Bangladesh University of Engineering and Technology

CSE 300

Report on

Fast Fourier Transform (FFT): A Revolutionizing Algorithm

Authors:

Tanvirul Islam Turad (2005011)
Tanvir Hossain (2005014)
MD. Jakaria Hossain (2005026)

Date: March 9, 2024

Contents

Chapter 1

Introduction

1.1 Introduction

The Fast Fourier Transform (FFT) is among the most important algorithms in applied and engineering mathematics and in computer science because this algorithm allows us to multiply two polynomials of length n in $O(n \log n)$ time, which is better than the trivial multiplication which takes $O(n^2)$ time.In the similar way two number of length n can be multiplied by FFT in $O(n \log n)$ time.

In 1805, Karl Fredrich Gauss developed a method for fast calculation of Discrete Fourier Transform (DFT).[12] But he never published it. Later on 1965, James Cooley and John Tukey published a very similar method. So, the credit of discovering the FFT is attributed to Cooley and Tukey. In fact the FFT has been discovered repeatedly before, but the importance of it was not understood before the inventions of modern computers.

Actually, the Fast Fourier Transform (FFT) is an algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. It drastically reduces the computational complexity of computing DFT, making it feasible for real-time processing.

1.2 Motivation

FFT is one of the top 10 algorithms of 20^{th} century included by the IEEE magazine *Computing in Science and Engineering*. FFT analyzes signals in the frequency domain to understand their composition.
Limitations of the Discrete Fourier Transform (DFT):

- Quadratic time complexity, $O(N^2)$
  computationally expensive for large signals.
- Need for a faster and more efficient algorithm. FFT resolves in $O(n \log n)$.

FFT has taken revolution in the field of Signal processing (noise removal, filtering,...), Image processing (compression, feature extraction,...), Speech and audio processing (compression, synthesis,...), Scientific computing (solving differential equations, analyzing time-series data) etc.

1.3 Fourier Transform

Fourier Transform decomposes a signal into its frequency components. It represents a signal in terms of sinusoidal basis functions.

- The continuous Fourier Transform is given by:

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

where $f(t)$ is the signal, and $F(\omega)$ is its frequency domain representation.

1.4 Discrete Fourier Transform

DFT is the discrete counterpart of the continuous Fourier Transform.

- It is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi kn/N}$$

where $x[n]$ is the discrete signal, and $X[k]$ is its frequency domain representation.

Direct computation of DFT is of $O(N^2)$ complexity.[1]

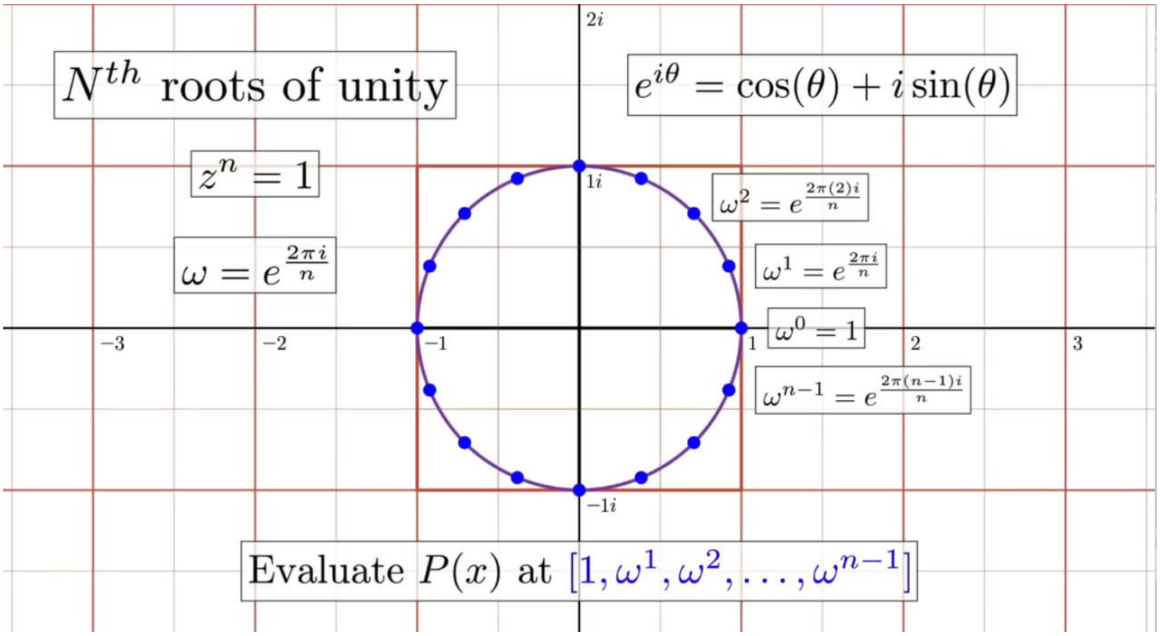
Polynomial Representation[8]

