Lab Assignment #8

Due 3/18/2016 at 4pm on bCourses

The assignment will be partially graded by an autograder which will check the values of the required variables, so it is important that your function names are exactly what they are supposed to be and the input and output arguments are in the correct order (names ARE case-sensitive). Instructions for submitting your assignment are included at the end of this document.

1 Linear Systems: the Basics

1.1 Solving Linear Systems

Shown below is a linear system of 3 equations with 3 unknowns, x_1, x_2, and x_3. The scalar quantities a_{ij} and b_i can be assumed known. Under certain conditions (which we will explore later), there is a unique solution for x_1, x_2, and x_3. Here we will assume that these conditions are met, so you don't have to check them.

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{1}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{2}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{3}$$

Your job is to write a function with the header

```
function [x1, x2, x3] = myLinearSolver(A1, b1, A2, b2, A3, b3)
```

where `A1` is a 1×3 double containing the three coefficients for equation 1 (a_{11}, a_{12}, a_{13}) and `b1` is a scalar double representing b_1, the right-hand-side of equation 1. Similarly, `A2` and `A3` are 1×3 doubles containing the coefficients of equations 2 and 3, respectively. `b2` and `b3` are scalar doubles representing the right-hand-sides of equations 2 and 3, respectively.

Your function should have 3 outputs `x1`, `x2`, and `x3`. Each output should be a scalar double. Together, the outputs `x1`, `x2`, and `x3` represent the solution (x_1, x_2, x_3) to equations 1-3 above.

Your function should use the backslash \ operator exactly once. **Hint:** This function can be completed in as few as 6 lines, not counting comments, header, and "end". You may assume that the inputs are of the appropriate class and size.

Test cases:

```
>> [x1,x2,x3] = myLinearSolver([1 0 -1],1,[0 -1 1],2,[2 1 1],0)
x1 =
     1
x2 =
    -2
x3 =
     0
```

```
>> [x1,x2,x3] = myLinearSolver([1 1 1], 8, [2 0 3], 13, [3, -1, -2], 11)
x1 =
    5
x2 =
    2
x3 =
    1
```

1.2 Matrix Inversion

Write a function with the header:

```
function [B] = matInv(A)
```

that returns the inverse of the matrix **A**, i.e., the matrix **B** such that **AB** = **I**, where **I** is the identity matrix. If **A** is not square or the determinant of **A** is within 0 ± 10^{-10}, the function should return the empty array [ ].

Test cases:

```
>> Input = [1 2; 3 4];
>> Output = matInv(Input)
Output =
   -2.0000    1.0000
    1.5000   -0.5000

>> Input*Output
ans =
    1.0000      0
    0.0000      1.0000


>> Input = [1 2; 3 4; 5 6];
>> Output = matInv(Input)
Output =
   []

>> Input = [1 0 1; 2 0 2; 1 2 0];
>> Output = matInv(Input)
Output =
   []
```

2 Linear Systems: Circuits

In this question, we will look at how to model an electric circuit using a system of linear equations to solve for the currents in different parts of the circuits.

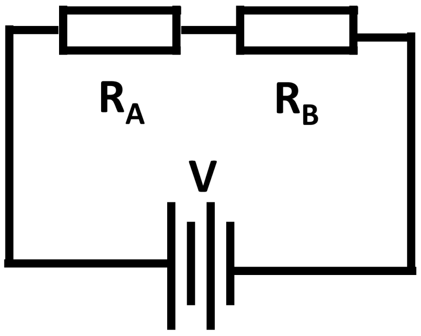

Figure 1: A circuit with two resistors in series

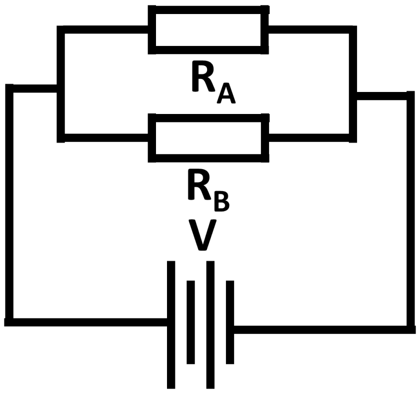

Figure 2: A circuit with two resistors in parallel

There are two types of simple circuits, series and parallel, as shown above. R_A and R_B denote the resistance of resistor A and B, respectively. R denotes the overall resistance, and V denotes the voltage difference across the battery. I_A and I_B are the current flowing through A and B, respectively, and V_A and V_B are the voltage differences across A and B. To analyze a circuit, you start at one point in the loop and "advance" until you are back where you started. We will guide you through the setup of the equations you need to solve.

