Bangladesh University of Business and Technology

Lab report - Mid All Lab

Course name :Computer Graphics Lab
Course code: CSE 342

Submitted by

Name: Md.Al-Amin
Id: 22234103066
Section : 5
Intake : 50

Submitted to

Fateha Jannat Ayrin
Lecturer
Dept of CSE BUBT

Lab:1

Lab Topic : Implementation of DDA Line Drawing Algorithm in Python with also given the following lab task:

1. Take user input for starting and ending points and draw a line using DDA.

2. Test the program with different types of lines:
- Horizontal
- Vertical
- Diagonal
- Steep and shallow slopes

3. Add options to change the color or thickness of the line.

1.Introduction

In computer graphics, line drawing algorithms are essential for rendering straight lines on raster displays. The Digital Differential Analyzer (DDA) algorithm is one of the most widely used techniques for generating lines by calculating intermediate points between two endpoints.

It uses incremental calculations based on the slope of the line, making it efficient and easy to implement.

This experiment focuses on applying the DDA algorithm in Python to draw lines for various cases, such as horizontal, vertical, diagonal, steep, and shallow slopes.

Additionally, customization features like color and thickness are integrated to enhance the visual representation of the lines.

2.Objectives

1. To implement the DDA line drawing algorithm in Python.
2. To take user input for starting and ending coordinates of a line.
3. To test the algorithm with different types of lines, including:
    - Horizontal lines
    - Vertical lines
    - Diagonal lines
    - Steep and shallow slopes
4. To provide options for changing the color and thickness of the drawn lines.

5. To understand the working principle of incremental point generation based on the line slope.

3. Algorithm Explanation

The DDA algorithm works by calculating either x or y depending on the line's slope and incrementally plotting the intermediate points between the start and end coordinates.

4. DDA Algorithm Steps

1. Input the starting point (x1, y1) and ending point (x2, y2).
2. Calculate dx = x2 - x1 and dy = y2 - y1.
3. Calculate slope m = dy / dx (handle dx = 0 case separately).
4. If |m|<=1:
- Increment x by 1 each step.
- Compute y = y + m each time.
- Stop when x > x2.
5. Else:
- Increment y by 1 each step.
- Compute x = x + 1/m each time.
- Stop when y > y2.
6. Round and plot each (x, y) value as a pixel.

All the 3 task solution:

I want to give solution of all the 3 tasks in one code.
Here is code screenshot from my jupyter notebook

```python
import matplotlib.pyplot as plt

def dda_line(x1, y1, x2, y2, color='blue', thickness=1):
    dx = x2 - x1
    dy = y2 - y1
    steps = int(max(abs(dx), abs(dy)))
    x_inc = dx / steps
    y_inc = dy / steps
    x = x1
    y = y1

    x_points = []
    y_points = []

    for _ in range(steps + 1):
        x_points.append(round(x))
        y_points.append(round(y))
        x += x_inc
        y += y_inc

    plt.plot(x_points, y_points, marker='o', color=color, linewidth=thickness,
             label=f"({x1},{y1})→({x2},{y2})")

# Menu
print("Choose mode:")
print("1. Draw line with user input applying DDA")
print("2. Test with different types of lines like Horizontal ,Vertical , Diagonal ,Steep and shallow slopes")
choice = int(input("Enter choice: "))
```

```python
if choice == 1:
    x1 = int(input("Enter x1: "))
    y1 = int(input("Enter y1: "))
    x2 = int(input("Enter x2: "))
    y2 = int(input("Enter y2: "))
    color = input("Enter line color (e.g., red, blue, green): ").strip()
    thickness = float(input("Enter line thickness (e.g., 1, 2, 3): "))
    dda_line(x1, y1, x2, y2, color, thickness)

elif choice == 2:
    # Test cases with custom colors & thickness
    dda_line(2, 5, 10, 5, color='red', thickness=2)     # Horizontal
    dda_line(5, 2, 5, 10, color='blue', thickness=2)    # Vertical
    dda_line(2, 2, 8, 8, color='green', thickness=2)    # Diagonal
    dda_line(3, 1, 5, 8, color='purple', thickness=2)   # Steep
    dda_line(1, 3, 8, 5, color='orange', thickness=2)   # Shallow

# Plot results
plt.title("DDA Line Drawing")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.grid(True)
plt.show()
```

