Project 1a

Yadu Bhageria

Imperial College London

Table of Contents

[Abstract 4](#_Toc442841788)

[Project 1a 5](#_Toc442841789)

[Introduction 5](#_Toc442841790)

[Part 1 6](#_Toc442841791)

[Example Outputs 6](#_Toc442841792)

[(-3) Contradictory Equation 6](#_Toc442841793)

[(-2) Tautological Equation 6](#_Toc442841794)

[(-1) Linear Equation 6](#_Toc442841795)

[(0) Complex Roots 6](#_Toc442841796)

[(1) Repeated Root 6](#_Toc442841797)

[(2) Two Real Roots 6](#_Toc442841798)

[Part 2 7](#_Toc442841799)

[Part 4 8](#_Toc442841800)

[<float.h> 8](#_Toc442841801)

[File Paths 8](#_Toc442841802)

[MLC Computers 8](#_Toc442841803)

[Linux Virtual Machine 9](#_Toc442841804)

[Values in float.h 10](#_Toc442841805)

[GCC 10](#_Toc442841806)

[Intel 10](#_Toc442841807)

[Microsoft Visual Studio 11](#_Toc442841808)

[Raspberry Pi 11](#_Toc442841809)

[float128 type 11](#_Toc442841810)

[Dealing with Accuracy Loss 12](#_Toc442841811)

[Mastery Section 14](#_Toc442841812)

[Appendix 17](#_Toc442841813)

[Representative C Code and Files 17](#_Toc442841814)

[Part 1 17](#_Toc442841815)

[prog\_1.c 17](#_Toc442841816)

[Part 2 18](#_Toc442841817)

[prog\_2.c 19](#_Toc442841818)

[Part 4 22](#_Toc442841819)

[prog\_4.c 22](#_Toc442841820)

[Mastery Section 22](#_Toc442841821)

[Matlab Code for producing Plot 22](#_Toc442841822)

[prog\_m2.c: 22](#_Toc442841823)

[prog\_m3.c: 24](#_Toc442841824)

Abstract

This project deals with creating a quadratic equation solver in the programming language C. Part 1, 2, and 4 involve quadratic equations that have real coefficients. Here attempts have been made to deal with situations that can involve complications in the computation of the roots. First of which is when when we have to ‘add’ numbers of differing signs. This leads to huge cancellations. The second of which is when we have very large or very small coefficient values with respect to the other coefficients. This leads to float overflows during the calculations. The mastery section allows for quadratic equations with complex coefficients and produces a table of results for a specific set case along with a plot of the outputted values on the complex plane.

Keywords: Quadratic Solver, Real Coefficients, Complex Coefficients, Float Overflow

Project 1a

Introduction The form of our quadratic equation is: . For all parts but the mastery . In the mastery section .

The quad\_roots functions input these coefficients, compute plus store the resulting roots for the equation (if there are any) and output the number of real roots. The function needs to identity the various cases that arise to deal with them efficiently. With real coefficients there are 6 cases to deal with:

1. The resulting equations is a contradiction and there are no roots. (-3)
2. The resulting equation is a tautology and every x is a root. (-2)
3. The equation is linear and there is one root. (-1)
4. The equation has 1 real root. (1)
5. The equation has 2 real roots. (2)
6. And finally the equation has 2 complex conjugate roots. (0)

As mentioned in the lecture notes/exercise sheet we follow convention to return a negative integer value for cases when the equation is not a quadratic equation and zero for when the roots are not real. The main program can then output the appropriate values. A similar approach is made for the complex case in the mastery section.

There is a bibliography at the end with appropriate references throughout the body of the project. All code is also in the appendix at the end.

Part 1

Here we amend a previously written linear root function to also become capable of dealing with quadratic equations. The function reads in values of and outputs to the standard output based on the number of real roots. It utilizes the lin\_root function to deal with the cases of and . It does not deal with any nuances that are analyzed later.

# Example Outputs

(-3) Contradictory Equation.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 0 0 2

The values of a2, a1, a0 resulted in a contradictory equation

(-2) Tautological Equation.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 0 0 0

a2=a1=a0=0 so any real number is a root as the equation is tautological

(-1) Linear Equation.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 0 1 2

a2=0 so we are dealing with a linear equation with one root

r1 = -2

(0) Complex Roots.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 2 0 1

The roots are complex.

r1 = -0 + 0.7071067812i, r2 = -0 - 0.7071067812i

(1) Repeated Root.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 1 -4 4

There are repeated real roots.

r1 = r2 = 2

(2) Two Real Roots.

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 1 5 1

There are two real roots.

r1 = -0.2087121525, r2 = -4.791287847

Part 2

In part two we deal with the first of two complications mentioned in the abstract. That is, we deal with the ‘addition’ of two numbers with differing signs when . We do this by noticing (as told in the exercise sheet) that the product of the two roots is . This means that if we know then we can add the positive discriminant term, or else similarly subtract it to get one root, . To get we simply divide by and realize that in the process we do not have to add numbers of differing signs.