The following relationships apply to this circuit in series (Figure 1):

$$\begin{aligned} &R = R_A + R_B \quad \text{(total resistance is the sum of the two resistors)} \\ &I_A = I_B \quad \text{(the current through resistor A is the same as through B)} \\ &V = V_A + V_B = I_A R_A + I_B R_B \quad \text{(the voltage difference across the battery is} \\ &\quad \text{equal to the sum of voltage differences across resistors A and B)} \end{aligned} \tag{4}$$

Notice that voltage is equal to the product of the resistance and the current ($V = IR$).

The following relationships apply to this circuit in parallel (Figure 2):

$$\begin{aligned} &\frac{1}{R} = \frac{1}{R_A} + \frac{1}{R_B} \quad \text{(the resistances are now inversely related)} \\ &V = V_A = V_B = I_A R_A = I_B R_B \quad \text{(the voltage difference across the battery is} \\ &\quad \text{equal to the voltage difference across each resistor)} \end{aligned} \tag{5}$$

For more on serial and parallel circuits, see links **here** and **here**.

As an optional first step, you can solve for the currents I_A and I_B in these simple circuits by hand, for the values given here: $R_A = 3\ \Omega$, $R_B = 6\ \Omega$, $V = 9$ V. Practice setting up the matrix that goes along with the equations, and see if your matrix solver from question 1 gives you the same answer you worked out by hand. You should find that for the circuit in series (Figure 1), $I_A = I_B = 1$ A, and for the circuit in parallel (Figure 2), $I_A = 3$ A, $I_B = 1.5$ A (where A denotes the unit amperes).

Now you are given a more complicated circuit, shown in Figure 3. Don't worry if you have not encountered circuits before. You should ask for help for the physics, since physics is not the main point of the question. You are to write a function that solves for the currents specifically in this circuit, with the following header:

```
function [I1,I2,I3,I5,I6,I7] = MyCircuit(V, R1, R2, R3, R4, R5, R6, R7, R8)
```

where the `I`'s are currents through resistors 1 to 8, in amperes; `V` is the battery's voltage, in volts; and `R`'s are the resistances in ohms.

Hint: Before you start coding, use a pencil and paper to write out a system of equations. Then, also with pencil and paper, rearrange the system of equations into the form $\mathbf{Ax} = \mathbf{b}$, creating the appropriate matrix $\mathbf{A}$ (which is a square matrix), and solution vector $\mathbf{b}$ (which is a column vector). Note the order of the outputs of the function: `I4` and `I8` are not output because they are equal to `I1`.

Test case:

```
>> [I1,I2,I3,I5,I6,I7] = MyCircuit(10,1,4,4,2,2,1,4,5)
I1 =
    0.9459
I2 =
    0.4730
I3 =
    0.4730
I5 =
    0.2703
I6 =
    0.5405
I7 =
    0.1351
```

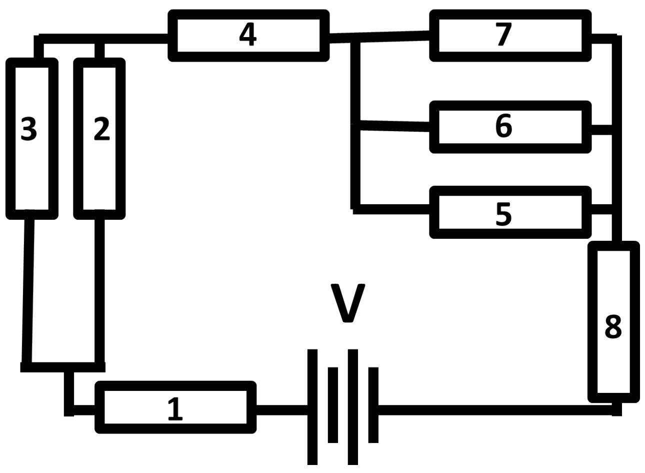

Figure 3: MyCircuit

Perspective: this problem shows the benefit of programming. Without a computer, you would need to solve a large algebraic system over again any time one of the 9 input values changed (voltage or resistance). Using MATLAB, you can formulate the equations once, write one function, and solve for any variation with only one command.

