

Polynomial Root Finding

Jack Fox

Degree Programme: Computer Science with Business and Management

Supervisor: Dr T. Leonard Freeman

3rd May 2011

This is a project report for a third year project done in the School of Computer Science at the University of Manchester.

Abstract

Polynomial Root Finding

Author: Jack Fox

Supervisor: Dr T. Leonard Freeman

This project aims to implement a polynomial root finding algorithm. The project aims to do this by implementing both the Durand-Kerner method and the Aberth method. A program was written that achieved this and was capable of displaying the results through a graphical user interface as well as plotting the outcome of each iteration on a graph. The results produced by the program were accurate, even though the program was tested using cases that are known to be difficult for these methods. The Aberth method required less iterations to achieve the required accuracy which was expected as it is a higher order method and should converge faster. However, there were some interesting cases that emerged.

This report documents the development of the algorithm and investigates the results that are found.

Acknowledgements

I would like to thank Dr T. Leonard Freeman for his support throughout this project. Without his knowledge and guidance, the project would not have been as successful.

1 Introduction

1.1 Existing work in the area	5
1.2 Aims/objectives	5
1.3 Report summary.....	6

2 Background

2.1 The Newton-Raphson method	7
2.2 Durand-Kerner method	8
2.3 Aberth method	9
2.4 Evaluation of a polynomial.....	9
2.4.1 Horner's method	9

3 Design

3.1 Programming Language	10
3.2 Structure.....	10
3.3 Input and Output.....	11

4 Implementation

4.1 Polynomial representation.....	13
4.2 Complex numbers	13
4.3 Initial approximations	14
4.4 Horner's method	15
4.5 Durand-Kerner method	15
4.6 Aberth method	16
4.6.1 Differentiation	17
4.7 Graphical User Interface	18

5 Testing and Results

5.1 Testing	19
5.2 Results	25
5.2.1 The Wilkinson Polynomial	27

6 Conclusions

6.1 Evaluation of Proposed Features	29
6.2 Evaluation of proposed plan	29
6.3 Areas For Further Development.....	30

Chapter 1 – Introduction

For centuries, Mathematicians have been both studying and using root-finding algorithms. Perhaps the best known root-finding algorithm would be the one named after Isaac Newton and Joseph Raphson (the Newton-Raphson method). This method was first suggested in 1669 and was based on the idea of continuously improving the estimate for a single root of any non-linear function before repeating the process on the next root.

E.Durand and I.Kerner constructed an algorithm in the 1960's that took a different approach by improving the estimates for all of the roots of a given polynomial during each iteration[1][2]. The algorithm built on ideas and formulas used by Karl Weierstrass(1815-1897)[3] to allow simultaneous root-finding which converges quadratically for simple roots and linearly for repeated roots. Oliver Aberth then further developed the Durand-Kerner method to offer an alternative method that used a different updating process from the Newton-Raphson method to give simultaneous root-finding with cubic convergence[4]. The goal of this project is to implement both the Durand-Kerner method and the Aberth method so that both methods can be analysed and their performance can be evaluated.

1.1 Existing work in the area

A root finding algorithm has already been implemented using the Durand-Kerner algorithm by T.E.Hull and R.Mathon. They explored the case of the root finder trying to find multiple roots of a polynomial. It is widely known that this is the worst case for the Durand-Kerner method causing the algorithm to converge linearly to a multiple root. Hull and Mathon proved that taking the mean of individual approximations to a multiple root will cause the algorithm to converge quadratically to that root rather than linearly[6].

1.2 Aims/objectives

The main aim of this project is to create and develop a program that uses both the Durand-Kerner and Aberth methods to calculate the roots of a given polynomial of any degree to a given accuracy.

In addition to this, the program will have the following features:

- To be able to read both real and imaginary coefficients of a polynomial.
- To be able to take input through a graphical user interface.
- To be able to take input from file.
- To be able to detect errors in user input.
- To be able to present the results.
- To be able to present the estimate of the roots after each iteration.
- To show how the estimates change by plotting the points on a graph.
- To be able to show how many iterations were needed.
- To be able show the convergence of different methods for a given root by plotting a graph of the error in each estimate of each method.

1.3 Report summary

This report will consist of seven main chapters. A description of each chapter is given below:

- 1) Introduction – a brief introduction giving details about the project.
- 2) Background - background on the methods and mathematics implemented in the program.
- 3) Design – description of the design of the program.
- 4) Implementation – description of the implementation of the program.
- 5) Results – results produced by the program.
- 6) Testing and Evaluation – details of testing and an analysis of results.
- 7) Conclusion – a short summary of the project and potential future work to be done.

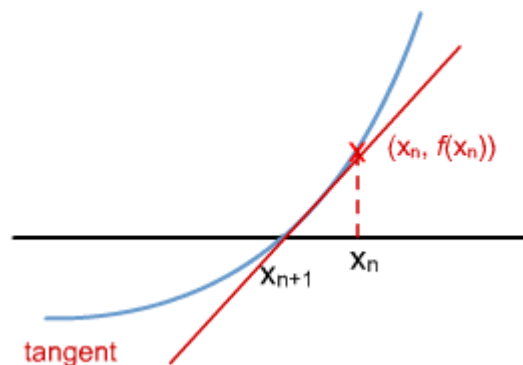
Chapter 2 – Background

A polynomial is a mathematical expression that has been constructed from variables and constants using only addition, subtraction, multiplication and non-negative integer exponents. For example, $x^2 + 3x + 2$ is an example of a polynomial whereas $\frac{1}{x^3} + 6x^{-2} + 3$ is not an example of a polynomial because the variable is in the denominator of the first term and the variable in the second term has a negative exponent. The roots of a polynomial are those values that can be substituted for x so that the polynomial evaluates to zero. That is the values of x such that $f(x) = 0$. In the first example the values that make $x^2 + 3x + 2 = 0$ are $x = 1$ and $x = 2$. This is an example of a basic polynomial; all of its roots are integers. Many polynomials have either roots or coefficients that are complex numbers.

The number of roots that exist for a given polynomial is equal to the value of the highest exponent – known as the degree - contained in the polynomial. Finding the roots of a polynomial is no easy task. For polynomials of degree four or lower, there are formulas that give the roots of a polynomial. However, there is no formula for the roots of polynomials of a degree higher than four so an iterative approach must be used to determine the roots of a polynomial.

2.1 The Newton-Raphson method

The Newton-Raphson method is not difficult to comprehend and is best explained by using a graph of the given polynomial. It takes the graph of the polynomial and uses an initial estimate that is relatively close to the root. The method uses the gradient of the polynomial at this estimate and creates a tangent to the polynomial at the estimate. The x-intercept of this tangent line is then used as a better estimate of the root. The x-intercept can then be used as the next initial estimate and the process can be repeated. This method is illustrated by the diagram below:

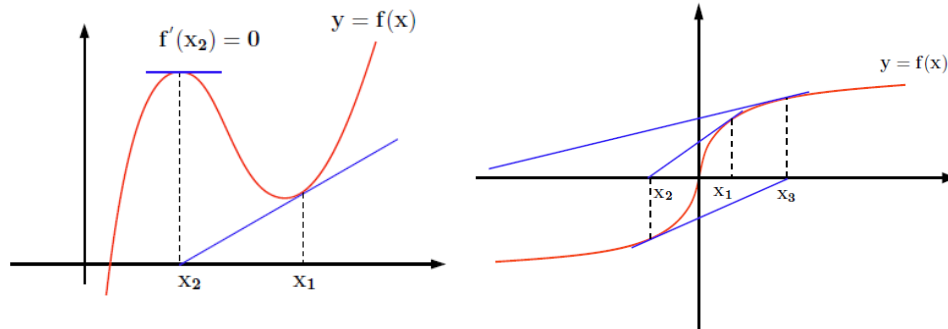


We know that $f'(x_n) \simeq \frac{\Delta y}{\Delta x} = \frac{f(x_n)}{x_n - x_{n+1}}$ where f' is the derivative of the function f .

With some simple algebra, this equation can be rearranged to give an iterative formula that calculates the x-intercept of the tangent, x_{n+1} , and therefore the improved estimate of the root.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

For most polynomials, the estimate of the root will improve as more iterations are performed. However, there are cases when the Newton-Raphson method is flawed. For example, if the initial approximation chosen is not close enough to the desired root, the iterative formula may converge to another root of the polynomial. Also, this method fails if the gradient of the polynomial is zero when an estimate is evaluated as in this case, the tangent is a horizontal line which does not intercept the x-axis. The diagram below left shows this case. The diagram below right shows the method failing to converge because of the fast changing gradient of the polynomial.



2.2 Durand-Kerner method

The Durand-Kerner method uses a different approach to the Newton-Raphson method in that a single iteration improves the estimation of all of the roots of a polynomial. This method is a second order method and uses formulas developed by Karl Weierstrass in 1891 although at the time they were not being used for polynomial root finding[5].

A polynomial:

$$P(x) = c_0 + c_1x + \dots + c_{n-1}x^{n-1} + c_nx^n$$

where the c 's are generally complex coefficients and $c_n \neq 0$, can be written in terms of its roots as

$$P(x) = c_n(x - \zeta_1)(x - \zeta_2) \dots (x - \zeta_n)$$

Where $\zeta_1, \zeta_2, \dots, \zeta_n$ are the (generally complex) roots of $P(x)$.

If the complex numbers $x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}$ are reasonable approximations to the roots ζ_1, \dots, ζ_n of $P(z)$ then consider the following iteration formulas[6]

$$x_i^{(k+1)} = x_i^{(k)} - \frac{P(x_i^{(k)})}{Q'(x_i^{(k)})} \text{ where } 1 \leq i \leq n \text{ and } k = 0, 1, \dots$$

where

$$Q(x_i^{(k)}) = c_n(x - x_1^{(k)})(x - x_2^{(k)}) \dots (x - x_n^{(k)})$$

These are the Weierstrass-Durand-Kerner(WDK) formulas.