To understand the use of Discrete Fourier Transform let's take an example of a polynomial. The polynomial is of $(n-1)^{th}$ order where n can be either power of 2 or not. If it is not a power of 2 then take some extra terms of higher order putting the coefficients of those terms p_i 0.

$$P(x) = p_0 + p_1 x + p_2 x^2 + ... + p_{n-1} x^{n-1}$$

From complex number we know that the equation $z^n = 1$ has n complex solutions (called the n-th roots of unity), and the solutions are of the form $w_k = e^{\frac{2k\pi i}{n}}$ with $k = 0 \dots n-1$. Additionally these complex numbers have some very interesting properties: e.g. the principal n-th root $w_1 = e^{\frac{2\pi i}{n}}$ can be used to describe all other n-th roots: $w_k = w^k$.

The following figure best describes the senario:



The polynomial $P(x)$ can equivalently be represented as the vector of coefficients $(p_0, p_1, \ldots, p_{n-1})$. The Discrete Fourier Transform of the polynomial is defined as the values of the polynomial at the points $x = w_k$, i.e. it is the vector:

$$\begin{aligned}\text{DFT}(p_0, p_1, \ldots, p_{n-1}) &= (y_0, y_1, \ldots, y_{n-1}) \\ &= (P(w_0), P(w_1), \ldots, P(w_{n-1})) \\ &= (P(w^0), P(w^1), \ldots, P(w^{n-1}))\end{aligned}$$

On the other hand, the Inverse Discrete Fourier Transform is defined: The inverse DFT of values of the polynomial $(y_0, y_1, \ldots, y_{n-1})$ are the coefficients of the polynomial $(p_0, p_1, \ldots, p_{n-1})$.

$$\text{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (p_0, p_1, \ldots, p_{n-1})$$

Thus by using the n^{th} roots of unity we can find the DFT of a polynomial(DFT on coefficients) and by the Inverse DFT (Inverse DFT on values) we again can restore the original coefficients of the polynomial.

For fast multiplication of two polynomials DFT and Inverse DFT can be used. First find the DFT of the two polynomials let say ($M(x)$ and $N(x)$) and their DFTs are $DFT(M)$ and $DFT(N)$.

If we multiply the vectors DFT(M) and DFT(N) - by multiplying each element of one vector by the corresponding element of the other vector - then we get nothing other than the DFT of the polynomial DFT($M \cdot N$):

$$\text{DFT}(M \cdot N) = \text{DFT}(M) \cdot \text{DFT}(N)$$

Here, on the right side multiplication of DFTs we mean the pairwise product of the vector elements. This can be computed in $O(n)$ time. Next we do Inverse DFT on the result

$$M \cdot N = \text{InverseDFT}(\text{DFT}(M) \cdot \text{DFT}(N))$$

If this calculation needs $O(n \log n)$ computational steps, then the overall time complexity of the algorithm will be $O(n \log n)$.

1.5 Fast Fourier Transform

Fast Fourier Transform(FFT) is the solution of the previous computational complexity problem. Applying FFT the complexity can be reduced to $O(n \log n)$.Basically this algorithm uses divide and conquer technique to resolve the issue.

Let's take a polynomial $P(x)$ of order is $n - 1$, where n is a power of 2.

$$P(x) = p_0 + p_1 x + p_2 x^2 + ... + p_{n-1} x^{n-1}$$

We divide it into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:

$$\begin{aligned}P_o(x) &= p_0 x^0 + p_2 x^1 + \cdots + p_{n-2} x^{\frac{n}{2}-1} \\ P_e(x) &= p_1 x^0 + p_3 x^1 + \cdots + p_{n-1} x^{\frac{n}{2}-1}\end{aligned}$$

The polynomial can also be represented as:

$$P(x) = P_o(x^2) + x P_e(x^2).$$

The polynomials P_o and P_e are only half as much coefficients as the polynomial P. If we can compute the DFT(P) in linear time using DFT(P_o) and DFT(P_e), then we get the recurrence $T_{\mathrm{DFT}}(n) = 2T_{\mathrm{DFT}}\left(\frac{n}{2}\right) + O(n)$ for the time complexity, which results in $T_{\mathrm{DFT}}(n) = O(n \log n)$ by the master theorem.

