Design & Implementation of FFT

Abstract:

Fast Fourier Transform is a computationally efficient algorithm[5] used to compute the "Discrete Fourier Transform" values of a sequence by Matrix Multiplication using Divide and Conquer Paradigm approach. This survey paper elaborates in detail about the underlying algorithms used to achieve it making it to reduce the Asymptotic analysis from $O(n^2)$ to $O(n\ logn)$. We further understand the nuances involved in the scenarios for FFT like : FFT2, D&C approaches, De-noising, etc.

Introduction:

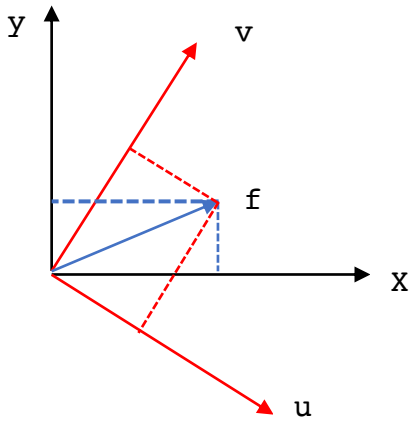
Before diving into the topic, we shall understand the history of Fourier Analysis in first place. Theory behind it states that, it is introduced as the co-ordinate transformation as cosines and sines as it's basis, where we can approximate the discrete values in the defined intervals using the summation of the sine and cosines of different frequencies. This concept is used to solve the heat equation of the metals. Later, it has become the basis for many of the scenarios like Audio & Image compression, Signal processing and many more. Thus, below is a 2-Dimensional vector converted using the Fourier series.

$$f(x) = \frac{A_0}{2} + \sum_{k=1}^{\infty} (A_k \cos(kx) + B_k \sin(kx))$$

From, the above equation, we have converted the function f(x) to polar co-ordinates with co-efficients A_k & B_k.

$$A_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(kx)\, dx$$

$$B_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(kx)\, dx$$



The above co-effiecients are considered the projections of the vectors on the polar-coordinates. The variable k depends on the number of the frequencies combined together to figure out the approximation values on a defined interval. Extrapolating the same idea to the extended intervals using the Fourier Transformations. This gives us to the new terminologies as below. This equation gives rise to the Fourier Transformation extending the periodicity to (-∞, ∞).

$$f(x) = \int_{-\infty}^{\infty} \frac{1}{2\pi} \int_{-\infty}^{\infty} f(\xi) e^{-i\omega\xi}\, d\xi\, e^{-i\omega x}\, d\omega$$

Discrete Fourier Transform:

DFT, can be pictured as a mathematical translation or transformation used to express in the form of a big matrix multiplication to evaluate the approximation value for the function of time and space to Fourier co-effiecients in the form of Complex Numbers.

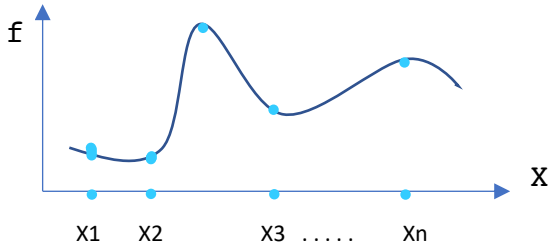


Fig 1.1

From the Figure1.1, we have a list of x points {x1, x2, x3,…., xn} where the values of the function f as {f1, f2, f3,…., fn}. These values are drawn out using different values to frequencies of cosine and sines and with the help of the matrix multiplication from the ω – the functional frequency as below.

$$f'_k = \sum_{j=0}^{n-1} f_j \, e^{-i2\pi jk/n} \qquad f_k = \frac{1}{n}\sum_{j=0}^{n-1} f'_k \, e^{i2\pi jk/n}$$

Where $\omega = e^{2\pi i/n}$ and the '*i*' is an imaginary value $\sqrt{-1}$, making it as the below matrix.

$$\begin{bmatrix} f'_0 \\ f'_1 \\ . \\ . \\ f'_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & . & . & 1 \\ 1 & \omega_n & \omega_n^2 & . & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & . & \omega_n^{2(n-1)} \\ . & . & . & . & . \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & . & \omega_n^{(n-1)^2} \end{bmatrix} . \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ f_n \end{bmatrix}$$

