A Serious Introduction to the Fast Fourier Transform (FFT)

Eric Gelphman

University of California Irvine Department of Mathematics

March 2021

In [23]:
```
import numpy as np
PI = np.pi
```

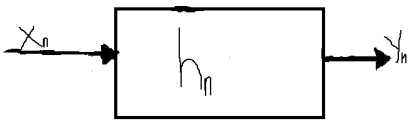
Section I: Intro to Discrete Time Signals and Systems, and Bits of Linear Algebra

Section I-1: Signals, Filters, and Convolution in Discrete Time

The FFT is quite possibly THE fundamental algorithm of digital signal processing(DSP). For this presentation, we will assume that the signals we will be dealing with are discrete time signals, i.e. ones that occur at discrete(and also finite in the case of the DFT/FFT) time intervals. Going forward, we need to know the following definitions:

Signal(Discrete-Time): A discrete time signal is a complex-valued sequence $\{x_n\}_{n\in\mathbb{N}}$ that represents some real-world quantity after it has been sampled(Section II-4). n is the time variable, which specifies the sample number.

Filter: A filter is anything that manipulates an incoming signal based on the frequency of the signal. A linear filter is described mathematically as a function of time by the sequence $\{h_n\}_{n\in\mathbb{N}}$. If $\{h_n\}$ is finite, then the filter is called a finite impulse response(FIR) filter, and the filter can be represented mathematically as a matrix. If $\{h_n\}$ is infinite, then the filter is called a ininfite impulse response(IIR) filter.



Convolution: Assume that the filter $\{h_n\}$ can be represented by some linear operator(i.e. matrix or polynomial coefficients). Then, the output $\{y_n\}$ of the filter is defined by the convolution of $\{h_n\}$ with the input $\{x_n\}$, which is denoted by $y_n = h_n * x_n$ and given by

$$y_n = \sum_{k=-\infty}^{+\infty} x_k h_{n-k}$$

This can be thought of as the result of "blending" $\{x_n\}$ and $\{h_n\}$. The indexing n - k must be done modulo N in the case of finite-length $\{x_n\}$ and $\{h_n\}$
Note that this is the same as the definition of the Cauchy Product of two power series: Let $A(z) = \sum_{n=0}^{\infty} a_n z^n, B(z) = \sum_{n=0}^{\infty} b_n z^n$. Then, C(z) = A(z)B(z) is given by

$$C(z) = \sum_{n=0}^{\infty} c_n z^n \text{ where } c_n = \sum_{k=0}^{n} a_k b_{n-k}$$

If $A(z), B(z)$ are finite sums, then the above product, described by the operation of convolution, is the product of two polynomials in z.
Computing the output sequence $\{y_n\}_{n=0}^{N-1}$ is clearly has a time complexity of $O(N^2)$. Assuming that $\{h_n\}$ is FIR, it is only necessary to store the filter coefficients $\{h_n\}_{n=0}^{K}$ where K does not necessarily equal N. This is computationally expensive for large N. In fact, if real digital signals were processed this way, LTE technology(among many other things, including Praveen's horseback riding radios) would not be possible. Thus, if we are to process large amounts of data quickly(and have Praveen's horseback riding radios, among many other things), we will need a better algorithm to compute $\{y_n\}$. To do this, we will, of course, first need to review some maths.

Section I-2: Enter Linear Algebra

If we are instead working with finite-length signals, the convergence of sequences and series is guaranteed and much of the math can be described in the language of matrices and vectors.
$\mathbf{Definition (Finite\text{-}Dimensional\ Hilbert\ Space):}$ A finite-dimensional Hilbert space H is a finite-dimensional vector space that is equipped with an inner product operation $< \cdot, \cdot >$ that defines an associated norm $\|\cdot\|$ on H as follows: For $X \in H, \|X\| = \sqrt{< X, X >}$. Moreover, the inner product $< \cdot, \cdot >$ must satisfy the following properties: Let $X, Y, Z \in H$ and let α, β be scalars

$$1.(\text{Symmetry}): < X, Y >=< Y, X > \quad (1)$$
$$2.(\text{Positive-Definiteness}): \|X\| = 0 \iff X \equiv 0 \text{ and } < X, X >> 0 \text{ if } X \neq 0 \quad (2)$$
$$3.(\text{Linearity in Both Variables}): < \alpha X + \beta Y, Z >= \alpha < X, Z > + \beta < Y, Z > \quad (3)$$

The finite-dimensional Hilbert space we are most concerned about in this presentation is $\mathbb{C}^n$ where n is a positive integer. This is the space of n-dimensiuonal vectors with complex entries, or the space of finite-length complex signals. The inner product of $z, w \in \mathbb{C}^n$ is given by

$$< z, w >= z_1\overline{w_1} + \ldots + z_n\overline{w_n}$$

where $\overline{z}$ denotes complex conjugation.
$\mathbf{Definition (Orthogonal/Orthonormal\ Set\ of\ Vectors):}$ Let $\{v_1, \ldots, v_n\}$ be a set of

vectors in a finite-dimensional Hilbert space H. If $< v_i, v_j >= 0 \text{ whenever } i \neq j$, then $\{v_1, \ldots, v_n\}$ is an orthogonal set of vectors in H. If additionally the $v_i's$ are linearly independent, then $\{v_1, \ldots, v_n\}$ is an orthogonal basis for H. Moreover, if all of the $v_i's$ are of unit magnitude, then $\{v_1, \ldots, v_n\}$ is an orthonormal basis for H.

$\mathbf{Theorem(Gram\text{-}Schmidt\ Process)}$: Let H be a nontrivial finite-dimensional Hilbert space. Then, H is guaranteed to have an orthogonal basis. Moreover, given an arbitrary basis $\{v_1, \ldots, v_n\}$ of H, an orthogonal basis $\{u_1, \ldots, u_n\}$ can be obtained from $\{v_1, \ldots, v_n\}$ using

$$u_1 = v_1 \tag{4}$$
$$u_2 = v_2 - \frac{< v_2, u_1 >}{< u_1, u_1 >} u_1 \tag{5}$$
$$\vdots \tag{6}$$
$$u_n = v_n - \frac{< v_n, u_{n-1} >}{< u_{n-1}, u_{n-1} >} u_{n-1} - \ldots - \frac{< v_n, u_1 >}{< u_1, u_1 >} u_1 \tag{7}$$

The Gram-Schmidt Process can be used to define a concept of "best approximation" in the Hilbert space H with respect to the norm $\|\cdot\|$.

$\mathbf{Theorem(Best\ Approximation)}$: Let H be a finite dimensional Hilbert space with inner product $< \cdot, \cdot >$ and associated norm $\|\cdot\|$. Let $\{u_1, \ldots, u_n\}$ be an orthonormal basis of H and let $x \in H$. Let

$$p = \sum_{j=1}^{n} \gamma_j u_j \text{ where } \gamma_j =< x, u_j >$$

Then, p is the element of H that best approximates x with respect to $\|\cdot\|$. In other words, for any $y \in H$,

$$\|x - p\| < \|y - x\|$$

Section II: The Discrete Fourier Transform(DFT)

Section II-1: The Discrete Time Fourier Transform(DTFT) and Hints of Fourier Analysis

Remember that n is a time variable. Therefore, explicitly computing the convolution is a time domain operation. What if we were to transform $\{h_n\}$ and $\{x_n\}$ into functions of a frequency variable, compute the convolution $x_n * h_n$ in the frequency domain, and then trasnform the result back into the time domain to obtain $\{y_n\}$? The operation that does this is called the Fourier transform. The Fourier transform is part of a fascinating area of mathematics called harmonic analysis. There exists a Fourier transform in both continuous and discrete time. We will focus on the discrete time Fourier transform(DTFT) and its finite version, called the discrete Fourier transform(DFT). The fast Fourier transform(FFT) is a class of efficient aglorithms for computing the DFT. Before we give a formula for the DFT is, we need to define the discrete-time Fourier transform(DTFT) and how the DFT is obtained from it.