Let's say we have already computed the $(y_k^o)_{k=0}^{n/2-1} = DFT(P_o)$ and $(y_k^e)_{k=0}^{n/2-1} = DFT(P_e)$

For the first $\frac{n}{2}$ values we can just use the previously noted equation $P(x) = P_o(x^2) + xP_e(x^2)$:

$$y_k = y_k^o + w^k y_k^e, \quad k = 0, 1, \ldots, \frac{n}{2} - 1.$$

The next $\frac{n}{2}$ values we need to find an expression which is pretty similar to our last expression of finding first half:

$$\begin{aligned} y_{k+n/2} &= P\left(w^{k+n/2}\right) \\ &= P_o\left(w^{2k+n}\right) + w^{k+n/2} P_e\left(w^{2k+n}\right) \\ &= P_o\left(w^{2k} w^n\right) + w^k w^{n/2} P_e\left(w^{2k} w^n\right) \\ &= P_o\left(w^{2k}\right) - w^k P_e\left(w^{2k}\right) \\ &= y_k^o - w^k y_k^e \end{aligned}$$

Here we used again $P(x) = P_o(x^2) + xP_e(x^2)$ and the two identities $w^n = 1$ and $w^{n/2} = -1$.
Therefore we get the desired formulas for computing the whole vector (y_k):

$$\begin{aligned} y_k &= y_k^o + w^k y_k^e, & k &= 0 \ldots \frac{n}{2} - 1, \\ y_{k+n/2} &= y_k^o - w^k y_k^e, & k &= 0 \ldots \frac{n}{2} - 1. \end{aligned}$$

Thus we learned how to compute the DFT in $O(n \log n)$ time.
To know more check this[7] book.

Chapter 2

Discovery

Let's consider product of two d degree polynomials $A(x)$ & $B(x)$ is $C(x)$. $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{d-1}x^{d-1} + a_dx^d$
$B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{d-1}x^{d-1} + b_dx^d$
$C(x) = c_0 + c_1x + c_2x^2 + \ldots + c_{2d-1}x^{2d-1} + c_{2d}x^{2d}$

The coefficients of $C(x)$ can be calculated as follows: $c_k = \sum_{i=0}^{k} a_i b_{k-i}$ for $k = 0, 1, 2, \ldots, 2d$
This calculation requires $O(d^2)$ operations. Can we do better?

2.1 Convolution

The product of two polynomials is called the convolution of the two coefficient sequences. The convolution of two sequences $a_0, a_1, \ldots, a_{d-1}$ and $b_0, b_1, \ldots, b_{d-1}$ is the sequence $c_0, c_1, \ldots, c_{2d-1}$ where $c_k = \sum_{i=0}^{k} a_i b_{k-i}$ for $k = 0, 1, 2, \ldots, 2d-1$

The convolution of two sequences can be calculated in $O(d^2)$ operations. Can we do better?

2.2 Value Representation

The polynomial $A(x)$ can be represented as a sequence of its values at $d+1$ distinct points. For example, the polynomial $A(x) = 3 + 4x + 5x^2$ can be represented as the sequence $(3, 7, 23)$ because $A(0) = 3$, $A(1) = 7$, and $A(2) = 23$. The polynomial $B(x)$ can also be represented as a sequence of its values at $d+1$ distinct points. The product of two polynomials can be represented as the convolution of the two sequences of values.

We can calculate the convolution of two sequences of values in $O(d)$ operations. So, we want to convert the coefficients of the polynomials to the values of the polynomials at $d+1$ distinct points. This can be done using the Fast Fourier Transform (FFT).

2.3 Bigger Picture

The goal is to convert the coefficient representation of $A(x)$ and $B(x)$ to the value representation. Then we can calculate the convolution of the two sequences of values. Finally, we can convert the value representation of the product of the two polynomials to the coefficient representation.
Conversion of the coefficient representation to the value representation can be done using the FFT. And the reverse process can be done using the inverse FFT.

2.4 Coefficient to Value Representation

To get $d+1$ distinct points, we can use the $d+1$ roots of unity. The $d+1$ roots of unity are the complex numbers $1, \omega, \omega^2, \ldots, \omega^d$ where $\omega = e^{2\pi i/(d+1)}$. The polynomial $A(x)$ can be represented as the sequence $A(1), A(\omega), A(\omega^2), \ldots, A(\omega^d)$. The polynomial $B(x)$ can also be represented as the sequence $B(1), B(\omega), B(\omega^2), \ldots, B(\omega^d)$. The product of two polynomials can be represented as the convolution of the two sequences of values.

2.5 What's the problem

Now, we have the domain of the polynomials as the $d+1$ roots of unity. But as we need to evaluate the value representation of each of the polynomial at $d+1$ distinct points, this is still $O(d^2)$. Can we do better?