Fourier Co-Efficients *DFT Matrix*

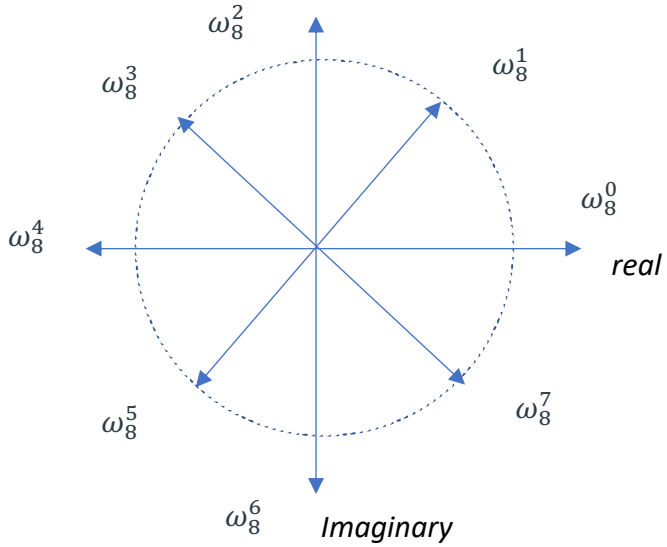
Thus using the above mentioned Matrix multiplication, we can obtain the Fourier transformation of the given function when we perform the multiplication with the frequencies obtained from the function itself and the general frequency values as per the index positions in the above equations. This turns out to the Discrete Fourier Matrix[6], which is coloured in yellow. The main agenda is to obtain the Fourier co-efficients which are coloured in blue. Each of the value represents a complex number, magnitude of the frequency applied for the cosine and sine function and the phase tells us about the angle with which cosine and sine values are displaced to obtain the approximation of the function value.

The process to compute the Fourier co-efficients is much more complex operation, which eventually takes long time when the value of *n* goes long enough to approximate the values of the function. Thus, there is a need to have a mathematically efficient and computationally feasible solution to extract it with keeping the larger size of the values for the possible frequencies in mind. Let us understand a little bit more on how the implementation of an algorithm to extract the matrix multiplication with less complexity.

Inorder to compute the DFT Matrix, we can obtain it by solving each and every "ω", making it to use two nested loops to iterate over all the possible values for every row and column producing a complexity of $O(n^2)$.

```
for row = 1 to rows
   for column = 1 to columns
        dft(row, column) = ω^((row−1)∗(column−1))
```

We can evaluate it efficiently considering several properties of "ω" and can reduce the following complexity to $O(n\ logn)$. It involves the understanding of the complex numbers in the polar co-ordinates, similar to the rectangular axis we have axis for the real and the imaginary numbers and the values are represented in the form of $re^{-i\theta}$ representing the magnitude and the phase between the real and the complex numbers. Also, the roots of unity (a number which is multiplied 'n' number of times with itself to get the value 1) helps us in evaluating the DFT with ease. From the above context, we understand that when we multiply the same number with itself if several interesting properties are obtained as below.
Let us consider an example[10] of eight roots of unity in the Polar co-ordinates.



We see from the above DFT matrix for a given general frequency we need to evaluate the higher order power values of "ω" and this forms the roots of unity as { ω_8^0, ω_8^1, ω_8^2, ω_8^3, ω_8^4, ω_8^5, ω_8^6, ω_8^7 } as {1,$\frac{1+i}{\sqrt{2}}$, *i*, $\frac{-1+i}{\sqrt{2}}$, -1, $\frac{-1-i}{\sqrt{2}}$, -*i*, $\frac{1-i}{\sqrt{2}}$}. The values suggest some properties as below and the DFT matrix framed using the unity values for the ω.