$\mathbf{Definition(DTFT)}$: Let $\{x_n\}_{n\in\mathbb{N}}$ be a sequence of complex numbers. The discrete time Fourier transform(DTFT) of $\{x_n\}$, denoted $X(e^{i\omega})$, is, assuming the series converges, given by

$$X(e^{i\omega}) = \sum_{n=-\infty}^{+\infty} x_n e^{-i\omega n}$$

where ω is a real variable defined on any closed interval of length 2π, often chosen as $[-\pi, \pi]$, which we will also do for the rest of this discussion. The $e^{i\omega}$ is used in the notation to denote that $X(e^{i\omega})$ is defined in the discrete-time frequency domain, or frequencies restricted to the interval $[-\pi, \pi]$. This is different from the continous-time frequency domain, which is all frequencies ω defined on the real line. This distinction will be explained further in Section $II-4$.

$\mathbf{Remark}$: The DTFT definition above is commonly given in most electrical engineering texts such as Oppenheim and Wilsky's $\mathit{Signals\ and\ Systems}$. The above definition is actually the definition of the Fourier series representation of a Riemann-integrable function given in most math textbooks, such as Walter Rudin's $\mathit{Principles\ of\ Mathematical\ Analysis}$ and Elias Stein's and Rami Shakarchi's $\mathit{Fourier\ Analysis}$.

Remark: The sequence of functions $\{e^{i\omega n}\}_{n\in\mathbb{Z}}$ forms an $\mathrm{infinite\ dimensional}$ basis for the $\mathit{infinite\ dimensional\ Hilbert\ Space}$ of all complex-valued square integrable functions on the interval $[-\pi, \pi]$, i.e. all functions $f : [-\pi, \pi] \to \mathbb{C}$ that satisfy

$$\int_{-\pi}^{\pi} |f(x)|^2 dx < \infty$$

or, in other words, the space of real signals that have $\mathit{finite\ energy}$. The functional basis $\{e^{i\omega n}\}_{n\in\mathbb{Z}}$ originates from the solution of the eigenvalue problem corresponding to the separation of variables of the $\mathit{heat\ equation}$, a partial differential equation(PDE) that discribes heat conduction, originally studied by the legendary French mathematician Joseph Fourier in the 1820s. One can also obtain this functional basis by solving the eigenvalue problem corresponding to another PDE called the $\mathit{wave\ equation}$, which makes sense in the context of describing things as a function of frequency.

Now, all of this theory is great, but what does the DTFT do, exactly? The DTFT can be seen as a way of expressing $\{x_n\}$ as a weighted sum of its harmonics. In other words, the DTFT describes how the total energy of $\{x_n\}$ is distributed amongs the (infinite) individual terms of the sequence. Assuming that it converges, $X(e^{i\omega})$ will be a continous function of ω, as the functions $e^{-in\omega}$ are themselves continuous functions of ω. Therefore, the DTFT can be seen as represernting how the energy of the signal $\{x_n\}$ is distributed across the range of frequencies $\omega \in [-\pi, \pi]$. A more precise mathematical description of this last statement is given by $\mathit{Parseval's\ Theorem}$.

$$\sum_{n=0}^{\infty} |x_n|^2 = \frac{1}{2\pi} \int_{-\pi}^{\pi} |X(e^{i\omega})|^2 d\omega$$

The most important property of the DTFT pertianing to our discussion is that it transforms the operation of convolution into multiplication. Before this, we must introduct another important property of the DTFT: If the sequence h_n is shifted by k, to generate h_{n-k}, then this is the same as multiplying its DTFT $H(e^{i\omega})$ by a factor of $e^{-i\omega k}$. This can be verified by direct alegbraic manipulation of the DTFT formula. Let $Y(e^{i\omega})$ denote the DTFT of the result of the convolution $x_n * h_n$:

$$Y(e^{i\omega}) = \sum_{n=-\infty}^{\infty} (\sum_{k=-\infty}^{\infty} x_k h_{n-k}) e^{-i\omega n} \tag{8}$$

$$Y(e^{i\omega}) = \sum_{n=-\infty}^{\infty} (\sum_{k=-\infty}^{\infty} x_k h_n(e^{-i\omega k})) e^{-i\omega n} \tag{9}$$

Let us assume for a moment that the convergence of the double infinite series above is such that the interchange of the infinite sums does not affect the overall result. This verification is not trivial for general x_n, h_n. To verify that we can interchange the order of the infinite series, we will appeal to an area of mathematics called $Distribution\ Theory$. Distribution theory is a branch of mathematical analysis that encompasses the general theory of objects like the "Dirac delta function" familiar to most students of physics and electrical engineering. This theory gives a result that, informally, states that for essentially all x_n, h_n we would ever encounter(even for "shitty filters" that are poorly designed and produce noisy and/or discontinuous outputs) and also for noisy and/or discontinuous input data), a weak form of convergence exists that permits the interchange of the infinite sums.

$$Y(e^{i\omega}) = \sum_{k=-\infty}^{\infty} x_k (\sum_{n=-\infty}^{\infty} h_n e^{-i\omega n}) e^{-i\omega k} \tag{10}$$

$$Y(e^{i\omega}) = \sum_{k=-\infty}^{\infty} x_k e^{-i\omega k} (H(e^{i\omega})) \tag{11}$$

$$Y(e^{i\omega}) = X(e^{i\omega}) H(e^{i\omega}) \tag{12}$$

Digital filters are often characterized by their frequency response, which is the DTFT of $\{h_n\}$, denoted by the function $H(e^{i\omega})$. If the filter is FIR, $H(e^{i\omega})$ is a polynomial in $e^{i\omega}$.
As can be seen, the DTFT is an infinite series, but a computer does not have inifnite memory. This is a problem if the signals involved are infinite in length, as is the case in communication systems. Thus, if the DTFT were to be evaluated on a computer, it is necessary to compute a truncation of the series. This finite version of the DTFT is called the discrete Fourier transform, or DFT.

Section II-2: Intro to the DFT

The DFT can be thought of as sampling the DTFT at uniformly space intervals on the unit circle. If N such points are sampled, then these points are called the Nth roots of unity

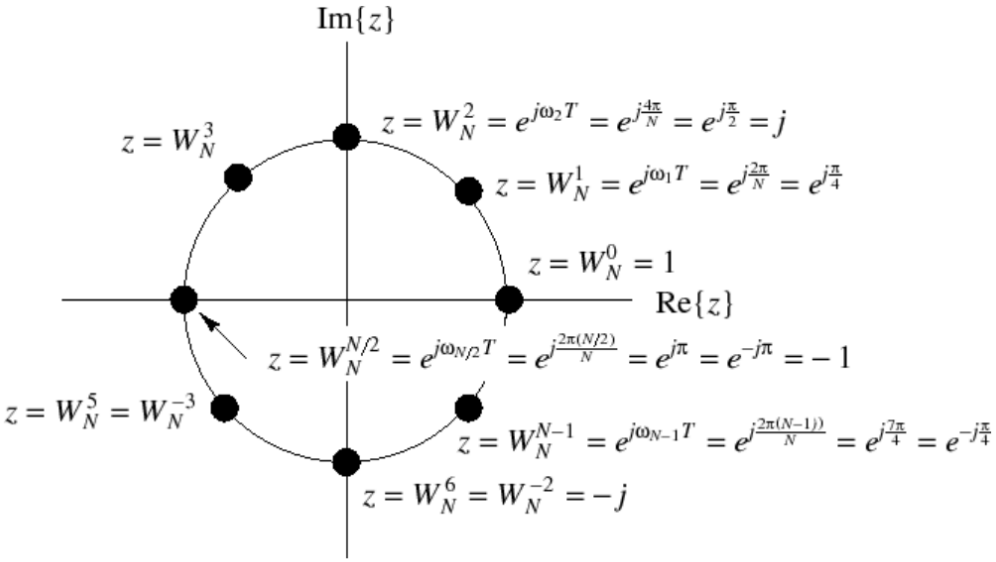


Figure 6.1: The N roots of unity for $N = 8$.

