M 348 HOMEWORK 2

ZUN CAO

1. Chapter 0 (Modified code is at the end of the file)

Coding Exercise: Modify the code quadraticEquation (Matlab, Python, or C++ version) to compute complex roots of quadratic equation. Apply the usual quadratic formula when the roots are complex. Print the roots in the main routine (driver). Make sure your output indicates the three different cases: real root(s), complex roots, or error. Test your code on the following equations:

(a) $3x^2 + 5x = 4$
(b) $3x^2 - 7.8x + 5.07 = 0$
(c) $2x^2 + 4 = 3x$
(d) $8 = 3x$

Solution. (a) In this case, $a = 3, b = 5, c = -4$.
(b) In this case $a = 3, b = -7.8, c = 5.07$
(c) In this case $a = 2, b = -3, c = 4$

Date: 2/4/2024.

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\quadratic_modified_corrected.py
Solve ax^2 + bx + c = 0 for real or complex roots.
Enter a: 3
Enter b: 5
Enter c: -4
Real and distinct roots
Roots are 0.590667 and -2.257334

Process finished with exit code 0
```

(a) According to python code, we find that

$$r_1 = 0.590667, \quad r_2 = -2.257334$$

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\quadratic_modified_corrected.py
Solve ax^2 + bx + c = 0 for real or complex roots.
Enter a: 3
Enter b: -7.8
Enter c: 5.07
Complex roots
Roots are (1.3+1.4048949503631345e-08j) and (1.3-1.4048949503631345e-08j)

Process finished with exit code 0
```

(b1) Note that r is very close to 1.3. After adding a tolerance $1e^{-14}$, the answer become 1.3.

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\quadratic_modified_corrected.py
Solve ax^2 + bx + c = 0 for real or complex roots.
Enter a: 3
Enter b: -7.8
Enter c: 5.07
Real and equal roots
Roots are 1.3 and 1.3

Process finished with exit code 0
```

(b2) This is the result after adding tolerance $1e^{-14}$.

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\quadratic_modified_corrected.py
Solve ax^2 + bx + c = 0 for real or complex roots.
Enter a: 2
Enter b: -3
Enter c: 4
Complex roots
Roots are (0.75+1.1989578808281798j) and (0.75-1.1989578808281798j)

Process finished with exit code 0
```

(c) The results are complex numbers, which are:

$$r_1 = 0.75 + 1.1989578808281798j, \quad r_2 = 0.75 - 1.1989578808281798j$$

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\quadratic_modified_corrected.py
Solve ax^2 + bx + c = 0 for real or complex roots.
Enter a: 0
Enter b: -3
Enter c: 8
ERROR:Invalid inputs for a quadratic equation, such as a=0