Here, we have $Q'(x)$ which is an approximation to $P'(x)$. The differentiation of $Q(x)$ is trivial.

$$\begin{aligned} Q'(x_i^{(k)}) &= (x_i^{(k)} - x_1^{(k)})(x_i^{(k)} - x_2^{(k)}) \dots (x_i^{(k)} - x_{i-1}^{(k)})(x_i^{(k)} - x_{i+1}^{(k)}) \dots (x_i^{(k)} - x_n^{(k)}) \\ &= \prod_{\substack{j=1 \\ j \neq i}}^n (x_i^{(k)} - x_j^{(k)}) \end{aligned}$$

So another way of writing the WDK formula as an iterative formula would be:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{P(x_i^{(k)})}{\prod_{\substack{j=1 \\ j \neq i}}^n (x_i^{(k)} - x_j^{(k)})}$$

As this formula iterates, the fraction in the formula quickly becomes smaller as the numerator gets smaller. This means that a smaller change is being made to the estimate at each iteration. However, the denominator ($Q'(x_i^{(k)})$) can be small when two roots are close together. This is because a term

$(x_i^{(k)} - x_j^{(k)})$ would be small when two roots are close to each other, making the whole product smaller. In practice this can be useful because the method would tend not to converge to a root multiple times if one or more roots are close together. However, this can also be a disadvantage of the method when a polynomial has the same root repeated two or more times. This is because the method tries to push the estimates away from each other. This case gives the method its slowest convergence for multiple roots.

2.3 Aberth method

The Aberth method is another technique for finding the roots of a polynomial. This method is a third order method. The iterative formula given by Oliver Aberth is:

$$x_i^{(k+1)} = - \frac{\frac{P(x_i^{(k)})}{P'(x_i^{(k)})}}{1 - \frac{P(x_i^{(k)})}{P'(x_i^{(k)})} \cdot \sum_{\substack{j=1 \\ j \neq i}}^n \frac{1}{x_i^{(k)} - x_j^{(k)}}}$$

For a polynomial $P(x)$ where $x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}$ are the current approximations of the polynomials roots.

This method is more efficient than the Durand-Kerner method and converges cubically.

2.4 Evaluation of a polynomial

2.4.1 Horner's method

Horner's method provides a way to evaluate a polynomial for a given value of the independent variable. This can be useful when applying root finding algorithms as many of the algorithms require a polynomial to be evaluated at each iteration. Horner's method can be thought of as nested multiplication. For example a fifth degree polynomial:

$$P(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Could be re-written in the form:

$$P(x) = \left(\left(\left((a_5x + a_4)x + a_3 \right)x + a_2 \right)x + a_1 \right)x + a_0$$

Re-writing a polynomial in this form is an advantage because we do not need to multiply a value by itself n times, instead we are able to evaluate the polynomial for a value using only basic addition and multiplication.

Chapter 3 – Design

3.1 Programming Language

The choice of programming language for this project is java. Java does not have any built in class that can handle complex numbers but a class can be written which implements complex numbers and will provide simple operations for complex numbers such as addition, subtraction, multiplication, division and printing complex numbers. Java is an object-oriented language that was designed with portability in mind, this means the program will run on most current day operating systems and can be exported as an applet to be used on a web page. A wide range of third party packages are available for java which could be useful, particularly when implementing the graphical interface which is required. Working in java also means that there will be an easy to work with class system. A disadvantage of using java is its execution speed since it is not as fast as either C or C++. This is because java compiles code to run on the java virtual machine whereas both C and C++ compile machine code for a system. However, modern processors can execute java much faster than was previously possible and the speed of execution is not so important for this project.

3.2 Structure

It is important that the program is robust. To achieve this, the program needs to be able to deal with errors in input that may result from the user's mistakes. If the program is not able to deal with these errors, it will crash which is not desirable. Instead, it is preferable for the program to recognise an error in input and report the error back to the user and ask them to correct it. It may require a large amount of code for this to be made possible as there may be many opportunities for input from the user through the graphical user interface. However, it is important for the program to be robust.

It is important that the structure of the code is well designed. Having all of the code within one main method is undesirable and inefficient. As previously mentioned, there will be a separate class to handle complex numbers and any operations performed on complex numbers. The class will also need to be able to construct a complex number, print a complex number, convert between polar and cartesian form, calculate the distance between the complex number and the origin, return the real or imaginary part of a complex number and return the modulus or argument of a complex number.

There will also be a separate class which will deal with the graphical output of the program. This will be necessary because many class imports may be needed to produce a graphical output and the class will contain many lines of code. Within the main class, there will be the main method which will control the program and call all other methods. There will be a method in the main class for the Durand-Kerner and Aberth methods. There will also be methods to assist these methods such as a method that will differentiate a polynomial which is required for the implementation of the Aberth method and an implementation of Horner's method. There will also be a method to calculate the initial approximations for the roots of a polynomial and a method to report the results back to the user. Having well organised code will make it easier to identify any problems during development and will also make it easier for someone other than the programmer to understand the functionality of the code.

Part of the design of the program involves consideration of the different types that will be needed by the program to store variables. The best estimations of the roots of the polynomial will be stored in an array of complex numbers as will the coefficients of the given polynomial. This is because both

the number of roots and the number of coefficients will be known once the degree of the polynomial has been specified by the user. To produce a graphical output, it will be necessary to give the results of each iteration to the graph-plotting method. For this to be feasible there will need to be a global variable that stores the results of each iteration. Since we do not know how many iterations each method will require to converge to the required accuracy, the most efficient way to store the iterates is to use a linked list of complex numbers. Using a linked list allows the storage of an indefinite number of variables as it just links one more variable to the end of the list. Java has a `toArray()` function that takes a linked list and returns an array with all of the elements in order which will be useful if any manipulation of the linked list is required.

3.3 Input and Output

An important part of the design is the graphical user interface. There will be two stages of user input followed by the presentation of results to the user. The first window presented to the user requires the user to enter the degree of the polynomial to be solved and the desired accuracy of the roots. At this stage, the different methods available to solve the polynomial will be available for the user to select. They will have a checkbox by their titles so that the user can select one or more methods to solve the polynomial. In addition, there will be a “next” and a “cancel” button at the bottom of the window. For the program to be robust, checks should be carried out to confirm that the degree of the polynomial is an integer, the required accuracy is a real number and that at least one method is selected to solve the polynomial. Also, at this stage the user will have the option to read a polynomial into the program from file. A “read from file” checkbox and a “choose file” button will allow the user to navigate to the text file containing the coefficients of the desired polynomial. If this option is chosen, the program will then present the results to the user.

Using the information collected at the first stage of user input, a second window will be created once the user has clicked “next” and there are no errors in input. The second window will dynamically create text fields for the user to input the coefficients of the polynomial given that the degree of the polynomial is known. Again, there will be “next” and “cancel” buttons at the bottom of the window. For the program to be robust, it must be checked that each coefficient that the user enters is a real number and not a letter or a symbol. Once the user clicks the “next” button, the program will store the coefficients in a complex number array and no further input is required from the user for the program to run. Any coefficients that are left blank will be interpreted as being zero.

An important part of the design is choosing the initial approximations to the roots of the polynomial. The initial approximations will be evenly spaced points around the circumference of a unit circle centred at (0,0) on the complex plane. The degree of the inputted polynomial will determine how many initial roots are chosen and how far apart they will be. The program will have to calculate what these points are and then store them in an array of complex numbers as well as add them as the first entries in the linked list that stores the results of each iteration.

There are four possible output windows that will be simultaneously presented to the user. The first will be a simple window stating the roots of the polynomial given by each method used and the number of iterations each method took to get to these roots. The second window will be a graph of the complex plane showing the convergence of roots from their initial approximations using the Durand-Kerner method (if this method has been selected by the user at the first stage of input). The third window will be the same as the second except that it will show the convergence of the roots from their initial approximations using the Aberth method if this has been selected by the user at the first stage of input. The fourth window will be a graph of the error in a root plotted against the number of iterations. It will be plotted for either, or both, the Durand-Kerner method and the Aberth method. This fourth graph will show how the error in an approximation decreases for a root

of the polynomial. The root chosen will be the first root in the array of solved roots once the program has solved all roots to the required accuracy.

Chapter 4 – Implementation

4.1 Polynomial representation

The first item to consider in the implementation process is how a polynomial will be represented within the program. Because of the design of the graphical user interface, the program will know the degree of the polynomial before the coefficients of the polynomial are inputted. This is useful because it means that an array can be created to store the coefficients of the polynomial as the required size of the array is known. The array will need to be an array of complex numbers to allow the user to input complex numbers as coefficients. If the user leaves the text field of a coefficient blank, the program will assume that the coefficient of that power is zero. The polynomial will be represented by this array and each index of the array will store the coefficient of that power of x . So the coefficient of a term x^i will be stored at index i of the complex number array.

4.2 Complex numbers

The choice of using java means that the program will have to have its own way of handling complex numbers. To do this, the program will have a separate method that has a constructor method for complex numbers as well as several functions which will manipulate complex numbers. The constructor method for complex numbers is simple; a complex number is made a real value and an imaginary value. Both of these values are of type double. There will be the trivial methods `getRe()`, `getIm()`, `setRe()` and `setIm()` which will allow the program to find the values of the real and imaginary parts of a complex number and set the real and imaginary parts of a complex number respectively.

The method “`getConjugate`” will take a complex number and multiply the imaginary part by negative one so that it can return the complex conjugate of the given complex number.

The methods “`add`” and “`sub`” are also trivial as they simply add the real and imaginary parts of two numbers and subtract the real and imaginary parts of two numbers respectively before returning the result. The multiply function is slightly less trivial as the real part of the result is calculated by subtracting the product of the imaginary parts from the product of the real parts and the imaginary part of the result is calculated by adding the result of the two multiplications of the real parts and imaginary parts of the inputs.

