# Numerical Methods in Scientific Computing 2021
# Answers Ex04

Jake Muff 9/02/21

## Problem 1

$$f(x) = \sin\left(3\pi \frac{x^3}{x^2 - 1}\right) + \frac{1}{2}$$

1. Function to implement the bisection method for finding the root of $f(x)$ in the interval $[a, b]$. This was completed in the roots.cpp file. The algorithm works first by checking if the input values of $a$ and $b$ are correct but having the condition that $func1(a) * func1(b) \geq 0$ as well as a condition that $b > a$. The bisection method then continues by finding the midpoint and assigning new values to the variables using a series of IF-ELSE statements.

2. The next bit was to find the roots using Newton's method or the Newton-Raphson method. This is listed under the newton_f function which takes single input, the initial guess, $x0$. This method works by calculating the ratio between the function and its derivative at an initial point and iterating until the fraction reaches a limit, in this case $\epsilon = 0.01$ for both methods so they are comparable. As shown in the figures below, the Newton Method provides an answer closer to the analytical value, however, Newton's Method is dependent on the initial guess which I put in as substantially close to the real value. I did this because of the anture of the function it has many and multiple roots, thereofore to choose the same root as in the bisection method an substantially close value was needed. Newton's method is also prone to more typos/errors in making the derivative of the function and is reliant on that derivative.

3. Both methods were used to find the root in the $[0, 1]$ interval. The Bisection method reached $\underline{0.367188}$ after 7 iterations whereas Newton Method reached $\underline{0.363923}$ after 3 iterations for the same $\epsilon = 0.01$. When the same accuracy is required this shows that Newton's method is more computationally efficient as it does it in 4 less iterations, however, this is also dependent on the initial guess whereas the Bisection method is not. Knowing this, the best method is to initialize the initial guess with the Bisection method to use in Newton's method to gain the most computationally efficient (almost) foolproof way to find the root in the interval which is very accurate.

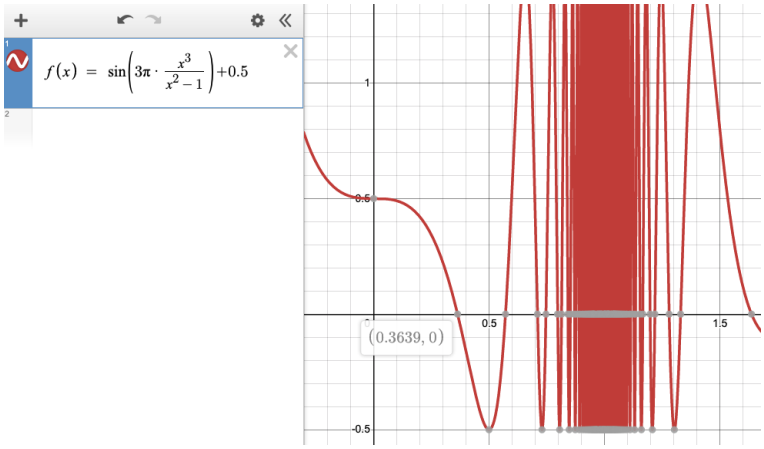



Figure 1: Plot of $f(x)$ using Desmos. In the interval $[0, 1]$ the root is shown as 0.3639

```
jake@Jakes-MBP Exercises 4 % make
clang++ -std=c++17 -Wall -pedantic roots.cpp -o roots
jake@Jakes-MBP Exercises 4 % ./roots
Iterations = 7
Root for f(x) using Bisection, f(x) =0 when x= 0.367188
jake@Jakes-MBP Exercises 4 %
```

Figure 2: Output from Bisection Method for $f(x)$ for the interval $[0, 1]$. It shows the root as 0.367188

```
jake@Jakes-MBP Exercises 4 % ./roots
Begin newton method
x0 d
0.3 -0.0781841
-------------
x_i d
0.378184 -0.0781841
0.364279 0.0139048
0.363923 0.000356409
-------------
end newton method
-------------
root is 0.363923 after 3 iterations
jake@Jakes-MBP Exercises 4 %
```

Figure 3: Output from Newton Method for $f(x)$ with initial guess 0.3. It shows the root as 0.363923

# Problem 2

Finding the unique zero of

$$g(x) = x + e^{-Bx^2}\cos(x)$$

1. The code for this is shown in roots.cpp under the function newton_g. The function takes to inputs and uses the same method shown as before. The only key difference is that the function and its derivative change with respect to $B$. Also a max iteration break statement is added to ensure that the function doesn't run away (infinite loop).

