Numerical Fourier Transform

Shoyo Inokuchi

1W16BG02-3

1. Introduction

1.1 Background

The Fourier Transform is an integral transform used in various fields of engineering and science. The process transforms a function of time-domain $f(t)$ to a function of frequency-domain $\hat{f}(\omega)$, and its applications range from analyzing complex waveforms for signal processing to efficiently solving partial differential equations. The Fourier Transform is mathematically denoted

$$\mathcal{F}(f) = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{1.1}$$

The inverse Fourier transform, as the name suggests, serves as an inverse process to the Fourier transform described by Equation (1.1). The process changes a frequency-domain function $\hat{f}(\omega)$ back into a function of time-domain $f(t)$, which is useful for extrapolating the original waveform given a frequency distribution function. The inverse Fourier Transform is defined as

$$\mathcal{F}^{-1}(\hat{f}) = f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega t} d\omega \tag{1.2}$$

The Fourier transform can only be applied to functions that are defined on continuous time signals. For most real-world applications, this condition is satisfied as signals are processed over a given period of time. However, in order to mathematically apply the Fourier transform, the time signal must be observed infinitely precisely. This poses a fundamental problem for applying the mathematical notations expressed by Equations (1.1) and (1.2). In real-world scenarios, a function $f(t)$ can only be sampled at points between finitely spaced intervals. In such cases, we apply numerical methods such as discrete Fourier transform and fast Fourier transform, which we explore in detail in this report.

1.2 Derivation of the Fourier Transform

The concept of Fourier transform is motivated by the study of the Fourier series, where a complex wave function is broken down to a sum of simpler sine and cosine functions. The complex Fourier series synthesis equation is denoted as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t} \tag{1.3}$$

where

$$c_n = \frac{1}{T} \int_T f(t) e^{-in\omega_0 t} dt \tag{1.4}$$

Rearranging Equation (1.4) gives

$$Tc_n = \int_T f(t)e^{-in\omega_0 t}dt \tag{1.5}$$

The Fourier transform is a special case of the Fourier series where the periodic interval is assumed to approach infinity. Therefore, as T approaches infinity ($T \to \infty$) in Equation (1.5), the fundamental frequency ω_0 ($= 2\pi/T$) approaches zero and $n\omega_0$ can take on any value as $-\infty < n < \infty$. Thus, we define $\omega = n\omega_0$ and $\hat{f}(\omega) = Tc_n$. Substituting these newly defined variables into Equation (1.5) yields a common variation of the Fourier transform denoted as

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \tag{1.6}$$

The inverse Fourier transform is similarly derived from the Fourier series sythesis equation. Equation (1.3) is modified to be

$$f(t) = \sum_{n=-\infty}^{\infty} Tc_n e^{in\omega_0 t}\frac{1}{T} \tag{1.7}$$

Using the previously specified definitions of $\omega_0 = 2\pi/T$, $\omega = n\omega_0$, and $\hat{f}(\omega) = Tc_n$ as T approaches infinity, we obtain

$$f(t) = \sum_{n=-\infty}^{\infty} \hat{f}(\omega)e^{i\omega t}\frac{d\omega}{2\pi} \tag{1.8}$$

which simplifies to a common variation of the inverse Fourier transform denoted as

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(\omega)e^{i\omega t}d\omega \tag{1.9}$$

Rearranging Equations (1.6) and (1.9) provides the respective equations for Fourier transform and inverse Fourier transform.

$$\mathcal{F}(f) = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

$$\mathcal{F}^{-1}(\hat{f}) = f(t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} \hat{f}(\omega)e^{i\omega t}d\omega$$

1.3 Objectives

The objectives of this report are the following:

1. Discuss and utilize the discrete Fourier transform (DFT) in order to computationally analyze a given function $f(t)$ defined only in terms of discretized sample points.

2. Reduce computational costs by using fast Fourier transform (FFT), and describe the differences between implementation and performance between DFT and FFT by analyzing time complexity and measured execution times.

3. Discuss inverse Fourier transform algorithms (IDFT and IFFT) and utilize them to transform a frequency spectrum back to its original waveform.

2. Numerical Approach

In this section, we discuss two numerical approaches for the Fourier transform: discrete Fourier transform (DFT) and fast Fourier transform (FFT). We also discuss two corresponding implementations for inverse Fourier transform: inverse discrete Fourier transform (IDFT) and inverse fast Fourier transform (IFFT). DFT and IDFT are straightforward approaches that are derived intuitively from the mathematical transforms shown in Equation (1.1) and (1.2). FFT and IFFT are slightly more obscure in their implementations, but drastically improve computational complexity by using a divide-and-conquer approach and caching intermediate calculation results.

2.1 Discrete Fourier Transform

The DFT of a set of given signals $\boldsymbol{f} = (f_0, \cdots, f_{N-1})^T$ is defined to be the vector $\hat{\boldsymbol{f}} = (\hat{f}_0, \cdots, \hat{f}_{N-1})^T$ where

$$\hat{f}_n = \sum_{k=0}^{N-1} f_k e^{-i2\pi nk/N}, \qquad n = 0, \cdots, N-1 \tag{2.1}$$

Equation (2.1) provides the frequency spectrum of the input signal. In vector notation, DFT is represented as $\hat{\boldsymbol{f}} = \boldsymbol{F}_N \boldsymbol{f}$, where the $N \times N$ Fourier matrix $\boldsymbol{F}_N = \{e_{nk}\}$ has the components

$$e_{nk} = e^{-i2\pi nk/N} = \omega^{nk}, \qquad \omega = \omega_N = e^{-i2\pi/N} \tag{2.2}$$

where $n, k = 0, \cdots, N-1$.