3 Root Finding: Methods

3.1 Newton's Method

Write a function with header:

```
function [R, E] = myNewton(f, df, x0, tol)
```

where `f` is a function handle representing the function $f(x)$, `df` is a function handle to the derivative of `f`, `x0` is an initial estimate of the root, and `tol` is a strictly positive scalar double. The function should return a row vector `R` of class `double`, where `R(1)` is the initial estimate `x0`, and `R(k+1)` is the estimate of the root of f after k iterations in the Newton Method. The function should also return the absolute error in a row vector `E`, of class `double`, where `E(k)` is the value of $|f(R(k))|$. The function should return when `E(end) < tol` or after 100 iterations have been performed. If N iterations of Newton's method have been performed, the outputs should each have $N+1$ elements. You may assume that the derivative of `f` will not be 0 during any iteration for any of the test cases given. MATLAB built-in functions `fzero` and `roots` may NOT be used. Do not use a `while` loop.
Test cases:

```
>> f = @(x) x^2 - 2;
>> df = @(x) 2*x;
>> [R, E] = myNewton(f, df, 1, 1e-5)
R =
    1.0000  1.5000 1.4167   1.4142

E =
    1.0000  0.2500 0.0069   0.0000

>> f = @(x) sin(x) - cos(x)
>> df = @(x) cos(x) + sin(x);
>> [R, E] = myNewton(f, df, 1, 1e-5)
R =
    1.0000  0.7820 0.7854

E =
    0.3012  0.0047  0.0000
```

3.2 Bisection Method

Write a function with header:

```
function [R, E] = myBisection(f, a, b, tol)
```

where `f` is a function handle, `a` and `b` are scalars such that `a < b`, and `tol` is a strictly positive scalar value. The function should return a row vector, `R` of class `double`, where `R(k)` is the estimation of the root of `f` after k iterations of the bisection method. The first element of `R` should be the midpoint of the interval defined by the inputs `a` and `b`, and counts as the first step of the bisection method. The function should also return a row vector `E` of class `double`, where `E(k)` is the value of $|f(R(k))|$. The function should return when `E(k) < tol` or after 100 iterations have been performed. If N iterations of the bisection method are performed, the outputs should each have N elements. You may assume that `sign(f(a)) = - sign(f(b))` for the first guess. (When you are testing your function, it will help to plot your function first, so you can pick appropriate values to bracket the root.)
Note: The inputs `a` and `b` constitute the first iteration of bisection, and therefore `R` and `E` should never be empty. Matlab built-in functions `fzero` and `roots` may NOT be used. Do not use a `while` loop.
Test cases:

```
>> f = @(x) x.^2 - 2;
>> [R, E] = myBisection(f, 0, 2, 1e-1)
R =
    1.0000  1.5000  1.2500  1.3750  1.4375
E =
    1.0000  0.2500  0.4375  0.1094  0.0664
```

```
>> f = @(x) sin(x) - cos(x);
>> [R, E] = myBisection(f, 0, 2, 1e-2)
R =
    1.0000  0.5000  0.7500  0.8750  0.8125  0.7813
E =
    0.3012  0.3982  0.0501  0.1265  0.0383  0.0059
```

3.3 Further Exploration (Optional)

In this question you explored two iterative root finding methods. Each of these functions provides a series of x values as the computer "zooms in" on the root. Experiment with different types of functions as well as different initial intervals and estimates, and see how the different root finding methods behave. Do they always converge? How quickly do they converge?
To get started, try entering the following:

```
f = @(x) x.^3 - 3*x - 5;
df = @(x) 3*x.^2 - 3;
[rn, en] = myNewton(f, df, 3, 1e-4);
[rb, eb] = myBisection(f, 1, 3, 1e-4);
x = 2:0.1:3;
plot(x,f(x),'r-', rn,f(rn),'bo', rb,f(rb),'go')
```

You should see "linear convergence" for one method and "quadratic convergence" for the other (quadratic meaning that the number of correct decimal places approximately doubles with each iteration). You may want to use a log plot. This section will not be graded, but completing it and thinking about the concepts will help improve your understanding of root finding.

4 Root Finding: Fractal generation

In this question, we are exploring root finding on the complex number plane. A complex number can be written as:

$$z = x + yi \tag{6}$$

where x and y are two real numbers, i is the imaginary number such that $i^2 = -1$, and z is a complex number. Note that x is the real part of z and y is the complex part of z.

