HW7 Aaryan Bhagat 862468325

(1)

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

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

$\text{Rank}(A) = 1 \to (I)$

(a) To Prove

$A = uv^T \to (V)$

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

From (I) it implies that each column of A can be represented as some basis

$\therefore c_i = \alpha_i \beta_1$ { β_1 is the basis $\in \mathbb{R}^n$; $\alpha_i \in \mathbb{R}$ $c_i \to i^{th}$ coluran of A ; $i \in \{1, 2 \dots n\}$ }

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

$\Rightarrow A = uv^T$

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

(b) To Prove

$u^T v$ is an eigenvalue of $A \rightarrow (iv)$

$u^T v = \beta^T \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}$ {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 $x = u$

$\Rightarrow uv^T u = u^T v u$

$v^T u$: Scalar

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

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

Hence proved

(c) To Find

No of iterations required to converge

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

$\text{Rank}(A) = 1$

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

we know $\text{Rank}(v) = 1$

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

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

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

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

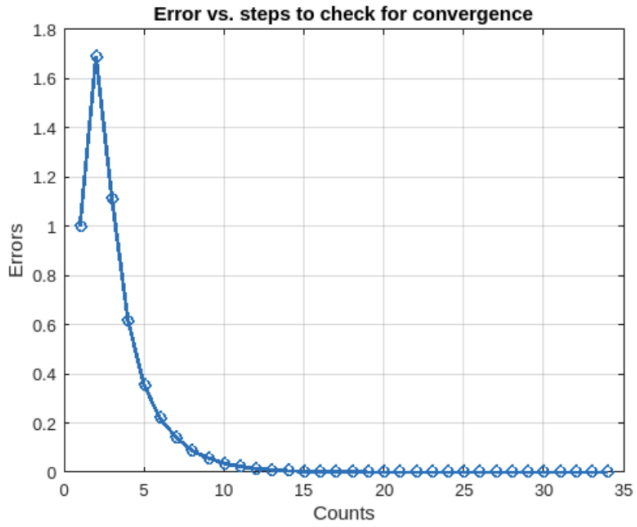
$y_1 = \frac{uv^T x_0}{\|Ax_0\|_2}$

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

Q3

(a)

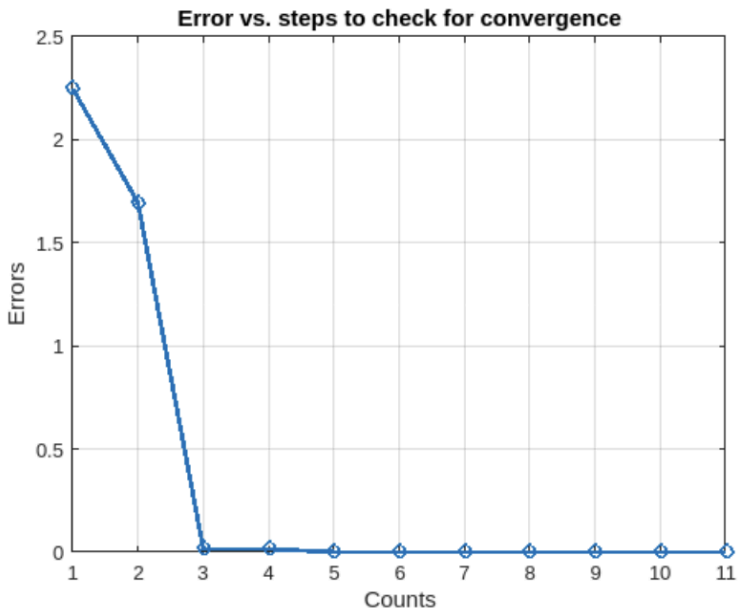
```
A = [3.5, 2, -1; 1, 2.5, 0; 1, 2, -3.5];
b = [1; 2; 3];
[m, n] = size(A);
d = diag(diag(A));
x = zeros(m, 1);
err = inf;
errs = [];
count = 0;
counts = [];
tolerance = 1e-6;
while err > tolerance
   count = count + 1;
   counts = [counts, count];
   dx = d\(b - A*x);
   x = x + dx;
   err = max(abs(dx./x));
   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)

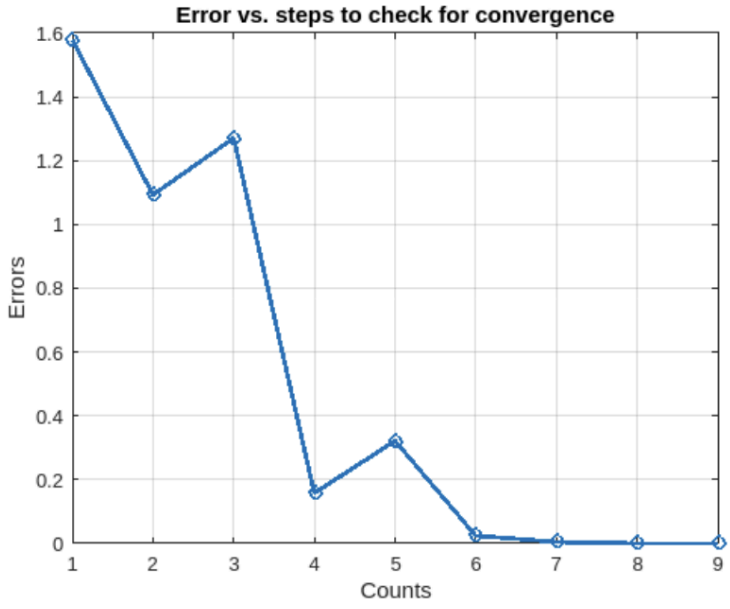
```
rng(12)
```

```
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

$d \rightarrow$ constant

Equation is $f(x) = 0$

(a) To find local convergence

⇒ Assume $[a, b]$ is the required interval

Then condition will be

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

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

(b) local convergence should satisfy

$|x_{k+1} - x^*| \leq q |x_k - x^*|$ {q is constant, x^* is true solution}

(b) Convergence Rate

Error Rate

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

Also

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

Ideally convergence rate is

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

For (III) to hold in the new scheme

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

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

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

$$e_{K+1} = \frac{e_k^2}{2} \frac{f''(x)}{f'(x)}$$

Hence $d \approx \frac{1}{f'(x^*)}$ where x^* is the actual root provided $f'(x^*) \neq 0$

Q6

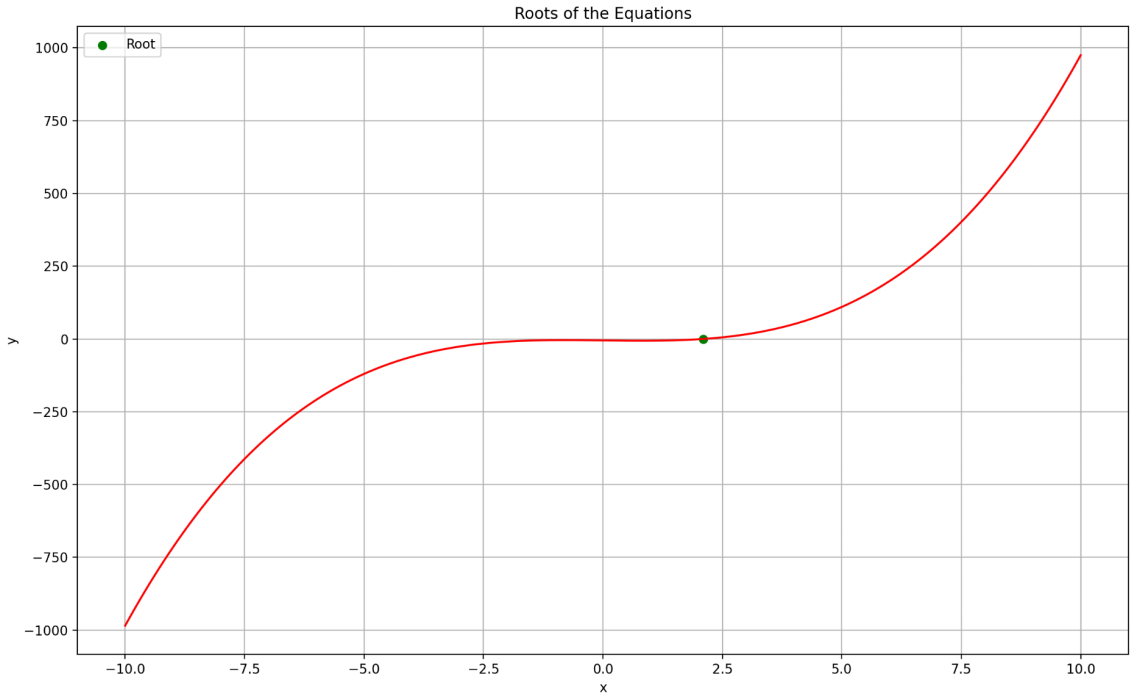
Newton

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()
```



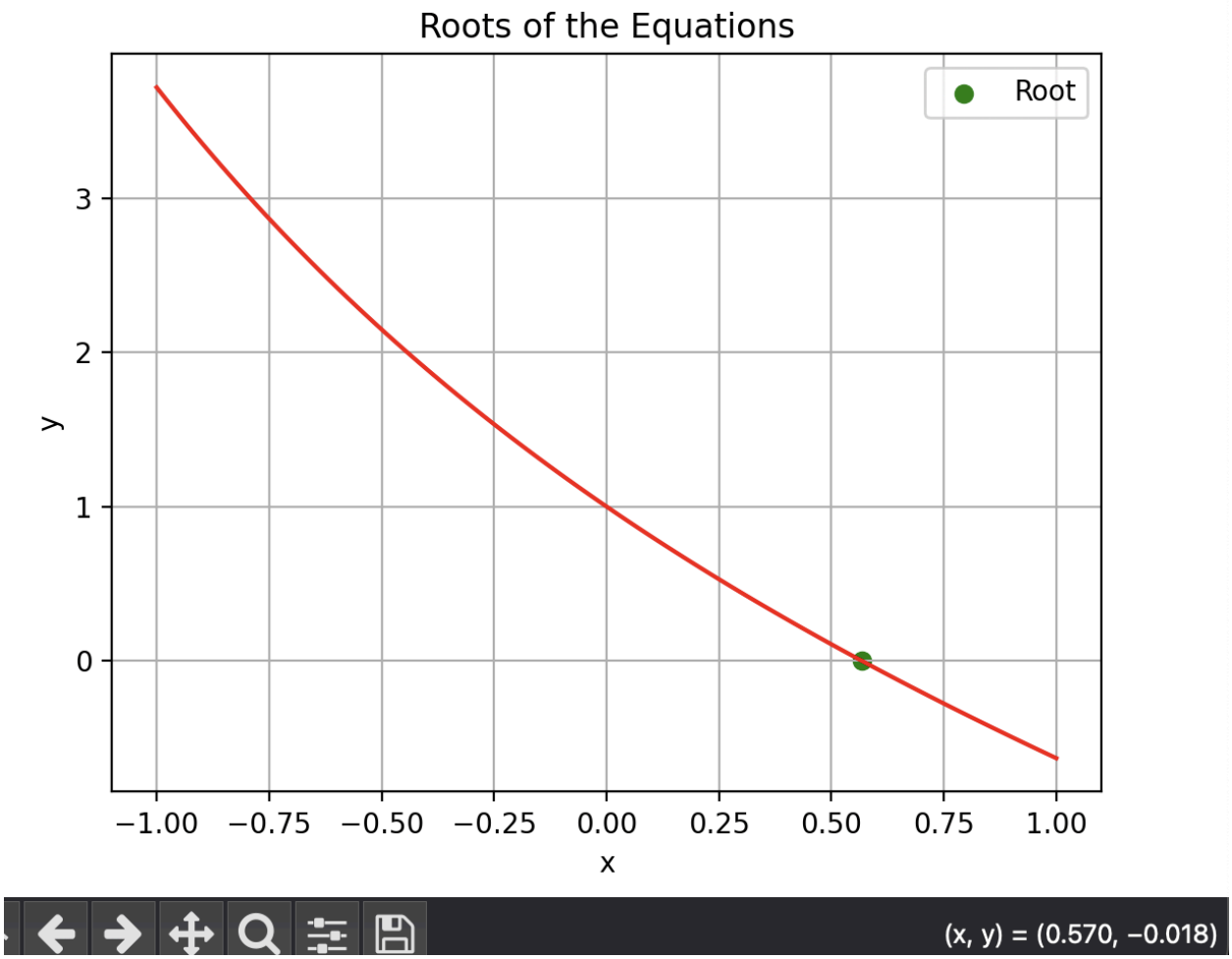