Process finished with exit code 0
```

(d)ERROR happens since $a = 0$.

(d) In this case, $a = 0, b = -3, c = 8$

(I also upload these pictures and the code on canvas)

2. Chapter 1 1.2 Ex # 14.
Which of the following three Fixed-Point Iterations converge to $\sqrt{2}$? Rank the ones that converge from fastest to slowest.

(a) $x \to \frac{1}{2}x + \frac{1}{x}$
(b) $x \to \frac{2}{3}x + \frac{2}{3x}$
(c) $x \to \frac{3}{4}x + \frac{1}{2x}$

Also, show that $r = \sqrt{2}$ is a fixed point of each of the three functions.

Solution. To analyze which of the given Fixed-Point Iterations converge to $\sqrt{2}$ and to rank them from fastest to slowest, we verified each function at $r = \sqrt{2}$ to prove it as a fixed point, and then we examined the convergence criteria based on the derivative of each function at the fixed point $r = \sqrt{2}$.

Fixed Point Verification. A fixed point r of a function $g(x)$ satisfies $g(r) = r$. For $r = \sqrt{2}$, we verify each function as follows:

(a) $g_1(x) = \frac{1}{2}x + \frac{1}{x}$
(b) $g_2(x) = \frac{2}{3}x + \frac{2}{3x}$
(c) $g_3(x) = \frac{3}{4}x + \frac{1}{2x}$

Verification for $r = \sqrt{2}$.

- For $g_1(\sqrt{2})$:
$$g_1(\sqrt{2}) = \frac{1}{2}\sqrt{2} + \frac{1}{\sqrt{2}} = \sqrt{2}$$
- For $g_2(\sqrt{2})$:
$$g_2(\sqrt{2}) = \frac{2}{3}\sqrt{2} + \frac{2}{3\sqrt{2}} = \sqrt{2}$$
- For $g_3(\sqrt{2})$:
$$g_3(\sqrt{2}) = \frac{3}{4}\sqrt{2} + \frac{1}{2\sqrt{2}} = \sqrt{2}$$

We can see that all of the results are $\sqrt{2}$, indicating that $r = \sqrt{2}$ is a fixed point.

Convergence Analysis Results. The convergence of a fixed-point iteration is guaranteed if $|g'(r)| < 1$, where $g'(r)$ is the derivative of $g(x)$ evaluated at the fixed point r. The derivatives at $\sqrt{2}$ and their implications for convergence are as follows:

- For $g_1(x)$, the derivative is:
$$\frac{1}{2} - \frac{1}{x^2}$$
The derivative at $\sqrt{2}$ is 0, indicating strong convergence.

- For $g_2(x)$, the derivative is:
$$\frac{2}{3} - \frac{2}{3x^2}$$
The derivative at $\sqrt{2}$ is approximately $\frac{1}{3} \approx 0.333$, suggesting moderate convergence.
- For $g_3(x)$, the derivative is:
$$\frac{3}{4} - \frac{1}{2x^2}$$
The derivative at $\sqrt{2}$ is $\frac{1}{2} = 0.5$, indicating slower convergence than $g_2(x)$ but still convergent.

Ranking from Fastest to Slowest.

(a) $g_1(x) = \frac{1}{2}x + \frac{1}{x}$ converges the fastest because its derivative at $\sqrt{2}$ is 0.
(b) $g_2(x) = \frac{2}{3}x + \frac{2}{3x}$ is the second fastest with a derivative of approximately 0.333.
(c) $g_3(x) = \frac{3}{4}x + \frac{1}{2x}$ is the slowest among the three, with a derivative of 0.5.

3. Additional Problem 1

Consider the equation $3x^5 - 2x = 5x^3 - 1$

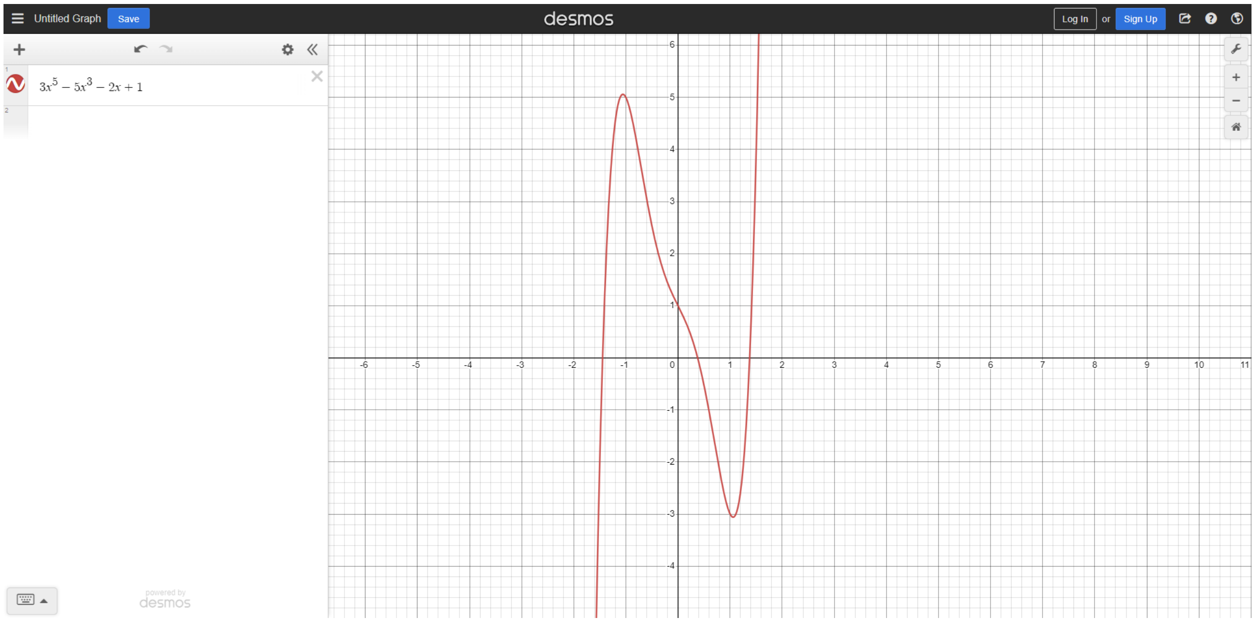
(a) How many real roots does it have? For each root, find an interval $[a, b]$ of length one that brackets it.
(b) Starting with $[a, b]$, how many steps of the Bisection Method are required to calculate the solution to ten correct decimal places? Answer with an integer. Note: You have to solve this problem by hand but you can verify (for yourself) the answer using the code bisection.m Download bisection.m (or bisection.cpp Download bisection.cppor bisection.py Download bisection.py). (Running the algorithm, manually or with the code, is NOT required and will NOT count as a solution!)
(c) Apply two steps of the method in (b) for one of the roots.

Solution. (a) We will use the Bisection Method to determine how many real roots this equation have. First, rearrange this equation:

$$3x^5 - 5x^3 - 2x + 1 = 0$$

Initial investigation and graphical analysis suggested the presence of **three** real roots. We identified the following intervals that potentially contain these roots:

- Near -1.5, we considered the interval $[-2, -1]$.
- Near 0.35, we considered the interval $[0, 1]$.
- Near 1.35, we considered the interval $[1, 2]$.



By plotting, we can roughly identify three intervals where sign changes.

0.1. **Initial Setup.** Let the initial interval be $[a, b] = [0, 1]$ with $f(a) = f(0)$ and $f(b) = f(1)$.

0.2. **Iteration Process.**

- **Iteration 1:** Compute $c_1 = \frac{0+1}{2} = 0.5$, and evaluate $f(c_1)$. If $f(0) \cdot f(c_1) < 0$, the root lies in $[0, c_1]$, else in $[c_1, 1]$.
- **Iteration 2:** Based on the sign of $f(c_1)$, select the new interval and compute c_2, the midpoint of the new interval. Evaluate $f(c_2)$ and determine the subinterval containing the root.
- Repeat this process, each time halving the interval, until the interval width is less than the tolerance, say 10^{-10}.

(b) To determine the number of steps required by the Bisection Method to achieve a precision of ten correct decimal places, we use the formula for the error after n steps:

$$\text{Error after } n \text{ steps} = \frac{b-a}{2^n}$$

where $b - a$ is the length of the initial interval, and n is the number of steps. To achieve an accuracy of ten decimal places, we require that the error be less than 10^{-10}. Assuming an initial interval of length 1, we have:

$$\frac{1}{2^n} < 10^{-10}$$

Rearranging and solving for n:

$$2^n > 10^{10}$$

Taking the logarithm of both sides gives us:

$$n \log(2) > 10 \log(10)$$

$$n > \frac{10 \log(10)}{\log(2)}$$

Calculating the value:

$$n > 33.219280948873624$$

Since n must be an integer, we round up to the nearest whole number:

$$n = 34$$

Thus, **34 steps** of the Bisection Method are required to calculate the solution to ten correct decimal places.

(c) We will apply two steps of the Bisection Method to the interval $[0, 1]$, aiming to find a root of the equation $3x^5 - 5x^3 - 2x + 1 = 0$.

Initial Interval: $[a, b] = [0, 1]$

Step 1:

- Calculate the midpoint: $c_1 = \frac{0+1}{2} = 0.5$
- Evaluate $f(c_1)$. We found that $f(0) \cdot f(0.5) = -0.531 < 0$, then the root lies in $[0, 0.5]$.

Step 2:

- Assuming the root lies in $[0, 0.5]$, we calculate the new midpoint: $c_2 = \frac{0+0.5}{2} = 0.25$.
- Evaluate $f(c_2)$. We found that $f(0) \cdot f(0.25) = 0.425 > 0$, then the root lies in $[0.25, 0.5]$.

Through these steps, we iteratively narrow down the interval containing the root, halving the interval length at each step.

4. Additional Problem 2

The following equation $1 - 6x^3 = e^{2x} - 5x$ has three roots.

(a) Apply Fixed-Point Iteration (use fixedPoint_err.m Download fixedPoint_err.m) to find each of the roots to 6 correct decimal places. Report:

- the function for which you applied the Fixed Point Iteration (different functions might be needed for the different roots!); See below notes on finding a true solution.
- the initial guess and tolerance used;
- the solution with 6 correct decimal places;
- the sequence of iterates x_i, the error e_i, and the error ratio e_i/e_{i-1}. (Modify the code to calculate and print this ratio for each iteration.)

(b) Calculate the theoretical rate of convergence S. Confirm that the error ratios in (a) are close to S.

Solution. (a) The function for which I applied the FPI is (I think might likely to converge)

$$f(x) = \sqrt[3]{\frac{5x + 1 - e^{2x}}{6}}$$

$$x = \sqrt[3]{\frac{5x + 1 - e^{2x}}{6}}$$

Function2(used to solve when xtrue=0):

$$x = \frac{e^{2x} + 6x^3 - 1}{5}$$