2. When running the code for $B = 0.1, 1$ an inittial guess of $x0 = 1$ is fine and it converges, however for $B = 10, 100$ you start encountering errors that require a more precise $x0$, partially due to lack of convergence and also to do with the nature of the function as you have a $e^{Bx0^2}$ in the derivative which can cause issues as well as $e^{-Bx0^2}$. Both of which can cause issues for large $B$ because it reaches the limits of the computer. To get around this simply use a more precise initial guess but recognize that the inittial guess can be finicky.

$$B = 0.1 \quad r = -0.71644 \quad x0 = 1$$

$$B = 1 \quad r = -0.588402 \quad x0 = 1$$

$$B = 10 \quad r = -0.326375 \quad x0 = 0.14$$

$$B = 100 \quad r = -0.139029 \quad x0 = 0.14$$

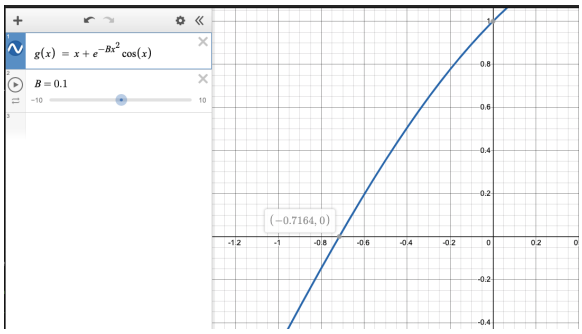
The roots given match the expected analytical values. When using an initial guess of 0 we see that for $B = 0.1, 1$ the root is found fine but for $B = 10, 100$ the max iterations is reached. If we break down what is happening here we notice that an infinite loop occurs where it never converges on a root as it simply goes from $-1$ to 0 to $-1$ every loop due the nature of the function for high $B$ values.

```
[jake@Jakes-MBP Exercises 4 % ./roots
----Problem 2----
Root for B= 0.1 root= -0.71644 for initial guess 1
Root for B= 1 root= -0.588402 for initial guess 1
Root for B= 10 root= -0.326375 for initial guess 0.14
Root for B= 100 root= -0.139029 for initial guess 0.14
jake@Jakes-MBP Exercises 4 %
```

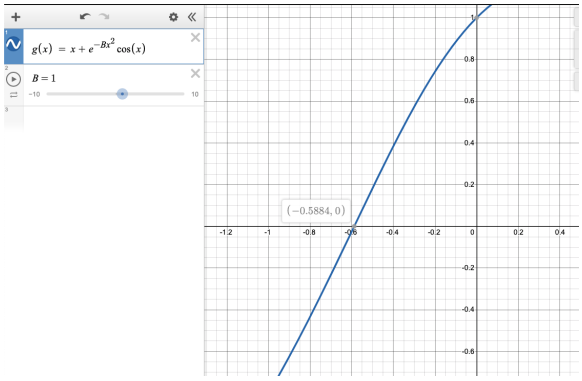
Figure 4: Output from Newton Method for $g(x)$

```
----Problem 2----
Root for B= 0.1 root= -0.71644 for initial guess 0
Root for B= 1 root= -0.588402 for initial guess 0
Root for B= 10 root= Max iters reached
-0.000745144 for initial guess 0
Root for B= 100 root= Max iters reached
0 for initial guess 0
jake@Jakes-MBP Exercises 4 %
```

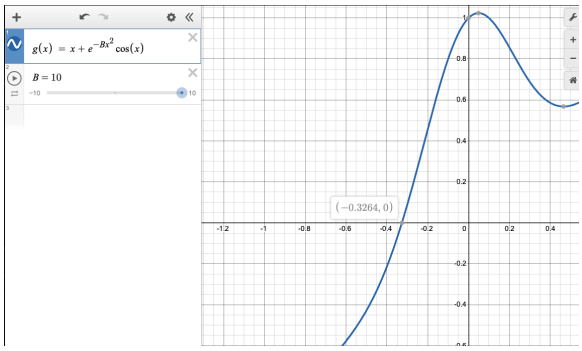
Figure 5: Output from Newton Method for $g(x)$ for initial guess of $x0 = 0$