```
$ python newton_solver.py
Root found: [2.09455148] in 28 steps
```

(b)

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

```
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')
```

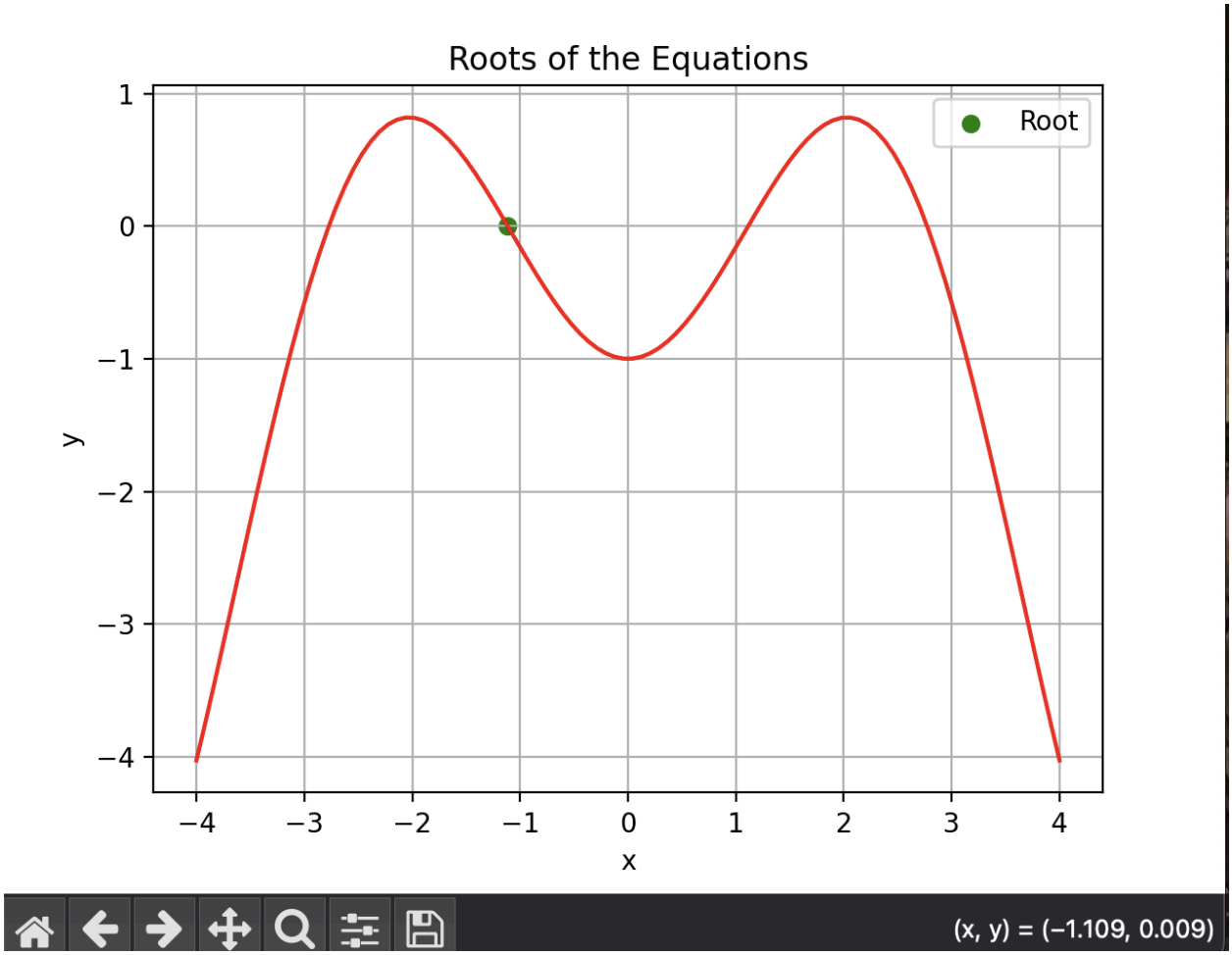
```
plt.ylabel('y')
plt.title('Roots of the Equations')
plt.legend()
plt.grid(True)
plt.show()
```



(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
```

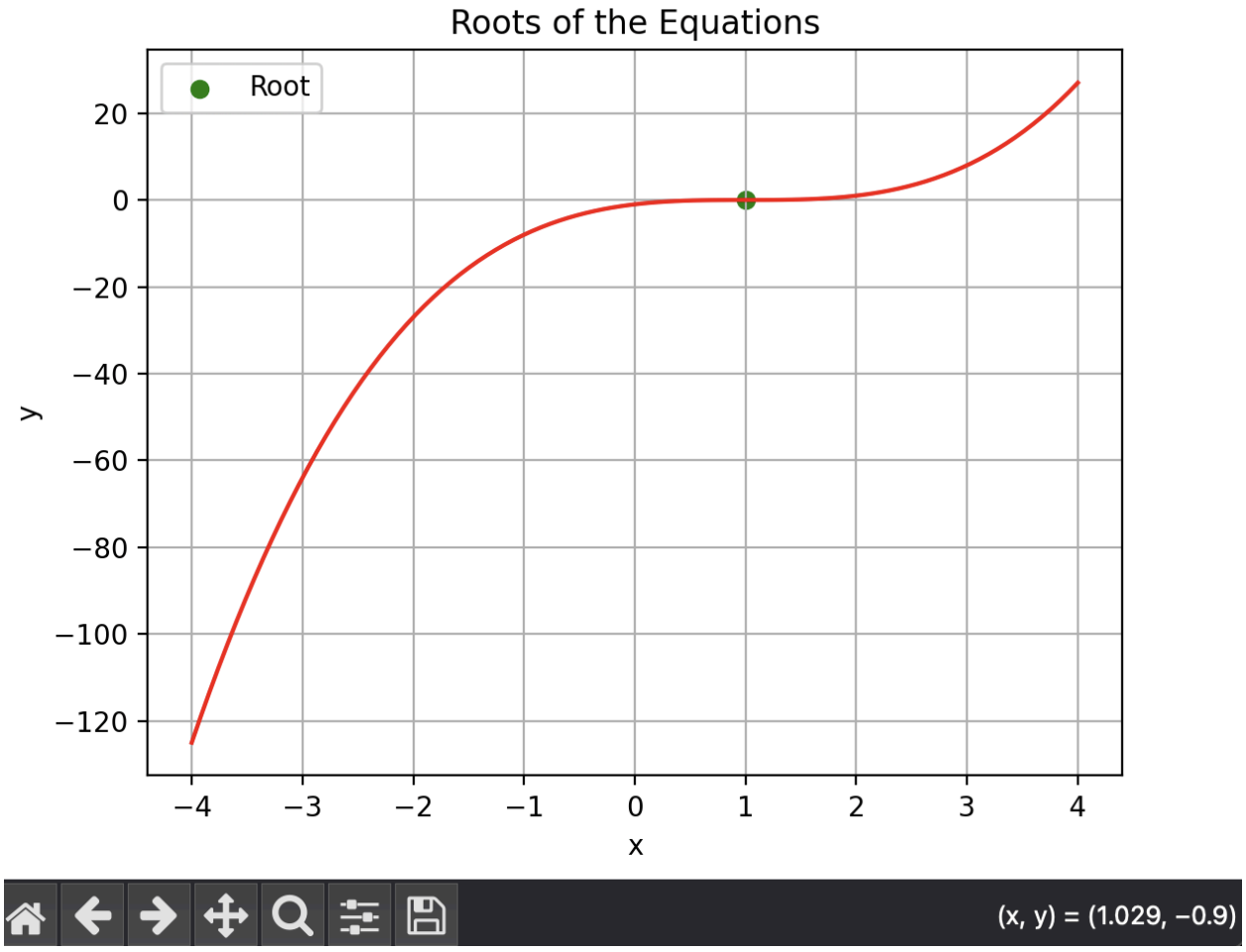
```
        if abs(delta) < threshold:
            return [x, i]

    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)

```python
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()
```



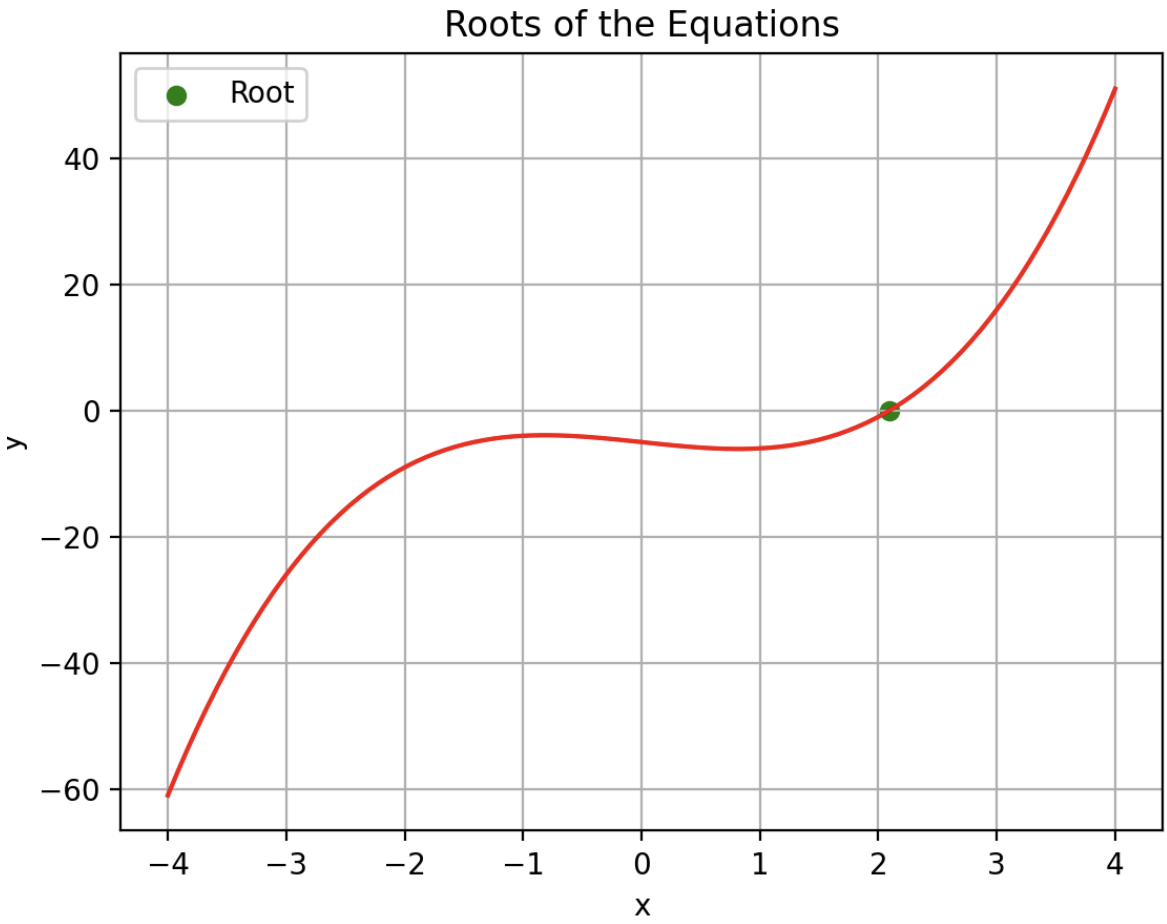
Secant

(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()
```