$$\begin{bmatrix} \omega_8^0 & \omega_8^0 & \omega_8^0 & \omega_8^0 & \omega_8^0 & \omega_8^0 & \omega_8^0 & \omega_8^0 \\ \omega_8^0 & \omega_8^1 & \omega_8^2 & \omega_8^3, & \omega_8^4 & \omega_8^5 & \omega_8^6 & \omega_8^7 \\ \omega_8^0 & \omega_8^2 & \omega_8^4 & \omega_8^6 & \omega_8^0 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ \omega_8^0 & \omega_8^3 & \omega_8^6 & \omega_8^1 & \omega_8^4 & \omega_8^7 & \omega_8^2 & \omega_8^5 \\ \omega_8^0 & \omega_8^4 & \omega_8^0 & \omega_8^4 & \omega_8^0 & \omega_8^4 & \omega_8^0 & \omega_8^4 \\ \omega_8^0 & \omega_8^5 & \omega_8^2 & \omega_8^7 & \omega_8^4 & \omega_8^1 & \omega_8^6 & \omega_8^3 \\ \omega_8^0 & \omega_8^6 & \omega_8^4 & \omega_8^2 & \omega_8^0 & \omega_8^6 & \omega_8^4 & \omega_8^2 \\ \omega_8^0 & \omega_8^7 & \omega_8^6 & \omega_8^5 & \omega_8^4 & \omega_8^3 & \omega_8^2 & \omega_8^1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \frac{1+i}{\sqrt{2}} & i & \frac{-1+i}{\sqrt{2}} & -1 & \frac{-1-i}{\sqrt{2}} & -i & \frac{1-i}{\sqrt{2}} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & \frac{-1+i}{\sqrt{2}} & -i & \frac{1+i}{\sqrt{2}} & -1 & \frac{1-i}{\sqrt{2}} & i & \frac{-1-i}{\sqrt{2}} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \frac{-1-i}{\sqrt{2}} & i & \frac{1-i}{\sqrt{2}} & -1 & \frac{1+i}{\sqrt{2}} & -i & \frac{-1+i}{\sqrt{2}} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & \frac{1-i}{\sqrt{2}} & -i & \frac{-1-i}{\sqrt{2}} & -1 & \frac{-1+i}{\sqrt{2}} & i & \frac{1+i}{\sqrt{2}} \end{bmatrix}$$

Properties:

1. $\omega_8^0\ or\ \omega_8^8$ are always equals to 1.
2. $\omega_n^k * \omega_n^l = \omega_n^{(k+l)} = \omega_n^{((k+l)modn)}$

Lemmas:

1. *Lemma for Cancellation:* For any positive integers *n, p* ≥ 0 and *l* > 0 then the value of the "ω" with the integral multiple can be converted to its least multiple as $\omega_{np}^{lp} = \omega_n^l$.

Proof: Let, $\omega = e^{2\pi i/n}$ and we need to compute ω_{np}^{lp}

$\omega_{np}^{lp} = (e^{\frac{2\pi i}{np}})^{lp} = (e^{\frac{2\pi i}{np}})^{lp} = e^{(2\pi il/n)} = \omega_n^l$

Using the properties of the $(e^a)^b = e^{ab}$ the common powers "p" is cancelled out and it results in the value "ω_n^l". Using this lemma we can extract the several corollary as $(\omega^{2k})^k = -1 \; and \; (\omega^{12})^6 = (\omega^2)$. This lemma helps us to easily find the higher order powers useful for computing the DFT matrix.

2. *Lemma for Halving:* The roots of every n where it is positive and even number, it's nth root of unity is same as the (n/2)th root of unity.

Proof: Let us evaluate the kth root of "$(\omega_n^k)^2$"

$(\omega_n^k)^2 = (e^{\frac{2\pi ik}{n}})^2 = (e^{\frac{2\pi i*2}{n}})^k = (e^{\frac{2\pi i}{(\frac{n}{2})}})^k = \omega_{n/2}^k$

With this lemma, the higher order powers of "ω" with the increase in the number of the frequencies less than the half of the total need be computed as they are equal to the half of the size of the same period value.

Fast Fourier Transform:

Generally, relying on DFT for the approximation is way costly as it requires to compute n^2 matrix vector multiplications and an efficient approach is to implement FFT to obtain the DFT values for the discrete values of function in $O(n\,logn)$. FFT has earned it's place in various domains for its robust nature to evaluate the complex number of values and has prominent scope in data processing and high-performance scientific computation. The value is drastically reduced from 'n' to 'log(n)'. It is very important to note with a sample of 1 Billion sample values, log(n) gives a value 9 which is way less than the prior value of 10^9. For example, to compress a sample audio output sampled at 40Khz which is 4 * 10^5. Using DFT to compress we need to perform about 10^{11} *(100 Billion)* multiplications, whereas the FFT takes $10^6(1\,Million)$ multiplications which is 10,000 faster than DFT making it feasible to transfer the content with ease and in less time.