2.6 Solution: even function

We can divide the polynomial into two parts: the even terms and the odd terms. The even terms are the terms with even powers of x and the odd terms are the terms with odd powers of x. For example,

$$\begin{aligned}
A(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 \\
&= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_1 + a_3x^2 + a_5x^4 + a_7x^6) \\
&= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_0 + a_2x^2 + a_4x^4 + a_6x^6)x \\
A(x) &= A_e(x^2) + xA_o(x^2) \\
A(x_i) &= A_e(x_i^2) + x_iA_o(x_i^2) \\
A(-x_i) &= A_e(x_i^2) - x_iA_o(x_i^2)
\end{aligned}$$

here, $A_e(x)$ is the polynomial with the even terms and $A_o(x)$ is the polynomial with the odd terms. Now, both $A_e(x)$ and $A_o(x)$ are of degree $d/2$. As both $A_e(x)$ and $A_o(x)$ are functions of x^2, we can evaluate $A(x)$ using $\pm$ pairs of points. Hence, we need to evaluate $A_e(x)$ and $A_o(x)$ at $d/2+1$ distinct points. Improvement!!

2.7 Generalization

We can generalize this to any degree polynomial. For a $n-1$ degree polynomial,

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

we need to evaluate $P(x)$ at n distinct points. $\pm x_1, \pm x_2, \ldots, \pm x_n$

$$\begin{aligned}
P(x) &= P_e(x^2) + xP_o(x^2) \\
P(x_i) &= P_e(x_i^2) + x_iP_o(x_i^2) \\
P(-x_i) &= P_e(x_i^2) - x_iP_o(x_i^2)
\end{aligned}$$

$P_e(x^2)$ and $P_o(x^2)$ are of degree $n/2-1$.
So, we need to evaluate both $P_e(x^2)$ and $P_o(x^2)$ at $x_1^2, x_2^2, \ldots, x_{n/2}^2 (n/2+1$ points.)
This is the start of a recursive algorithm to evaluate a polynomial at n distinct points.

2.8 Divide and Conquer

Now we have a recursive algorithm to evaluate a polynomial at n distinct points. We can use the divide and conquer paradigm to evaluate the polynomial at n distinct points.In each step, We can divide the polynomial into two parts: the even terms and the odd terms. We can evaluate the even and odd terms at $n/2 + 1$ distinct points. This is the divide step. We can then combine the results of the even and odd terms to get the result for the entire polynomial. This is the conquer step. As the polynomial is divided into two parts at each step, the depth of the recursion is $O(\log n)$. The time complexity of the algorithm is $O(n \log n)$.

2.9 Another Problem

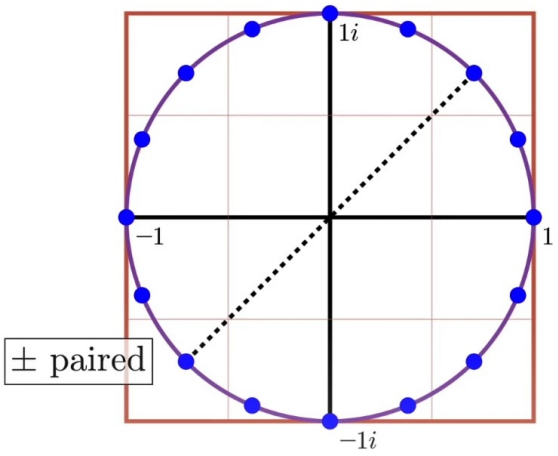
In the first step divide and conquer algorithm, we have squared the x in the polynomial. So, we don't have the $\pm$ pairs in the very second step of the algorithm. It means, the recursion will break! We need to find a way to get the$\pm$ pairs in the very second step of the algorithm. Or a way where we can get the $\pm$pairs after squaring the x.

2.10 Solution: Roots of Unity

We can solve the problem of getting negative number after squaring the x by using the n roots of unity. The n roots of unity are the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ where $\omega = e^{2\pi i/n}$. So, for a polynomial of degree d, we need $n \geq d + 1$ points to evaluate where $n = 2^k$, $k \in Z$.

2.11 Why does this work?

For any of jth root of unity $\omega^j = -\omega^{j+n/2} \implies (\omega^j and \omega^{j+n/2})$ are $\pm$ pairs.



2.12 Finalize

For a polynomial $P(x)$ we can now convert the coefficients of the polynomial to the values of the polynomial at n distinct points using the FFT. We can then calculate the convolution of two sequences of values.

Chapter 3

Inverse FFT

Let the value representation of $P(x)$ of degree $n-1$ is given. We want to convert the value representation to coefficient representation. We can use the inverse FFT to do this.

