BINUS UNIVERSITY INTERNATIONAL Even Semester (2022)

# BINUS UNIVERSITY
# BINUS INTERNATIONAL

**Assignment Cover Letter**

**(Individual Work )**

| Student Information | Surname | Given Names | Student ID Number |
|---|---|---|---|
| 1. | Choa | Bernard | 2502022414 |

| | | | |
|---|---|---|---|
| **Course Code** | : COMP6699001 | **Course Name** | : Object Oriented Programming |
| **Class** | : L2AC | **Name of Lecturer(s)** | : Jude Joseph Lamug Martinez |
| **Major** | : Computer Science | | |
| **Title of Assignment** | : Fast Fourier Transform for the Fast Multiplication of Polynomials | | |
| **Type of Assignment** | : Final Project | | |
| **Submission Pattern** | | | |
| **Due Date** | : 10 June 2022 | **Submission Date** | : 8 June 2022 |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Bernard Choa

Bernard Choa

# Table of Contents

**“Fast Fourier Transform for the Fast Multiplication of Polynomials”**

**Name : Bernard Choa**

**ID : 2502022414**

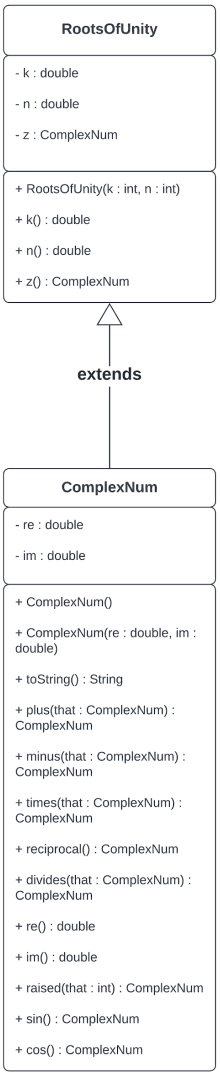
## I. Program Description

The Fast Fourier Transform is one of the most seminal pieces of ingenuity in computer science that helped shape the modern world we live in today, responsible for technologies such as GPS (Ahamed et al., 2016), MRI scans (Elster, 2001), medical image reconstruction (MIR) (NVIDIA, 2005), Fourier transform infrared (FTIR) spectroscopy (ThermoFisher, n.d.), convolutional neural networks (CNNs) for deep learning (Sun, 2021), and even for studying Schrodinger’s equation in quantum mechanics (Serov & Sergeeva, 2016). It is no coincidence that the Fast Fourier Transform is also nominated as one of the world’s most important emerging technologies by MIT News in 2012 ("Faster fourier transform named one of world’s most important emerging technologies," 2012).

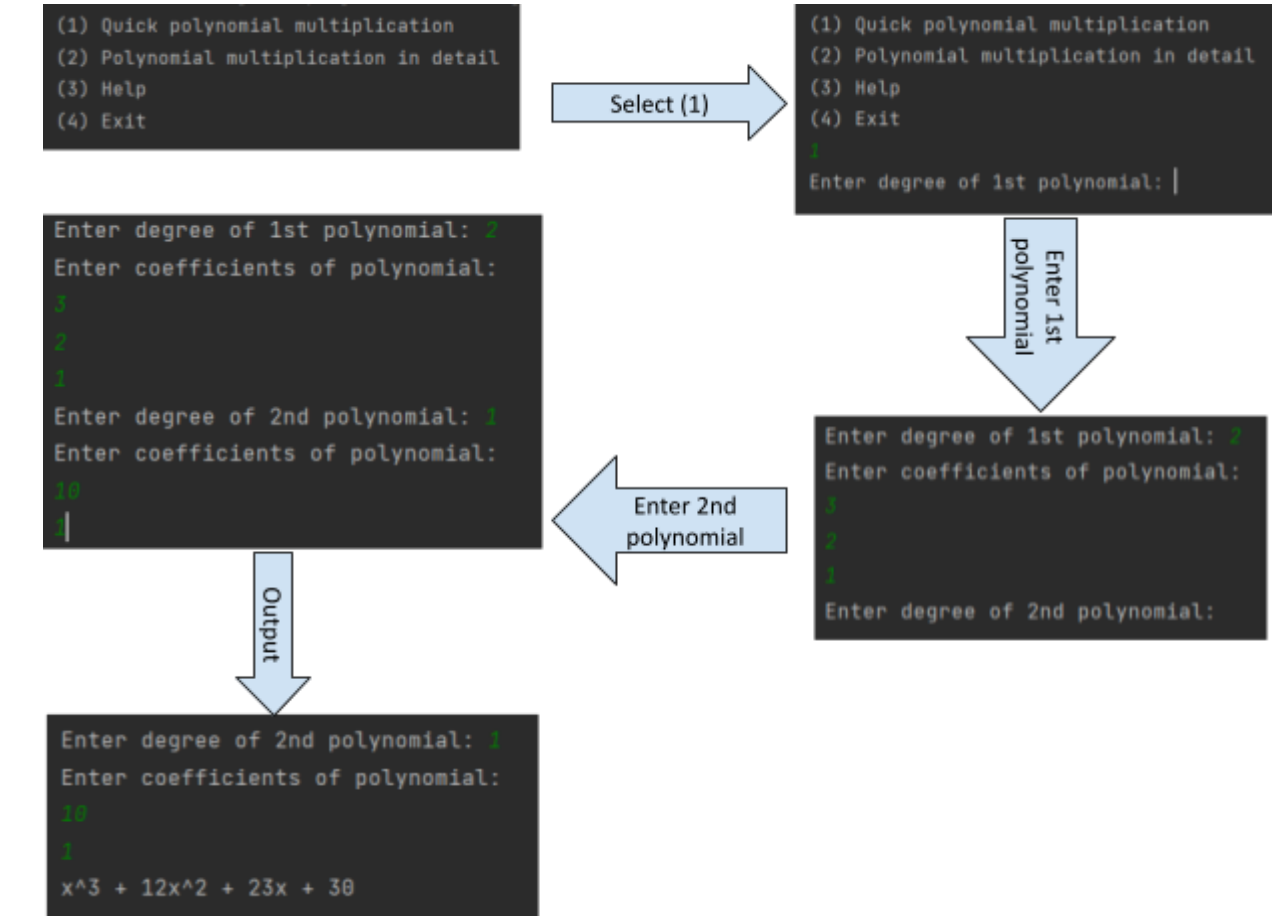
This program is intended to serve both as a polynomial multiplication calculator, and as an introductory learning tool for the playful exploration of the Fast Fourier Transform, not meant to replace university courses on the subject, but to simply supplement and facilitate the learning process.