Complex number calculations work similarly to real numbers. Below are four examples using z_1 and z_2 for addition, subtraction, multiplication, and division (to help refresh your memory):

$$\begin{gathered}
z_1 = 1 + i; \qquad z_2 = 3 - 2i \\
z_1 + z_2 = 1 + i + 3 - 2i = 4 - i \\
z_1 - z_2 = 1 + i - (3 - 2i) = -2 + 3i \\
z_1 z_2 = (1 + i)(3 - 2i) = 3 - 2i + 3i - i(2i) = 3 + i - 2(i^2) = 3 + i - 2(-1) = 5 + i \\
\frac{z_1}{z_2} = \frac{1 + i}{3 - 2i} = \frac{(1 + i)(3 + 2i)}{(3 - 2i)(3 + 2i)} = \frac{3 + 2i + 3i - 2}{9 + 6i - 6i - 4i^2} = \frac{1 + 5i}{13} = \frac{1}{13} + \frac{5}{13}i
\end{gathered}$$

Complex numbers can be roots to a polynomial. For instance, for the equation:

$$\begin{aligned}
z^3 + 1 &= 0 \\
z^3 &= -1
\end{aligned} \tag{7}$$

the solutions are: $z_1 = -1$, $z_2 = \frac{1}{2} + \frac{\sqrt{3}}{2}i$, and $z_3 = \frac{1}{2} - \frac{\sqrt{3}}{2}i$. (Try plugging z_1, z_2 and z_3 back into the equation to check for yourself).

4.1 Exact complex solutions to a polynomial equation

In this section, you will learn how to solve for the exact complex solutions to a polynomial in the form of Eq. 8:

$$z^d = 1 \tag{8}$$

where d is a positive integer. Eq. 8 has d roots. For any complex number, we can transform it into the polar form:

$$z = x + iy = |z|(\cos\phi + i\sin\phi) = re^{i\phi} \tag{9}$$

where $x = Re(z)$, the real part of the complex number z; $y = Im(z)$, the imaginary part of z; $r = |z| = \sqrt{x^2 + y^2}$, the magnitude of z; and $\phi = arg(z)$, the angle in radians between the x axis on a complex plane and the vector made by z. For instance, when $r = 1$, Figure 4 shows the argument ϕ.

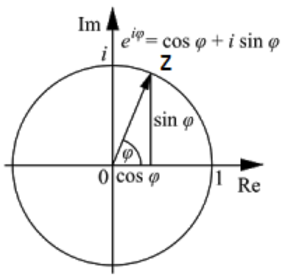

Figure 4: $z = x + iy = re^{i\phi}$ on a complex plane. Image from `https://sites.google.com/site/greatmathmoments/identity`.

Rewriting Eq. 8 in the exponential form, we have:

$$\begin{aligned}(re^{i\phi})^d &= e^{2\pi ni}\\ r^d(e^{i(\phi d)}) &= (1)e^{2\pi ni}\end{aligned} \tag{10}$$

where n equals $0, 1, 2, 3...d-1$. Eq. 10 gives us $r^d = 1$ and $\phi d = 2\pi n$. Therefore, $r = 1$ and $\phi = \frac{2\pi}{d}n$. So the solutions to Eq. 8 are: 1, $e^{i\frac{2\pi}{d}}$, $e^{2i\frac{2\pi}{d}}$, $e^{3i\frac{2\pi}{d}}$, $...e^{i(2\pi)\frac{d-1}{d}}$.
To practice solving complex polynomial with the polar form, let's look at the polynomial $z^3 = -1$ again (Equation 7). To solve this equation using the polar form, we have:

$$\begin{aligned}z^3 &= -1\\ (re^{i\phi})^3 &= e^{(2n+1)\pi i}\end{aligned} \tag{11}$$

where n equals $0, 1, 2$. Therefore:

$$r^3 = 1$$
$$3\phi = (2n+1)\pi$$

Solving for r and ϕ:

$$\begin{aligned}r &= 1\\ \phi &= \frac{2n+1}{3}\pi\\ &= \frac{1}{3}\pi, \pi, \frac{5}{3}\pi\end{aligned} \tag{12}$$

Therefore, the solutions to $z^3 = -1$ are: $z_1 = e^{i\frac{1}{3}\pi} = -\frac{1}{2} + \frac{\sqrt{3}}{2}i$, $z_2 = e^{i\pi} = -1$, and $z_3 = e^{i\frac{5}{3}\pi} = -\frac{1}{2} - \frac{\sqrt{3}}{2}i$, as we found before.