This algorithm is structured to use "*Divide and Conquer*" Paradigm approach (assuming that "n" is a power of 2) into Even and Odd sequences of the discrete values of the function, when the input values are less not making the power of 2, we pad zeros to it to make it equivalent to the some power of 2. DFT for prime numbers requires re-indexing and is implemented using [13]Prime-Factor FFT Algorithm(PFT).

In FFT, instead of solving the DFT Matrix directly we try to separate out the even and odd matrix value ordinal positions and label them as f_{even} & f_{odd} by shuffling the rows and stack them to obtain as below. Let us perform the example for $n = 2^{11} = 2048$ discrete points of the function we evaluate the following.

$$f'_k = F_{2048} \cdot f$$

$$f'_k = \begin{bmatrix} I_{1024} & -D_{1024} \\ I_{1024} & -D_{1024} \end{bmatrix} \begin{bmatrix} F_{1024} & 0 \\ 0 & F_{1024} \end{bmatrix} \begin{bmatrix} f_{even} \\ f_{odd} \end{bmatrix}$$

$$D_{1024} = \begin{bmatrix} 1 & 0 & . & 0 \\ 0 & \omega^1 & . & 0 \\ . & . & \omega^{1023} & . \\ 0 & 0 & . & \omega^{1024} \end{bmatrix}$$

Where, upon segregating the even and odd rows from the discrete values and reshuffling the rows we obtain the equivalent DFT matrix as above with the combination of two matrices with "I_{1024}" as Identity matrix and "D_{1024}" as the Diagonal matrix with the general frequency values "ω" in the diagonal places. Computation of the first matrix is simple and easy and multiplying with the diagonal matrix provides the necessary result.

The key takeaway is that, in order to compute the F_{1024}, we use the same Divide and Conquer strategy and rely on F_{512}. Thus, forming an iterative process till we reach the smallest $(2x2)$ matrix. We use the DFT to evaluate it and use the result to calculate the higher order values of "F".

$$F_{2048} \rightarrow F_{1024} \rightarrow F_{512} \rightarrow F_{256} \rightarrow . \ldots \rightarrow F_4 \rightarrow F_2$$

Recursion Tree: Based on the recursion we have developed with the DFT matrix, here is the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$