The division method is not as simple as the other arithmetic functions but can be described in following way:

$$\frac{(a + bi)}{(c + di)} = \frac{(ac + bd) + (bc - ad)i}{(c^2 + d^2)}$$

Thinking about the division in this way makes it easier to implement than standard complex number division because this provides a general formula that will not need to call the `getConjugate()` function.

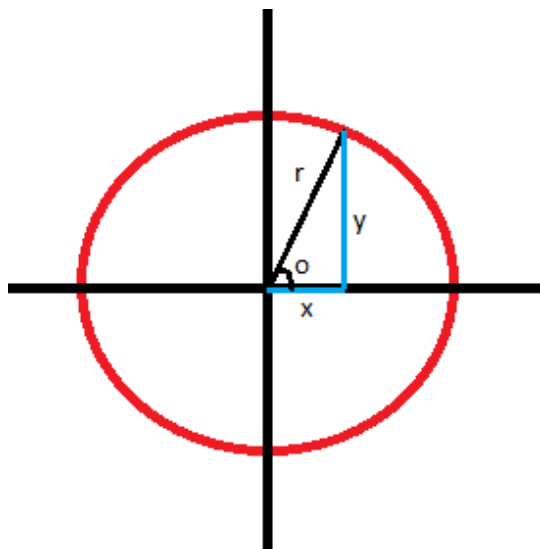
In some cases, the program may need to interpret complex numbers that are in a different form. For this program, that standard form will be rectangular form so it is necessary to have functions that can convert complex numbers to this form. A complex number may be in polar form when the initial approximations to roots are taken because for any given root, only the distance from the point (0,0) and the angle that it makes with the real axis is known. In this case, the function “`fromPolar`” will take the angle and distance and calculate the number in rectangular form using java’s math package which contains both sine and cosine functions.

The complex class will need to be able to take a string and create a complex number if there are two numbers within the string. This will be useful when reading input from a file. The two numbers can be separated by splitting the string into two parts where there is a comma. This can be achieved using java's split method for strings. Once the string has been split, the program can try to parse two numbers of type double from the two strings. These two doubles can then be used to create a complex number which is returned. This method assumes that each string has two numbers within it which are separated by a comma so this has to be the standard way that complex coefficients are inputted from a file.

The final two methods of the complex class will deal with presenting a complex number in a form that is understandable to the user. This means taking a complex number and converting it to a string. There will be two methods to do this, one that will take a single complex number as an input and one which will take an array of complex numbers as an input. The two methods will be similar; the only difference being that for a complex array, the process will be repeated for each complex number in the array. The code to do this is not intricate, it simply takes the real and imaginary parts and presents them in the form " $x + yi$ ".

4.3 Initial approximations

The next main problem that the program will have to overcome will be choosing initial approximations to the roots of the polynomial. To do this, the program needs to find n numbers that are equally spaced around the circumference of a unit circle centred at (0,0). To do this, there will be a method called "initialise". This method will use n - the degree of the polynomial - as an input and will return an array of complex numbers that will contain n estimations to roots. The method will have to divide the 2π radians of the unit circle by the degree of the polynomial in order to determine the angle between each of the numbers. Once the angle has been calculated, this can be used with the length of the hypotenuse - which will be of length one - to form a right-angled triangle to calculate each of the initial root approximations. This calculation can be illustrated by the following diagram:



Where r is the radius of the circle (which is always one), O is the result of the division of 2π by the number of roots and x and y are the real and imaginary parts of the estimation respectively.

This method can be represented by the following pseudo-code:

```
Public static Complex[] initialise(int degree, Complex[] roots)
    Calculate the angle between roots by dividing  $2\pi$  by degree
    For each root
        Use the cos function to find the real part of the complex number
        Use the sin function to find the imaginary part of the complex number
        Make a complex number from the two parts and store it in the roots array
        Add the current angle value to the original angle value to give the angle of
        the next approximation from the real axis
    Return roots
```

4.4 Horner's method

Because the methods to be implemented require the evaluation of a polynomial, it would be advantageous to have a separate method for Horner's method that will evaluate a given polynomial at a given value. The following pseudo-code describes how Horner's method will do this:

```
Public static Complex Horner(Complex coefficients[], Complex x)
    Set the result equal to the coefficient of the highest term
    While there is a next coefficient
        Result = result * x
        result = result + next coefficient
    return result
```

The code for this method is a very simple way to evaluate a polynomial. It does assume that the coefficient of the highest power of x is not zero but the first stage of user input will make sure that this does not happen.

4.5 Durand-Kerner method

The next step of implementation is to implement the Durand-Kerner method. This should not be difficult as it is already known the iterative formula to be implemented:

$$x_i^{(k+1)} = x_i - \frac{P(x_i^{(k)})}{\prod_{\substack{j=1 \\ i \neq j}}^n (x_i^{(k)} - x_j^{(k)})}$$

Pseudo code for this iterative formula can be used to show how this iterative formula will be implemented:

```

Public static Complex[] DurandKerner(Complex[] roots, Complex[] coefficients,
LinkedList<Durand>)
    Increase the amount of iterations by one
    for each root, execute a for loop
        evaluate the polynomial for the current root using Horner's method
    for each root execute a for loop
        if the two roots are not equal
            subtract the two root estimations
                multiply with the current value of the denominator
            divide the numerator result by the denominator result
            subtract the fraction from the current estimation of the root
            store the result in the roots array
            add the result to the end of the linkedlist
    use Horner's method to find the residual when the original polynomial is evaluated
    using the current estimate
    if this modulus of this residual is greater than the minimum acceptable error
        call the DurandKerner method
    else
        return roots array

```

This method is recursive as it calls itself if the approximations to the roots are not accurate enough. Using recursion means that the code is simple and easy to understand. However, recursion can require large amounts of memory if the recursion is very deep. For this case though, it is not expected that this will be a problem because the Durand-Kerner method converges quadratically in the worst case.

4.6 Aberth method

The implementation of the Aberth method can be done in a similar way. A single method in the class will be used to implement the Aberth method so that it is easily identifiable and is easily implementable from the main method. Having a method that takes the coefficients of a polynomial and returns a linked list of all the iterations along with the array of the estimations to the roots will be the way that root finding algorithms will be implemented in this program so that it is easy to use as many methods as necessary from the main method. The Aberth method requires the following iterative method to be implemented:

$$x_i^{(k+1)} = - \frac{\frac{P(x_i^{(k)})}{P'(x_i^{(k)})}}{1 - \frac{P(x_i^{(k)})}{P'(x_i^{(k)})} \cdot \sum_{\substack{j=1 \\ j \neq i}}^n \frac{1}{x_i^{(k)} - x_j^{(k)}}}$$

The following pseudo-code describes the how the Aberth method can be implemented:


```

Public static Complex[] Aberth(Complex[] roots, Complex[] coefficients,
LinkedList<Aberth>)
    Increase the amount of iterations by one
    Create a differentiated polynomial
    For each root, execute a for loop
        Evaluate the original polynomial for the current estimate
        Evaluate the differentiated polynomial for the current estimate
        Calculate the numerator of the formula by dividing the evaluation of
        the original polynomial by the differentiated polynomial

        for each root, execute a for loop
            if the root is not the same as the root being used by the
            outer for loop
                subtract the two roots
                divide one by the result of the subtraction
                add the result of the division to the total so far
            multiply the total calculated by the for loop and multiply it by the
            already calculate numerator value
            subtract the result of this from one to give the value of the
            denominator
            divide the numerator value by the denominator value
            multiply this by negative one to give the result of an iteration
            add the result to the end of the linked list
            update the estimate of the root
        use Horner's method to find the residual when the original polynomial is
        evaluated using the current estimate
        if this modulus of this residual is greater than the minimum acceptable error
            call the Aberth method
    else
        return the roots array

```

This implementation uses the same way of checking errors as the Durand-Kerner implementation.

4.6.1 Differentiation

The Aberth method requires the evaluation of a second polynomial which is the derivative of the original polynomial. There will be a separate method called “differentiate” which will take the array containing the coefficients of the inputted polynomial and return an array one less in size that will contain the coefficients of the differentiated polynomial. This method may be useful if any other root finding methods are implemented that require the evaluation of the derivative of the inputted polynomial. This method can be described by the following pseudo-code:

```

public static Complex[] differentiate(Complex[] coefficients)
    create a new array of complex numbers that is one shorter in length than the input
    for each coefficient from the input except the coefficient of  $x^0$ 
        multiply the current coefficient by the power of  $x$  that it is bound to
        store the result in the new array bound to a power of  $x$  that is one less than
        it was originally bound to
    return the new array

```

The Aberth method uses this method to differentiate a polynomial before using the already described implementation of Horner's method to evaluate for a particular value.

4.7 Graphical User Interface

The first stages of the graphical user interface that gather information from the user can be created by using the libraries that are already built into the java programming language. The JFrame class will be used to create windows and instances of JLabel and JTextField will be created dynamically to ask the user for input and take input from the user. In the initial window presented to the user, there will also be instances of Checkbox which will allow a user to select one or more methods to find the roots of a polynomial. In the same screen, JFileChooser will also be implemented which will allow a user to input coefficients of a polynomial from a text file. At the bottom of each window, the JButton class will be invoked to create the buttons "next" and "cancel" as the user is guided through a wizard-like input system. The instances of JButton will implement the ActionListener class. The use of the ActionListener class will mean that actions can be performed once the user clicks on one of the buttons. This will enable the storage of variables inputted by a user into any JTextField or Checkbox and will also prompt the program to dispose of the current window that is visible to the user. JFrame and instances of JLabel will also be dynamically created for the final window that will be presented to the user. This window will report the final estimations of the roots for each method used as well as the number of iterations that were required to achieve the desired accuracy.

The choice of using java means that there are a lot of class packages that have been created by third parties in order to assist programmers with development. This will be useful for the program being created as there are many third party libraries that assist with graph plotting. By using a library developed by a third party, the program will be able to display the required graphs that show convergence in a way that is easier to understand and more aesthetically pleasing than what would be possible otherwise. JFreeChart is the package that will be used in this application as it supports a wide range of chart types and has a flexible design that is easy to extend. JFreeChart also has a consistent and well-documented API which will be useful when exploring the capabilities of the package.

