HW7 Aaryan Bhagat 862468325

①

(a) It depends upon $\left|\frac{\lambda_2}{\lambda_1}\right|$ where λ_2 is second largest eigenvalue.

(a) If ratio approaches 1 then rate is small

(b) To improve the power iteration we can use:

- Rayleigh Quotient to adaptively choose the shift param
- Deflating a matrix by subtracting the outer product of the dominant eigenvector from the matrix

② Given

$A \to (n,n)$ Real Matrix

$\text{Rank}(A) = 1 \to$ ①

(a) To Prove

$A = uv^T \to$ ⓥ

$u, v \in \mathbb{R}^n$

From ① it implies that each column of A can be represented as some basis

$\therefore c_i = \alpha_i \beta_1$ $\left\{ \begin{array}{l} \beta_1 \text{ is the basis } \in \mathbb{R}^n \\ \alpha_i \in \mathbb{R} \quad c_i \to i^{th} \text{ column of } A \\ \quad i \in \{1, 2 \ldots n\} \end{array} \right\}$

$\Rightarrow \begin{bmatrix} c_1, c_2 - c_n \end{bmatrix} = \begin{bmatrix} \beta_1 \end{bmatrix} \begin{bmatrix} \alpha_1, \alpha_2 - \alpha_n \end{bmatrix}$

$\Rightarrow A = uv^T$

where $u \to \beta_1$ and $v \to \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}$ $\alpha_i \in \mathbb{R}$

(ii) (iii)

(b) To Prove

$u^T v$ is an eigenvalue of $A \to$ (IV)

$u^T v = \beta_1^T \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}$ {From (II) and (III)}

If (IV) is true then

$Ax = u^T v x$ for some $x \in \mathbb{R}^n$

$uv^T x = u^T v x$ [From (V)]

Assume $x = u$

$\Rightarrow \underbrace{uv^T u}_{\text{Scalar}} = u^T v u$

$\Rightarrow (v^T u) u = (u^T v) u$

$\Rightarrow (u^T v) u = (u^T v) u$ [Since $a^T b = b^T a$]

Hence proved

(c) To Find

No of iterations required to converge

Given one eigenvalue $u^T v$, null(A) = $n-1$

Rank(A) = 1

Assume $d_1, \ldots d_{n-1}$ are basis for null(v)

we know Rank(v) = 1

$uv^T d_i = 0 \quad \forall i \in \{1 \ldots n-1\}$

Hence we have eigenvalues 0 for $n-1$ times

∴ Convergence Rate is $\frac{0}{u^T v}$ → very fast

$\Rightarrow y_1 = Ax_0 / (\|Ax_0\|_2)$

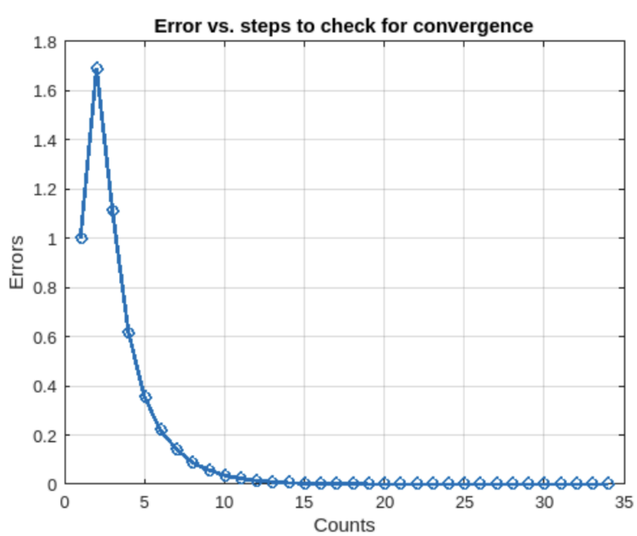
$y_1 = \frac{uv^T x_0}{\|Ax_0\|_2}$

This converges in one iteration if x_0 is taken to be cu where $c \in \mathbb{R}$

Q3

(a)

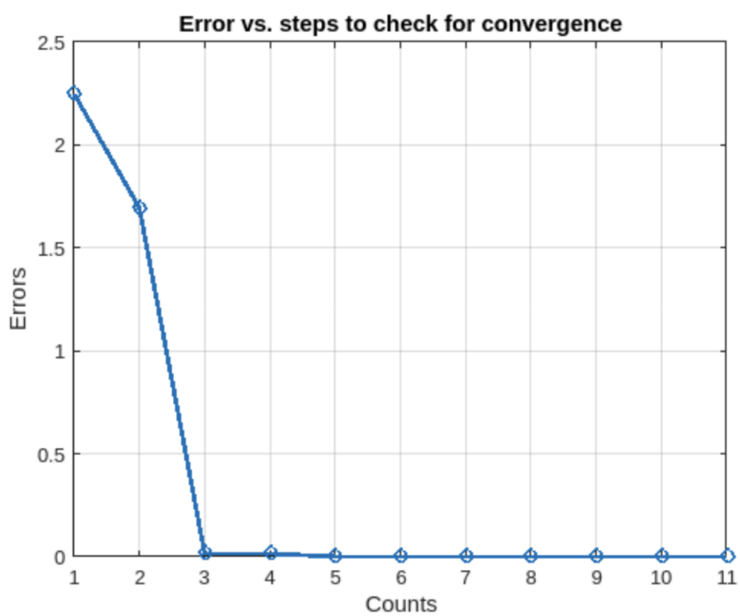
```
A = [3.5, 2, -1; 1, 2.5, 0; 1, 2, -3.5];
b = [1; 2; 3];
[m, n] = size(A);
d = diag(diag(A));
x = zeros(m, 1);
err = inf;
errs = [];
count = 0;
counts = [];
tolerance = 1e-6;
while err > tolerance
   count = count + 1;
   counts = [counts, count];
   dx = d\(b - A*x);
   x = x + dx;
   err = max(abs(dx./x));
   errs = [errs, err];
end
f = figure;
p = plot(counts, errs, '-o');
p(1).LineWidth = 2;
xlabel('Counts');  % x-axis label
ylabel('Errors');  % y-axis label
title('Error vs. steps to check for convergence');
grid on;
x =

   -0.3784
    0.9514
   -0.4216
>> A\b

ans =

   -0.3784
    0.9514
   -0.4216
>>
```



(b)

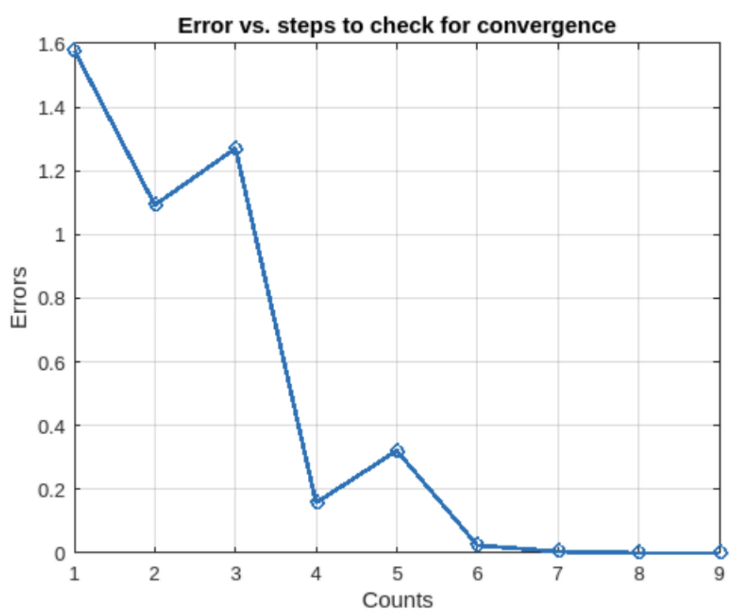
```
rng(12)
```

```
A = rand(4, 4);
A = A + A';
x0 = rand(4, 1);
m=0;
n=length(x0);
y_final=x0;
tol=1e-6;
count = 0;
counts = [];
errs = [];
while(1)
   count = count + 1;
   counts = [counts, count];
   mold = m;
   y_old=y_final;
   y_final=A*y_final;
   m=max(y_final);
   y_final=y_final/m;
   errs = [errs, abs(m-mold)];
   if abs(m-mold) < tol && norm(y_final-y_old,2) < tol
       break;
   end
end
f = figure;
p = plot(counts, errs, '-o');
p(1).LineWidth = 2;
xlabel('Counts');  % x-axis label
ylabel('Errors');  % y-axis label
title('Error vs. steps to check for convergence');
grid on;
>> y_final

y_final =

    0.7999
    1.0000
    0.7444
    0.8596
```