Program code description:

1. Drawing a Custom Line (User-Defined Mode)

What happens in `choice == 1` block

- User enters starting and ending coordinates (`x1, y1, x2, y2`).
- User chooses:
  - Color (e.g., `"red"`, `"blue"`, `"green"` — any Matplotlib color).
  - Thickness (e.g., 1, `2.5` — controls line width).
- These inputs are passed into `dda_line()` which:
  - Calculates steps = maximum of |dx| and |dy| → ensures consistent plotting for all slopes.
  - Finds increments (`x_inc`, `y_inc`) → amount to move in each axis per step.
  - Generates all intermediate points using the DDA algorithm.
  - Rounds each point to simulate raster pixels.
  - Plots the line on a graph with the selected style.

Outcome:

A single custom line is drawn exactly as the user specifies.

2. Drawing Predefined Test Lines

What happens in `choice == 2` block

- The program automatically calls `dda_line()` five times with:
    1. Horizontal line: `(2, 5)` → `(10, 5)`
        - Slope = 0, moves purely along X-axis.
    2. Vertical line: `(5, 2)` → `(5, 10)`
        - Infinite slope, moves purely along Y-axis.
    3. Diagonal line: `(2, 2)` → `(8, 8)`
        - Slope = 1, perfectly diagonal.
    4. Steep slope line: `(3, 1)` → `(5, 8)`
        - Slope > 1, moves more vertically than horizontally.
    5. Shallow slope line: `(1, 3)` → `(8, 5)`
        - Slope < 1, moves more horizontally than vertically.
- Each line is given its own color and thickness, so you can see them clearly.

Outcome:
All five lines appear on one plot — a quick way to test the DDA algorithm on different slopes and directions.

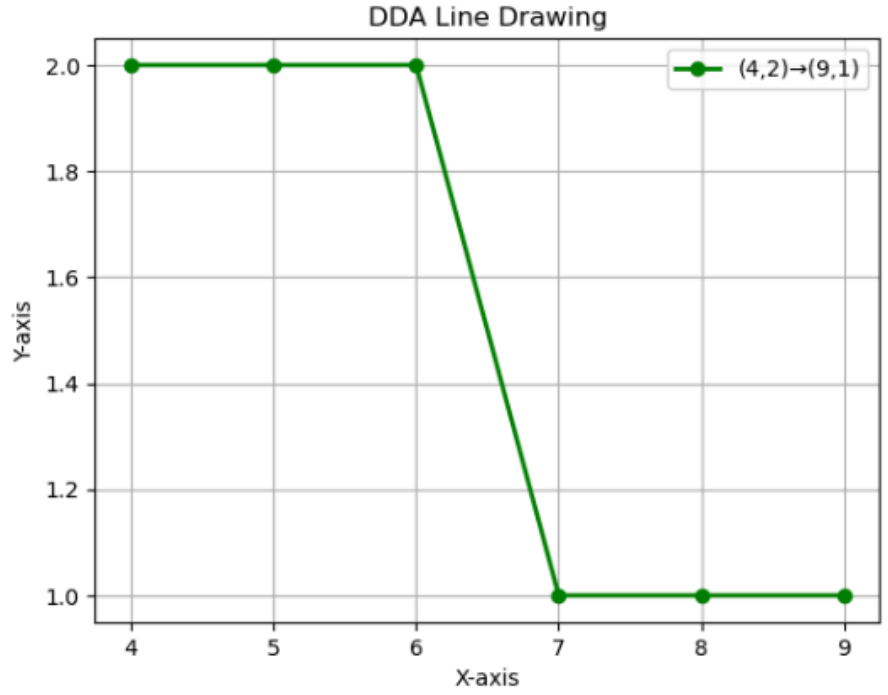
3. Handling the Plotting & Visualization