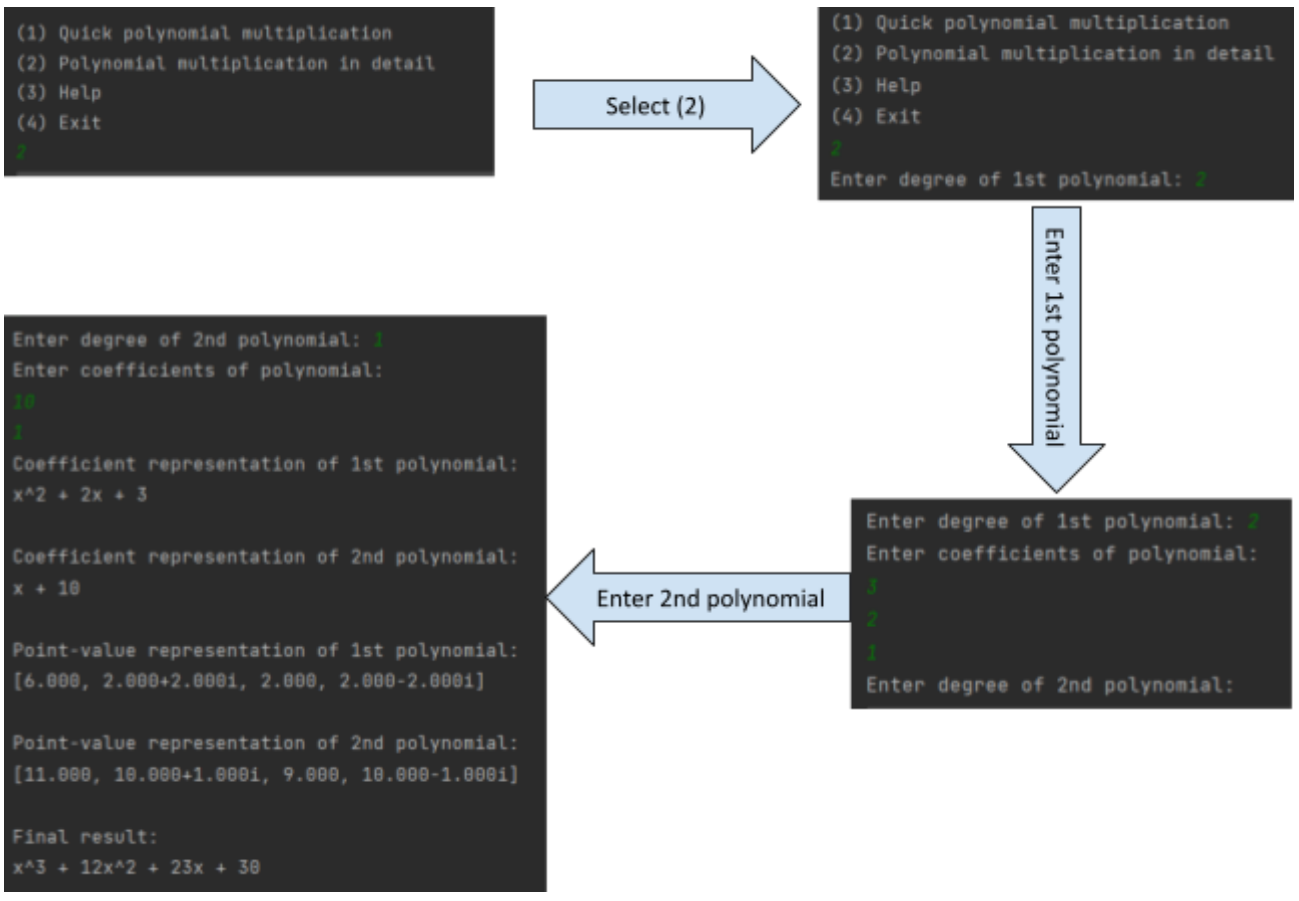
## II. Class Diagram

- ***public class*** “ComplexNum” as a public parent class for “RootsOfUnity” and responsible for the instantiation of ComplexNum objects which imitates complex numbers.
- ***public class*** “RootsOfUnity” as a public child class of the “ComplexNum” parent class and instantiates the *k*th point of the *n*th root of unity.


RootsOfUnity
- k : double
- n : double
- z : ComplexNum
+ RootsOfUnity(k : int, n : int)
+ k() : double
+ n() : double
+ z() : ComplexNum
extends
ComplexNum
- re : double
- im : double
+ ComplexNum()
+ ComplexNum(re : double, im : double)
+ toString() : String
+ plus(that : ComplexNum) : ComplexNum
+ minus(that : ComplexNum) : ComplexNum
+ times(that : ComplexNum) : ComplexNum
+ reciprocal() : ComplexNum
+ divides(that : ComplexNum) : ComplexNum
+ re() : double
+ im() : double
+ raised(that : int) : ComplexNum
+ sin() : ComplexNum
+ cos() : ComplexNum

## III. Application Flow




To quit, simply type 4 in the selection menu.