(c)

```
rng(12);
A = rand(4, 4);
x0 = rand(4, 1);
tolerance = 1e-6;
x0 = x0 / norm(x0);
l_old = 0;
x_old = x0;
count = 0;
counts = [];
errs = [];
while true
   count = count + 1;
   counts = [counts, count];
   l = (x_old' * A * x_old) / (x_old' * x_old);
   errs = [errs, abs(l - l_old)];
   if abs(l - l_old) < tolerance
       break;
   end
   x_old = (A - l_old * eye(size(A))) \ x_old;
   x_old = x_old / norm(x_old);
   l_old = l;
end
eigenvector = x_old;
f = figure;
p = plot(counts, errs, '-o');
p(1).LineWidth = 2;
xlabel('Counts');  % x-axis label
ylabel('Errors');  % y-axis label
title('Error vs. steps to check for convergence');
grid on;
>> eigenvector

eigenvector =

   -0.4394
   -0.6872
   -0.4439
   -0.3709
```



4 Given

$$f(x) = x^2 - 2 = 0$$

(a) Starting Point

$$x_0 = 1 \rightarrow (1)$$

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

$n = 0$

$$x_1 = x_0 - \frac{x_0^2 - 2}{2x_0}$$

$$x_1 = 1 - \frac{1^2 - 2}{2 \times 1}$$ [From (1)]

$$x_1 = 1 + \frac{1}{2}$$

$$\Rightarrow 3/2$$

(b) Starting Points

$x_0 = 1 \quad x_1 = 2$

Secant Method

$$x_2 = x_0 - f(x_0) \times \frac{(x_1 - x_0)}{f(x_1) - f(x_0)}$$

$$x_2 = 1 - (1^2 - 2) \times \frac{(2-1)}{(2^2-2) - (1^2-2)}$$

$$x_2 = 1 - (-1) \times \frac{1}{2+1}$$

$$x_2 = 4/3$$

(5) $x_{k+1} = x_k - \frac{f(x)}{d}$ } → Given
$d \to$ constat }
Equation is $f(x)=0$

(a) <u>To find local convergence</u>

⇒ <u>Assume $[a,b]$ is the required interval</u>
<u>Then condition will be</u>

$$\left| \frac{d\left(x^* - \frac{f(x^*)}{d}\right)}{d(x)} \right| < 1 \quad \forall x \in [a,b]$$

$$\left| 1 - \frac{f'(x^*)}{d} \right| < 1 \qquad \left\{ \begin{array}{l} x^* \text{ is true solution} \\ x^* \text{ lies in } [a,b] \end{array} \right\}$$

<u>local convergence should satisfy</u>

$|x_{k+1} - x^*| \le q|x_k - x^*|$ {q is constat}
{x^* is true solution}
↓

(b) Convergence Rate

Error Rate

$e_{k+1} = x_{k+1} - x^*$

Also

$e_{k+1} = e_k - f(x_k)\,d \rightarrow$ (11)