```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\modified_fpi_with_error_ratio.py
Solve the problem g(x)=x using fixed point iteration
Enter guess at root: -1
Enter tolerance: 1e-6
Enter maxIteration: 100
Monitor iterations? (1/0): 1
Iter 0: x= -1.000000, error = 0.116676
Iter 1: x= -0.883324, error = 0.116676, error ratio = inf
Iter 2: x= -0.842457, error = 0.040866, error ratio = 0.350255
Iter 3: x= -0.827332, error = 0.015125, error ratio = 0.370108
Iter 4: x= -0.821617, error = 0.005715, error ratio = 0.377863
Iter 5: x= -0.819440, error = 0.002177, error ratio = 0.380854
Iter 6: x= -0.818609, error = 0.000831, error ratio = 0.382002
Iter 7: x= -0.818291, error = 0.000318, error ratio = 0.382442
Iter 8: x= -0.818169, error = 0.000122, error ratio = 0.382610
Iter 9: x= -0.818123, error = 0.000047, error ratio = 0.382674
Iter 10: x= -0.818105, error = 0.000018, error ratio = 0.382699
Iter 11: x= -0.818098, error = 0.000007, error ratio = 0.382709
Iter 12: x= -0.818095, error = 0.000003, error ratio = 0.382712
Iter 13: x= -0.818094, error = 0.000001, error ratio = 0.382714
The root is -0.8180943537281646
The number of iterations is 13
errors = [1.16676417e-01 1.16676417e-01 4.08664478e-02 1.51250129e-02
 5.71517626e-03 2.17664503e-03 8.31481989e-04 3.17993267e-04
 1.21667401e-04 4.65590078e-05 1.78180923e-05 6.81913690e-06
 2.60976689e-06 9.98793237e-07]
error ratios = [       inf 0.35025457 0.37010833 0.37786257 0.38085353 0.38200165
 0.38244156 0.38260999 0.38267447 0.38269914 0.38270859 0.3827122
 0.38271358]