Before we start, let's see another perspective of forward FFT or Evaluation.

$$P(x) = p_0 + p_1x + p_2x^2 + ... + p_{n-1}x^{n-1}$$

$$P(x_0) = p_0 + p_1x_0 + p_2x_0^2 + ... + p_{n-1}x_0^{n-1}$$
$$P(x_1) = p_0 + p_1x_1 + p_2x_1^2 + ... + p_{n-1}x_1^{n-1}$$
$$P(x_2) = p_0 + p_1x_2 + p_2x_2^2 + .. + p_{n-1}x_2^{n-1}$$

$$P(x_{n-1}) = p_0 + p_1x_{n-1} + p_2x_{n-1}^2 + ... + p_{n-1}x_{n-1}^{n-1}$$

We can represent this as *Matrix Multiplication* as follows:

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ P(x_2) \\ \vdots \\ P(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

In forward FFT, the nth roots of unity was used to evaluate a polynomial of $n-1$ degree. So, we can replace x_k with ω^k, where $\omega_i = e^{\frac{2\pi i}{n}}$

$$\begin{bmatrix} P(w^0) \\ P(w^1) \\ P(w^2) \\ \vdots \\ P(w^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

This matrix is called the *Vandermonde Matrix*. Here we need to get the values of $[p_0, p_1, ..., p_{n-1}]$. We can rewrite the equation as follows:

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix}^{-1} \begin{bmatrix} P(w^0) \\ P(w^1) \\ P(w^2) \\ \vdots \\ P(w^{n-1}) \end{bmatrix}$$

$$\implies \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w^{-}1 & w^{-2} & \cdots & w^{-n-1} \\ 1 & w^{-2} & w^{-4} & \cdots & w^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{-n-1} & w^{-2(n-1)} & \cdots & w^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} P(w^0) \\ P(w^1) \\ P(w^2) \\ \vdots \\ P(w^{n-1}) \end{bmatrix}$$

The inverse matrix and original matrix look quiet similar! The ω in the original matrix is now $\frac{1}{n}\omega^{-1}$. So, we get the formula as:

$$p_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Chapter 4

Implementation

4.1 FFT

```
import cmath
import math

def FFT(P):
        # P - [p_0, p_1, p_2, ... , p_n-1] coeff representation
        n = len(P) # n = 2^k
        if n == 1:
                return P
        omega = cmath.exp(2 * cmath.pi * 1j / n)

        # divide the polynomial into even and odd terms
        P_even, P_odd = P[0::2], P[1::2]
        # get the value representation of the even and odd terms
        y_even, y_odd = FFT(P_even), FFT(P_odd)
        # initialize the value representation of the polynomial P
        y = [0] * n

        # combine the value representation of the even and odd terms
        for j in range(n/2):
                y[j] = P_even[j] + omega**j * P_odd[j]
                y[j + n/2] = P_even[j] - omega**j * P_odd[j]

        return y
```

4.2 Inverse FFT

```
import cmath
import math

def IFFT(P):
        # P - [p_0, p_1, p_2, ... , p_n-1] coeff representation
        n = len(P) # n = 2^k
        if n == 1:
                return P

        # Here's the difference from the forward FFT:
```

```
omega = cmath.exp(-2 * cmath.pi * 1j / n) / n

# divide the polynomial into even and odd terms
P_even, P_odd = P[0::2], P[1::2]
# get the value representation of the even and odd terms
y_even, y_odd = IFFT(P_even), IFFT(P_odd)
# initialize the value representation of the polynomial P
y = [0] * n

# combine the value representation of the even and odd terms
for j in range(n/2):
        y[j] = P_even[j] + omega**j * P_odd[j]
        y[j + n/2] = P_even[j] - omega**j * P_odd[j]

return y
```

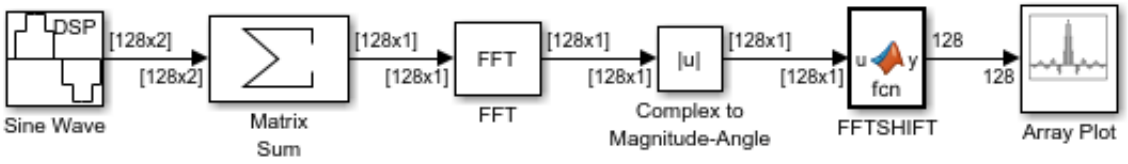
Chapter 5

Applications

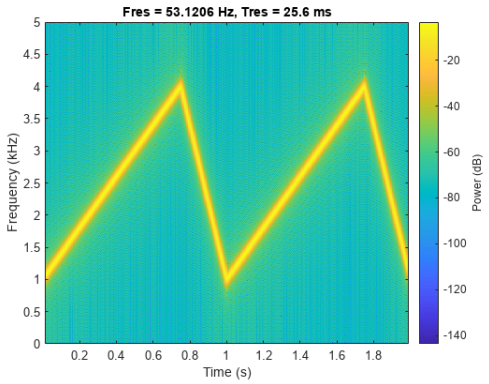
FFT is the core component of frequency domain analysis, also known as spectrum analysis, in signal processing. It is widely used in data compression, spectral estimation, signal filtering, and other applications. Analysis in both the frequency and time domains can be done simultaneously using FFT variations like the short-time Fourier transform. These methods can be applied to a wide range of signals, including communication, radar, audio and speech, and other sensor data signals. Additionally, FFT is occasionally employed as a transitional step before more sophisticated signal processing methods. FFT is used for filtering and picture compression in image processing. In mathematics and physics, partial differential equations (PDEs) are also solved using FFT.