Ideally convergence rate is

(111) $\Leftarrow |e_{k+1}| \leq Q|e_k|^2$ {Q is constant}

For (111) to hold in the new scheme

$$|e_k - f(x_k)\,d| \leq Q|e_k|^2$$

$|e_k| - |f(x_k)\,d| \leq Q|e_k|^2$ $\{|a| - |b| \leq |a-b|\}$

As $|e_k|$ converges to 0 for the inequality to hold

$$|f(x_k)\,d| \approx 0$$

So the actual convergence rate will depend upon the function chosen

(c) To find value of d which will still yield quadratic convergence

Since original is of the form

$$\epsilon_{k+1} = \frac{\epsilon_k^2}{2} \frac{f''(x)}{f'(x)}$$

Hence $d \approx \frac{1}{f'(x^*)}$ where x^* is the actual root

provided $f'(x^*) \neq 0$

Q6

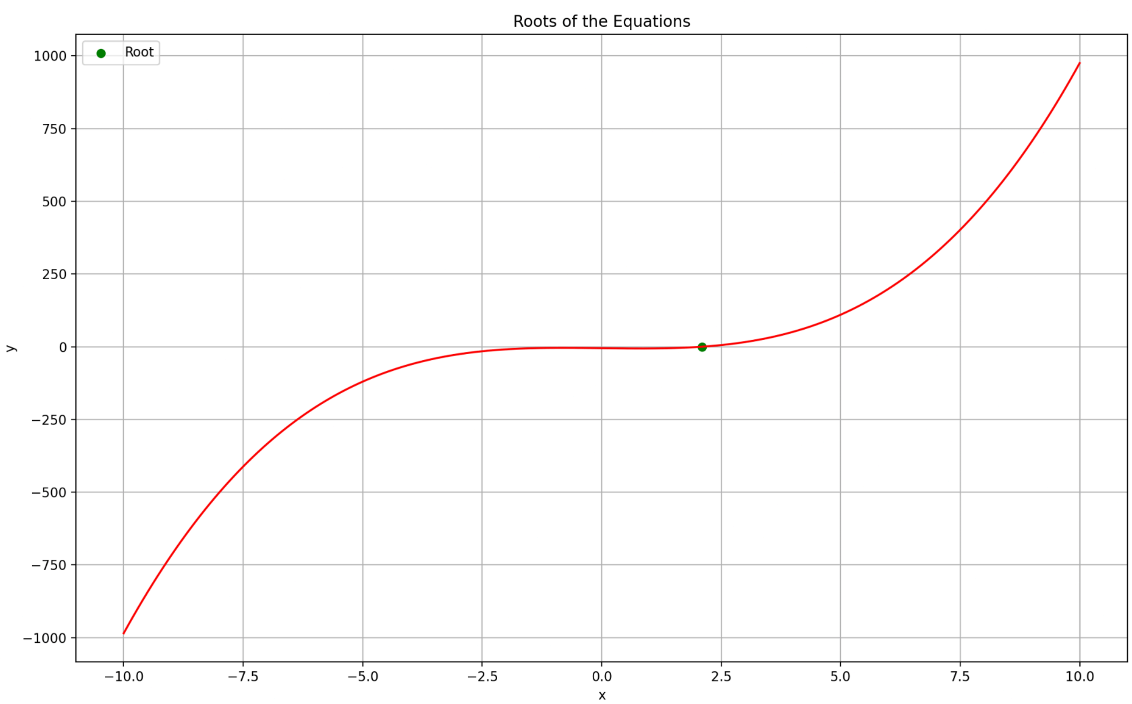
Newton

Threshold is 1e-6

(a)

```
import numpy as np
import matplotlib.pyplot as plt


def newton(eq, derivatives, start_point, threshold=1e-6, max_steps=1000):
   x = start_point.copy()
   delta = eq(*x) / derivatives(*x)
   for i in range(max_steps):
       delta = eq(*x) / derivatives(*x)
       x = x - delta

       if abs(delta) < threshold:
           return [x, i]

   return None

def f1(x):
   return x**3 - 2*x - 5


def df1_dx(x):
   return 3*(x**2) - 2


eq = f1
derivatives = df1_dx


start_point = np.array([-20])


root = newton(eq, derivatives, start_point)

if root is not None:
   print("Root found:", root[0], " in ", root[1], " steps")
else:
```

```
    print("Newton's method did not converge.")

# Plot the functions
x_vals = np.linspace(-10, 10, 400)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```



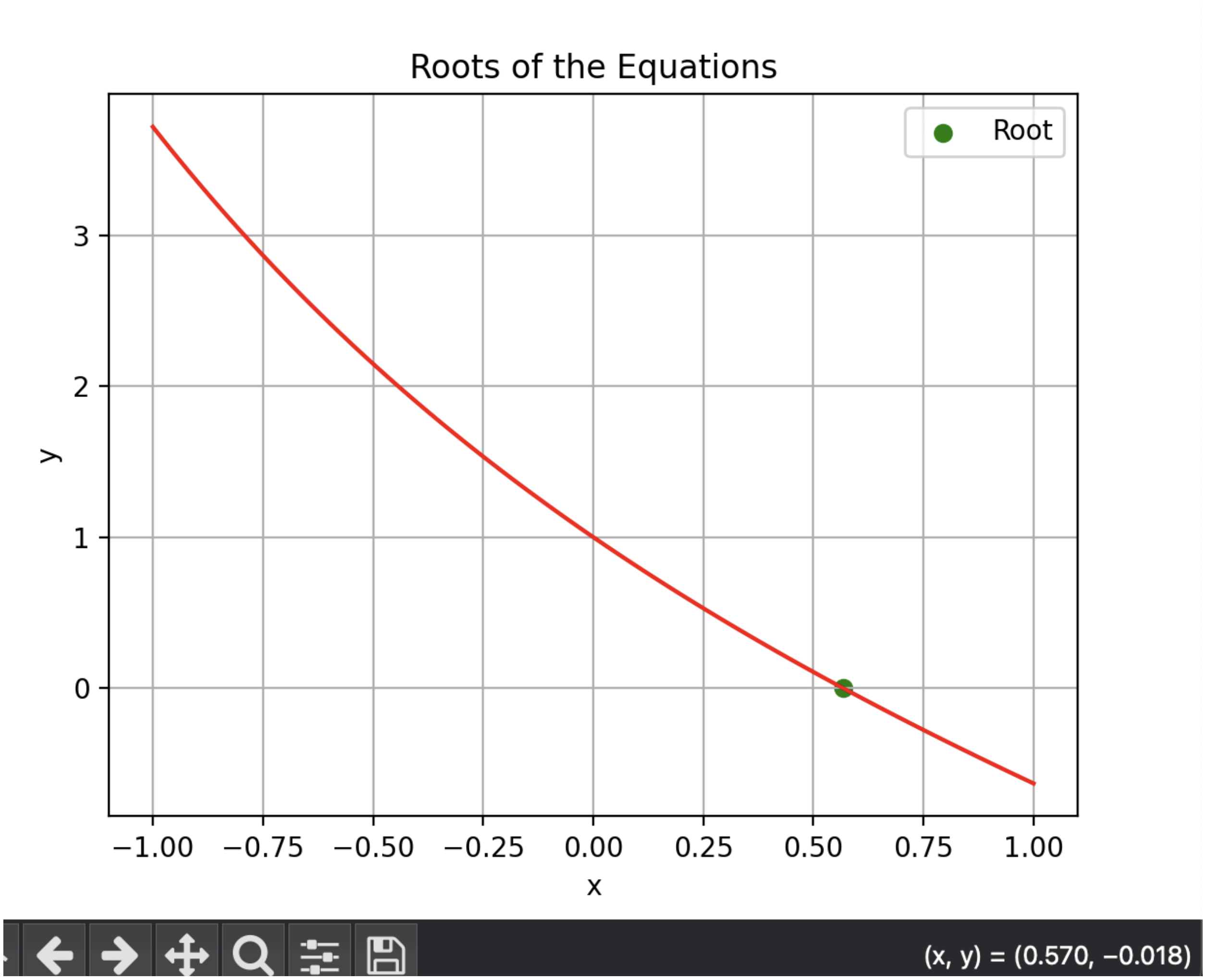

```
$ python newton_solver.py
Root found: [2.09455148]  in  28  steps
```

(b)

```
import numpy as np
import matplotlib.pyplot as plt
```

```python
def newton(eq, derivatives, start_point, threshold=1e-6, max_steps=1000):
   x = start_point.copy()
   delta = eq(*x) / derivatives(*x)
   for i in range(max_steps):
       delta = eq(*x) / derivatives(*x)
       x = x - delta

       if abs(delta) < threshold:
           return [x, i]

   return None

def f1(x):
   return np.exp(-x) - x


def df1_dx(x):
   return (-1) * np.exp(-x) - 1


eq = f1
derivatives = df1_dx


start_point = np.array([-3])


root = newton(eq, derivatives, start_point)

if root is not None:
   print("Root found:", root[0], " in ", root[1], " steps")
else:
   print("Newton's method did not converge.")

# Plot the functions
x_vals = np.linspace(-1, 1, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
```

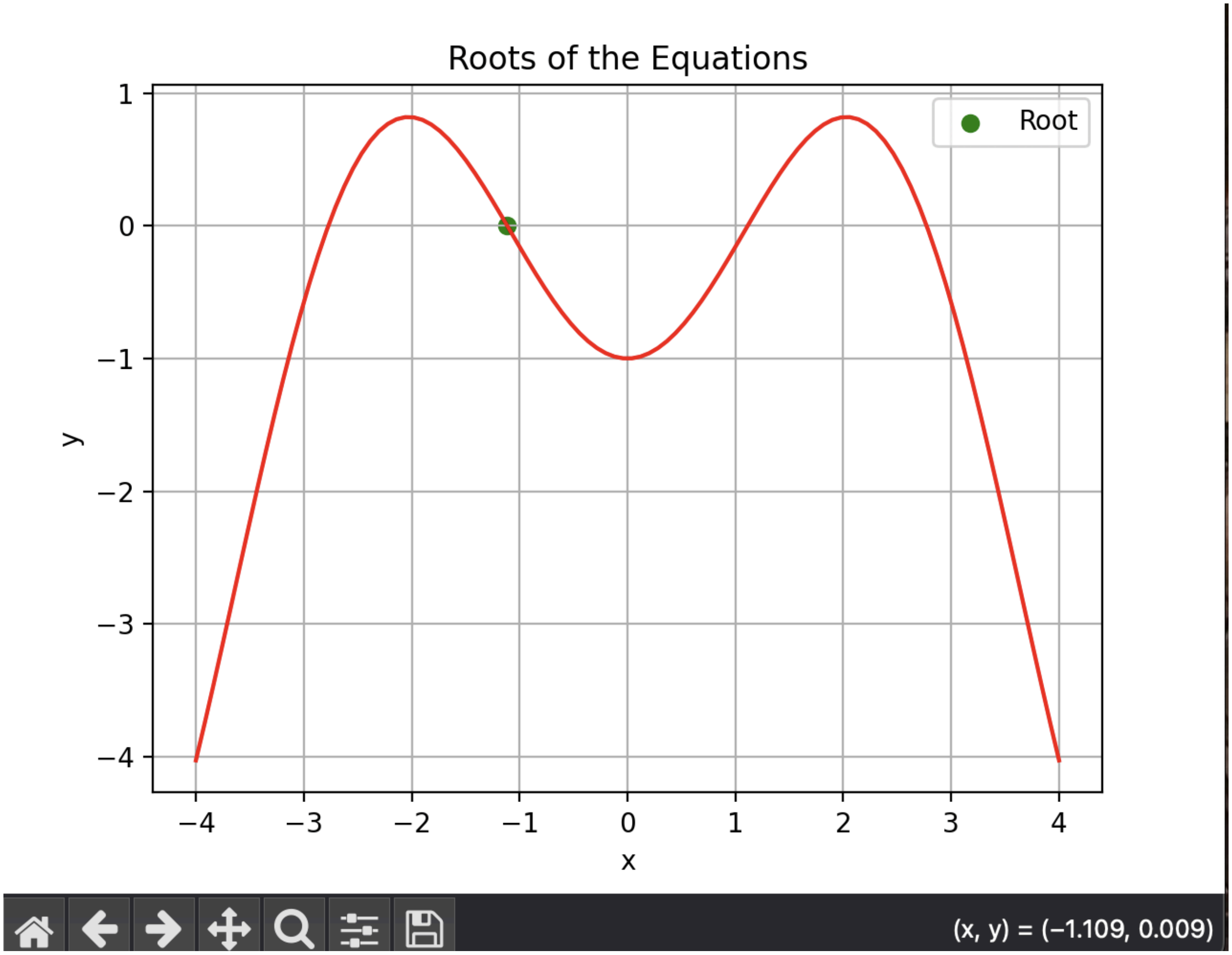
```
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```