The radii for Pluto is 1187km[[1]](#footnote-1) and the radii for Charon is 606km[[2]](#footnote-2). Since we know what our input will be we can use a data file (see appendix) to input the values into the program in a loop. The file contains the values corresponding to for Pluto and then Charon using the given equation where R is the radius of the body in question and d is how far in the distance we want to see.

Table : Distance results for Pluto

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Traditional Method with Floats | Accurate Method with Floats | Traditional Method with Doubles | Accurate Method with Floats |
| 1m | 0 | 4.212299984e-07 | 4.211906344e-07 | 4.212299916e-07 |
| 100m | 0 | 0.004212299827 | 0.004212299827 | 0.004212299908 |
| 1km | 0.375 | 0.4212298989 | 0.4212299169 | 0.4212299168 |
| 100km | 4204.875 | 4204.852051 | 4204.85224 | 4204.85224 |

Table : Distance results for Charon

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Traditional Method with Floats | Accurate Method with Floats | Traditional Method with Doubles | Accurate Method with Floats |
| 1m | 0 | 8.25082509e-07 | 8.250353858e-07 | 8.250825083e-07 |
| 100m | 0 | 0.008250825107 | 0.008250824991 | 0.008250825026 |
| 1km | 0.8125 | 0.8250820041 | 0.8250819466 | 0.8250819466 |
| 100km | 8195.375 | 8195.40918 | 8195.408645 | 8195.408645 |

When d is small the use of the traditional formula is less accurate than the new method. Significantly so in the case of floats as can be seen by the 0 values for d=1m and d=100m. Even in the case of doubles, the values only start to diverge after about 5 significant figures for small d.

For large d we notice that doubles are significantly more accurate than floats, for the traditional and accurate method. Furthermore, the traditional method gives the exact same result for large d in the case of doubles but not in the case of floats.

Part 4

First we analyse the inbuilt information about the types double and float.

# <float.h>

## File Paths

### MLC Computers

Below is a table with the paths for the float.h files found on the MLC computers along with the compiler type they belong to.

Table : <float.h> files found on the MLC computers[[3]](#footnote-3)

|  |  |  |
| --- | --- | --- |
| No. | Compiler Type | Path |
| The 18 float.h files below are for C compilers | | |
| 1 | GCC (5.2.0) | C:\cygwin64\lib\gcc\x86\_64-pc-cygwin\5.2.0\include |
| 2 |  | C:\matlab\R2015a\polyspace\verifier\cxx\include\include-libc |
| 3 |  | C:\matlab\R2015a\extern\include\physmod\common\foundation\core\util\flt |
| 4 | LCC | C:\matlab\R2015a\sys\lcc\include |
| 5 | LCC | C:\matlab\R2015a\sys\lcc64\lcc64\include64 |
| 6 | TCC | C:\matlab\R2015a\sys\tcc\win64\include |
| 7 | Clang | C:\matlab\R2015a\sys\extern\win64\clang\lib\clang\3.2\include |
| 8 | GCC (4.7.2) | C:\mingw\include |
| 9 | GCC (4.7.2) | C:\mingw\lib\gcc\mingw32\4.7.2\include |
| 10 | GCC (4.6.2) | C:\octave3.6.1\_gcc4.6.2\include\cln |
| 11 | GCC (4.6.2) | C:\octave3.6.1\_gcc4.6.2\mingw32\include |
| 12 | GCC (4.6.2) | C:\octave3.6.1\_gcc4.6.2\mingw32\lib\gcc\mingw32\4.6.2\include |
| 13 | Intel | C:\Program Files (x86)\Intel\Composer XE 2013 SP1\compiler\include |
| 14 | Intel | C:\Program Files (x86)\Intel\Composer XE\compiler\include |
| 15 | Microsoft Visual Studio Compiler 2012 | C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\crt\src |
| 16 | Microsoft Visual Studio Compiler 2012 | C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\include |
| 17 | Microsoft Visual Studio Compiler 2013 | C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\crt\src |
| 18 | Microsoft Visual Studio Compiler 2013 | C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include |
| The 3 float.h files below are for C++ compilers | | |
| 19 | GCC (5.2.0) | C:\cygwin64\lib\gcc\x86\_64-pc-cygwin\5.2.0\include\c++\tr1 |
| 20 | GCC (4.6.2) | C:\octave3.6.1\_gcc4.6.2\mingw32\lib\gcc\mingw32\4.6.2\include\c++\tr1 |
| 21 | GCC (4.7.2) | C:\mingw\lib\gcc\mingw32\4.7.2\include\c++\tr1 |

There are 21 copies of foat.h on the MLC computers. 18 for C compilers and 3 for C++ compilers.