This sampling operation can be described algebraically as:

$$\hat{X}_k = X(e^{i(\frac{2\pi}{N})k}) \text{for} k = 0, 1, \ldots, N-1$$

We are now ready to define the DFT:
$\mathbf{Definition (DFT)}$: Let $\{x_n\}$ be a finite length sequence with length N. The DFT of $\{x_n\}$, denoted by the sequence $\{\ \hat{X}_k\}$, is given by:

$$\hat{X}_k = \sum_{n=0}^{N-1} x_n e^{-i(\frac{2\pi}{N})kn} \text{ for } k = 0, 1, .., N-1$$

Section II-2: Matrix Representation of the DFT and its Properties

The formula for the DFT is more easily visualized by its matrix formulation: Let $x = [x_0 x_1 \ldots x_{N-1}]^T \in \mathbb{C}^N$ denote the vectorial representation of the sequence $\{x_n\}$ and let $\hat{X} = [\hat{X}_0 \hat{X}_1 \ldots \hat{X_{N-1}}]^T \in \mathbb{C}^N$ denote the vectorial representation of the discrete Fourier transform of $\{x\}$. $\hat{X}$ is given by

$$\begin{bmatrix} \hat{X}_0 \\ \hat{X}_1 \\ \hat{X}_2 \\ \vdots \\ \hat{X_{N-1}} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & W_N & W_N^2 & \ldots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \ldots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \ldots & W_N^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

where $W_N = e^{-i(\frac{2\pi}{N})}$. These are collectively called "Twiddle Factors". This can be restated more

compactly, with $\boldsymbol{W}$ denoting the DFT matrix, as

$$\hat{X} = \boldsymbol{W}x$$

The DFT matrix $\boldsymbol{W}$ has several useful properties. One is that it is symmetric. Moreover, it its conjugate symmetric, or $hermitian$, i.e. $\boldsymbol{W} = \overline{\overline{\boldsymbol{W^T}}} = (\overline{\overline{\boldsymbol{W}}})^T$ where T denotes transposition and $\overline{\overline{\boldsymbol{W}}}$ denotes complex conjugation.
Let $e_j = [1 W_N^{(1)j} \dots W_N^{(k)j} \dots W_N^{(N-1)j}]^T$. The DFT matrix $\boldsymbol{W}$ can thus be given by

$$\boldsymbol{W} = [e_0 e_1 \dots e_{N-1}]$$

It follows that

$$< e_j, e_k >= \begin{cases} N & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases}$$

So the vectors e_j for $j = 0, 1, \dots, N-1$ are orthogonal to each other and also form a spanning set(proof omitted) of $\mathbb{C}^N$. Thus, $\{\frac{1}{\sqrt{N}} e_0 \frac{1}{\sqrt{N}} e_1 \dots \frac{1}{\sqrt{N}} e_{N-1}\}$ is an orthonormal basis of $\mathbb{C}^N$. Thus, by the best approximation theorem proved above, the DFT $\{\hat{X}_k\}$ represents the best approximation of $\{x_n\}$ in the finite-dimensional Hilbert space spanned by $\{e_0 e_1 \dots e_{N-1}\}$, which is a subspace of $\mathbb{C}^N$
.
It is then a direct consequence that the matrix $\frac{1}{\sqrt{N}} \boldsymbol{W}$ is unitary. This means that,

$$\frac{1}{N} \boldsymbol{W} \overline{\overline{\boldsymbol{W}^T}} = \frac{1}{N} \overline{\overline{\boldsymbol{W}^T}} \boldsymbol{W} = I_N$$

where I_N is the NxN identity matrix. The matrix formulation of the DFT can be used to prove all of the DFT properties, which are summarized in the table below

TABLE 8.2 SUMMARY OF PROPERTIES OF THE DFT

Finite-Length Sequence (Length N)	N-point DFT (Length N)
1. $x[n]$	$X[k]$
2. $x_1[n], x_2[n]$	$X_1[k], X_2[k]$
3. $ax_1[n] + bx_2[n]$	$aX_1[k] + bX_2[k]$
4. $X[n]$	$Nx[((-k))_N]$
5. $x[((n-m))_N]$	$W_N^{km}X[k]$
6. $W_N^{-\ell n}x[n]$	$X[((k-\ell))_N]$
7. $\sum_{m=0}^{N-1} x_1[m]x_2[((n-m))_N]$	$X_1[k]X_2[k]$
8. $x_1[n]x_2[n]$	$\frac{1}{N}\sum_{\ell=0}^{N-1} X_1[\ell]X_2[((k-\ell))_N]$
9. $x^*[n]$	$X^*[((-k))_N]$
10. $x^*[((-n))_N]$	$X^*[k]$
11. $\mathcal{R}e\{x[n]\}$	$X_{ep}[k] = \frac{1}{2}\{X[((k))_N] + X^*[((-k))_N]\}$
12. $j\mathcal{I}m\{x[n]\}$	$X_{op}[k] = \frac{1}{2}\{X[((k))_N] - X^*[((-k))_N]\}$
13. $x_{ep}[n] = \frac{1}{2}\{x[n] + x^*[((-n))_N]\}$	$\mathcal{R}e\{X[k]\}$
14. $x_{op}[n] = \frac{1}{2}\{x[n] - x^*[((-n))_N]\}$	$j\mathcal{I}m\{X[k]\}$
Properties 15–17 apply only when $x[n]$ is real.	
15. Symmetry properties	$\begin{cases} X[k] = X^*[((-k))_N] \\ \mathcal{R}e\{X[k]\} = \mathcal{R}e\{X[((-k))_N]\} \\ \mathcal{I}m\{X[k]\} = -\mathcal{I}m\{X[((-k))_N]\} \\ \lvert X[k]\rvert = \lvert X[((-k))_N]\rvert \\ \angle\{X[k]\} = -\angle\{X[((-k))_N]\} \end{cases}$
16. $x_{ep}[n] = \frac{1}{2}\{x[n] + x[((-n))_N]\}$	$\mathcal{R}e\{X[k]\}$
17. $x_{op}[n] = \frac{1}{2}\{x[n] - x[((-n))_N]\}$	$j\mathcal{I}m\{X[k]\}$