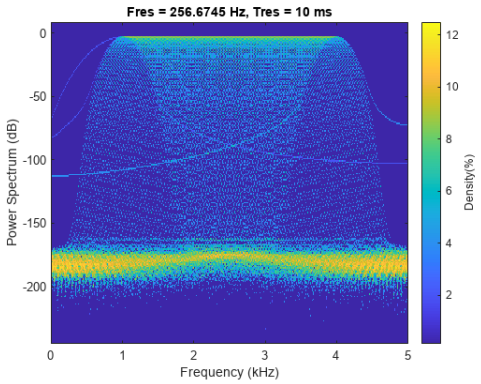
5.1 FFT in signal processing

FFT is most frequently used to convert signals from the time domain to the frequency domain.



The fundamental algorithm used for time-frequency based signal analysis in many simulation programs, such as MATLAB, is FFT. The FFT algorithm was utilized by MATLAB's[5] Signal Processing $Toolbox^{TM}$ to display spectograms and persistent spectrums, which are time-frequency views that display the proportion of a signal's duration that a certain frequency is present.





FFT

helps in radar signal analysis, target detection, and tracking. FFT aids in underwater acoustics, target detection, and classification.

5.2 FFT in Communications

- Digital communication systems: FFT is utilized in modulation and demodulation schemes such as OFDM (Orthogonal Frequency Division Multiplexing) used in Wi-Fi, LTE, and digital television.
- Channel estimation and equalization: FFT helps in estimating the channel characteristics and compensating for distortions in communication channels.

5.3 FFT in Instrumentation

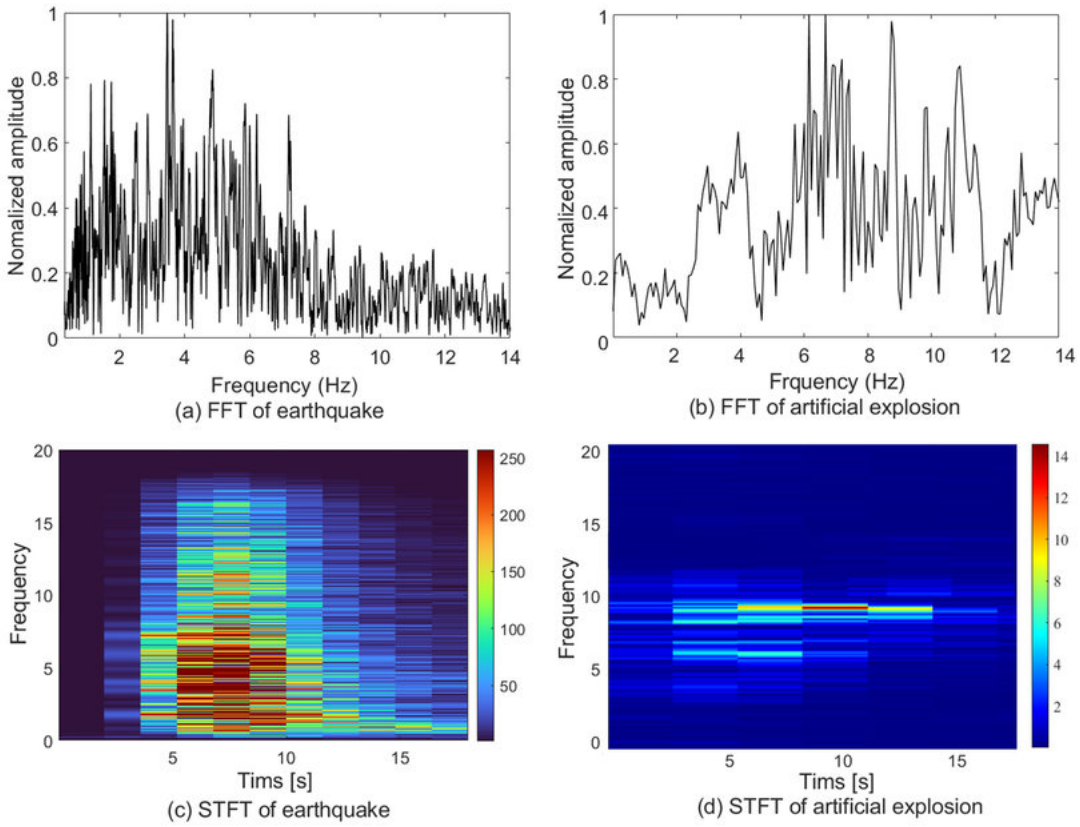
- Spectrum analysis: FFT is employed in spectrum analyzers for measuring frequency components of signals.
- Biomedical signal analysis: FFT is used in the analysis of EEG (Electroencephalography), ECG (Electrocardiography), and other biomedical signals for diagnosis and monitoring.