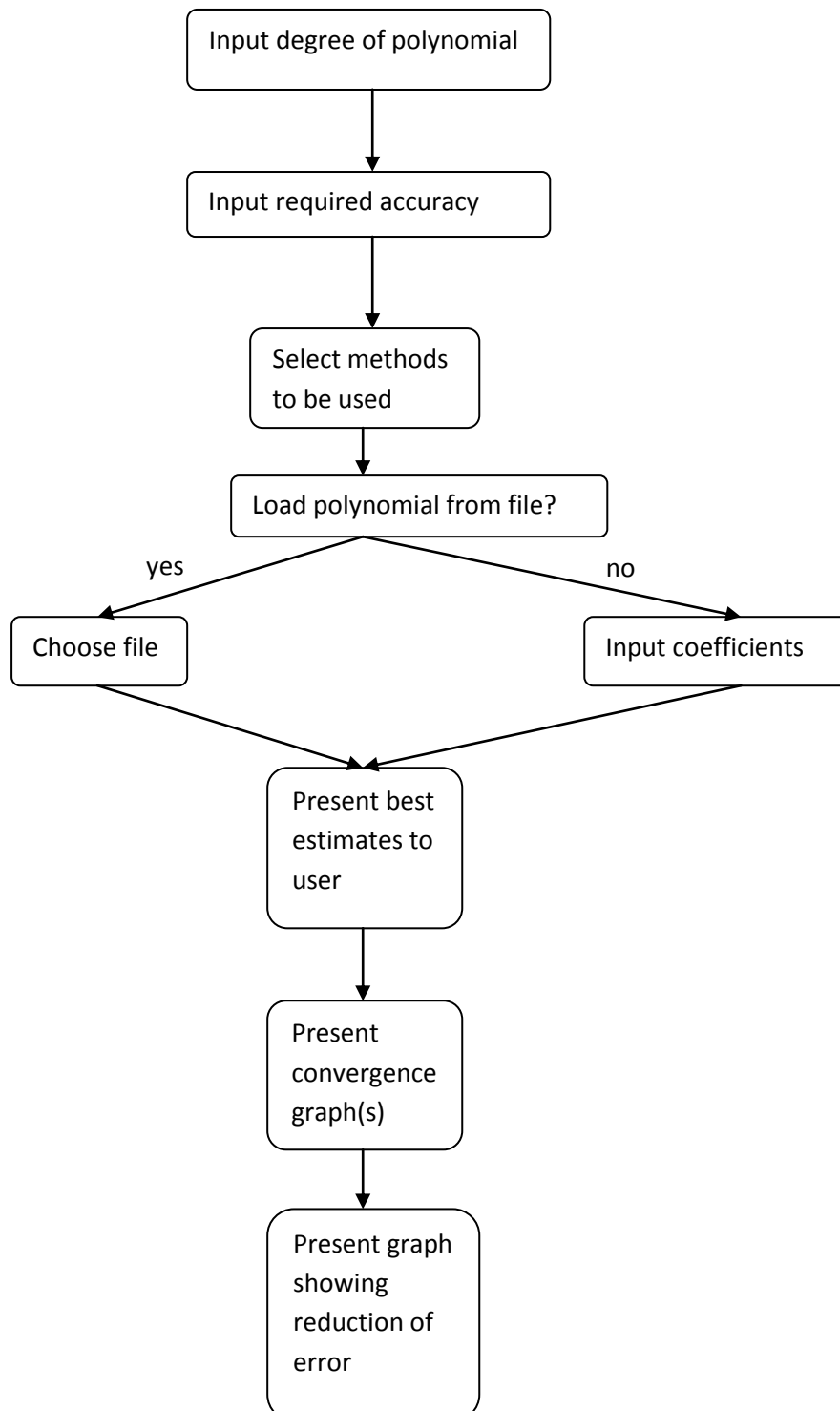
To create the graphs that are required by this program, JFreeChart requires the data to be plotted to be stored in a dataset. A dataset is a collection of series' and this is trivial for the program as the linked list containing all the data from the iterations can be converted to an array using java's toArray() function. The resulting array can then be used to create a series for each root of the polynomial. Each series will store all of the approximations for a root and the series' will make up the dataset so that the convergence of each root is plotted using a separate line. If the user selects more than one root finding algorithm to be used, the program will produce a graph for each method used which will show how the roots converge for each method.

The aim is for the code to have high cohesion and low coupling which means that the code will have well structured methods and classes while being both flexible and easily extendable.

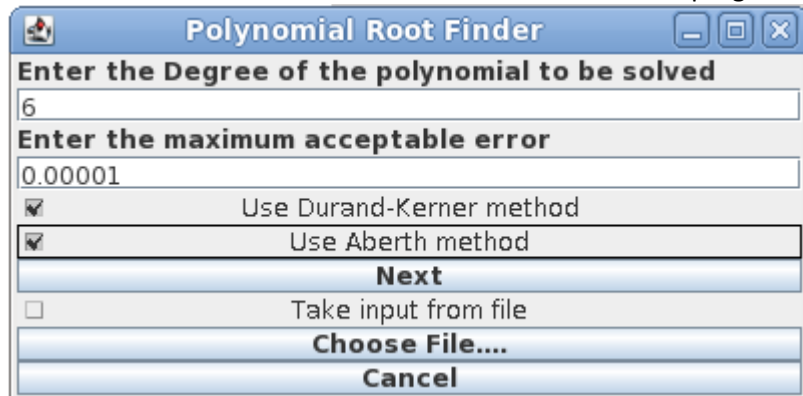
Chapter 5 – Testing and Results

5.1 Testing

The way that the user interacts with the program is a component that needs to be tested. Through this, the robustness can be tested. The way that the user interacts with the system is best shown through the following flow diagram:



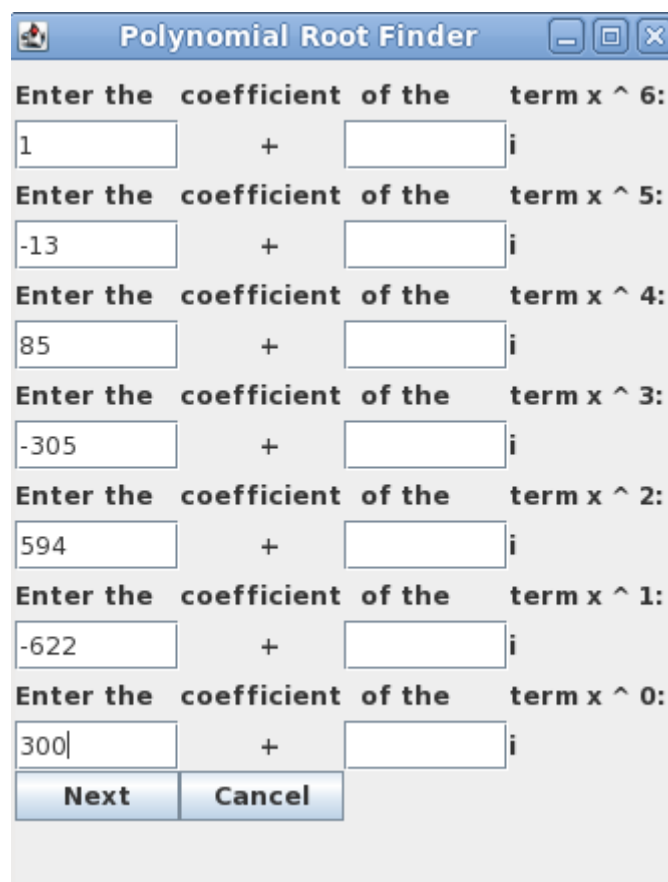
The following screenshots show the interaction between the user and the program:



The screenshot shows a dialog box titled "Polynomial Root Finder". It contains the following elements:

- A text box for "Enter the Degree of the polynomial to be solved" with the value "6".
- A text box for "Enter the maximum acceptable error" with the value "0.00001".
- Two checked checkboxes: "Use Durand-Kerner method" and "Use Aberth method".
- A "Next" button.
- An unchecked checkbox for "Take input from file".
- A "Choose File...." button.
- A "Cancel" button.

Fig 5.1



The screenshot shows the same dialog box, but with the "Next" button disabled and the "Choose File...." button highlighted. The input fields for coefficients are now active, and the following values are entered:

- term x^6 : coefficient 1, imaginary part 0
- term x^5 : coefficient -13, imaginary part 0
- term x^4 : coefficient 85, imaginary part 0
- term x^3 : coefficient -305, imaginary part 0
- term x^2 : coefficient 594, imaginary part 0
- term x^1 : coefficient -622, imaginary part 0
- term x^0 : coefficient 300, imaginary part 0

The "Next" and "Cancel" buttons are visible at the bottom.

