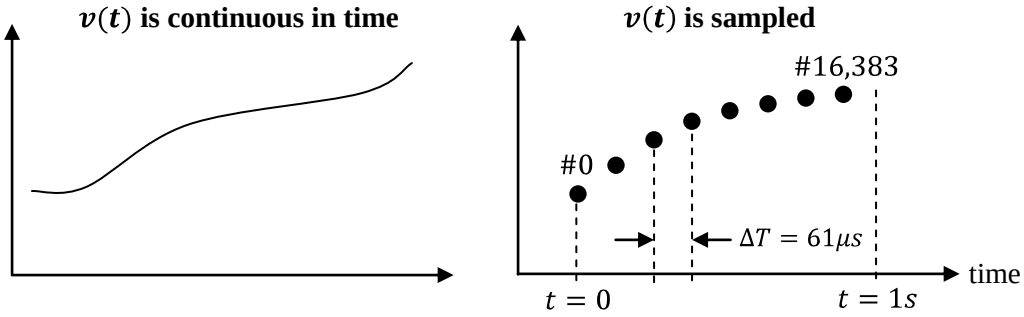
A VisualBasic subroutine which calculates a fast Fourier Transform

The subroutine described here is an implementation of the Danielson-Lanczos Lemma coded in VisualBasic Express 2010.

Statement of the problem

We have a set of $16{,}384 = 2^{14}$ numbers. These happen to be voltages, expressed in Volts, measured over the terminals of a microphone. The measurements were made over a one-second period by a device which recorded the voltage once every 61.035 microseconds. We want to extract from this set of numbers the relative significance of the frequencies of the various sinusoidal waveforms whose sum best represents the data. "Significance" could be stated as the relative power, or perhaps relative magnitude, of each particular frequency compared with the others which are present.



Some preliminary observations

It may well be that the microphone is an analogue device and generates a voltage which is mathematically continuous in time. If we made some assumption about how the continuous waveform extended out to positive and infinite time (perhaps by some repetition), we could expand the waveform into an infinite series of cosine and sine functions in the manner of a Fourier series.

But, our data is not continuous. Nor does it extend out indefinitely in time. These are separate issues.

We are going to deal with the latter shortcoming by assuming the entire set of data is repeated outwards in time (in both directions). By forcing the data to become periodic we place an upper limit on how long the period of the constituent cosine and sine components needs to be. When we repeat the data set to the right, referring to the figure above, the first recurrence of data point #1 is not synchronous with data point #16,383, but rather one sample increment ΔT later. This means the fundamental period has duration $N\Delta T$.

It is an important assumption that the sampling interval ΔT be uniform. When that is the case, the fact that the data is discrete automatically solves the first issue. It places a lower bound on how short the period of the constituent cosine and sine components need to be. There is no point including in the Fourier series any trigonometric components whose period is shorter than ΔT. Our set of data has nothing meaningful to say about things that happen more quickly than ΔT.

Taken together, these two issues restrict the range of periods for the constituent cosine and sine terms. The periods have to lie within the range from ΔT to $N\Delta T$. Furthermore, since the periods of the constituent cosine and sine components must be an integral multiple of the shortest period ΔT, there are in total only N frequency components.

That can be stated a little differently. The spectrum of frequencies which represents our sample of N data points consists of exactly N non-zero frequencies.

A word about notation. We could identify the constituent frequencies as $f_n = 1/n\Delta T$ for $n = 1, 2, 3, \cdots, N$ or as $f_n = 1/(n+1)\Delta T$ for $n = 0, 1, 2, \cdots, N-1$. These frequencies are in Hertz. While that's true, the different frequencies are not normally referred to in Hertz, but instead simply by their index number n. And, it is usual to use the $n = 0, 1, 2, \cdots, N-1$ numbering schedule, which then mirrors the numbering of the data points.

The discrete-time Fourier Transform

Messr. Fourier developed a "transform" for his eponymous series. His transform is an integral which maps a continuous periodic waveform (function) into its discrete frequency components. Later, the transform was extended so that a continuous non-periodic waveform is mapped into a continuous function of frequencies, called the spectrum. Even later, the transform was extended to apply to regularly sampled data, which is what we have here. The transform used for this purpose is called the discrete-time Fourier Transform. It can be written as follows.

$$\mathcal{F}(k) = \sum_{n=0}^{N-1} x_n e^{-j2\pi\frac{n}{N}k} \text{ for } k = 0, 1, 2, \cdots, N-1 \quad (1)$$

A few notes:

1. $\mathcal{F}(k)$ is the strength of the k^{th} frequency component, where $f_k = 1/(k+1)\Delta T$ in Hertz.
2. $\mathcal{F}(k)$ is a complex number. Its Real part corresponds to the magnitude of the cosine component at frequency number k. Its Imaginary part corresponds to the magnitude of the sine component.
3. It is because $\mathcal{F}(k)$ is complex that I described it as the "strength" of frequency k, rather than the magnitude. Magnitude has a special meaning when applied to complex numbers.
4. x_n are the N data points. In our case, the voltage measurements are real numbers. They don't have to be. The transform in Equation (1) applies equally well when the x_n are complex numbers.
5. Equation (1) is a summation over all N data points. It yields one frequency component. To calculate the full spectrum, a separate summation must be done for each of the N frequency components. Each term in each summation requires a multiplication, so the complete analysis is going to require $N \times N = N^2$ multiplications. In our case, analyzing one second's worth of data is going to require $16{,}384^2 = 268{,}000{,}000$ multiplications. That is a huge, and slow, number.

The inverse discrete-time Fourier Transform

Equation (1) takes data from the time-domain and generates a corresponding representation in the frequency-domain. Sometimes, it is necessary to take the frequency-domain representation and "back out" the time-domain waveform that corresponds to it. The former process is called analysis; the latter process is called synthesis. The transform which does the work in reverse is called the inverse discrete-time Fourier Transform. It is:

$$x_n = \sum_{k=0}^{N-1} \mathcal{F}(k) e^{+j2\pi\frac{n}{N}k} \text{ for } n = 0, 1, 2, \cdots, N-1 \quad (2)$$

The fast Fourier Transform

The computational difficulty I mentioned in Note 5 above can be greatly reduced if the number of data points *N* is a power of two. Our sample size $N = 16{,}384$ is such a power of two. It is the 14th power of two.

The basis for all fast Fourier algorithms is the regularity in the exponential term with the imaginary coefficient. That in turn rests on the periodicity of the cosine and sine functions which the exponential term embodies.

Let's look at the summation in Equation (1) for some arbitrary value of k. Since *N* is even (as a power of two, by assumption, *N* must be even), there are an even number of terms in the summation. We will divide them into two groups. Not the top half and the bottom half. Instead, we will group the terms for which index n is even and, separately, the terms for which index n is odd.

$$\mathcal{F}(k) = \sum_{\substack{n=0 \\ n\ even}}^{N-1} x_n e^{-j2\pi\frac{n}{N}k} + \sum_{\substack{n=0 \\ n\ odd}}^{N-1} x_n e^{-j2\pi\frac{n}{N}k} \qquad (3)$$

An even number $n = 0, 2, 4, \cdots, N-2$ can always be expressed as $n = 2m$ for $m = 0, 1, 2, \cdots, ½N - 1$. Similarly, an odd number $n = 1, 3, 5, \cdots, N-1$ can always be expessed as $n = 2m + 1$ for $m = 0, 1, 2, \cdots, ½N - 1$. Re-indexing in this manner changes Equation (3) to:

$$\begin{aligned} \mathcal{F}(k) &= \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi\frac{2m}{N}k} + \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi\frac{2m+1}{N}k} \\ &= \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi\frac{m}{(½N)}k} + \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi\frac{m}{(½N)}k} e^{-j2\pi\frac{1}{N}k} \qquad (4) \end{aligned}$$

Note that the second exponential factor in the second summation does not depend on index m. It is the same for all terms in that summation and can be brought out of the summation as a constant coefficient.

$$\mathcal{F}(k) = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi\frac{m}{(½N)}k} + e^{-j2\pi\frac{1}{N}k} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi\frac{m}{(½N)}k} \qquad (5)$$

Now, look at where *N* appears in the two summations. The upper index depends on $½N$. The denominator of the exponential terms is $½N$. Other than in the constant coefficient of the second summation, *N* never appears by itself, only $½N$. Each summarion by itself looks like a miniature version of the original summarion in Equation (1). The only differences are: (i) that each has only half as many terms as the original summation, and (ii) that only half of the original *N* data points are used in each summation (points picked on an alternating even-odd basis from the original data set).

But, we have not yet reduced the amount of work involved. Although each summation has only half as many multiplications as before, there are still two summations, and still *N* multiplications in all. But, there's some good stuff coming.

Equation (5) is an expansion that applies for each of the *N* frequencies. What we are now going to do is to compare what this expression looks like for the bottom half of the k's ($k = 0, 1, 2, \cdots, ½N$) with what it

looks like for the top half of the k's ($k = ½N, ½N + 1, ½N + 2, \cdots, N$). Since N is even (by assumption), there will always be two equal halves for this purpose. Here's the comparison:

$$\begin{array}{lll} k & \mathcal{F}(k) & = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}k} + e^{-j2\pi \frac{1}{N}k} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}k} \quad (6A) \\ \text{in bottom half} & & \\ k' = ½N + k & \mathcal{F}(k') & = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}k'} + e^{-j2\pi \frac{1}{N}k'} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}k'} \\ \text{in top half} & & = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}(½N+1)} + e^{-j2\pi \frac{1}{N}k'} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}(½N+1)} \\ & & = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}} e^{-j2\pi m} + e^{-j2\pi \frac{1}{N}k'} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}} e^{-j2\pi m} \\ & & = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}} + e^{-j2\pi \frac{1}{N}k'} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}} \quad (6B) \end{array}$$

Taking the step from the second-to-last line to the last line is very important. Each term in each of the summations has a factor $e^{-j2\pi m}$. For all of the indices $m = 0, 1, 2, \cdots, N - 1$, the exponent is an integral multiple of 2π, and therefore $e^{-j2\pi m} = 1$ for all values of m. This is periodicity at play.

Now, observe that the summations in Equation $(6A)$ and $(6B)$ are identical. If we let $E(k)$ and $O(k)$ be the two summations for k's in the bottom half, then we arrive at:

$$\begin{array}{lll} k \text{ in bottom half} & \mathcal{F}(k) & = E(k) + e^{-j2\pi \frac{1}{N}k} O(k) \quad (7A) \\ k' = ½N + k \text{ in top half} & \mathcal{F}(k') & = E(k) + e^{-j2\pi \frac{1}{N}k'} O(k) \quad (7B) \end{array}$$

Now, we have achieved a real savings. We only need to calculate the summations for one half of the k's. If we do the summations for $k = 0, 1, 2, \cdots, ½N - 1$, and calculate their $\mathcal{F}(k)$'s, then the values of $\mathcal{F}(k)$ for the high-order half of the k's can be calculated with very little work. The ability to do this reduces the overall work required by almost a factor of two.

We can repeat this trick. I mentioned above that the $E(k)$ and $O(k)$ summations look like half-size versions of the original transform. If N is a power of two, then $½N$ will also be a power of two. These half-size summations can be tackled anew, as if they were two separate discrete time Fourier Transforms with $½N$ data points apiece. If we can do that, then we will have reduced the scope of the arithmetic by nearly a factor of four.

Indeed, if N is a power of two, then $½N$, $¼N$, and all the successive divisions of N by two will be even. In principle, we can continue to repeat the trick all the way down until we reach rock bottom, with a discrete-time Fourier Transform containing only two data points.

That's what we are going to do. The objective now is to find a way to repeat the trick efficiently.

The $N = 2$ root case

Let's take a step back and look at the "rock bottom" case with only two data points. The complete discrete-time Fourier Transform for the two data points x_0 and x_1 is:

$$\left.\begin{array}{lll} \text{For frequency } k = 0: & \mathcal{F}(0) = \sum_{n=0}^{1} x_n e^{-j2\pi\frac{n}{2}0} & = x_0 e^{-j2\pi 0} + x_1 e^{-j2\pi 0} = x_0 + x_1 \\ \text{For frequency } k = 1: & \mathcal{F}(1) = \sum_{n=0}^{1} x_n e^{-j2\pi\frac{n}{2}1} & = x_0 e^{-j2\pi 0} + x_1 e^{-j2\pi\frac{1}{2}} = x_0 - x_1 \end{array}\right\} \quad (8)$$

The $N = 4$ case - Brute force expansion

I am simply going to use brute force to expand all four frequencies.

$$\begin{array}{lll} \mathcal{F}(0) = x_0 e^{-j\frac{0\pi}{4}0} + x_1 e^{-j\frac{2\pi}{4}0} + x_2 e^{-j\frac{4\pi}{4}0} + x_3 e^{-j\frac{6\pi}{4}0} = x_0 + x_1 + x_2 + x_3 & = (x_0 + x_2) + (x_1 + x_3) \\ \mathcal{F}(1) = x_0 e^{-j\frac{0\pi}{4}1} + x_1 e^{-j\frac{2\pi}{4}1} + x_2 e^{-j\frac{4\pi}{4}1} + x_3 e^{-j\frac{6\pi}{4}1} = x_0 - jx_1 - x_2 + jx_3 & = (x_0 - x_2) - j(x_1 - x_3) \\ \mathcal{F}(2) = x_0 e^{-j\frac{0\pi}{4}2} + x_1 e^{-j\frac{2\pi}{4}2} + x_2 e^{-j\frac{4\pi}{4}2} + x_3 e^{-j\frac{6\pi}{4}2} = x_0 - x_1 + x_2 - x_3 & = (x_0 + x_2) - (x_1 + x_3) \\ \mathcal{F}(3) = x_0 e^{-j\frac{0\pi}{4}3} + x_1 e^{-j\frac{2\pi}{4}3} + x_2 e^{-j\frac{4\pi}{4}3} + x_3 e^{-j\frac{6\pi}{4}3} = x_0 + jx_1 - x_2 - jx_3 & = (x_0 - x_2) + j(x_1 - x_3) \end{array}$$