Process finished with exit code 0
```

The initial guess $= -1$, tolerance $= 1e^{-6}$, solution $= -0.818094$

The other is printed in the picture

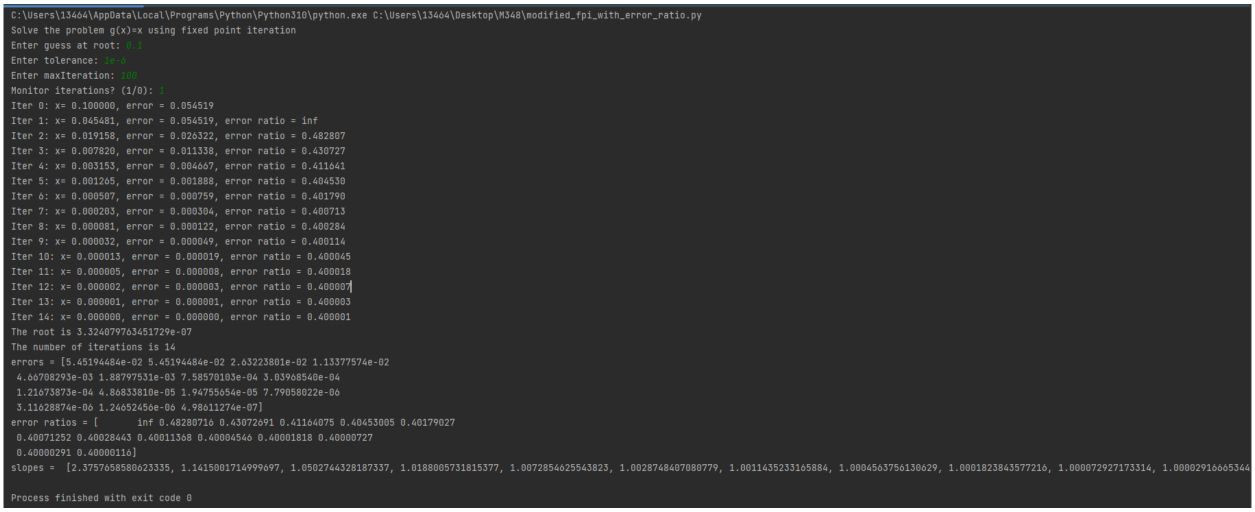
```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\modified_fpi_with_error_ratio.py
Solve the problem g(x)=x using fixed point iteration
Enter guess at root: 0.5
Enter tolerance: 1e-6
Enter maxIteration: 100
Monitor iterations? (1/0): 1
Iter 0: x= 0.500000, error = 0.006951
Iter 1: x= 0.506951, error = 0.006951, error ratio = inf
Iter 2: x= 0.506237, error = 0.000714, error ratio = 0.102724
Iter 3: x= 0.506316, error = 0.000079, error ratio = 0.110268
Iter 4: x= 0.506307, error = 0.000009, error ratio = 0.109494
Iter 5: x= 0.506308, error = 0.000001, error ratio = 0.109579
The root is 0.506308379638491
The number of iterations is 5
errors = [6.95139102e-03 6.95139102e-03 7.14073829e-04 7.87391789e-05
 8.62146349e-06 9.44733640e-07]
error ratios = [       inf 0.10272388 0.11026756 0.10949395 0.10957927]

Process finished with exit code 0
```

The initial guess $= 0.5$, tolerance $= 1e^{-6}$, solution $= 0.506308$

The other is printed in the picture

Note that there's still another solution with solution $= 0.000000$. But this requires less than 2 iterations, so it is solved by another function.