Fig 5.2

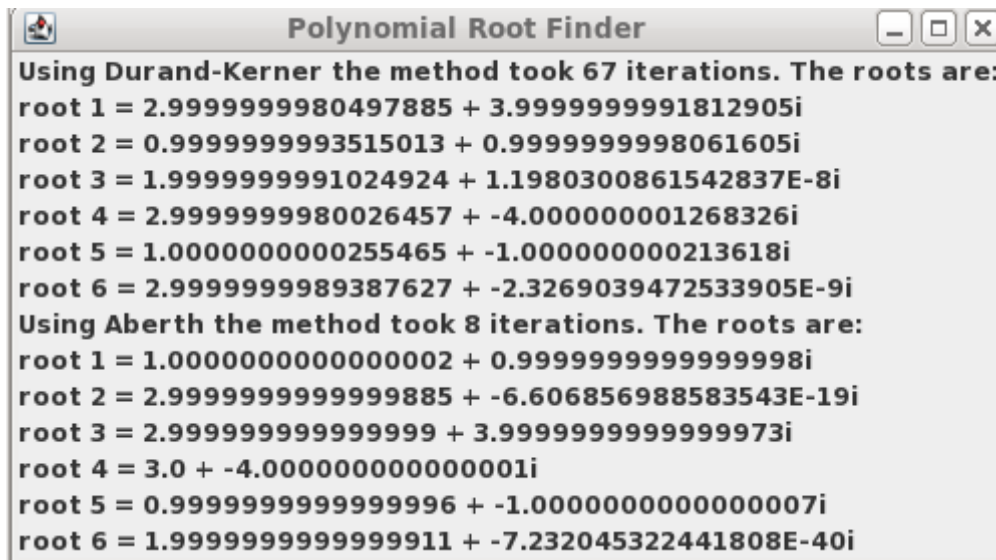


Fig 5.3

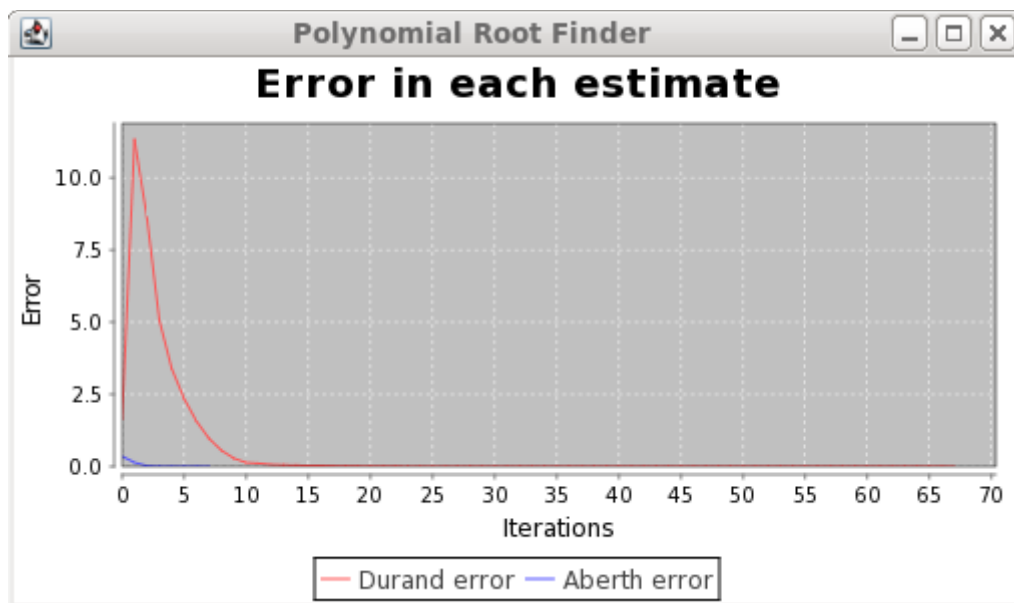


Fig 5.4

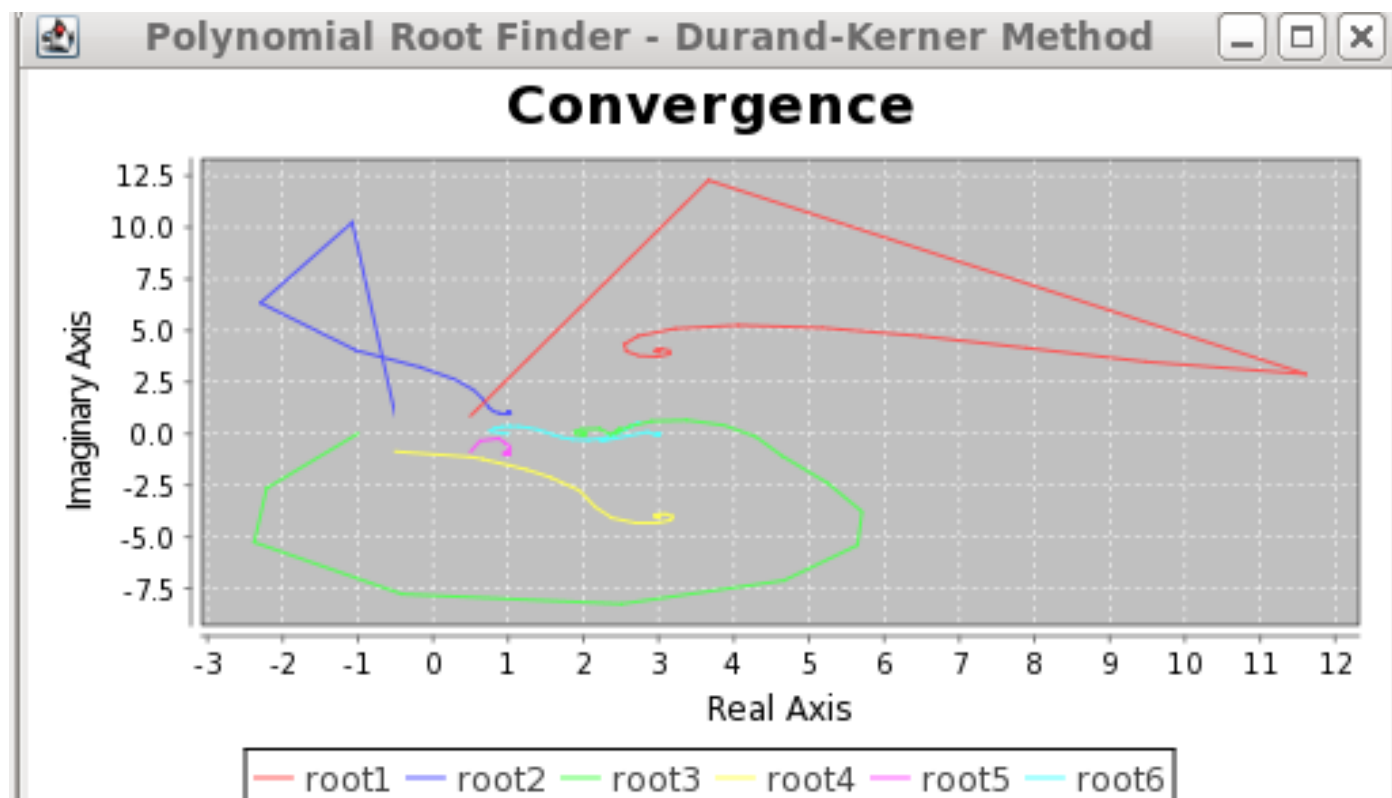


Fig 5.5

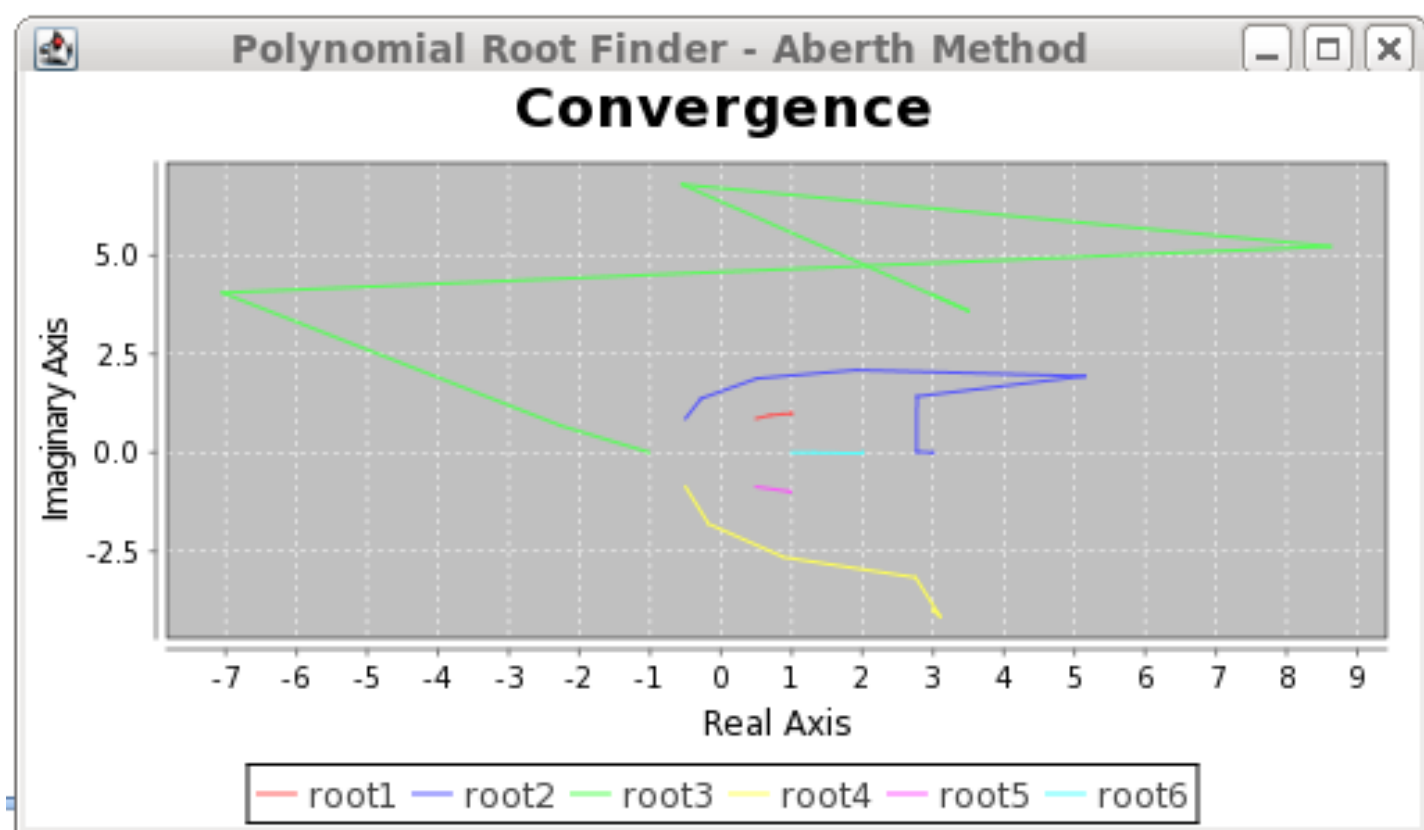


Fig 5.6

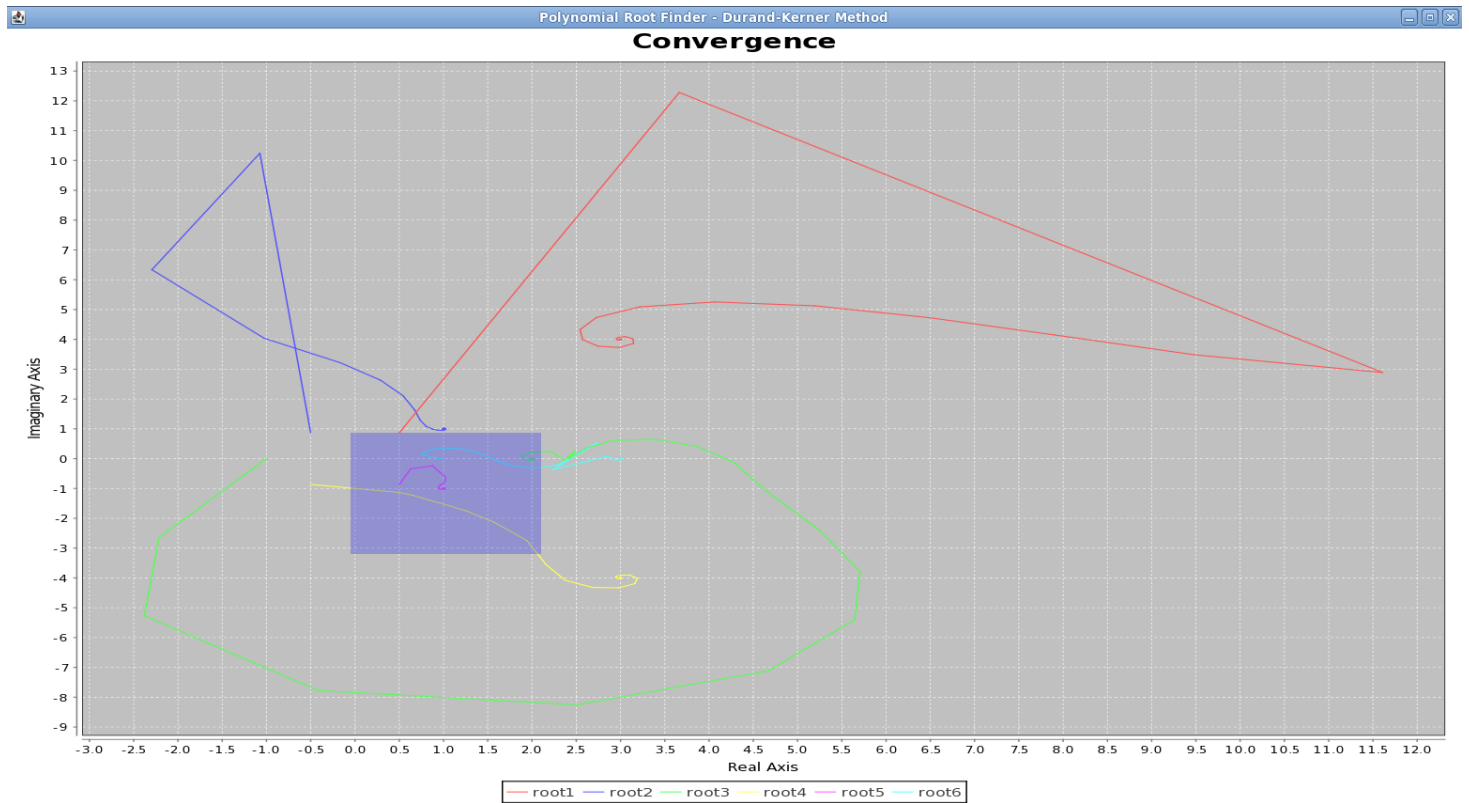


Fig 5.7

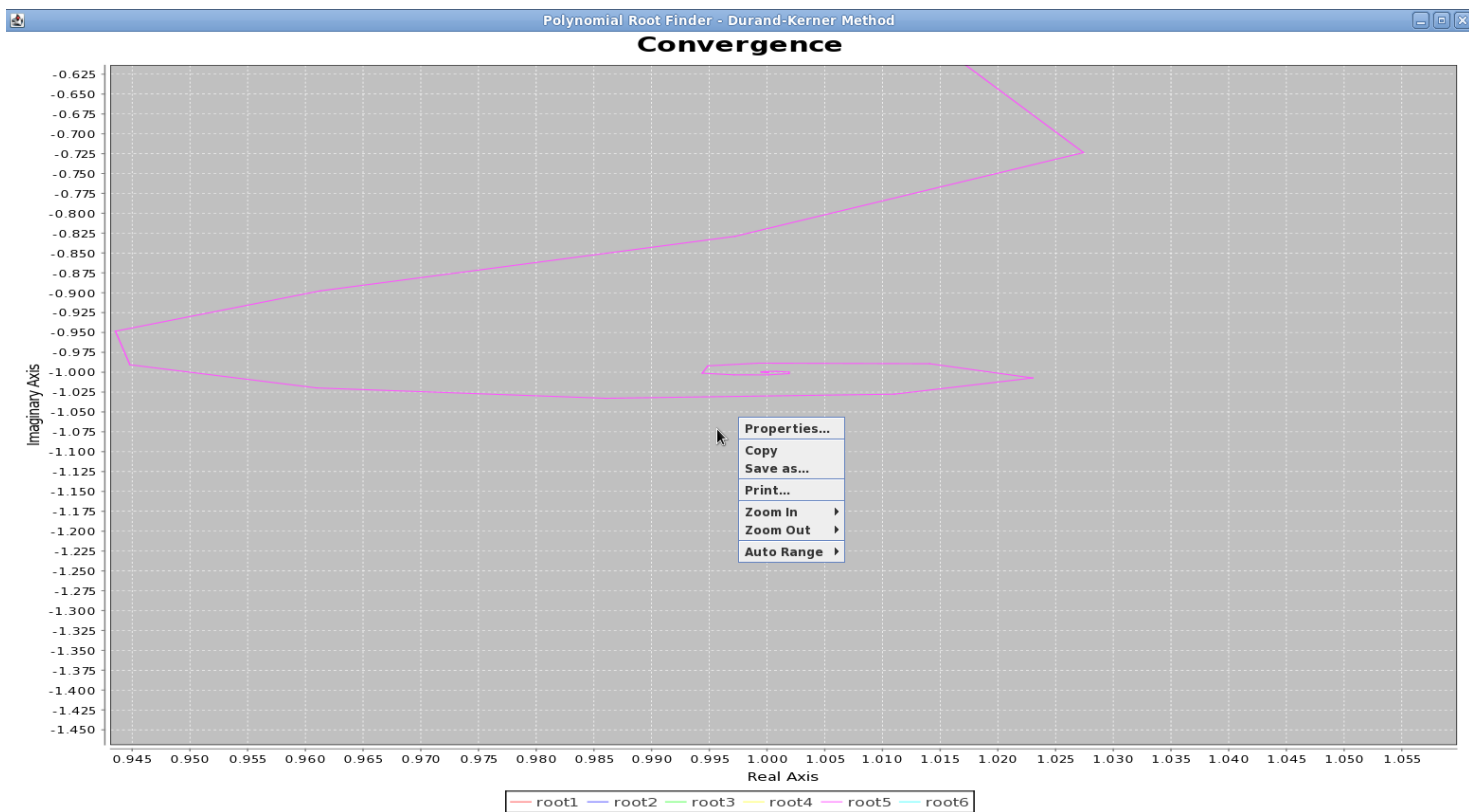


Fig 5.8

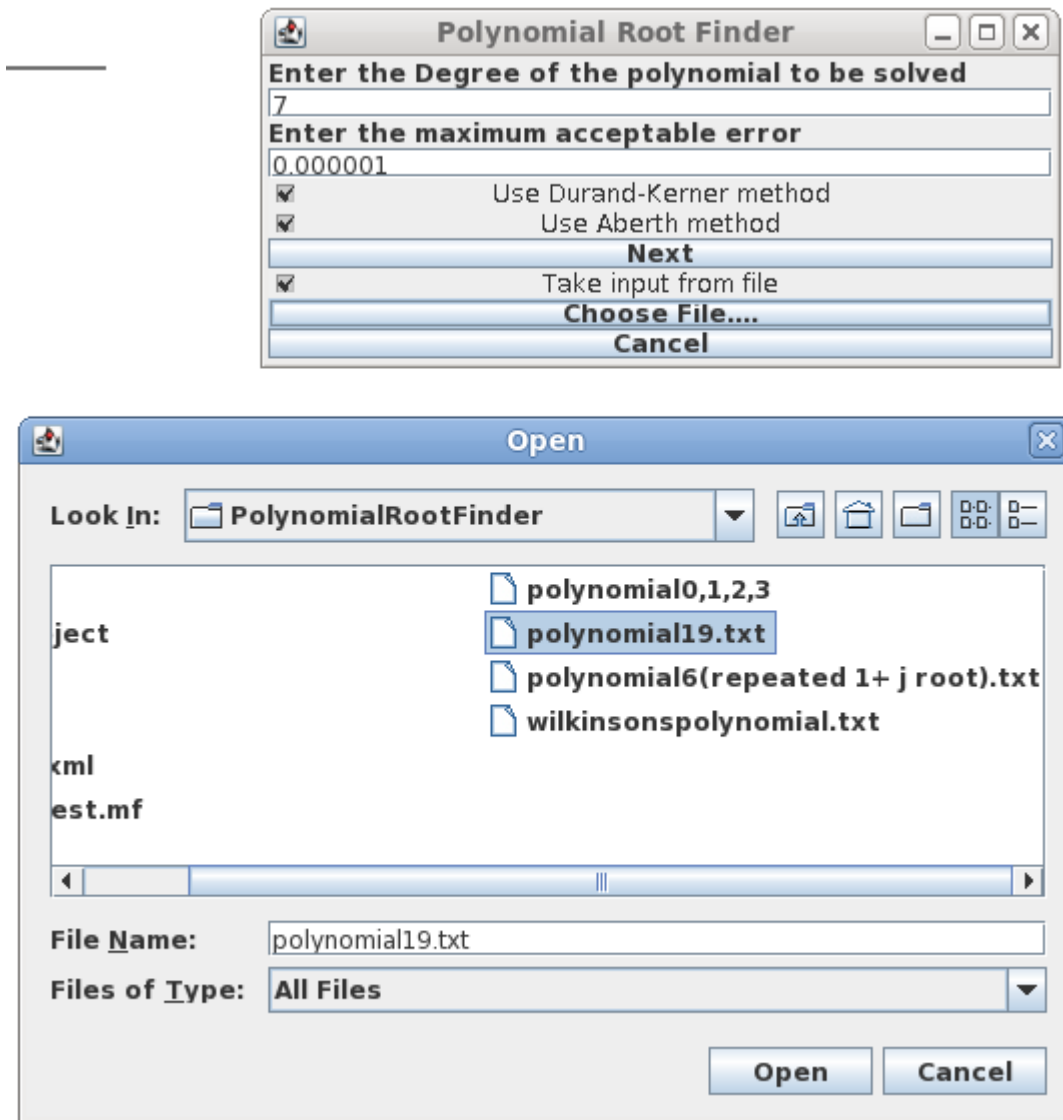


Fig 5.9

Fig 5.1 shows the first window presented to the user when the program is run. It asks the user for the degree of the polynomial to be solved as well as the accuracy to which the approximations must be found. Then the user must select which of the methods is to be used to generate approximations. If the user then clicks “next”, the program will check that the degree given is a positive integer and that the required accuracy is an acceptable number before continuing. If these requirements have been met and the user does not select to input a polynomial from a file, the user is presented with fig 5.2. Fig 5.2 uses the degree given in fig 5.1 to determine how many coefficients to ask the user to supply. Once the user clicks “next”, the program checks that the input from the text fields are numbers and assigns the number zero to any fields that have been left blank. If the input for coefficients is valid, the user is presented with fig 5.3, fig5.4, fig5.5 and fig5.6 simultaneously. Fig 5.3 simply reports the best estimations of the roots according to each method used along with the number of iterations that were necessary to achieve this level of accuracy. Fig 5.4 shows the graph of how the error decreased in a root for both methods. Fig5.5 and fig5.6 are the graphs showing the convergence for each method used. The user can highlight any part of the graph by selecting it with their cursor as shown in Fig 5.7. By right clicking on these graphs, the user can zoom in or out, reset

the axis or save the graph as shown in Fig5.8. Also, if the user hovers the cursor over any point on any of the lines plotted, the graph shows the value of that particular point.

If the user chooses to input from file instead, they will be navigated from fig5.1 to fig 5.9 to select a file to input. Once the file has been selected, the program will check that there is the correct number of coefficients within the file chosen before continuing. Once the user clicks “open”, the coefficients will be checked and the user will be taken to fig5.3, fig5.4, fig5.5 and fig5.6 as before.