For another example, let's look at the polynomial $z^5 = 1$. To solve this equation using the polar form, we have:

$$\begin{aligned}z^5 &= 1\\ (re^{i\phi})^5 &= e^{(2n)\pi i}\end{aligned} \tag{13}$$

where n equals $0, 1, 2, 3, 4$. Therefore:

$$r^5 = 1$$
$$5\phi = (2n)\pi$$

Solving for r and ϕ:

$$\begin{aligned} r &= 1 \\ \phi &= \frac{2n}{5}\pi \\ &= 0, \frac{2}{5}\pi, \frac{4}{5}\pi, \frac{6}{5}\pi, \frac{8}{5}\pi \end{aligned} \tag{14}$$

Therefore, the solutions to $z^5 = 1$ are:

$$\begin{aligned} z_1 &= e^{i0} = 1 + 0i \\ z_2 &= e^{i\frac{2}{5}\pi} = 0.309 + 0.9511i \\ z_3 &= e^{i\frac{4}{5}\pi} = -0.809 + 0.5878i \\ z_4 &= e^{i\frac{6}{5}\pi} = -0.809 - 0.5878i \\ z_3 &= e^{i\frac{8}{5}\pi} = 0.309 - 0.9511i \end{aligned} \tag{15}$$

Now that you know how to solve complex polynomial, write a function with the header:

```
function [solutions] = PolynomialExactSolutions(d)
```

where d is the order of the polynomial from Eq. 8. Your function should output an array of `d` elements that contain the complex solutions to Eq. 8.
Note: You need to find ϕ for $n = 0, 1, 2...$ using the pattern given above, then transform the polar form to rectangular form ($z = x + yi$). You are not allowed to use any root-finding MATLAB functions such as `fzero` or `roots` in the function `PolynomialExactSolutions(d)`. Your function should output the roots in order, as seen in the test cases below.

Test Cases (roots for the first case are illustrated in Figure 5):

```
>>solutions = PolynomialExactSolutions(3)
solutions =

  1.0000      -0.5000 + 0.8660i    -0.5000 - 0.8660i

>> solutions=PolynomialExactSolutions(9)

solutions =

  Columns 1 through 3

   1.0000    0.7660 + 0.6428i   0.1736 + 0.9848i
  Columns 4 through 6

  -0.5000 + 0.8660i   -0.9397 + 0.3420i   -0.9397 - 0.3420i

  Columns 7 through 9

  -0.5000 - 0.8660i   0.1736 - 0.9848i    0.7660 - 0.6428i
```

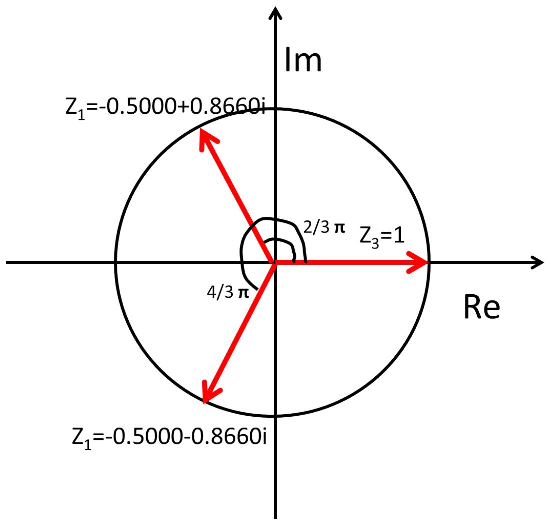

Figure 5: Solutions to $z^3 = 1$ presented on a complex plane. (Your function `PolynomialExactSolutions` is not expected to output this figure.)

4.2 Convergence using Newton's method

Different starting points on a complex plane, when chosen as the initial guess to Newton's method, will converge to different roots. In this section, even though we know the roots to a complex polynomial, we are trying to see whether Newton's method can reach a root in a given number of iterations, and if yes, which roots these different initial guesses will converge to.

Now pick any point on a complex plane, `z`, use it as the initial guess to a root of Eq. 8, and apply Newton's method `n` times on the complex plane `z`. Write a function with the header:

```
function [ConvergeOrNot, ConvergeToRoot] = NewtonConv(d, z, n, tol)
```

where `d` is the order of the polynomial from Eq. 8, `z` is a complex number as the initial guess, `n` is the number of iterations for which we apply Newton's method, and `tol` is the tolerance which determines if the output has converged after `n` iterations of Newton's method. If the iteration result is within tolerance of a certain root, `ConvergeOrNot` equals 1, and `ConvergeToRoot` is the root it's converging to; if the iteration result is not within tolerance of any roots, `ConvergeOrNot` should be 0, and `ConvergeToRoot` is `NaN`. All output should be of class `double`.

