{ for } b = 0 \rightarrow N/4-1,\ d = 0 \rightarrow 3 \quad (15)$$

The data is organized into four groups (indexed by c) of N/4 values formed by taking every 4th value of the original indices. On each group, an N/4-pt DFT is performed and the outputs (indexed by b) are twiddled by W_N^{bc}. These products are then combined by the N/4, (outer) 4-pt DFTs. The procedure can, of course, be applied recursively to implement the smaller N/4-pt DFTs which leads to the decimation-in-time radix-4 algorithm. The decimation-in-frequency version can be found by taking A = 4 and C = N/4.

A 4-pt DFT is quite simple to implement since the W_4^{nk} terms leads to trivial multiplications by ± 1 and $\pm j$. If we view the 4-pt DFT as a matrix multiplication **S** = **As**, then:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \quad (16)$$

–	–	–
–	–	W_{16}^0
–	W_8^0	–
–	W_8^0	W_{16}^4
W_4^0	–	–
W_4^0	–	W_{16}^2
W_4^0	W_8^2	–
W_4^0	W_8^2	W_{16}^6
–	–	–
–	–	W_{16}^1
–	W_8^1	–
–	W_8^1	W_{16}^5
W_4^1	–	–
W_4^1	–	W_{16}^3
W_4^1	W_8^3	–
W_4^1	W_8^3	W_{16}^7

Table 2: Twiddle-factors for re-ordered radix-2 algorithms

However, using the radix-2 algorithms the 4-pt DFT can be implemented in terms of butterflies with only one multiplication by $-j$ (as shown in figure 10) which, while simple in itself, is even simpler if the multiplication is performed by merely re-ordering the real and imaginary inputs to the bottom-left complex subtracter.

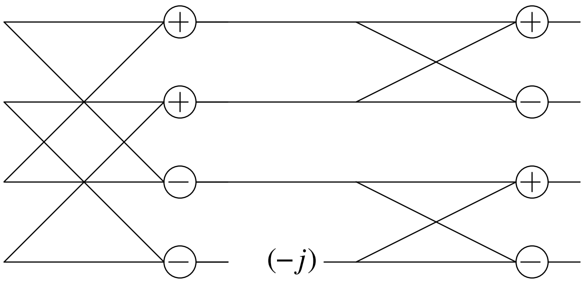


Figure 10: Simple 4-pt DFT

Comparing figure 10 with the flow diagram in figure 9, we see that the larger flow diagram may have many 4-pt DFT flow diagrams embedded within it. Applying the radix-4 algorithm to a 16-pt DFT, we obtain the twiddle-factor table shown in table 3.

By comparing this with the radix-2 twiddle factors we see that radix-4 algorithms require fewer non-trivial multiplications than radix-2. For this reason we might consider a radix-8 algorithm. However, the savings are less than might be hoped since there are non-trivial complex multiplications in the 8-pt DFT itself (although these are not as costly as other complex multiplications as we will see below). In fact, the best algorithm is the following *split-radix* algorithm.

VI. SPLIT-RADIX

Let us consider the complex decimation-in-frequency split-radix algorithm [3] where-by an N-pt DFT is replaced by one N/2-pt and two N/4-pt DFTs: a sort of compromise between radix-2 and

$-$	$-$	$-$
$-$	$-$	$-$
$-$	$-$	$-$
$-$	$-$	$(-j)$
$-$	W_{16}^{0}	$-$
$-$	W_{16}^{1}	$-$
$-$	W_{16}^{2}	$-$
$-$	W_{16}^{3}	$(-j)$
$-$	W_{16}^{0}	$-$
$-$	W_{16}^{2}	$-$
$-$	W_{16}^{4}	$-$
$-$	W_{16}^{6}	$(-j)$
$(-j)$	W_{16}^{0}	$-$
$(-j)$	W_{16}^{3}	$-$
$(-j)$	W_{16}^{6}	$-$
$(-j)$	W_{16}^{9}	$(-j)$