The $N = 4$ case - Method #2

There is another way we could have handled the case with four data points. We could have done an $N = 2$ root case using the two data points x_0 and x_2 and, completely separately, done an $N = 2$ root case on the "odd" points x_1 and x_3. Let's do it that way now. To keep all the cases separate, I will use a superscript on the $\mathcal{F}$'s listing the specific data points included. For the first pair of points:

$$\left.\begin{array}{ll} \mathcal{F}^{0,2}(0) & = x_0 + x_2 \\ \mathcal{F}^{0,2}(1) & = x_0 - x_2 \end{array}\right\} \quad (9A)$$

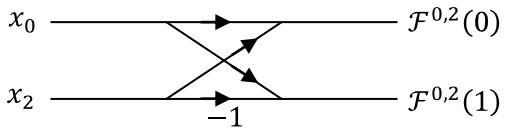
and for the second pair of points:

$$\left.\begin{array}{ll} \mathcal{F}^{1,3}(0) & = x_1 + x_3 \\ \mathcal{F}^{1,3}(1) & = x_1 - x_3 \end{array}\right\} \quad (9B)$$

Observe that the right-hand sides for these two subcases include all four variants which appear in the brute force expansion, which can therefore be written as:

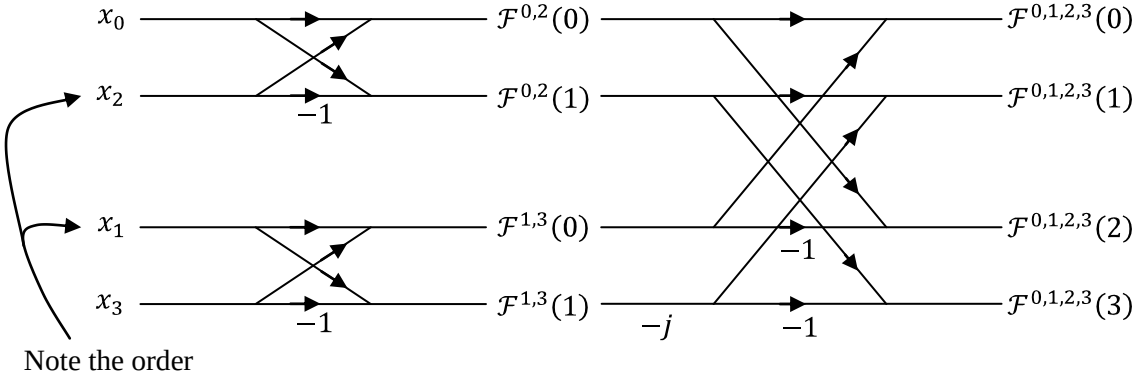
$$\left.\begin{array}{ll} \mathcal{F}^{0,1,2,3}(0) & = \mathcal{F}^{0,2}(0) + \mathcal{F}^{1,3}(0) \\ \mathcal{F}^{0,1,2,3}(1) & = \mathcal{F}^{0,2}(1) - j\mathcal{F}^{1,3}(1) \\ \mathcal{F}^{0,1,2,3}(2) & = \mathcal{F}^{0,2}(0) - \mathcal{F}^{1,3}(0) \\ \mathcal{F}^{0,1,2,3}(3) & = \mathcal{F}^{0,2}(1) + j\mathcal{F}^{1,3}(1) \end{array}\right\} \quad (10)$$

The notation is becoming unwieldy, even here in the $N = 4$ case. We could surely use something better. It turns out that a very good way is to show the additions and multiplications graphically. The procedure by which the calculations in Equation $(9A)$ is carried out is shown below.



The data points listed on the left "travel" towards the right. Along the way, they get multiplied by any coefficient which lies next to their path. Where two arrows join up, the incoming values are added at that spot.

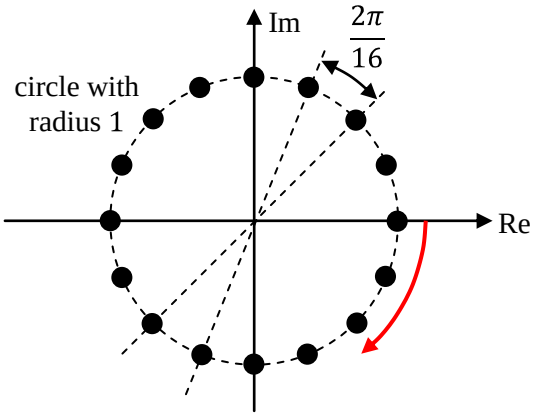
An $N = 4$ analysis brings together two such simpler ones. The diagram illustrating Equation (10) is:



It is not for nothing that diagrams like this are called butterfly diagrams. And, although this particular one is satisfactory for the $N = 4$ case, the notation is still not quite general enough to be expanded easily to higher orders.

The twiddle factors

In the $N = 2$ transform, the values of the two data points are multiplied by only two different coefficients, $+1$ and -1. In the $N = 4$ transform, the values of the four underlying data points are multiplied by four different coefficients, $+1$, $+j$, -1 and $-j$. These four coefficients happen to be the values of a unit vector which starts at $+1$ in the complex plane and is rotated by 90° successively through 90°, 180° and 270°. When we scale up to $N = 8$, the eight coefficients are going to be separated by 45° around the complex plane. More generally, for any N, there are going to be N coefficients which arise from a circumnavigation around the unit circle in increments of $2\pi/N$. These coefficients have been given a name, "twiddle factors". The twiddle factors for $N = 16$ are illustrated in the following figure. The angular spacing is $2\pi/16 = 22.5°$.



I am going to use the following symbol for the twiddle factors:

$$W_N^n = e^{-j\frac{2\pi}{N}n} \quad (11)$$

Since the exponent is algebraically negative, the "direction" of the enumeration as n increases is in the clockwise direction, as shown by the red arrow.

Note the placement of the superscript and subscript in the twiddle factor. W_N^n lies a fraction n/N of the way around the circle.

Successive applications of the Danielson-Lanczos Lemma

Their Lemma is Equation (5), which we've already looked at.

$$\mathcal{F}(k) = \sum_{m=0}^{½N-1} x_{2m} e^{-j2\pi \frac{m}{(½N)}k} + e^{-j2\pi \frac{1}{N}k} \sum_{m=0}^{½N-1} x_{2m+1} e^{-j2\pi \frac{m}{(½N)}k} \qquad (5)$$

I will use notation such that:

$$\mathcal{F}(k) = E(½N, 0) + W_N^k O(½N, 1) \qquad (12)$$

where $E(\mathcal{N}, \mathcal{M})$ is the sum of a series of $\mathcal{N}$ individual terms which are the evenly numbered terms in a series twice as long, where the summing begins with term $x_{\mathcal{M}}$ in the original data. $O(\mathcal{N}, \mathcal{M})$ is the sum of a series of $\mathcal{N}$ individual terms which are the odd numbered terms in a series twice as long, where the summation begins with term $\mathcal{M}$ in the original data. And, of course, W_N^k is the twiddle factor.

Suppose we separate each summation in Equation (5) into two summations. The sub-summations will start at the first and second index, respectively, and will include alternating terms in the series. We can write $\mathcal{F}(k)$ as:

includes data points $x_0, x_4, x_8 \cdots$ — includes data points $x_2, x_6, x_{10} \cdots$

$$\mathcal{F}(k) = \left\{ \begin{array}{l} \sum_{p=0}^{½(½N)-1} x_{2(2p)} e^{-j2\pi \frac{2p}{(½N)}k} + \sum_{p=0}^{½(½N)-1} x_{2(2p+1)} e^{-j2\pi \frac{2p+1}{(½N)}k} + \cdots \\ \cdots + e^{-j2\pi \frac{1}{N}k} \sum_{p=0}^{½(½N)-1} x_{2(2p)+1} e^{-j2\pi \frac{2p}{(½N)}k} + e^{-j2\pi \frac{1}{N}k} \sum_{p=0}^{½(½N)-1} x_{2(2p+1)+1} e^{-j2\pi \frac{2p+1}{(½N)}k} \end{array} \right\} \qquad (13)$$

includes data points $x_1, x_5, x_9 \cdots$ — includes data points $x_3, x_7, x_{11} \cdots$

These summations do not have exactly the same form as the template $E()$ and $O()$ summations defined by Equation (12). They are close, but not exactly the same. Consider the last summation for example. The data point subscript is $2(2p+1)+1$, which starts at 3 when index p is zero. The exponent is $2p+1$, which is 1 when index p is zero. To be equivalent to the defined summation $O()$, the exponent of the first term in the series needs to be 0, not 1. We can reorganize the exponent to make the comparison exact. Any typical term in the last summation can be rearranged as follows.

$$\begin{aligned} x_{2(2p+1)+1} e^{-j2\pi \frac{2p+1}{(½N)}k} &= x_{2(2p+1)+1} e^{-j2\pi \frac{2p}{(½N)}k} e^{-j2\pi \frac{1}{(½N)}k} \\ &= x_{2(2p+1)+1} e^{-j2\pi \frac{p}{½(½N)}k} W_{½N}^k \end{aligned} \qquad (14)$$

The $W_{½N}^k$ factor does not depend on the summation index p, so it can be taken outside the summation. The series being summed now appears in exactly the same form as summation $O()$, and we can write:

$$\sum_{p=0}^{½(½N)-1} x_{2(2p+1)+1} e^{-j2\pi\frac{2p+1}{(½N)}k} = W_{½N}^{k} \sum_{p=0}^{½(½N)-1} x_{2(2p+1)+1} e^{-j2\pi\frac{p}{½(½N)}k}$$
$$= W_{½N}^{k} O(½(½N), 3) \qquad (15A)$$

The other three summations can be rearranged in the same way. The principal step is to convert the denominator in the exponent from $½N$ to $½(½N)$.

First summation:
$$\sum_{p=0}^{½(½N)-1} x_{2(2p)} e^{-j2\pi\frac{2p}{(½N)}k} = \sum_{p=0}^{½(½N)-1} x_{2(2p)} e^{-j2\pi\frac{p}{½(½N)}k}$$
$$= E(½(½N), 0) \qquad (15B)$$

Second summation:
$$\sum_{p=0}^{½(½N)-1} x_{2(2p+1)} e^{-j2\pi\frac{2p+1}{(½N)}k} = \sum_{p=0}^{½(½N)-1} x_{2(2p+1)} e^{-j2\pi\frac{p}{½(½N)}k} e^{-j2\pi\frac{1}{(½N)}k}$$
$$= W_{½N}^{k} O(½(½N), 2) \qquad (15C)$$

Third summation:
$$\sum_{p=0}^{½(½N)-1} x_{2(2p)+1} e^{-j2\pi\frac{2p}{(½N)}k} = \sum_{p=0}^{½(½N)-1} x_{2(2p)+1} e^{-j2\pi\frac{p}{½(½N)}k}$$
$$= E(½(½N), 1) \qquad (15D)$$

We can substitute these expressions back into Equation (13) to get:

$$\mathcal{F}(k) = \begin{Bmatrix} E(½(½N), 0) + W_{½N}^{k} O(½(½N), 2) + \cdots \\ \cdots + e^{-j2\pi\frac{1}{N}k} E(½(½N), 1) + e^{-j2\pi\frac{1}{N}k} W_{½N}^{k} O(½(½N), 3) \end{Bmatrix} \qquad (16)$$

Note that the exponential coefficient which remains is itself a twiddle factor, so this can be reduced to:

$$\mathcal{F}(k) = \begin{Bmatrix} E(½(½N), 0) + W_{½N}^{k} O(½(½N), 2) + \cdots \\ \cdots + W_{N}^{k} E(½(½N), 1) + W_{½N}^{k} W_{N}^{k} O(½(½N), 3) \end{Bmatrix} \qquad (17)$$

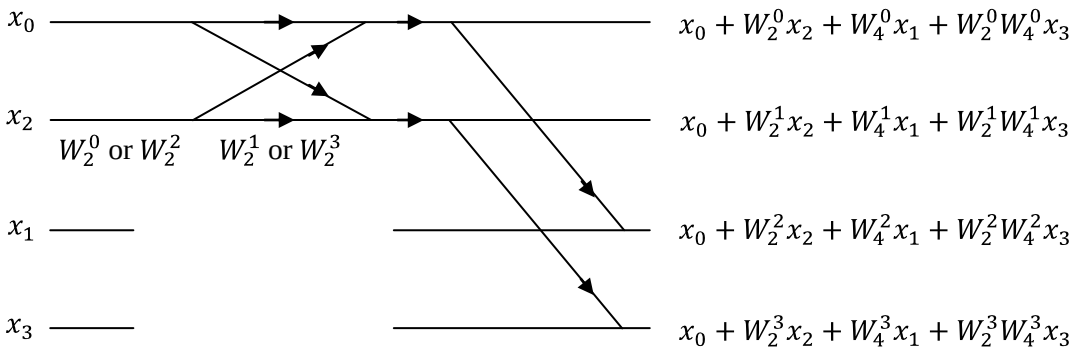
If our set of data consisted of only four points, so $N = 4$, then the summations would each consist of one term. We would find that:

$$\left.\begin{aligned} E(½(½N), 0) &= x_0 \\ O(½(½N), 2) &= x_2 \\ E(½(½N), 1) &= x_1 \\ O(½(½N), 3) &= x_3 \end{aligned}\right\} \qquad (18)$$

and that Equation (17) reduces to:

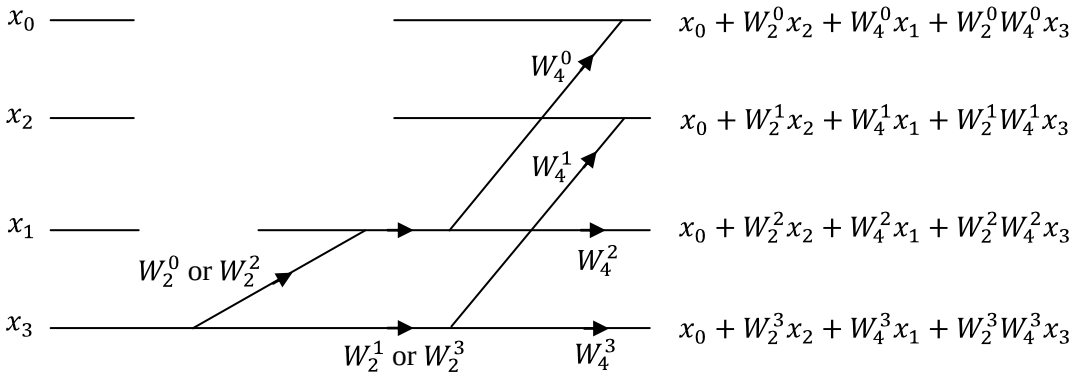
$$\mathcal{F}(k) = x_0 + W_2^k x_2 + W_4^k x_1 + W_2^k W_4^k x_3 \qquad (19)$$

When we try to construct a butterfly diagram for the four frequency components, we immediately run into problems. The following partial diagram shows the problem just moving the second data point x_2 from the left side to the right side. To meet the needs of $\mathcal{F}(0)$ and $\mathcal{F}(1)$, we need to use twiddle factors W_2^0 and W_2^1. But to meet the needs of $\mathcal{F}(2)$ and $\mathcal{F}(3)$, we need to use twiddle factors W_2^2 and W_2^3.