Hint: Call the function `PolynomialExactSolutions` you wrote to generate an array containing the true roots. Since `NewtonConv` relies on `PolynomialExactSolutions`, be extra careful to check that it works correctly before you start `NewtonConv`!

Test cases:

```
>>[ConvergeOrNot,ConvergeToRoot]=NewtonConv(3,1+i,5,0.001)

ConvergeOrNot =
     0
ConvergeToRoot =
   NaN

>> [ConvergeOrNot,ConvergeToRoot]=NewtonConv(3,1+i,10,0.001)

ConvergeOrNot =
     1
ConvergeToRoot =
   1.0000 - 0.0000i

>> [ConvergeOrNot,ConvergeToRoot]=NewtonConv(3,10+10*i,10,0.001)

ConvergeOrNot =
     0
ConvergeToRoot =
   NaN

>> [ConvergeOrNot,ConvergeToRoot]=NewtonConv(3,-10+10*i,100,0.001)

ConvergeOrNot =
     1
ConvergeToRoot =
  -0.5000 + 0.8660i
```

-

4.3 Newton fractal

As you can see from the function `NewtonConv`, different starting points `z` will converge to different roots. You can color the points on the complex plane according to which root of the polynomial each of them converge to. This will generate a Newton fractal. You can Google "Newton Fractals" and look at the pretty pictures. Here you will write a function to generate your own Newton's fractal for polynomials like Eq. 8. The function should have the header:

```
function [output] = NewtonFractal(d,n,tol,res,ULcorner,sqrL)
```

where `d`, `n` and `tol` are as defined in the function `NewtonConv`, `res` is a scalar double indicating the total number of points in the x and y axis of your plot (resolution), `ULcorner` is a 1-by-2 double array containing the x and y coordinates of the upper-left corner of your plot's domain, and `sqrL` is the length of the square which makes up your domain (your domain will always be a square). (An example domain is illustrated in Figure 6.)

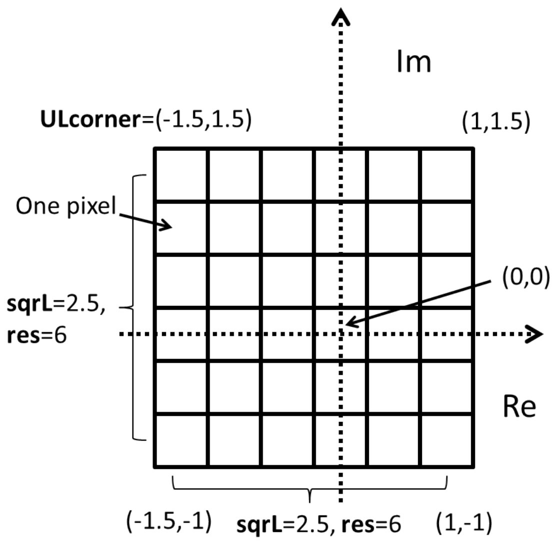

Figure 6: The example domain created by input `ULcorner=(-1.5,1.5), sqrL=2.5, res=6`.

Your function will output two things: a `res`-by-`res` matrix containing values $1, 2, 3...d$ that refer to which root a point in the plotting domain converges to, or 0 for a point which does not converge to any root. Your function will also output a square image of length `sqrL` of this matrix, with 0 colored white. You can define the colors of other values $(1, 2, 3, ..., d)$ to your own liking. Your *output* has to match the rule of assignment given in the guide below; your plot should have the same title structure, domain size, and axis labels, but it does not need to have the same color scheme.

Your plot will show the range of your x and y axes, and have a title indicating `d`, `n`, `tol`, and `res`. Refer to Figure 7 and 8 for plotting examples. You should suppress the *call* of the function at the command line, as `output` can be a very large matrix.
You can follow the guide below to start the problem:

1. Define your x and y axis arrays based on `res`, `ULcorner` and `sqrL`. Then use `meshgrid` to generate `X` and `Y`.

2. Define `Z` using `X` and `Y`. Every element in `Z` should be a complex number.

3. Perform Newton's method `n` times using each value of `Z` as a starting point.

4. Find the roots of the polynomial using `solutions=PolynomialExactSolutions(d)`. We need to know the exact roots. This is in addition to what we do using Newton's method next.

