Fast Fourier Transform

Team *random-theory*
May 1, 2022

1 INTRODUCTION

A Fourier transform is a mathematical transform that decomposes functions depending on space or time into functions depending on spatial frequency or temporal frequency.

The central question that fourier transform is trying to solve is that if we have a signal, can we decompose it into the pure frequencies that make it up? such that adding up all the invidual decompositions gives us the original signal back. Its just like unmixing some colors that have been mixed together or finding the recipe of a given smoothie.

It is one of the most powerful tools in digital signal processing and digital image processing that is used for the the frequency analysis of signals. It is used in a wide range of applications, such as signal interpolation, signal smoothing, image filtering, image reconstruction and image compression. In fact any field of physical science that uses sinusoidal signals, will make use of Fourier series and Fourier transforms. [Press, 2007].

Fast Fourier Transform is one of the most celebrated algorithm of the 20^{th} century. The purpose of this report is to use traditional slow approach used for Fourier Transform, along with the modern Fast alternative, and compare their performance. We wanted to explore the history of Fourier Transform, which we included briefly in this report, and its endless applications. For our project, we applied DFT and FFT on audio conversions from time to frequency domain, and on image compression using some image proecessing techniques.

1.1 FOURIER TRANSFORM OF A FUNCTION

Initially, Fourier Transforms were assumed to be just a concept of physics. However, with the progress in computation, scientists realized that it can be used in mathematical, and later, in computational domain.

It gives us a unique way of viewing any function as the sum of simple sinusoids. Fourier Transform works as a medium between two domains. It is used to convert a sequence of signals from one time domain to frequency domain, but it is not restricted. The units change according to the domain pair in use. If we take time-frequency as our transform pair, we can define the forward and backward Fourier Transform as follows:

$$G(f) = \int_{-\infty}^{\infty} g(t)\, \mathrm{e}^{-2\pi i f t} \tag{1.1}$$

In addition, g can be obtained from G via the inverse Fourier Transform:

$$g(t) = \int_{-\infty}^{\infty} G(F)\, \mathrm{e}^{2\pi i f t} \tag{1.2}$$

1.1.1 Why use Complex Numbers?

The reason for using Complex Numbers is to present the formula in an elegant way. But more importantly, they are used to preserve the properties of the signals, such as phase, during conversion. In time domain, the sample values lie on x-axis, while y-axis holds the amplitude. However, in frequency domain, we also need to store the phase of a signal, thus the real part is used to store the amplitude, while the complex part stores the phase.

2 Discrete Fourier Transform

Discrete Fourier Transform, as the name suggests, is the discretized version of the continuous Fourier Transform. We take the discrete sum of a finite sequence of signals such that we modify the above integral equations to Fourier transform the discrete finite samples. Finding the discrete sum of the continuous equations above gives us:

$$G_k = \sum_{n=0}^{N-1} g_n \cdot e^{\frac{-i2\pi}{N} \cdot kn} \tag{2.1}$$

Raw implementation of this formula is called slow or naive Discrete Transform, and it works in $O(n^2)$ time. The detailed analysis is done in section 6.1.

2.1 Fourier Matrix

An N-point DFT is expressed as the multiplication $X = Wx$ where x is the original input signal and W is the N-by-N square DFT matrix.

$$\omega = e^{\frac{-2\pi i}{N}} \tag{2.2}$$

$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{2.3}$$

$$W = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix} \tag{2.4}$$

$$DFT(x) = X = x \cdot W \tag{2.5}$$

```python
def compute_dft(x):
    N = len(x)
    X = [0 for _ in range(N)]
    for i in range(N):
        for j in range((N)):
            omega = np.exp(complex(0, -2) * np.pi / N)
            X[i] += x[j] * omega**(i * j)
    return X
```

Listing 1: A python program to compute DFT of a given signal

3 Fast Fourier Transform

Till 20^{th} scientists believed that we can not do better than $O(n^2)$, but in 1942, Danielson and Lanczos proposed their lemma which proved that the DFTs can be implemented via Divide and Conquer.