```
C:\Users\13464\AppData\Local\Programs\Python\Python310\python.exe C:\Users\13464\Desktop\M348\modified_fpi_with_error_ratio.py
Solve the problem g(x)=x using fixed point iteration
Enter guess at root: 0.1
Enter tolerance: 1e-6
Enter maxIteration: 100
Monitor iterations? (1/0): 1
Iter 0: x= 0.100000, error = 0.054519
Iter 1: x= 0.045481, error = 0.054519, error ratio = inf
Iter 2: x= 0.019158, error = 0.026322, error ratio = 0.482807
Iter 3: x= 0.007820, error = 0.011338, error ratio = 0.430727
Iter 4: x= 0.003153, error = 0.004667, error ratio = 0.411641
Iter 5: x= 0.001265, error = 0.001888, error ratio = 0.404530
Iter 6: x= 0.000507, error = 0.000759, error ratio = 0.401790
Iter 7: x= 0.000203, error = 0.000304, error ratio = 0.400713
Iter 8: x= 0.000081, error = 0.000122, error ratio = 0.400284
Iter 9: x= 0.000032, error = 0.000049, error ratio = 0.400114
Iter 10: x= 0.000013, error = 0.000019, error ratio = 0.400045
Iter 11: x= 0.000005, error = 0.000008, error ratio = 0.400018
Iter 12: x= 0.000002, error = 0.000003, error ratio = 0.400007
Iter 13: x= 0.000001, error = 0.000001, error ratio = 0.400003
Iter 14: x= 0.000000, error = 0.000000, error ratio = 0.400001
The root is 3.324079763451729e-07
The number of iterations is 14
errors = [5.45194484e-02 5.45194484e-02 2.63223801e-02 1.13377574e-02
 4.66708293e-03 1.88797531e-03 7.58570103e-04 3.03968540e-04
 1.21673873e-04 4.86833810e-05 1.94755654e-05 7.79058022e-06
 3.11628874e-06 1.24652456e-06 4.98611274e-07]
error ratios = [       inf 0.48280716 0.43072691 0.41164075 0.40453005 0.40179027
 0.40071252 0.40028443 0.40011368 0.40004546 0.40001818 0.40000727
 0.40000291 0.40000116]
slopes =  [2.3757658580623335, 1.1415001714999697, 1.0502744328187337, 1.0188005731815377, 1.0072854625543823, 1.0028748407080779, 1.0011435233165884, 1.0004563756130629, 1.0001823843577216, 1.000072927173314, 1.00002916665344