Regardless of whether the user chose custom mode or test mode:

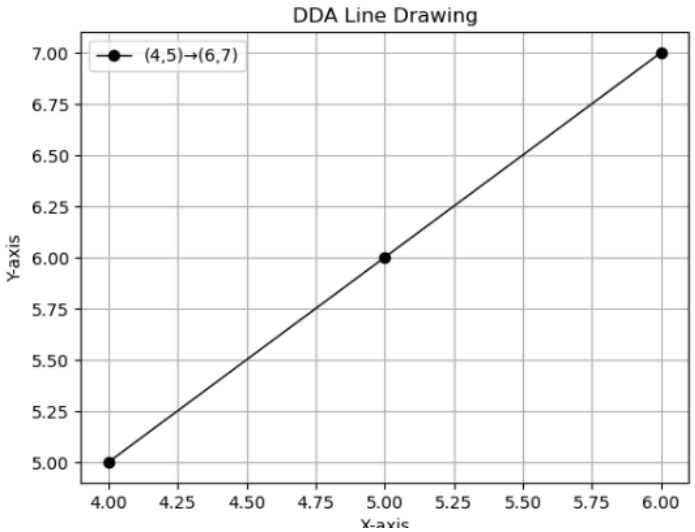
- `plt.title()` sets the chart title.
- `plt.xlabel()` and `plt.ylabel()` label the axes.
- `plt.legend()` creates a label for each line, showing start → end coordinates.
- `plt.grid(True)` shows a background grid (like raster pixels).
- `plt.show()` finally renders the plot with all drawn lines.

Output : if(choice == 1)then output

```
Choose mode:
1. Draw line with user input applying DDA
2. Test with different types of lines like Horizontal ,Vertical , Diagonal ,Steep and shallow slopes
Enter choice:  1
Enter x1:  4
Enter y1:  2
Enter x2:  9
Enter y2:  1
Enter line color (e.g., red, blue, green):  green
Enter line thickness (e.g., 1, 2, 3):  2
```



```
Choose mode:
1. Draw line with user input applying DDA
2. Test with different types of lines like Horizontal ,Vertical , Diagonal ,Steep and shallow slopes
Enter choice:  1
Enter x1:  4
Enter y1:  5
Enter x2:  6
Enter y2:  7
Enter line color (e.g., red, blue, green):  black
Enter line thickness (e.g., 1, 2, 3):  1
```