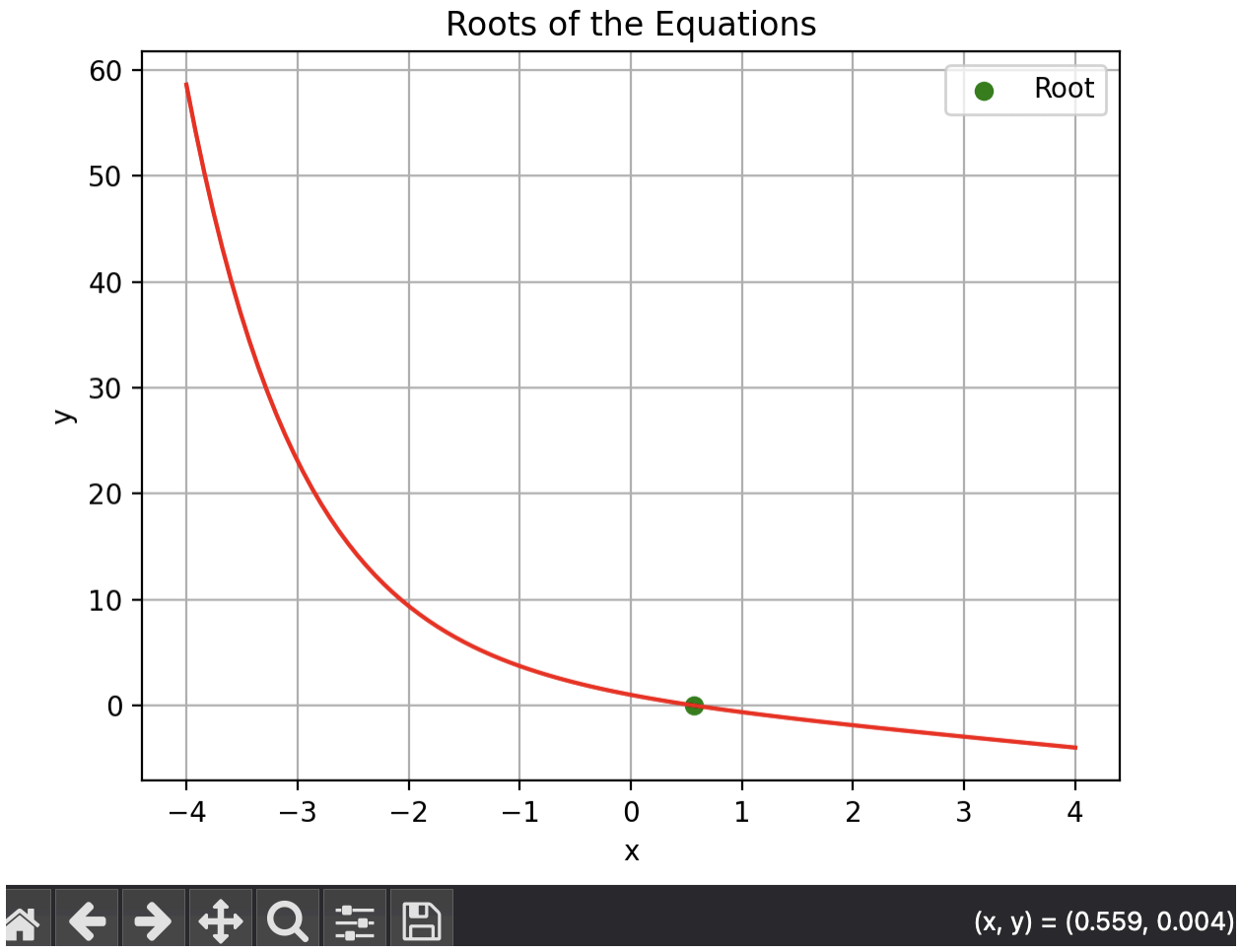

```
Root found: 2.094551859448897  in  10  steps
```

(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()
```

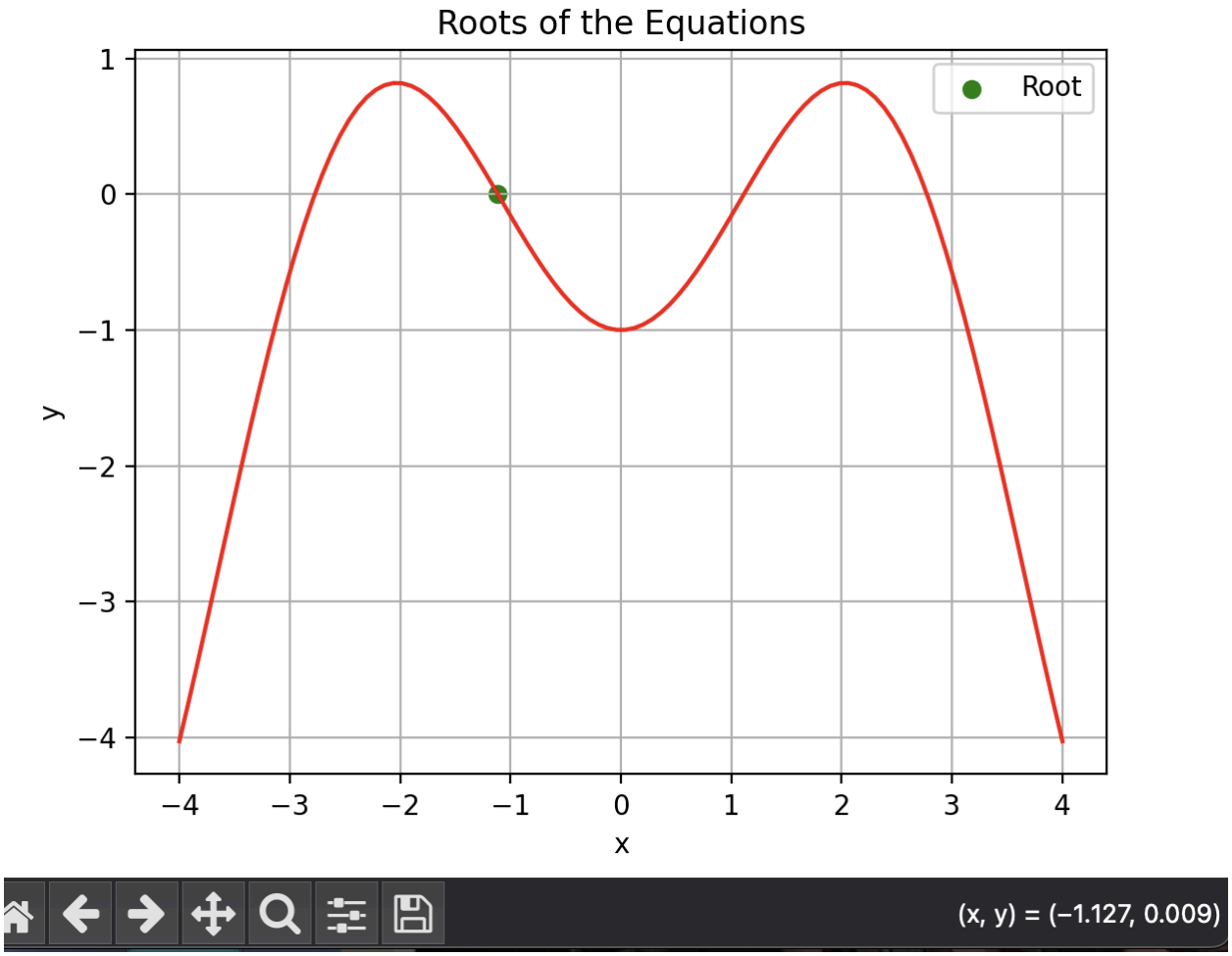


(c)

```
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()
```

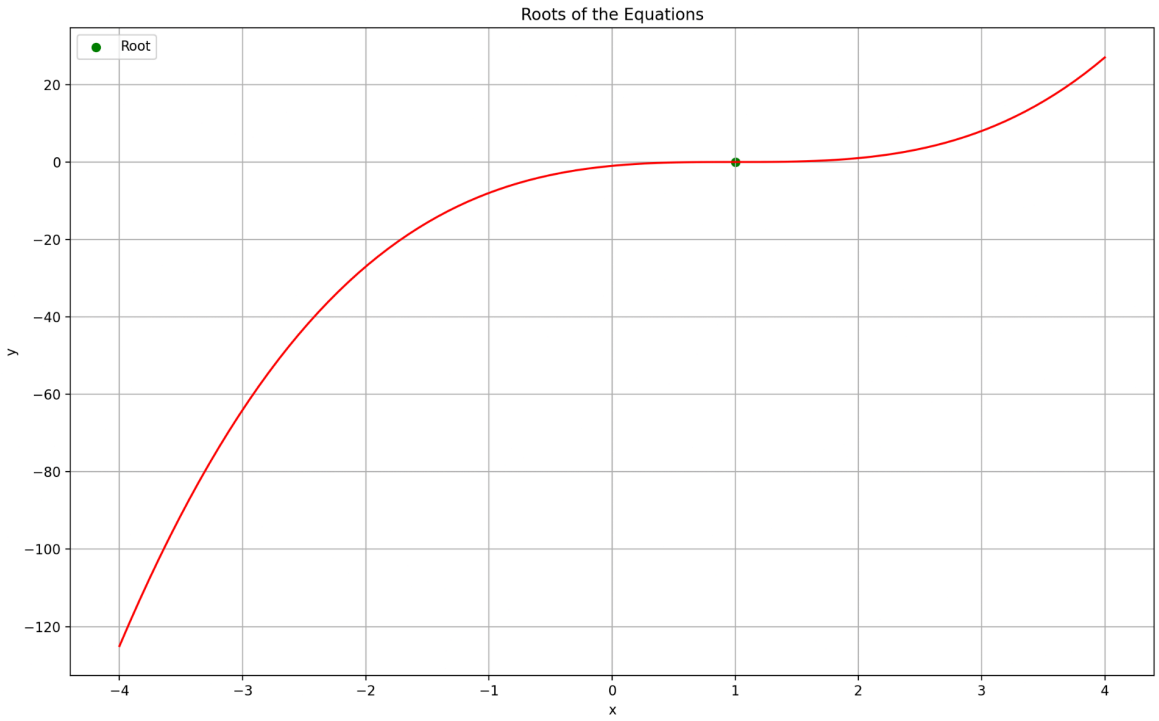


(d)