Table 3: Twiddle-factors for radix-4 algorithm

radix-4. In this formulation, equation 7 is rewritten as:

$$\begin{aligned} S(2d) &= \sum_{c=0}^{N/2-1} (s(c) + s(N/2+c))\, W_{N/2}^{(cd)} \\ S(4d+1) &= \sum_{c=0}^{N/4-1} \Big[(s(c) - s(N/2+c)) - j(s(N/4+c) - s(3N/2+c)) \Big] W_N^{c}\, W_{N/4}^{(cd)} \\ S(4d+3) &= \sum_{c=0}^{N/4-1} \Big[(s(c) - s(N/2+c)) + j(s(N/4+c) - s(3N/2+c)) \Big] W_N^{3c}\, W_{N/4}^{(cd)} \\ &\text{where } \ d = 0 \rightarrow N/4 - 1 \ \text{ and } \ W_X^x = e^{-j2\pi x/X} \end{aligned} \tag{17}$$

This equation can be visualized as in figure 11. Here the $s(n)$ values are stored in an ordered array on the left and transformed by the connection "mesh" in the centre into three data sets on the right. The mesh contains the additions and trivial multiplications by $\pm j$ and is thought to move in N/4 steps down the arrays; its flow diagram is shown in figure 12. As with other radix algorithms, this scheme can be applied recursively to each of the smaller DFTs and the twiddle factors for our order-16 example becomes even more sparse as shown in table 4.

While we have so far been counting non-trivial complex multiplications, the observant reader may have noticed that if we count *all* the complex multiplications in the twiddle-factor tables (including those by $-j$) then the number has always been the same. The key point is that the split-radix algorithm can be implemented with the least number of *real* multiplications of currently known algorithms for N=2^n and it achieves this by maximizing the number of multiplications by $-j$.

For completeness, we note that the decimation-in-time split-radix decomposition also exists and can be written as:

$$S(k) = \sum_{n=0}^{N/2-1} W_{N/2}^{nk} s(2n) + W_N^{k} \sum_{n=0}^{N/4-1} W_{N/4}^{nk} s(4n+1) + W_N^{3k} \sum_{n=0}^{N/4-1} W_{N/4}^{nk} s(4n+3) \tag{18}$$

$$\text{where } \ k = 0 \rightarrow N-1 \ \text{ and } \ W_X^x = e^{-j2\pi x/X}$$

–	–	–
–	–	–
–	–	–
–	–	$(-j)$
–	–	W_8^0
–	–	W_8^1
–	$(-j)$	W_8^0
–	$(-j)$	W_8^3
–	W_{16}^0	–
–	W_{16}^1	–
–	W_{16}^2	–
–	W_{16}^3	$(-j)$
$(-j)$	W_{16}^0	–
$(-j)$	W_{16}^3	–
$(-j)$	W_{16}^6	–
$(-j)$	W_{16}^9	$(-j)$

Table 4: Twiddle-factors for split-radix algorithm

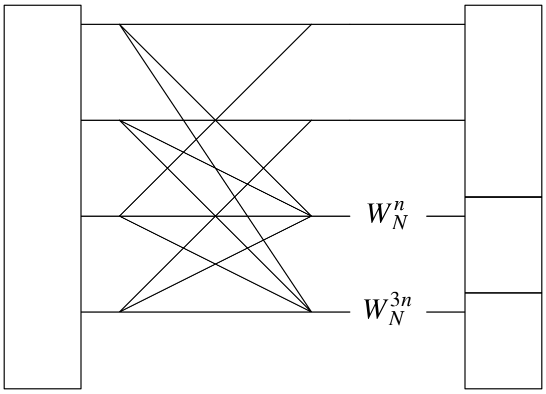


Figure 11: Split-radix transformation

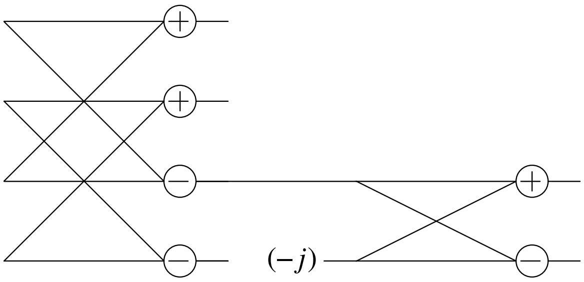


Figure 12: Split-radix 4-pt transform mesh

VII. COMPARISONS AND CONCLUSIONS

In terms of implementation, it should be noted that all of these radix algorithms can be performed "in-place"; that is, at each stage of the algorithm the output data may overwrite the input data and so no storage is required beyond the size of the original data. Furthermore, each algorithm is amenable to simple recursive programming techniques (as illustrated in figure 6) and whether programmed recursively or not, simple compact encoding is practical and easily converted to transforms for different powers of two.