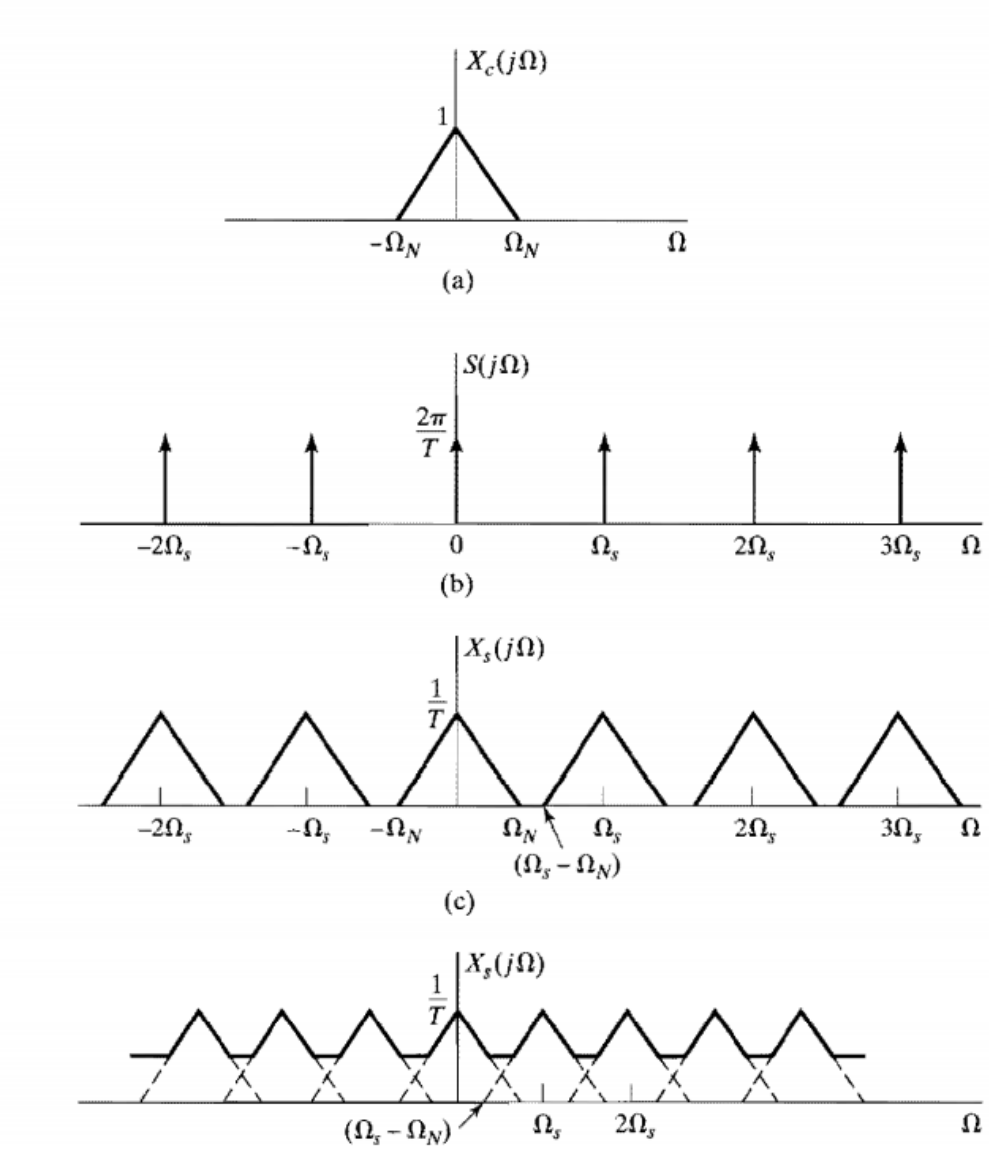
Note that the DFT transform the operation of convolution into termwise multiplication, just like the DTFT does. Thus, once we have computed $\{\hat{X}_k\}$ and $\{\hat{H}_k\}$, computing $\hat{Y}_k = \hat{X}_k\hat{H}_k$ for $k = 0, 1, \ldots, N-1$ is an $O(N)$ operation. $\{y_n\}$ can then be obtained from $\{\hat{Y}_k\}$ by computing the inverse DFT, which is described in Section IV.

Section II-3 Time and Space Complexity of the Explicit Evaluation of the DFT

It is a well-known fact that the time complexity of matrix-vector multiplication is $O(N^2)$, so the time complexity of direct evaluation of an N-point DFT is $O(N^2)$. The space complexity is also $O(N^2)$ if the matrix is stored explicity, which is not necessary due to symmetry in the twiddle factors. It is actually only necessary to explicity store $W_0, W_1, \ldots, W_{N-1}$ so the space complexity can be reduced to $O(N)$. Anyway, the time complexity of $O(N^2)$ can result in excessivly long computation times if N is large so an improved algorithm for computing the DFT is necessary fo even moderately large N. Thus, we obtain no computational savings by evaluating the convolution by computing the DFT (and its inverse) explicity. Fortunately, numerous faster algorithms exist for computing the DFT and its inverse.

Section II-4: The Shannon-Nyquist Sampling Theorem and Aliasing

Before we discuss a class of highly efficient algorithms for computing the DFT, we must first introduce one more very important concept from signals and systems theory, that is, the Shannon-Nyquist sampling theorem. This theorem was published by Claude Shannon at MIT in 1948, building upon the earlier research of Harry Nyquist, a scientist at Bell Labs, in the 1920s. Discrete time signals $\{x_n\}$ do not exist in nature, they must be generated by some man-made process. This process of transforming a continuous-time signal $x(t)$ into a discrete time signal($\{x_n\}$) is called sampling.
$\mathbf{Theorem (Sampling\ Theorem)}$:Let $x(t)$ be a real signal with continuous time Fourier transform $\hat{X(i\Omega)} = \int_{-\infty}^{\infty} x(t)e^{-2\pi i\Omega t}dt$ where $\Omega \in \mathbb{R}$. Suppose that $X(i\Omega)$ is band-limited, i.e., is 0 whenever $|\Omega| > B$ for some $B \in \mathbb{R}$. B is denoted as Ω_N in the picture below. Then, if $x(t)$ is sampled at a rate Ω_s greater than or equal to 2B to produce $\{x_n\}$, then there exists a process in which $x(t)$ can be recovered from $\{x_n\}$ without any loss of information.



A complete discussion of the mathematics involved would take far too long for the pruposes of this presentation. Therefore, we will summarize a bit: Whenever a continous-time signal $x(t)$ with band-limited Fourier transform $X(e^{i\omega})$ is sampled, the sampling operation condenses its entire spectrum, that is, its band-limited Fourier transform, into a "normalized frequency" interval of length 2π. This

spectrum is the DTFT, $X(e^{i\omega})$, and is why the DTFT is defined on an interval of length 2π. A result of the sampling operation is that periodic copies of the condensed spectrum, with period Ω_s, are made, across the entire real frequency axis. A real analog-to-digital converter would then apply a filtering operation to recover exactly one copy of the spectrum and obtain a finite length $\{x_n\}$. The reason the sampling rate needs to be at least 2B is so the copies of the spectrum $X(e^{i\omega})$ do not overlap in the real frequency domain. as is the case in (c.) in the figure above. Aliasing occurs when copies of the spectrum overlap in the real frequency domain, as is the case in (d.) in the figure above. In this situation, the original signal $x(t)$ cannot be recovered without loss of information, which is undesirable.

A logical question one may ask is what does this have to do with the DFT? The DFT can be viewed as sampling the DTFT in the frequency domain. The DFT samples exist in a finite frequency domain $\{W_N^0, W_N^1, \ldots, W_N^{N-1}\}$. If $\{x_n\}$ is of finite length, then to reconstruct it after calculating its DFT, the length of the DFT must be at least N.

Section III: The Cooley-Tukey Radix-2 Decimation In Time Fast Fourier Transform(FFT)

Section III-1: The Original FFT Algorithm

The problem of devising a more efficient algorithm to compute the DFT was studied for many decades before 1965, when two American mathematicians, James W. Cooley and John Tukey, invented a algorithm for computing the DFT in $O(N \log N)$ time). This was done by exploiting the conjugate symmetry and periodicity of the twiddle factors W_N, to devise a "divide and conquer algorithm for computing the DFT called the Fast Fourier Transform(FFT). For the rest of this presentation, assume N is a power of 2.
We want to recursively apply the DFT to length N/2 subsquences of the original sequence x_n. This is done by splitting the computation into two parts, one of the even-indexed terms and one over the odd-indexed terms:

$$\hat{X}_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \text{ for } k = 0, 1, \ldots, N-1 \tag{13}$$

$$\hat{X}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_N^{(2n)k} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{(2n+1)k} \tag{14}$$

$$\hat{X}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{2nk+k} \tag{15}$$

$$\hat{X}_k = \begin{cases} \sum\limits_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk} + W_N^k \sum\limits_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{N/2}^{nk} & \text{if } k = 0, 1, \ldots, N/2-1 \\ \sum\limits_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk} - W_N^k \sum\limits_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{N/2}^{nk} & \text{if } k = N/2, N/2+1, \ldots, N-1 \end{cases}$$