From the table we can also see that there are 6 types of compilers:

1. GCC – in three versions: 5.2.0, 4.7.2, and 4.6.2
2. TCC for MATLAB
3. LCC for MATALB
4. Intel
5. Clang
6. Microsoft Visual Studio – in two versions: 2012 and 2013

### Linux Virtual Machine

I found two file paths for the Linux virtual machine which are for the GCC (4.8.2) compiler with one for C and the other for C++

1. /usr/lib/gcc/x86\_64-redhat-linux/4.8.2/include
2. /usr/include/c++/4.8.2/tr1

## Values in float.h

### GCC

We find these using the command “echo | gcc -dM -E - | grep DBL” in the command line as GCC uses predefined values that come with the compiler rather than storing them in the float files. Note they are the same for all the versions listed.

Table : GCC Names and Values of Numbers

|  |  |
| --- | --- |
| Defined As | Value |
| \_\_FLT\_EPSILON\_\_ | 1.19209289550781250000e-7F |
| \_\_FLT\_MIN\_\_ | 1.17549435082228750797e-38F |
| \_\_FLT\_MAX\_\_ | 3.40282346638528859812e+38F |
| \_\_DBL\_EPSILON\_\_ | ((double)2.22044604925031308085e-16L) |
| \_\_DBL\_MIN\_\_ | ((double)2.22507385850720138309e-308L) |
| \_\_DBL\_MAX\_\_ | ((double)1.79769313486231570815e+308L) |
| \_\_LDBL\_EPSILON\_\_ | 1.08420217248550443401e-19L |
| \_\_LDBL\_MIN\_\_ | 3.36210314311209350626e-4932L |
| \_\_LDBL\_MAX\_\_ | 1.18973149535723176502e+4932L |

### Intel

Table : Intel Names and Values of Numbers

|  |  |
| --- | --- |
| Defined As | Value |
| FLT\_EPSILON | 1.19209290e-07F |
| FLT\_MAX | 3.40282347e+38F |
| FLT\_MIN | 1.17549435e-38F |
| DBL\_EPSILON | 2.2204460492503131e-16 |
| DBL\_MAX | 1.7976931348623157e+308 |
| DBL\_MIN | 2.2250738585072014e-308 |

#if (\_\_IMFLONGDOUBLE == 64) || defined(\_\_LONGDOUBLE\_AS\_DOUBLE)

   #define LDBL\_EPSILON        2.2204460492503131e-16L

   #define LDBL\_MAX            1.7976931348623157e+308L

   #define LDBL\_MIN            2.2250738585072014e-308L

#else

   #define LDBL\_EPSILON        1.0842021724855044340075E-19L

   #define LDBL\_MAX            1.1897314953572317650213E+4932L

   #define LDBL\_MIN            3.3621031431120935062627E-4932L

#endif

### Microsoft Visual Studio

Table : Microsoft Visual Studio Names and Values of Numbers

|  |  |
| --- | --- |
| Defined As | Value |
| FLT\_EPSILON | 1.192092896e-07F |
| FLT\_MIN | 1.175494351e-38F |
| FLT\_MAX | 3.402823466e+38F |
| DBL\_EPSILON | 2.2204460492503131e-016 |
| DBL\_MIN | 2.2250738585072014e-308 |
| DBL\_MAX | 1.7976931348623158e+308 |
| LDBL\_EPSILON | DBL\_EPSILON |
| LDBL\_MIN | DBL\_MIN |
| LDBL\_MAX | DBL\_MAX |

### Raspberry Pi

Raspberry Pi A

Chip: ARM BCM2835

Table : Raspberry Pi Names and Values of Numbers for GCC

|  |  |
| --- | --- |
| Defined As | Value |
| FLT\_MAX | 3.4028234663852886e+38 |
| FLT\_MIN | 1.1754943508222875e-38 |
| FLT\_EPSILON | 1.1920928955078125e-7 |
| DBL\_MAX | 1.7976931348623157e+308 |
| DBL\_MIN | 2.2250738585072014e-308 |
| DBL\_EPSILON | 2.2204460492503131e-16 |

### float128 type

This is done for the GCC compiler.

Table : Names and Values of 128bit floating point numbers

|  |  |
| --- | --- |
| Defined As | Value |
| FLT128\_MAX | 1.18973149535723176508575932662800702e4932Q |
| FLT128\_MIN | 3.36210314311209350626267781732175260e-4932Q |
| FLT128\_EPSILON | 1.92592994438723585305597794258492732e-34Q |

The values are not identical. For doubles they vary in how many decimal places they are defined to but otherwise are the same. For floats they vary in how many decimal places they are defined to and also differ in their last 2 degrees of accuracy. Furthermore, DBL\_MAX seems to be increased for Microsoft but rounded down correctly for Intel (in comparison to GCC).

Long doubles are defined for Intel but only if the if statement holds. Microsoft Visual Studio simply gives the values of double for long double. GCC explicitly defines these values but to fewer decimal places than Intel.

# Dealing with Accuracy Loss

In this part we further improve our quadratic solver to deal with the cases where the calculation of the roots leads to float overflow, that is we produce a number that is too large to fit in a standard double precision variable.

In order to do this I relied heavily on the methods explain in a document titled “Scilab is not Naïve” [[4]](#footnote-4).

