```
import math
import numpy as np
from skimage import io
from matplotlib import pyplot as plt
```

1. Solve a system of linear equations that are supplied in the form of a matrix X and the output vector y. Should return the solution vector x.

Example: the following equations are represented as matrices

$2x + 3y = 8$

$5x - y = -2$

$$\begin{bmatrix} 2 & 3 \\ 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \end{bmatrix}$$
$$Ax = y$$

```
def solve_lin_eq(A, y):
    # your code here

    # A --> Coefficient matrix
    # B --> Output vector

    # Solve the system of linear equations
    x = np.linalg.solve(A, y)

    return x
```

```
A = np.array([[2, 3], [1, 2]])
b = np.array([8, 5])
x = solve_lin_eq(A, b)
print("Solution x:", x)
```

```
Solution x: [1. 2.]
```

```
A = np.array([[1, 2, 3], [0, 1, 4], [5, 6, 0]])
y = np.array([4, 5, 6])
x = solve_lin_eq(A, y)
print("Solution for Example 2:", x)
```

```
Solution for Example 2: [ 24. -19.   6.]
```

2. Find the Equation of a Plane from the given points in a 3D space

```
def is_collinear(point1, point2, point3):

    # Calculate vectors AB and AC
    AB = point2 - point1
    AC = point3 - point1

    print(AB)
    print(AC)

    # Check if the points are collinear
    is_collinear = True

    normal_vec = np.cross(AB, AC)
    if not normal_vec[0] == normal_vec[1] == normal_vec[2] == 0:   # Since sin θ = 0 for θ = 0° (collinear)
      is_collinear = False

    return is_collinear
```

```
'''def eq_plane_3D(point1, point2, point3):
    # your code here

    if not is_collinear(point1, point2, point3):
      #coeff_matrix = np.array([point1, point2, point3, np.ones(3)])
      #y = np.array([0, 0, 0, 1])

      coeff_matrix = np.array([point1, point2, point3])
```

```
        coeff_matrix = np.append(coeff_matrix, np.ones(3).reshape(3, 1), axis = 1)
        y = np.array([0, 0, 0])


        # Solve the system of linear equations
        solution = np.linalg.lstsq(coeff_matrix, y, rcond=None)[0]
        print('solution = ', solution)

        # Extract the coefficients of the equation of the plane ax +by +cz +d = 0
        a, b, c, d = solution
        d = -d

        return [a, b, c, d]
    return [None, None, None, None]'''
```

```
'def eq_plane_3D(point1, point2, point3):\n    # your code here\n\n    if not is_collinear(point1, point2, point3):\n      #coeff_matrix = np.ar
t2, point3, np.ones(3)])\n      #y = np.array([0, 0, 0, 1])\n\n      coeff_matrix = np.array([point1, point2, point3])\n      coeff_matrix = np.
ix, np.ones(3).reshape(3, 1), axis = 1)\n      y = np.array([0, 0, 0])\n\n\n      # Solve the system of linear equations\n      solution = np.li
matrix, y, rcond=None)[0]\n      print('solution = ', solution)\n\n      # Extract the coefficients of the equation of the plane ax +by +cz +d =
c, d = solution\n      d = -d\n\n      return [a, b, c, d]\n    return [None, None, None, None]'
```

```
'''
print(coeff_matrix)
print('\n', y, '\n')
np.linalg.lstsq(coeff_matrix, y, rcond=None)


Ans -

[[13. 22.  3.  1.]
 [ 4. 51.  6.  1.]
 [57.  8.  9.  1.]]

 [0 0 0]

(array([0., 0., 0., 0.]),
 array([], dtype=float64),
 3,
 array([66.14509684, 48.32725906,  1.14113769]))

 '''
```

```
'\nprint(coeff_matrix)\nprint('\n', y, '\n')\nnp.linalg.lstsq(coeff_matrix, y, rcond=None)\n\n\nAns - \n\n[[13. 22.  3.  1.]\n [ 4. 51.  6.  1.]
1.]]\n\n [0 0 0] \n\n(array([0., 0., 0., 0.]),\n array([], dtype=float64),\n 3,\n array([66.14509684, 48.32725906,  1.14113769]))\n \n '
```

```
def eq_plane_3D(point1, point2, point3):
    # your code here

    # Calculate vectors AB and AC
    AB = point2 - point1
    AC = point3 - point1

    normal_vec = np.cross(AB, AC)

    # Check if the points are collinear
    if not normal_vec[0] == normal_vec[1] == normal_vec[2] == 0:      # Since sin θ = 0 for θ = 0° (collinear)
      coeff_matrix = np.array([point1, point2, point3, np.ones(3)])
      y = np.array([0, 0, 0, 1])

      # Choose a reference point (x0, y0, z0)
      x0, y0, z0 = point1

      # Calculate D
      d = (normal_vec[0] * x0 + normal_vec[1] * y0 + normal_vec[2] * z0)

      # The equation of the plane is now determined
      a, b, c = normal_vec

      return a, b, c, d
    return None, None, None, None
```

```
# Given three points in 3D space
point1 = np.array([13, 22, 3], dtype=float)
point2 = np.array([4, 51, 6], dtype=float)
point3 = np.array([57, 8, 9], dtype=float)

sol = eq_plane_3D(point1, point2, point3)
```

```
print("Equation:")
print(f"{sol[0]}x + {sol[1]}y + {sol[2]}z = {sol[3]}")
```

```
Equation:
216.0x + 186.0y + -1150.0z = 3450.0
```

```
point1 = np.array([1, 22, 3], dtype=float)
point2 = np.array([2, 42, 6], dtype=float)
point3 = np.array([33, 6, 9], dtype=float)

sol = eq_plane_3D(point1, point2, point3)

print("Equation:")
print(f"{sol[0]}x + {sol[1]}y + {sol[2]}z = {sol[3]}")
```

```
Equation:
168.0x + 90.0y + -656.0z = 180.0
```