## IV. Lessons that Have Been Learned

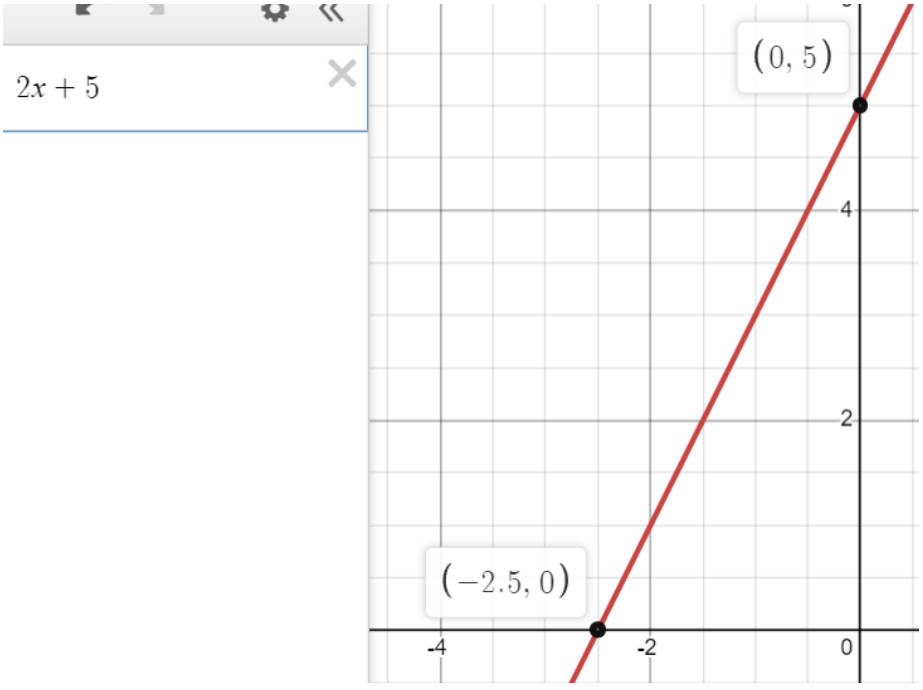
The lessons learned in the process of creating this project are heavily involved in mathematics. I used the ArrayList package extensively, I summoned the Math class and its methods multiple times, but ultimately it is an exercise in mathematical rigor and translating that rigor into the domain of computers. I learned the concepts behind the Fast Fourier Transform, how those concepts are assembled into the Fast Fourier Transform, and applied the algorithm as a polynomial multiplication calculator. The concepts include point-value representation of polynomials, polynomial interpolation, symmetric monomials, and the roots of unity.

When the theoretical elegance of mathematics collides with the practical efficiency of computers, the result can be magical. The Fast Fourier Transform certainly falls into that category; mathematically simple and beautiful, while at the same time efficient, versatile, and is at the forefront of many integral technologies which drive the modern world.

## V. Project Technical Description

**Point-value representation of polynomial**

The standard notation of representing polynomials is to write all the terms down and their coefficients. For the purposes of this report, the notation will be referred to as the coefficient representation of polynomials. There is also the point-value representation of polynomials, where points that are passed through by the polynomial are written down instead of the function of the polynomial.




*Courtesy of: Desmos*

Any line can be represented by two dots. In the graph above, the linear equation *2x+5* intersects the points (-2.5, 0) and (0,5). Turns out, there is a generalization of this mathematical observation for any *n*th degree polynomial, as will be explained in the following paragraph (Desmos, n.d.).

**Polynomial interpolation**

It is possible to convert a polynomial to its point-value representation. Likewise, it is also possible to convert point-values to a unique polynomial. This mathematical procedure is known as interpolation. To prove that it is possible to obtain a polynomial from its point-value representation, the polynomial must always be unique.

Given $n + 1$ distinct points on a Cartesian plane, there exists a unique polynomial of $n$ degree which interpolates the distinct points (Wood, 2021).

After proving the above theorem, the next hint reveals itself when we attempt to represent the $y$ values as a general polynomial.

$$p(x_0) = y_0$$

The polynomial in question can be of any degree n.

$$p(x_0) = 1 + x_0 + x_0^2 + x_0^3 + x_0^4 + \ \dots \ + x_0^n = y_0$$

Each term also has a coefficient.

$$p(x_0) = 1 + x_0 + x_0^2 + x_0^3 + x_0^4 + \ \dots \ + x_0^n = y_0$$

$$\Downarrow$$

$$p(x_0) = a_0 + a_1x_0 + a_2x_0^2 + a_3x_0^3 + a_4x_0^4 + \ \dots \ + a_nx_0^n = y_0$$

This general form can be extended for all points of an $n + 1$ degree polynomial.

$$p(x_0) = a_0 + a_1x_0 + a_2x_0^2 + a_3x_0^3 + a_4x_0^4 + \ \dots \ + a_nx_0^n = y_0$$

$$p(x_1) = a_1 + a_1x_1 + a_2x_1^2 + a_3x_1^3 + a_4x_1^4 + \ \dots \ + a_nx_1^n = y_1$$

$$p(x_2) = a_2 + a_1x_2 + a_2x_2^2 + a_3x_2^3 + a_4x_2^4 + \ \dots \ + a_nx_2^n = y_2$$

$$p(x_3) = a_3 + a_1x_3 + a_2x_3^2 + a_3x_3^3 + a_4x_3^4 + \ \dots \ + a_nx_3^n = y_3$$

$$p(x_4) = a_4 + a_1x_4 + a_2x_4^2 + a_3x_4^3 + a_4x_4^4 + \ \dots \ + a_nx_4^n = y_4$$

$$\dots$$

$$p(x_n) = a_n + a_1x_n + a_2x_n^2 + a_3x_n^3 + a_4x_n^4 + \ \dots \ + a_nx_n^n = y_n$$

Arranging these equations side-by-side allows us to write the equations in matrix form.

$$\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & x_0^4 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 & x_1^4 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 & \dots & x_2^n \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 & \dots & x_3^n \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 & \dots & x_4^n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & x_n^3 & x_n^4 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ \dots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ \dots \\ y_n \end{bmatrix}$$