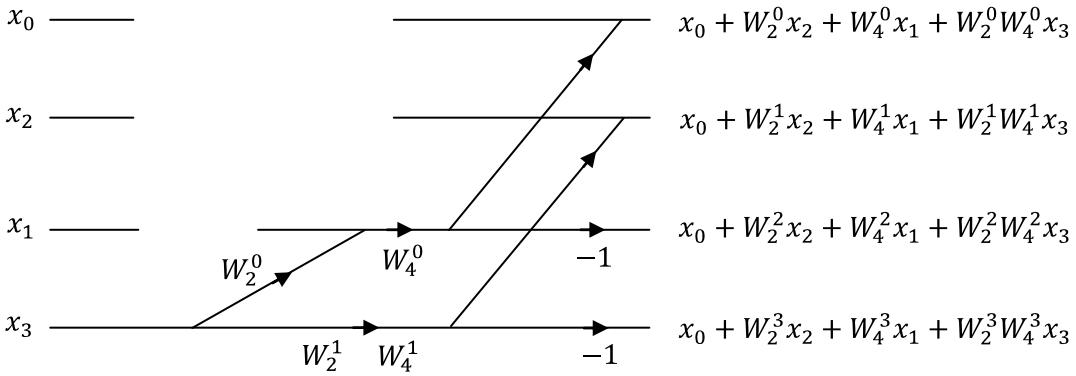


Because of the nature of the twiddle factors – being unit vectors with regularly spaced polar angles in the complex plane – it turns out that $W_2^0 = W_2^2$ (and both equal $+1$ to boot). Similarly, $W_2^1 = W_2^3$ (and both are equal to -1). It both cases, the second twiddle factor is rotated by an angle 2π from the first one, and represents exactly the same spot in the complex plane.

Now let's see what is required to move the data point x_3 from left to right.

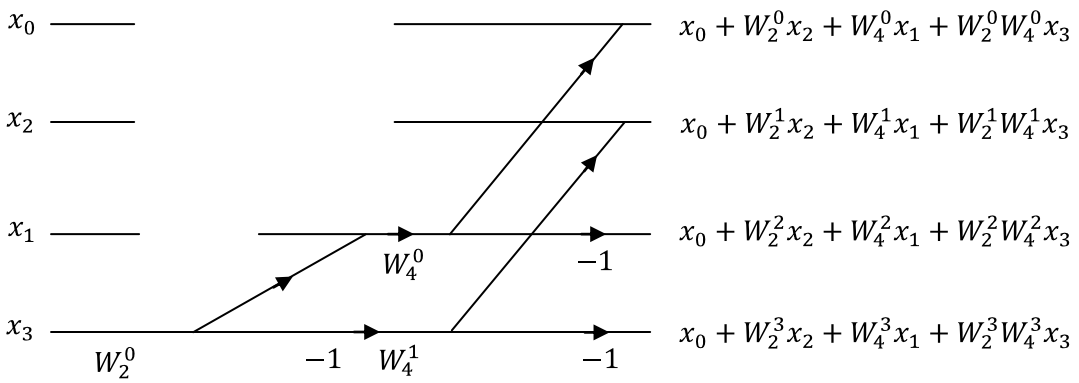


The choice of coefficients "W_2^0 or W_2^2" and "W_2^1 or W_2^3" are more occurrences of the twiddle relationship we just encountered. They are easily dealt with. What we need to do is simply the group of coefficients W_4^0, W_4^1, W_4^2 and W_4^3. They have a similar relationship, but in the odd variant. Here, $W_4^2 = -W_4^0$ and $W_4^3 = -W_4^1$. The difference between superscripts in this group corresponds to a half-rotation in the complex plane, which is to say, a reversal of sign. These relationships allow us to simplify the diagram for x_3 to:

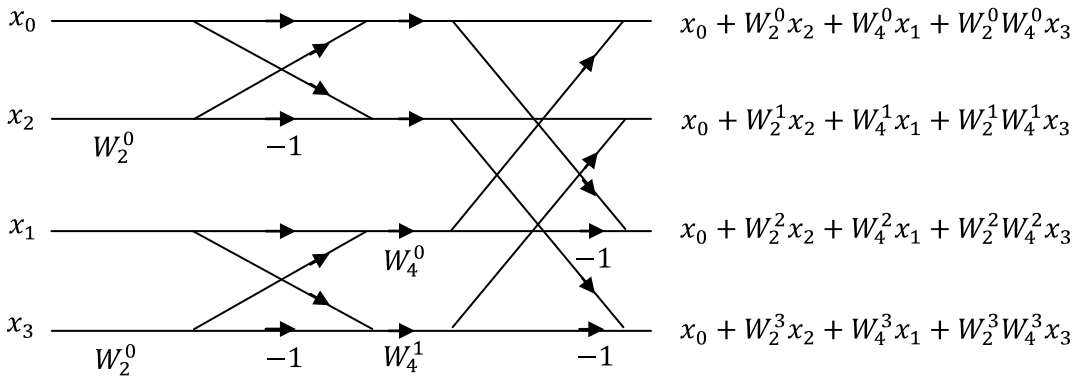


.

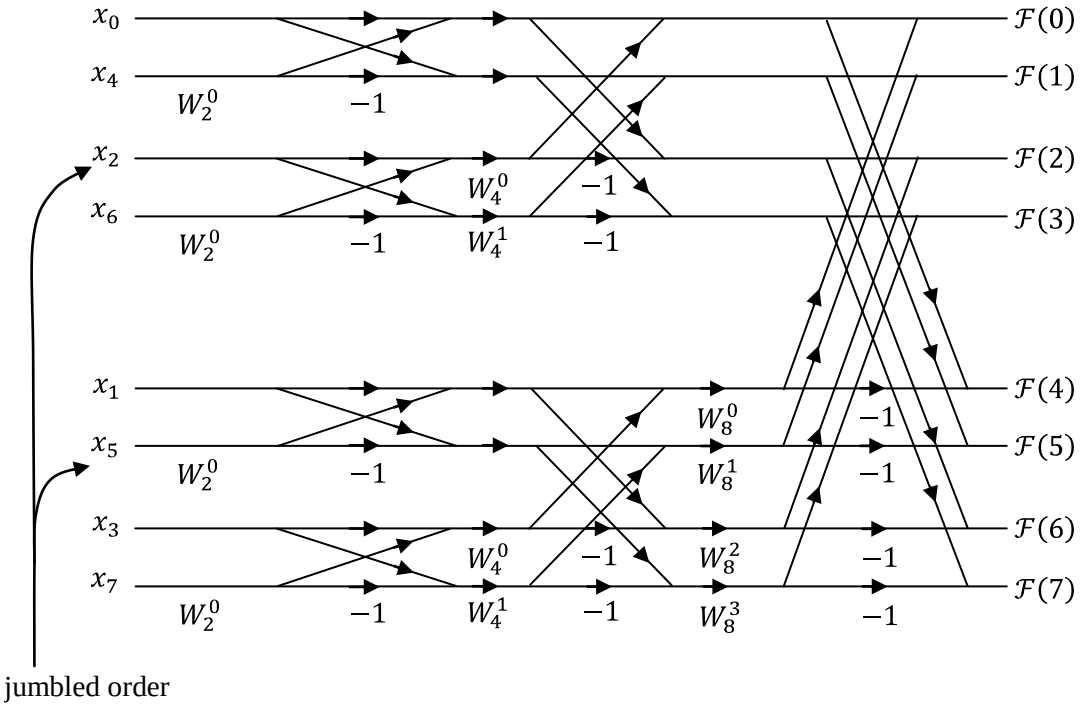
I should have observed that we could handle the pair of coefficients W_2^0 and W_2^1 in the same way. Without getting specific about their particular values, it is the case that $W_2^1 = -W_2^0$. We can move the location of the coefficient so it appears before the fork in x_3's path. Then we have:



I am not going to examine the trajectory of x_1 on a stand-alone basis. Instead, I will combine the network we have developed for x_0, x_2 and x_3 and see what changes need to be made to accomodate x_1. Here is what we have so far.



I have not made the obvious substitution (yet) that $W_2^0 = W_4^0 = 1$. I have not done so because it would hide some of the regularity which is otherwise apparent when the procedure is extended to $N = 8$.



The so-called "bit reversal" scheme

As the number of data points increases, the ordering of the points as they enter the Transform becomes more and more jumbled up. Fortunately, a regularity has been observed which makes keeping track of the points quite simple. Since everything in this procedure depends on powers of two and factors of two, it will not be a surprise to learn that this phenomenon is also a binary one. The order in which the data points are accessed in the $N = 8$ case is as follows:

$$x_0 \;\; x_4 \;\; x_2 \;\; x_6 \;\; x_1 \;\; x_5 \;\; x_3 \;\; x_7 \quad (20)$$

Let's write down the integers $0, 1, 2, 3, 4, 5, 6, 7$ in binary format. We get:

$$000 \;\; 001 \;\; 010 \;\; 011 \;\; 100 \;\; 101 \;\; 110 \;\; 111 \quad (21)$$

Now, let us reverse the ordering of the bits in each number. Note that reversing the order is not the same as complementing the bits, which is a quite different binary operation. We get:

$$000 \;\; 100 \;\; 010 \;\; 110 \;\; 001 \;\; 101 \;\; 011 \;\; 111 \quad (22)$$

Now, let's convert this sequence back into normal integers. They are:

$$0 \;\; 4 \;\; 2 \;\; 6 \;\; 1 \;\; 5 \;\; 3 \;\; 7 \quad (23)$$

which happens to be exactly the indices of the data points in the order which they appear on the left-hand side. This is all there is to "bit reversal". It is a very convenient way to be able to select the data points in the order we are going to want to process them.

An overview of the subroutine

I am going to change tack here, and talk about some of the details of how the VisualBasic subroutine implements the procedure. For this description, I am going to use N as the number of data points in the sample. That's how we have been using N above. What I really mean is that I will not fix $N = 16{,}384$, so tht ny power of two can be used (up to the limit of the computer's ability to store integers).

The input data values are going to be stored in two N-length vectors. These will be the Real and Imaginary parts of the data points. Even though I expect to process real values only, I am going to set up the subroutine to handle complex data sets as well. The vector names will be DataRe() and DataIm() and the indexing in each vector will run from 0 to $N - 1$. VisualBasic allows for zero-based indexing and I will use it.

After the subroutine has done all of its work, the discrete-time Fourier Transform will be returned in another pair of N-length vectors. These will be the Real and Imaginary parts of the frequency components. The vector names will be DTFTRe() and DTFTIm() and the indexing in each vector will also run from 0 to $N - 1$.

The main subroutine is called DTDT_Forward(). Since passing arguments to and from subroutines is not a particularly fast operation, I will not do it. Instead, both the input data vectors and the DTFT output vectors will be declared as Public variables. The calling routine can therefore load the data directly into the input data vectors and read the results directly from the DTFT result vectors. One of the objectives I set for the subroutine is that it not change the values in the input data vectors DataRe() and DataIm().

I will pre-compute the bit-reversal integers and store them in such a way that they do not need to be re-computed to process a second set of data points. In fact, what I will do is use a special purpose subroutine to initialize all the things needed to process samples of size N, including the bit-reversal indices and the twiddle factors. This subroutine will be called DTFT_Initialization(). It must be called by the calling routine before any real processing gets under way. The pre-computed bit-reversed indices will be stored in another Public variable, called ReverseBits().

I want to say a word about how the bit reversal is carried out. In VisualBasic, there are several types of integer data types. For example, there are Int32 and UInt32 data types (and others as well). These are both integers, and both are stored in 32 bits, or four bytes. But the former are signed integers and the latter are unsigned integers. In the former, negative integers are stored in two's-complement notation, so the most significant bit of the four bytes indicates the algebraic sign of the integer, with a negative bit there for algebraically negative numbers. The low-order 31 bits are the same for both data types. Now, the low order 31 bits can accommodate positive integers in the range from zero to b'111...1', where there are 31 ones in the binary representation. This corresponds to our base-10 number $2^{32} - 1$, or about 4.3 billion. There is no way (on my ThinkPad) that we are going to have that many data points in every sample. From a practical point-of-view, it does not matter whether we carry out the binary arithmetic for bit reversal using Int32 data types or UInt32 data types.

Aside: A reader might ask why I even raised the point, then? Here's why. In VisualBasic, addressing a specific item in a vector or array requires that the index be a signed integer. Addressing using an unsigned integer is not permitted. There are ways to convert from one data type to the other using the CInt() and CUInt() functions, but these conversions make the code very dense. All of these conversions can be avoided if the Int32 data type suffices, as it does.

Binary arithmetic is carried out in VisualBasic using the same And, Or and Not logical operators that are more commonly applied to Boolean variables. Some interesting things follow. For example, the expression (Integer And 1) selects the least significant bit of variable Integer. Another example: the expression (Integer * 2) has the effect of shifting the binary contents of Integer to the left by one space. Similarly, the expression (Integer / 2) has the effect of shifting the binary contents to the right by one. A review of subroutine PopulateReverseBitVector() will show how expressions like these are used to produce a N-length vector whose elements are the reversed-bit integers.