(c)

```
import numpy as np
import matplotlib.pyplot as plt


def newton(eq, derivatives, start_point, threshold=1e-6, max_steps=1000):
   x = start_point.copy()
   delta = eq(*x) / derivatives(*x)
   for i in range(max_steps):
       delta = eq(*x) / derivatives(*x)
       x = x - delta
```

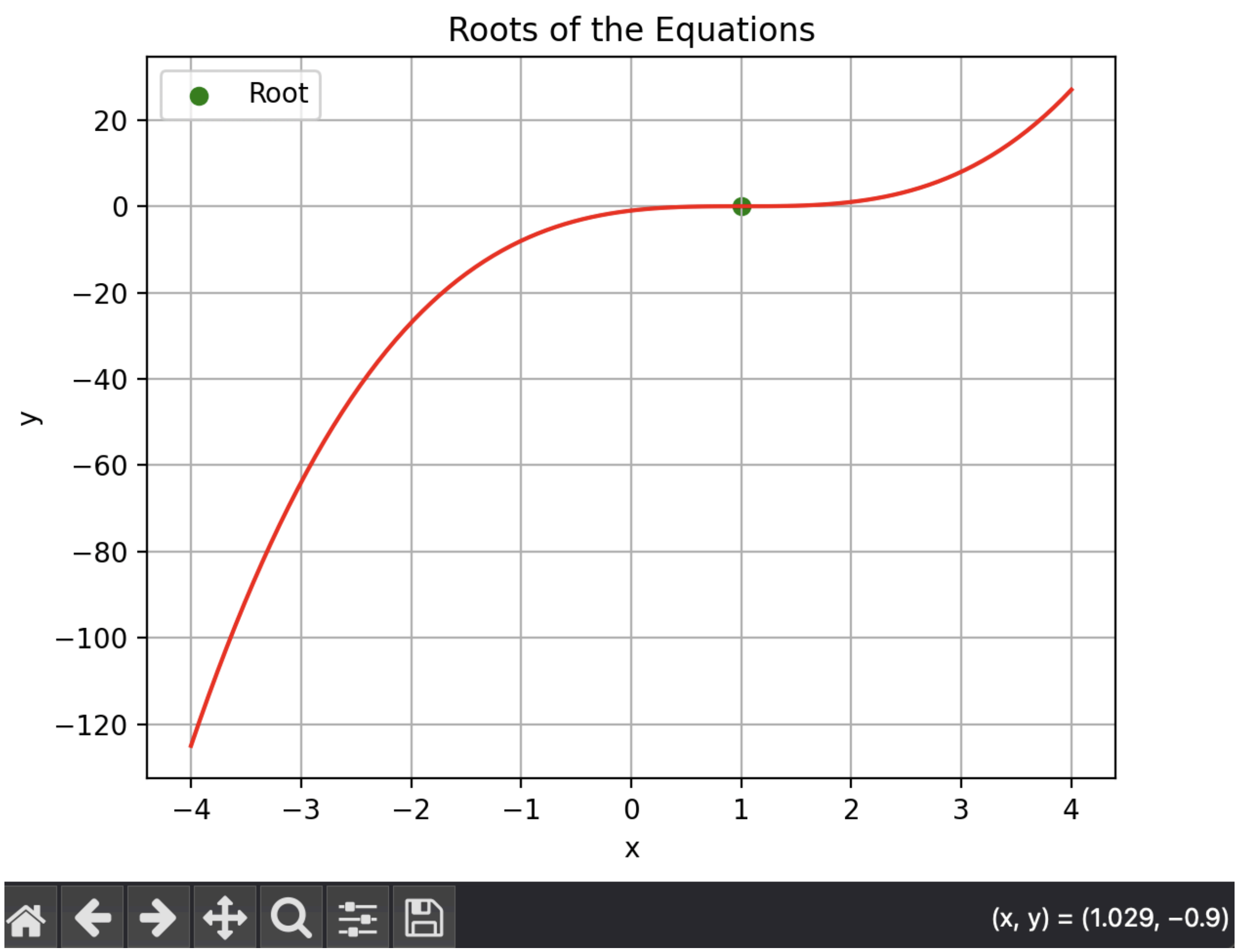
```
        if abs(delta) < threshold:
            return [x, i]

    return None

def f1(x):
    return x * np.sin(x) - 1


def df1_dx(x):
    return np.sin(x) + np.cos(x) * x


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = newton(eq, derivatives, start_point)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("Newton's method did not converge.")

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```



(d)

```
import numpy as np
import matplotlib.pyplot as plt


def newton(eq, derivatives, start_point, threshold=1e-6, max_steps=1000):
    x = start_point.copy()
    delta = eq(*x) / derivatives(*x)
    for i in range(max_steps):
        delta = eq(*x) / derivatives(*x)
        x = x - delta

        if abs(delta) < threshold:
            return [x, i]

    return None

def f1(x):
```

```
   return x**3 - 3 * (x**2) + 3*x - 1


def df1_dx(x):
   return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = newton(eq, derivatives, start_point)

if root is not None:
   print("Root found:", root[0], " in ", root[1], " steps")
else:
   print("Newton's method did not converge.")