The *n* * *n* matrix containing the *x* variables is known as the *Vandermonde matrix*. By representing this system of equations in its matrix form, it becomes apparent that the coefficients can be solved for by inverting the *Vandermonde matrix* and multiplying it with the *y* vector.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ \dots \\ a_n \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & x_0^4 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 & x_1^4 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 & \dots & x_2^n \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 & \dots & x_3^n \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 & \dots & x_4^n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & x_n^3 & x_n^4 & \dots & x_n^n \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ \dots \\ y_n \end{bmatrix}$$

**Point-value multiplication**

Performing a point-value multiplication is rather easy. Simply multiply the y value of each function at the same x value (Reducible, 2020).

$$f(x) = x^2 + 2x + 1$$
$$(-2, 1),(-1, 0),(0, 1), (1, 4), (2, 9)$$

$$g(x) = x^2 - 2x + 1$$
$$(-2, 9), (-1, 4), (0, 1), (1, 0), (2, 1)$$

$$f(x) \cdot g(x) = x^4 - 2x^2 + 1$$
$$(-2, 1),(-1, 0),(0, 1), (1, 4), (2, 9)$$
$$(-2, 9), (-1, 4), (0, 1), (1, 0), (2, 1)$$
$$\Downarrow$$
$$(-2, 9), (-1, 0), (0, 1), (1, 0), (2, 9)$$

**The symmetry underlying polynomials**

Graphing polynomials is a challenge, however a nice pattern can be observed when monomials are graphed.

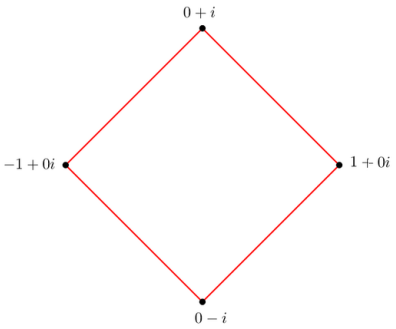
$$x^2 = (-1)^2 = 1$$
$$x^4 = (-1)^4 = 1$$
$$x^6 = (-1)^6 = 1$$
$$\vdots$$
$$P_e(-x) = P_e(x)$$

$$x = -1$$
$$x^3 = (-1)^3 = -1$$
$$x^5 = (-1)^3 = -1$$
$$\vdots$$
$$P_o(-x) = -P_o(x)$$

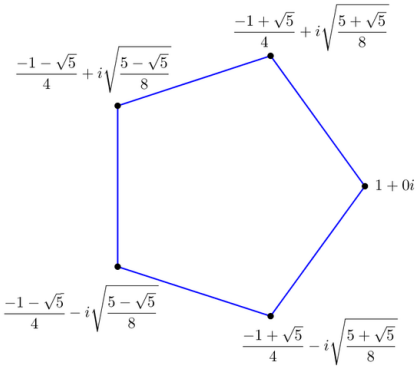
Odd degree monomials $P_o$ will return a positive value if the input is positive, and vice versa for negative inputs. Even degree monomials $P_e$ will always return a positive value, regardless of whether the input is positive or not. By knowing the *y* value of a monomial at *x* and taking advantage of the symmetry, we can also know the *y* value at *-x* without additional calculation, reducing computation time by half (Reducible, 2020).

**Roots of unity**

As stated before, given *n + 1* distinct points on a Cartesian plane, there exists a unique polynomial of *n* degree which interpolates the distinct points. For example, to obtain a 10th degree polynomial, 11 distinct points must be evaluated.




The 4th roots of unity



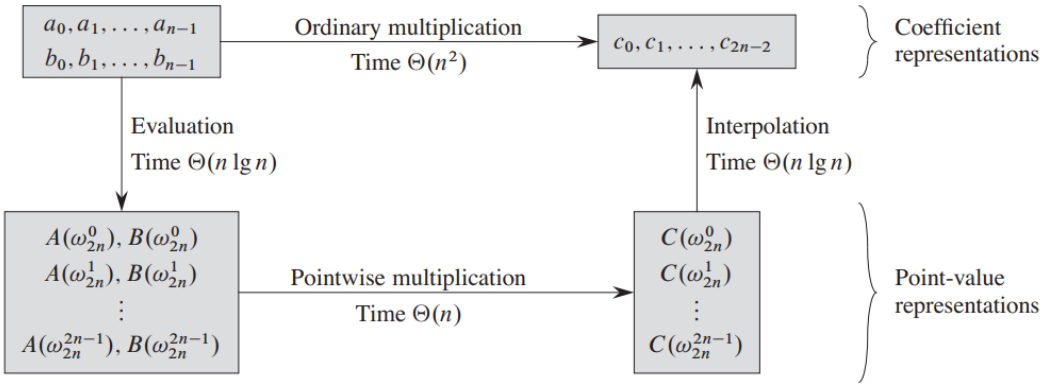

The 5th roots of unity

Rather than being limited to real numbers, the polynomial interpolation process can be improved upon by extending the plane to the complex numbers, especially the roots of unity. To put it simply, the roots of unity are complex numbers which satisfy the following equation.

$$z^n = 1$$
$$\forall\, n \,\epsilon\, N$$

**Putting it all together**

The program works by evaluating two polynomials into its point-value representation, multiplying the point-values, and interpolating the resultant point-value into the new polynomial (Cormen et al., 2009, 898-925).