Example:

Suppose $\boldsymbol{f} = (0, 1, 4, 9)^T$. There are 4 discretized measurements, thus $N = 4$. To find the Fourier transform of $\boldsymbol{f}$, we simply calculate

$$\begin{aligned}
\hat{\boldsymbol{f}} &= \boldsymbol{F}_N \boldsymbol{f} \\
&= \begin{bmatrix} e^{-i2\pi(0)(0)/4} & e^{-i2\pi(0)(1)/4} & e^{-i2\pi(0)(2)/4} & e^{-i2\pi(0)(3)/4} \\ e^{-i2\pi(1)(0)/4} & e^{-i2\pi(1)(1)/4} & e^{-i2\pi(1)(2)/4} & e^{-i2\pi(1)(3)/4} \\ e^{-i2\pi(2)(0)/4} & e^{-i2\pi(2)(1)/4} & e^{-i2\pi(2)(2)/4} & e^{-i2\pi(2)(3)/4} \\ e^{-i2\pi(3)(0)/4} & e^{-i2\pi(3)(1)/4} & e^{-i2\pi(3)(2)/4} & e^{-i2\pi(3)(3)/4} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 4 \\ 9 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 4 \\ 9 \end{bmatrix} \\
&= \begin{bmatrix} 14 \\ -4 - 8i \\ -6 \\ -4 + 8i \end{bmatrix}
\end{aligned}$$

Thus, we can see that $\mathcal{F}(\boldsymbol{f}) = (14, -4 - 8i, -6, -4 + 8i)^T$.

2.2 Inverse Discrete Fourier Transform

From the definition of DFT ($\hat{\boldsymbol{f}} = \boldsymbol{F}_N \boldsymbol{f}$), the corresponding input signal in time-domain can be recreated by

$$\boldsymbol{f} = \boldsymbol{F}_N^{-1} \hat{\boldsymbol{f}} \tag{2.3}$$

In Equation (2.3), $\boldsymbol{F}_N$ and its complex conjugate $\bar{\boldsymbol{F}}_N = \{\bar{\omega}^{nk}\}$ satisfy

$$\bar{\boldsymbol{F}}_N \boldsymbol{F}_N = \boldsymbol{F}_N \bar{\boldsymbol{F}}_N = N\boldsymbol{I} \tag{2.4}$$

where $\boldsymbol{I}$ denotes the $N \times N$ identity matrix, and thus $\boldsymbol{F}_N$ has the inverse

$$\boldsymbol{F}_N^{-1} = \frac{1}{N} \bar{\boldsymbol{F}}_N \tag{2.5}$$

Example:

Suppose $\hat{\boldsymbol{f}} = (14, -4 - 8i, -6, -4 + 8i)^T$. To calculate the original vector in time domain, we first calculate $\boldsymbol{F}_N^{-1}$ as follows

$$\begin{aligned}
\boldsymbol{F}_N^{-1} &= \frac{1}{N} \bar{\boldsymbol{F}}_N \\
&= \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \\
&= \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & -i/4 & -1/4 & i/4 \\ 1/4 & -1/4 & 1/4 & -1/4 \\ 1/4 & i/4 & -1/4 & -i/4 \end{bmatrix}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\boldsymbol{f} &= \boldsymbol{F}_N^{-1} \hat{\boldsymbol{f}} \\
&= \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & -i/4 & -1/4 & i/4 \\ 1/4 & -1/4 & 1/4 & -1/4 \\ 1/4 & i/4 & -1/4 & -i/4 \end{bmatrix} \begin{bmatrix} 14 \\ -4 - 8i \\ -6 \\ -4 + 8i \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 4 & 9 \end{bmatrix}^T
\end{aligned}$$

The result is as expected at $\boldsymbol{f} = (0, 1, 4, 9)^T$, and verifies that DFT and IDFT are inverse operations.

2.3 Fast Fourier Transform

This section will explore the most common implementation of FFT referred to as the "decimation-in-time" algorithm. Decimation refers to the process of recursively dividing an N-length input vector into halves until $N = 1$, after which vectors are recombined by calculating ω values and caching intermediate results. The algorithm is detailed below.

When the number of sampled points N satisfies $N = 2^p$, where p denotes an integer, the Fourier matrix can be broken down into smaller subproblems where each resulting matrix is of length $M = N/2$. Because length $N = 2M$,

$$\omega_N^2 = \omega_{2M}^2 = (e^{-2\pi i/N})^2 = e^{-4\pi i/(2M)} = e^{-2\pi i/M} = \omega_M$$

Furthermore, a given N-length vector $\boldsymbol{f} = (f_0, \cdots, f_{N-1})^T$ can be split into two M-length vectors

$$\boldsymbol{f}_{even} = (f_0, f_2, \cdots, f_{N-2})^T$$
$$\boldsymbol{f}_{odd} = (f_1, f_3, \cdots, f_{N-1})^T$$

which respectively contain the even and odd components of $\boldsymbol{f}$. The Fourier transforms are thus

$$\hat{\boldsymbol{f}}_{even} = (\hat{f}_{even,0}, \hat{f}_{even,2}, \cdots, \hat{f}_{even,N-2})^T = \boldsymbol{F}_M \boldsymbol{f}_{even} \tag{2.6}$$
$$\hat{\boldsymbol{f}}_{odd} = (\hat{f}_{odd,1}, \hat{f}_{odd,3}, \cdots, \hat{f}_{odd,N-1})^T = \boldsymbol{F}_M \boldsymbol{f}_{odd} \tag{2.7}$$

which involve the same $M \times M$ matrix F_M. From Equations (2.6) and (2.7), the components of the FFT of vector f can be calculated by

$$\hat{f}_n = \hat{f}_{even,n} + \omega_N^n \hat{f}_{odd,n} \qquad n = 0, \cdots, M-1 \tag{2.8}$$
$$\hat{f}_{n+M} = \hat{f}_{even,n} - \omega_N^n \hat{f}_{odd,n} \qquad n = 0, \cdots, M-1 \tag{2.9}$$

The process of constructing the corresponding N-length vector using M-length vectors as shown in Equations (2.8) and (2.9) is better explained visually. To serve as a basis for more complex illustrations, we first define a visual representation of a complex multiplication and addition operation in Fig. 2.1 *Visual representation of complex multiplication and addition.*

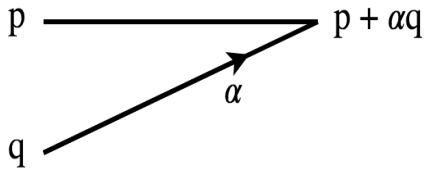


Fig. 2.1 Visual representation of complex multiplication and addition

Let p, q, and α denote complex numbers. Fig. 2.1 illustrates the operation of obtaining the value $p + \alpha q$. To build on this concept, Fig. 2.2 *Illustration of butterfly operation* shows two input values p and q being calculated into two output values $p + \alpha q$ and $p - \alpha q$ according to the operation defined by Fig. 2.1. We refer to the process shown in Fig. 2.2 as a 2-point butterfly operation, due to the number of operations and visual crossing of the lines.