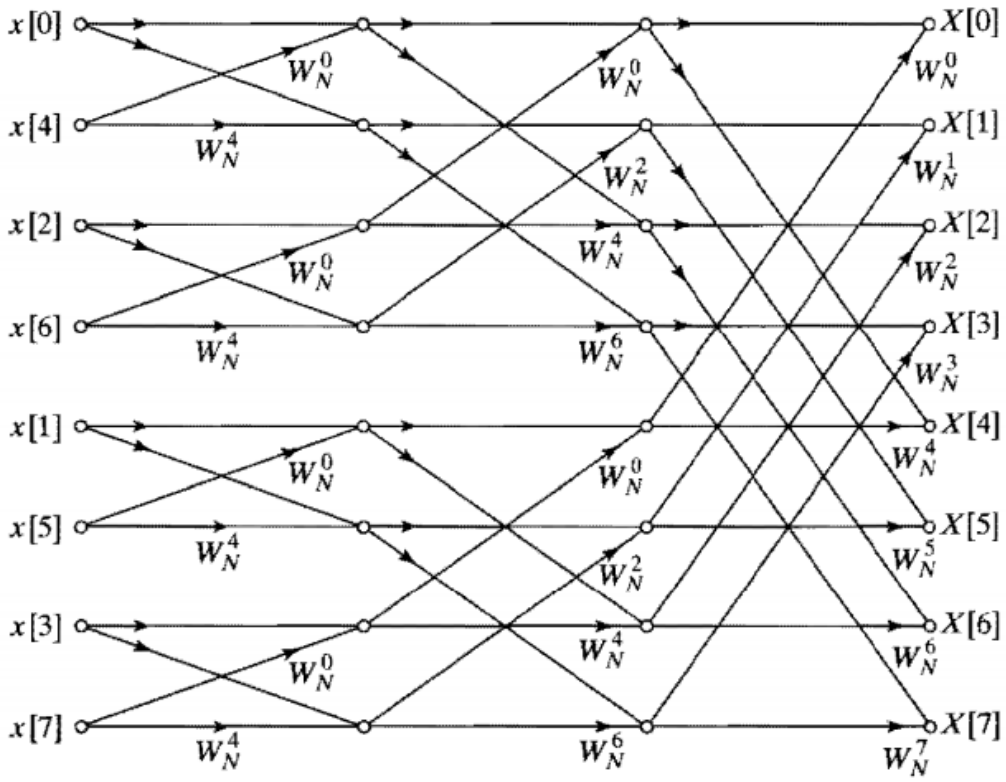
Define $\hat{G}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk}, \hat{H}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk}$ for k $= 0, 1, \ldots, \frac{N}{2} - 1$ so $\hat{X}_k$ can be re-written as

$$\hat{X}_k = \begin{cases} \hat{G}_k + W_N^k \hat{H}_k & \text{if } k = 0, 1, \ldots, N/2 - 1 \\ \hat{G}_k - W_N^k \hat{H}_k & \text{if } k = N/2, N/2 + 1, \ldots, N - 1 \end{cases}$$

This procedure can be recursively applied until the size of the subsequence reaches a base case(usually 2, but can also be 4), at which point the DFT is computed explicitly. The two-point DFT of $\{x_0, x_1\}$ is given by

$$\{\hat{X}_0 = x_0 + x_1, \hat{X}_1 = x_0 - x_1\}$$

The overall procedure for N = 8, which involves two recursions, is shown below:



Note the ordering on the $x_n's$ on the left. The ordering shown is after all of the initial recursions are complete, which is perhaps more clearly explained by the code below. The above graph shows the recursive "reassembly" of the decimated $x_n's$ into the $X_k's$

Python3 Implementation of Cooley-Tukey Decimation In Time FFT Algorithm

In [2]:

```
def ogfft(x, N):
    """
    Recurisve radix-2 FFT implementation directly from divide-and-conquer description
    """
    PI = np.pi
    if N == 2:
        return np.array([x[0] + x[1], x[0] - x[1]])
    else:
```

```
        m = int(N/2)
        big_G = ogfft(x[0:N:2],m)
        big_H = ogfft(x[1:N:2],m)
        w = np.exp((-2.0*PI*1.0j)/N)
        d = np.power(w, np.arange(m))
        d = np.multiply(d, big_H)
        X = np.concatenate((big_G + d, big_G - d))
        return X
```

In [3]:

```
def brc(x):
    """
    Returns bit-reversed order copy of array x
    """
    N = len(x)
    y = np.zeros(N, dtype=complex)
    width = int(np.log2(N))
    for ii in np.arange(N):
        idx = '{:0{width}b}'.format(ii, width=width)
        y[ii] = x[int(idx[::-1],2)]#Reverse order of bits of integer ii
    return y
```

In [12]:

```
def ogfft2(x, N):
    """
    In-place, non-recursive implementation of radix-2 Cooley-Tukey decimation in time F
    """
    x_p = brc(x)
    PI = np.pi
    for ii in np.arange(1,int(np.log2(N)) + 1):
        M = int(2**ii)
        w_M = np.exp(1j*((2*PI)/M))
        for kk in np.arange(0,N,M):
            w = 1
            m = int(M/2)
            for jj in np.arange(m):
                t = w*x_p[kk + jj + m]
                u = x_p[kk + jj]
                x_p[kk + jj] = u + t
                x_p[kk + jj + m] = u - t
                w = w*w_M
    return x_p
```

Section III-2: Time and Space Complexity of the Radix 2 FFT

The function ogfft2() reveals the time and space complexity of the radix-2 FFT. The array x needs to be copied into its bit-reversed order, so the space complexity is $O(N)$. It is possible to do this independently of the FFT iteration/recursion, so the FFT by itself has a space complexity of $O(1)$. The function brc() that does the bit reverse ordering clearly has a time complexity of $O(N \log N)$ as it iterates through the entire list, and the reversal takes $O(\log N)$ operations.
Let F(n) denote the number of times the innermost for loop of ogfft2() is executed: The code block containing the inner most for loops will run $\frac{N}{M} = \frac{N}{2^i}$ times for each value of i. The innermost for loop will then run 2^{i-1} times. Thus, the time complexity is given by

$$F(n) = \sum_{i=1}^{\log_2 N} \frac{N}{2^i}(2^{i-1}) = \sum_{i=1}^{\log_2 N} \frac{N}{2} = O(N \log_2 N)$$

Thus, the time complexity of computing the DFT can be reduced from $O(N^2)$ to $O(N \log N)$, a huge reduction. This discovery is considered to be one of the most important developments in applied mathematics in the past six decades, and is one of the reasons why we have many technologies, such as LTE and 5G wireless communications, and Praveen's horseback riding radios, today.

Section IV: Other Radix-2 FFT Algorithms

Section IV-1: FFT For Purely Real Data: The Fast Hartley Transform(FHT)

The FFT subroutines implemented above are designed to operate on complex numbers. This is necessary in many applications, such as radar processing and optical filter design. In other applications of the FFT however, such as audio processing, or constructing a fast subroutine for the multiplication of real-valued polynomials, it is only necessary to apply the FFT to real data. This is done by changing the basis of the DFT matrix from one having complex exponentials to one having real-valued functions. One way of doing this is to use the following orhtogonal basis for R^N:

$$\{H_0 H_1 \dots H_{N-1}\}, \text{ where } H_j = [1 H_N^{(1)j} \dots H_N^{(k)j} \dots H_N^{(N-1)j}]^T$$

.
where $N \in \mathbb{Z}^+$ and H_N^k is given by

$$H_N^k = \cos\left(\frac{2\pi}{N}k\right) + \sin\left(\frac{2\pi}{N}k\right) \text{ for k = 0,1,...,N-1}$$