5. Define `output` as a matrix of 0's and of same size as `Z`. Loop through every root in `solutions` using a `for` loop. For each root, check which elements in complex plane `Z` converge to this root, and assign an integer from 1 to d to these elements by recording the integer in the appropriate position in `output`. For instance, if an element in `Z` converges to the first root output by `PolynomialExactSolutions`, you assign 1 to the same location in `output`; if it converges to the second root output by `PolynomialExactSolutions`, assign 2 to the same location in `output`; so on and so forth.

6. You end up with the matrix `output` that has `d` different integers and may or may not have 0's. (The higher the iteration `n`, the less likely you will get 0's, because Newton's method will have more iterations to get within the tolerance `tol` of an exact root. But the higher the iteration, the more computing time it takes.)

7. Now plot `output`. Define your own set of colors associated with the different integers. Zeros should be white.
   To define your own color scheme for a `Z` that contains 0's, you can generate a `(d+1)`-by-3 array, such as:

```
mycolormap=[1 1 1;
            1 0 0; 0 1 0; 0 0 1;
            1 1 0; 0 1 1; 1 0 1];
% This gives a 7-element array of white, red, green, blue,
%yellow, cyan, purple.
% OR
R=linspace(1,0,d+1); % size of d+1
G=R;
B=G; % Define a GRAY scale from (1 1 1) to (0 0 0);
mycolormap=[R;G;B]';
% OR
mycolormap=[1 1 1; hsv(d)]; % The first row is white color, and
% then pick d number of colors from the built-in palette hsv.

% You can also look up other built-in palettes,
% such as 'spring','hot','jet', etc.
```

   Then you can use `imagesc` to plot `Z` and apply your colors using `colormap(mycolormap)`.

Test case (Figure 7):

```
>>d=7;
>>n=200;
>>tol=1e-4;
>>res=200;
>>ULcorner=[-1 1];
>>sqrL=2;
>>output=NewtonFractal(d,n,tol,res,ULcorner,sqrL);
```

Test case (Figure 8):

```
>>n=25; % Keep d,tol,res,ULcorner,sqrL the same, just change the iterations
>>output=NewtonFractal(d,n,tol,res,ULcorner,sqrL);
```

Test case (Figure 9):

```
>>d=3;
>>n=200;
>>tol=1e-5;
>>res=500;
>>ULcorner=[-1.5 1.5];
>>sqrL=3;
>>output=NewtonFractal(d,n,tol,res,ULcorner,sqrL);
```

Note how Figure 9 compares to Figure 5, both for $z^3 = 1$. It's along the mid-point edge of each pair of roots where fractals get generated.

Try varying the size of your domain and look at the intricate details of the fractal. You can zoom in and out by clicking the zoom icons on top of the plot window. As you zoom into particular regions, you will discover the self-similarity of the fractal.

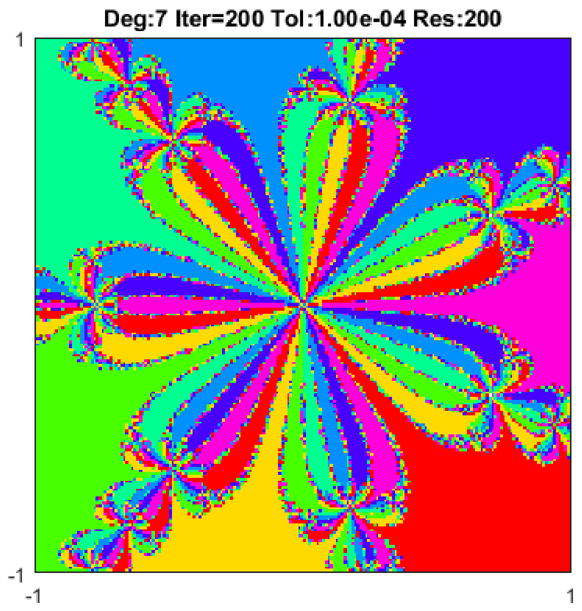

Figure 7: Example plot with $n = 200$.