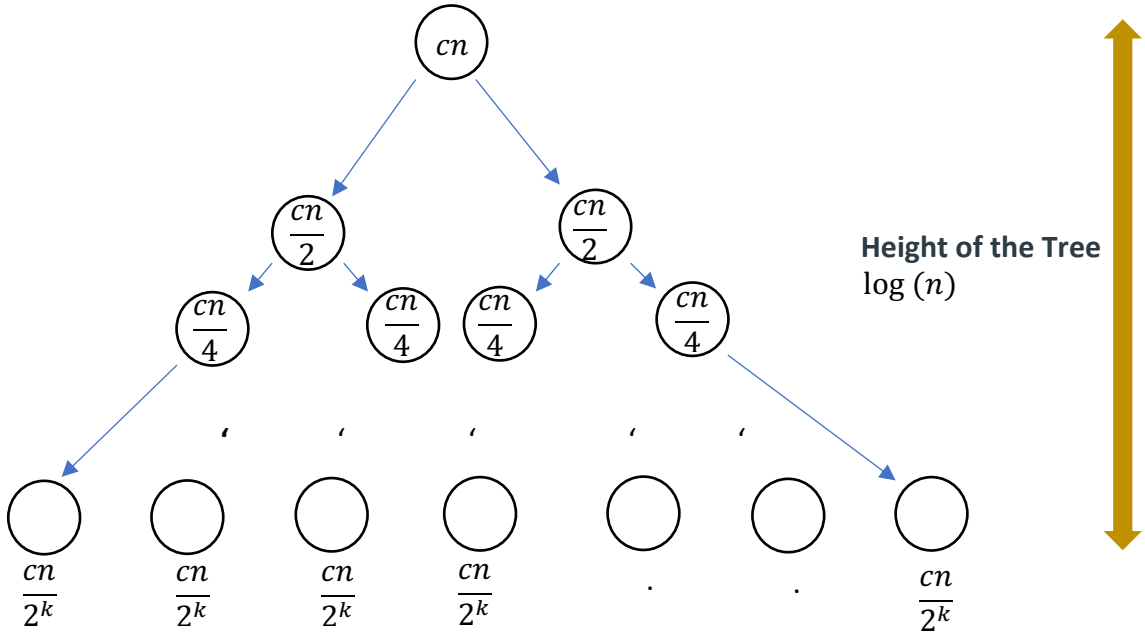


Figure 1.2

Maximum height of the graph can grow is up to $h = \log(n)$, thus the last term of the graph which is $\left\{\frac{cn}{2^{\log_2 n}}\right\}$. When we split the DFT matrix *once* into two parts each of (n/2) elements (each for even and odd elements), we obtain the final result with the multiplication of the DFT result from even group and odd group and we have two such DFT groups and finally reducing all the results by adding to obtain the final result.

$$Number\ of\ Operations\ for\ \left\{\frac{n}{2}\right\}\ Elements = 2 * \left\{\frac{n}{2}\right\} * \left\{\frac{n}{2}\right\} + N$$

From the algorithm of DFT, it takes $O(n^2)$ to compute the elements represented in Blue colour, two such (n/2) groups represented in green colour. Finally, adding the result to obtain DFT for 'n' elements. Expanding the same results $\left\{\frac{n^2}{2} + n\right\} \sim O(n^2)$ which have considerably little low when compare with $O(n^2)$. When repeated to divide the DFT elements until it is of size 2. Continuing the same till we reach the end of the graph in Figure 1.2, for $height(h)$ times results:

$$\left\{\frac{n^2}{2^h} + nh\right\} = \left\{\frac{n^2}{2^{\log_2 n}} + n\log_2(n)\right\} = \frac{n^2}{n} + n\log_2(n) = n + n\log_2(n)$$

Dividing the DFT elements until the depth of $\log_2(n)$ results in the reduction of the time complexity from $O(n^2) \rightarrow O(n\ logn)$, which is a great improvement to improve the efficiency and reduce time to perform the transformations.

Pseudo Code: Below is the pseudo code[9] for the recursive fft algorithm which takes an array "arr" as input and starts with a base case check for unit length to return the same element. We segregate the input array to have even and odd ordinal positional elements into $arr^0\ and\ arr^1$ respectively for performing to

separate DFT computations and combine them once done(Divide and Conquer). Upon, recursive calls to itself and computing the Fourier co-effiecients it return the output column vector "output" for the input vector "arr".

// Recursive Approach to FFT

```
RFFT(arr)
 array_length = arr.length

 if n equals 1 then return a

 ω_n = e^(2πi/n)
 ω = 1

 // gathering all the even elements
 arr^0 = [arr_0, a_rr2, arr_4, …. , arr_(n−2)]

 // gathering all the odd elements
 arr^1 = [arr_1, arr_3, arr_5, …. , arr_(n−1)]

 output^0 = RFFT(arr^0)
 output^1 = RFFT(arr^1)

 for k =0 to (n/2-1)
       // for the first half of the elements
       output_k = output^0 + ω * output_k^1

       // for the second half of the elements
       output_(k+(n/2)) = output^0 − ω * output_k^1

       ω = ωω_n
return output
```

The implementation of the above algorithm runs in $O(logn)$ time. It can be implemented via an *Iterative Approach*. Indeed it removes the duplicate computation of the same value twice "$output_k^1$", we can implement it once and store it in a dummy variable and reuse it whenever needed. In order to perform on the input array we need to fetch the leaves of the graph in Figure 1.2, thus enables us to iteratively perform the DFT and obtain the results. The process of fetching the leaves in the order that it will be accumulated is obtained with the logic "bit-reversal" arrangement.

Let us understand it with an example, $\{a_0, a_2, a_3, a_4, a_5, a_6, a_7\}$ and these are input to the DFT. Let us find the order in which they will be available when separated into even and odd groups as $\{a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7\}$ these are obtained when we use the bit-reversal logic.