We start off by noting that normally . We can then define

and . We are interested in computing s (i.e. sqrt(s^2)) and want to deal with the cases where are large which would result in us having to multiply two possibly large numbers together that could result in a number too large to fit in a float.

To do this we deal with two separate scenarios:

Here we can define

which does not ever compute .

For

Here we continue to implement the accurate method we obtained in Part 2 where we don't add numbers of differing signs.

This results in greatly increased effectivity.

For the old program (prog\_1.c):

Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 1 1e200 -1

There are two real roots.

r1 = inf, r2 = -inf

For the new program (prog\_4.c):

enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0

in the order a2,a1,a0, separated by spaces: 1 1e200 -1

There are two real roots.

r1 = 1e+200, r2 = 0

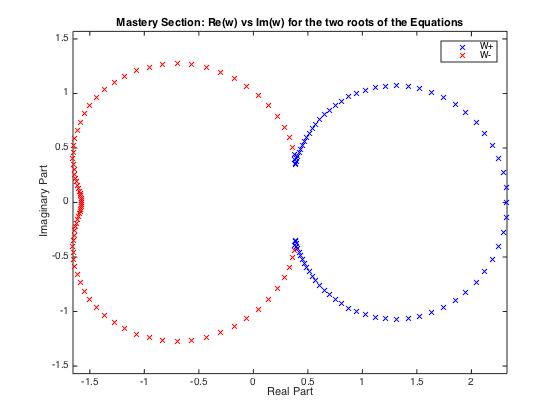
Clearly the new program deals much better with overflow of numbers than the old program.

Mastery Section

Here we deal with the case where .