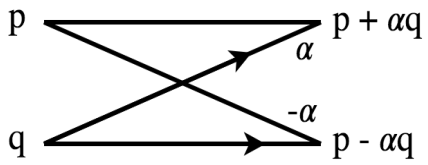


Fig. 2.2 Illustration of butterfly operation

In the case of FFT, α represents the corresponding ω value calculated according to $\omega_N = e^{-i2\pi/N}$, as shown in Equation (2.2). A visual example of FFT when $N = 8$ is shown below in Fig. 2.3 *Visual illustration of an 8-point FFT butterfly computation.*

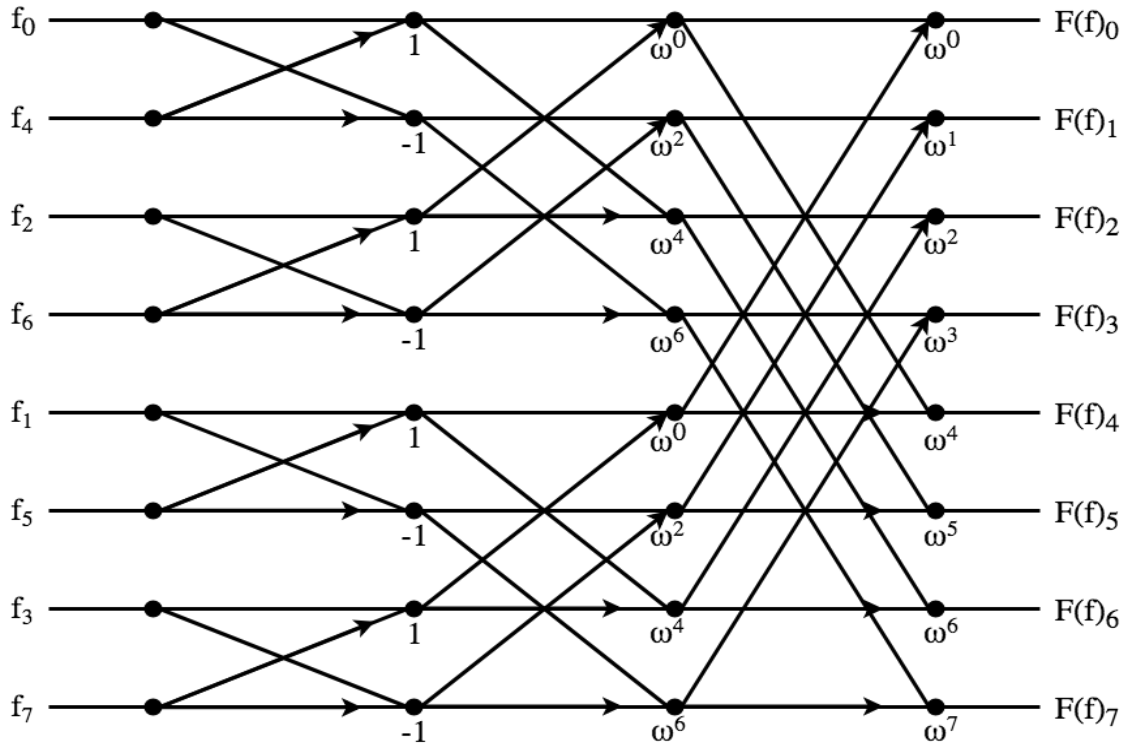


Fig. 2.3 Visual illustration of an 8-point FFT butterfly computation

As Fig. 2.3 shows, the values for vector $\hat{\boldsymbol{f}}$ can be calculated by repeating butterfly operations across with adjacent values, doubling the crossover index with each iteration. It should be noted that the indexes for $\boldsymbol{f}$ on the left of Fig. 2.3 are not in numerical order. This is due to the process of splitting N-length vector $\boldsymbol{f}$ into its even and odd indexes efficiently with bit reversal. When $\boldsymbol{f}$ is split into its even and odd-indexed components, simply reversing the bit order of each index allows the lower half of the vector to become even valued indexes and the upper half to become odd valued indexes. This is shown in Table 2.1 *Relationship between index i and bit-reversed index* n_i. Splitting the input vector as shown in Table 2.1 allows the decimation process to be performed in linear time O(N) with constant space complexity O(1).

Table 2.1: Relationship between index i and bit-reversed index n_i

i	0	1	2	3	4	5	6	7
n_i	0	4	2	6	1	5	3	7
i (base 2)	000	001	010	011	100	101	110	111
n_i (base 2)	000	100	010	110	001	101	011	111

Even in cases where the number of sampled points N is not a power of 2, there are two common methods in order to retain the fast performance of FFT:

1. Break down N into small prime factors other than 2. For example, if N is divisible by 3 (such as when $N = 48$) the problem can be split into a 3-point transform.