3. Define a linear transformation function that doubles the input vector

```
def linear_transformation(vector):
    # your code here

    # Double each component of the vector
    doubled_vector = 2 * vector

    return doubled_vector
```

```
# Apply the linear transformation to a vector
input_vector = np.array([3, 4])
output_vector = linear_transformation(input_vector)

print("Input Vector:", input_vector)
print("Output Vector:", output_vector)
```

```
Input Vector: [3 4]
Output Vector: [6 8]
```

```
input_vector = np.array([-1, 0.5])
output_vector = linear_transformation(input_vector)

print("Input Vector:", input_vector)
print("Output Vector:", output_vector)
```

```
Input Vector: [-1.   0.5]
Output Vector: [-2.  1.]
```

4. Apply a Matrix Transformation to an input vector

```
def tranform_matrix(matrix, input_vector):
    # your code here
    transformation_matrix = matrix
    # Apply the matrix transformation by multiplying the matrix and the vector
    resultant_vector = np.dot(transformation_matrix, input_vector)

    return resultant_vector
```

```
matrix = np.array([[2, 1],
                   [1, 2]])

input_vector = np.array([3, 4])

print(tranform_matrix(matrix, input_vector))
```

```
[10 11]
```

```
matrix = np.array([[2, 1, 0],
                  [0, 3, -1],
                  [1, 2, 1]])
```

```
input_vector = np.array([1, 2, 3])

print(tranform_matrix(matrix, input_vector))
```