H_N^k satisifies the following crucial identity:

$$H_N^{j+k} = C_N^k H_N^j + S_N^k H_N^{-j}$$

where $C_N^k = \cos\left(\frac{2\pi}{N}k\right)$ and $S_N^k = \sin\left(\frac{2\pi}{N}k\right)$
Define the discrete Hartley transform(DHT) of a length N sequence $\{x_n\}$ to be the sequence $\{X_k^H\}$, given by

$$X_k^H = \sum_{n=0}^{N-1} x_n H_N^{nk} \text{ for } k = 0, 1, \dots, N-1$$

Using the identity, a recursive decimation-in-time algorithm for when N is a power of 2 similar to the FFT can be derived to efficiently compute the DHT:

$$X_k^H = \sum_{n=0}^{N-1} x_n H_N^{nk} \text{ for } k = 0, 1, \ldots, N-1 \tag{16}$$

$$X_k^H = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} H_N^{(2n)k} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} H_N^{(2n+1)k} \tag{17}$$

$$X_k^H = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} H_{N/2}^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} H_N^{2nk+k} \tag{18}$$

$$X_k^H = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} H_{N/2}^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} (C_N^k H_{N/2}^{nk} + S_N^k H_{N/2}^{-nk}) \tag{19}$$

$$X_k^H = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} H_{N/2}^{nk} + C_N^k (\sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} H_{N/2}^{nk}) + S_N^k (\sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} H_{N/2}^{-nk}) \tag{20}$$

Remember to evaluate terms with negative indices modulo $\frac{N}{2}$.

The Python function below implements the above procedure, which is called the Fast Hartley Transform(FHT). Many FFT implementations that are designed to operate with purely real input such as Numpy's rfft() do not employ the FHT, but the FHT captures the general idea of how to construct such FFTs for purely real data. The time complexity of the FHT is identical to that of the FFT, $O(N \log N)$ but fewer total operations will be perfomed as we will be adding/multiplying real and not complex numbers. The space complexity can be reduced to $O(1)$ by storing a "lookup table" of the values of sines and cosines needed before the computation takes place and use of clever indexing.

In [25]:
```
def fht(x, N):
    """
    Perform recursive radix-2 FHT on real input vector x
    """
    if N == 2:
        return np.array([x[0] + x[1], x[0] - x[1]])
    else:
        m = int(N/2)
        y_t = fht(x[0:N:2],m)
        y_b = fht(x[1:N:2],m)
        H_N_C = np.cos((2*PI*np.arange(m))/N)
        H_N_S = np.sin((2*PI*np.arange((m)))/N)
        idx_map = np.concatenate(([0], np.arange(m - 1, 0, -1)))
        y_1 = y_t + np.multiply(H_N_C,y_b) + np.multiply(H_N_S, y_b[idx_map])
        idx_map = np.concatenate(([0], np.arange(m - 1, 0, -1)))#Modulo N/2
        y_2 = y_t - np.multiply(H_N_C,y_b) - np.multiply(H_N_S, y_b[idx_map])
        y = np.concatenate((y_1, y_2))
        return y
```

Section IV-2: FFT With Purely Imaginary Twiddle Factors: The Rader-Brenner Radix-2 FFT

In 1976 Rader and Brenner devised an alternative formulation for the radix-2 decimation in time FFT using purely imaginary twiddle factors. This substantially cuts down the total number of multiplcation operations performed, as all of the multiplications will include at most one complex

number. This comes at a cost of an increased number of real addition operations.
Let $\{x_n\}_{n=0}^{N-1}$ be a complex sequence and let N be a power of 2. Define the finite subsequences $< br >$

$$y_{t_n} = x_{2n} \text{ for } n = 0, 2, \ldots, \frac{N}{2} - 1 \tag{21}$$

$$y_{b_n} = x_{2n+1} - x_{2n-1} + q \text{ for } n = 1, 3, \ldots, \frac{N}{2} - 1 \tag{22}$$

where $q = \frac{2}{N} \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1}$.
q is a constant, so it appears in only one term, $y_{b_0} = x_{\frac{N}{2}}$.

Using periodicity and some properties of the complex exponential $W_N^k = e^{-i\frac{2\pi}{N}k}$, we can derive, for $k = 1, 2, \ldots, \frac{N}{2} - 1$

$$Y_{b_k} = \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{nk} - \sum_{n=0}^{N/2-1} x_{2n-1} W_{N/2}^{nk} \tag{23}$$

$$Y_{b_k} = \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{nk} - \sum_{n=0}^{N/2-1} x_{2n-1} (W_{N/2}^{0}) W_{N/2}^{nk} \tag{24}$$

$$Y_{b_k} = \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{nk} - \sum_{n=0}^{N/2-1} x_{2n+1} (W_{N/2}^{2k}) W_{N/2}^{nk} \tag{25}$$

$$Y_{b_k} = \hat{X_{2k+1}}(1 - W_{N/2}^{2k}) \tag{26}$$

$$Y_{b_k} = \hat{X_{2k+1}}(-W_{N/2}^{k})(W_{N/2}^{k} - W_{N/2}^{-k}) \tag{27}$$

$$Y_{b_k} = \hat{X_{2k+1}}(W_{N/2}^{k})(2i \sin \frac{4\pi}{N}) \tag{28}$$

$$W_{N/2}^{k} \hat{X_{2k+1}} = \frac{\hat{Y_{b_k}}}{2i \sin \frac{4\pi}{N}} \tag{29}$$

Therefore,

$$\hat{Y_k} = \begin{cases} \hat{Y_{t_0}} + \hat{Y_{t_0}} & \text{if } k = 0 \\ \hat{Y_{t_0}} + \frac{\hat{Y_{b_k}}}{2i \sin \frac{4\pi}{N}} & \text{if } k = 1, 2, \ldots, \frac{N}{2} - 1 \\ \hat{Y_{t_0}} - \hat{Y_{t_0}} & \text{if } k = \frac{N}{2} \\ \hat{Y_{t_0}} - \frac{\hat{Y_{b_k}}}{2i \sin \frac{4\pi}{N}} & \text{if } k = \frac{N}{2} + 1, \frac{N}{2} + 2, \ldots, N - 1 \end{cases}$$

A recursive Python3 implementation of this algorithm is shown below.

In [6]:
```
def fftRB(x, N):
    """
    Perform recursive radix-2 Rader-Brenner FFT on input vector x

    x: Input vector/array
    N: Length of input vector, must be a power of 2
    """
```

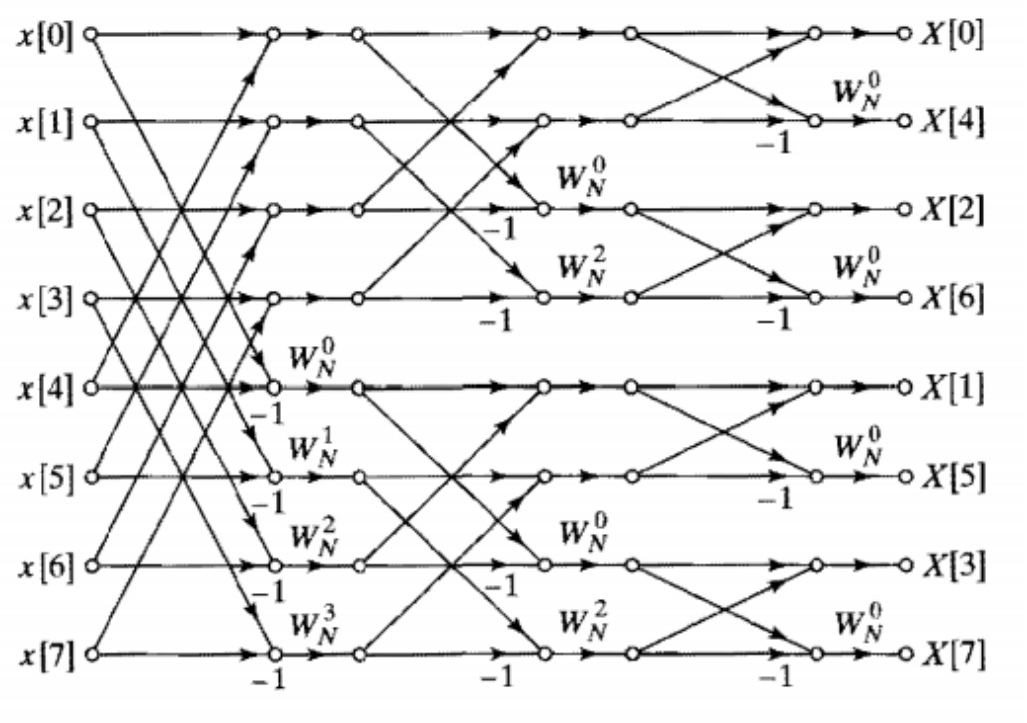
```
    if N == 2:
        return np.array([x[0] + x[1], x[0] - x[1]])
    else:
        m = N // 2
        y_t = fftRB(x[0:N:2],m)
        x_b = x[1:N:2]
        q = (2.0/N)*np.sum(x_b)
        b_n = x_b - np.concatenate(([x_b[m-1]], x_b[0:(m-1)])) + q
        y_b = fftRB(b_n,m)
        w_ky_b = np.multiply(y_b[1:m], 1.0/(2.0j*np.sin((2*PI*np.arange(1,m))/N)))
        y = np.concatenate(([y_t[0] + y_b[0]], y_t[1:m] + w_ky_b, [y_t[0] - y_b[0]], y_
        return y
```