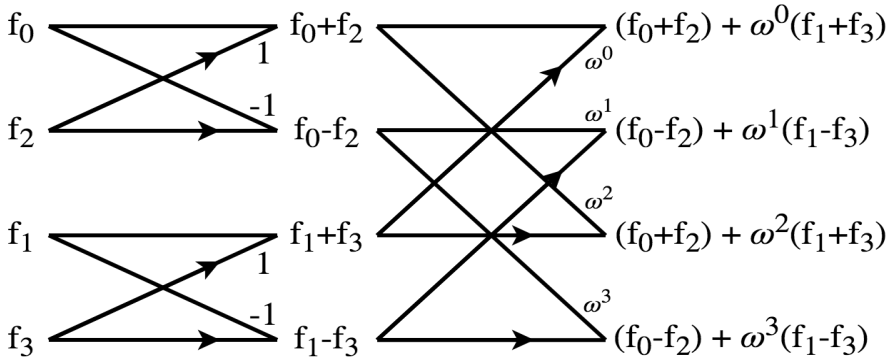
2. Append "dummy" values of zeros into the data in order to make N a power of 2. For example, 15 zeros can be appended to a data set of 49 data points in order to make $N = 64$. It should be noted that this method may cause abrupt transitions in the resulting data.

Example:

Suppose $\boldsymbol{f} = (0, 1, 4, 9)^T$. To apply FFT, we first calculate ω_N where $N = 4$, which is

$$\omega_4 = e^{-i2\pi/4} = i$$

The calculation of $\hat{\boldsymbol{f}}$ can be visualized as follows: Therefore,



$$\hat{f}_0 = (0 + 4) + i^0(1 + 9) = 14$$

$$\hat{f}_1 = (0 - 4) + i^1(1 - 9) = -4 - 8i$$

$$\hat{f}_2 = (0 + 4) + i^2(1 + 9) = -6$$

$$\hat{f}_3 = (0 - 4) + i^3(1 - 9) = -4 + 8i$$

Thus, we find that $\hat{\boldsymbol{f}} = (14, -4 - 8i, -6, -4 + 8i)^T$, which is the same result that was computed by using DFT.

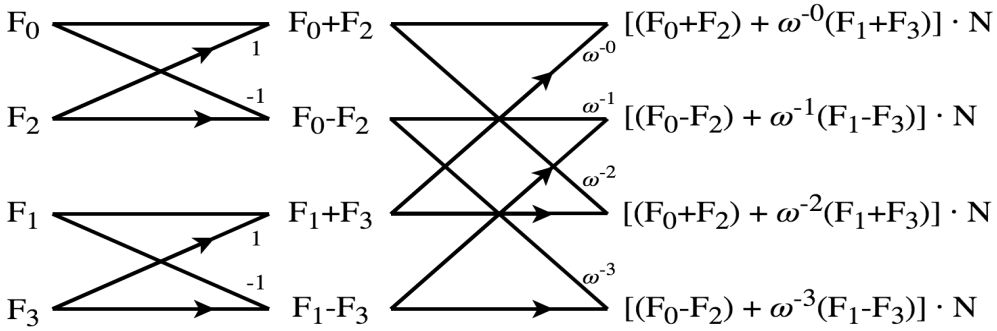
2.4 Inverse Fast Fourier Transform

The inverse fast Fourier transform is nearly identical in implementation to the fast Fourier transform, but with a few differences. We utilize the same process visualized in Fig. 2.3, but reverse the roles of $\boldsymbol{f}$ and $\hat{\boldsymbol{f}}$, negate each exponent n of the ω_N^n value expressed in Equations (2.8) and (2.9), and divide each term by N at the end.

Example: Suppose $\hat{\boldsymbol{f}} = (14, -4-8i, -6, -4+8i)^T$. As with the fast Fourier transform, we calculate ω_N where $N = 4$, which is

$$\omega_4 = e^{-i2\pi/4} = i$$

We then apply the slightly modified decimation-in-time algorithm: Therefore,



$$f_0 = [(14-6) + i^{-0}(-4-8i-4+8i)] \cdot \frac{1}{4} = 0$$

$$f_1 = [(14+6) + i^{-1}(-4-8i+4-8i)] \cdot \frac{1}{4} = 1$$

$$f_2 = [(14-6) + i^{-2}(-4-8i-4+8i)] \cdot \frac{1}{4} = 4$$

$$f_3 = [(14+6) + i^{-3}(-4-8i+4-8i)] \cdot \frac{1}{4} = 9$$

Thus, we can see that $\boldsymbol{f} = (0, 1, 4, 9)^T$.

3. Application

In this section, we utilize the Matlab/Octave programming language to observe Fourier transform algorithms for more complex examples.

3.1 Code Implementation

For discrete Fourier transform and inverse Fourier transform, we implement the algorithms in the Octave language. For fast Fourier transform and inverse fast Fourier transform, we utilize the built-in functions provided by the Matlab library. All of these algorithms are based upon the details discussed in the previous section.

Firstly, the code implementations for DFT and IDFT are shown below.

Discrete Fourier Transform:

```
function outvec = DFT(invec)
    N = size(invec, 2);
    outvec = zeros(1, N);
    for n = 1 : N
        for k = 1 : N
            outvec(n) = outvec(n) + invec(k) ...
                * exp(-i * 2 * pi * (n-1) * (k-1) / N);
        end
    end
```

Inverse Discrete Fourier Transform:

```
function outvec = IDFT(invec)
    N = size(invec, 2);
    outvec = zeros(1, N);
    for n = 1 : N
        for k = 1 : N
            outvec(n) = outvec(n) + invec(k) ...
                * exp(i * 2 * pi * (n-1) * (k-1) / N);
        end
    end
    outvec = outvec / N;
```

These are naive implementations of the Fourier transform which have very slow computation times compared to fast Fourier transforms. The functions used for FFT and IFFT are their respective Matlab functions: fft() and ifft().