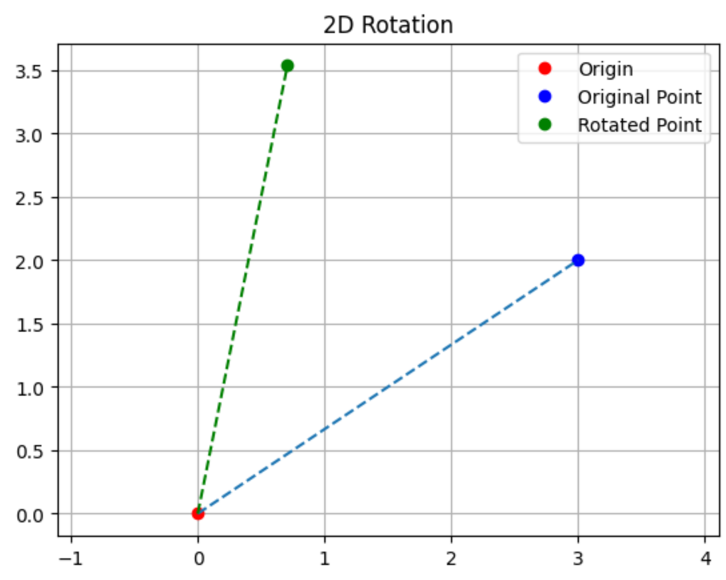
```
[4 3 8]
```

5. Write a function to apply a 2D rotation on a point

```
def rotation_2D(point, angle):
    # your code here

    # Compute sine and cosine values for the given angle
    sin_theta = np.sin(angle)
    cos_theta = np.cos(angle)

    # Create 2D rotation matrix
    # for positive angles/theta --> anti-clockwise rotation
    # for negative angles/theta --> clockwise rotation
    rotation_matrix = np.array([[cos_theta, -sin_theta],
                                [sin_theta, cos_theta]])

    # Apply rotation transformation to the point
    rotated_point = np.dot(rotation_matrix, point)

    return rotated_point
```

```
# Define an angle of rotation (in radians)
angle = np.pi / 4  # 45 degrees

# Define a point in 2D space
point = np.array([3, 2])

rotated_point = rotation_2D(point, angle)
print('Original:', point)
print('Rotated:', rotated_point)

# Plot the original and rotated points for visualization
plt.plot(0, 0, 'ro', label="Origin")
plt.plot(point[0], point[1], 'bo', label="Original Point")
plt.plot(rotated_point[0], rotated_point[1], 'go', label="Rotated Point")
plt.plot((0, point[0]), (0, point[1]), linestyle='dashed')
plt.plot((0, rotated_point[0]), (0, rotated_point[1]), 'g', linestyle='dashed')
plt.axis('equal')
plt.legend()
plt.title("2D Rotation")
plt.grid(True)
plt.show()
```

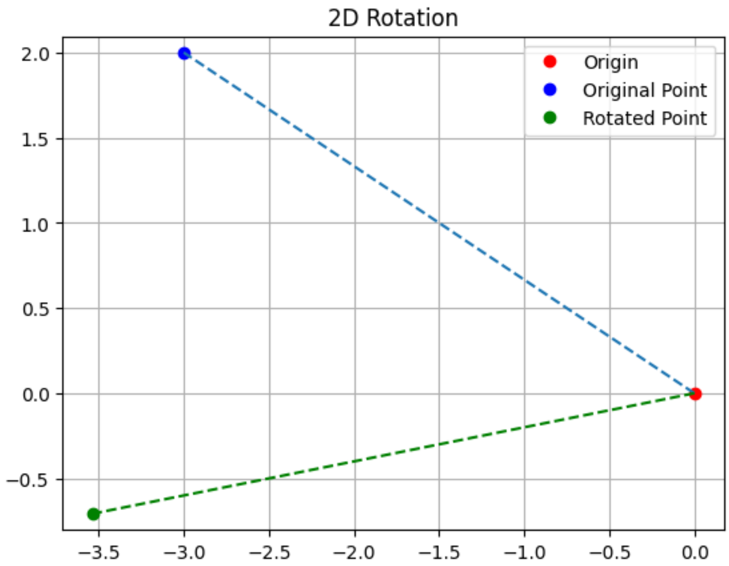
```
Original: [3 2]
Rotated: [0.70710678 3.53553391]
```




```
# Define an angle of rotation (in radians)
angle = np.pi / 4  # 45 degrees
```

```
# Define a point in 2D space
point = np.array([-3, 2])

rotated_point = rotation_2D(point, angle)
print('Original:', point)
print('Rotated:', rotated_point)

# Plot the original and rotated points for visualization
plt.plot(0, 0, 'ro', label="Origin")
plt.plot(point[0], point[1], 'bo', label="Original Point")
plt.plot(rotated_point[0], rotated_point[1], 'go', label="Rotated Point")
plt.plot((0, point[0]), (0, point[1]), linestyle='dashed')
plt.plot((0, rotated_point[0]), (0, rotated_point[1]), 'g', linestyle='dashed')
plt.axis('equal')
plt.legend()
plt.title("2D Rotation")
plt.grid(True)
plt.show()
```

```
Original: [-3  2]
Rotated: [-3.53553391 -0.70710678]
```