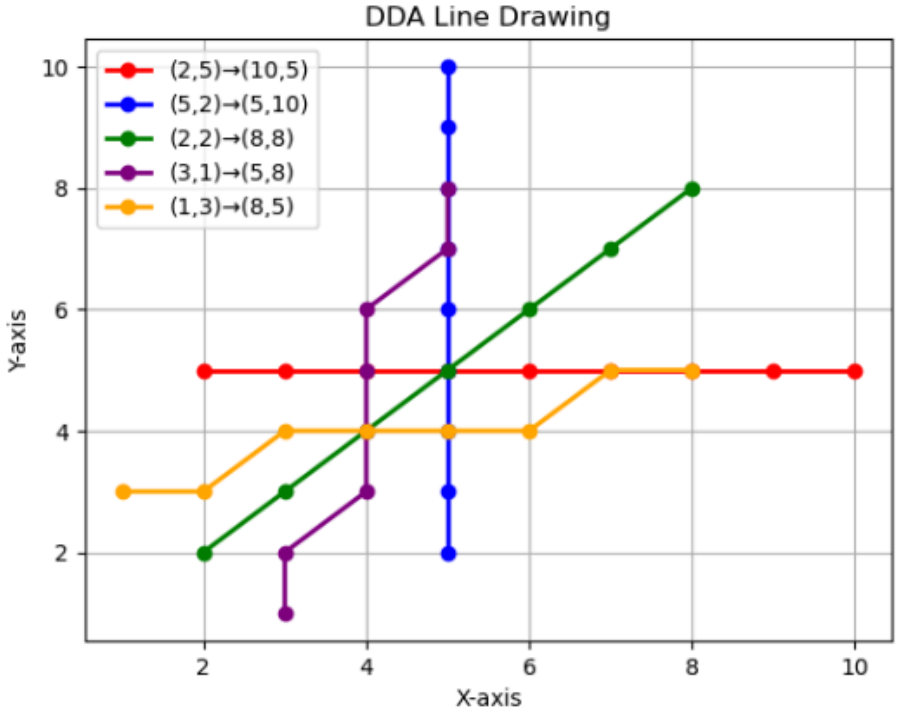


if(choice == 2)then output

Here test with different line output is given below.

Red line is horizontal,blue is vertical,green is diagonal,purple is steep,orange is shallow.

```
Choose mode:
1. Draw line with user input applying DDA
2. Test with different types of lines like Horizontal ,Vertical ,Diagonal ,Steep and shallow slopes
Enter choice:  2
```



Practice Exercises (Home Tasks)

1. Modify the DDA program to draw a triangle by connecting three vertices..
2. Plot points using integer-only rounding logic (no round() function) and observe differences.
3. Draw a grid and visualize line drawing over it using matplotlib.

Solution of home task(code):

Task 1 and task 3 is done in below code here task 3 is only to Draw a grid and visualize line drawing over it using matplotlib

Code in next page

```python
import matplotlib.pyplot as plt

# DDA line drawing function
def dda_line(x1, y1, x2, y2, color='blue'):
    dx = x2 - x1
    dy = y2 - y1

    m = float('inf') if dx == 0 else dy / dx

    x_points, y_points = [], []

    if abs(m) <= 1:  # Move in x-direction
        if x1 > x2:
            x1, y1, x2, y2 = x2, y2, x1, y1
        x, y = x1, y1
        while x <= x2:
            x_points.append(round(x))
            y_points.append(round(y))
            x += 1
            y += m
    else:  # Move in y-direction
        if y1 > y2:
            x1, y1, x2, y2 = x2, y2, x1, y1
        x, y = x1, y1
        while y <= y2:
            x_points.append(round(x))
            y_points.append(round(y))
            y += 1
            x += 1/m

    plt.plot(x_points, y_points, marker='o', color=color)

# Function to draw a triangle with grid
def draw_triangle_with_grid(v1, v2, v3):
    plt.figure()

    # Draw grid background
    plt.grid(True, which='both', color='gray', linestyle='--', linewidth=0.5)
    plt.xticks(range(0, 21))
    plt.yticks(range(0, 21))

    # Draw triangle edges
    dda_line(v1[0], v1[1], v2[0], v2[1], color='red')
    dda_line(v2[0], v2[1], v3[0], v3[1], color='green')
    dda_line(v3[0], v3[1], v1[0], v1[1], color='blue')
```

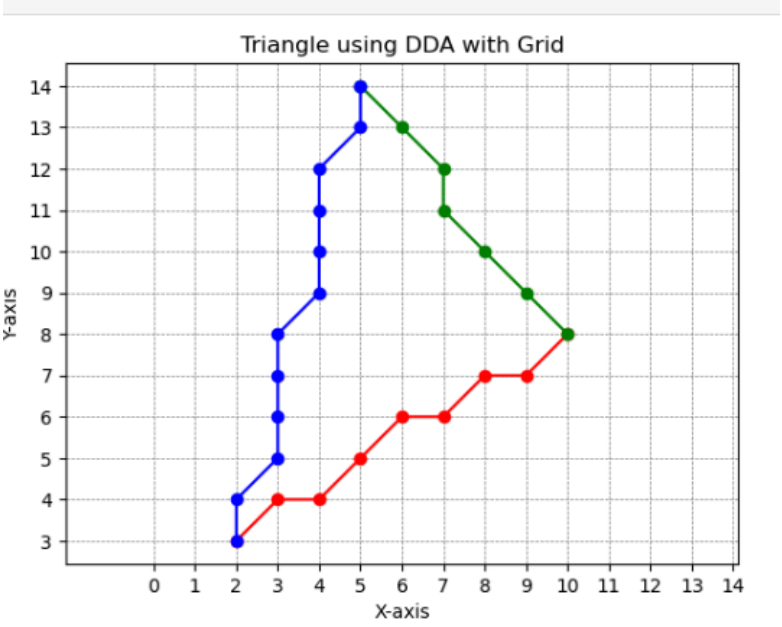
```
    plt.title('Triangle using DDA with Grid')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.axis('equal')
    plt.show()

# Sample vertices
vertex1 = (2, 3)
vertex2 = (10, 8)
vertex3 = (5, 14)

# Draw the triangle with grid
draw_triangle_with_grid(vertex1, vertex2, vertex3)
```