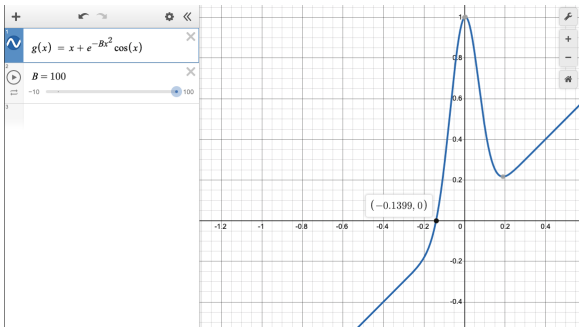

(a) $G(x)$ when $B = 0.1$. Root at $-0.7164$.




(b) $G(x)$ when $B = 1$. Root at $-0.5884$.




(c) $G(x)$ when $B = 10$. Root at $-0.3264$.




(d) $G(x)$ when $B = 100$. Root at $-0.1399$

Figure 6: Plots of $G(x)$ for $B = 0.1, 1, 10, 100$

# Problem 3

$$x_{i+1} = \mu x_i(1 - x_i)$$

1. To find the function which produces that iteration for newtons method we have to break down what is happening.

   $$x_{i+1} = \mu x_i - \mu x_i^2$$

   Newton's method is

   $$x_{i+1} = x_i - \frac{f(x)}{f'(x)}$$

   We want to find $f(x)$. Comparing equations we have

   $$x_i - z = \mu x_i - \mu x_i^2$$

   Where $z = \frac{f(x)}{f'(x)}$

   $$z = x_i - \mu x_i + \mu x_i^2$$

$$\frac{f(x)}{f'(x)} = x_i - \mu x_i + \mu x_i^2$$

$$\frac{d}{dx}\ln(f(x)) = \frac{f'(x)}{f(x)}$$

$$\int \frac{d}{dx}\ln(f(x)) = \int \frac{1}{x_i - \mu x_i + \mu x_i^2}$$

$$\ln(f(x)) = \int \frac{1}{x_i - \mu x_i + \mu x_i^2}$$

Using Mathematica to solve this integral and get rid of the log.

$$f(x) = x_i^{-\frac{1}{\mu-1}}(\mu x_i - \mu + 1)^{\frac{1}{\mu-1}}$$

Lets check and see if applying Newton's method gets us the original iteration. The derivative of $f(x)$ is

$$f'(x) = x_i^{-\frac{1}{\mu-1}-1}(\mu x_i - \mu + 1)^{\frac{1}{\mu-1}-1}$$

$$\frac{f(x)}{f'(x)} = x_i(x_i\mu - \mu + 1)$$

$$= \mu x_i^2 - \mu x_i + x_i$$

So the iteration is

$$x_{i+1} = x_i - (\mu x_i^2 - \mu x_i + x_i)$$

$$= \mu x_i^2 - \mu x_i = \mu x_i(1 - x_i)$$

2. Plotting the function $f(x)$. At $\mu = 3.0$ it seems as if half of the function is missing. What is happening here is that we are effectively trying to take the sqquare root of a negative number. Analytically this means there are no real roots for $\mu = 3.0$ (obviously there are imaginary ones), so by taking Newton's method we see the bifurcation behaviour caused by the iterations diverging instead of converging on the intended root. Effcitvely as we go past the point $\mu = 3.0$ the iterations ceases to work because of the lack of real roots it finds.
   To plot the figures below I created a simple c++ program to write values to a file and plotted via python matplotlib.