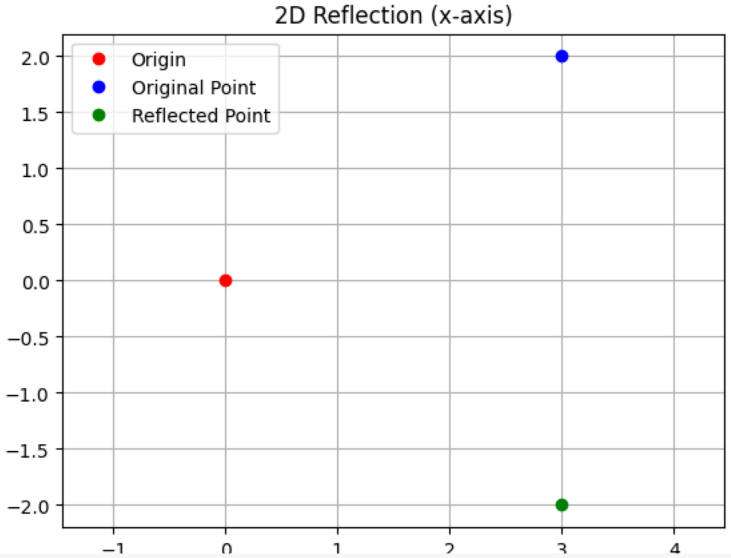


6. Write a function to apply a 2D reflection along the x-axis on a point

```
def reflection_2D(point, angle):
    # your code here

    # Create the 2D reflection matrix for the x-axis
    reflection_matrix = np.array([[1, 0],
                                  [0, -1]])

    # Apply the reflection transformation the point
    reflected_point = np.dot(reflection_matrix, point)

    return reflected_point
```

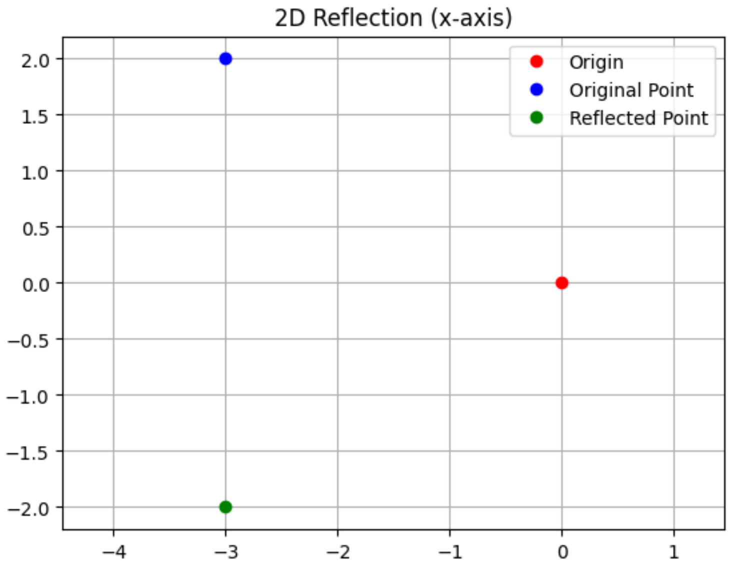
```
# Define a point in 2D space
point = np.array([3, 2])

reflected_point = reflection_2D(point, angle)
print('Original:', point)
print('Rotated:', reflected_point)

# Plot the original and reflected points for visualization
plt.plot(0, 0, 'ro', label="Origin")
plt.plot(point[0], point[1], 'bo', label="Original Point")
plt.plot(reflected_point[0], reflected_point[1], 'go', label="Reflected Point")
plt.axis('equal')
plt.legend()
plt.title("2D Reflection (x-axis)")
plt.grid(True)
plt.show()
```

```
Original: [3 2]
Rotated: [ 3 -2]
```




```
# Define a point in 2D space
point = np.array([-3, 2])

reflected_point = reflection_2D(point, angle)
print('Original:', point)
print('Rotated:', reflected_point)

# Plot the original and reflected points for visualization
plt.plot(0, 0, 'ro', label="Origin")
plt.plot(point[0], point[1], 'bo', label="Original Point")
plt.plot(reflected_point[0], reflected_point[1], 'go', label="Reflected Point")
plt.axis('equal')
plt.legend()
plt.title("2D Reflection (x-axis)")
plt.grid(True)
plt.show()
```

```
Original: [-3  2]
Rotated: [-3 -2]
```