The twiddle factors will also be pre-computed by subroutine DTFT_Initialization() before the real data processing gets under way. The twiddle factors themselves will be stored as Public variables. Since the twiddle factors are complex numbers, I will store them in vectors named TwiddleRe() and TwiddleIm(). Note that the twiddle vectors only need to be declared with length $½N$, not length N. We only need the twiddle factors for half the unit circle in the complex plane, since the other half can be dealt with (and was dealt in the butterfly diagrams) using sign reversals.

Because the twiddle factors are only calculated once, the routine which does the calculations does not have to be particularly efficient. The code does not take advantage of the many cosine and sine regularities which could make their computation faster.

As we work through the stages from the left side of the butterfly diagram to the right side, we are going to need a bit of temporary storage. One could declare two N-length vectors to hold the interim results, one for the Real parts and the other for the Imaginary parts. However, the two output vectors DTFTRe() and DTFTIm() are not relevant until all the calculations have been completed. I have therefore used them as the main temporary storage for interim results. Indeed, if we use them for temporary storage, then we can even skip the final step at the end of the calculations which would otherwise be needed to move the results into these vectors. Despite the use of vectors DTFTRe() and DTFTIm() as temporary storage, I

have written the code in such a way that it uses two additional pairs of temporary storage vectors. Let me describe these other vectors.

The complete vector of twiddle factors has length ½N. When we are processing the interim stages, however, only some of the twiddle factors are needed for each stage. For example, when processing stage #4, the relevant twiddle factors are the following:

$$\text{Twiddle}() = \left(W_{16}^{0}\ W_{16}^{1}\ W_{16}^{2}\ W_{16}^{3}\ W_{16}^{4}\ W_{16}^{5}\ W_{16}^{6}\ W_{16}^{7}\right) \quad (24)$$

But the complete vectors of twiddle factors TwiddleRe() and TwiddleIm() are not based on denominator 16 (splitting the unit circle into 16 equal angles), they are based, say, on denominator 16,384 (splitting the unit circle into N equal angles). While it is true that the needed $1/16^{\text{th}}$ values are included in TwiddleRe() and TwiddleIm(), because $W_{16}^{3} = W_{16,384}^{3,072}$ and $W_{16}^{7} = W_{16,384}^{7,168}$ for example, some playing around with indices is required to select the right numbers. It would speed things up if a separate pair of twiddle vectors was pre-computed for each stage of processing. But, I have not taken this route. It takes too much storge space. For $N = 16{,}348$, an $8{,}192 \times 8{,}192$ array of twiddle factors would be needed. What I have done instead is this. At the beginning of each stage, I have extracted from TwiddleRe() and TwiddleIm() only those factors needed for this stage, and saved them in temporary storage vectors TempTwiddleRe() and TempTwiddleIm() where they can be indexed directly by the loop index for that stage.

The second pair of additional storage vectors holds the products of multiplying the top-half inputs for each stage by the twiddle factors for that stage. These two vectors are called TempProductRe() and TempProductIm(). There use is not essential, of course, as more complicated formulae could be used to carry these products right through to the end of the stage. However, as is always the case, more complicated formulae take more time to execute, particularly if they involve repeated references to vector or array elements.

The VisualBasic module includes a special purpose subroutine ComplexMult() which multiplies two complex numbers. The first four arguments are the real and imaginary parts of the multiplicand and the multiplier, respectively. The last two arguments are the real and imaginary parts of the product. They are declared as ByRef arguments, and are passed back to the calling routine by position in the list of arguments. Using a subroutine for multiplication is not as fast as would be coding of the steps at each required point in the procedure. In my case, execution speed was good enough without having to do so.

In the attached code, I have not condensed all stages into one single loop. I have coded the first three stages (up to stride length 8) separately and explicitly. It's not so elegant, but it certainly makes it much easier to see how the generalized loop for stages 4 and up works. And, it is likely faster since less indexing is required for each stage which is coded explicitly.

First numerical example

The module DTDTmodule.vb is listed in Appendix "A" attached. The listing also includes a small Windows Form called Form1.vb which I used to test the results. The form has a couple of buttons and a label area in which results can be displayed. As a first numerical example, I used the following test waveform:

$$v(t) = 5\cos(2\pi 1024t) + 2\sin(2\pi 128t) \quad (25)$$

I digitized this assuming $N = 16{,}384$, which is equivalent to $N = 2^{14}$. The following is the loop used to sample the data and then to initialize and execute the forward transform.

```
' Generate some test data for a dual-tone waveform in the time domain
NF = CInt(2 ^ 14)
For I As Int32 = 0 To (NF - 1) Step 1
    DataRe(I) = _
        (5 * Math.Cos(TwoPi * 1024 * I / NF)) + _
        (2 * Math.Sin(TwoPi * 128 * I / NF))
    DataIm(I) = 0
Next I
' Initialize and then execute the forward transform
DTFT_Init_Forward()
DTFT_Forward()
```

After a bit of post-processing, the display on the screen was this:

```
K=128:        DTFTRe = 0          DTFTIm = -1
K=1024:       DTFTRe = 2.5        DTFTIm = 0
K=15360:      DTFTRe = 2.5        DTFTIm = 0
K=16256:      DTFTRe = 0          DTFTIm = 1
```

Before I explain why this is (or is not) what one expects, I will describe the post-processing. The default arithmetic in VisualBasic is double precision, using floating point (scientific notation) numbers stored in eight bytes. This allows for 15 digits of precision in representing base-10 numbers. After tens of thousands of multiplications and additions, and subtractions of numbers which are theoretically equal, a certain degree of imprecision creeps into the numbers. Numbers which ought to be zero are not exactly equal to zero. In this numerical example, the residuals for values which ought to be zero was in the order of 10^{-11}. To avoid being distracted by these negligible bits of noise, I zeroed-out all frequency components whose magnitude was 10^{-8} or less. I used the following loop for this purpose.

```
' Set exactly to zero the near-zero results of machine imprecision
For K As Int32 = 0 To (NF - 1) Step 1
    If (Math.Abs(DTFTRe(K)) < Val("1E-8")) Then
        DTFTRe(K) = 0
    End If
    If (Math.Abs(DTFTIm(K)) < Val("1E-8")) Then
        DTFTIm(K) = 0
    End If
Next K
```

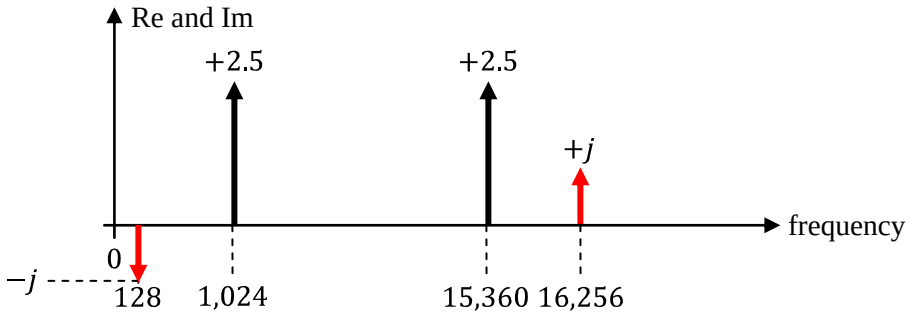
As the next step in the post-processing, I divided all frequency components by $N = 16{,}384$. Why? The DTFT forward transform is a summation. If a particular cosine term in the time domain function $f(t)$ has a magnitude of one every time it is sampled, it will appear in all x_n data points, and its amplitude (one) will be added into the summation N times. Its amplitude is over-counted by a factor of N. We can undo this over-counting by dividing the summation for each frequency component by N. In fact, many authors actually include a leading factor $1/N$ in their definitions of the forward transform in Equation (1). I chose not to do so simply to avoid including in the code a division which does not actually have to be carried out during the transform process itself. The division can be done in post-processing, as I have done here, where it can be included along with other operations which are essential to one's application. This saves execution time. My division by N, often called a normalization, was carried out by the following loop.

```
' Divide all DTFTRe() and DTFTIm() by NF
For I As Int32 = 0 To (NF - 1) Step 1
    DTFTRe(I) = DTFTRe(I) / NF
    DTFTIm(I) = DTFTIm(I) / NF
Next I
```

For the sake of completeness, the loop which prepared the output string was the following.

```
' Prepare for display a string with any non-zero frequencies
Dim DisplayString As String = ""
For K As Int32 = 0 To (NF - 1) Step 1
    If ((DTFTRe(K) <> 0) Or (DTFTIm(K) <> 0)) Then
        DisplayString = DisplayString & _
            "K=" & Str(K) & ":" & _
            "  DTFTRe(K)=" & Str(DTFTRe(K)) & _
            "  DTFTIm(K)=" & Str(DTFTIm(K)) & vbCrLf
    End If
Next K
labelResults.Text = DisplayString
```

All right, having described how the results of the forward transform were processed before displaying them, let's turn to the results themselves. The time-domain waveform consisted of one cosine term and one sine term, yet four different frequencies appeared in the spectrum. This is not a mistake. I am going to take the liberty of plotting both the Real components (in black) and the Imaginary components (in red) on the same vertical axis. Here is the frequency spectrum as reported by subroutine DTFT_Forward().



The black arrows happen to represent the cosine term in the original waveform. The sum of the magnitudes of the two black arrows $(2.5 + 2.5 = 5)$ is equal to the magnitude (in Volts, say) of this component in the original waveform. The red arrows correspond to the sine term. The magnitudes (in the sense of length) of these two arrows is 2, the magnitude of the sine term in the original waveform.

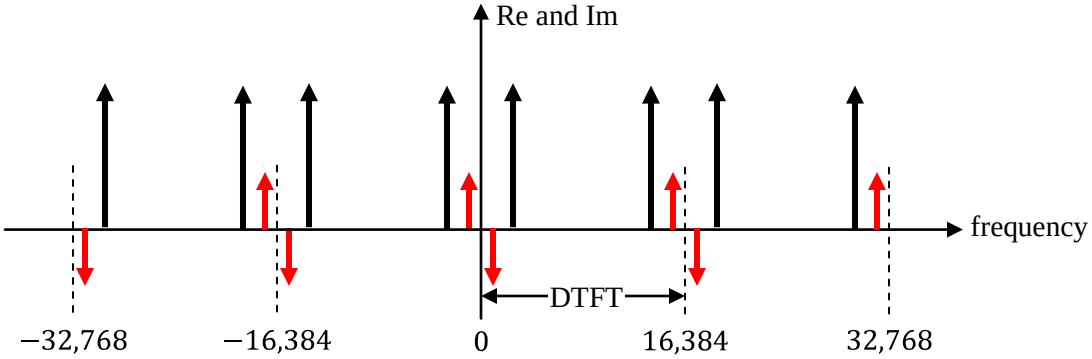
The lower two frequencies shown in the plot correspond to their time-domain source waveforms. It is the upper two frequencies which seem to be out-of-place. These two frequencies are not random, though. Notice that they are 16,384 less the two fundamental frequencies. In particular,

$$\left.\begin{aligned} 16{,}384 - 16{,}256 &= 128 \\ 16{,}384 - 15{,}360 &= 1{,}024 \end{aligned}\right\} \quad (26)$$

The conundrum is resolved if we recall an important assumption we made at the outset. We assumed that the N discrete data points constituted one "period" and that the sequence was repeated over and over into both positive and negative times. In other words, the sample of data was assumed to be periodic in time with a period equal to $N\Delta P$, where ΔP is the time step between data points.

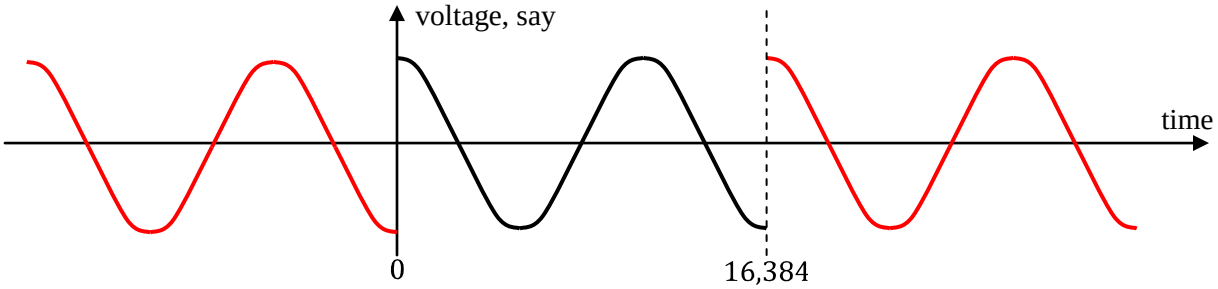
Since the data stream was assumed to be periodic, the frequency spectrum of its components will also be periodic. This is shown in the following graph. The results of the forward transform only represent one period of the spectrum, being the period whose extent I have labeled as DTFT in the graph. The components in this base period are repeated every 16,384 Hz in both frequency directions.

:



Second numerical example

The frequencies of the two waveforms in the time-domain data in the first example were integral fractions of 16,384 Hz. The periodic extensions of the data did not involve any discontinuities. Let's contrive an example where the periodic extension is discontinuous. An example is the following cosine waveform, whose period is such that its value at the end of the sample period is not the same as its value at the start of the sample period.