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```



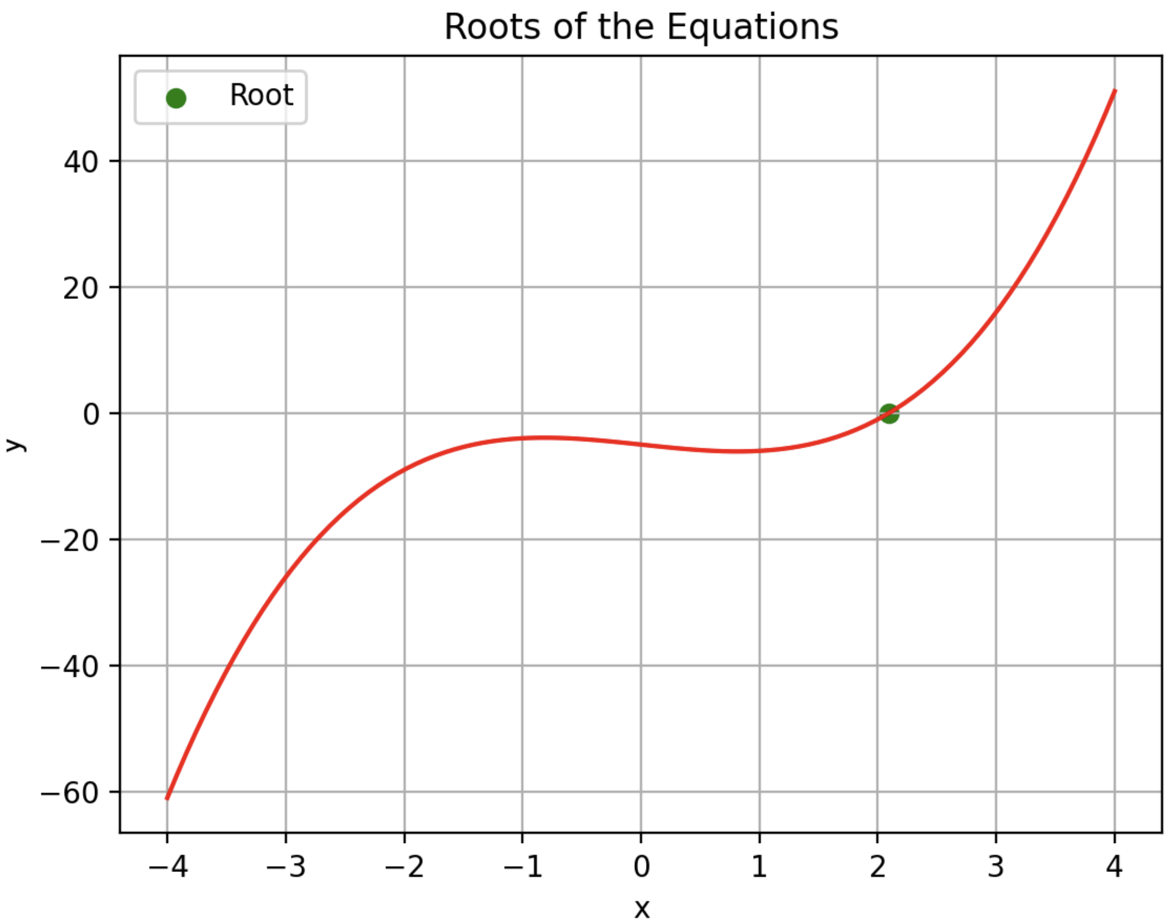
Secant

(a)

```
import numpy as np
import matplotlib.pyplot as plt


def secant(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   x0 = 0
   xm = 0
   c = 0
   if (f1(x1) * f1(x2) < 0):
       for i in range(max_steps):
           x0 = ((x1 * f1(x2) - x2 * f1(x1))/ (f1(x2) - f1(x1)))
           c = f1(x1) * f1(x0)
```

```
            x1 = x2
            x2 = x0
            if(c == 0):
                return [x0, i]
            xm = ((x1 * f1(x2) - x2 * f1(x1)) / (f1(x2) - f1(x1)))
            if(abs(xm - x0) < threshold):
                return [x0, i]

    return None

def f1(x):
    return x**3 - 2*x - 5


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = secant(eq, derivatives, -10, 10)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("secant's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
```

```
plt.legend()
plt.grid(True)
plt.show()
```



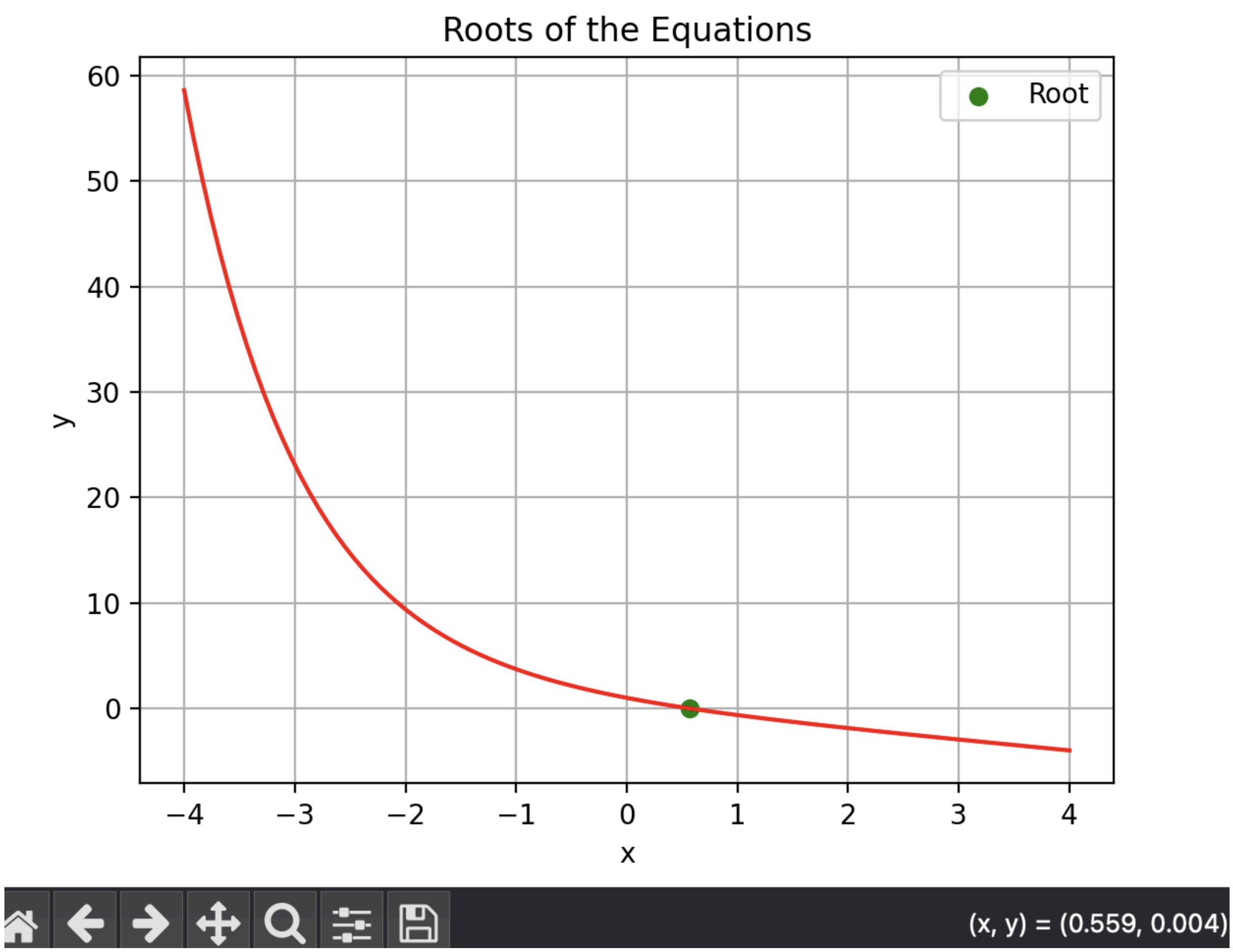

```
Root found: 2.09455185944897  in  10  steps
```

(b)

```
import numpy as np
import matplotlib.pyplot as plt


def secant(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   x0 = 0
   xm = 0
   c = 0
   if (f1(x1) * f1(x2) < 0):
       for i in range(max_steps):
           x0 = ((x1 * f1(x2) - x2 * f1(x1))/ (f1(x2) - f1(x1)))
```

```
            c = f1(x1) * f1(x0)
            x1 = x2
            x2 = x0
            if(c == 0):
                return [x0, i]
            xm = ((x1 * f1(x2) - x2 * f1(x1)) / (f1(x2) - f1(x1)))
            if(abs(xm - x0) < threshold):
                return [x0, i]

   return None

def f1(x):
   return np.exp(-x) - x


def df1_dx(x):
   return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = secant(eq, derivatives, -1, 2)

if root is not None:
   print("Root found:", root[0], " in ", root[1], " steps")
else:
   print("secant's method did not converge.")
   exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```

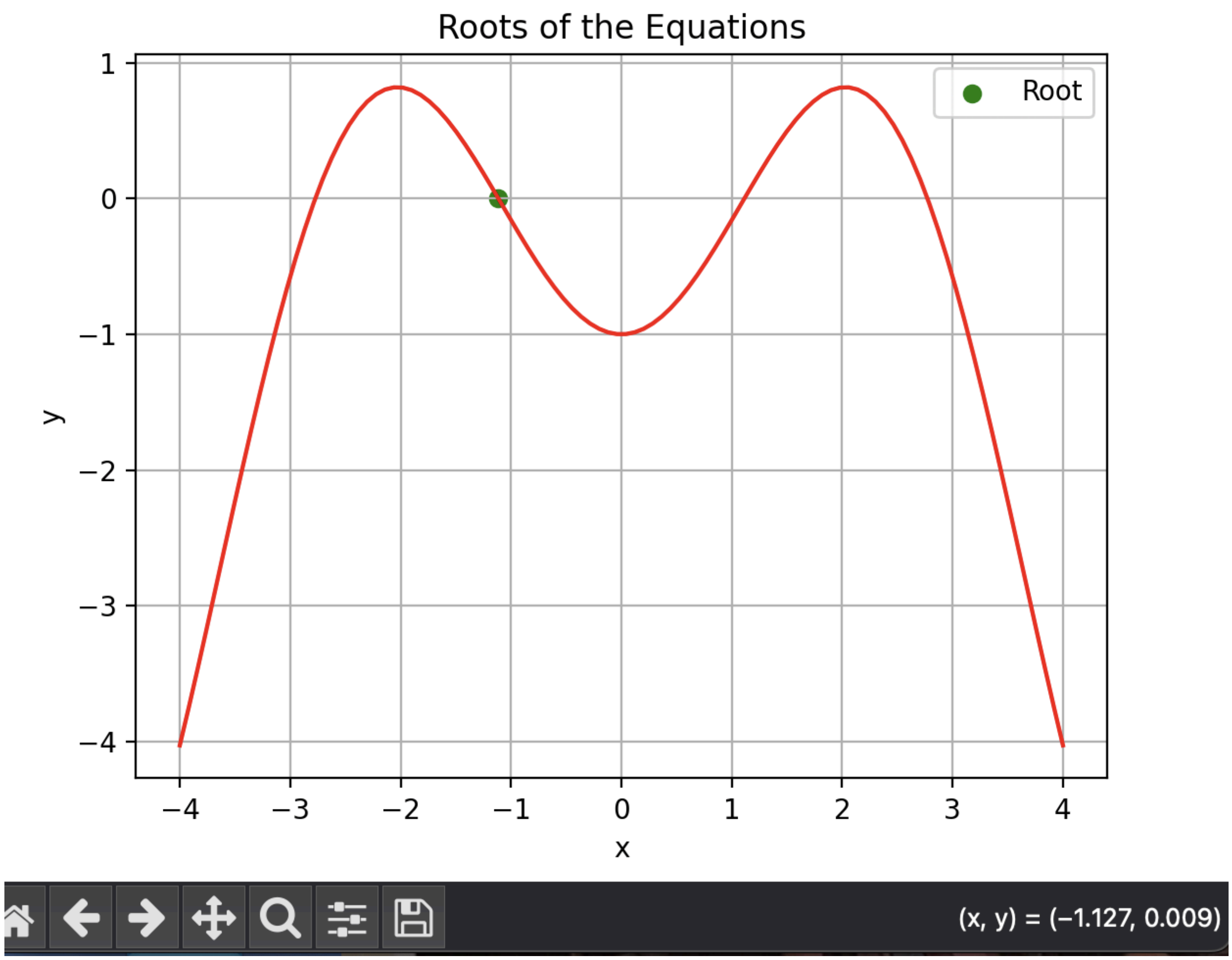


(c)