5.4 FFT in Control Systems

- Control system analysis: FFT is used in identifying system dynamics, frequency response analysis, and stability analysis.
- Control system design: FFT helps in designing controllers and compensators for systems with frequency-dependent characteristics.

5.5 FFT in Geophysics and Seismology

- Earthquake analysis: FFT is used in seismic data processing for earthquake detection, location, and magnitude estimation.



- Exploration geophysics: FFT assists in the interpretation of seismic data for oil and gas exploration.

5.6 FFT in Medical Imaging

- MRI (Magnetic Resonance Imaging): FFT is utilized in MRI for image reconstruction from raw data acquired during scans.
- CT (Computed Tomography): FFT is used in image reconstruction algorithms to convert raw projection data into cross-sectional images.

5.7 FFT in Astrophysics and Cosmology

- Radio astronomy: FFT is employed in processing radio signals received from celestial objects for analysis and imaging.
- Cosmology: FFT assists in analyzing cosmic microwave background radiation data for studying the early universe.

5.8 FFT in Computer Science

DFT, a special variation of FFT can be used in a huge variety of other problems, which at the first glance have nothing to do with multiplying polynomials.

5.8.1 All possible sums

We are given two arrays $a[]$ and $b[]$. We have to find all possible sums $a[i] + b[j]$, and for each sum count how often it appears.

For example for $a = [1,\ 2,\ 3]$ and $b = [2,\ 4]$ we get: then sum 3 can be obtained in 1 way, the sum 4 also in 1 way, 5 in 2, 6 in 1, 7 in 1.

We construct for the arrays a and b two polynomials A and B. The numbers of the array will act as the exponents in the polynomial ($a[i] \Rightarrow x^{a[i]}$); and the coefficients of this term will be how often the number appears in the array.

Then, by multiplying these two polynomials in $O(n \log n)$ time, we get a polynomial C, where the exponents will tell us which sums can be obtained, and the coefficients tell us how often. To demonstrate this on the example:

$$(1x^1 + 1x^2 + 1x^3)(1x^2 + 1x^4) = 1x^3 + 1x^4 + 2x^5 + 1x^6 + 1x^7$$

5.8.2 All possible scalar products

We are given two arrays $a[]$ and $b[]$ of length n. We have to compute the products of a with every cyclic shift of b.

We generate two new arrays of size $2n$: We reverse a and append n zeros to it. And we just append b to itself. When we multiply these two arrays as polynomials, and look at the coefficients $c[n-1],\ c[n],\ \ldots,\ c[2n-2]$ of the product c, we get:

$$c[k] = \sum_{i+j=k} a[i]b[j]$$

And since all the elements $a[i] = 0$ for $i \geq n$:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i]$$

It is easy to see that this sum is just the scalar product of the vector a with the $(k-(n-1))$-th cyclic left shift of b. Thus these coefficients are the answer to the problem, and we were still able to obtain it in $O(n \log n)$ time. Note here that $c[2n-1]$ also gives us the n-th cyclic shift but that is the same as the 0-th cyclic shift so we don't need to consider that separately into our answer.

5.8.3 Two stripes

We are given two Boolean stripes (cyclic arrays of values 0 and 1) a and b. We want to find all ways to attach the first stripe to the second one, such that at no position we have a 1 of the first stripe next to a 1 of the second stripe.

The problem doesn't actually differ much from the previous problem. Attaching two stripes just means that we perform a cyclic shift on the second array, and we can attach the two stripes, if scalar product of the two arrays is 0.

5.8.4 String matching

We are given two strings, a text T and a pattern P, consisting of lowercase letters. We have to compute all the occurrences of the pattern in the text.

We create a polynomial for each string ($T[i]$ and $P[I]$ are numbers between 0 and 25 corresponding to the 26 letters of the alphabet):

$$A(x) = a_0 x^0 + a_1 x^1 + \cdots + a_{n-1} x^{n-1}, \quad n = |T|$$

with

$$a_i = \cos(\alpha_i) + i \sin(\alpha_i), \quad \alpha_i = \frac{2\pi T[i]}{26}.$$

And

$$B(x) = b_0x^0 + b_1x^1 + \cdots + b_{m-1}x^{m-1}, \quad m = |P|$$

with

$$b_i = \cos(\beta_i) - i\sin(\beta_i), \quad \beta_i = \frac{2\pi P[m-i-1]}{26}.$$

Notice that with the expression $P[m-i-1]$ explicitly reverses the pattern.