5.2 Results

A paper by Henrich and Watkins[7] gives a table of polynomial coefficients along with the corresponding roots according to their Q-D program. There are errors in this paper which were corrected by Richard F. Thomas, Jr[8]. Some of the polynomials used contain multiple roots, some contain roots that are very close together and some polynomials are of a high degree. These test cases seem sensible and should test the robustness of a program which is why they were used to test this polynomial root finding program.

Problem	Degree of polynomial	Coefficients	Expected roots(from Q-D program)
1	3	1, 8, 21, 18	-3, -3, -2
2	3	1, 1.0004, -1.0002, -1.0006	-1, 1.00010, -1.00050
3	4	128, -256, 160, -32, 1	0.96194, 0.69134, 0.30866, 0.03806
4	4	1, -3, -12, 52, -48	2, 2, 3, -4
5	4	1, -8, 24, -32, 16	2, 2, 2, 2
6	6	1, 0, 0, 0, 0, -1	1, -1, $0.5 \pm 0.86603i$, $-0.5 \pm 0.86603i$
7	6	1, -13, 85, -305, 594, -622, 300	2, 3, $3 \pm 4i$, $1 \pm 1i$
8	19	1, 5, 10, 9, 50, 40, 30, 105, 98, 15, 81, 72, 60, 48, 36, 25, 130, 100, 520, 672	-1.0507, -2.9028 \pm 1.8927i, 1.0770 \pm 0.32915i, -0.41890 \pm 1.0476i, 0.50824 \pm 1.6396i, 0.95083 \pm 0.85507i, 0.76888 \pm 1.2373i, -1.1915 \pm 0.41270i, 0.16098 \pm 1.1971i, -0.92737 \pm 0.72873i
9	36	-9265.3, 6468, -42.015, 70.311, 3072.4, 2.9530, 5.6163, 870.73, -7.9141, -74.110, -22.964, 9.2252, -2.4987, -39.063, 6.5810, -6.8461, -7.8867, -32.151, -34.637, 67.916, -390.57, 60.247, 265.74, -453.86, -7015.6, -309.67, -2.0574, -85.581, -99.394, -20.775, 49.225, 3924.5, -0.083830, 73.941, 0.049060, 88.312, -993.56	0.81276 \pm 0.061480i, 0.58270 \pm 0.48197i, 0.22083 \pm 0.7i, -0.88460 \pm 0.30152i, 0.63308 \pm 0.69314i, -0.20948 \pm 0.89996i, 1.0530 \pm 0.087703i, -0.83729 \pm 0.39630i, -0.072582 \pm 0.99088i, 0.57888 \pm 0.77201i, -0.583111 \pm 0.78522i, 0.15853 \pm 1.0003i, 0.82646 \pm 0.57735i, -0.98285 \pm 0.11248i, 0.39262 \pm 0.92500i, 0.95385 \pm 0.36071i, -0.74665 \pm 0.60412i, -0.37723 \pm 0.89807i

Table 5.1: Polynomial Coefficients and Q-D Program Results

Problem	Roots (from Durand-Kerner)	Number of iterations
1	-2.99942+0.00082i, -3.00038+0.00082i, -2	50
2	-0.998554+0.00011i, -1.00174-0.00037i, 1	31
3	0.0381+0.00002i, 0.6914+0.00005i, 0.30859+0.00008i, 0.96194	30
4	1.99984+0.00004i, -4, 2.00015, 3	52
5	2.0383+0.03544i, 2.03635-0.03708i, 1.96495-0.03565i, 1.96429+0.03159i	39