I have used the following waveform in this second numerical example:

$$v(t) = 5\cos\left(2\pi\frac{13}{2}t\right) \quad (27)$$

In discretized form, it is:

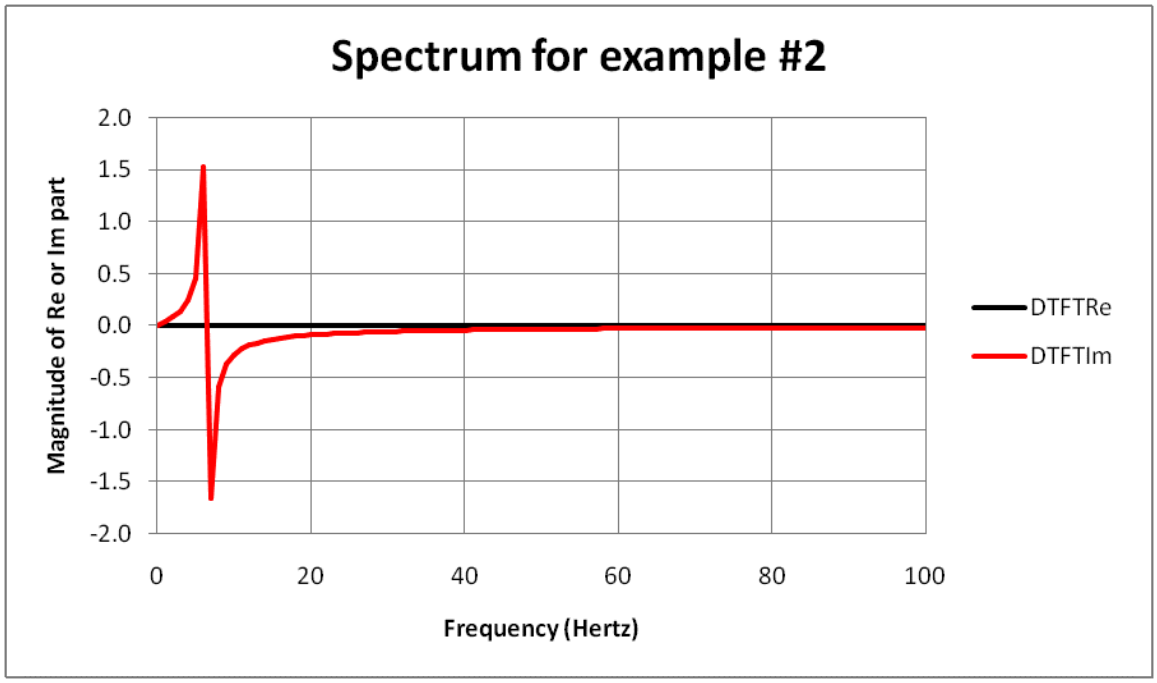
$$x_n = 5\cos\left(2\pi\frac{13}{2}\frac{n}{16384}\right) \; for\; n = 0, 1, 2, \cdots, 16{,}383 \quad (28)$$

Check that, at $n = 0$ when the sample period begins, $x_0 = 5$. At $n = 16{,}384$, which is the time of the first data point in the first periodic repetition to the right, $x_{16,384} = 5\cos(13\pi) = -5$. Physically, the waveform in Equation (27) has a frequency of 6½ Hz.

After running the forward transform, reducing to zero the artifacts of machine arithmetic, and normalizing by dividing all frequency components by 16,384, the first few frequency components are:

```
K=0:          DTFTRe = 0.000305175781249763       DTFTIm = 0
K=1:          DTFTRe = 0.000305175781249759       DTFTIm = 0.0385830360127066
K=2:          DTFTRe = 0.000305175781249733       DTFTIm = 0.0832183099088844
K=3:          DTFTRe = 0.000305175781249682       DTFTIm = 0.1435985034116860
K=4:          DTFTRe = 0.000305175781249600       DTFTIm = 0.2425218960671190
K=5:          DTFTRe = 0.000305175781249370       DTFTIm = 0.4613187731565900
K=6:          DTFTRe = 0.000305175781248174       DTFTIm = 1.5278875707156900
K=7:          DTFTRe = 0.000305175781251779       DTFTIm = -1.650495569599100
K=8:          DTFTRe = 0.000305175781250512       DTFTIm = -0.585397335787591
K=9:          DTFTRe = 0.000305175781250309       DTFTIm = -0.369650014856724
K=10:         DTFTRe = 0.000305175781250189       DTFTIm = -0.275592779995049
K=11:         DTFTRe = 0.000305175781250216       DTFTIm = -0.222311451979611
K=12:         DTFTRe = 0.000305175781250123       DTFTIm = -0.187700927318971
K=13:         DTFTRe = 0.000305175781250086       DTFTIm = -0.163235585495960
K=14:         DTFTRe = 0.000305175781250095       DTFTIm = -0.144921301119241
K=15:         DTFTRe = 0.000305175781250129       DTFTIm = -0.130633039639302
K=16:         DTFTRe = 0.000305175781250050       DTFTIm = -0.119133212564099
K=17:         DTFTRe = 0.000305175781250049       DTFTIm = -0.109650490383235
K=18:         DTFTRe = 0.000305175781250059       DTFTIm = -0.101678050874574
K=19:         DTFTRe = 0.000305175781250057       DTFTIm = -0.0948684582170478
K=20:         DTFTRe = 0.000305175781250039       DTFTIm = -0.0889751196528697
```

Whatever is going on here is not obvious from the table. A more informative way to present the data is to plot it. The following graph shows the Real and Imaginary frequency components up to 100 Hz.



Although the time-domain waveform is a cosine, there aren't any cosine-type frequency components, which are represented by the Real part of the spectrum. The reason is that the time-domain waveform is not symmetric around the time $t = 0$. The only way the Transform can handle asymmetric waveforms is to use sine components, which show up as the Imaginary components in the spectrum.

The example I have used here is pretty extreme. Even so, this phenomenon is a definite problem when real-time data is being sampled and transformed on an ongoing basis. Taking slices, or windows, of the data stream as it passes by compromises those waveform components which do not fit nicely into the duration of the window.

I am going to try a few more examples but, before I do, I want to describe the inverse transform. Alert readers may have seen that I referred to the DTFT transform we looked at above as the "forward" transform. That is to distinguish it from the inverse, or reverse, transform which does the work in the opposite direction, taking frequency components and transforming them back into the time domain.

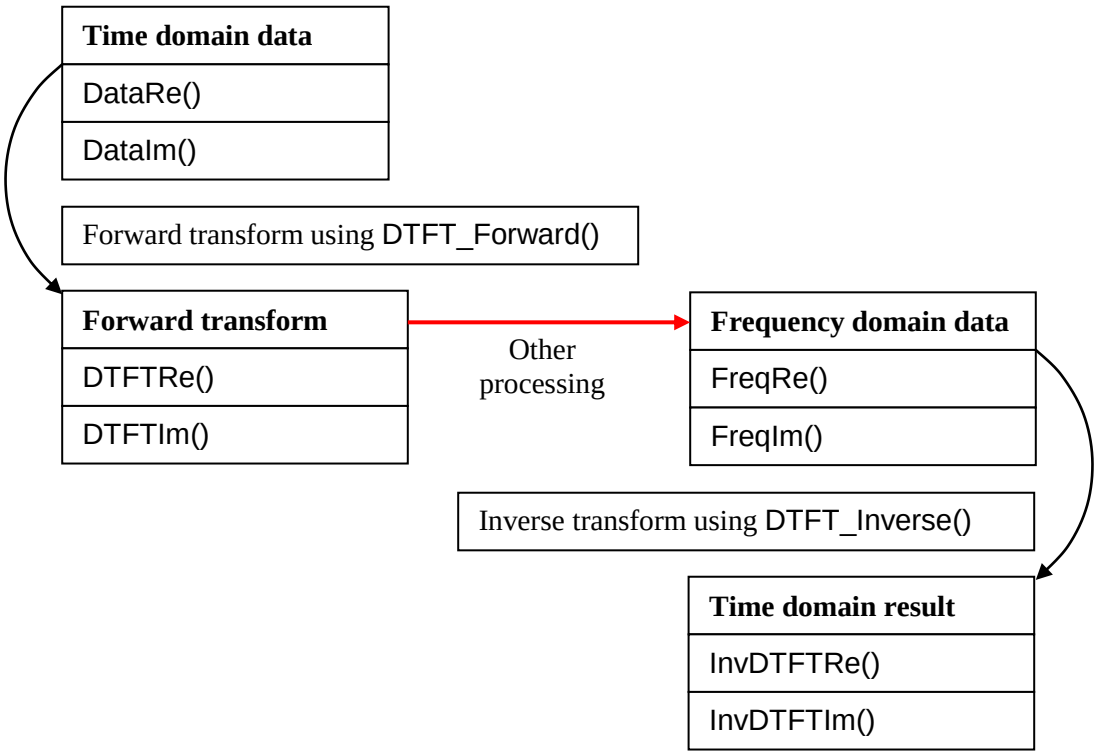
The inverse discrete-time Fourier Transform

The inverse transform, set out in Equation (2) above, is almost identical to the normal, or "forward", transform in Equation (1). The sole difference is the algebraic sign of the exponent of the exponential term.

Given all I have said about the regularities and symmetries inherent in the twiddle factors, it will come as no surprise that the procedures to solve the two transforms are almost identical. The sole difference is the algebraic sign of the imaginary part of the twiddle factors. Instead of proceeding clockwise around the unit circle in the complex plane as we did above, the twiddle factors for the inverse transform arise from a counter-clockwise traverse around the unit circle.

Indeed, many implementations of fast Fourier Transforms use exactly the same code for both transformations. If the twiddle factors are computed on-the-fly as they are needed, a Boolean flag can be used to tell the procedure whether a positive or negative sign should be used.

I have chosen to include a separate and distinct subroutine in DTFTmodule.vb to handle the inverse transform. It is called DTFT_Inverse(). I have a reason for doing this. The application I have in mind for this module is going to require real-time inversion at the same time as real-time transformation of the raw data. I do not want to use the same buffers for the input and output data. The way it works is this.



To use the forward transform, time-domain data is loaded into vectors DataRe() and DataIm(). After the forward transform DTFT_Forward() has been executed, the frequency components can be read from the

vectors DTFTRe() and DTFTIm(). To use the inverse transform, the frequency components are loaded into input vectors FreqRe() and FreqIn(). After the inverse transform DTFT_Inverse() has been run, the sequence of time-domain samples can be read from output vectors InvDTFTRe() and InvDTFTIm().

It will almost certainly be the case that one will want to carry out some processing on the results of a forward transform before synthesizing a new time-domain waveform using the inverse transform. This processing is illustrated by the red arrow in the flowchart above.

It is not necessary that the forward and inverse transforms use the same sample size N. The subroutines have been written so that separate sample sizes can be used for the two directions: NF for the forward transform and NI for the inverse transform. The code for the two subroutines is virtually the same, but to avoid any confusion I have included a suffix "F" to the variables names used in the forward transform and suffix "I" to the variable names used in the inverse transform.

Third numerical example

Let's run a simple cosine waveform through the forward transform, and then try to reconstruct it using the inverse transform. For convenience, I am going to use the same sample size (NF = NI = 16,384) going both directions. As the original data, I used the following $512\ Hz$ cosine waveform.

$$v(t) = 3\cos(2\pi 512t) \quad (28)$$

whose sampled values are:

$$x_n = 3\cos\left(2\pi 512\frac{n}{16384}\right)\ for\ n = 0, 1, 2, \cdots, 16{,}383 \quad (29)$$

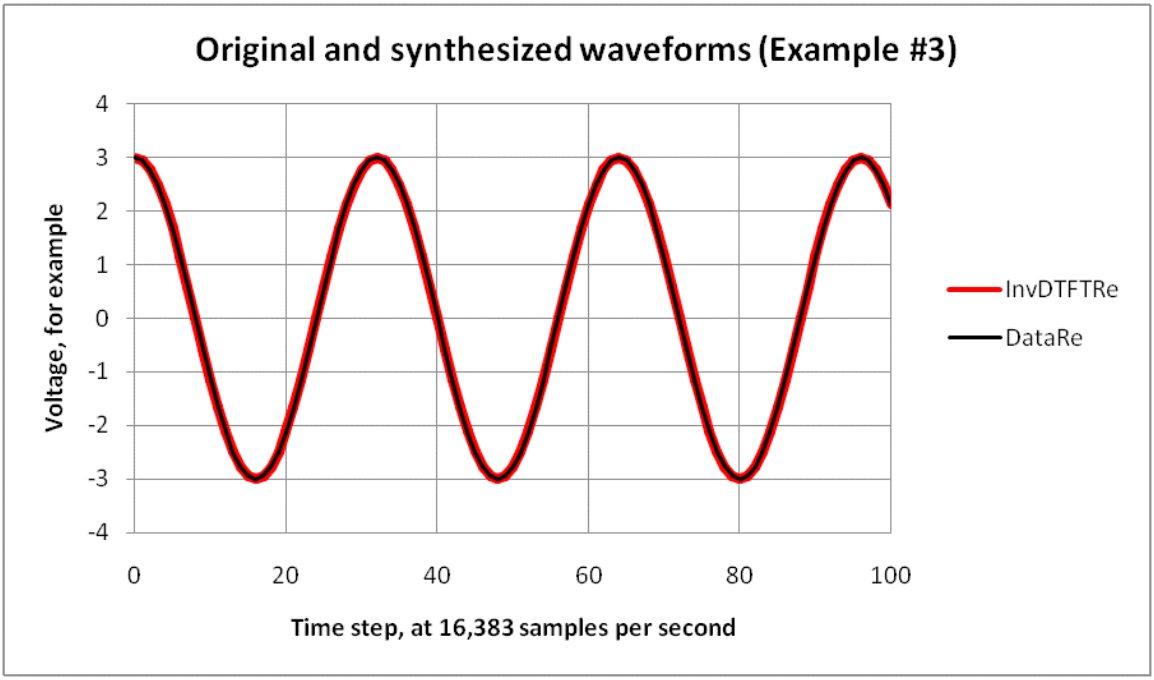
The forward transform (after the same kind of post-processing as above) gives two frequency components, one at $512\ Hz$ and the second at $15{,}872\ Hz$:

```
K=512:          DTFTRe = 1.5          DTFTIm = 0
K=15872:        DTFTRe = 1.5          DTFTIm = 0
```

I will take these two components, exactly as they are, transfer their values to the input vectors FreqRe() and FreqIm() used by the inverse transform, and then run the inverse transform. This is the code that carries out these steps.

```
NI = 16384
' Third numerical example
' Generate some test data in the frequency domain
For I As Int32 = 0 To (NF - 1) Step 1
    FreqRe(I) = 0
    FreqIm(I) = 0
Next I
FreqRe(512) = 1.5
FreqRe(16384 - 512) = 1.5
' Initialize and then execute the inverse transform
DTFT_Init_Inverse()
DTFT_Inverse()
```

The results of the inverse transform are located in the output vectors InvDTFTRe() and InvDTFTIm(). I exported these values into Excel and plotted the Real component (the red trace in the following graph). For the sake of comparison, I also plotted the original cosine function on the same graph (the black trace).