7. Write a function to rotate an image by a given angle using only numpy functions

~~PIL.rotate()~~

~~cv2.rotate()~~

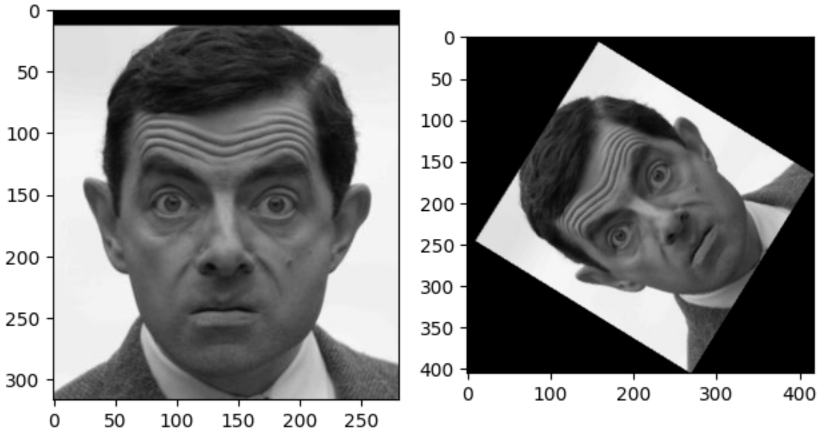
~~skimage.transform.rotate()~~

~~imageio.mimwrite()~~

😖

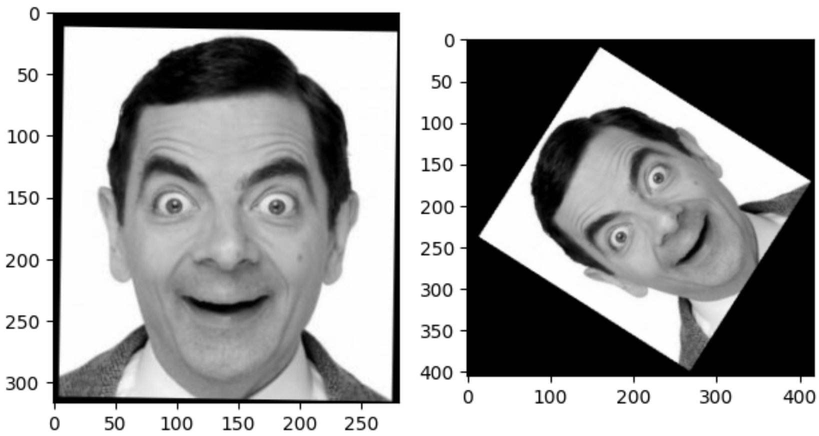
```
def rotate_image(image, angle):
  if len(image.shape) == 2:     # i.e only grayscale image

      # Compute sine and cosine values for the given angle
      sin_theta = np.sin(angle)
      cos_theta = np.cos(angle)

      # Get the image dimensions
      height, width = image.shape[:2]

      # Calculate the new image dimensions
      new_width = int(width * np.abs(cos_theta) + height * np.abs(sin_theta))
      new_height = int(width * np.abs(sin_theta) + height * np.abs(cos_theta))

      # Create an empty image with the new dimensions
      rotated_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)

      # Calculate the center of the new image
      center_x = new_width // 2
      center_y = new_height // 2

      # Calculate the center of the original image
      original_center_x = width // 2
      original_center_y = height // 2

      # Apply the rotation to each pixel in the new image
      for y in range(new_height):
          for x in range(new_width):
              # Calculate the corresponding position in the original image

              point =  np.array([x - center_x, y - center_y])
              original_x, original_y = map(int, rotation_2D(point, angle))
              original_x += original_center_x
              original_y += original_center_y

              # Check if the corresponding position is within the bounds of the original image
              if 0 <= original_x < width and 0 <= original_y < height:
                  # Copy the pixel value from the original image to the new image
                  rotated_image[y, x] = image[original_y, original_x]

      return rotated_image

  else:
    try:
      raise Exception('The image provided is not grayscale')
    except Exception as e:
      print(e)
```

```
image = io.imread('bean1.jpg') # replace with your own grayscale image here
img_transformed = rotate_image(image, 45)

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(image, cmap='gray')
ax2.imshow(img_transformed, cmap='gray')
fig.tight_layout()
fig.show()
```




```
image = io.imread('bean2.jpg') # replace with your own grayscale image here
img_transformed = rotate_image(image, 45)

fig, (ax1, ax2) = plt.subplots(1, 2)
```