```
import numpy as np
import matplotlib.pyplot as plt


def secant(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   x0 = 0
   xm = 0
   c = 0
   if (f1(x1) * f1(x2) < 0):
       for i in range(max_steps):
           x0 = ((x1 * f1(x2) - x2 * f1(x1))/ (f1(x2) - f1(x1)))
           c = f1(x1) * f1(x0)
           x1 = x2
```

```
           x2 = x0
           if(c == 0):
               return [x0, i]
           xm = ((x1 * f1(x2) - x2 * f1(x1)) / (f1(x2) - f1(x1)))
           if(abs(xm - x0) < threshold):
               return [x0, i]

   return None

def f1(x):
   return x * np.sin(x) - 1


def df1_dx(x):
   return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = secant(eq, derivatives, -2, 0)

if root is not None:
   print("Root found:", root[0], " in ", root[1], " steps")
else:
   print("secant's method did not converge.")
   exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

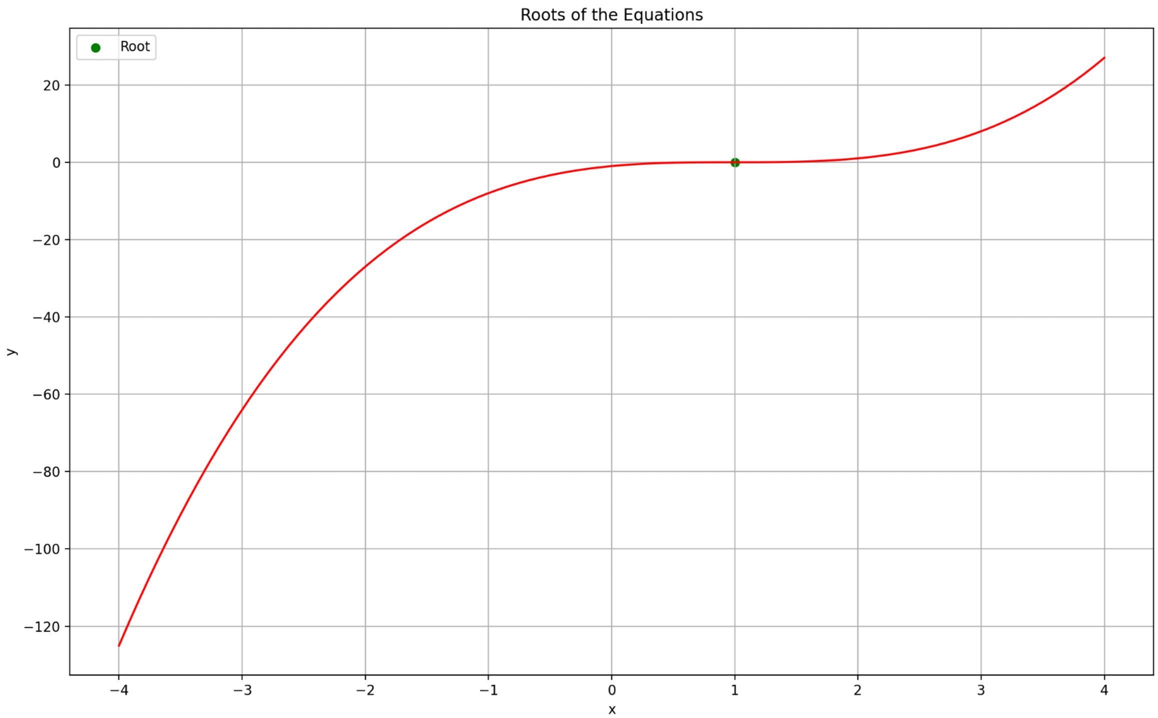


(d)

```
import numpy as np
import matplotlib.pyplot as plt


def secant(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   x0 = 0
   xm = 0
   c = 0
   if (f1(x1) * f1(x2) < 0):
       for i in range(max_steps):
           x0 = ((x1 * f1(x2) - x2 * f1(x1))/ (f1(x2) - f1(x1)))
           c = f1(x1) * f1(x0)
           x1 = x2
           x2 = x0
```

```
            if(c == 0):
                return [x0, i]
            xm = ((x1 * f1(x2) - x2 * f1(x1)) / (f1(x2) - f1(x1)))
            if(abs(xm - x0) < threshold):
                return [x0, i]

    return None

def f1(x):
    return x**3 - 3 * (x**2) + 3*x - 1


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = secant(eq, derivatives, -2, 2)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("secant's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
```

```
plt.show()
```



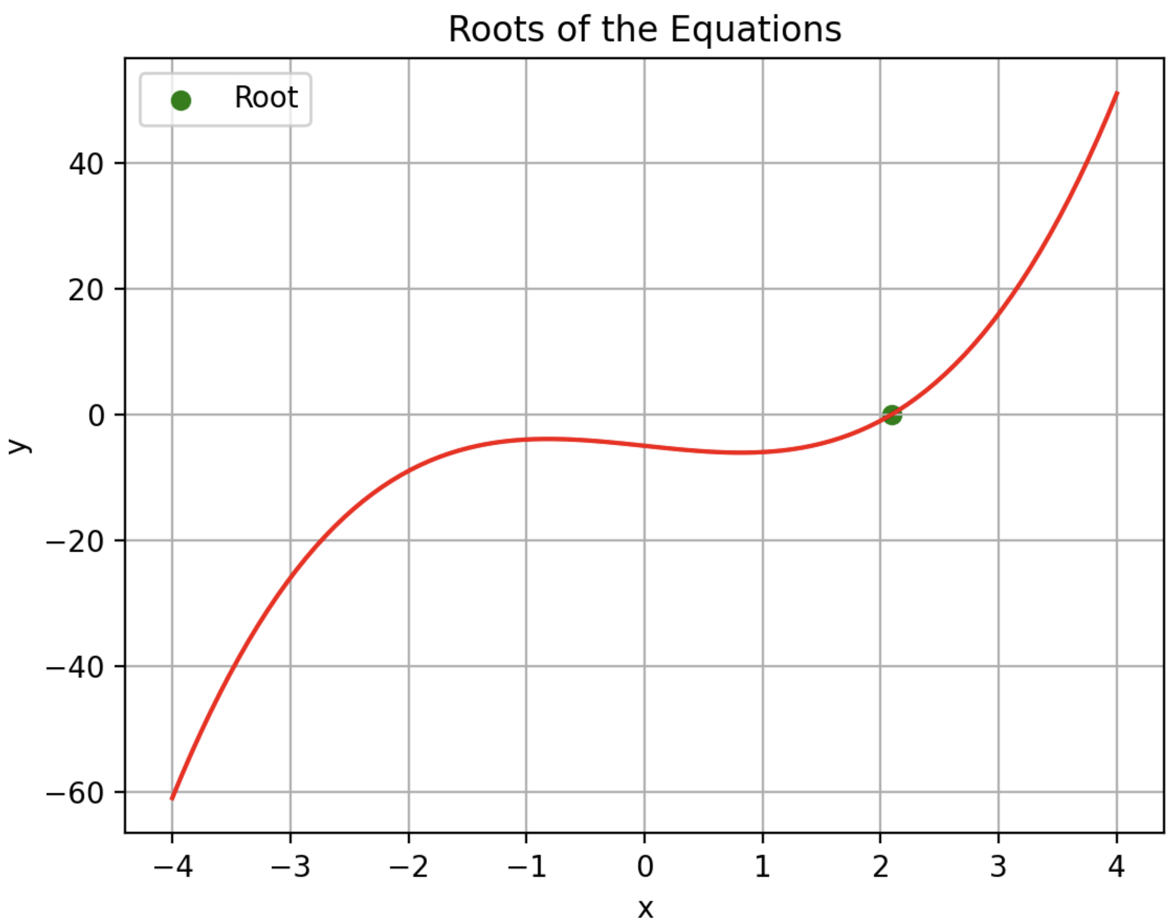

```
Root found: 1.0000034439815826  in  42  steps
```