Table : Results of solving the given quadratic equation for j=0...80

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| j | Re(w+) | Im(w+) | Re(w-) | Im(w-) |
| 0 | 0.375 | 0.438986 | 0.375 | -0.438986 |
| 1 | 0.380595 | -0.390489 | 0.360172 | 0.507815 |
| 2 | 0.381975 | -0.361835 | 0.331317 | 0.593597 |
| 3 | 0.382646 | -0.349785 | 0.285609 | 0.690277 |
| 4 | 0.384344 | -0.350429 | 0.222419 | 0.791268 |
| 5 | 0.387718 | -0.360434 | 0.142613 | 0.890764 |
| 6 | 0.392953 | -0.377339 | 0.047886 | 0.984101 |
| 7 | 0.400101 | -0.399433 | -0.059608 | 1.067688 |
| 8 | 0.409207 | -0.42555 | -0.177444 | 1.138842 |
| 9 | 0.420349 | -0.454893 | -0.303023 | 1.19566 |
| 10 | 0.433655 | -0.486914 | -0.433655 | 1.236914 |
| 11 | 0.449303 | -0.521221 | -0.566629 | 1.261987 |
| 12 | 0.467519 | -0.557523 | -0.699282 | 1.270816 |
| 13 | 0.488582 | -0.595587 | -0.829074 | 1.263842 |
| 14 | 0.512821 | -0.6352 | -0.95366 | 1.241962 |
| 15 | 0.540619 | -0.676142 | -1.070949 | 1.206472 |
| 16 | 0.572409 | -0.71816 | -1.179172 | 1.158999 |
| 17 | 0.608673 | -0.760941 | -1.276928 | 1.101434 |
| 18 | 0.649929 | -0.804081 | -1.363221 | 1.035844 |
| 19 | 0.696718 | -0.847063 | -1.437484 | 0.964388 |
| 20 | 0.749578 | -0.889223 | -1.499578 | 0.889223 |
| 21 | 0.809009 | -0.929734 | -1.549775 | 0.812408 |
| 22 | 0.875427 | -0.967588 | -1.588719 | 0.735826 |
| 23 | 0.949106 | -1.001597 | -1.61736 | 0.661104 |
| 24 | 1.030118 | -1.030404 | -1.63688 | 0.589565 |
| 25 | 1.118274 | -1.052528 | -1.648604 | 0.522198 |
| 26 | 1.213071 | -1.066419 | -1.65391 | 0.459657 |
| 27 | 1.313659 | -1.070535 | -1.654152 | 0.40228 |
| 28 | 1.418828 | -1.06343 | -1.650591 | 0.350137 |
| 29 | 1.527029 | -1.043848 | -1.644355 | 0.303082 |
| 30 | 1.63641 | -1.010806 | -1.63641 | 0.260806 |
| 31 | 1.744884 | -0.963665 | -1.627559 | 0.222899 |
| 32 | 1.850207 | -0.90218 | -1.618444 | 0.188887 |
| 33 | 1.950062 | -0.826529 | -1.609569 | 0.158275 |
| 34 | 2.042156 | -0.737327 | -1.601317 | 0.130564 |
| 35 | 2.124302 | -0.635605 | -1.593972 | 0.105275 |
| 36 | 2.194503 | -0.522789 | -1.58774 | 0.08195 |
| 37 | 2.251023 | -0.40065 | -1.582768 | 0.060157 |
| 38 | 2.292446 | -0.271253 | -1.579154 | 0.03949 |
| 39 | 2.317729 | -0.136888 | -1.576963 | 0.019562 |
| 40 | 2.326228 | 0 | -1.576228 | 0 |
| 41 | 2.317729 | 0.136888 | -1.576963 | -0.019562 |
| 42 | 2.292446 | 0.271253 | -1.579154 | -0.03949 |
| 43 | 2.251023 | 0.40065 | -1.582768 | -0.060157 |
| 44 | 2.194503 | 0.522789 | -1.58774 | -0.08195 |
| 45 | 2.124302 | 0.635605 | -1.593972 | -0.105275 |
| 46 | 2.042156 | 0.737327 | -1.601317 | -0.130564 |
| 47 | 1.950062 | 0.826529 | -1.609569 | -0.158275 |
| 48 | 1.850207 | 0.90218 | -1.618444 | -0.188887 |
| 49 | 1.744884 | 0.963665 | -1.627559 | -0.222899 |
| 50 | 1.63641 | 1.010806 | -1.63641 | -0.260806 |
| 51 | 1.527029 | 1.043848 | -1.644355 | -0.303082 |
| 52 | 1.418828 | 1.06343 | -1.650591 | -0.350137 |
| 53 | 1.313659 | 1.070535 | -1.654152 | -0.40228 |
| 54 | 1.213071 | 1.066419 | -1.65391 | -0.459657 |
| 55 | 1.118274 | 1.052528 | -1.648604 | -0.522198 |
| 56 | 1.030118 | 1.030404 | -1.63688 | -0.589565 |
| 57 | 0.949106 | 1.001597 | -1.61736 | -0.661104 |
| 58 | 0.875427 | 0.967588 | -1.588719 | -0.735826 |
| 59 | 0.809009 | 0.929734 | -1.549775 | -0.812408 |
| 60 | 0.749578 | 0.889223 | -1.499578 | -0.889223 |
| 61 | 0.696718 | 0.847063 | -1.437484 | -0.964388 |
| 62 | 0.649929 | 0.804081 | -1.363221 | -1.035844 |
| 63 | 0.608673 | 0.760941 | -1.276928 | -1.101434 |
| 64 | 0.572409 | 0.71816 | -1.179172 | -1.158999 |
| 65 | 0.540619 | 0.676142 | -1.070949 | -1.206472 |
| 66 | 0.512821 | 0.6352 | -0.95366 | -1.241962 |
| 67 | 0.488582 | 0.595587 | -0.829074 | -1.263842 |
| 68 | 0.467519 | 0.557523 | -0.699282 | -1.270816 |
| 69 | 0.449303 | 0.521221 | -0.566629 | -1.261987 |
| 70 | 0.433655 | 0.486914 | -0.433655 | -1.236914 |
| 71 | 0.420349 | 0.454893 | -0.303023 | -1.19566 |
| 72 | 0.409207 | 0.42555 | -0.177444 | -1.138842 |
| 73 | 0.400101 | 0.399433 | -0.059608 | -1.067688 |
| 74 | 0.392953 | 0.377339 | 0.047886 | -0.984101 |
| 75 | 0.387718 | 0.360434 | 0.142613 | -0.890764 |
| 76 | 0.384344 | 0.350429 | 0.222419 | -0.791268 |
| 77 | 0.382646 | 0.349785 | 0.285609 | -0.690277 |
| 78 | 0.381975 | 0.361835 | 0.331317 | -0.593597 |
| 79 | 0.380595 | 0.390489 | 0.360172 | -0.507815 |
| 80 | 0.375 | 0.438986 | 0.375 | -0.438986 |

Plotting the results using MATLAB gives us

Appendix

# Representative C Code and Files

## Part 1

### prog\_1.c

#include <stdio.h>

#include <math.h>

int lin\_root(double, double, double \*);

int quad\_roots(double, double, double, double \*, double \*);

int main(void) {

printf(" Name: <Bhageria, Yadu>\n CID: <00733164>\n LIBRARY NO: <0246618471>\nEmail Address: <yrb13@ic.ac.uk>\n Course Code: <M3SC>\n");

double a2,a1,a0,r1,r2;

int quad\_case;

printf("Enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0\n");

printf("in the order a2,a1,a0, separated by spaces: ");

scanf("%lf %lf %lf",&a2,&a1,&a0);

quad\_case = quad\_roots(a2,a1,a0,&r1,&r2);

switch (quad\_case) {

case -3: printf("The values of a2, a1, a0 resulted in a contradictory equation\n"); break;

case -2: printf("a2=a1=a0=0 so any real number is a root as the equation is tautological\n"); break;

case -1: printf("a2=0 so we are dealing with a linear equation with one root\nr1 = %.10g\n", r1); break;

case 0: printf("The roots are complex.\nr1 = %.10g + %.10gi, r2 = %.10g - %.10gi \n", r1,r2,r1,r2); break;

case 1: printf("There are repeated real roots.\n r1 = r2 = %.10g \n", r1); break;

case 2: printf("There are two real roots.\nr1 = %.10g, r2 = %.10g \n", r1,r2); break;