The lemma suggests that instead of applying the DFT on the whole sequence, we can recursively divide it into halves. The elements of even indexes can be separated from the odd, giving us 2 sequence of equal size, given our N is in powers of 2. We apply DFT on both sequences separately, and combine them with some "twiddle factor" W. We exploit the symmetries of sin and $cosine$ functions to obtain a much faster $D\&C$ algorithm with highly reduced time complexity. The mathematical proof is done is section 5.2.

The following equations may explain the idea behind FFT:

$$F_k = \sum_{j=0}^{N-1} e^{-2\pi i j k/N} f_j \tag{3.1}$$

$$= \sum_{j=0}^{N/2-1} e^{-2\pi i k(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{-2\pi i k(2j+1)/N} f_{2j+1} \tag{3.2}$$

$$= \sum_{j=0}^{N/2-1} e^{-2\pi i k j/(N/2)} f_{2j} + e^{-2\pi i k/N} \sum_{j=0}^{N/2-1} e^{-2\pi i k j/(N/2)} f_{2j+1} \tag{3.3}$$

The FFT employs a divide-and-conquer strategy to divide the DFT computation into odd and even parts. These parts are very similar to the original DFT, and we only need to perform half the computation. This would convert our O(n^2) algorithm to O(m^2) where m is n/2. As

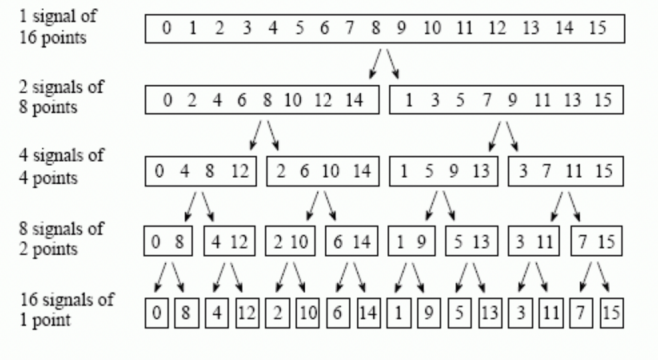
long as m is divisble by 2, we can recursively repeat this till our running time is O($nlogn$). The code is as follows:

```
def FFT(X):
    N = len(X)
    if N <= 1: return X
    omega = np.exp(complex(0, -2 * np.pi) / N)

    X_even = FFT(np.array(X[0:N:2]))
    X_odd = FFT(np.array(X[1:N:2]))

    X = [0 for _ in range(N)]

    for j in range(0, N // 2):
        X[j] = X_even[j] + omega**j * X_odd[j]
        X[j + N // 2] = X_even[j] - omega**j * X_odd[j]
    return X
```

Listing 2: A python program to compute DFT of a given signal using the fast fourier transform

After recursively dividing the input, P, we only need to run m, or n/2 times. This is what leads to the recurrence of T(n) = 2T(n/2) + O(n). This is only possible because of the symmetric nature of the DFT.

We can summarize the above computation by this example below.