```
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()
```



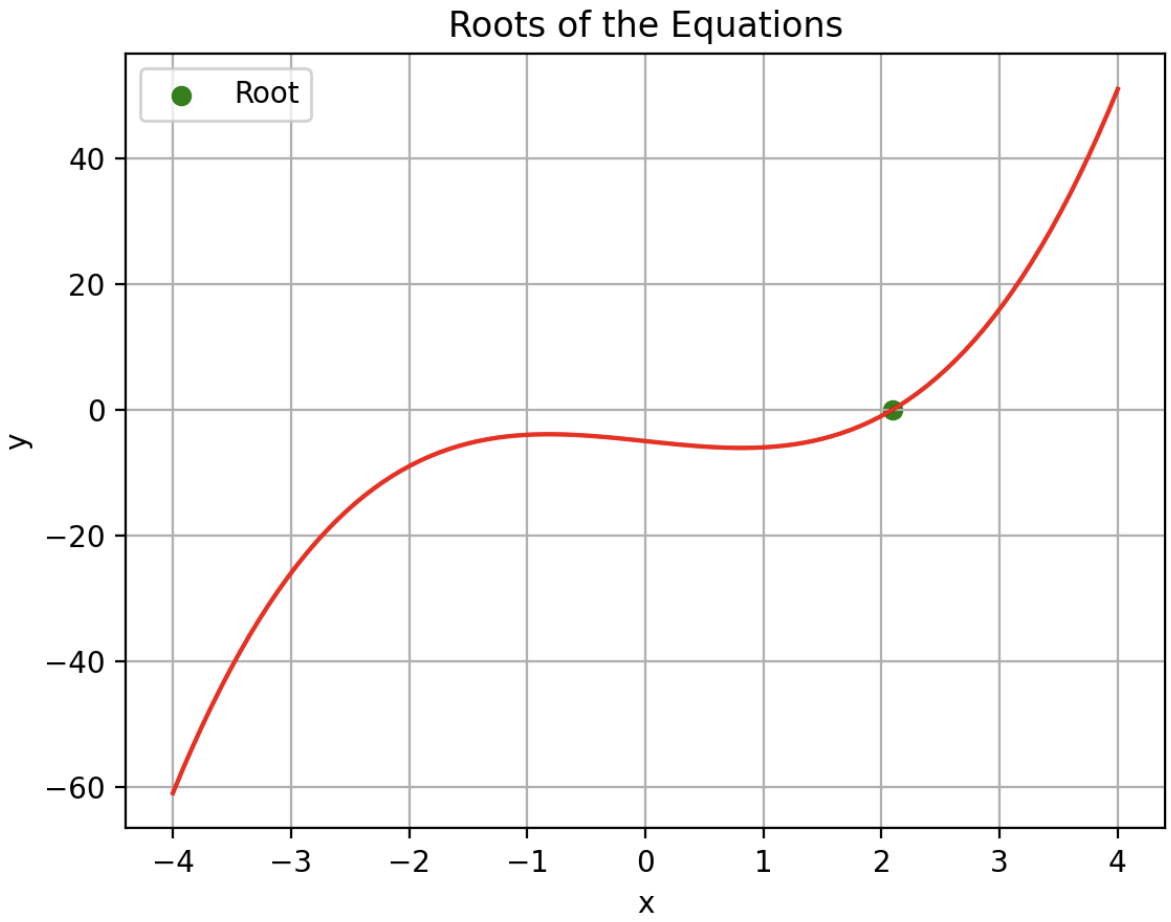

```
Root found: 1.0000034439815826  in  42  steps
```

Bisection

(a)

```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**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()
```

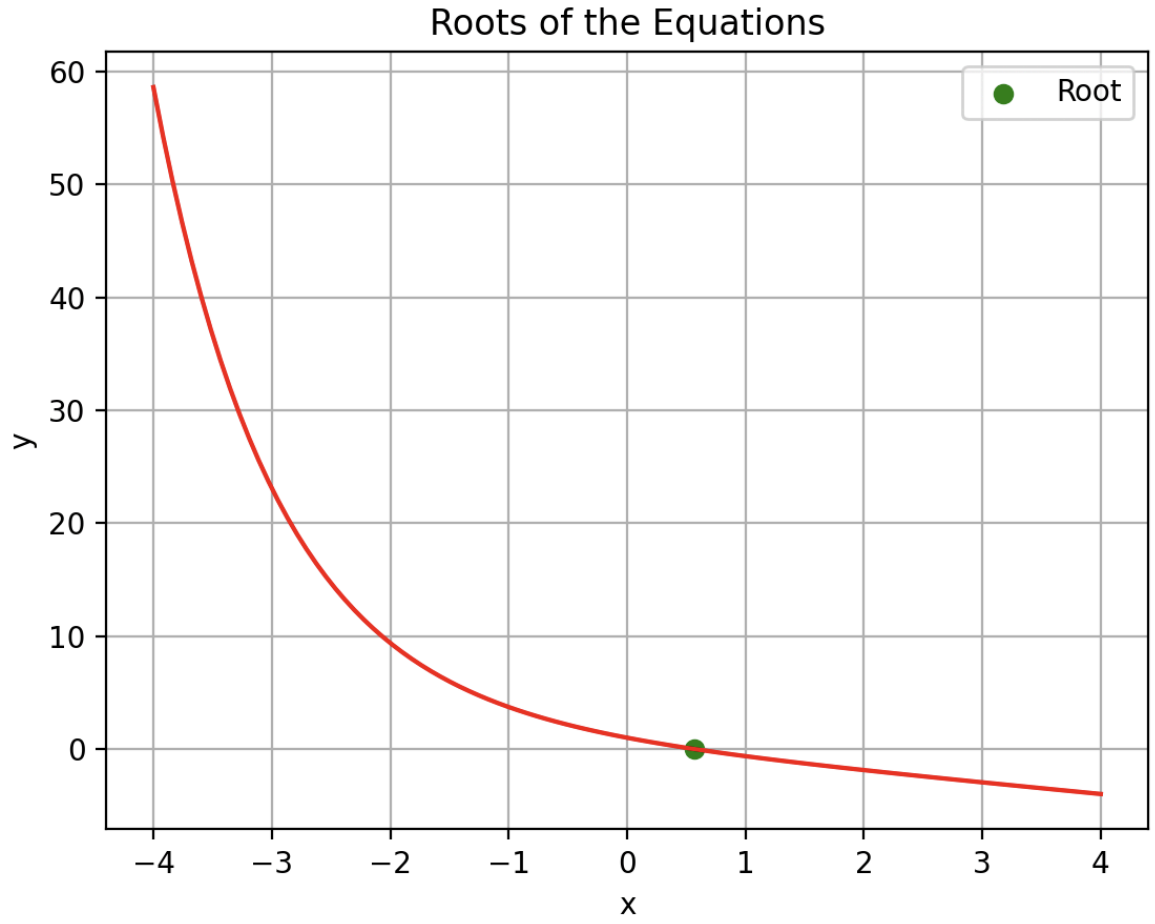



```
$ python newton_solver.py
Root found: 2.094551920890808  in  24  steps
```

(b)

```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 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()
```

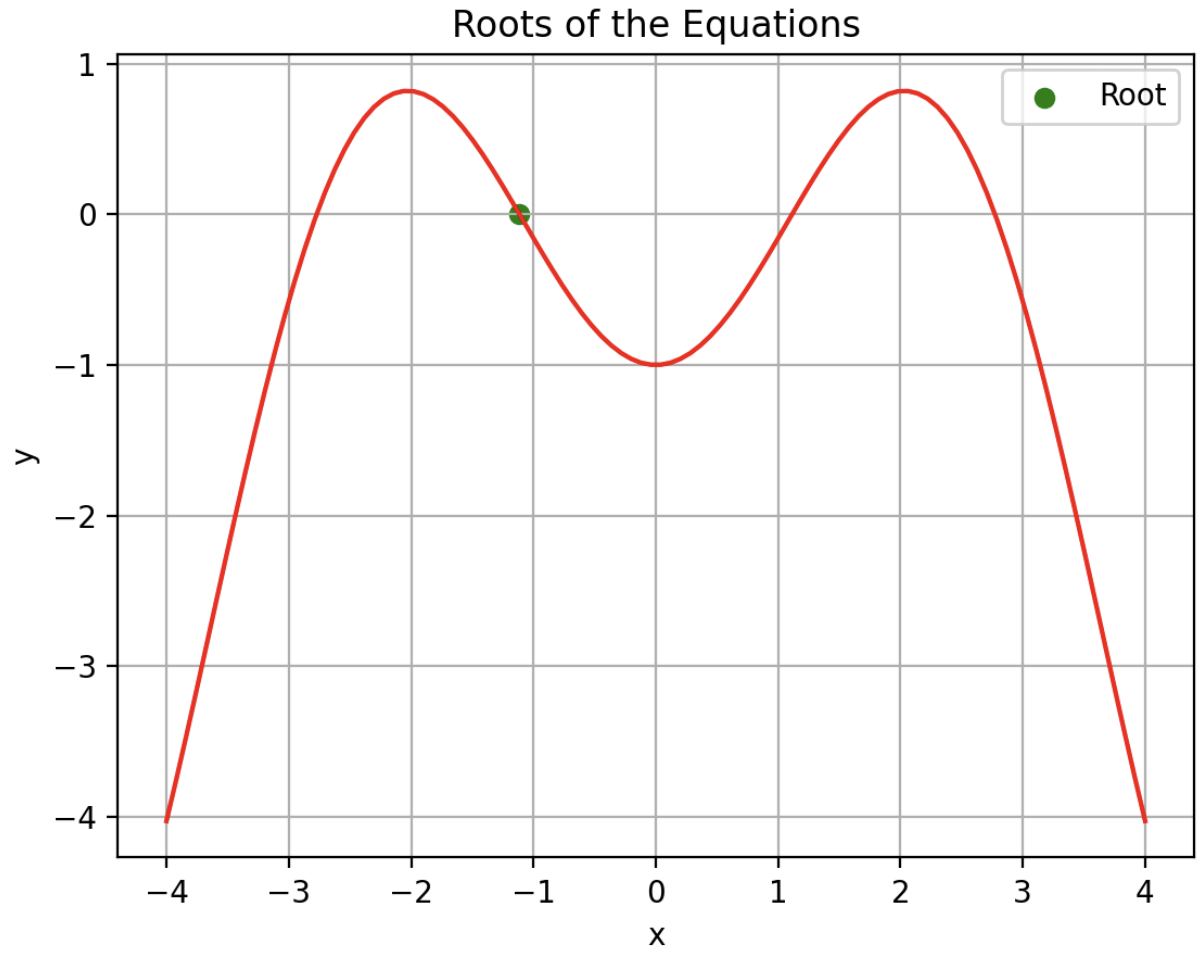



```
$ python newton_solver.py
Root found: 0.5671435594558716  in  24  steps
```

(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()
```