}

}

int lin\_root(double a1, double a0, double\* r) {

if (a1==0){

if (a0==0){

return(0); // any number is a root

} else{

return(-1); // contradictory

}

}

\*r = -a0/a1; // real root

return(1);

}

int quad\_roots(double a2, double a1, double a0, double\* r1, double\* r2){

double d,dr,four=4,two=2,zero=0;

if (a2==zero) {

return (-2 + lin\_root(a1,a0,r1));

} else if (a0==zero) {

lin\_root(a2,a1,r1);

if (r1<0) {

\*r2=\*r1;

\*r1=zero;

return(2);

} else {

if (\*r1==0) {return (1);}

else {

\*r2=zero;

return(2);}

}

}

d=a1\*a1-four\*a2\*a0;

if (d>zero) {

dr=sqrt(d);

if (a2>zero) {

\*r1=(-a1+dr)/(two\*a2);

\*r2=(-a1-dr)/(two\*a2);

} else {

\*r1=(-a1-dr)/(two\*a2);

\*r2=(-a1+dr)/(two\*a2);

}

return(2);

}

else if (d==zero){

\*r1=(-a1)/(two\*a2);

\*r2=(-a1)/(two\*a2);

return(1);

}

else {

dr=sqrt(d);

\*r1=(-a1)/(two\*a2);

\*r2=(sqrt(-d))/(two\*fabs(a2));

return(0);

}

}

## Part 2

The **data\_in.txt** file contains  
1 2374000 -1

1 2374000 -10000

1 2374000 -1000000

1 2374000 -10000000000

1 1212000 -1

1 1212000 -10000

1 1212000 -1000000

1 1212000 -10000000000

99 99 99

I use the terminal command “./prog\_2 < data\_in.txt” to get the my output from the compiled form of prog\_2.c .

### prog\_2.c

#include <math.h>

#include <stdio.h>

/\*

This version of the code deals with the case where a1^2 >> 4 a0 a2.

I have also tried to implement a function that can easily swap between floats

and double precision numbers and also between the new and old method.

\*/

//#define AS\_FLOAT float; //coment out to change precision from float to double.

#ifdef AS\_FLOAT

#define REAL\_VAR float

#define SQRT sqrtf

#define SCANF scanf("%f %f %f",&a2,&a1,&a0)

#else

#define REAL\_VAR double

#define SQRT sqrt

#define SCANF scanf("%lf %lf %lf",&a2,&a1,&a0)

#endif

int lin\_root(REAL\_VAR, REAL\_VAR, REAL\_VAR \*);

int quad\_roots(REAL\_VAR, REAL\_VAR, REAL\_VAR, REAL\_VAR \*, REAL\_VAR \*);

int quad\_roots\_old(REAL\_VAR, REAL\_VAR, REAL\_VAR, REAL\_VAR \*, REAL\_VAR \*);

int main(void) {

printf(" Name: <Bhageria, Yadu>\n CID: <00733164>\n LIBRARY NO: <0246618471>\nEmail Address: <yrb13@ic.ac.uk>\n Course Code: <M3SC>\n");

REAL\_VAR a2,a1,a0,r1,r2;

int quad\_case;

while (a2!=99 & a1!=99 & a0!=99){

//printf("enter coefficients of Linear Equation a2\*x^2+a1\*x+a0=0\n");

//printf("in the order a2,a1,a0, seperated by spaces\n");

SCANF;

/\*

printf("a2 = %f\n",a2);

printf("a1 = %f\n",a1);

printf("a0 = %f\n",a0);

\*/

//Comment out either depending on which method wants to be used

quad\_case = quad\_roots(a2,a1,a0,&r1,&r2);

//quad\_case = quad\_roots\_old(a2,a1,a0,&r1,&r2);

switch (quad\_case) {

case -3: printf("The values of a2, a1, a0 resulted in a contradictory equation\n"); break;

case -2: printf("a2=a1=a0=0 so any real number is a root as the equation is tautological\n"); break;

case -1: printf("a2=0 so we are dealing with a linear equation with one root\nr1 = %.10g\n", r1); break;

case 0: printf("The roots are complex.\nr1 = %.10g + %.10gi, r2 = %.10g - %.10gi \n", r1,r2,r1,r2); break;

case 1: printf("There are repeated real roots.\n r1 = r2 = %.10g \n", r1); break;

case 2: /\*printf("r1 = %g and r2 = %g \n \n", r1,r2);\*/ printf("%.10g\n", r1); break; //the new printf statement is implemented as we are only interested in the postive root in a simple format.