Array Element	Binary Value	Bit-Reversal	Number from Rev
a_0	000	000	1
a_4	100	001	2
a_2	010	010	3
a_6	110	011	4
a_1	001	100	5
a_5	101	101	6
a_3	011	110	7
a_7	111	111	8

Table 1.1

The values obtained in the leaves are framed by taking the reverse order of the elements in the array ordinal positions. Below is the Pseudo code for the same. Reverse() is used to get the $logn$ bit number when its binary order is changed.

```
// Iterative FFT Algorithm

        IFFT(arr)
        bit_reverse(arr, Arr)
        array_length = arr.length

        for n = 1 to logn
                twiddle_factor = 2^n
                ω_twiddle_factor = e^(2πi/twiddle_factor)

                for arr_i= 0 to (array_length -1)/twiddle_factor
                        ω = 1

                        for arr_j = 0 to (twiddle_factor/2)-1

                                temp1 = ω * Arr[arr_i + arr_j + twiddle_factor/2]
                                temp2 = Arr[arr_i + arr_j]

                                Arr[arr_i + arr_j] = temp1 + temp2
                                Arr[arr_i + arr_j + twiddle_factor/2] = temp2 - temp1

                                ω = ω * ω_twiddle_factor
        return Arr


//Getting the position of the leaves for DFT.

        bit_Reverse(arr, Arr)
          n = arr.length
          for k =0 to n-1
                Arr[reverse(k)] = a_k
```

Bo the algorithms perform in $O(logn)$ time, Iterative approach maintain the lower constant value which is hidden in the Asymptotic notation than the Recursive algorithm also eliminates the re computation of the same value.

Applications of FFT[7]:

1. With the advent of FFT, a reliable solution to process data in less time than DFT has become a key algorithm in the field of Audio & Image compression. When FFT performed on the input elements, we can represent it as the transformation of a vector input to complex numbers output with few of them to have higher values (1%) and rest of them are very close to negligible(99%). Using the higher Fourier co-efficient values we can inverse Fourier transform to obtain the original input. Enabling to have 100% compression, which is has become the whole and sole principle for the Digital media processing.
2. For compressing a 2D image, we first apply FFT to the rows of the image and then FFT to the columns (irrespective of the order) produces the Fourier co-efficients for an image, generally termed as FFT2[11] Algorithm. Rest is up to us to perform the transmission of the image, compression, denoising the image with the consideration of a certain threshold values to eliminate and reduce the size enabling the compression of an image.
3. Understanding the Optical Power Intensity[12] to recreate and evaluate the pixel intensity.
4. It can be used to perform the multiplication of polynomials.

Conclusion:

The results of this survey paper on FFT has thrown some light in understanding the nuances involved and the logic behind the scenes to perform several operations like compression, Inverse transform, denoising etc. Trade-off[4][8] with DFT in time by a huge factor aided FFT to be used in several areas. Several futuristic advancements like wavelet transformations is introduced as an improvement. This is a real-world scenario where implementation of the right paradigm (D&C) in different approaches made a reliable and an efficient solution with $O(logn)$ time complexity. Based on the requirement many variants of FFT are implemented like 'Inverse Transform, 'Hermite Transform', 'n-Dimensional Transform', '2-Dimensional Transform'. NumPy has taken over the functionality and designed a module for varieties of FFT[2] under *"numpy/fft/fft"* and also, several free-software libraries such as FFTW3[3] are created to serve the purpose.

References:

[1]. https://github.com/numpy/numpy/tree/v1.22.0/numpy/fft
[2]. https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html
[3]. https://ieeexplore.ieee.org/document/1386650
[4]. http://www.cs.tau.ac.il/~stoledo/Bib/Pubs/pp97-fft.pdf
[5]. https://en.wikipedia.org/wiki/Fast_Fourier_transform
[6]. https://www.maths.ed.ac.uk/~ateckent/vacation_reports/summer_project_gillian_smith.pdf
[7]. https://w.astro.berkeley.edu/~jrg/ngst/fft/fft.html
[8]. https://whatmaster.com/difference-between-fft-and-dft/
[9]. http://robotics.itee.uq.edu.au/~elec3004/ebooks/CLRS%20(2nd%20Ed)%20-%20Chapter%2030.FFT.pdf
[10]. https://www.cs.fsu.edu/~burmeste/slideshow/chapter32/32-2.html
[11]. http://www.databookuw.com/page-2/page-21/
[12]. https://medium.com/optalysys/higher-numerical-precision-for-optical-fourier-transforms-de515f575e76
[13]. https://en.wikipedia.org/wiki/Prime-factor_FFT_algorithm