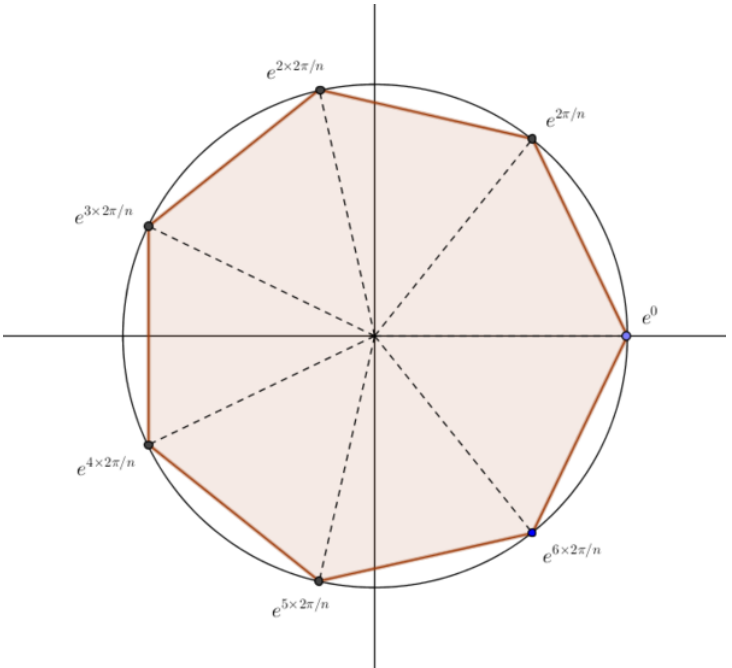



Rather than being limited to real numbers, the polynomial interpolation process can be improved upon by extending the plane to the complex numbers, especially the roots of unity. To put it simply, the roots of unity are complex numbers which satisfy the following equation.

$$P(x_j) = P_e(x_j^2) + x_j P_o(x_j^2)$$
$$P(-x_j) = P_e(x_j^2) - x_j P_o(x_j^2)$$
$$j \epsilon \{0, 1, 2, \ldots, ((n/2) - 1)\}$$

Any polynomial consists of even degree terms and odd degree terms, which can be considered as their own polynomials with their own even degree terms and odd degree terms, and so on. This indicates that the polynomial evaluation will be performed by a recursive algorithm. There is still a missing piece left; the odd and even degree terms will be evaluated in $x^2$, rendering the recursion impossible due to the strictly positive $x^2$ points, and this is where complex numbers can solve the problem.

$$\omega^j = e^{\frac{j2\pi i}{n}}$$
$$x_j = \omega^j$$



Passing in complex numbers which satisfy some *n*th root of unity, as opposed to passing in real numbers, circumvents the problem of strictly positive $x^2$ points. Moreover, it must be mentioned that the roots of unity should be a power of 2, i.e. 2, 4, 8, 16, etc., because it ensures that there will always be positive-negative pairs of $x$, or in other words, so that there will always be a $P(x_j)$ and $P(-x_j)$.

For a more complete explanation on the complex numbers, see (Welch Labs, 2015).

This is how the pseudocode should look like (based loosely on Python code):

```
def FFT(P):
n = len(P)
if n == 1:
      return P
ω = e^(2πi)/n
P_e, P_o = P[::2],P[1::2]
y_e, y_o = FFT(P_e),FFT(P_o)
y = [0] * n
for j in range(n/2):
      y[j] = y_e[j] + ω^j y_o[j]
      y[j+n/2] = y_e[j] - ω^j y_o[j]
return y
```

The next step of the procedure; point-value multiplication, is fairly simple, so it will not be discussed in depth. On the other hand, the Inverse FFT deserves an explanation. As mentioned before, to reverse the Fast Fourier Transform, the inverse of the *Vandermonde matrix* must be calculated.

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}$$

The *Vandermonde matrix* has been modified into the *DFT matrix* (Discrete Fourier Transform), and inverting this matrix yields the following result:

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}$$

Therefore, the pseudocode of the IFFT should be as follows;

```
def IFFT(P):
n = len(P)
if n == 1:
     return P
ω = e^(-(2πi)/n)
P_e, P_o = P[::2],P[1::2]
y_e, y_o = IFFT(P_e),IFFT(P_o)
y = [0] * n
for j in range(n/2):
     y[j] = y_e[j] + ω^j y_o[j]
     y[j+n/2] = y_e[j] - ω^j y_o[j]
return y
// somewhere outside the scope of the function
ifft_result = 1/n * IFFT(values);
```

This concludes the technical description of the project and its fundamental concepts.

## VI. Code Explanation

Project Structure

The Java class “PolynomFFTCalculator” serves as the driver code of the project, housed in the “driver” package, and containing the FFT algorithm itself as well as the features of the menu-driven program.

“ComplexNum” enables the creation and manipulation of complex numbers in the program.

“RootsOfUnity” enables the creation of complex numbers that are defined by the roots of unity, allowing for the simple implementation of the roots of unity.

Both the “ComplexNum” and “RootsOfUnity” custom classes are housed in the “externals” package.

```
package driver;
import externals.ComplexNum;
import externals.RootsOfUnity;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
```