6	1, -1, 0.5+0.86603i, 0.5-0.86603i, -0.5+0.86603i, -0.5+0.86603i	1
7	3+4i, 1+i, 2, 3-4i, 1-i, 3	67
8	0.95083-0.85507i, 0.95083+0.85507i, 0.50824-1.63955i, 0.50824+1.63955i, 0.16098+1.19712i, 0.16098-1.19712i, -0.4189+1.0476i, -0.4189-1.0476i, -0.92737+0.72873i, -0.92737-0.72873i, -1.19151+0.4127i, -1.19151-0.4127i, -1.05065, 0.76888-1.23729i, 0.76888+1.23729i, -2.90279-1.89268i, -2.90279+1.89268i, 1.07697-0.32915i, 1.07697+0.32915i	113
9	0.81276+0.06148i, 0.81276-0.06148i, 0.95385+0.36071i, 0.95385-0.36071i, 0.82646+0.57735i, 0.82646-0.57735i, 0.6331+0.69314i, 0.6331-0.69314i, 0.57888+0.77201i, 0.57888-0.77201i, 0.22083+0.7i, 0.22083-0.7i, 0.39262+0.925i, 0.39262-0.925i, 0.15853+1.00033i, 0.15853-1.00033i, -0.07258+0.99088i, -0.07258-0.99088i, -0.20948+0.89996i, 0.20948-0.89996i, -0.37723+0.89807i, -0.37723-0.89807i, -0.5872+0.48197i, 0.5872-0.48197i, -0.58311+0.78522i, -0.58311-0.78522i, -0.74665+0.60412i, 0.74665-0.60412i, -0.83729+0.3963i, -0.83729-0.3963i, -0.8846+0.30152i, -0.8846-0.30152i, -0.98285+0.11248i, -0.98285-0.11248i, 1.05301+0.0877i, 1.05301-0.0877i	40

Table 5.2: Results using the Durand-Kerner method

Problem Number	Roots (from Aberth method)	Iterations
1	-3,-3,-2	9
2	-1.00025+0.00168i, -1.00025+0.00082i, 1	5
3	0.6913, 0.03806, 0.30866, 0.96194	5
4	2,-4,3,2	9
5	1.99233+0.0303i, 2.02637+0.00648i, 2.00532-0.02233i, 1.98127-0.00469i	8
6	1, -1, 0.5+0.86667i, 0.5-0.86667i, -0.5+0.86667i, -0.5+0.86667i	1
7	1+i,3,3+4i,3-4i,1-i,2	8
8	1.07697-0.32915i, 1.07697+0.32915i, 0.95083-0.85507i, 0.95083+0.85507i, 0.76888-1.23729i, 0.76888+1.23729i, 0.16098+1.19712i, 0.16098-1.19712i, 0.50824-1.63955i, 0.50824+1.63955i, -0.4189+1.0476i, -0.4189-1.0476i, -0.92737+0.72873i, -0.92737-0.72873i, -1.19151+0.4127i, -1.19151-0.4127i, -2.90279-1.89268i, -2.90279+1.89268i, -1.05065	9
9	0.81276+0.06148i, 0.81276-0.06148i, 0.95385+0.36071i, 0.95385-0.36071i, 0.82646+0.57735i, 0.82646-0.57735i, 0.6331+0.69314i, 0.6331-0.69314i, 0.57888+0.77201i, 0.57888-0.77201i, 0.22083+0.7i, 0.22083-0.7i, 0.39262+0.925i, 0.39262-0.925i, 0.15853+1.00033i, 0.15853-1.00033i, -0.07258+0.99088i, -0.07258-0.99088i, -0.20948+0.89996i, 0.20948-0.89996i, -0.37723+0.89807i, -0.37723-0.89807i, -0.5872+0.48197i, 0.5872-0.48197i, -0.58311+0.78522i, -0.58311-0.78522i, -0.74665+0.60412i, 0.74665-0.60412i, -0.83729+0.3963i, -0.83729-0.3963i, -0.8846+0.30152i, -0.8846-0.30152i, -0.98285+0.11248i, -0.98285-0.11248i, 1.05301+0.0877i, 1.05301-0.0877i	7