4 Visualization

4.1 Audio processing

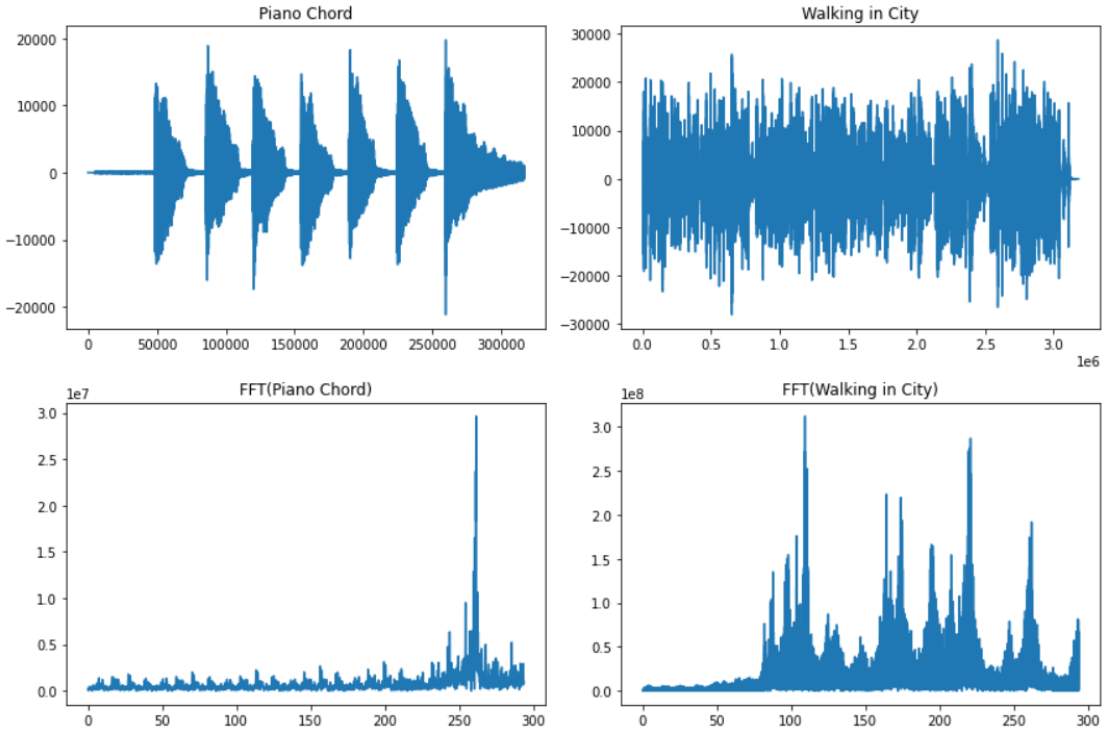
To show the workings of the FFT, we applied it in one of it's most popular applications: signal processing. We use 2 audio files to show the main functionality the FFT allows.

- Piano chord being played repeatedly.
- Background noise while walking in a city.

Looking at the first graph below, we see the signal for the piano chord being played repeatedly. We can physically see the chord being repeated in this signal. In the second graph, we see the

signal for the noises in the city. It is more noisy and there are alot more noises mixed in.

Essentially, the FFT converts our time domain signal into the frequency domain. It unmixes the frequencies mixed in to the final sound to seperate distinct sounds. In signal processing, this would allow us to identify background noises, or noises to isolate. The FFT identifies the piano chord and we can see this as a peak on the graph. For the second FFT graph, there are alot more noises in the final audio track. These different noises are identified and the most frequent ones are displayed as the highest peaks.



4.2 Image processing

We can take any digital image, get the discrete pixel values, and transform it into a frequency domain. We essentially preserve the information of the image but just represent it in another domain.

We took inspiration and helper functions from `Github - Fast Fourier Transform` and applied a 2D fourier transform on an image. [Meharizghi, 2017].

We can apply fast fourier transform to an image and shift the zero-frequency component to the center of the spectrum such that the centre part contains the low frequency components where as other contains high frequency. We can also reverse back the image from the frequency domain by using inverse fourier transformation. Low frequencies in images are pixel values that are changing slowly (smooth areas). Whereas high frequencies are pixels whose values are changing fast. In the figure below, we first generated an image, computed the fast fourier

transform and then applied the inverse fast fourier transformation to get back the original image. The generated image has a white square in the middle from which we can see that as the white square decreases in size the low frequency component in the corresponding centered FFT image is larger.