The $(m-1+i)$th coefficients of the product of the two polynomials $C(x) = A(x) \cdot B(x)$ will tell us, if the pattern appears in the text at position i.

$$c_{m-1+i} = \sum_{j=0}^{m-1} a_{i+j} \cdot b_{m-1-j} = \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i\sin(\alpha_{i+j})) \cdot (\cos(\beta_j) - i\sin(\beta_j))$$

with $\alpha_{i+j} = \frac{2\pi T[i+j]}{26}$ and $\beta_j = \frac{2\pi P[j]}{26}$

If there is a match, than $T[i+j] = P[j]$, and therefore $\alpha_{i+j} = \beta_j$. This gives (using the Pythagorean trigonometric identity):

$$\begin{aligned} c_{m-1+i} &= \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i\sin(\alpha_{i+j})) \cdot (\cos(\alpha_{i+j}) - i\sin(\alpha_{i+j})) \\ &= \sum_{j=0}^{m-1} \cos(\alpha_{i+j})^2 + \sin(\alpha_{i+j})^2 = \sum_{j=0}^{m-1} 1 = m \end{aligned}$$

If there isn't a match, then at least a character is different, which leads that one of the products $a_{i+1} \cdot b_{m-1-j}$ is not equal to 1, which leads to the coefficient $c_{m-1+i} \neq m$.

5.8.5 String matching with wildcards

This is an extension of the previous problem. This time we allow that the pattern contains the wildcard character , which can match every possible letter. E.g. the pattern $a * c$ appears in the text *abccaacc* at exactly three positions, at index 0, index 4 and index 5.

We create the exact same polynomials, except that we set $b_i = 0$ if $P[m-i-1] = *$. If x is the number of wildcards in P, then we will have a match of P in T at index i if $c_{m-1+i} = m - x$.

More problems to explore[10] [3]

- POLYMUL - Polynomial Multiplication
- MAXMATCH - Maximum Self-Matching
- ADAMATCH - Ada and Nucleobase
- Yet Another String Matching Problem[6]
- Lightsabers (hard)[9]
- Running Competition[2]
- Dasha and cyclic table[11]
- Centroid Probabilities[4]
- Expected Number of Customer
- Power Sum
- A+B Problem
- K-Inversions

Chapter 6

Conclusion

In conclusion, the Fast Fourier Transform (FFT) technique is a cornerstone in signal processing and beyond. Its extraordinary efficiency in computing the Discrete Fourier Transform (DFT) has enabled advances in a wide range of applications.

Throughout this report, we have explored the diverse applications of the FFT algorithm, spanning signal processing, communications, instrumentation, control systems, geophysics, medical imaging, finance, astrophysics, and cosmology. From audio and image processing to radar and sonar signal analysis, from medical diagnostics to seismic data interpretation, the FFT algorithm finds itself at the heart of countless innovative solutions.

The versatility of the FFT algorithm supports vital technologies like digital communication networks, medical imaging equipment, and astronomical observatories in addition to making signal analysis and modification easier. Its widespread use highlights its significance as a vital instrument in contemporary research and engineering.

Looking ahead, as technology continues to evolve, the FFT algorithm is poised to remain an indispensable asset, enabling further breakthroughs in fields ranging from data science to space exploration. Its impact on our understanding of the world and our ability to manipulate and interpret signals is undeniable, making it a subject of continued research and application for years to come.

In essence, the FFT algorithm exemplifies the power of computational methods to unlock new insights and capabilities across a broad spectrum of disciplines, shaping the landscape of scientific inquiry and technological innovation.

References

[1] CP Algorithms. Fast fourier transform, 2024. Accessed: March 7, 2024.

[2] Ivan Androsov. 1398g - running competition, 2020. Accessed: March 8, 2024.

[3] Codechef. Codechef problemset, 2024. Accessed: March 8, 2024.

[4] Péter Gyimesi. 1667e - centroid probabilities, 2022. Accessed: March 8, 2024.

[5] MATLAB. Fast fourier transform, 2024. Accessed: March 9, 2024.

[6] Mikhail Piklyaev. Yet another string matching problem, 2018. Accessed: March 8, 2024.

[7] Robert W. Ramirez. *The Fft, Fundamentals and Concepts*. Prentice Hall, 1st edition, 1984.

[8] Reducible. The fast fourier transform (fft): Most ingenious algorithm ever?, 2020. Accessed: February 12, 2024.

[9] Damian Straszak. Helvetic coding contest 2018 solution sketches, 2018. Accessed: March 8, 2024.

[10] Tähvend Uustalu. Fft [tutorial], 2023. Accessed: March 8, 2024.

[11] Aliaksei Vistiazh. 754e - dasha and cyclic table, 2017. Accessed: March 8, 2024.

[12] Wikipedia. Fast fourier transform, 2024. Accessed: March 9, 2024.