To allow us to compare the efficiency of these algorithms, we therefore need to count the basic arithmetic operations: real additions and real multiplications. To do this we must first look at the implementation of complex multiplications.

Let us consider two complex numbers: (a + jb) and (c + jd) where a, b, c, d are real numbers. Their product is ((ac - bd) + j(bc + ad)) which at first sight involves four real multiplications and two real additions (counting a subtraction as an addition). However, the same product can also be expressed as:

$$[(a+b)c - b(d+c)] \ + \ j\,[(a+b)c + a(d-c)] \tag{19}$$

which involves only three real multiplications since the $(a+b)c$ term is used twice. The total number of additions is five but in the context of the DFT these are counted as three since the term (c + jd) are thought of as the coefficients $(\cos(\theta) + j\sin(\theta))$. Thus the additions (d ± c) would generally be precalculated and so considered as an overhead. A flow diagram of this algorithm is shown in figure 13; readers concerned with fixed point implementations should consider the central path with some care.

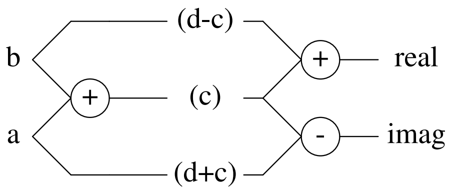


Figure 13: Complex-multiplication flow diagram

It is also worth noting that the coefficients in the DFT have special cases. Obviously multiplications by ±1 and $\pm j$ are easily implemented. In addition a saving can be made for coefficients in the form of $(\pm c \pm jc)$ since the product then reduces to two additions of a and b (with various signs) and two multiplications by c. In the DFT algorithm this applies when θ = odd multiples of $\pi/4$. In this way, an 8-pt DFT can be calculated with only four real multiplications.

Number of non-trivial real multiplications				
N	radix-2	radix-4	radix-8	split-radix
16	24	20		20
32	88			68
64	264	208	204	196
128	712			516
256	1800	1392		1284
512	4360		3204	3076
1024	10248	7856		7172

Table 5: Comparison of multiplications for radix algorithms

On this basis, table 5 gives the number of real multiplications[3] for various orders of DFTs using the algorithms discussed so far. To consider the number of additions, recall that each algorithm uses the same flow graph with different multiplication factors. For a complex transformation of order N (a power of 2), the flow graph contains $\log_2 N$ layers of N complex additions (or subtractions) each requiring 2 real adders (subtractors). Since for each complex multiplication the same number of real additions as real multiplications are required, the number of real additions for each algorithm is equal to the numbers shown in table 5 for multiplications plus $2N \log_2 N$.

Clearly then, the split-radix algorithm is the best of the radix algorithms in terms of the number real operations. It is interesting to note that it took nearly 20 years to emerge after the original fast Fourier transform (FFT) was published by Cooley in 1965. This makes one wonder how many other "established" algorithms are waiting for refinement

References

1. Frederic J Harris, ‘‘On the use of windows for harmonic analysis with the discrete Fourier transform’’, *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51-83, Jan 1978.
2. P Kraniauskas, ‘‘A plain man’s guide to the FFT’’, *IEEE Spectrum*, pp. 24-35, Apr 1994.
3. Pierre Duhamel, ‘‘Implementation of "split-radix" FFT algorithms for complex, real and real-symmetric data’’, *IEEE Trans Acoustics Speech and Signal Processing*, vol. ASSP-34, no. 2, pp. 285-295, Apr 1986.

Dr Gerard M Blair is currently a Visiting Senior Researcher at Hitachi Ltd, Central Research Laboratories, Tokyo, Japan. He is returning as a Senior Lecturer in October to the Department of Electrical Engineering, The University of Edinburgh, The King’s Buildings, Edinburgh EH9 3JL.