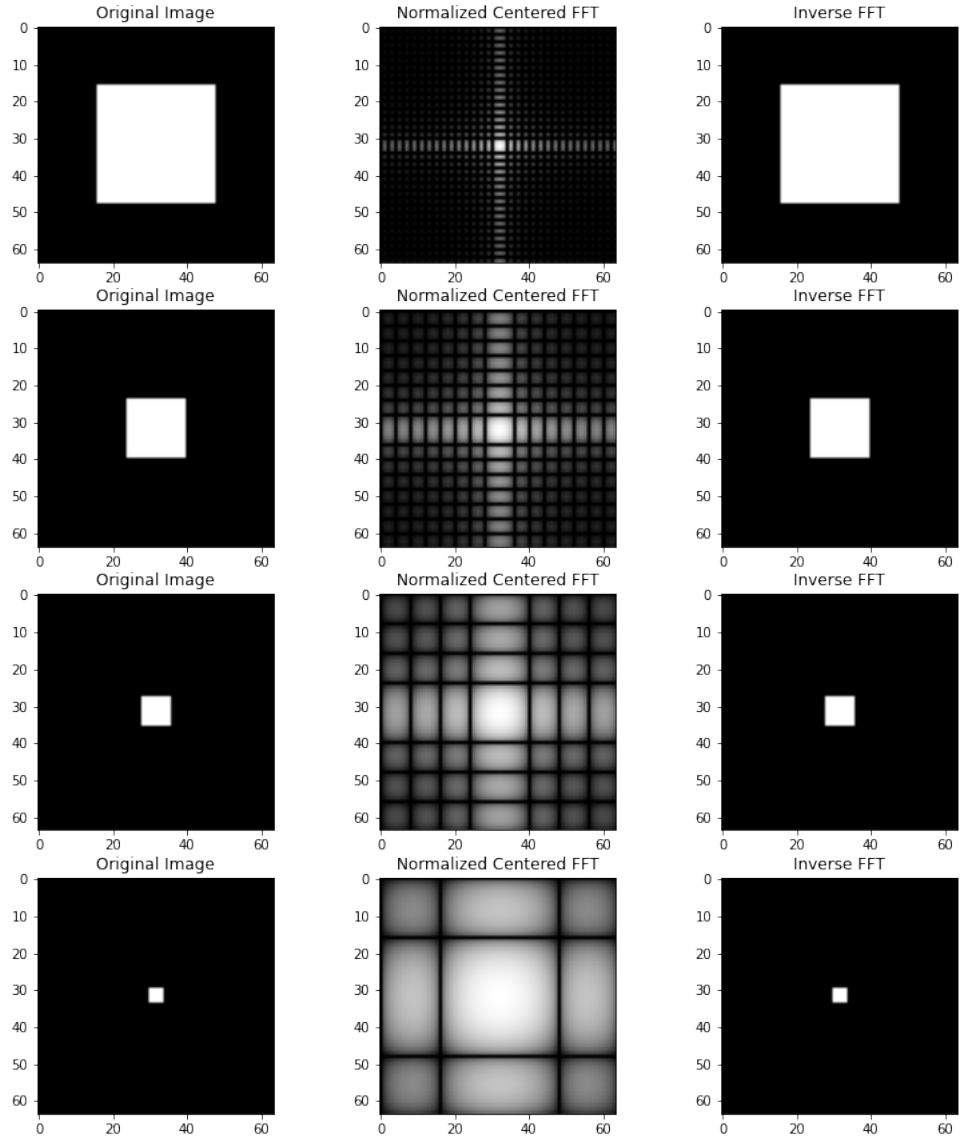


Figure 4.1: Fast Fourier Visualization on a 64 x 64 image

5 Theoretical runtime Analysis

5.1 Discrete Fourier Transform

To apply naive Discrete Fourier Transform means implementing the dicrete sum over the whole sequence. The formula we get is as follows,

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}$$

From the above formula, we can see that the code implementation will use two loops. The outer loop will be for the i^{th} term, and run for all N elements. The inner loop will be for k^{th} term, and run for all the N elements. The total cost for the nested loops will be,

$$T(n) = n.n$$

$$T(n) \leq n^2 \equiv O(n^2)$$

5.2 Danielson And Lanczos Lemma

To understand the recursion, we first need to see the working of Danielson-Lanczo lemma. As the lemma suggests, we can first divide the sequence into two parts according to even and odd indexes. We do this recursively and apply DFT on the halved sequence. [Wolberg, 1988] The proof of the lemma is as follows,

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

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

$$F_n = \sum_{k=0}^{\frac{N}{2}-1} f_{2k} e^{-2\pi i k n/\frac{N}{2}} + W^n \sum_{k=0}^{\frac{N}{2}-1} f_{2k+1} e^{-2\pi i k n/\frac{N}{2}} \tag{3}$$

$$F_n = F_n^e + W^n F_n^o \tag{4}$$

In the above derivation, equation (1) is the simple formula of DFT. In step (2), we half the original sequence, and apply DFT on even ($2k$) and odd ($2k$+1) sequences. In step (3), we express both even and odd parts from step (2) in a DFT form for $\frac{N}{2}$ elements, instead of specifying even and odd term through k. Then we add them with some "twiddle factor" W, which is an n^{th} complex root of unity.