Bisection

(a)

```
import numpy as np
import matplotlib.pyplot as plt


def ss(a, b):
    return a*b > 0
def bisection(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
    assert not ss(eq(x1), eq(x2))
    for i in range(max_steps):
        mid = (x1 + x2) / 2.0
        if ss(eq(x1), eq(mid)):
            x1 = mid
        else:
            x2 = mid
        if abs(x2 - x1) < threshold:
```

```
            break
    return [mid, i]

def f1(x):
    return x**3 - 2*x - 5


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = bisection(eq, derivatives, -10, 10)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("bisection's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```

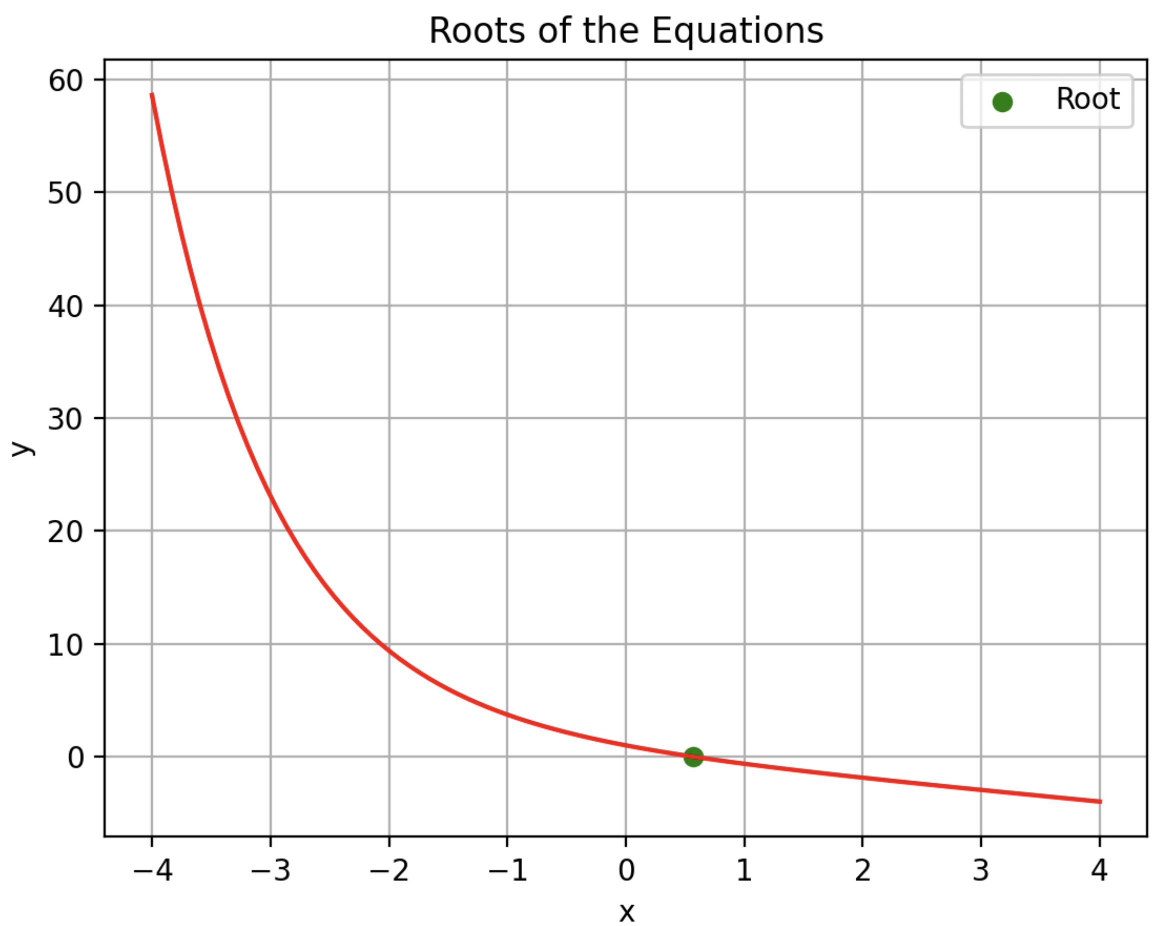



```
$ python newton_solver.py
Root found: 2.094551920890808  in  24  steps
```

(b)

```
import numpy as np
import matplotlib.pyplot as plt


def ss(a, b):
   return a*b > 0
def bisection(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   assert not ss(eq(x1), eq(x2))
   for i in range(max_steps):
       mid = (x1 + x2) / 2.0
       if ss(eq(x1), eq(mid)):
           x1 = mid
       else:
           x2 = mid
```

```python
        if abs(x2 - x1) < threshold:
            break
    return [mid, i]

def f1(x):
    return np.exp(-x) - x


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = bisection(eq, derivatives, -10, 10)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("bisection's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```

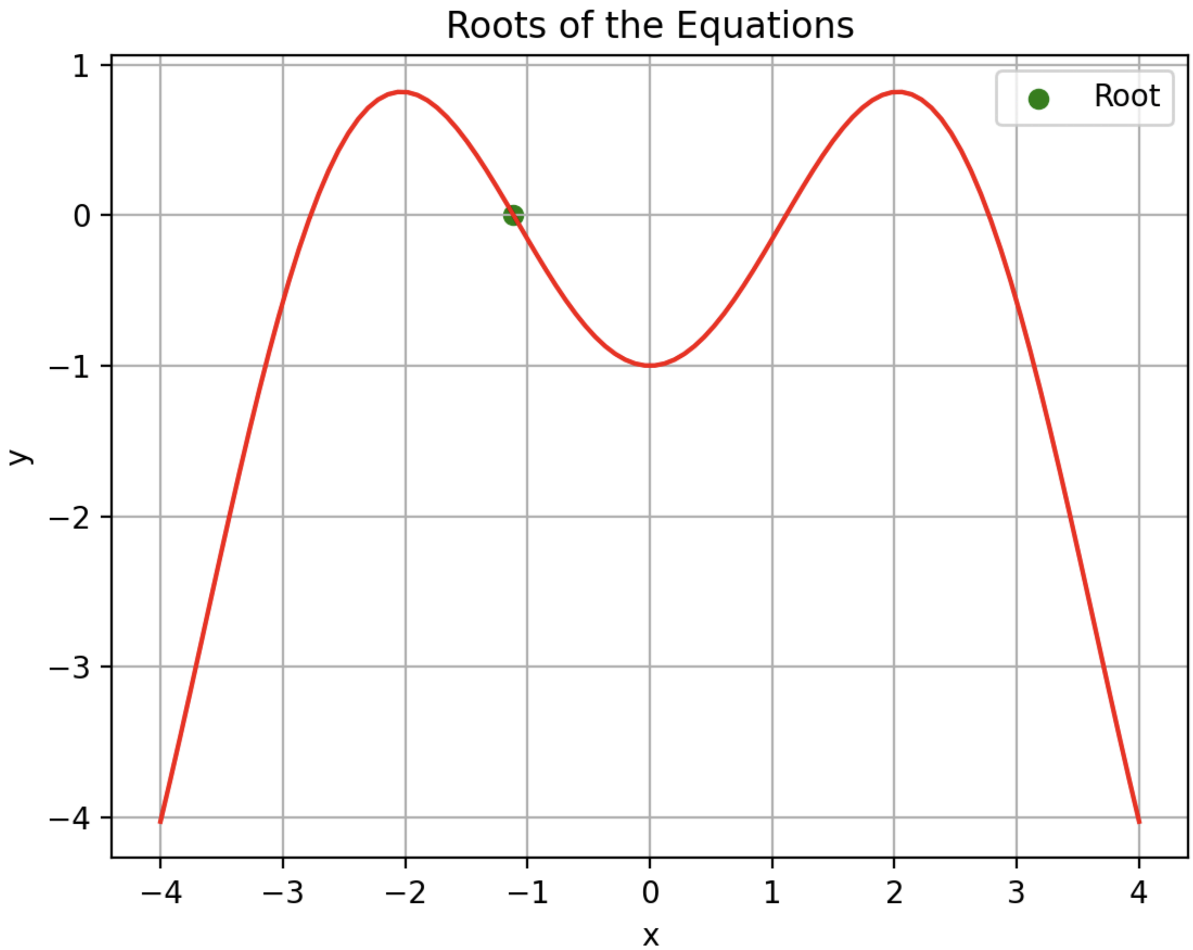



```
└─$ python newton_solver.py
Root found: 0.5671435594558716  in  24  steps
```