*PolynomFFTCalculator*

**displayPoly()**

```
public static void displayPoly(ArrayList<ComplexNum> Poly) { // Handles the display of polynomials in coefficient representation
    boolean firstCoeff = true;
    for (int i = Poly.size()-1; i >= 0; i--) {
        // Numbers and its signs
        if (Math.abs(Math.round(Poly.get(i).re())) < 1) { // Will not print number if smaller than 1
            System.out.print("");
        } else if (firstCoeff) { // Only applies for number which appears first in the polynomial
            if (Math.round(Poly.get(i).re()) > 1) { // Will print number without the plus sign if number greater than 1
                System.out.print(String.format("%d", Math.round(Poly.get(i).re())));
                firstCoeff = false;
            } else if (Math.round(Poly.get(i).re()) == 1) { // Will not print anything if number equal to 1
                System.out.print("");
                firstCoeff = false;
            } else if (Math.round(Poly.get(i).re()) == -1) { // Will only print the minus sign if number equal to -1
                System.out.print("-");
                firstCoeff = false;
            } else { // Will print negative number if number smaller than -1
                System.out.print(String.format("%d", Math.round(Poly.get(i).re())));
                firstCoeff = false;
            }
        } else if (Math.round(Poly.get(i).re()) > 1 || i == 0 && Math.round(Poly.get(i).re()) == 1) { // Will print positive integer and its sign if greater than 1 OR for constants (x^0)
            System.out.print(" + ");
            System.out.print(String.format("%d", Math.round(Poly.get(i).re())));
        } else if (Math.round(Poly.get(i).re()) == 1) { // Will only print the positive sign if number equal to 1
            System.out.print(" + ");
        } else if (Math.round(Poly.get(i).re()) < -1 || i == 0 && Math.round(Poly.get(i).re()) == -1) { // Will print negative integer and its sign if smaller than -1 OR for constants (x^0)
            System.out.print(" - ");
            System.out.print(String.format("%d", -Math.round(Poly.get(i).re())));
        } else if (Math.round(Poly.get(i).re()) == -1) { // Will only print the negative sign if number equal to -1
            System.out.print(" - ");
        }

        // X-variables and its exponents
        if (Math.round(Poly.get(i).re()) == 0) { // Will not print variable if number equal to 0
            System.out.print("");
        } else if (i == 1) { // Will only print variable without the exponent if x^1
            System.out.print("x");
        } else if (i == 0) { // Will not print variable if x^0
            System.out.print("");
        } else { // Otherwise, the variable and its exponent will be printed
            System.out.print(String.format("x^%d", i));
        }
    }
    System.out.print("\n");
```

displayPoly is a void function that takes in an ArrayList<ComplexNum> (for more on ComplexNum, see pg. ) and prints the array list as a polynomial function in the terminal.

**displayVal()**

```
public static void displayVal(ArrayList<ComplexNum> Val) { // Handles the display of polynomials in point-value representation
    System.out.print("[");
    // All values printed will be accurate up to 3 digits
    for (int j = 0; j<Val.size();j++) {
        if (Math.round(Val.get(j).re()) > 0) { // Only applies when real component greater than 0
            if (Val.get(j).im() > 0) { // Will print real and imaginary component with plus sign if imaginary component greater than 0
                System.out.print(String.format("%.3f", (float) Val.get(j).re()));
                System.out.print("+");
                System.out.print(String.format("%.3f", (float) Val.get(j).im()));
                System.out.print("i");
            } else if (Val.get(j).im() < 0) { // Will only print real and imaginary component if imaginary component smaller than 0
                System.out.print(String.format("%.3f", (float) Val.get(j).re()));
                System.out.print(String.format("%.3f", (float) Val.get(j).im()));
                System.out.print("i");
            } else if (Val.get(j).im() == 0) { // Will only print real component if imaginary component equal to 0
                System.out.print(String.format("%.3f", (float) Val.get(j).re()));
            }
        } else { // Will only print imaginary component
            System.out.print(String.format("%.3f", (float) Val.get(j).im()));
            System.out.print("i");
        }

        if (j == Val.size()-1) { // Will print "]" after all values are printed
            System.out.print("]");
        } else { // Will print ", " after every value that is not the last value
            System.out.print(", ");
        }
    }
    System.out.print("\n");
}
```

displayVal is a void function that takes in an ArrayList<ComplexNum> and prints the array list as the point-value representation of the polynomial.

**calcMaxROU()**

```
public static double calcMaxROU(int poly1Degree, int poly2Degree) {
    // Calculates the largest roots of unity required to perform the polynomial multiplication
    double maxROU;
    if (poly1Degree+poly2Degree == Math.pow(2,Math.log(poly1Degree+poly2Degree)/Math.log(2))) {
        maxROU = Math.pow(2, Math.ceil(Math.log(poly1Degree + poly2Degree) / Math.log(2))+1);
    } else {
        maxROU = Math.pow(2, Math.ceil(Math.log(poly1Degree + poly2Degree) / Math.log(2)));
    }
    return maxROU;
}
```

This function calculates the largest root of unity required to interpolate the resulting polynomial obtained from multiplying two other polynomials.

**FFT()**

```java
public static ArrayList<ComplexNum> FFT(ArrayList<ComplexNum> polyCoeff) { // Fast Fourier Transform
    // n = len(P)
      double temp = Math.ceil(Math.log(polyCoeff.size())/Math.log(2));
    int n = (int) Math.pow(2,Math.ceil(Math.log(polyCoeff.size())/Math.log(2)));
    // Take in an array of integers representing polynomial coefficients (n must be a power of 2)

    // if n == 1
    if (n == 1) {
        return polyCoeff;
    }
    // Base case: If array contains only one integer, return the integer without modification
    // ω = e^((2pi*i)/n) has been replaced with the RootsOfUnity class instance
    // Pe,Po = P[::2],P[1::2]
    ArrayList<ComplexNum> polyEven = new ArrayList<>();
    ArrayList<ComplexNum> polyOdd = new ArrayList<>();
        // Creates empty ArrayList objects to store even degree and odd degree coefficients
    for (int j = 0; j < polyCoeff.size(); j += 2) {
        polyEven.add(polyCoeff.get(j));
    }
        // Stores even degree coefficients to Pe
    for (int j = 1; j < polyCoeff.size(); j += 2) {
        polyOdd.add(polyCoeff.get(j));
    }
        // Stores odd degree coefficients to Po
    // Separate the polynomial to even degree terms and odd degree terms

    // ye,yo = FFT(Pe),FFT(Po)
    ArrayList<ComplexNum> yEven = FFT(polyEven);
    ArrayList<ComplexNum> yOdd = FFT(polyOdd);
    // Recursively calls FFT on the even degree terms and odd degree terms of the polynomial

    // y = [0] * n
    ArrayList<ComplexNum> y = new ArrayList<>(Collections.nCopies(n,new ComplexNum()));
    // Creates a y ArrayList containing n-amount of zeroes

    // for (j in range(n/2):
    for (int j = 0; j < n/2; j++) {
        // y[j] = ye[j] + ω^j yo[j]
        y.set(j, yEven.get(j).plus(new RootsOfUnity(j,n).z().times(yOdd.get(j))));
        // y[j+n/2] = ye[j] - ω^j yo[j]
        y.set(j+(n/2), yEven.get(j).minus(new RootsOfUnity(j,n).z().times(yOdd.get(j))));
    }
    // Calculates the point-value representation of the polynomial at the nth root of unity
    return y;
```

None other than the Fast Fourier Transform itself. The Fourier Transform is a mathematical transform that decomposes functions in the time domain (time in the *x*-axis) to the frequency domain (frequency in the *x*-axis). In this case, the Fast Fourier Transforms evaluates the coefficient representation of polynomials to its point-value representation.