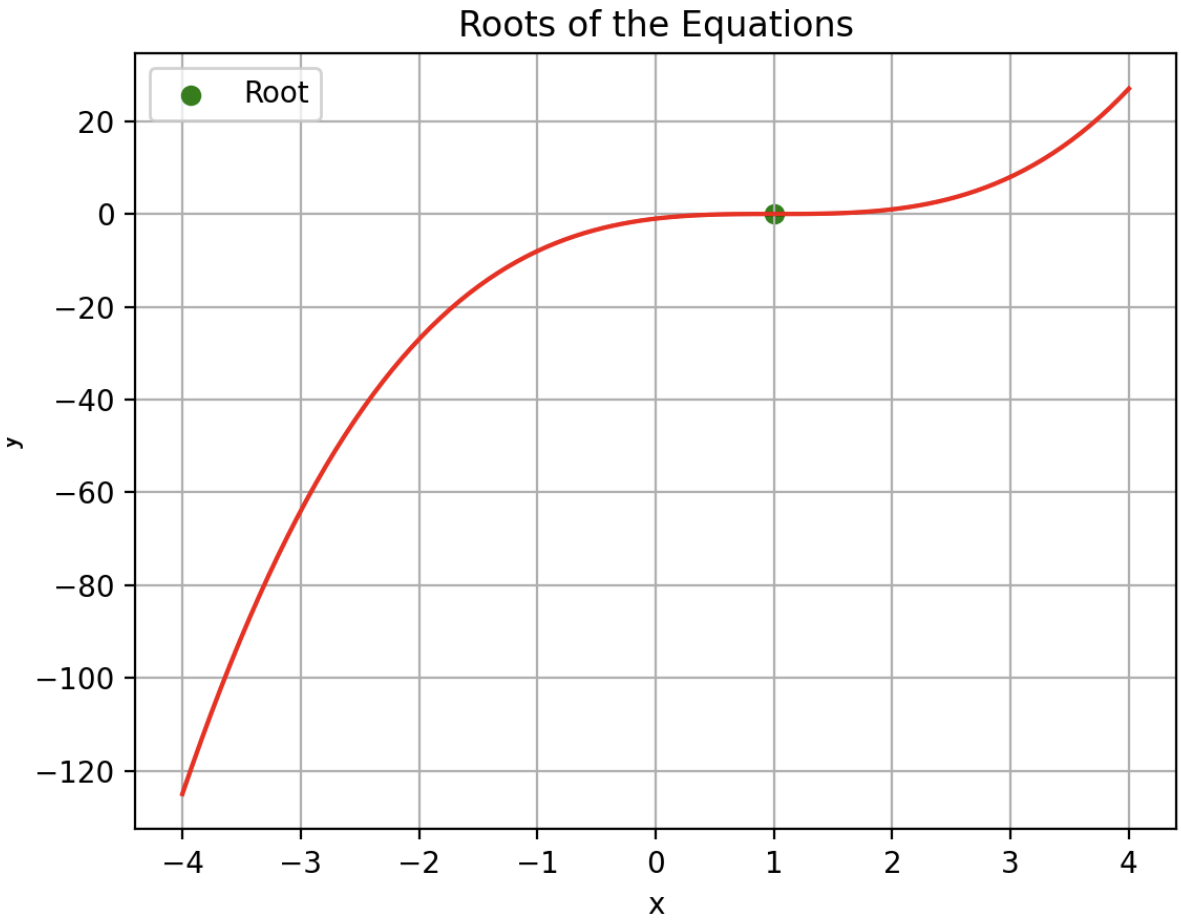



```
└─$ python newton_solver.py
Root found: -1.1141576766967773  in  20  steps
```

(d)

```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**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()
```




```
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

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

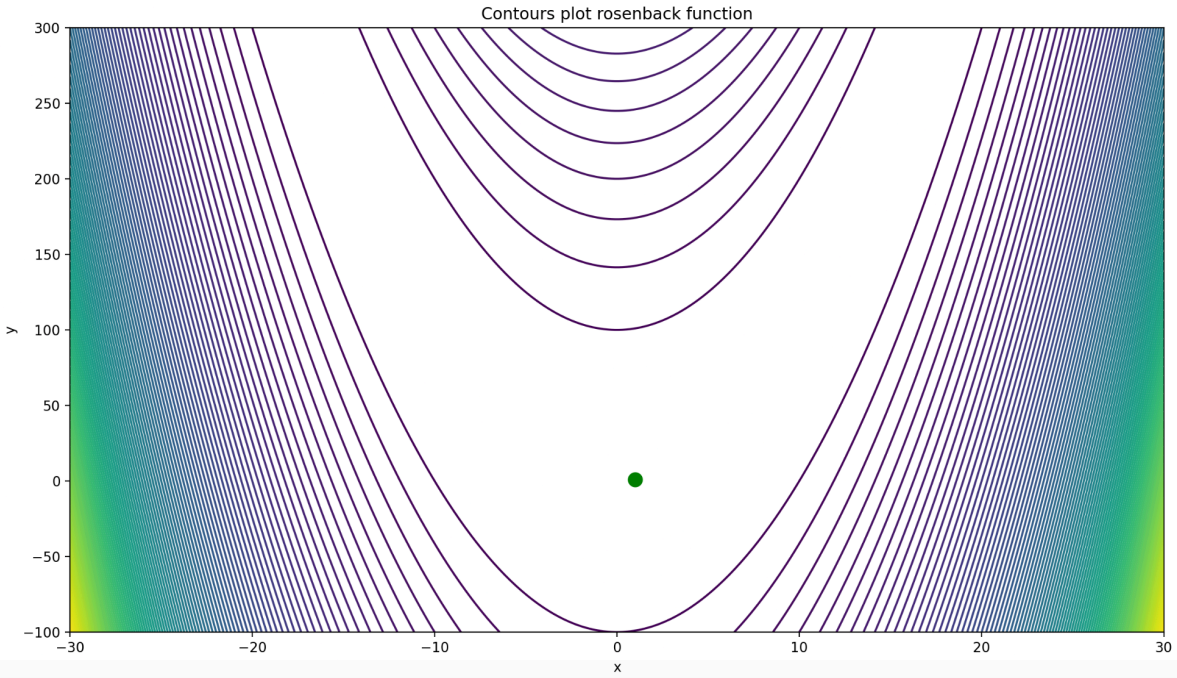
$$\Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 & -2 \\ 2 & 0 \end{bmatrix}^{-1} \times \bar{f}(0,1)$$

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

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

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

∴ After 1 iteration, its [0.5, 0.5]

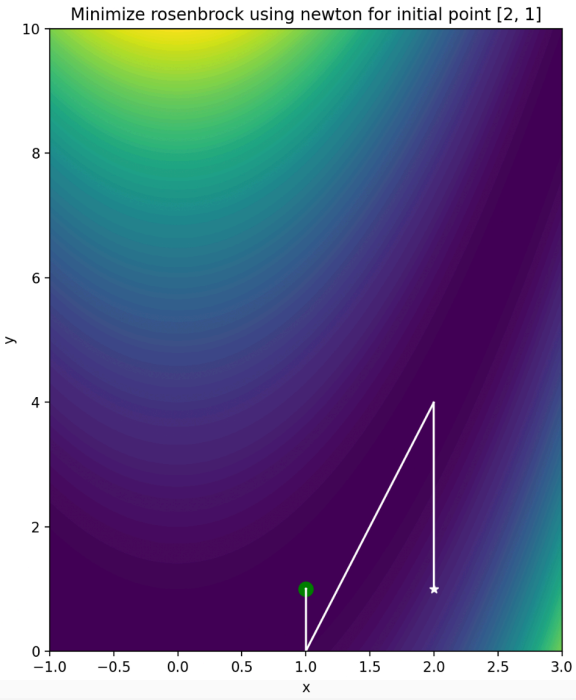


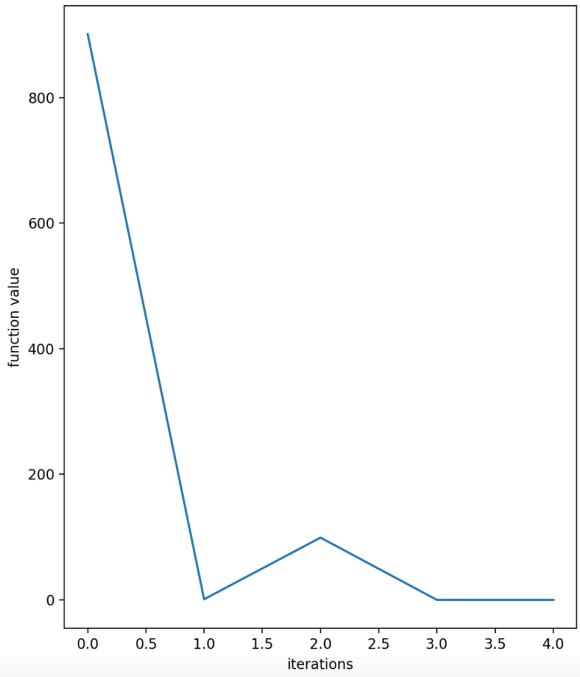

```
import matplotlib.pylab as plt
from matplotlib.pyplot import figure
import numpy as np

def rosenbrock(x, y):
   return 100 * ((y - (x ** 2)) ** 2) + ((1-x) ** 2)

def gradient(x, y):
   return np.array([-400 * x * (y - (x ** 2)) - 2 * (1-x), 200 * (y - (x ** 2))])

def hessian(x, y):
   return np.array([[(-400 * y) + (1200 * (x ** 2)) + 2, -400*x], [-400 * x, 200]])