Table 5.3: Results using the Aberth method

Table 5.1 shows the polynomials that were used to test this polynomial root finding program. Table 5.2 shows how the Durand-Kerner method within the program performed on the test case polynomials. The program was asked to find the roots of the polynomials with a maximum error of 0.0001 as the test cases did not have any roots that were closer than 0.0001 together. The method was successful for all cases except for problems two and four. Problem two had two roots that were extremely close together and from the results in the table it was not possible to tell if the method was converging towards the same root twice. Problem four was where the same root was repeated four times. In this case, the method was not a total failure as it was converging towards the roots but failed to achieve the required accuracy.

Table 5.3 shows how the Aberth method performed on all of the test case polynomials. Many of the test cases were similar to the Durand-Kerner method except they required less iterations and generally returned more accurate approximation to roots than the Durand-Kerner method. Like the Durand-Kerner method, the implementation struggled on problem number two where there were two roots very close to each other and on problem four where the root was a four-fold root.

5.2.1 The Wilkinson Polynomial

Another way to test the program is to use the idea put forward by James Wilkinson in 1963[9]. He created a polynomial of degree 20 whose roots are all the integer values between one and 20. This is commonly known as Wilkinson's polynomial and can be expressed in the following way:

$$p(x) = \prod_{i=1}^{20}(x - i) = (x - 1)(x - 2)(x - 3) \dots (x - 20)$$

This polynomial is very ill conditioned. It is easy to see this when the coefficients are evaluated in the following way:

$$p(x) = x^{20} - 120x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} \\ + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} \\ - 135585182899530x^{11} + 1307535010540395x^{10} - 10142299865511450x^9 \\ + 63030812099294896x^8 - 311333643161390640x^7 \\ + 1206647803780373360x^6 - 3599979517947607200x^5 \\ + 8037811822645051776x^4 - 12870931245150988800x^3 \\ + 13803759753640704000x^2 - 8752948036761600000x \\ + 2432902008176640000$$

A slight change to the value of the coefficient of the term x^{19} from -210 to -210.0000001192 the twenty roots become (to 5 decimal places):

1.00000, 2.00000, 3.00000, 4.00000, 5.00000, 6.00001, 6.99970, 8.00727, 8.91725, 20.84691, 10.09527±0.64350i, 11.79363±1.65233i, 13.99236±2.51883i, 16.73074±2.81262i, 19.50244±1.94033i

These roots have changed significantly from their initial integer values. Some of the roots have now become complex conjugates. This example shows that a slight loss in accuracy can cause the wrong roots to be found. This makes a good test case for the program.

Roots from the Aberth method
2.000000000000157
4.000000004685773
4.9999999148118786
6.999998344010249
9.0000846783288-8.735E-321i
10.999635801164697+1.6005674E-318i
12.998422230447803+1.6005674E-316i
14.997566010848045+3.57752667E-316i
16.998458267092342+2.764839E-317i
18.9996760922222+1.416811E-318i
20.00001011407452+8.740693E-318i
17.999473596445764+9.5276815E-317i
16.003818729928362+2.14445355E-316i
14.007726804924788+3.5454872E-316i
12.004508028214142-4.9042196E-317i
9.999761915492812+6.5418E-319i
8.000024347728386-4.38E-321i
6.000001018688912
2.999999999375127
1

The Aberth method was used to try and solve this polynomial and the program ran until it reached the maximum number of iterations which was set at 200. The program proved to be accurate as it found all 20 roots to at least two decimal places. The Aberth method is the more efficient of the two methods implemented so it is expected that it performs better than the Durand-Kerner method for the Wilkinson polynomial. The Aberth method did return more accurate approximations to roots and was in the process of converging towards the correct roots which shows the program is capable of taking input of very ill-conditioned problems.

Chapter 6 – Conclusions

Overall the project has been a success; the methods that were proposed have been implemented and operate efficiently. The efficiency of the program can be considered to be better than what was expected because it performed well on most test cases. On the test cases where the program did not achieve the desired accuracy, it looked as though it would have been able to if the maximum iterations allowed was increased.

6.1 Evaluation of Proposed Features

The best way to evaluate if the program is successful is by comparing it to goals that were proposed at the start of the project. The first of the objectives was for the program to be able to handle polynomials with complex coefficients. The program is able to do this as it allows the user to input coefficients of a polynomial as complex numbers. Also, it can manipulate and operate on complex numbers through its complex number class before presenting the results in complex number form.

Another of the objectives was to implement a graphical user interface which allowed the user to input a polynomial. This was achieved; the graphical user interface is reliable and deals well with user input. The interface is aesthetically pleasing and easy for a user to understand. Through the graphical user interface, the next couple of objectives are achieved. By implementing the classes included in java, it was possible to load an input from a file which was one of the objectives. Also, when the user inputs anything into the program, the validity of the input data is checked. If the input of the user is invalid or does not make sense, the program asks the user to correct their input. This is useful because if these checks were not in place, the program could take invalid input which would lead to unpredictable results or the program crashing.

One of the main objectives was to show how the estimations of roots converge by showing it on a graph. This has been achieved and allowed the achievement of another objective through it. On the graph, the set of results from each iteration is plotted. The user is able to hover their cursor over any plotted point and the graph tells the user what that point is. This is a more efficient way of achieving the objective to present all of the estimations to the user. The user is also able to zoom in and out on the graph as well as save a version of the graph. This is more than what was planned but is useful when investigating the final stages of convergence.

Another aim was to inform the user of how many iterations were needed to achieve the final result. This has been done and the user is informed of this when the best estimations of roots are reported.

The final goal was to show how the error decreased in the estimations of the roots as the iterations continued. A graph is plotted by the program to show this. The program picks the first solved root of the given polynomial and uses the final estimation of the root to calculate the previous errors in estimation. A curve is then plotted for each method used that shows how the error decreased over the number of iterations which is useful as it clearly shows how each method performed for a random root.

6.2 Evaluation of proposed plan

None of the set objectives had to be altered or adjusted to become achievable; they were all completed without any real problems. The project did differ from the plan in terms of time spent at

each stage. In the project plan, more time was allocated to research and planning than was actually needed. This was because the plan anticipated the development process of planning how to write the program and then writing the program when in reality, the development process was more about getting a basic root finding algorithm implemented correctly before adding features and making the program code more intricate and complex. This more practical development process meant that research and planning was done alongside the implementation because sometimes it was necessary to research how a feature could be added before implementing it and then researching the next stage of development.

6.3 Areas For Further Development

Despite the program being operational and reliable, there are still ways that it could be developed further to enhance the user's experience. One way the program could be developed further would be to add a higher order method that converges to roots even faster than the Aberth method. This could make the program more efficient although a small amount of research suggests that it would be hard to find a higher order method that is as reliable as the Aberth method. Another area for development is create a java web applet for the program. Having the program available on the internet would increase the amount of people who have access to the program. The types used in the program could be changed in order to make sure that the handling of long numbers is more efficient. Using the type "long double" would provide more precision within the program. Finally, an area for development would be how the program handles multiple roots. The research by Hull and Mathon[6] suggests that if the program could detect multiple roots at an early stage, the mean of the two approximations would give a better estimate to a root and would improve the efficiency of the worst case of both the Durand-Kerber and the Aberth method.

Bibliography

- [1] E. Durand. Equations du type $f(x)=0$: Racines d'un polynome, 1960.
- [2] I. Kerner. Ein gesamtschrittverfahren zur berechnung der nullstellen von polynomen, 1966.
- [3] K. Weierstrass. Neuer beweis des satzes, dass jede ganze rationale funktion einer veränderlichen dargestellt werden kann als ein produkt aus linearen funktionen derselben veränderlichen, 1903.
- [4] Oliver Aberth. Iteration methods for finding all zeros of a polynomial simultaneously, 1973.
- [5] Miodrag S. Petković, Carsten Carstensen, Miroslav Trajković. Weierstrass formula and zero-finding methods, 1993
- [6] T.E. Hull and R. Mathon. The mathematical basis for a new polynomial root finder with quadratic convergence, 1994
- [7] P. Henrich and Bruce O. Watkins. Finding zeros of a polynomial by the q-d algorithm, 196
- [8] Jr. Richard F. Thomas and Peter Henrich. Corrections to numerical data on q-d algorithm, 1966.
- [9] J.H Wilkinson Rounding Errors in Algebraic Processes, 1963
- [10] "dido". Wilkinson's polynomial.
<http://www.everything2.com/e2node/Wilkinson's%20polynomial>, 2004, accessed on 3rd May 2011
- [11] D. Bini. Numerical computation of polynomial zeros by means of Aberth's method, 1996
- [12] J. Verschelde. The method of Weierstrass (also known as the Durand-Kerner method), 2003
- [13] M.S.Petkovic, C.Carstensen and M.Trajkovic. Weierstrass formula and zero-finding methods, 1995
- [14] Kripasgar. Efficient Micro Mathematics. Circuit Cellar p.62, issue 212. March 2008.
- [15] R.Kress. Numerical Analysis p.112, 1991