The inverse transform re-creates the original waveform exactly. The Imaginary component generated by the inverse transform is zero for all times, just like the original time-domain waveform.

Fourth numerical example

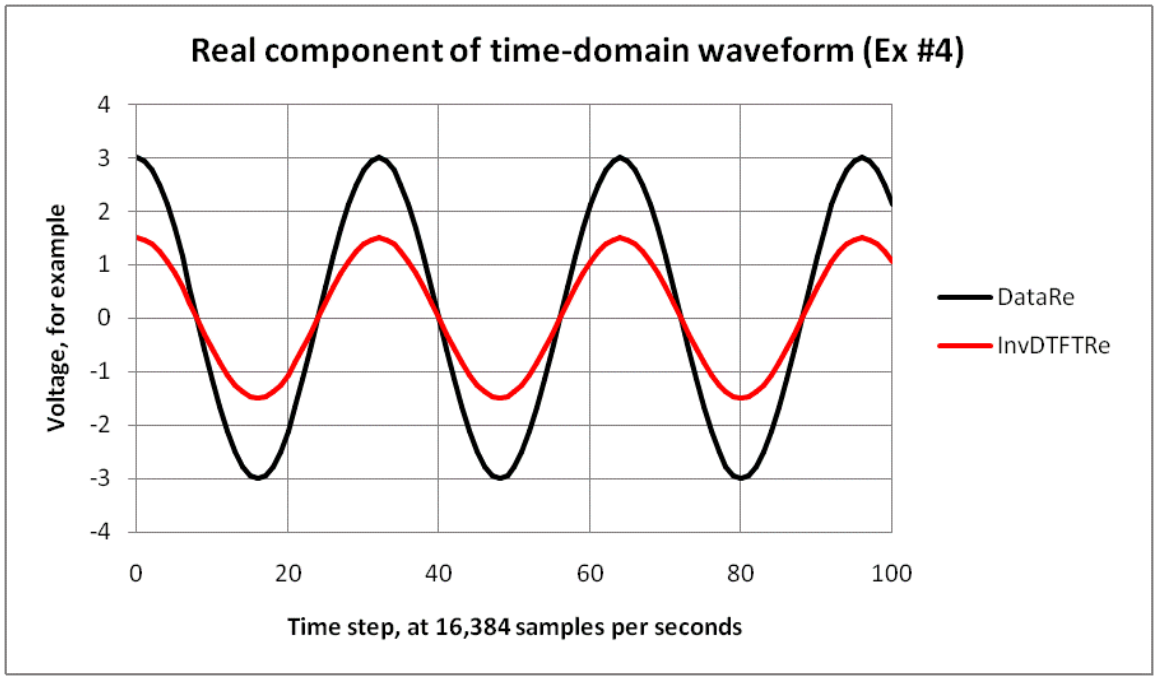
I am going to repeat Example #3, but with a small difference. The frequency spectrum of the cosine waveform has two components, one at 512 Hz and the other at 15,872 Hz. In Example #3, both of these frequency componnts were input into the inverse transform, which then synthesized the original waveform.

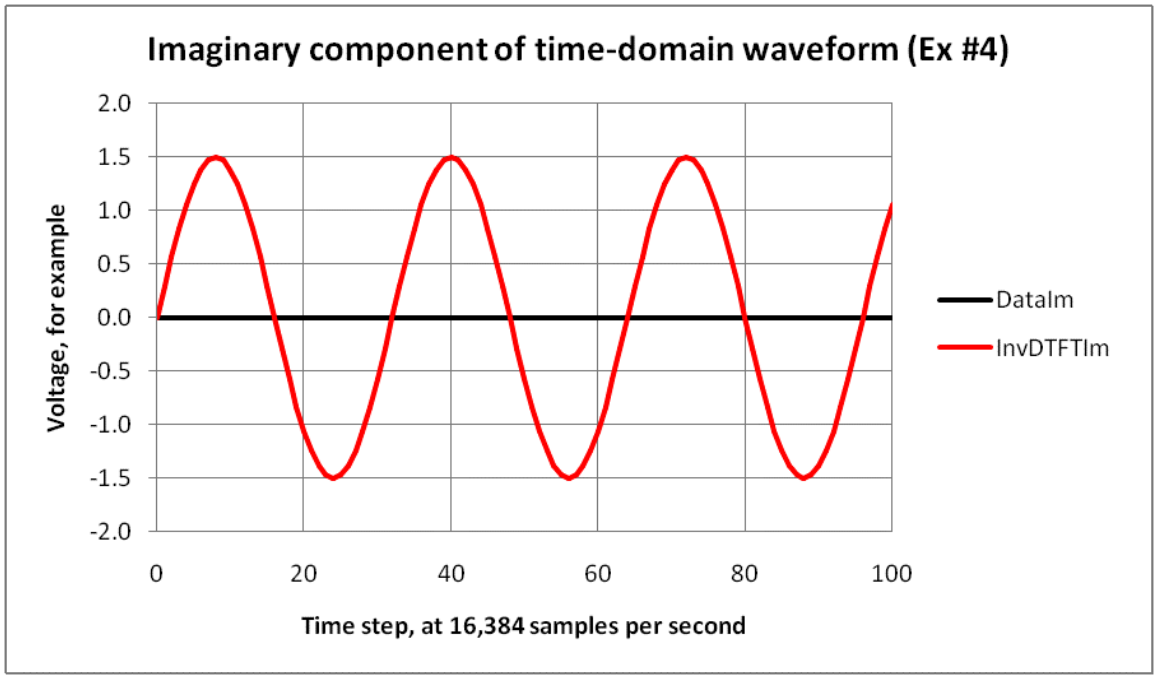
In this example, I want to find out what happens if only one of the frequency components, the one at $512\ Hz$, is sent to the inverse transform. Here's the control program I used.

```
NI = 16384
' Fourth numerical example
' Generate some test data in the frequency domain
For I As Int32 = 0 To (NF - 1) Step 1
    FreqRe(I) = 0
    FreqIm(I) = 0
Next I
FreqRe(512) = 1.5
' Initialize and then execute the inverse transform
DTFT_Init_Inverse()
DTFT_Inverse()
```

The following two graphs show the Real and Imaginary parts, respectively, of the original data (the black curve) and of the time sequence generated by the inverse transform acting on only the one frequency component (the red curve).

The synthesis is not perfect.





The Real part of the inverse transform has the same shape as the original cosine, but only one-half the amplitude. The inverse transform also generates an Imaginary waveform which has no counterpart in the original signal.

The original cosine function is <u>not</u> recovered, at least not in the perfect shape one would like. It looks as if both frequency components are required for the inverse transform to do a proper job.

Appendix "A" is a listing of the DTFT module and the Windows Form application which carried out the numerical examples. The code was developed in Visual Basic 2010 Express.

Jim Hawley

If you found this description helpful, please let me know. If you spot any errors or omissions, please send an e-mail. Thank you.

Appendix "A"

Listing of the VB2010 code

The program consists of a Windows Forms application (Form1) and the module DTFT_Module.vb.

Windows Form application Form1

```
Option Strict On
Option Explicit On

Public Class Form1
    Inherits System.Windows.Forms.Form

    Public Sub New()
        InitializeComponent()
        With Me
            Text = "Fast Fourier transform"
            FormBorderStyle = Windows.Forms.FormBorderStyle.None
            Size = New Drawing.Size(1000, 700)
            MinimizeBox = True
            MaximizeBox = True
            FormBorderStyle = Windows.Forms.FormBorderStyle.Fixed3D
            With Me
                Controls.Add(buttonGo)
                Controls.Add(buttonExit)
                Controls.Add(labelResults)
            End With
            Visible = True
            PerformLayout()
            BringToFront()
        End With
    End Sub

    '////////////////////////////////////////////////////////////////////////////
    '// Controls for MainForm.

    Private WithEvents buttonGo As New Windows.Forms.Button With _
        {.Size = New Drawing.Size(100, 30), _
         .Location = New Drawing.Point(5, 5), _
         .Text = "Execute", .TextAlign = ContentAlignment.MiddleCenter}

    Private WithEvents buttonExit As New Windows.Forms.Button With _
        {.Size = New Drawing.Size(100, 30), _
         .Location = New Drawing.Point(5, 40), _
         .Text = "Exit", .TextAlign = ContentAlignment.MiddleCenter}

    Private labelResults As New Windows.Forms.Label With _
        {.Size = New Drawing.Size(800, 700), _
         .Location = New Drawing.Point(110, 5), _
         .Text = "", .TextAlign = ContentAlignment.TopLeft}

    '////////////////////////////////////////////e/////////////////////////////////////
    '// Handlers for controls for MainForm.

    ' File names
    Private ThisDirectory As String = FileSystem.CurDir.ToString & "\"
```

```
Private FileReader As IO.StreamReader
Private FileWriter As IO.StreamWriter

Private Sub buttonGo_Click() Handles buttonGo.MouseClick

    NF = 16384

    ' First numerical example
    ' Generate some test data for a dual-tone waveform in the time domain
    For I As Int32 = 0 To (NF - 1) Step 1
        DataRe(I) = _
            (5 * Math.Cos(TwoPi * 1024 * I / NF)) + _
            (2 * Math.Sin(TwoPi * 128 * I / NF))
        DataIm(I) = 0
    Next I

    '' Second numerical example
    '' Waveform with non-integral frequency
    'For I As Int32 = 0 To (NF - 1) Step 1
    '    DataRe(I) = 5 * Math.Cos(TwoPi * 6.5 * I / NF)
    '    DataIm(I) = 0
    'Next I

    ' Initialize and then execute the forward transform
    DTFT_Init_Forward()
    DTFT_Forward()

    ' Set exactly to zero the near-zero results of machine imprecision
    For K As Int32 = 0 To (NF - 1) Step 1
        If (Math.Abs(DTFTRe(K)) < Val("1E-8")) Then
            DTFTRe(K) = 0
        End If
        If (Math.Abs(DTFTIm(K)) < Val("1E-8")) Then
            DTFTIm(K) = 0
        End If
    Next K

    ' Divide all DTFTRe() and DTFTIm() by NF
    For I As Int32 = 0 To (NF - 1) Step 1
        DTFTRe(I) = DTFTRe(I) / NF
        DTFTIm(I) = DTFTIm(I) / NF
    Next I

    ' Prepare for display a string with any non-zero frequencies
    Dim DisplayString As String = ""
    For K As Int32 = 0 To (NF - 1) Step 1
        If ((DTFTRe(K) <> 0) Or (DTFTIm(K) <> 0)) Then
            DisplayString = DisplayString & _
                "K=" & Str(K) & ":" & _
                "  DTFTRe(K)=" & Str(DTFTRe(K)) & _
                "  DTFTIm(K)=" & Str(DTFTIm(K)) & vbCrLf
        End If
    Next K
    labelResults.Text = DisplayString

    '' Write spectrum to a text output file in csv format
    'FileWriter = New IO.StreamWriter(ThisDirectory & "DTFT_results.txt")
    'For K As Int32 = 0 To (NF - 1) Step 1
```

```
        '    FileWriter.WriteLine( _
        '        "K=," & Str(K) & "," & _
        '        "DTFTRe(K)=," & Str(DTFTRe(K)) & "," & _
        '        "DTFTIm(K)=," & Str(DTFTIm(K)) & ",")
        'Next K
        'FileWriter.Close()

        'NI = 16384

        '' Third numerical example
        '' Set the frequency domain data directly
        'For I As Int32 = 0 To (NI - 1) Step 1
        '    FreqRe(I) = 0
        '    FreqIm(I) = 0
        'Next I
        'FreqRe(512) = 1.5
        'FreqRe(16384 - 512) = 1.5

        '' Fourth numerical example
        '' Set the frequency domain data directly
        'For I As Int32 = 0 To (NF - 1) Step 1
        '    FreqRe(I) = 0
        '    FreqIm(I) = 0
        'Next I
        'FreqRe(512) = 1.5

        '' Initialize and then execute the inverse transform
        'DTFT_Init_Inverse()
        'DTFT_Inverse()

        '' Write synthesized waveform to a text output file in csv format
        '' Also write the expected time-domain signal
        'FileWriter = New IO.StreamWriter(ThisDirectory & "DTFT_results.txt")
        'For T As Int32 = 0 To (NI - 1) Step 1
        '    DataRe(T) = 3 * Math.Cos(TwoPi * 512 * T / NF))
        '    DataIm(T) = 0
        '    FileWriter.WriteLine( _
        '        "T=," & Str(T) & "," & _
        '        "DataRe(T)=," & Str(DataRe(T)) & "," & _
        '        "DataIm(T)=," & Str(DataIm(T)) & "," & _
        '        "InvDTFTRe(T)=," & Str(InvDTFTRe(T)) & "," & _
        '        "InvDTFTIm(T)=," & Str(InvDTFTIm(T)) & ",")
        'Next T
        'FileWriter.Close()

        MsgBox("All done")
    End Sub

    Private Sub buttonExit_Click() Handles buttonExit.MouseClick
        Application.Exit()
    End Sub

End Class
```