def plotc(fun, xmin, xmax, ymin, ymax, contour=100, colour=True):
   x = np.linspace(xmin, xmax, 300)
   y = np.linspace(ymin, ymax, 300)
   X, Y = np.meshgrid(x, y)
   Z = fun(X, Y)
   if colour:
       plt.contourf(X, Y, Z, contour)
   else:
       plt.contour(X, Y, Z, contour)
   plt.scatter(1, 1, marker='o', s=100, color='g')
```

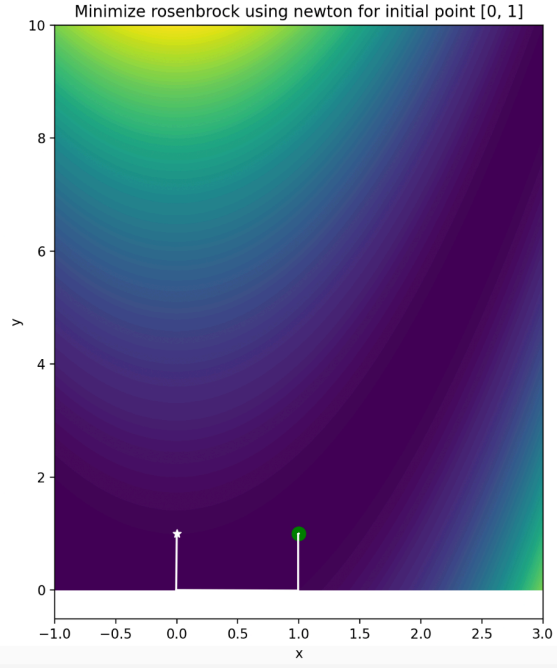
```
figure(figsize=(15, 8))
plotc(rosenbrock, -30,30, -100, 300, colour=False)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contours plot rosenback function")
# plt.show()
def newton(fun, gradient, hessian, init, tol=1e-5, max_iter=10000):
   a = init
   count = 0
   values = [a]
   f_prev = fun(a[0], a[1])
   f_values = [f_prev]
   while count < max_iter:
       # computer hessian and gradient
       print("current values ", a[0], a[1], f_prev)
       g = gradient(a[0], a[1])
       h = hessian(a[0], a[1])
       a = a - np.linalg.solve(h, g) # Let numpy handle the complex calculations
       current = fun(a[0], a[1])
       if np.isnan(current):
           break
       values.append(a)
       f_values.append(current)
       if abs(current - f_prev) < tol:
           # under tolerance hence end the algorithm
           break
       f_prev = current
       count = count + 1
   return np.array(values), np.array(f_values)
for elem in [[-1, 1], [0, 1], [2, 1]]:
   x_values, y_values = newton(rosenbrock, gradient, hessian, init=elem)
   figure(figsize=(15, 8))
   plt.subplot(1,2,1)
   plotc(rosenbrock, -1,3,0,10)
   plt.xlabel("x")
   plt.ylabel("y")
   title = "Minimize rosenbrock using newton for initial point " + str(elem)
   plt.title(title)
   plt.scatter(x_values[0,0],x_values[0,1],marker="*",color="w")
   for i in range(1,len(x_values)):
           plt.plot((x_values[i-1,0],x_values[i,0]), (x_values[i-1,1],x_values[i,1]) ,
"w")
```

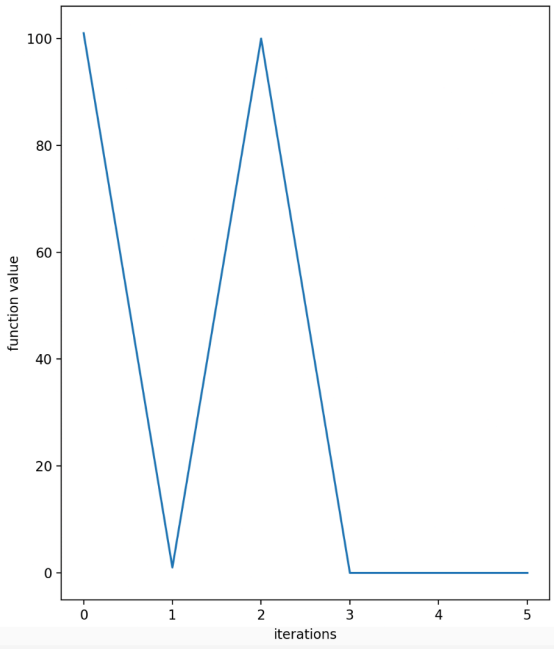
```
    plt.subplot(1,2,2)
    plt.plot(y_values)
    plt.xlabel("iterations")
    plt.ylabel("function value")
    print("DOne", x_values[-1])
plt.show()
```

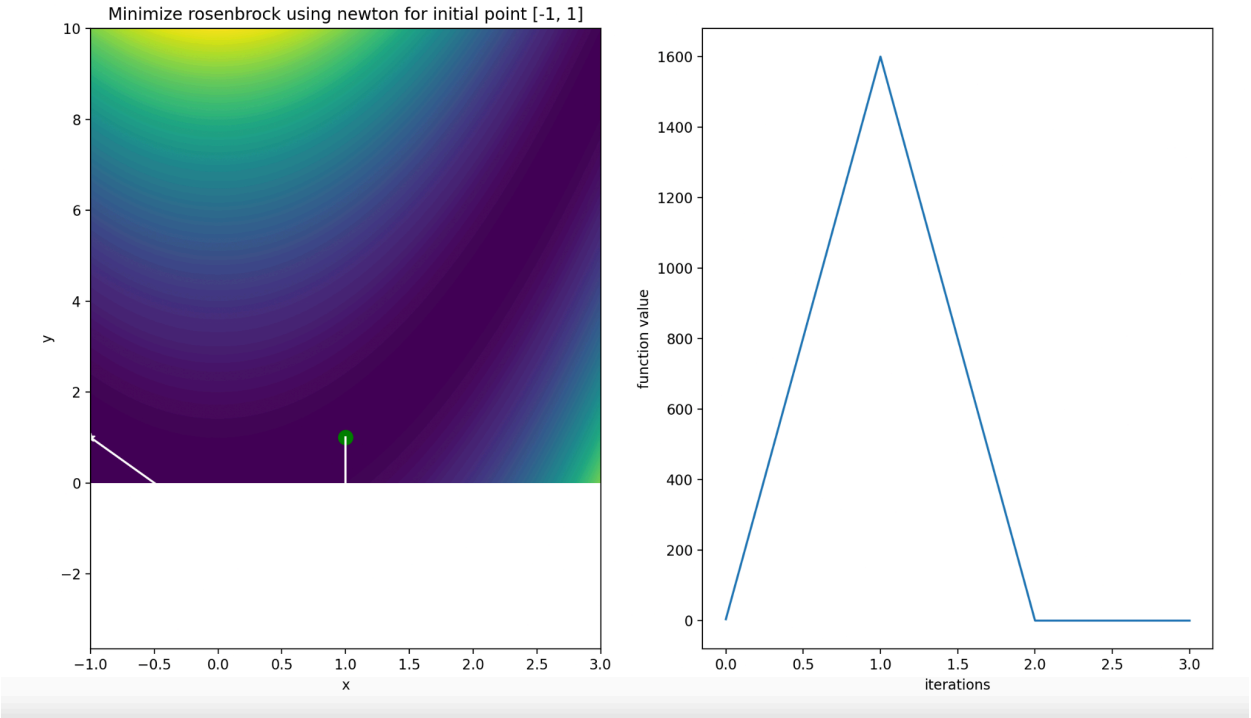
NEWTON’S METHOD PLOTS


Minimize rosenbrock using newton for initial point [2, 1]
x
y


function value
iterations


Minimize rosenbrock using newton for initial point [0, 1]
x
y


function value
iterations

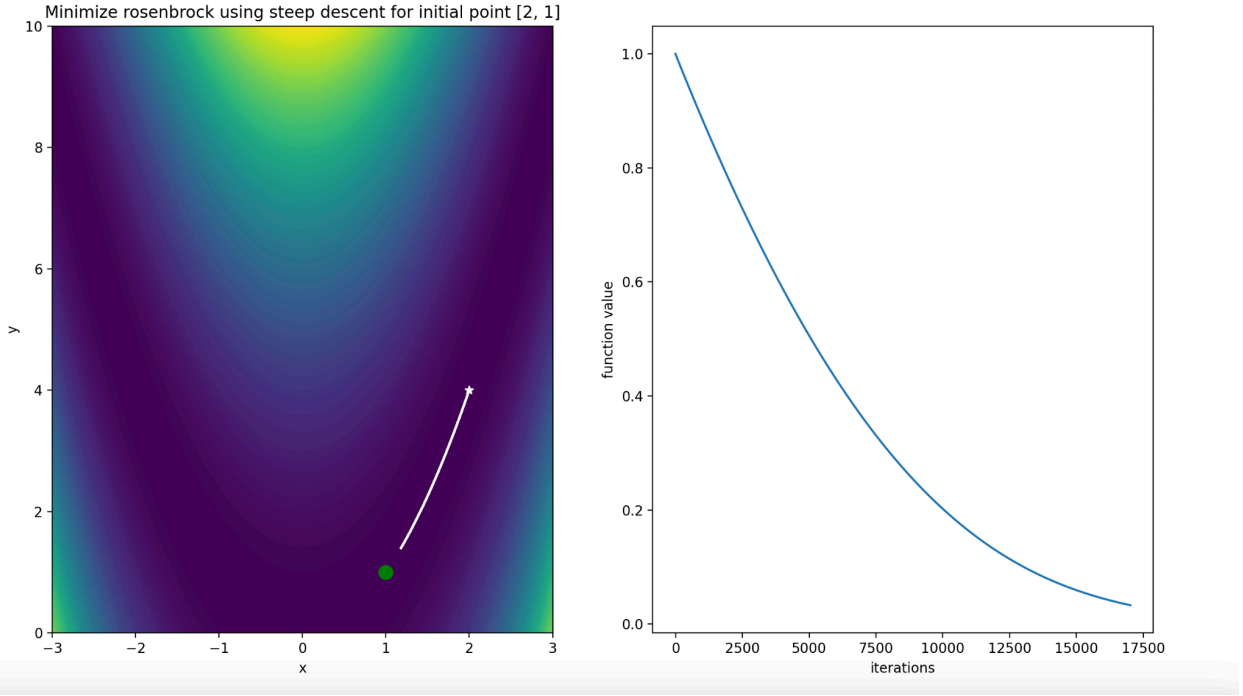


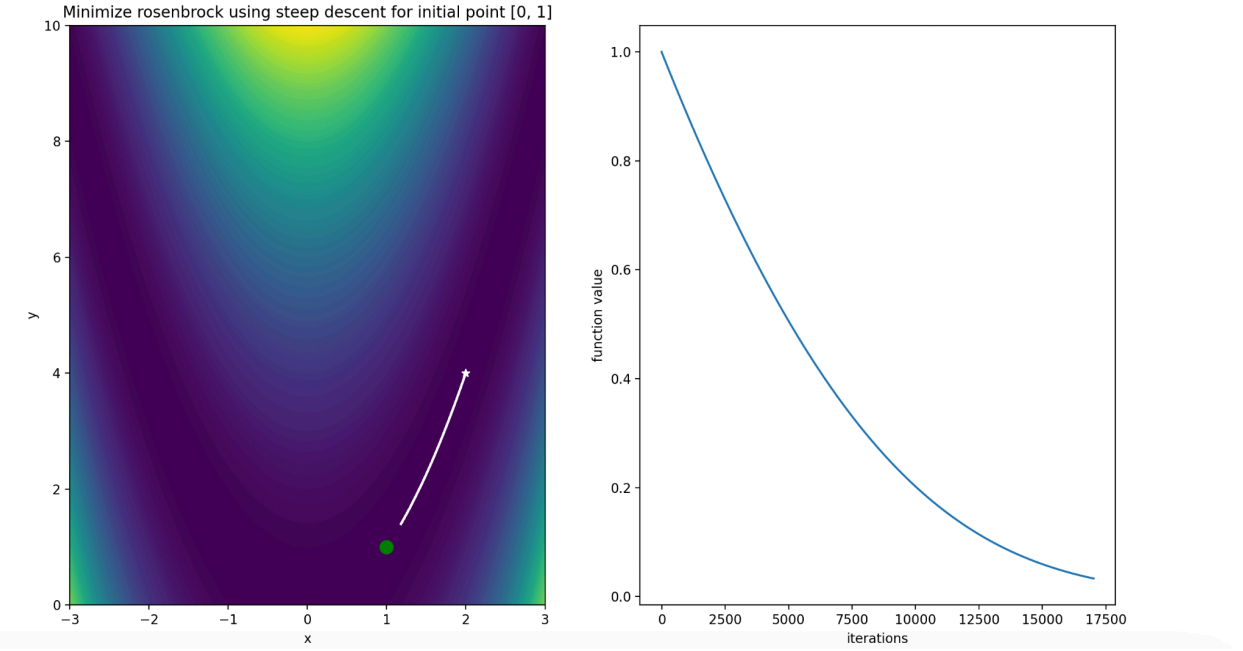
STEEP DESCENT CODE AND PLOT (have used different step size and max iterations for this)

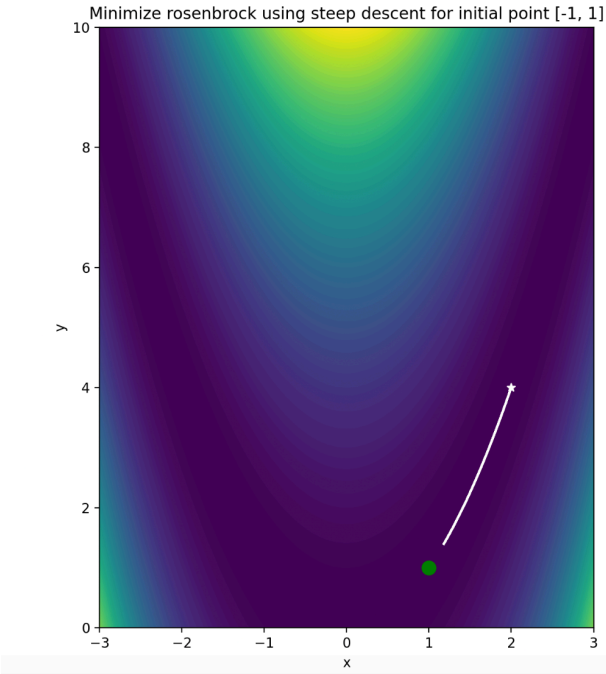
```
import matplotlib.pylab as plt
from matplotlib.pyplot import figure
import numpy as np