Process finished with exit code 0
```

The initial guess $= 0.1$, tolerance $= 1e^{-6}$, solution $= 0.000000$
The other is printed in the picture

(b) The derivative of the function $g(x)$ is calculated as:

$$g'(x) = \frac{5 - 2e^{2x}}{3\sqrt[3]{6}(-e^{2x} + 5x + 1)^{\frac{2}{3}}}$$

Evaluating $g'(x)$ at the reported roots gives us:

- For $x_1 = -0.818094$, $g'(x_1) \approx 0.382714$, which is close to the error ratio calculated in picture1, 0.038271358.
- For $x_2 = 0.506308$, $g'(x_3) \approx -0.109570$. This suggests a linear convergence rate for iterations close to this root, with $S \approx 0.109570$, which is close to the error ratio calculated in picture2, 0.10957927.

The derivative of the another function $g_2(x)$ is calculated as:

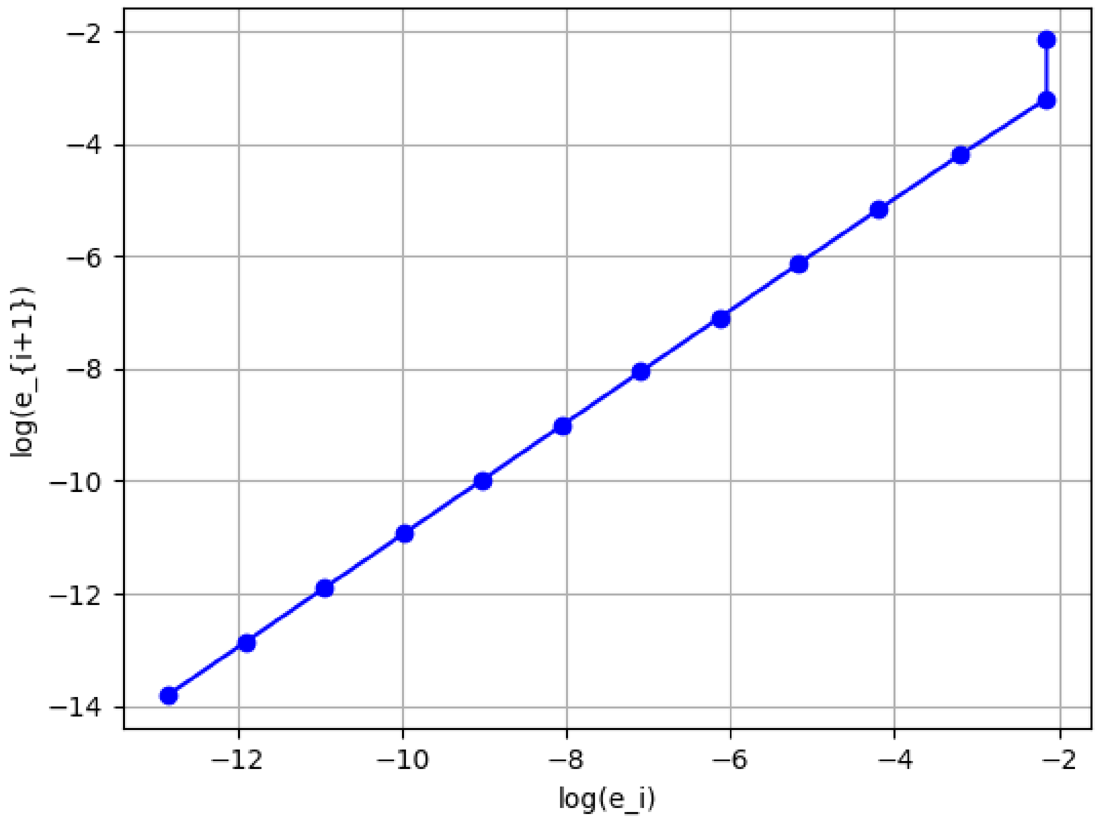
$$g'(x) = \frac{2e^{2x} + 18x^2}{5}$$

- For $x_3 = 0.000000$, $g'(x_3) \approx 0.400000$, which is close to the error ratio calculated in picture3, 0.40000116.

The rates of convergence that are less than 1, indicative of linear convergence for those iterations that are converging to real roots.

5. Additional Problem 3

It is known that for a general iterative method, the error (or error estimate) has the form $e_{i+1} = Ce_i^{\alpha}$. To confirm numerically the order of convergence (α), we can take log of both sides to get $\log(e_{i+1}) = \log(C) + \alpha \log(e_i)$, i.e. plotting $\log(e_{i+1})$ vs $\log(e_i)$ should produce points forming roughly a line with a slope α.

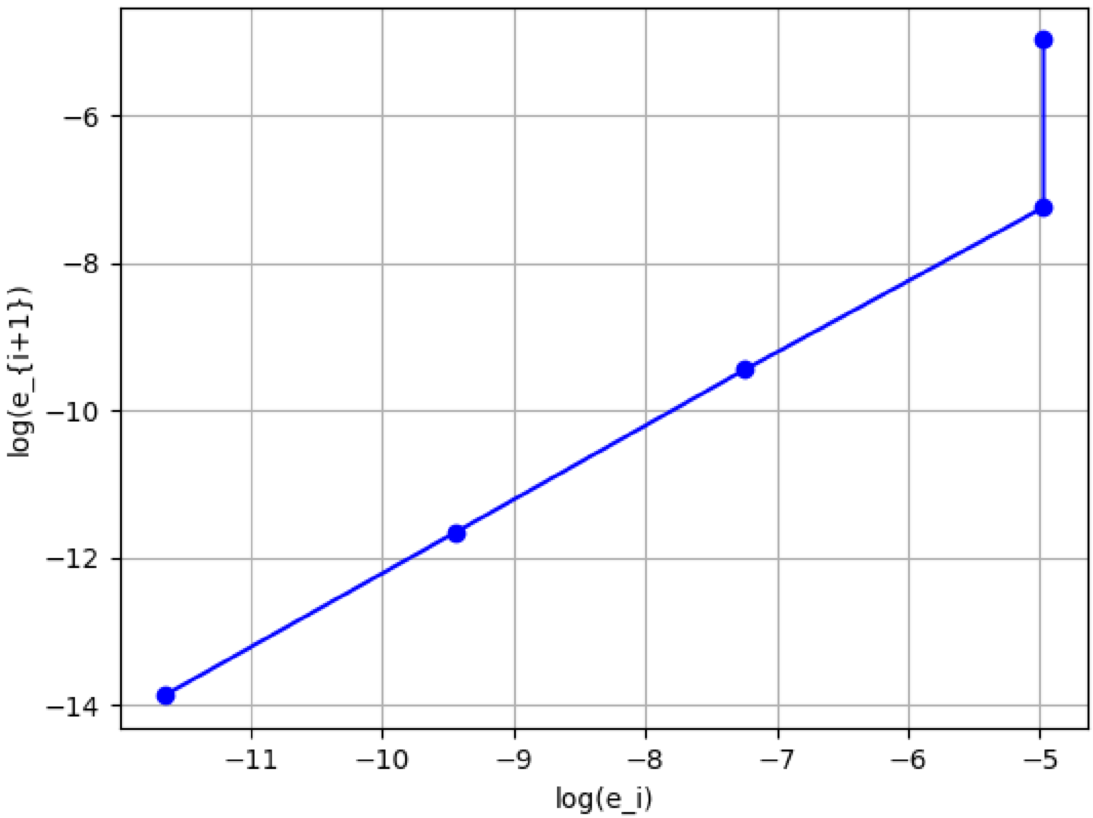


The plot of root1

Report these slopes and graphs for the roots in Additional Problem 2. Since FPI is used, the expected order (i.e. slopes) should be around 1.

Solution. From the two figures, we can conclude that the slope is close to 1.
(All the following input and output can be found in the doc file called numerical results in zip file in canvas)
Slope for root1:
slopes = [-inf, 0.9474446721125698, 0.9791391897341561, 0.9918987946762178, 0.9968818542200617, 0.9988040199447701, 0.9995418957005836, 0.9998246203166842, 0.9999328713754053, 0.9999743076973548, 0.9999901670578712, 0.9999962366568056]

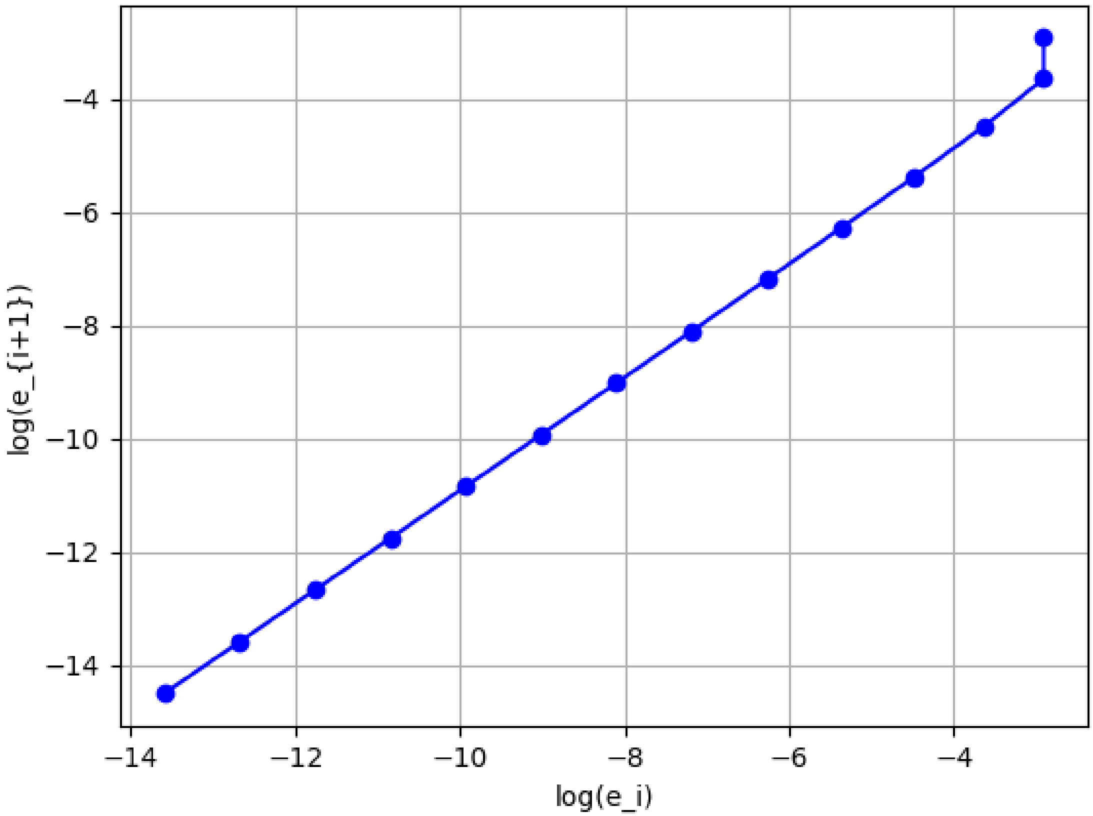
Slope for root2:



The plot of root2

Slope for root3:

slopes = [-inf, 1.1567602107312414, 1.0538101466686016, 1.0196314560348703, 1.007508937791567, 1.0029456931069607, 1.0011688236625746, 1.000466004403507, 1.000186156950749, 1.0000744235516281, 1.0000297631485222, 1.0000119042550049, 1.0000047616094374]



The plot of root3