Section IV-3: Cooley-Tukey Radix-2 Decimation In Frequency Algorithm

The previously defined FFT algorithms share one common feature: the sequence $\{x_n\}$ of length N is split into two smaller subsequences of length $\frac{N}{2}$ before the call to fft() is applied. The smaller DFT/FFT sequences are then combined to form the overall length N DFT/FFT $\{\hat{X_k}\}$. In other words, the decimation of $\{x_n\}$ is done in the time domain. What if instead the decimation can instead be done in the frequency domain?. This is indeed possible, and FFT algorithms that employ this technique are called decimation-in-frequency algorithms. The formulas that describe the radix-2 decimation in frequency algorithms can be derived, a bit less rigorously, from the decimation in time formulas using the transposition theorem of signal flow graphs. If the input and output nodes of the signal flow graph in Section III-1 are interchanged and the direction of each branch in flipped, we arrive at the signal flow graph for N=8 and equations below:

$$\hat{X_{2k}} = \sum_{n=0}^{\frac{N}{2}-1}(x_n + x_{n+N/2})W_{N/2}^{nk} \text{ for k } = 0, 1, \ldots, N/2-1 \tag{30}$$

$$\hat{X_{2k+1}} = \sum_{n=0}^{\frac{N}{2}-1}(x_n - x_{n+N/2})W_N^n W_{N/2}^{nk} \text{ for k } = 0, 1, \ldots, N/2-1 \tag{31}$$



The decimation in frequency formulas can also be derived using the sampling theorem: Splitting $\{\hat{X}_k\}$ into two smaller sequences effectively reducesthe sampling rate by 1/2. Therefore, there will be aliasing when $\{x_n\}$ is reconstructed from $\{\hat{X}_k\}$. The even-index samples of $\{\hat{X}_k\}$ are not circularly shifted. Therefore, we do two copies of its inverse DFT $\{x_{2n}\}$ will lie in the interval $0 \leq n \leq \frac{N}{2} - 1$, itself and itself shifted by $\frac{N}{2}$. Let $\{x_{2n}\}$ denote the even indexed samples of $\{x_n\}$. They are given by:

$$x_{2n} = x_n + x_{n+N/2} \text{ for } n = 0, 1, .., N/2 - 1$$

The odd-indexed terms $\{\hat{X}_{2k+1}\}$ are each circularly shifted by one. By the circular shifting peoperty of the DFT and the effect of time-domain aliasing, we arrive at:

$$x_{2n+1} = x_n W_N^n + x_{n+N/2} W_N^{n+N/2} = (x_n - x_{n+N/2}) W_N^n$$

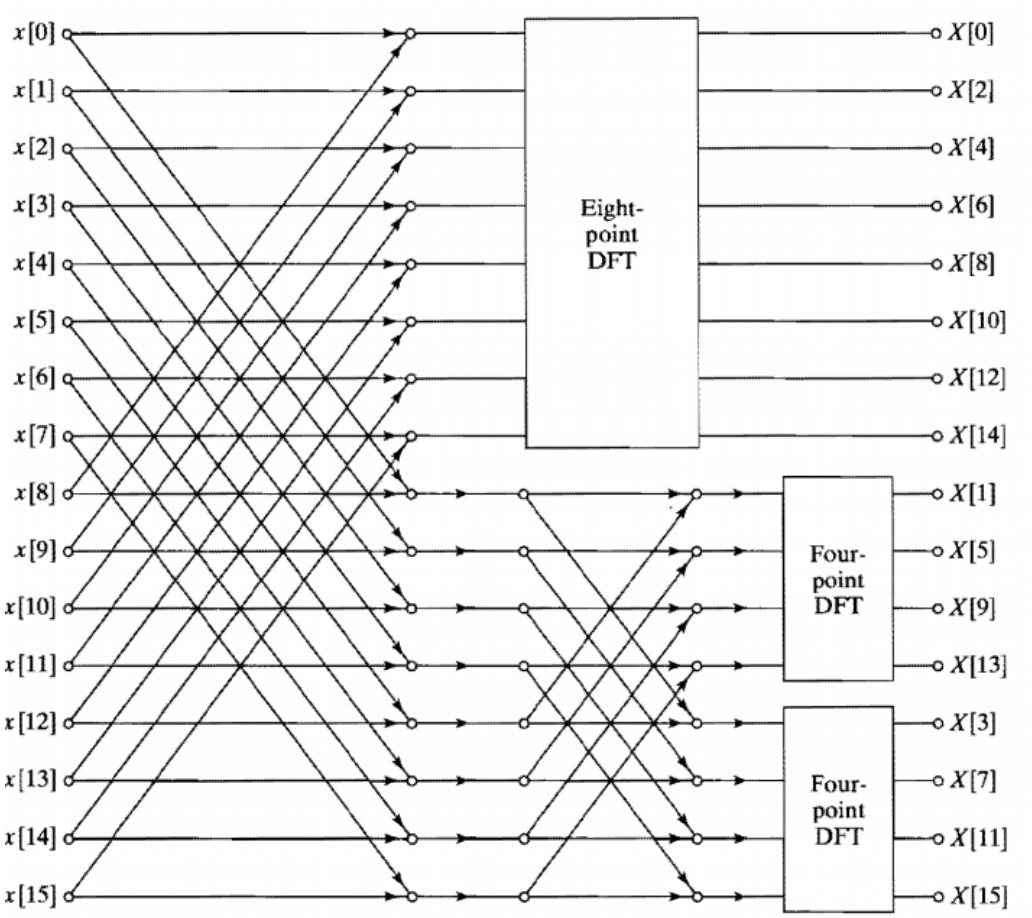
An in-place Python3 implementation is shown below. Note the similarity to the in-place decimation in time FFT algorithm, including the placement of the call the brc().

In [7]:
```
def ogfftf(x, N):
    """
    Performs in-place Cooley-Tukey radix-2 decimation in frequency FFT
    """
    m = N // 2
    for ii in np.arange(1,int(np.log2(N)) + 1):
        M = N // (2**(ii-1))
        for kk in np.arange(0,N,M):
            for jj in np.arange(m):
                idx = jj + kk
                p = (2**(ii-1))*jj
                w = np.exp(-1j*2*PI*(p/N))
                x_0 = x[idx] + x[idx + m]
                x_1 = (x[idx] - x[idx + m])*w
```

```
                x[idx] = x_0
                x[idx + m] = x_1
        m = m // 2
    return brc(x)
```

Section IV-4: Split-Radix FFT

The split-radix FFT is a variation of the standard radix-2 decimation in frequency FFT. The general idea is the same, except that the base cases sizes of the recursion are not equal i.e. for N = 16 one recursion could terminate in a base case size of 8, the other in a base case size of 4. This can result in substantive computational savings for sparse FFTs, but the exact details of the savings depend greatly on the situation the FFT is being applied in as well as hardware-spec ific details. A sample flow chart for a split radix FFT for N=16 is shown below.