Description of code:

This program uses the Digital Differential Analyzer (DDA) algorithm to draw straight lines between three vertices, forming a triangle.

- The `dda_line()` function calculates intermediate pixel coordinates based on the slope and plots them.
- The `draw_triangle_with_grid()` function draws a grid background using `matplotlib` and connects the vertices with three DDA-generated lines in different colors.
- The grid helps visualize how each plotted pixel aligns with the underlying coordinate system

Output:

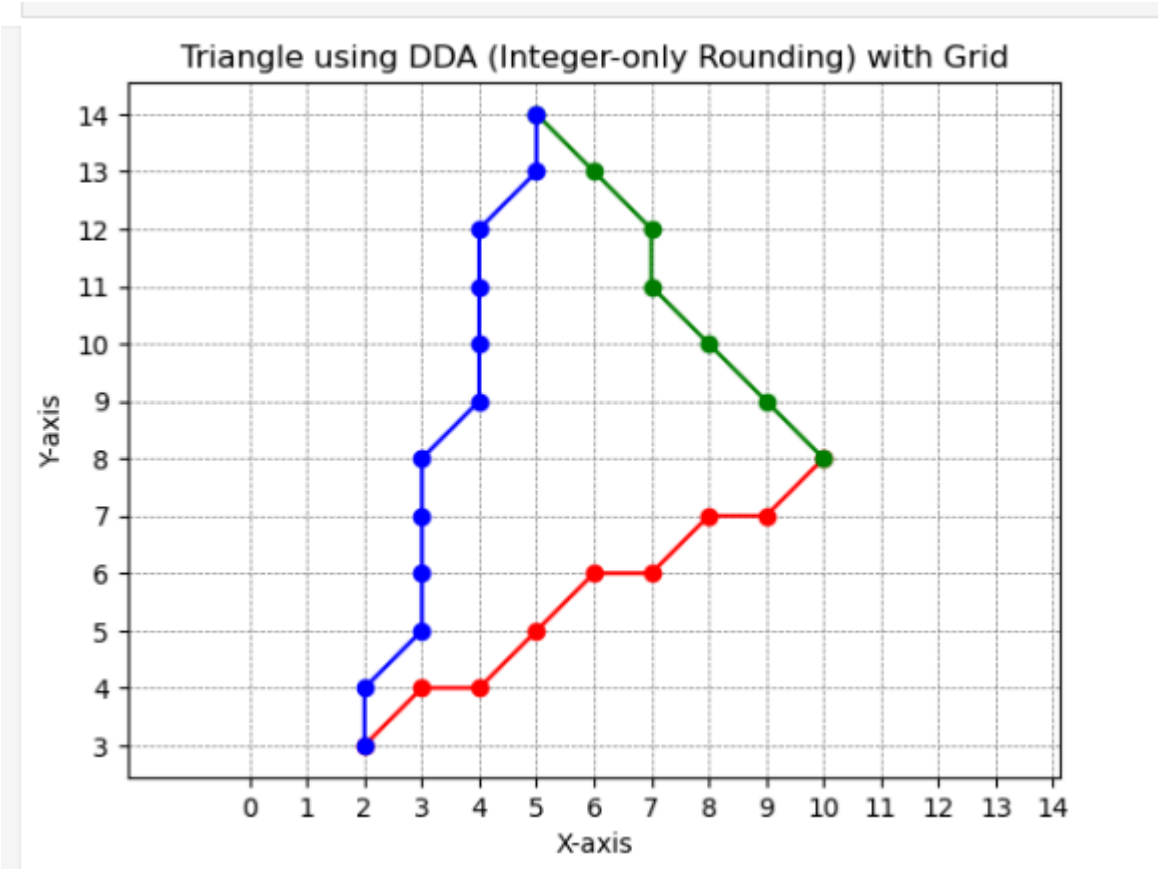


Task 2 code is given below adding with task 3 (Draw a grid and visualize line drawing over it using matplotlib.)in it

```python
import matplotlib.pyplot as plt
import math

# DDA line drawing with integer-only rounding logic
def dda_line_integer_rounding(x1, y1, x2, y2, color='blue'):
    dx = x2 - x1
    dy = y2 - y1

    m = float('inf') if dx == 0 else dy / dx

    x_points, y_points = [], []

    if abs(m) <= 1:  # Move in x-direction
        if x1 > x2:
            x1, y1, x2, y2 = x2, y2, x1, y1
        x, y = x1, y1
        while x <= x2:
            # Integer-only rounding logic
            x_points.append(int(x + 0.5) if x >= 0 else int(x - 0.5))
            y_points.append(int(y + 0.5) if y >= 0 else int(y - 0.5))
            x += 1
            y += m
    else:  # Move in y-direction
        if y1 > y2:
            x1, y1, x2, y2 = x2, y2, x1, y1
        x, y = x1, y1
        while y <= y2:
            # Integer-only rounding logic
            x_points.append(int(x + 0.5) if x >= 0 else int(x - 0.5))
            y_points.append(int(y + 0.5) if y >= 0 else int(y - 0.5))
            y += 1
            x += 1/m

    plt.plot(x_points, y_points, marker='o', color=color)