```
errors = [1.16676417e-01 1.16676417e-01 4.08664478e-02 1.51250129e-02
 5.71517626e-03 2.17664503e-03 8.31481989e-04 3.17993267e-04
 1.21667401e-04 4.65590078e-05 1.78180923e-05 6.81913690e-06
 2.60976689e-06 9.98793237e-07]
error ratios = [       inf 0.35025457 0.37010833 0.37786257 0.38085353 0.38200165
 0.38244156 0.38260999 0.38267447 0.38269914 0.38270859 0.3827122
 0.38271358]
slopes =  [1.8751248806828147, 0.9488562708646957, 0.9799136964908889, 0.9922337582647088, 0.9970159591136125, 0.9988562247470724, 0.9995620051329573, 0.9998323355681015, 0.9999358269411005, 0.9999754392389176]

Process finished with exit code 0
```

The slope of root1 (old verson before the code change (2 to 1))

```
errors = [6.95139102e-03 6.95139102e-03 7.14073829e-04 7.87391789e-05
 8.62146349e-06 9.44733640e-07]
error ratios = [       inf 0.10272388 0.11026756 0.10949395 0.10957927]
slopes =  [1.9408141373204857, 0.9871526892766216]

Process finished with exit code 0
```

The slope of root2 (old verson before the code change (2 to 1))

```
00291 0.40000116]
 =  [2.3757658580623335, 1.1415001714999697, 1.0502744328187337, 1.0188005731815377, 1.0072854625543823, 1.0028748407080779, 1.0011435233165884, 1.0004563756130629, 1.0001823843577216, 1.000072927173314, 1.0000291666534498]

s finished with exit code 0
```

The slope of root3 (old verson before the code change (2 to 1))

Modified code of Coding exercise 1 quadratic

```
#!/usr/bin/env python
"""
Solve ax^2 + bx + c = 0 for real or complex roots

return 0 in no error, 1 otherwise
"""

import argparse
from math import sqrt, fabs
from cmath import sqrt as csqrt

parser = argparse.ArgumentParser(description="Solve ax^2 + bx + c =0 for real or complex
parser.add_argument("-d","--debug",action="store_true")
DEBUG = parser.parse_args().debug
r1 = None
r2 = None
############################## FUNCTIONS ##############################

def quadraticFormula(a,b,c):
    global r1, r2
    discriminant = b*b - 4*a*c
    eps = 1e-20