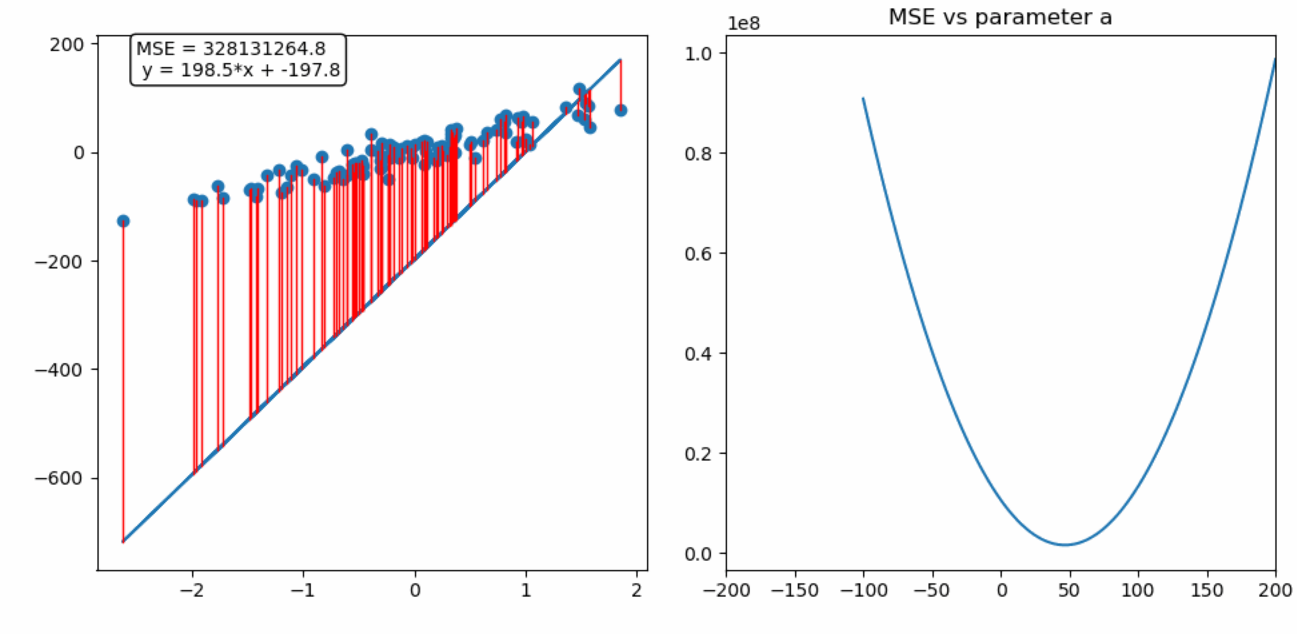
```
ax1.imshow(image, cmap='gray')
ax2.imshow(img_transformed, cmap='gray')
fig.tight_layout()
fig.show()
```



8. Implement Gradient Descent to find the root of a function.

The Gradient Descent method in an iterative process that involves starting with an initial guess of x_i, and updating it to x_{i+1} by moving along the gradient $f'(x_i)$ of the function and tempering the shift with a learning rate lr.

$$x_{i+1} = x_i - lr\frac{d(f(x_i))}{dx}$$




```
# Gradient Descent to find the minima
def GradDes(func, derivative, x0, alpha, max_iterations):
    # your code here
    x = x0
    tolerance = 1e-6    # Tolerance for convergence

    for i in range(max_iterations):
        gradient_x = gradient(x)
        x -= alpha * gradient_x
        if abs(gradient_x) < tolerance:
            break
    return x, func(x)
```

```
# Define the objective function
def f(x):
    return (1 - x) ** 2 + 100 * (x - x ** 2) ** 2
```

```
# Define the gradient of the objective function
def gradient(x):
    return 2 * (200 * x ** 3 - 200 * x ** 2 - x + 1)

# Test the univariate Rosenbrock function
x0 = 2.0  # Initial guess
alpha = 0.0001  # Reduced learning rate
max_iterations = 10000  # Increased maximum iterations
min_x, min_value = GradDes(f, gradient, x0, alpha, max_iterations)

print(f"Minimum x: {min_x}")
print(f"Minimum value: {min_value}")
```

```
Minimum x: 1.0000000023875453
Minimum value: 5.757376152989386e-16
```

```
# Define the objective function
def f(x):
    return 3 * x**4 - 4 * x**3 - 12 * x**2 + 3

# Define the gradient of the objective function
def gradient(x):
    return 12 * x**3 - 12 * x**2 - 24 * x

# Test the univariate Rosenbrock function
x0 = 0.5  # Initial guess
alpha = 0.0001  # Reduced learning rate
max_iterations = 10000  # Increased maximum iterations
min_x, min_value = GradDes(f, gradient, x0, alpha, max_iterations)

print(f"Minimum x: {min_x}")
print(f"Minimum value: {min_value}")
```

```
Minimum x: 1.9999999862130597
Minimum value: -28.999999999999993
```