}

}

}

int lin\_root(REAL\_VAR a1, REAL\_VAR a0, REAL\_VAR\* r) {

if (a1==0){

if (a0==0){

return(0); // any number is a root

} else{

return(-1); // contradictory

}

}

\*r = -a0/a1; // real root

return(1);

}

int quad\_roots(REAL\_VAR a2, REAL\_VAR a1, REAL\_VAR a0, REAL\_VAR\* r1, REAL\_VAR\* r2){

REAL\_VAR d,dr,four=4,two=2,zero=0,dummy;

if (a2==0) {

return (-2 + lin\_root(a1,a0,r1));

} else if (a0==0) {

lin\_root(a2,a1,r1);

if (r1<0) {

\*r2=\*r1;

\*r1=zero;

return(2);

} else {

\*r2=zero;

if (\*r1==zero) {return (1);} else {return(2);}

}

}

d=a1\*a1-four\*a2\*a0;

if (d>zero) {

dr=SQRT(d);

if (a1>zero) {

if (a2>zero) {

\*r2=(-a1-dr)/(two\*a2);

\*r1=two\*a0/(-a1-dr);

} else {

\*r1=(-a1-dr)/(two\*a2);

\*r2=two\*a0/(-a1-dr);

}

} else {

if (a2>zero) {

\*r1=(-a1+dr)/(two\*a2);

\*r2=two\*a0/(-a1+dr);

} else {

\*r2=(-a1+dr)/(two\*a2);

\*r1=two\*a0/(-a1+dr);

}

}

return(2);

} else if (d==zero){

\*r1=(-a1)/(two\*a2);

\*r2=(-a1)/(two\*a2);

return(1);

} else {

\*r1=(-a1)/(two\*a2); //real part

\*r2=(SQRT(-d))/(two\*fabs(a2)); //imaginary part (positive)

return(0);

}

}

//included the old quad roots code to make it easier to output values.

int quad\_roots\_old(REAL\_VAR a2, REAL\_VAR a1, REAL\_VAR a0, REAL\_VAR\* r1, REAL\_VAR\* r2){

REAL\_VAR d,dr,four=4,two=2,zero=0,dummy;

if (a2==0) {

return (-2 + lin\_root(a1,a0,r1));

} else if (a0==0) {

lin\_root(a2,a1,r1);

if (r1<0) {

\*r2=\*r1;

\*r1=zero;

return(2);

} else {

if (\*r1==zero) {return (1);}

else {

\*r2=zero;

return(2);

}

}

}

d=a1\*a1-four\*a2\*a0;

if (d>zero) {

dr=sqrt(d);

if (a2>zero) {

\*r1=(-a1+dr)/(two\*a2);

\*r2=(-a1-dr)/(two\*a2);

} else {

\*r1=(-a1-dr)/(two\*a2);

\*r2=(-a1+dr)/(two\*a2);

}

return(2);

}

else if (d==zero){

\*r1=(-a1)/(two\*a2);

\*r2=(-a1)/(two\*a2);

return(1);

}

else {

dr=sqrt(d);

\*r1=(-a1)/(two\*a2);

\*r2=(sqrt(-d))/(two\*fabs(a2));

return(0);

}

}

## Part 4

**THIS IS THE VERSION OF MY CODE THAT I WANT TESTED**

### prog\_4.c

## Mastery Section

### Matlab Code for producing Plot

filename = 'dataout\_mastery.csv';

D = csvread(filename);

plot(D(:,2),D(:,3),'xb',D(:,4),D(:,5),'xr');

axis('equal');

xlabel('Real Part');

ylabel('Imaginary Part');

legend('W+','W-');

title('Mastery Section: Re(w) vs Im(w) for the two roots of the Equations');

### prog\_m2.c:

produces table in Standard Output (hard to copy values)

#include <stdio.h>

#include <math.h>

#include <complex.h> // Standard Library of Complex Numbers

/\* This version of the code deals with complex numbers and the case where a1^2 >> 4 a0 a2. \*/

int lin\_root(double complex, double complex, double complex \*);

int quad\_roots(double complex, double complex, double complex, double complex \*, double complex \*);

int main(void) {

printf(" Name: <Bhageria, Yadu>\n CID: <00733164>\n LIBRARY NO: <0246618471>\nEmail Address: <yrb13@ic.ac.uk>\n Course Code: <M3SC>\n");

double complex a2,a1,a0,r1,r2,z;

int quad\_case,j;

printf("------------------------------------------------------------------\n");

printf("| j | Re(w+) | Im(w+) | Re(w-) | Im(w-) | \n");

for (j=0;j<81;j=j+1) {

printf("------------------------------------------------------------------\n");

z = cos(j\*M\_PI/40.0) + I \* sin(j\*M\_PI/40.0);

a2 = 12.0;

a1 = -9\*z\*z;

a0 = 24\*z - 8\*z\*z - 12;

/\*

printf("a2 = %.1f%+.1fi\n", creal(a2), cimag(a2));

printf("a1 = %.1f%+.1fi\n", creal(a1), cimag(a1));

printf("a0 = %.1f%+.1fi\n", creal(a0), cimag(a0));