(c)

```python
import numpy as np
import matplotlib.pyplot as plt


def ss(a, b):
   return a*b > 0
def bisection(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   assert not ss(eq(x1), eq(x2))
   for i in range(max_steps):
       mid = (x1 + x2) / 2.0
       if ss(eq(x1), eq(mid)):
           x1 = mid
       else:
```

```
            x2 = mid
        if abs(x2 - x1) < threshold:
            break
    return [mid, i]

def f1(x):
    return x * np.sin(x) - 1


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3


eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = bisection(eq, derivatives, -2, 0)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("bisection's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```

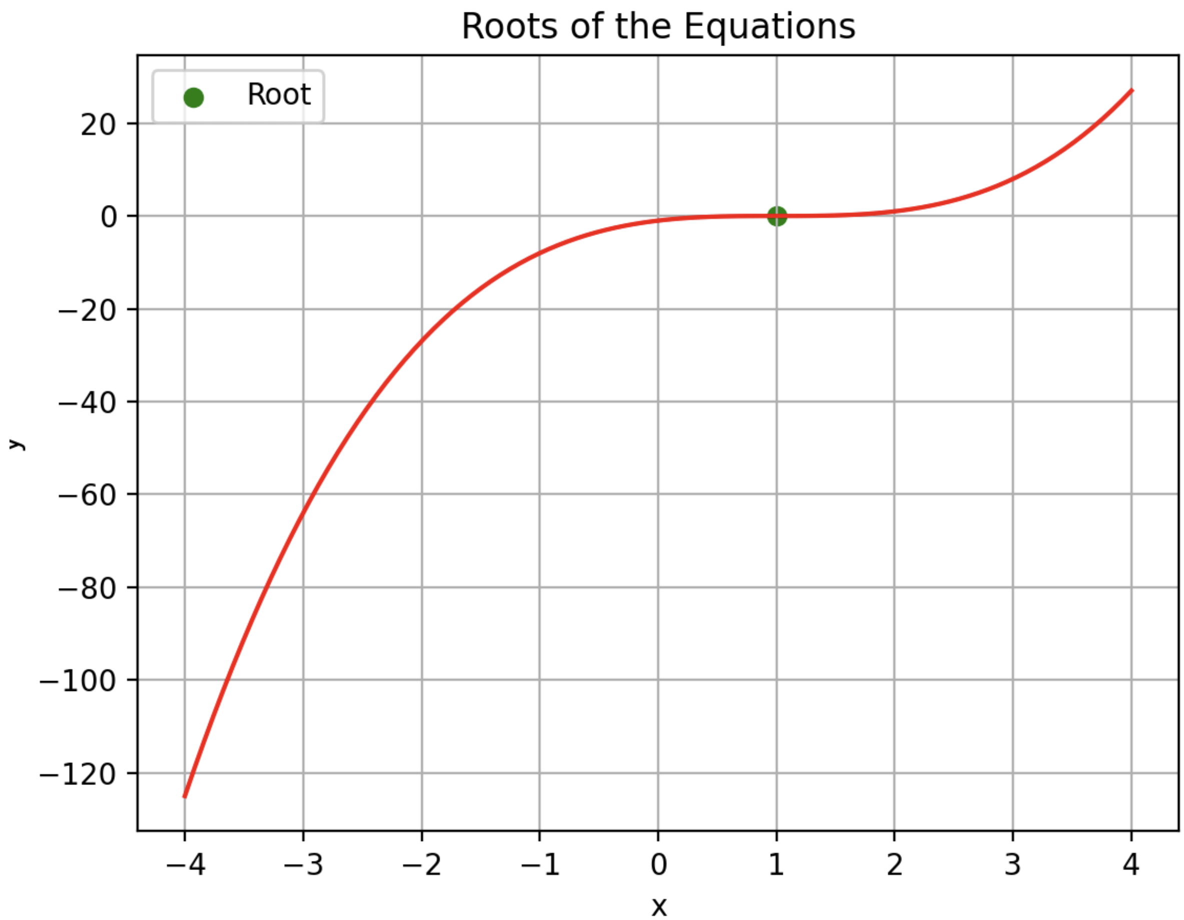



```
└─$ python newton_solver.py
Root found: -1.1141576766967773  in  20  steps
```

(d)

```
import numpy as np
import matplotlib.pyplot as plt


def ss(a, b):
   return a*b > 0
def bisection(eq, derivatives, x1, x2, threshold=1e-6, max_steps=1000):
   assert not ss(eq(x1), eq(x2))
   for i in range(max_steps):
       mid = (x1 + x2) / 2.0
       if ss(eq(x1), eq(mid)):
           x1 = mid
```

```
        else:
            x2 = mid
        if abs(x2 - x1) < threshold:
            break
    return [mid, i]

def f1(x):
    return (x**3) - 3 * (x**2) + 3*x -1


def df1_dx(x):
    return 3*(x**2) - 6 * x + 3



eq = f1
derivatives = df1_dx


start_point = np.array([-1.5])


root = bisection(eq, derivatives, -10, 10)

if root is not None:
    print("Root found:", root[0], " in ", root[1], " steps")
else:
    print("bisection's method did not converge.")
    exit()

# Plot the functions
x_vals = np.linspace(-4, 4, 100)

y = f1(x_vals)
plt.plot(x_vals, y, color='r')
plt.scatter(root[0], 0, color='green', marker='o', label='Root')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```




```
$ python newton_solver.py
Root found: 0.9999948740005493  in  24  steps
```

7 Given

$x_1^2 - x_2^2 = 0$

$2x_1x_2 = 1$

$x_0 = (0,1)^T$ {Starting Value}

To Find

1st iteration of the Newton's Method

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{2,2} \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}$$

$$\begin{bmatrix} 2x_1 & -2x_2 \\ 2x_2 & 2x_1 \end{bmatrix}$$

Now

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} - \begin{bmatrix} 2x_1 & -2x_2 \\ 2x_2 & 2x_1 \end{bmatrix}^{-1} \times \bar{f}(x_1, x_2)$$

$$\Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 & -2 \\ 2 & 0 \end{bmatrix}^{-1} \times \bar{f}(0,1)$$

$$\Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 & -2 \\ 2 & 0 \end{bmatrix}^{-1} \times \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 & 1/2 \\ -1/2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} +1/2 \\ 1/2 \end{bmatrix}$$

$\therefore$ After 1 iteration, its $[0.5, 0.5]$