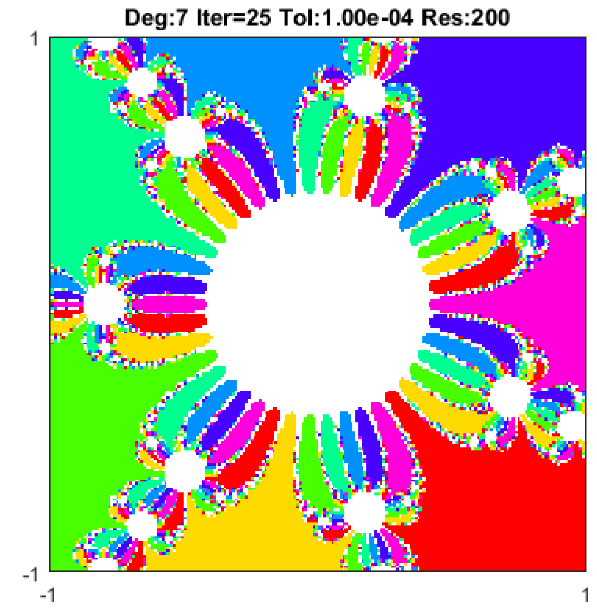

Figure 8: Example plot with $n = 25$.

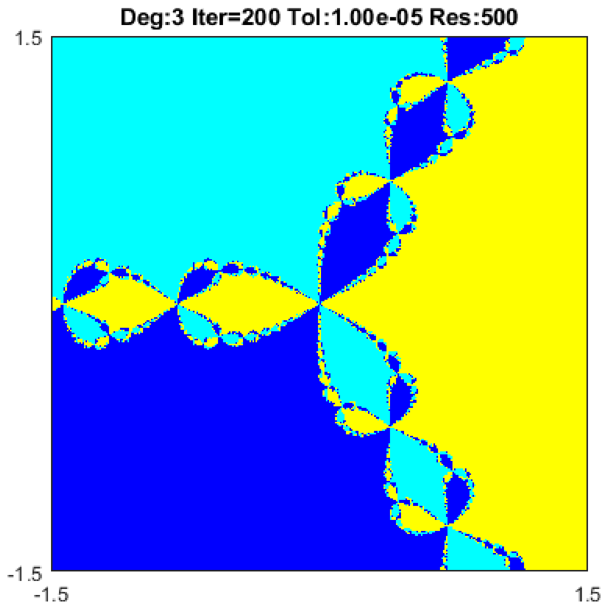

Figure 9: Example plot with $d = 3$.

4.4 Optional: Add shading to Newton's fractal

You have plotted Newton fractal with discrete colors. Now, when you Google "Newton fractal", you may come across something like Figure 10.

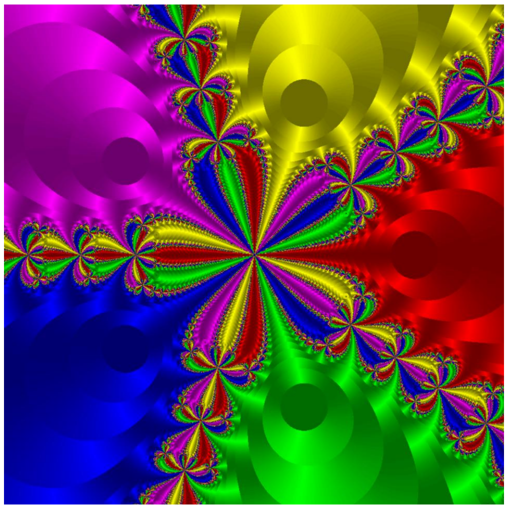

Figure 10: A Newton fractal plotted with shading

Note that in Figure 10, there are different shades of red, yellow, green, etc. The colors are determined by which root a guess converges to, as we programmed in `NewtonFractal`, and the shades are determined by how many iterations are needed to reach the roots. To plot something similar to Figure 10, you count the number of iterations in the function, instead of having the number of iterations as an input variable. Your function may look like:

```
function [output] = ShadedNewtonFractal(d,tol,res,ULcorner,sqrL)
```

where your `output` is no longer a matrix containing integer values, but a matrix containing numbers with decimal points that indicate *both* which root an initial guess converges to *and* how many iterations it takes. The integer part of the number indicates the root, and the decimal part of the number indicates the number of iterations. For instance, 1.005 means it takes 5 iterations to reach the first root; and 4.580 means it takes 580 iterations to reach the fourth root. Then you can play around with `mycolormap` to obtain the right color and shade for each number in `output`. Only try this part if you finish all the required problems above.

Submission Instructions

Your submission should include the following function files in a single zip file named Lab8.zip

- myLinearSolver.m
- matInv.m
- MyCircuit.m
- myNewton.m
- myBisection.m
- PolynomialExactSolutions.m
- NewtonConv.m
- NewtonFractal.m

Don't forget that the function headers and file names you use must appear *exactly* as they do in this problem set. So, make sure the variables have the right capitalization, the functions have the correct name, and the inputs and outputs are in the correct order.