The Java implementation does not vary much from the pseudocode, however modifications have to be made. For instance, since `int n` must be a power of 2 that is greater than the degree of the polynomial, the logarithm of the array list length representing the polynomial by 2 ($^2$log(len)) is first calculated, then it will be raised to its ceiling (ceil($^2$log(len))), lastly the integer 2 will be raised to the power of the resulting exponent (2^ceil($^2$log(len))). Other examples include the usage of for loops to add the polynomial coefficients to `ArrayList<Complex> polyEven` and `ArrayList<Complex> polyOdd` instead of the start-stop-step notation commonly found in Python code, as well as copying *n* amount of zeroes to `ArrayList<Complex> y` with the nCopies method from the Collections package.

**FFT_polynomMultiply()**

```
public static ArrayList<ComplexNum> FFT_polynomMultiply(ArrayList<ComplexNum> leftPoly, ArrayList<ComplexNum> rightPoly) {
    ArrayList<ComplexNum> leftPolyVal = FFT(leftPoly);
    ArrayList<ComplexNum> rightPolyVal = FFT(rightPoly);
    // Calculates the Fast Fourier Transform of each polynomial

    ArrayList<ComplexNum> multVal = new ArrayList<>();
    for (int i = 0; i < leftPoly.size(); i++) {
        multVal.add(leftPolyVal.get(i).times(rightPolyVal.get(i)));
    }
    // Multiplies the point-value representations of both polynomials together

    return IFFT(multVal);
    // Call IFFT on the point-value representation product
}
```

This function receives two ArrayList<ComplexNum> objects meant to represent polynomials, evaluates each polynomial in *O*(n*logn), perform point-value multiplication in *O*(n), and interpolate the resulting point-values to its polynomial function in *O*(n*logn).

**IFFT()**

```
public static ArrayList<ComplexNum> IFFT(ArrayList<ComplexNum> polyCoeff) {
    // n = len(P)
    double temp = Math.ceil(Math.log(polyCoeff.size())/Math.log(2));
    int n = (int) Math.pow(2,temp);
    // Take in an array of integers representing polynomial coefficients (n must be a power of 2)

    // if n == 1
    if (n == 1) {
        return polyCoeff;
    }
    // Base case: If array contains only one integer, return the integer without modification
    // ω = e^(-(2pi*i)/n) has been replaced with the RootsOfUnity class instance
    // Pe,Po = P[::2],P[1::2]
    ArrayList<ComplexNum> polyEven = new ArrayList<>();
    ArrayList<ComplexNum> polyOdd = new ArrayList<>();
        // Creates empty ArrayList objects to store even degree and odd degree coefficients
    for (int j = 0; j < polyCoeff.size(); j += 2) {
        polyEven.add(polyCoeff.get(j));
    }
        // Stores even degree coefficients to Pe
    for (int j = 1; j < polyCoeff.size(); j += 2) {
        polyOdd.add(polyCoeff.get(j));
    }
        // Stores odd degree coefficients to Po
    // Separate the polynomial to even degree terms and odd degree terms

    // ye,yo = IFFT(Pe),IFFT(Po)
    ArrayList<ComplexNum> yEven = IFFT(polyEven);
    ArrayList<ComplexNum> yOdd = IFFT(polyOdd);
    // Recursively calls IFFT on the even degree terms and odd degree terms of the polynomial

    // y = [0] * n
    ArrayList<ComplexNum> y = new ArrayList<>(Collections.nCopies(n,new ComplexNum()));
    // Creates a y ArrayList containing n-amount of zeroes

    // for (j in range(n/2):
    for (int j = 0; j < n/2; j++) {
        // y[j] = ye[j] + ω^j yo[j]
        y.set(j, yEven.get(j).plus(new RootsOfUnity(j,n).z().reciprocal().times(yOdd.get(j))));
        // y[j+n/2] = ye[j] - ω^j yo[j]
        y.set(j+(n/2), yEven.get(j).minus(new RootsOfUnity(j,n).z().reciprocal().times(yOdd.get(j))));
    }
    // Calculates the point-value representation of the polynomial at the nth root of unity
    return y;
```

Reverses the Fast Fourier Transform. The IFFT transforms functions in the frequency domain back to the time domain. In the context of polynomials, it interpolates the point-value representation of the polynomials back to the polynomial function itself.

Because the Inverse Fast Fourier Transform is essentially the same with the Fast Fourier Transform, with only 2 minor changes ($\omega_j$ is in reciprocal form, and divide the final result by $n$), the modifications made in the FFT Java code is also exactly the same in the IFFT Java code.

**Externals**

*ComplexNum.java*

```java
package externals;

public class ComplexNum {
    private double re;   // Real component
    private double im;   // Imaginary component

    // Create new ComplexNum object with default parameters
    public ComplexNum() {
        this.re = 0;
        this.im = 0;
    }

    // Create new ComplexNumNum object with the provided real and imaginary components
    public ComplexNum(double real, double imag) {
        this.re = real;
        this.im = imag;
    }

    // Return string representation of the invoked ComplexNum object
    public String toString() {
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im <  0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
    }
```

*"ComplexNum"* is a public class implementation of complex numbers modified from an already existing program created for an introductory computer science class in Princeton University, with an assortment of functions for addition, subtraction, multiplication, division, and many other operations. It has two attributes (the real component and the imaginary component), a default constructor, and a toString implementation of the instances.