3.2 Demonstrations

Example 1. Fourier transform of $f(t) = 0.7\sin(2\pi * 250 * t) + \sin(2\pi * 410 * t)$

We use the code shown below in order to perform a Fourier transform with either DFT() or fft(), and graph its result.

```
Fs = 1000;                  % Sampling frequency
T = 1/Fs;                   % Sampling period
L = 1500;                   % Length of signal
t = (0:L-1)*T;              % Time vector

% Function to be sampled (Example)
f = 2*sin(2*pi*5*t) + sin(2*pi*3*t);

% Fourier transform and inverse Fourier transform
Y = fft(f);

% Adjusting data and Plotting
```

```
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
plot(f,P1)
h = xlabel('f (Hz)');
hh = ylabel('|P1(f)|');
```

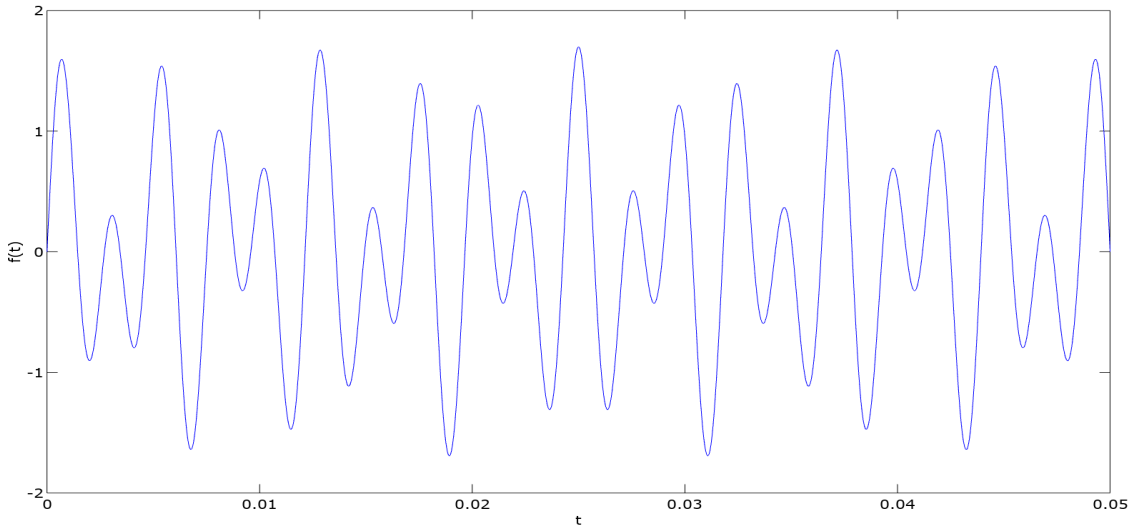


Fig. 3.1 Graph of $f(t) = 0.7\sin(2\pi * 250 * t) + \sin(2\pi * 410 * t)$

Fig. 3.1 above shows a graph of function $f(t)$. As we can observe from the equation, this function is composed of sine functions, which allows us to easily verify the output of the Fourier transform. In this case, we can see that the first and second term of $f(t)$ have frequencies of 250 Hz and 410 Hz, respectively. The Fourier transform using DFT is shown in Fig. 3.2 *Fourier transform of* $f(t)$ *using DFT* below.

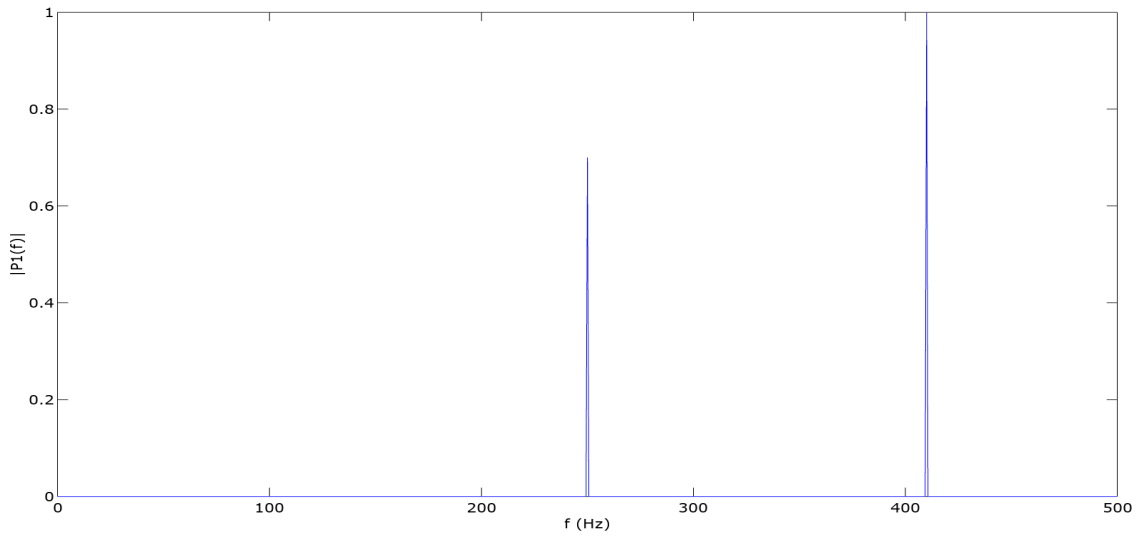


Fig. 3.2 Graph of $\hat{f}(\omega)$ produced with DFT

Unsurprisingly, the Fourier transform of $f(t)$ using FFT produces the same output as DFT. This is shown in Fig. 3.3 *Graph of* $\hat{f}(\omega)$ *with FFT* below.