def rosenbrock(x, y):
   return 100 * ((y - (x ** 2)) ** 2) + ((1-x) ** 2)

def gradient(x, y):
   return np.array([-400 * x * (y - (x ** 2)) - 2 * (1-x), 200 * (y - (x ** 2))])

def hessian(x, y):
   return np.array([[(-400 * y) + (1200 * (x ** 2)) + 2, -400*x], [-400 * x, 200]])

def plotc(fun, xmin, xmax, ymin, ymax, contour=100, colour=True):
   x = np.linspace(xmin, xmax, 300)
   y = np.linspace(ymin, ymax, 300)
   X, Y = np.meshgrid(x, y)
   Z = fun(X, Y)
   if colour:
       plt.contourf(X, Y, Z, contour)
   else:
```

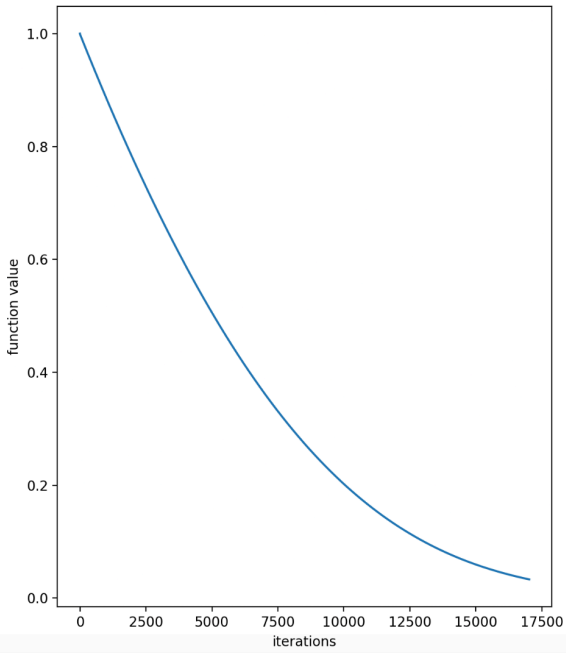
```
        plt.contour(X, Y, Z, contour)
    plt.scatter(1, 1, marker='o', s=100, color='g')

figure(figsize=(15, 8))
plotc(rosenbrock, -30,30, -100, 300, colour=False)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contours plot rosenback function")
# plt.show()
def steep(fun, gradient, hessian, init, tol=1e-5, maxiter=30000, steplength=0.0005):
    a = init
    count=0
    values = [a]
    f_prev = fun(a[0],a[1])
    f_values = [f_prev]
    while count < maxiter:
        print("current values ", a[0], a[1], f_prev)
        a = a - steplength*gradient(a[0],a[1])
        cur = fun(a[0], a[1])
        if np.isnan(cur):
            break
        values.append(a)
        f_values.append(cur)
        if abs(cur-f_prev)<tol:
            break
        f_prev = cur
        count += 1
    return np.array(values), np.array(f_values)
for elem in [[-1, 1], [0, 1], [2, 1]]:
    x_values, y_values = steep(rosenbrock, gradient, hessian, init=[2,4])
    figure(figsize=(15, 8))
    plt.subplot(1,2,1)
    plotc(rosenbrock, -3,3,0,10)
    plt.xlabel("x")
    plt.ylabel("y")
    title = "Minimize rosenbrock using steep descent for initial point " + str(elem)
    plt.title(title)
    plt.scatter(x_values[0,0],x_values[0,1],marker="*",color="w")
    for i in range(1,len(x_values)):
            plt.plot((x_values[i-1,0],x_values[i,0]), (x_values[i-1,1],x_values[i,1]) ,
"w")
    plt.subplot(1,2,2)
```

```
    plt.plot(y_values)
    plt.xlabel("iterations")
    plt.ylabel("function value")
    print("DOne")
plt.show()
```


Minimize rosenbrock using steep descent for initial point [2, 1]
x
y
function value
iterations


Minimize rosenbrock using steep descent for initial point [0, 1]
x
y
function value
iterations



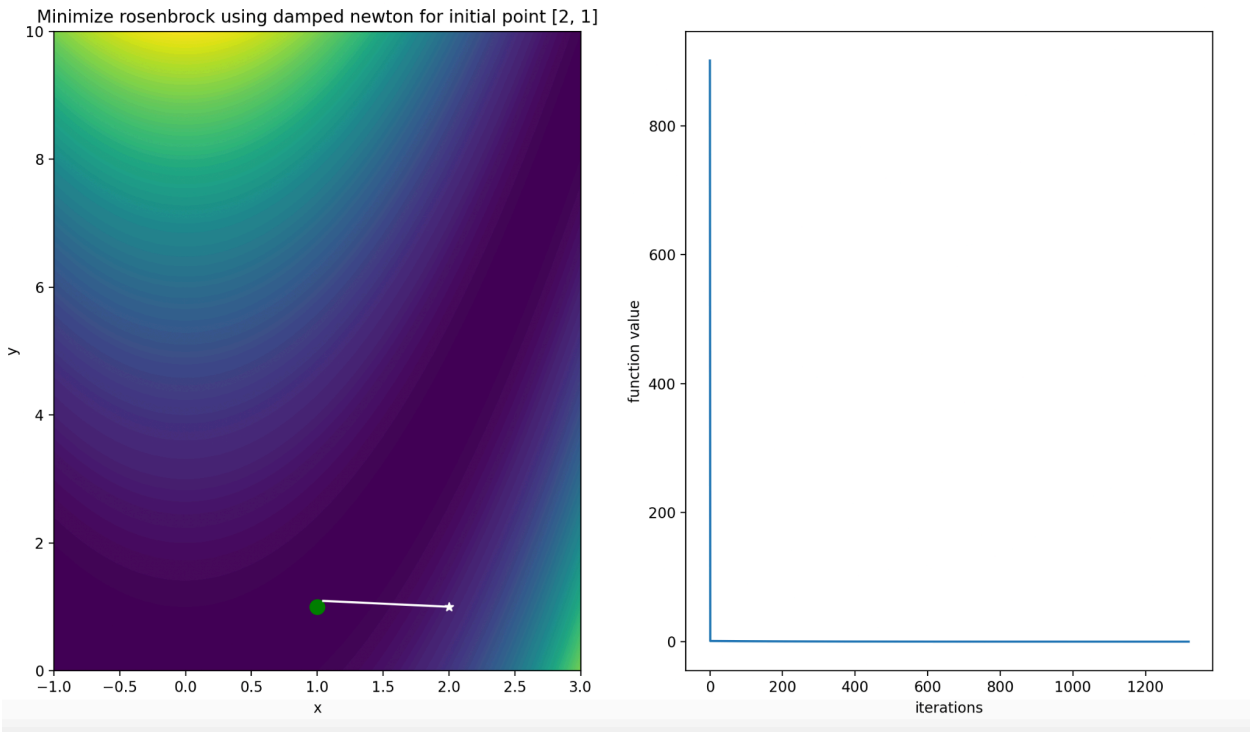


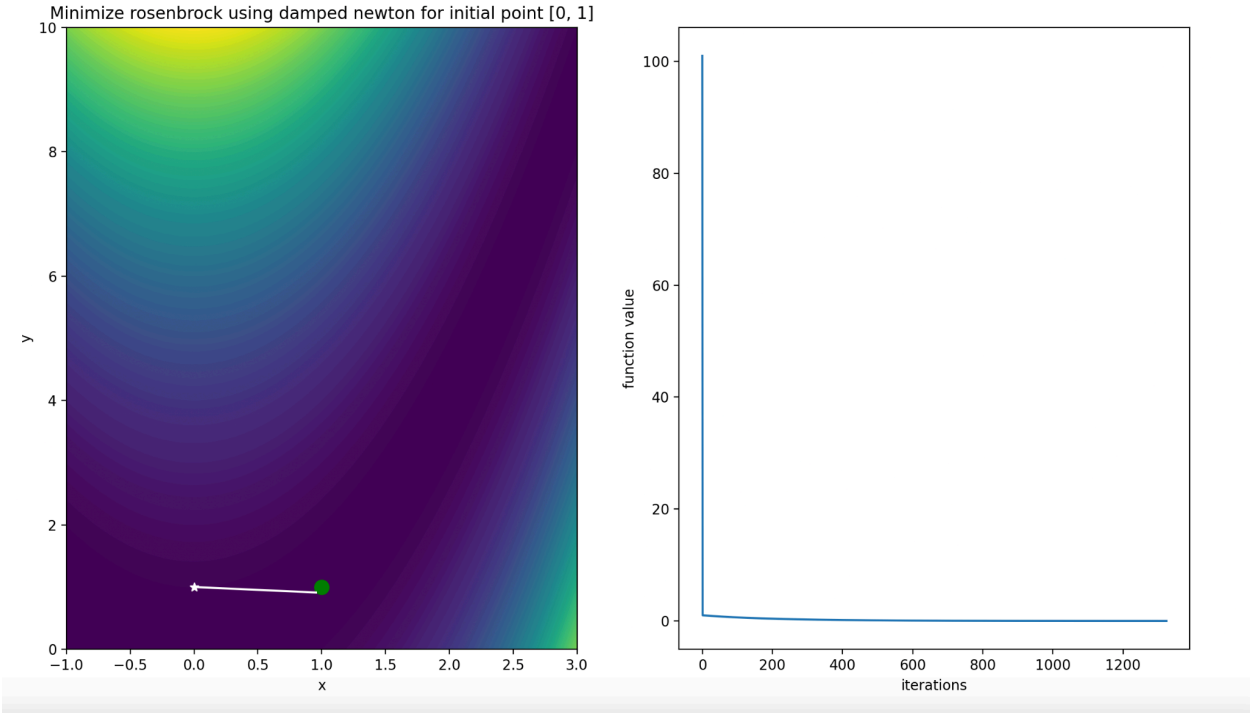
DAMPED NEWTON DESCENT (have changed max iter, alpha, beta)

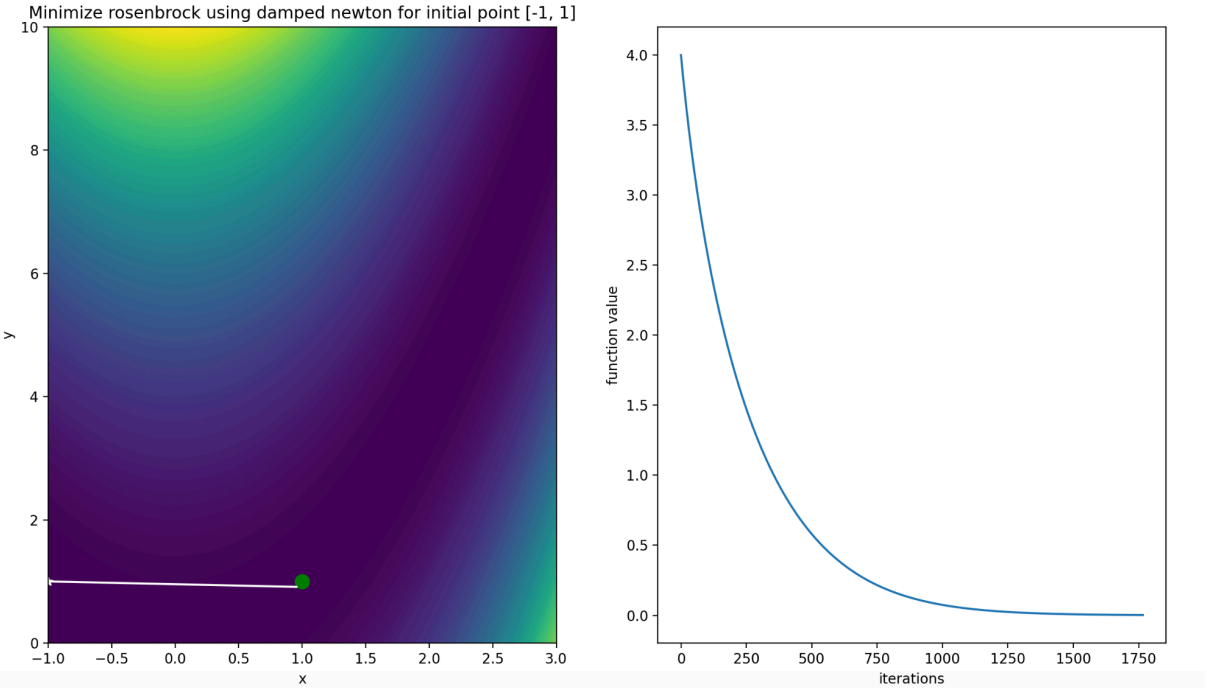
```
import matplotlib.pylab as plt
from matplotlib.pyplot import figure
import numpy as np