    # Define a tolerance for considering the discriminant effectively zero
    tolerance = 1e-14

    if -tolerance < discriminant < tolerance:
        # Treat the discriminant as zero
        discriminant = 0

    if DEBUG:
        print("a = %.20f" % a)
        print("b = %.20f" % b)
        print("c = %.20f" % c)
        print("D = %.20f" % discriminant)
```

```
    if(discriminant < 0 and fabs(discriminant) < eps):
        print("|abs(D)| =  %.6e; Setting D to 0" % fabs(discriminant))
        discriminant = 0

    if fabs(a) < eps:
        return 1

    if discriminant > 0:
        r1 = (-b + sqrt(discriminant)) / (2*a)
        r2 = (-b - sqrt(discriminant)) / (2*a)
        print("Real and distinct roots")
    elif discriminant == 0:
        r1 = r2 = -b / (2*a)
        print("Real and equal roots")
    else:
        r1 = (-b + csqrt(discriminant)) / (2*a)
        r2 = (-b - csqrt(discriminant)) / (2*a)
        print("Complex roots")
    return 0

############################## MAIN #############################

print("Solve ax^2 + bx + c = 0 for real or complex roots.")
a = float(input("Enter a: "))
b = float(input("Enter b: "))
c = float(input("Enter c: "))

error = quadraticFormula(a,b,c)


if error:
    print("ERROR:Invalid inputs for a quadratic equation, such as a=0")
else:
    print("Roots are {} and {}".format(r1, r2))
```

Modified code of Additional Problem 2 FPI

```
#!/usr/bin/env python3
'''
FIXED POINT (PICARD) ITERATION METHOD

Solves the problem g(x) = x using fixed point iteration.

The main function is fpi:

[state, x, errors, iter, error_ratios] = fpi(g, x0, tolerance, maxIteration, debug);

Inputs:
  g               Handle to function g
  x0              The initial guess at the fixed point
  tolerance       The convergence tolerance (must be > 0).
  maxIteration    The maximum number of iterations that can be taken.
  debug           Boolean for printing out information on every iteration.
Outputs:
  x               The solution
  errors          Array with errors at each iteration
  iter            number of iterations to convergence
  error_ratios    Array with error ratios e_i/e_{i-1} for each iteration
Return:
  state           An error status code.
    SUCCESS       Successful termination.
    WONT_STOP     Error: Exceeded maximum number of iterations.
'''
import numpy as np

SUCCESS = 0
WONT_STOP = 1


# Define g(x) based on the given equation 1 - 6x^3 = e^{2x} - 5x
# This needs to be adjusted to an appropriate form for fixed point iteration
def g(x):
```

```
    # Placeholder function, adjust according to your derivation for the equation
    return np.cbrt((-np.exp(2 * x) + 5 * x + 1) / 6)
    # (np.exp(2*x)-1+6*x**3)/5 this is function2

def fpi(func, x0, TOL, MAX_ITERS, debug):
    errors = np.zeros(MAX_ITERS + 1)
    error_ratios = np.zeros(MAX_ITERS)  # To store error ratios
    x = x0
    errors[0] = np.abs(func(x0) - x0)  # Initial error

    if debug:
        print(f"Iter 0: x= {x:.6f}, error = {errors[0]:.6f}")

    for itn in range(1, MAX_ITERS + 1):
        gx = func(x)
        err = np.abs(gx - x)
        errors[itn] = err
        error_ratios[itn - 1] = errors[itn] / errors[itn - 1] if itn > 1 else np.inf

        if debug:
            print(f"Iter {itn}: x= {gx:.6f}, error = {err:.6f}, error ratio = {error_rat

        if err <= TOL:
            return SUCCESS, gx, errors[:itn + 1], itn, error_ratios[:itn]

        x = gx

    return WONT_STOP, x, errors[:itn + 1], itn, error_ratios[:itn]


### MAIN
print("Solve the problem g(x)=x using fixed point iteration")
x0 = float(input("Enter guess at root: "))
tol = float(input("Enter tolerance: "))
maxIter = int(input("Enter maxIteration: "))
debug = input("Monitor iterations? (1/0): ") == '1'
```

```
state, x, errors, iters, error_ratios = fpi(g, x0, tol, maxIter, debug)
if state == SUCCESS:
    print(f"The root is {x:.16g}")
    print(f"The number of iterations is {iters}")
else:
    print(f"ERROR: Failed to converge in {maxIter} iterations!")

# Optionally, print errors and error ratios if needed
print("errors =", errors)
print("error ratios =", error_ratios)
```

```
#additional problem 3
from numpy import log
import matplotlib.pyplot as pyp
x = log(errors[:-1])
y = log(errors[1:])
dx = x[1:]-x[:-1]
dy = y[1:]-y[:-1]
slopes = [dy[i]/dx[i] for i in range(len(dx))]
print("slopes = ",slopes)
pyp.plot(x,y,"bo-")
pyp.xlabel("log(e_i)")
pyp.ylabel("log(e_{i+1})")
pyp.grid(True)
# This saves to a file
pyp.savefig("./LogErrorsPlot.png")
# This shows it on your screen
pyp.show()
```