5.3 Fast Fourier Transform

Using the Danielson-Lanczos lemma, we know that during the recursion, our input sequence is divided by the factor of 2. We also know that the combining cost $C(n)$ is $O(n)$. [Tanimoto, 2016] Following this description, we can write the following recurrence,

$$T_n = 2T(\frac{n}{2}) + n$$

We can use various techniques to solve this recurrence. Here, we will unfold the terms,

$$T_{\frac{n}{2}} = 2T(\frac{n}{4}) \ + \ \frac{n}{2}$$

$$T_{\frac{n}{4}} = 2T(\frac{n}{8}) \ + \ \frac{n}{4}$$

Applying back substituition, we get,

$$T_{\frac{n}{2}} = 2[2T(\frac{n}{8}) + \frac{n}{4}] + \frac{n}{2}$$

$$T_{\frac{n}{2}} = 4T(\frac{n}{8}) + \frac{n}{2} + \frac{n}{2} = 2^2T(\frac{n}{2^3}) + n$$

$$T_n = 2[2^2T(\frac{n}{2^3}) + n] + n$$

$$T_n = 2^3T(\frac{n}{2^3}) + 2n + n = 2^3T(\frac{n}{2^3}) + 3n$$

We can generalize the above recurrence for k terms,

$$T_k = 2^kT(\frac{n}{2^k}) + kn$$

To reduce $T(\frac{n}{2^k})$ to the base case, we equate it to 1,

$$\frac{n}{2^k} = 1$$

$$n{=}2^k$$

Taking log on sides,

$$log(n) = k.log(2) \equiv log(n) = k$$

Substituiting the value of k, we get

$$T_k = 2^{log(n)}T(1) + log(n).n$$

$$T_k = n^{log(2)}(1) + n.log(n)$$

$$T_k = n + n.log(n)$$

$$T_k \leq n.log(n) \equiv O(n.log(n))$$

6 Experimental runtime Analysis

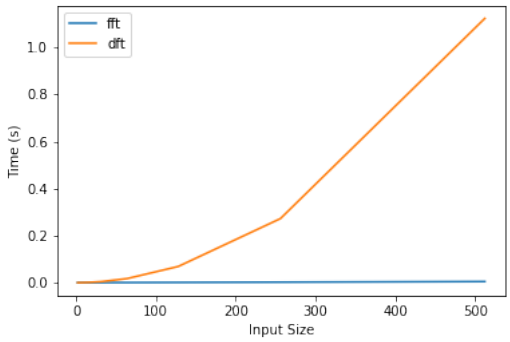


Figure 6.1: Runtime comparison based on 1D signals

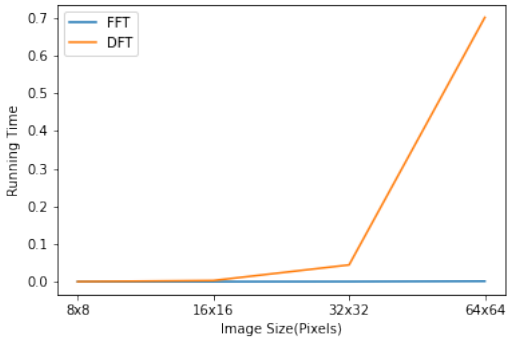


Figure 6.2: Runtime comparison based on images

7 Conclusion

After reading the literature, and experimenting it ourselves, we can conclude that Fast Fourier Transform is indeed a remarkable algorithm. It not only works in exponentially lesser time, but it also caters to huge input sizes with the same efficiency and performance.

References

[Meharizghi, 2017] Meharizghi, T. (2017). Digital-image-processing/02-implementing-fast-fourier-transform-using-python.ipynb at master · tesfagabir/digital-image-processing.

[Press, 2007] Press, W. H. (2007). *Numerical recipes: The art of scientific computing*. Cambridge University Press, 3 edition.

[Tanimoto, 2016] Tanimoto, S. (2016). Divide and conquer: The fast fourier transform.

[Wolberg, 1988] Wolberg, G. (1988). Fast fourier transforms: A review. *Department of Computer Science Columbia University*.