def rosenbrock(x, y):
   return 100 * ((y - (x ** 2)) ** 2) + ((1-x) ** 2)

def gradient(x, y):
   return np.array([-400 * x * (y - (x ** 2)) - 2 * (1-x), 200 * (y - (x ** 2))])

def hessian(x, y):
   return np.array([[(-400 * y) + (1200 * (x ** 2)) + 2, -400*x], [-400 * x, 200]])

def plotc(fun, xmin, xmax, ymin, ymax, contour=100, colour=True):
   x = np.linspace(xmin, xmax, 300)
   y = np.linspace(ymin, ymax, 300)
   X, Y = np.meshgrid(x, y)
   Z = fun(X, Y)
   if colour:
       plt.contourf(X, Y, Z, contour)
   else:
       plt.contour(X, Y, Z, contour)
```

```
   plt.scatter(1, 1, marker='o', s=100, color='g')

figure(figsize=(15, 8))
plotc(rosenbrock, -30,30, -100, 300, colour=False)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contours plot rosenback function")
def newton_line(fun, gradient, hessian, init, tol=1e-5, max_iter=10000, alpha=0.01,
beta=0.05):
   a = init
   t = 1 #step size
   count = 0
   values = [a]
   f_prev = fun(a[0], a[1])
   f_values = [f_prev]
   while count < max_iter:
       # computer hessian and gradient
       g = gradient(a[0], a[1])
       h = hessian(a[0], a[1])
       dir = -np.linalg.solve(h, g) # # Let numpy handle the complex calculations,
negative gradient direction
       val = a + t*dir
       while(fun(val[0], val[1]) > (fun(a[0], a[1]) + alpha * t * np.dot(g.T, dir))):
#alpha and beta for backtracking
           val = a + t*dir
           t *= beta
       a += t * dir # Update
       current = fun(a[0], a[1])
       if np.isnan(current):
           break
       values.append(a)
       f_values.append(current)
       if abs(current - f_prev) < tol:
           # under tolerance hence end the algorithm
           break
       f_prev = current
       count = count + 1
   return np.array(values), np.array(f_values)
for elem in [[-1, 1], [0, 1], [2, 1]]:
   x_values, y_values = newton_line(rosenbrock, gradient, hessian, init=elem)
   figure(figsize=(15, 8))
   plt.subplot(1,2,1)
```

```
   plotc(rosenbrock, -1,3,0,10)
   plt.xlabel("x")
   plt.ylabel("y")
   title = "Minimize rosenbrock using damped newton for initial point " + str(elem)
   plt.title(title)
   plt.scatter(x_values[0,0],x_values[0,1],marker="*",color="w")
   for i in range(1,len(x_values)):
           plt.plot((x_values[i-1,0],x_values[i,0]), (x_values[i-1,1],x_values[i,1]) ,
"w");
   plt.subplot(1,2,2)
   plt.plot(y_values)
   plt.xlabel("iterations")
   plt.ylabel("function value")
plt.show()
```


Minimize rosenbrock using damped newton for initial point [2, 1]
y
x
function value
iterations


Minimize rosenbrock using damped newton for initial point [0, 1]
y
x
function value
iterations


Minimize rosenbrock using damped newton for initial point [-1, 1]
10
8
6
4
2
0
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
function value
0
250
500
750
1000
1250
1500
1750
iterations

(10) Given

$f: R^n \to R$

$A \to n \times n$ symmetric positive definite

$b \to n \times 1$ vector

$c \to$ scalar

(1) $\leftarrow f(x) = \frac{1}{2} x^T A x - x^T b + C$

(a) Starting point x_0

$H_f(x_0) s_0 = A s_0$ [Because $\nabla f(x) = Ax - b$, $H_f(x) = A$]

$= b - A x_0$

$\Rightarrow -\nabla f(x_0)$

Since $x_1 = x_0 + s_0$

(11) $\leftarrow Ax_1 = A(x_0 + s_0) \Rightarrow Ax_0 + As_0 \Rightarrow (b - Ax_0) + As_0 = b$

Since we know $\nabla f(x) = Ax - b$

$\Rightarrow$ Hence $Ax = b$ is an inflection point

Since $H_f(x) = A$ [Positive Semidefinite]

$\therefore Ax = b$ is minima

Hence eq (11) directly gives us x^*

(b) To compute $\min_{\alpha} f(x_0 + \alpha s_0)$ where

$s_0 = -\nabla f(x) \Rightarrow b - Ax$

For min α we get the general formula

$$\alpha = \frac{s_k^T s_k}{s_k^T A s_k} \quad [\text{From Q9}] \rightarrow (III)$$

$x_0 - x^*$ is eigenvector of $A \Rightarrow$ Given

$$\lambda(x_0 - x^*) = A(x_0 - x^*) \Rightarrow Ax_0 - b \rightarrow (IV)$$

$\therefore A s_0 = \lambda A(x^* - x_0) \Rightarrow \lambda s_0$

Hence s_0 is also an eigenvector

Putting $As_0 \Rightarrow \lambda s_0$ in (III) we get

$$\alpha = \frac{s_0^T s_0}{s_0^T \lambda s_0} \Rightarrow \frac{1}{\lambda}$$

Hence $x_1 = x_0 - \frac{1}{\lambda} \nabla f(x_0)$

$$\Rightarrow x_0 + \frac{(b - Ax_0)}{\lambda}$$

From (IV)

$$x_0 - \frac{\lambda(x_0 - x^*)}{\lambda}$$

$\Rightarrow x^*$

Hence Steepest Descent Converges in one iteration

(8) Given $f: \mathbb{R}^2 \to \mathbb{R}$

$$f(x) = \frac{1}{2}(x_1^2 - x_2)^2 + \frac{1}{2}(1 - x_1)^2$$

(a) Clearly f is of the form $\frac{X^2}{2} + \frac{Y^2}{2}$

where $X, Y \in \mathbb{R}$

So min value of $X^2, Y^2 = 0$ {Both} $\to$ (1)

Since x_1, x_2 are independent {No constraints} we can choose them in such a way that (1) holds

$\therefore x_1 = 1, x_2 = 1$ are the values

$\Rightarrow f([1,1]) = \frac{1}{2}(1^2 - 1)^2 + \frac{1}{2}(1-1)^2 \Rightarrow 0$ is the minimum value

(b) Newton's Method

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

For multivariable

$$\bar{x}_{n+1} = \bar{x}_n - \left[J_f(\bar{x}_n)\right]^{-1} f(\bar{x}_n) \qquad \bar{x} = [x_1, x_2]$$

$$J_f(\bar{x}_n) = \left[\frac{\partial f(\bar{x})}{\partial x} \quad \frac{\partial f(\bar{x})}{\partial y}\right]^T$$

$$\Rightarrow \left[\frac{\partial f(\bar{x})}{\partial x_1} \quad \frac{\partial f(\bar{x})}{\partial x_2}\right]^T$$

$$\Rightarrow \left[(x_1^2 - x_2) \times 2x_1 + (1 - x_1)(-1), -(x_1^2 - x_2)\right]^T$$

$$\Rightarrow \left[2x_1^3 - 2x_1x_2 + x_1 - 1, x_2 - x_1^2\right]^T$$

$$H_f(x_1, x_2) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 6x_1^2 - 2x_2 + 1 & -2x_1 \\ -2x_1 & 1 \end{bmatrix}$$

Now, to solve

$$H_f(\bar{x}) s_0 = -\nabla f(x_0)$$

and $x_1 = x_0 + s_0$

$$\begin{bmatrix} 21 & -4 \\ -4 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = -\begin{bmatrix} 9 \\ -2 \end{bmatrix}$$

$$\therefore s_0 = \begin{bmatrix} -1/5 \\ 6/5 \end{bmatrix}$$

$$\Rightarrow x_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} -1/5 \\ 6/5 \end{bmatrix}$$

$\therefore$

$$x_1 \Rightarrow \begin{bmatrix} 9/5 \\ 16/5 \end{bmatrix}$$

(c) Since the function ≥ 0, newton iteration will be good if $f(x_{k+1}) < f(x_k)$

(d) Since $|x_1 - x^*| > |x_0 - x^*|$ it goes further away in this one step

(9) $\phi(x) = \frac{1}{2} x^T A x - b^T x + c$ → Given

$$A = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 3 \end{bmatrix} \leftarrow (1)$$

$$b = [1 \; 0 \; 0]^T$$

$$x_0 = [0 \; -1 \; 0]^T$$

(a) Gradient Direction g at x_0

$$\nabla f(x_0) = Ax_0 - b$$

$$H_f(x_0) = A \Rightarrow$$

Given the value of A at (1) we obtain the eigenvalues of 2, $2-\sqrt{3}$, $2+\sqrt{3}$ all +ve hence its positive definite

∴ Gradient is "curving upwards"

(b) To find

Step size α which minimizes $\phi(x_0 + \alpha d)$

$d = -g$

$\phi(x_0 + \alpha d)$

$$\Rightarrow \frac{1}{2}(x_0 + \alpha d)^T A(x_0 + \alpha d) - b^T(x_0 + \alpha d) + c$$

$$x_0 + \alpha d = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} + \alpha \begin{bmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -3 \end{bmatrix}$$

For choosing such an alpha we do

$$\frac{d\phi(x_1)}{d\alpha} \Rightarrow \nabla\phi(x_1)^T \frac{d}{d\alpha} x_1 = 0$$

$$\Rightarrow (Ax_1 - b)^T \frac{d}{d\alpha}(x_0 + \alpha d) = 0$$

Now, using residual direction

$$r_1 = b - Ax_0 \Rightarrow b - A(x_0 + \alpha d)$$

$$\therefore \alpha = -r_1^T d / d^T A d$$