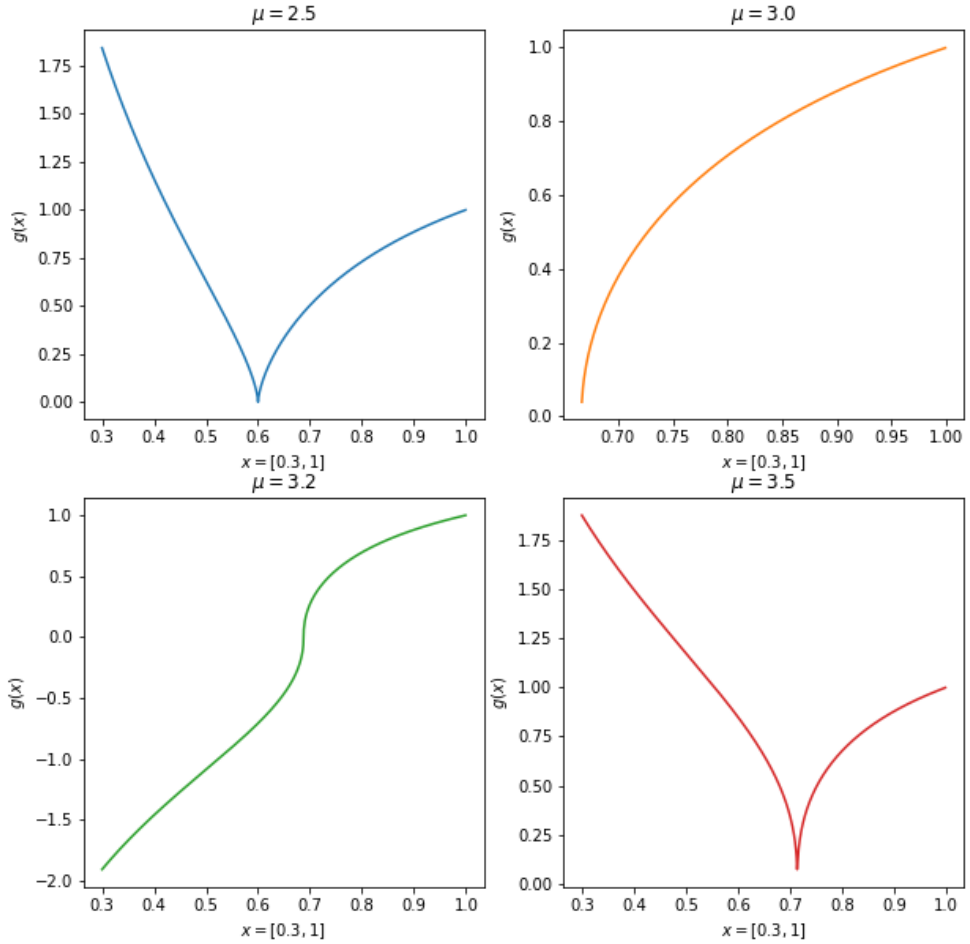



Figure 7: Plots of $f(x)$

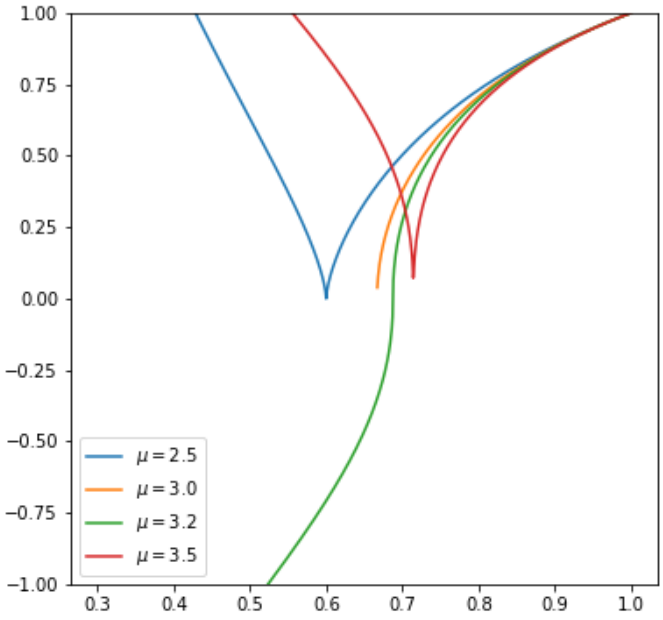



Figure 8: Plots of $f(x)$

# Problem 4

My solution to problem 4 is in the myroots.cpp file. The program uses the Eigen (`https://eigen.tuxfamily.org/`) Library to calculate eigenvalues using the ComplexEigenSolver. The program works by taking an input array $p$ containing the polynomial coefficients, i.e an array consisting of $p = \{3, -2, -4\}$ is equivalent to finding the roots to

$$3x^2 - 2x - 4 = 0$$

The function myroots works by taking input $N = p.length() - 1$, so if $p$ contains 5 values $N = 4$ and array $p$. The function that makes an NxN matrix and fills the values according to the table in the lecture notes, remembering to normalize the polynomial (i.e diving by the leading coefficient), for the example polynomial above this would be

$$C = \begin{pmatrix} -(-\frac{2}{3}) & -(-\frac{4}{3}) \\ 1 & 0 \end{pmatrix}$$

And computes the complex eigenvalues of the matrix $C$.

```
jake@Jakes-MBP Exercises 4 % ./myroots
Companion Matrix is
0.666667  1.33333
       1        0
Eigenvalues are
(-0.868517,0)
  (1.53518,0)
jake@Jakes-MBP Exercises 4 %
```