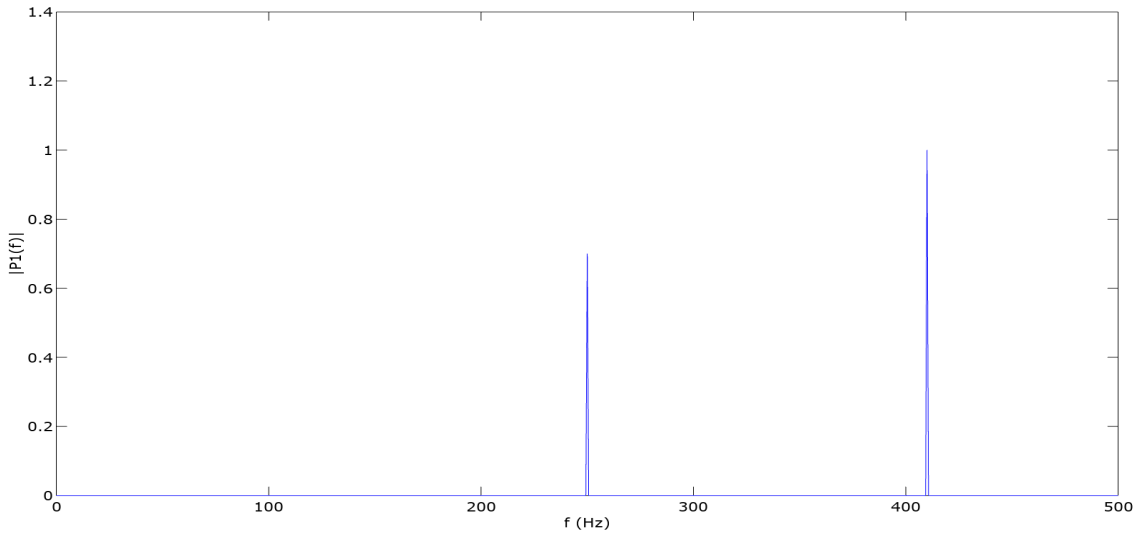


Fig. 3.3 Graph of $\mathcal{F}(f)$ produced with FFT

As we can observe from both Fig. 3.2 and 3.3, $f(t)$ is composed of two simpler sine functions. One has a frequency of 250 Hz and an amplitude of 0.7, while the second has a frequency of 410 Hz and an amplitude of 1. These can be confirmed by the two corresponding peaks in Fig. 3.2 and 3.3. These results confirm that these implementations of DFT and FFT are indeed effective at calculating the Fourier transform.

Example 2. Inverse Fourier transform of $\mathcal{F}(f)$

We demonstrate that we can obtain the original waveform $f(t)$ by applying IFFT after FFT has already been applied. In this case, we use the function

$$f(t) = 2\sin(2\pi * 5 * t) + \sin(2\pi * 3 * t)$$

This function $f(t)$ is shown below in Fig. 3.4.

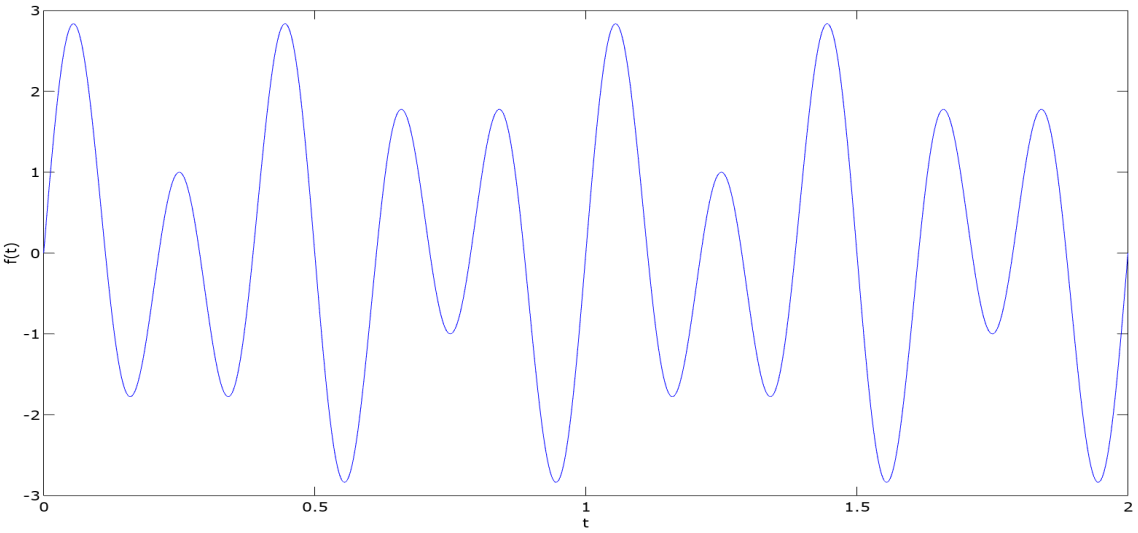


Fig. 3.4 Graph of $f(t) = 2\sin(2\pi * 5 * t) + \sin(2\pi * 3 * t)$

We use the following code to demonstrate this example. As we can observe in the code, FFT and IFFT are successively applied to $f(t)$, and its result is graphed with respect to the original time interval. This result is provided in Fig. 3.5 *Graph of* $\mathcal{F}(\mathcal{F}^{-1}(f(t)))$.

```
% Function and parameters
f = @(t) 2*sin(2*pi*5*t) + sin(2*pi*3*t);
N = 1500;
t = linspace(0, 1, N);

% Fourier transform + inverse Fourier
F = fft(f(t));
ff = ifft(F);

plot(t, ff);
```