**plus()**

```java
// Returns a new ComplexNumNum object whose value is (this + that)
public ComplexNum plus(ComplexNum that) {
    double real = this.re + that.re;
    double imag = this.im + that.im;
    return new ComplexNum(real, imag);
}
```

Performs addition between two `ComplexNum` instances.

**minus()**

```
// Returns a new ComplexNum object whose value is (this - that)
public ComplexNum minus(ComplexNum that) {
    double real = this.re - that.re;
    double imag = this.im - that.im;
    return new ComplexNum(real, imag);
}
```

Performs subtraction between two `ComplexNum` instances.

**times()**

```
// Returns a new ComplexNum object whose value is (this * that)
public ComplexNum times(ComplexNum that) {
    double real = this.re * that.re - this.im * that.im;
    double imag = this.re * that.im + this.im * that.re;
    return new ComplexNum(real, imag);
}
```

Performs multiplication between two `ComplexNum` instances.

**reciprocal()**

```
// Returns a new ComplexNum object whose value is the reciprocal of this
public ComplexNum reciprocal() {
    double scale = re*re + im*im;
    return new ComplexNum( real: re / scale,  imag: -im / scale);
}
```

Returns the reciprocal of an `ComplexNum` instance.

**divides()**

```
// Returns a new ComplexNum object whose value is (this / that)
public ComplexNum divides(ComplexNum that) { return this.times(that.reciprocal()); }
```

Performs division between two `ComplexNum` instances.

**re()**

```
// Returns the real component
public double re() { return this.re; }
```

Returns the real component of an `ComplexNum` instance.

**im()**

```
// Returns the imaginary component
public double im() { return this.im; }
```

Returns the imaginary component of an `ComplexNum` instance.

**raised()**

```
// Returns a new ComplexNum object whose value is (this ^ that)
public ComplexNum raised(int that) {
    ComplexNum pow = new ComplexNum( real: 1, imag: 0);
    for (int i = 1; i <= that; i++) {
        pow = pow.times(this);
    }
    return pow;
}
```

Performs exponentiation between an `ComplexNum` instance as the base and an integer as the power.

*RootsOfUnity.java*

```java
package externals;

public class RootsOfUnity extends ComplexNum {
    private double k;
    private double n;
    private ComplexNum z;
```

*"RootsOfUnity"* is another public class implementation similar to ComplexNum, although dedicated towards the roots of unity. Two integers can be passed into the constructor, integer *k* and integer *n*, which denotes the *k*th point of the *n*th root of unity.

```java
// Returns the kth point
public double k() { return this.k;}

// Returns the nth root
public double n() { return this.n;}

// Returns the ComplexNum z at the kth point of the nth root
public ComplexNum z() { return this.z;}
```

**k()**

Returns the *k*th point of a `RootsOfUnity` instance.

**n()**

Returns the *n*th root of a `RootsOfUnity` instance.

**z()**

Returns the `ComplexNum` of a `RootsOfUnity` instance at the *k*th point and the *n*th root.

## VII. Project Link

https://github.com/Bernard-Choa/oop-finalproject-sem-2

## VIII. References


https://introcs.cs.princeton.edu/java/97data/Complex.java.html

https://introcs.cs.princeton.edu/java/32class/RootsOfUnity.java.html

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

https://www.tutorialspoint.com/java/lang/java_lang_math.htm

https://www.softwaretestinghelp.com/java-arraylist-tutorial/

## IX. Further Reading

Ahamed, S. F., Rao, G. S., & Ganesh, L. (2016). Fast acquisition of GPS signal using FFT decomposition. *Procedia Computer Science*, *87*, 190-197. https://doi.org/10.1016/j.procs.2016.05.147

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Desmos. (n.d.). *Graphing calculator*. https://www.desmos.com/calculator

Elster, A. D. (2001). *Fourier transform (FT)*. Questions and Answers in MRI. https://mriquestions.com/fourier-transform-ft.html

*Faster fourier transform named one of world's most important emerging technologies*. (2012, May 7). MIT News | Massachusetts Institute of Technology. https://news.mit.edu/2012/faster-fourier-transform-named-one-of-worlds-most-important-emerging-technologies

NVIDIA. (2005, April). *Chapter 48. Medical image reconstruction with the FFT*. NVIDIA Developer. https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-48-medical-image-reconstruction

Reducible. (2020, November 15). *The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?* [Video]. YouTube. https://www.youtube.com/watch?v=h7apO7q16V0&t=261s

Serov, V., & Sergeeva, T. (2016). Solution of the schrodinger equation using exterior complex scaling and fast fourier transform. *Mathematical Modelling and Geometry*, *4*(1). https://doi.org/10.26456/mmg/2016-411

Sun, M. (2021, January 20). *How fourier transform speeds up training CNNs*. Medium. https://towardsdatascience.com/how-fourier-transform-speeds-up-training-cnns-30bedfdf70f6

ThermoFisher. (n.d.). *Fourier transform infrared spectroscopy (FTIR) Academy | Thermo Fisher scientific*. Thermo Fisher Scientific - US. https://www.thermofisher.com/id/en/home/industrial/spectroscopy-elemental-isotope-analysis/spectroscopy-elemental-isotope-analysis-learning-center/molecular-spectroscopy-information/ftir-information.html

Welch Labs. (2015, August 28). *Imaginary Numbers Are Real [Part 5: Numbers are Two Dimensional]* [Video]. YouTube. https://www.youtube.com/watch?v=65wYmy8Pf-Y&list=PLiaHhY2iBX9g6KIvZ_703G3KJXapKkNaF&index=5

Wood, W. (2021, 6). *The Vandermonde Matrix and Polynomial Interpolation* [Video]. YouTube. https://www.youtube.com/watch?v=Cov_kLatdlc