Figure 9: Complex Eigenvalues and roots of $3x^2 - 2x - 4 = 0$

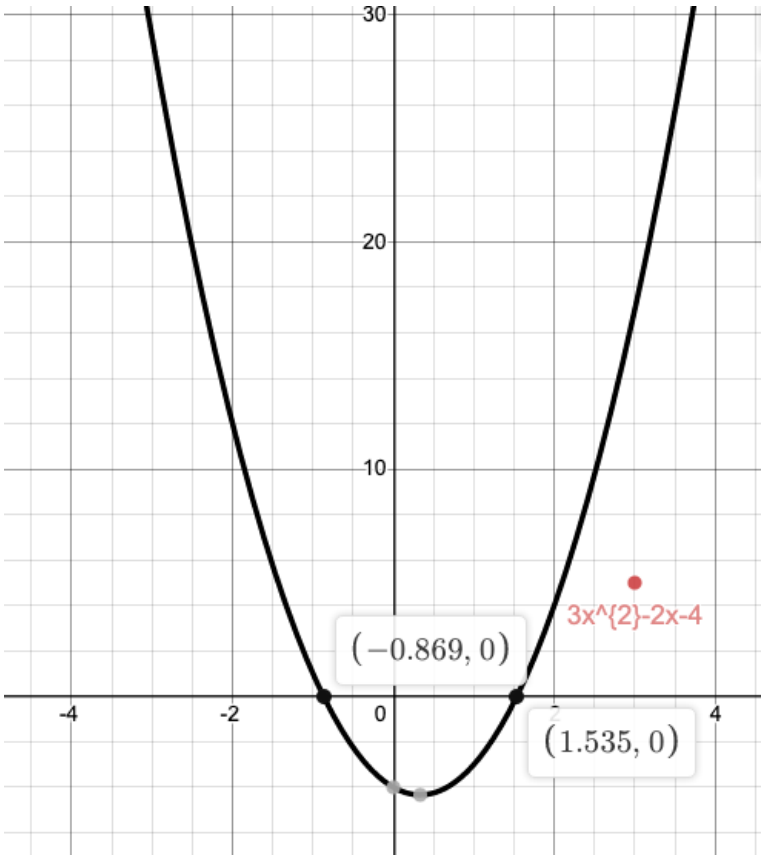



Figure 10: Plot of $3x^2 - 2x - 4 = 0$

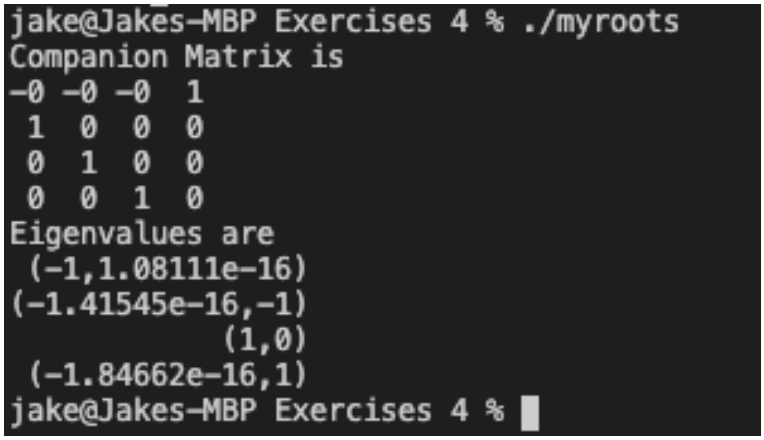



Figure 11: Complex Eigenvalues and roots of $x^4 - 1 = 0$. Notice that due to rounding errors it gives values that aren't 0 but are sufficiently close to 0. The analytical roots are equal to these