Module FFT_module.vb

```
Option Strict On
Option Explicit On

' Subroutines in this module
'    DTFT_Init_Forward()
'    DTFT_Forward()
'    DTFT_Init_Inverse()
'    DTFT_Inverse()
'    ComplexMult()

Public Module FFT_module

    '////////////////////////////////////////////////////////////////////////////////
    ' Variables for forward transform
    Public NF As Int32 = CInt(2 ^ 14)
    Public NFminus1 As Int32 = NF - 1
    ' Time-domain data points
    Public DataRe(NFminus1) As Double
    Public DataIm(NFminus1) As Double
    ' DTFT transform result
    Public DTFTRe(NFminus1) As Double
    Public DTFTIm(NFminus1) As Double
    ' Constants
    Public HalfNF As Int32 = CInt(NF / 2)
    Public HalfNFminus1 As Int32 = HalfNF - 1
    Public NumBitsF As Int32
    Public TwoPi As Double = 2 * Math.PI
    Public TwoPiOverNF As Double = TwoPi / NF
    ' Supporting vectors
    Public ReverseBitsF(NFminus1) As Int32
    Public TwiddleReF(HalfNFminus1) As Double
    Public TwiddleImF(HalfNFminus1) As Double
    ' Temporary storage vectors
    Private TempTwiddleReF(HalfNFminus1) As Double
    Private TempTwiddleImF(HalfNFminus1) As Double
    Private TempProductReF(HalfNFminus1) As Double
    Private TempProductImF(HalfNFminus1) As Double
    Private TempRe, TempIm As Double

    '////////////////////////////////////////////////////////////////////////////////
    ' Variables for inverse transform
    Public NI As Int32 = CInt(2 ^ 14)
    Public NIminus1 As Int32 = NF - 1
    ' Frequency data points
    Public FreqRe(NIminus1) As Double
    Public FreqIm(NIminus1) As Double
    ' Inverse DTFT transform result
    Public InvDTFTRe(NIminus1) As Double
    Public InvDTFTIm(NIminus1) As Double
    ' Constants
    Public HalfNI As Int32 = CInt(NI / 2)
    Public HalfNIminus1 As Int32 = HalfNI - 1
    Public NumBitsI As Int32
    Public TwoPiOverNI As Double = TwoPi / NI
    ' Supporting vectors
    Public ReverseBitsI(NIminus1) As Int32
```

```
Public TwiddleReI(HalfNIminus1) As Double
Public TwiddleImI(HalfNIminus1) As Double
' Temporary storage vectors
Private TempTwiddleReI(HalfNIminus1) As Double
Private TempTwiddleImI(HalfNIminus1) As Double
Private TempProductReI(HalfNIminus1) As Double
Private TempProductImI(HalfNIminus1) As Double

'////////////////////////////////////////////////////////////////////////////////
' Subroutine DTFT_Init_Forward must be called once before any forward transform
' processing is done using DTFT_Forward().
Public Sub DTFT_Init_Forward()
    ' Step #1: Ensure that NF is a power of two, and calculate NumBitsF
    Dim TempInt As Int32 = 1
    NumBitsF = -1
    For I As Int32 = 1 To 31 Step 1
        TempInt = 2 * TempInt
        If (TempInt = NF) Then
            NumBitsF = I
            Exit For
        End If
    Next I
    If (NumBitsF < 0) Then
        MsgBox("Error: NF is not a power of two.")
        Application.Exit()
    End If
    ' Step #2: Populate the ReverseBitsF() vector
    Dim IndexBitRegister As Int32
    Dim ReverseBitRegister As Int32
    For I As Int32 = 0 To NFminus1 Step 1
        IndexBitRegister = I
        ReverseBitRegister = 0
        For J As Int32 = 0 To (NumBitsF - 1) Step 1
            ' Shift the ReverseBitRegister one space to the left
            ReverseBitRegister = 2 * ReverseBitRegister
            ' Inspect the least significant bit of IndexBitRegister
            If ((IndexBitRegister And 1) <> 0) Then
                ' The LSB of IndexBitRegister is one, so set the
                ' LSB of ReverseBitRegister
                ReverseBitRegister = ReverseBitRegister Or 1
                ' There is no Else.  If the LSB of IndexBitRegister is zero,
                ' then the LSB of ReverseBitRegister should be cleared.  But
                ' nothing needs to be done since ReverseBitRegister was
                ' initialized with all of its bit cleared to zero.
            End If
            ' Shift the IndexBitRegister one space to the right so the new LSB
            ' can be examined during the next iteration.  The backslash is
            ' used for integer division.
            IndexBitRegister = IndexBitRegister \ 2
        Next J
        ReverseBitsF(I) = ReverseBitRegister
    Next I
    ' Step #3: Populate the twiddle factors
    Dim TempArgument As Double
    For I As Int32 = 0 To HalfNFminus1 Step 1
        TempArgument = I * TwoPiOverNF
        TwiddleReF(I) = Math.Cos(TempArgument)
        TwiddleImF(I) = -Math.Sin(TempArgument)
```

```
    Next I
End Sub

'////////////////////////////////////////////////////////////////////////////
Public Sub DTFT_Forward()
    ' Stage #1: Move the input data DataRe() and DataIm() into the working vectors
    ' in bit-reversed order.  While doing this, implement the stage #1 changes of
    ' sign.  It is easiest to do this in groups, or strides, of 2.  After having
    ' re-ordered the data points in this stage using the bit-reversal selection
    ' technique, there will be no further re-ordering in any of the following stages.
    For I As Int32 = 0 To (NFminus1 - 1) Step 2
        DTFTRe(I) = DataRe(ReverseBitsF(I)) + DataRe(ReverseBitsF(I + 1))
        DTFTIm(I) = DataIm(ReverseBitsF(I)) + DataIm(ReverseBitsF(I + 1))
        DTFTRe(I + 1) = DataRe(ReverseBitsF(I)) - DataRe(ReverseBitsF(I + 1))
        DTFTIm(I + 1) = DataIm(ReverseBitsF(I)) - DataIm(ReverseBitsF(I + 1))
    Next I
    ' Stage #2: Let's do the Stage #2 processing explicitly.  It is easiest to do
    ' this in groups, or strides, of 4.
    ' Step #1: Figure out exactly which twiddle factors are needed.  Two factors are
    ' needed and they are stored in TempTwiddleReF(1) and TempTwiddleImF(1),
    ' zero-based.  Since the same twiddle factors are used for all of these blocks
    ' of four, we can figure them out before starting to process the blocks.
    TempTwiddleReF(0) = 1
    TempTwiddleImF(0) = 0
    TempTwiddleReF(1) = TwiddleReF(NF \ 4)
    TempTwiddleImF(1) = TwiddleImF(NF \ 4)
    For I As Int32 = 0 To (NFminus1 - 3) Step 4
        ' Step #2: Multiply the upper half of the points by their twiddle factors.
        ' The results could be stored in their corresponding DTFTRe() and DTFTIm()
        ' locations.  However, to avoid overwriting DTFTRe(2) and DTFTIm(2) as the
        ' multiplications are being carried out, I will store the products in
        ' temporary vectors TempProductReF(1) and TempProductImF(1), zero-based.
        ComplexMult( _
            DTFTRe(I + 2), DTFTIm(I + 2), _
            TempTwiddleReF(0), TempTwiddleImF(0), _
            TempProductReF(0), TempProductImF(0))
        ComplexMult( _
            DTFTRe(I + 3), DTFTIm(I + 3), _
            TempTwiddleReF(1), TempTwiddleImF(1), _
            TempProductReF(1), TempProductImF(1))
        ' Step #3: Add and subtract as necessary to calculate the four results.
        ' Since the results overwrite terms in the DTFT() vectors, we need a
        ' temporary complex variable to hold interim results before they are
        ' overwritten.
        TempRe = DTFTRe(I) + TempProductReF(0)
        TempIm = DTFTIm(I) + TempProductImF(0)
        DTFTRe(I + 2) = DTFTRe(I) - TempProductReF(0)
        DTFTIm(I + 2) = DTFTIm(I) - TempProductImF(0)
        DTFTRe(I) = TempRe
        DTFTIm(I) = TempIm
        TempRe = DTFTRe(I + 1) + TempProductReF(1)
        TempIm = DTFTIm(I + 1) + TempProductImF(1)
        DTFTRe(I + 3) = DTFTRe(I + 1) - TempProductReF(1)
        DTFTIm(I + 3) = DTFTIm(I + 1) - TempProductImF(1)
        DTFTRe(I + 1) = TempRe
        DTFTIm(I + 1) = TempIm
    Next I
    ' Stage #3: Let's also do the Stage #3 processing explicitly.  This is done
```

```
' in groups, or strides, of 8.
' Step #1: Figure out exactly which twiddle factors are needed.  Four factors
' are needed and they are stored in TempTwiddleReF(3) and TempTwiddleImF(3),
' zero-based.  Since the same twiddle factors are used for all of these blocks,
' we can figure them out before starting into the stride.
TempTwiddleReF(0) = 1
TempTwiddleImF(0) = 0
TempTwiddleReF(1) = TwiddleReF(NF \ 8)
TempTwiddleImF(1) = TwiddleImF(NF \ 8)
TempTwiddleReF(2) = TwiddleReF(2 * NF \ 8)
TempTwiddleImF(2) = TwiddleImF(2 * NF \ 8)
TempTwiddleReF(3) = TwiddleReF(3 * NF \ 8)
TempTwiddleImF(3) = TwiddleImF(3 * NF \ 8)
For I As Int32 = 0 To (NFminus1 - 7) Step 8
    ' Step #2: Multiply the upper half of the points by their twiddle factors.
    ' The results could be stored in their corresponding DTFTRe() and DTFTIm()
    ' locations.  However, to avoid overwriting DTFTRe(4), DTFTIm(4), DTFTRe(6)
    ' and DTFTIm(6) as the multiplications are being carried out, I will store
    ' the products in temporary vectors TempProductReF(3) and TempProductImF(3),
    ' zero-based.
    ComplexMult( _
        DTFTRe(I + 4), DTFTIm(I + 4), _
        TempTwiddleReF(0), TempTwiddleImF(0), _
        TempProductReF(0), TempProductImF(0))
    ComplexMult( _
        DTFTRe(I + 5), DTFTIm(I + 5), _
        TempTwiddleReF(1), TempTwiddleImF(1), _
        TempProductReF(1), TempProductImF(1))
    ComplexMult( _
        DTFTRe(I + 6), DTFTIm(I + 6), _
        TempTwiddleReF(2), TempTwiddleImF(2), _
        TempProductReF(2), TempProductImF(2))
    ComplexMult( _
        DTFTRe(I + 7), DTFTIm(I + 7), _
        TempTwiddleReF(3), TempTwiddleImF(3), _
        TempProductReF(3), TempProductImF(3))
    ' Step #3: Add and subtract as necessary to calculate the eight results.
    ' Since the results overwrite terms in the DTFT() vectors, we need a
    ' temporary complex variable to hold interim results before they are
    ' overwritten.
    TempRe = DTFTRe(I) + TempProductReF(0)
    TempIm = DTFTIm(I) + TempProductImF(0)
    DTFTRe(I + 4) = DTFTRe(I) - TempProductReF(0)
    DTFTIm(I + 4) = DTFTIm(I) - TempProductImF(0)
    DTFTRe(I) = TempRe
    DTFTIm(I) = TempIm
    TempRe = DTFTRe(I + 1) + TempProductReF(1)
    TempIm = DTFTIm(I + 1) + TempProductImF(1)
    DTFTRe(I + 5) = DTFTRe(I + 1) - TempProductReF(1)
    DTFTIm(I + 5) = DTFTIm(I + 1) - TempProductImF(1)
    DTFTRe(I + 1) = TempRe
    DTFTIm(I + 1) = TempIm
    TempRe = DTFTRe(I + 2) + TempProductReF(2)
    TempIm = DTFTIm(I + 2) + TempProductImF(2)
    DTFTRe(I + 6) = DTFTRe(I + 2) - TempProductReF(2)
    DTFTIm(I + 6) = DTFTIm(I + 2) - TempProductImF(2)
    DTFTRe(I + 2) = TempRe
    DTFTIm(I + 2) = TempIm
```

```
            TempRe = DTFTRe(I + 3) + TempProductReF(3)
            TempIm = DTFTIm(I + 3) + TempProductImF(3)
            DTFTRe(I + 7) = DTFTRe(I + 3) - TempProductReF(3)
            DTFTIm(I + 7) = DTFTIm(I + 3) - TempProductImF(3)
            DTFTRe(I + 3) = TempRe
            DTFTIm(I + 3) = TempIm
        Next I
        ' Stages #4+: Now that the pattern for processing has been pretty well
        ' established, I will use a loop to run through all the remaining stages.
        For Istage As Int32 = 4 To NumBitsF Step 1
            Dim Stride As Int32 = CInt(2 ^ Istage)
            Dim HalfStride As Int32 = Stride \ 2
            Dim HalfStrideMinus1 As Int32 = HalfStride - 1
            ' Step #1: Figure out exactly which twiddle factors are needed.  The
            ' number of factors needed is Stride/2 and the factors are stored in
            ' TempTwiddleReF(Stride/2 - 1) and TempTwiddleImF(Stride/2 - 1), zero-based.
            For J As Int32 = 0 To HalfStrideMinus1 Step 1
                ' Carry out the division before the multipliction to ensure there
                ' is not an arithmetic overlow.
                TempTwiddleReF(J) = TwiddleReF(J * (NF \ Stride))
                TempTwiddleImF(J) = TwiddleImF(J * (NF \ Stride))
            Next J
            For I As Int32 = 0 To (NFminus1 + 1 - Stride) Step Stride
                ' Step #2: Multiply the upper half of the points by their twiddle
                ' factors.  The results could be stored in their corresponding DTFTRe()
                ' and DTFTIm() locations.  However, to avoid overwriting some of the
                ' interim results as the multiplications are being carried out, I will
                ' store the products in temporary vectors TempProductReF(Stride/2 - 1)
                ' and TempProductImF(Stride/2 - 1), zero-based.
                For J As Int32 = 0 To HalfStrideMinus1 Step 1
                    ComplexMult( _
                        DTFTRe(I + J + HalfStride), DTFTIm(I + J + HalfStride), _
                        TempTwiddleReF(J), TempTwiddleImF(J), _
                        TempProductReF(J), TempProductImF(J))
                Next J
                ' Step #3: Add and subtract as necessary to calculate the results.