\*/

quad\_case = quad\_roots(a2,a1,a0,&r1,&r2);

switch (quad\_case) {

case -3: printf("The values of a2, a1, a0 were contradictory\n"); break;

case -2: printf("a2=a1=a0=0 so any real number is a root as the equation is a tautology\n"); break;

case -1: printf("a2=0 so we are dealing with a linear equation with one root\n r1 = %.10g + %.10gi\n", creal(1),cimag(r1)); break;

case 1: printf("There are repeated real roots.\n r1 = r2 = %.10g \n", creal(r1)); break;

case 2: printf("| %2d | %12.6f | %12.6f | %12.6f | %12.6f |\n", j,creal(r1),cimag(r1),creal(r2),cimag(r2)); break;

}

}

printf("------------------------------------------------------------------\n");

printf("The roots are ordered by increasing real part. i.e. re(r1) > re(r2)\n");

}

int lin\_root(double complex a1, double complex a0, double complex \* r) {

double complex zero=0;

if (a1==zero){

if (a0==zero){

return(0); // any number is a root

} else{

return(-1); // contradictory

}

} \*r = -a0/a1; // real root

return(1);

}

int quad\_roots(double complex a2, double complex a1, double complex a0, double complex \* r1, double complex \* r2){

double complex four=4,two=2,zero=0,d,dr;

if (a2==zero) {

return (-2 + lin\_root(a1,a0,r1));

} else if (a0==zero) {

lin\_root(a2,a1,r1);

\*r2=zero;

return(2);

}

d=a1\*a1-four\*a2\*a0;

if (d==zero) {

\*r1=(-a1)/(two\*a2);

return(1);

} else {

dr=csqrt(d);

\*r1 = (-a1+dr)/(two\*a2);

\*r2 = (-a1-dr)/(two\*a2);

} return(2);

}

### prog\_m3.c:

produces table in separated by commas. Easy for piping data to .csv file. i.e. by running the command ./prog\_m3.c > dataout\_mastery.csv

#include <stdio.h>

#include <math.h>

#include <complex.h> // Standard Library of Complex Numbers

/\* This version of the code deals with complex numbers and the case where a1^2 >> 4 a0 a2. \*/

int lin\_root(double complex, double complex, double complex \*);

int quad\_roots(double complex, double complex, double complex, double complex \*, double complex \*);

int main(void) {

printf(" Name: <Bhageria, Yadu>\n CID: <00733164>\n LIBRARY NO: <0246618471>\nEmail Address: <yrb13@ic.ac.uk>\n Course Code: <M3SC>\n");

double complex a2,a1,a0,r1,r2,z;

int quad\_case,j;

printf("j, Re(w+), Im(w+), Re(w-), Im(w-)\n");

for (j=0;j<81;j=++j) {

z = cos(j\*M\_PI/40.0) + I \* sin(j\*M\_PI/40.0);

a2 = 12.0;

a1 = -9\*z\*z;

a0 = 24\*z - 8\*z\*z - 12;

/\*

printf("a2 = %.1f%+.1fi\n", creal(a2), cimag(a2));

printf("a1 = %.1f%+.1fi\n", creal(a1), cimag(a1));

printf("a0 = %.1f%+.1fi\n", creal(a0), cimag(a0));

\*/

quad\_case = quad\_roots(a2,a1,a0,&r1,&r2);

switch (quad\_case) {

case -3: printf("The values of a2, a1, a0 were contradictory\n"); break;

case -2: printf("a2=a1=a0=0 so any real number is a root as the equation is a tautology\n"); break;

case -1: printf("a2=0 so we are dealing with a linear equation with one root\n r1 = %.10g + %.10gi\n", creal(1),cimag(r1)); break;

case 1: printf("There are repeated real roots.\n r1 = r2 = %.10g \n", creal(r1)); break;

case 2: printf("%2d,%10.6f,%10.6f,%10.6f,%10.6f\n", j,creal(r1),cimag(r1),creal(r2),cimag(r2)); break;

}

}

}

int lin\_root(double complex a1, double complex a0, double complex \* r) {

double complex zero=0;

if (a1==zero){

if (a0==zero){

return(0); // any number is a root

} else{

return(-1); // contradictory

}

} \*r = -a0/a1; // real root

return(1);

}

int quad\_roots(double complex a2, double complex a1, double complex a0, double complex \* r1, double complex \* r2){

double complex four=4,two=2,zero=0,d,dr;

if (a2==zero) {

return (-2 + lin\_root(a1,a0,r1));

} else if (a0==zero) {

lin\_root(a2,a1,r1);

\*r2=zero;

return(2);

}

d=a1\*a1-four\*a2\*a0;

if (d==zero) {

\*r1=(-a1)/(two\*a2);

return(1);

} else {

dr=csqrt(d);

\*r1 = (-a1+dr)/(two\*a2);

\*r2 = (-a1-dr)/(two\*a2);

} return(2);

}

1. ,2 http://nssdc.gsfc.nasa.gov/planetary/factsheet/plutofact.html [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. <https://en.wikipedia.org/wiki/Microsoft_Visual_Studio> for finding out Microsoft Visual Studio 11.0 is 2012 and 12.0 is 2013. [↑](#footnote-ref-3)
4. http://forge.scilab.org/index.php/p/docscilabisnotnaive/ [↑](#footnote-ref-4)