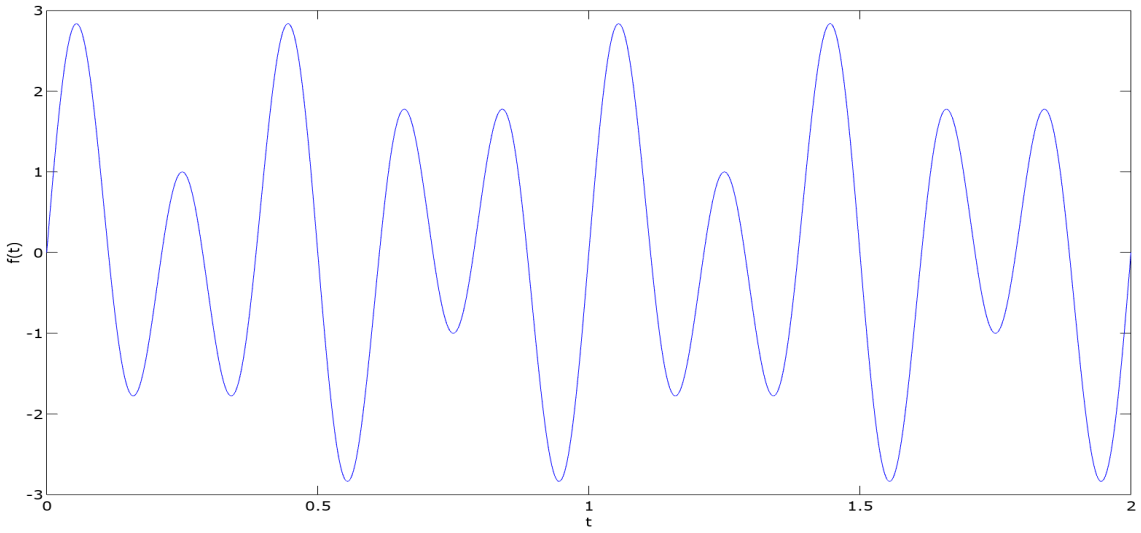


Fig. 3.5 Graph of $\mathcal{F}(\mathcal{F}^{-1}(f(t)))$

As expected, by performing IFFT after FFT, the function obtained is identical to $f(t)$. This result proves that IFFT and FFT are inverse operations.

Example 3. Fourier transform of $f(t) = 0.7\sin(2\pi * 250 * t) + \sin(2\pi * 410 * t)$ with corrupted signal

For a more realistic demonstration, we corrupt the sampled function $f(t)$ used in Example 1 by multiplying each vector component with zero-mean random values. This simulates noise that is often picked up when obtaining waveforms in real-world scenarios such as radio transmission. The corrupted waveform for $f(t)$ is shown in Fig. 3.6 $f(t)$ *corrupted with zero-mean random noise.*

By applying FFT, we are still able to analyze $f(t)$, despite the corrupted signal. This is demonstrated in Fig. 3.7 *Graph of* $\mathcal{F}(f)$ *with a corrupted input vector.*

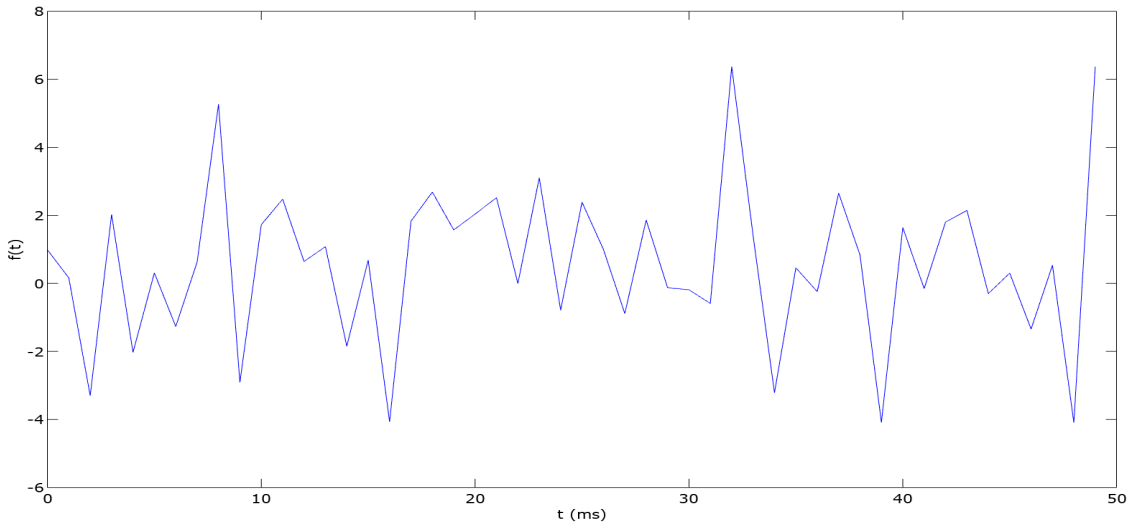


Fig. 3.6 Graph of $f(t)$ corrupted with zero-mean random noise

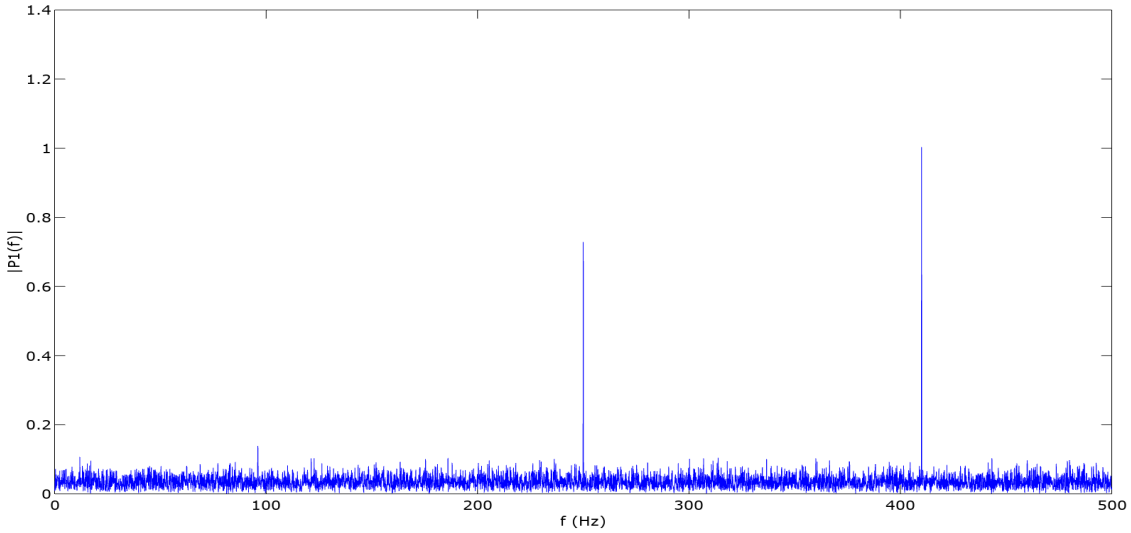


Fig. 3.7 Graph of $\mathcal{F}(f)$ with a corrupted input vector

As shown in Fig. 3.7, we can still distinguish the composition of $f(t)$ from the two prominent peaks that are visible at 250 Hz and 410 Hz. We can also observe that the amplitudes of each peak are not exactly 0.7 and 1, respectively, due to the random noise that was introduced to the function.