                ' Since the results overwrite terms in the DTFT() vectors, we need a
                ' temporary complex variable to hold interim results before they are
                ' overwritten.  These will be processed in groups of two.
                For J As Int32 = 0 To HalfStrideMinus1 Step 1
                    TempRe = DTFTRe(I + J) + TempProductReF(J)
                    TempIm = DTFTIm(I + J) + TempProductImF(J)
                    DTFTRe(I + J + HalfStride) = DTFTRe(I + J) - TempProductReF(J)
                    DTFTIm(I + J + HalfStride) = DTFTIm(I + J) - TempProductImF(J)
                    DTFTRe(I + J) = TempRe
                    DTFTIm(I + J) = TempIm
                Next J
            Next I
        Next Istage
    End Sub

    '////////////////////////////////////////////////////////////////////////////////
    ' Subroutine DTFT_Init_Inverse must be called once before any inverse transform
    ' processing is done using DTFT_Inverse().
    Public Sub DTFT_Init_Inverse()
        ' Step #1: Ensure that NI is a power of two, and calculate NumBitsI
        Dim TempInt As Int32 = 1
        NumBitsI = -1
```

```
    For I As Int32 = 1 To 31 Step 1
        TempInt = 2 * TempInt
        If (TempInt = NI) Then
            NumBitsI = I
            Exit For
        End If
    Next I
    If (NumBitsI < 0) Then
        MsgBox("Error: NI is not a power of two.")
        Application.Exit()
    End If
    ' Step #2: Populate the ReverseBitsI() vector
    Dim IndexBitRegister As Int32
    Dim ReverseBitRegister As Int32
    For I As Int32 = 0 To NIminus1 Step 1
        IndexBitRegister = I
        ReverseBitRegister = 0
        For J As Int32 = 0 To (NumBitsI - 1) Step 1
            ' Shift the ReverseBitRegister one space to the left
            ReverseBitRegister = 2 * ReverseBitRegister
            ' Inspect the least significant bit of IndexBitRegister
            If ((IndexBitRegister And 1) <> 0) Then
                ' The LSB of IndexBitRegister is one, so set the
                ' LSB of ReverseBitRegister
                ReverseBitRegister = ReverseBitRegister Or 1
                ' There is no Else.  If the LSB of IndexBitRegister is zero,
                ' then the LSB of ReverseBitRegister should be cleared.  But
                ' nothing needs to be done since ReverseBitRegister was
                ' initialized with all of its bit cleared to zero.
            End If
            ' Shift the IndexBitRegister one space to the right so the new LSB
            ' can be examined during the next iteration.  The backslash is
            ' used for integer division.
            IndexBitRegister = IndexBitRegister \ 2
        Next J
        ReverseBitsI(I) = ReverseBitRegister
    Next I
    ' Step #3: Populate the twiddle factors
    Dim TempArgument As Double
    For I As Int32 = 0 To HalfNIminus1 Step 1
        TempArgument = I * TwoPiOverNI
        TwiddleReI(I) = Math.Cos(TempArgument)
        TwiddleImI(I) = Math.Sin(TempArgument)
    Next I
End Sub

'////////////////////////////////////////////////////////////////////////////
Public Sub DTFT_Inverse()
    ' Stage #1: Move the input data FreqRe() and FreqIm() into the working vectors
    ' in bit-reversed order.  While doing this, implement the stage #1 changes of
    ' sign.  It is easiest to do this in groups, or strides, of 2.  After having
    ' re-ordered the data points in this stage using the bit-reversal selection
    ' technique, there will be no further re-ordering in any of the following stages.
    For I As Int32 = 0 To (NIminus1 - 1) Step 2
        InvDTFTRe(I) = FreqRe(ReverseBitsI(I)) + FreqRe(ReverseBitsI(I + 1))
        InvDTFTIm(I) = FreqIm(ReverseBitsI(I)) + FreqIm(ReverseBitsI(I + 1))
        InvDTFTRe(I + 1) = FreqRe(ReverseBitsI(I)) - FreqRe(ReverseBitsI(I + 1))
        InvDTFTIm(I + 1) = FreqIm(ReverseBitsI(I)) - FreqIm(ReverseBitsI(I + 1))
```

```
Next I
' Stage #2: Let's do the Stage #2 processing explicitly.  It is easiest to do
' this in groups, or strides, of 4.
' Step #1: Figure out exactly which twiddle factors are needed.  Two factors are
' needed and they are stored in TempTwiddleReI(1) and TempTwiddleImI(1),
' zero-based.  Since the same twiddle factors are used for all of these blocks
' of four, we can figure them out before starting to process the blocks.
TempTwiddleReI(0) = 1
TempTwiddleImI(0) = 0
TempTwiddleReI(1) = TwiddleReI(NI \ 4)
TempTwiddleImI(1) = TwiddleImI(NI \ 4)
For I As Int32 = 0 To (NIminus1 - 3) Step 4
    ' Step #2: Multiply the upper half of the points by their twiddle factors.
    ' The results could be stored in their corresponding InvDTFTRe() and
    ' InvDTFTIm() locations.  However, to avoid overwriting InvDTFTRe(2) and
    ' InvDTFTIm(2) as the multiplications are being carried out, I will store
    ' the products in temporary vectors TempProductReI(1) and TempProductImI(1),
    ' zero-based.
    ComplexMult( _
        InvDTFTRe(I + 2), InvDTFTIm(I + 2), _
        TempTwiddleReI(0), TempTwiddleImI(0), _
        TempProductReI(0), TempProductImI(0))
    ComplexMult( _
        InvDTFTRe(I + 3), InvDTFTIm(I + 3), _
        TempTwiddleReI(1), TempTwiddleImI(1), _
        TempProductReI(1), TempProductImI(1))
    ' Step #3: Add and subtract as necessary to calculate the four results.
    ' Since the results overwrite terms in the InvDTFT() vectors, we need a
    ' temporary complex variable to hold interim results before they are
    ' overwritten.
    TempRe = InvDTFTRe(I) + TempProductReI(0)
    TempIm = InvDTFTIm(I) + TempProductImI(0)
    InvDTFTRe(I + 2) = InvDTFTRe(I) - TempProductReI(0)
    InvDTFTIm(I + 2) = InvDTFTIm(I) - TempProductImI(0)
    InvDTFTRe(I) = TempRe
    InvDTFTIm(I) = TempIm
    TempRe = InvDTFTRe(I + 1) + TempProductReI(1)
    TempIm = InvDTFTIm(I + 1) + TempProductImI(1)
    InvDTFTRe(I + 3) = InvDTFTRe(I + 1) - TempProductReI(1)
    InvDTFTIm(I + 3) = InvDTFTIm(I + 1) - TempProductImI(1)
    InvDTFTRe(I + 1) = TempRe
    InvDTFTIm(I + 1) = TempIm
Next I
' Stage #3: Let's also do the Stage #3 processing explicitly.  This is done
' in groups, or strides, of 8.
' Step #1: Figure out exactly which twiddle factors are needed.  Four factors
' are needed and they are stored in TempTwiddleReI(3) and TempTwiddleImI(3),
' zero-based.  Since the same twiddle factors are used for all of these blocks,
' we can figure them out before starting into the stride.
TempTwiddleReI(0) = 1
TempTwiddleImI(0) = 0
TempTwiddleReI(1) = TwiddleReI(NI \ 8)
TempTwiddleImI(1) = TwiddleImI(NI \ 8)
TempTwiddleReI(2) = TwiddleReI(2 * NI \ 8)
TempTwiddleImI(2) = TwiddleImI(2 * NI \ 8)
TempTwiddleReI(3) = TwiddleReI(3 * NI \ 8)
TempTwiddleImI(3) = TwiddleImI(3 * NI \ 8)
For I As Int32 = 0 To (NIminus1 - 7) Step 8
```

```
        ' Step #2: Multiply the upper half of the points by their twiddle factors.
        ' The results could be stored in their corresponding InvDTFTRe() and
        ' InvDTFTIm() locations.  However, to avoid overwriting InvDTFTRe(4),
        ' InvDTFTIm(4), InvDTFTRe(6) and InvDTFTIm(6) as the multiplications are
        ' being carried out, I will store the products in temporary vectors
        ' TempProductReI(3) and TempProductImI(3), zero-based.
        ComplexMult( _
            InvDTFTRe(I + 4), InvDTFTIm(I + 4), _
            TempTwiddleReI(0), TempTwiddleImI(0), _
            TempProductReI(0), TempProductImI(0))
        ComplexMult( _
            InvDTFTRe(I + 5), InvDTFTIm(I + 5), _
            TempTwiddleReI(1), TempTwiddleImI(1), _
            TempProductReI(1), TempProductImI(1))
        ComplexMult( _
            InvDTFTRe(I + 6), InvDTFTIm(I + 6), _
            TempTwiddleReI(2), TempTwiddleImI(2), _
            TempProductReI(2), TempProductImI(2))
        ComplexMult( _
            InvDTFTRe(I + 7), InvDTFTIm(I + 7), _
            TempTwiddleReI(3), TempTwiddleImI(3), _
            TempProductReI(3), TempProductImI(3))
        ' Step #3: Add and subtract as necessary to calculate the eight results.
        ' Since the results overwrite terms in the InvDTFT() vectors, we need a
        ' temporary complex variable to hold interim results before they are
        ' overwritten.
        TempRe = InvDTFTRe(I) + TempProductReI(0)
        TempIm = InvDTFTIm(I) + TempProductImI(0)
        InvDTFTRe(I + 4) = InvDTFTRe(I) - TempProductReI(0)
        InvDTFTIm(I + 4) = InvDTFTIm(I) - TempProductImI(0)
        InvDTFTRe(I) = TempRe
        InvDTFTIm(I) = TempIm
        TempRe = InvDTFTRe(I + 1) + TempProductReI(1)
        TempIm = InvDTFTIm(I + 1) + TempProductImI(1)
        InvDTFTRe(I + 5) = InvDTFTRe(I + 1) - TempProductReI(1)
        InvDTFTIm(I + 5) = InvDTFTIm(I + 1) - TempProductImI(1)
        InvDTFTRe(I + 1) = TempRe
        InvDTFTIm(I + 1) = TempIm
        TempRe = InvDTFTRe(I + 2) + TempProductReI(2)
        TempIm = InvDTFTIm(I + 2) + TempProductImI(2)
        InvDTFTRe(I + 6) = InvDTFTRe(I + 2) - TempProductReI(2)
        InvDTFTIm(I + 6) = InvDTFTIm(I + 2) - TempProductImI(2)
        InvDTFTRe(I + 2) = TempRe
        InvDTFTIm(I + 2) = TempIm
        TempRe = InvDTFTRe(I + 3) + TempProductReI(3)
        TempIm = InvDTFTIm(I + 3) + TempProductImI(3)
        InvDTFTRe(I + 7) = InvDTFTRe(I + 3) - TempProductReI(3)
        InvDTFTIm(I + 7) = InvDTFTIm(I + 3) - TempProductImI(3)
        InvDTFTRe(I + 3) = TempRe
        InvDTFTIm(I + 3) = TempIm
    Next I
    ' Stages #4+: Now that the pattern for processing has been pretty well
    ' established, I will use a loop to run through all the remaining stages.
    For Istage As Int32 = 4 To NumBitsI Step 1
        Dim Stride As Int32 = CInt(2 ^ Istage)
        Dim HalfStride As Int32 = Stride \ 2
        Dim HalfStrideMinus1 As Int32 = HalfStride - 1
        ' Step #1: Figure out exactly which twiddle factors are needed.  The
```

```
            ' number of factors needed is Stride/2 and the factors are stored in
            ' TempTwiddleReI(Stride/2 - 1) and TempTwiddleImI(Stride/2 - 1), zero-based.
            For J As Int32 = 0 To HalfStrideMinus1 Step 1
                ' Carry out the division before the multipliction to ensure there
                ' is not an arithmetic overlow.
                TempTwiddleReI(J) = TwiddleReI(J * (NI \ Stride))
                TempTwiddleImI(J) = TwiddleImI(J * (NI \ Stride))
            Next J
            For I As Int32 = 0 To (NIminus1 + 1 - Stride) Step Stride
                ' Step #2: Multiply the upper half of the points by their twiddle
                ' factors.  The results could be stored in their corresponding
                ' InvDTFTRe() and InvDTFTIm() locations.  However, to avoid overwriting
                ' some of the interim results as the multiplications are being carried
                ' out, I will store the products in temporary vectors
                ' TempProductReI(Stride / 2 - 1) and TempProductImI(Stride/2 - 1),
                ' zero-based.
                For J As Int32 = 0 To HalfStrideMinus1 Step 1
                    ComplexMult( _
                        InvDTFTRe(I + J + HalfStride), InvDTFTIm(I + J + HalfStride), _
                        TempTwiddleReI(J), TempTwiddleImI(J), _
                        TempProductReI(J), TempProductImI(J))
                Next J
                ' Step #3: Add and subtract as necessary to calculate the results.
                ' Since the results overwrite terms in the InvDTFT() vectors, we need a
                ' temporary complex variable to hold interim results before they are
                ' overwritten.  These will be processed in groups of two.
                For J As Int32 = 0 To HalfStrideMinus1 Step 1
                    TempRe = InvDTFTRe(I + J) + TempProductReI(J)
                    TempIm = InvDTFTIm(I + J) + TempProductImI(J)
                    InvDTFTRe(I + J + HalfStride) = InvDTFTRe(I + J) - TempProductReI(J)
                    InvDTFTIm(I + J + HalfStride) = InvDTFTIm(I + J) - TempProductImI(J)
                    InvDTFTRe(I + J) = TempRe
                    InvDTFTIm(I + J) = TempIm
                Next J
            Next I
        Next Istage
    End Sub

    '////////////////////////////////////////////////////////////////////////////
    Public Sub ComplexMult( _
        ByVal Re1 As Double, ByVal Im1 As Double, _
        ByVal Re2 As Double, ByVal Im2 As Double, _
        ByRef ProductRe As Double, ByRef ProductIm As Double)
        ProductRe = (Re1 * Re2) - (Im1 * Im2)
        ProductIm = (Re1 * Im2) + (Im1 * Re2)
    End Sub

End Module
```