Section V: The Inverse FFT(IFFT)

Section V-1: Inverse DFT

To perfrom the complete filtering operation mentioned in Section I-1, we need to calculate the DFT of the s the sequences $\{x_n\}_{n=0}^{N-1}$ and $\{h_n\}_{n=0}^{N-1}$ into the

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} \hat{X}_k W_N^{-kn} \text{ for } n = 0, 1, \ldots, N-1$$

As mentioned in Section II-2, the inverse of the DFT matrix $\boldsymbol{W}$ is given by $\frac{1}{N}\overline{\boldsymbol{W}^T}$.
The inverse DFT operation can be described using matrices and vectors as

$$x = \frac{1}{N}\overline{\boldsymbol{W}^T}\hat{X}$$

This suggests that, with a few algebraic manipulations to $\{\hat{X}_k\}$ one can use the forward FFT to compute the inverse FFT. This is extremely advantageous in hardware implementations of the FFT. We will study two of the most common approaches for doing this.

Section V-2: Computing the Inverse FFT Using Conjugation and Forward FFT

The above matrix-vector formulation of the inverse DFT suggests a method for computing the IFFT. By using the conjugation property of the DFT and some complex number manipulation, we arrive at the formula

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} \overline{\hat{X}_k} W_N^{nk} \text{ for } n = 0, 1, \ldots, N-1$$

This approach is used to construct an inverse FFT subroutine in Python3 using the in-place forward FFT subroutine ogfft2() from section II-2

In [13]:
```
def ogifft2(x_hat, N):
    """
    Perform radix-2 inverse FFT on transform vector x_hat

    x_hat: Input vector/array
    N: Length of input vector, must be a power of 2
    """
    x_hat_star = np.conjugate(x_hat)
    y_star = ogfft2(x_hat_star, N)
    y = (1.0/N)*np.conjugate(y_star)
    return np.real_if_close(y)
```

In [16]:
```
x = np.array([-0.5, 2.2, 3.7, 2.1j, 5.6, -3.3, 16.7, 8.8], dtype=complex)
x_og = np.copy(np.real_if_close(x))
N = len(x)
x_hat = ogfft2(x, N)
y = ogifft2(x_hat, N)
print(x_og)
print(np.real_if_close(y))
```

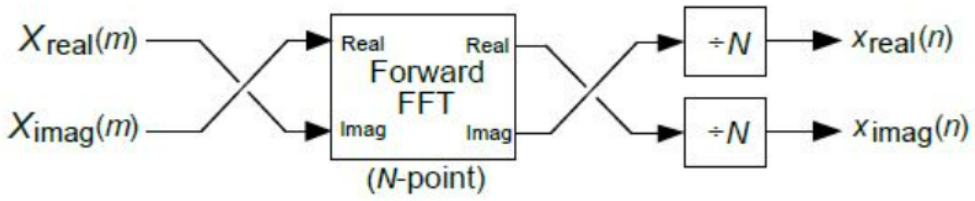
```
[-0.5+0.j   2.2+0.j   3.7+0.j   0. +2.1j  5.6+0.j  -3.3+0.j  16.7+0.j
  8.8+0.j ]
[-5.0000000e-01-0.00000000e+00j  2.2000000e+00+6.66133815e-16j
```

```
  3.7000000e+00+4.44089210e-16j -8.8817842e-16+2.10000000e+00j
  5.6000000e+00+0.00000000e+00j -3.3000000e+00-6.66133815e-16j
  1.6700000e+01-4.44089210e-16j  8.8000000e+00-8.88178420e-16j]
```

Since the forward FFT can be computed with a time complexity of $O(N \log N)$ and with a space complexity of $O(1)$, so can the inverse FFT.

Section V-3: Computing the inverse FFT By Interchanging the Real and Imaginary Parts and Forward FFT

It follows by properties 11-14 in the table of DFT properties that $\{x_n\}$ can be obtained from $\{\hat{X}_k\}$ by interchanging the real and imaginary parts of $\{\hat{X}_k\}$, computing the forward DFT of $\{\hat{X}_k\}$ and scaling by $\frac{1}{N}$, and then interchanging the real and imaginary parts of $\{\hat{\hat{X}}_k\}$ to recover $\{x_n\}$.



In [17]:
```
def ogifft2R(x_hat, N):
    """
    Perform radix-2 inverse FFT on transform vector x_hat
    by interchanging real and imaginary parts

    x_hat: Input vector/array
    N: Length of input vector, must be a power of 2
    """
    x_hat = np.imag(x_hat) + 1j*np.real(x_hat)
    y_star = ogfft2(x_hat, N)
    y = (1.0/N)*(np.imag(y_star) + 1j*np.real(y_star))
    return np.real_if_close(y)
```

In [18]:
```
x = np.array([-0.5, 2.2, 3.7, 2.1j, 5.6, -3.3, 6.7, 8.8], dtype=complex)
x_og = np.copy(np.real_if_close(x))
N = len(x)
x_hat = ogfft2(x, N)
y = ogifft2R(x_hat, N)
print(x_og)
print(y)
```

```
[-0.5+0.j   2.2+0.j   3.7+0.j   0. +2.1j  5.6+0.j  -3.3+0.j   6.7+0.j
  8.8+0.j ]
[-5.0000000e-01+0.00000000e+00j  2.2000000e+00+2.22044605e-16j
  3.7000000e+00+0.00000000e+00j -8.8817842e-16+2.10000000e+00j
  5.6000000e+00+0.00000000e+00j -3.3000000e+00-6.66133815e-16j
  6.7000000e+00+0.00000000e+00j  8.8000000e+00-6.66133815e-16j]
```

Section V-4: Inverse Fast Hartley Transform(IFHT)

Deriving the inverse of the Hartley transform is straightforward. As mentioned earlier, the Hartley matrix $\boldsymbol{H} = [H_0 H_1 \dots H_{N-1}]$ has orthogonal columns, as the column vectors

$H_j = [1 H_N^{(1)j} \dots H_N^{(N-1)j}]^T$ for $j = 0, 1, \dots, N-1$ are orthogonal. By using the normalization constant $\frac{1}{\sqrt{N}}$, $\{H_0, H_1, \dots, H_{N-1}\}$ can be made into an orthonormal basis. Since $\boldsymbol{H}$ is real-valued, it follows that

$$\frac{1}{N}\boldsymbol{H}\boldsymbol{H^T} = I_N$$

So H^T is the inverse of H (ignoring the factor $\frac{1}{N}$). By the properties of sine and cosine, it follows that $\boldsymbol{H}$ is symmetric. Thus, barring the factor of $\frac{1}{N}$, $\boldsymbol{H}$ is its own inverse. Thus, the inverse Hartley transform of $\{\hat{X}_k^H\}$ can be computed by taking its forward Hartley transform, and then scaling by 1/N.

In [28]:
```
def ifht(x_hat, N):
    """
    Perform radix-2 inverse FHT on real transform vector x_hat

    x_hat: Input vector/array
    N: Length of input vector, must be a power of 2
    """
    y = (1.0/N)*fht(x_hat,N)
    return y
```

In [29]:
```
x = np.array([-0.5, 2.2, 3.7, 2.1, 5.6, -3.3, 6.7, 8.8])
N = len(x)
x_hat = fht(x, N)
y = ifht(x_hat, N)
print(x)
print(y)
```

```
[-0.5  2.2  3.7  2.1  5.6 -3.3  6.7  8.8]
[-0.5  2.2  3.7  2.1  5.6 -3.3  6.7  8.8]
```

In [ ]:
```

```