4. Performance

An important distinction between DFT and FFT is computational complexity. Assuming N is the length of the sampled vector for a given function, DFT requires $O(N^2)$ operations, while FFT requires $O(N\log_2 N)$ operations. This clearly makes FFT a much more desirable implementation of the Fourier transform.

4.1 Analysis for DFT Computational Complexity

For each element in the input vector $\boldsymbol{f}$, an N-dimensional inner product is calculated. Each inner product consists of N multiplication and N-1 addition operations. Thus, computing an inner product requires O(N) operations. We repeat this procedure N times, which results in a total complexity on the order of $O(N^2)$.

4.2 Analysis for FFT Computational Complexity

One of the key ways FFT optimizes computation is by avoiding redundant calculations. Firstly, we note that for the recombining butterfly process described by Equations (2.8) and (2.9) shown again below, the terms $\hat{f}_{even,n}$ and $\hat{f}_{odd,n}$ are both used twice.

$$\hat{f}_n = \hat{f}_{even,n} + \omega_N^n \hat{f}_{odd,n} \qquad n = 0, \cdots, M-1$$

$$\hat{f}_{n+M} = \hat{f}_{even,n} - \omega_N^n \hat{f}_{odd,n} \qquad n = 0, \cdots, M-1$$

Therefore, once the values for $\hat{f}_{even,n}$ and $\hat{f}_{odd,n}$ are calculated, they are cached in order to prevent recalculations. This is the case for every iteration of Equation (2.8) and (2.9), and greatly saves computational resources. Additionally, we can observe that the value ω_N^n only changes signs and remains the same magnitude. Therefore, ω_N^n is also cached and added or subtracted accordingly.

The calculation of vector $\hat{\boldsymbol{f}}_{n/2}$ takes half of the number of operations required for the calculation of vector $\hat{\boldsymbol{f}}_n$. An $N/2$-point butterfly operation requires $N/2$ multiplication operations. Therefore, we can obtain the recurrence relationship described in Equation (4.1).

$$T(N) = 2T(\frac{N}{2}) + 2(\frac{N}{2}) = 2T(\frac{N}{2}) + N \tag{4.1}$$

Because FFT continuously splits the input vector into halves, we can assume that the term $T(N/2)$ requires $\log_2$N recursive steps before N = 1. For each step, FFT performs O(N) operations. Therefore, FFT has a total time complexity of $O(N\log_2 N)$.

4.3 Comparison

The difference between $O(N^2)$ and $O(N\log_2 N)$ becomes substantial as N becomes large. This is shown in Table 4.1 below.

Table 4.1: Theoretical comparison between N^2 and $N\log_2 N$

N	N^2	$N^2 \times 1$ ns	$N\log_2 N$	$N\log_2 N \times 1$ ns
2^{10}	2^{20}	≈ 1.05 ms	10×2^{10}	≈ 10.2 ns
2^{20}	2^{40}	≈ 18.3 min	20×2^{20}	≈ 21.0 ms
2^{30}	2^{60}	≈ 36.6 yrs	30×2^{30}	≈ 32.2 s

We test the theoretical analysis between DFT and FFT described above by testing the actual time taken by each algorithm to compute a Fourier transform. In order to test the actual computation speeds of both algorithms, we construct N-length vectors filled with random values and utilize the *cputime* function built-in to Matlab to measure the time it takes for both functions to complete their computations. The difference in computation time for both algorithms are shown in Table 4.2.

Table 4.2: Actual comparison between DFT and FFT computation time

N	DFT [s]	FFT [s]
10	$4.01 \cdot 10^{-3}$	$5.50 \cdot 10^{-5}$
100	0.294	$8.80 \cdot 10^{-5}$
1000	29.4	$9.1 \cdot 10^{-3}$
3000	115	$1.32 \cdot 10^{-2}$

As the analysis in Table 4.1 would imply, FFT is much faster than DFT. The data was measured using the code shown below.

```
disp("Beginning tests...")
N = 1000;
X = rand(1, N); % N length random input vector

t_0 = cputime;
DFT(X);
elapsed = cputime - t_0;
disp("DFT time:")
disp(elapsed);

t_0 = cputime;
fft(X);
elapsed = cputime - t_0;
disp("FFT time:")
disp(elapsed);
```

5. Conclusions

Fourier transform is an essential tool for various fields in engineering and science. As this report outlined, numerical methods such as FFT and IFFT allow for the application of the Fourier transform in real-world scenarios where input functions are discretized at finite intervals. This report highlighted the common methods for discrete analysis that exist, as well as implementations and examples to illustrate each concept. This report compared the various implementation details, as well as theoretical and actual computational times for Fourier algorithms.

6. References

[1] Cheever, E. (2015). Introduction to the Fourier Transform. Swarthmore College. Date Accessed: July 29, 2018. URL: http://lpsa.swarthmore.edu/Fourier/Xforms/FXform Intro.html

[2] Dou, X. (2018). Discrete Fourier Transform and Fast Fourier Transform. Waseda University.

[3] Lohne, M. (2017). The Computational Complexity of the Fast Fourier Transform.

[4] Rao, K. R., Kim, D. N., Hwang, J. J. (2011). Fast Fourier transform-algorithms and applications. Springer Science and Business Media.

[5] Sevgi, L. (2007). Numerical Fourier transforms: DFT and FFT.

[6] Wong, M. W. (2011). Discrete fourier analysis (Vol. 5). Springer Science and Business Media.

[7] Matlab Documentation. Fast Fourier Transform. Mathworks. Date Accessed: July 23, 2018. URL: https://www.mathworks.com/help/matlab/ref/fft.html