# Function to draw a triangle with grid
def draw_triangle_with_grid_integer_rounding(v1, v2, v3):
    plt.figure()

    # Draw grid background
    plt.grid(True, which='both', color='gray', linestyle='--', linewidth=0.5)
    plt.xticks(range(0, 21))
    plt.yticks(range(0, 21))
```

```
    # Draw triangle edges with integer-only rounding
    dda_line_integer_rounding(v1[0], v1[1], v2[0], v2[1], color='red')
    dda_line_integer_rounding(v2[0], v2[1], v3[0], v3[1], color='green')
    dda_line_integer_rounding(v3[0], v3[1], v1[0], v1[1], color='blue')

    plt.title('Triangle using DDA (Integer-only Rounding) with Grid')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.axis('equal')
    plt.show()

# Sample vertices
vertex1 = (2, 3)
vertex2 = (10, 8)
vertex3 = (5, 14)

# Draw triangle with grid using integer rounding
draw_triangle_with_grid_integer_rounding(vertex1, vertex2, vertex3)
```

Output:



The Differences is Observed for Plot points using integer-only rounding logic (no round() function)

When plotting points using integer-only rounding logic instead of Python's `round()` function:

- Pixel placement changes slightly because integer rounding is done manually:
    - If value ≥ 0 → add `0.5` before truncating.
    - If value < 0 → subtract `0.5` before truncating.
- This method mimics hardware-like rounding in older graphics systems, where floating-point rounding was avoided.
- The line may appear shifted or have slightly different pixel alignment compared to `round()` due to the way half-values are handled.
- In some cases, this method produces more consistent diagonal pixel steps, while `round()` can cause uneven gaps.

Lab2: Bresenham Line Drawing Algorithm

Lab Topic :Implement the Bresenham's line drawing algorithm

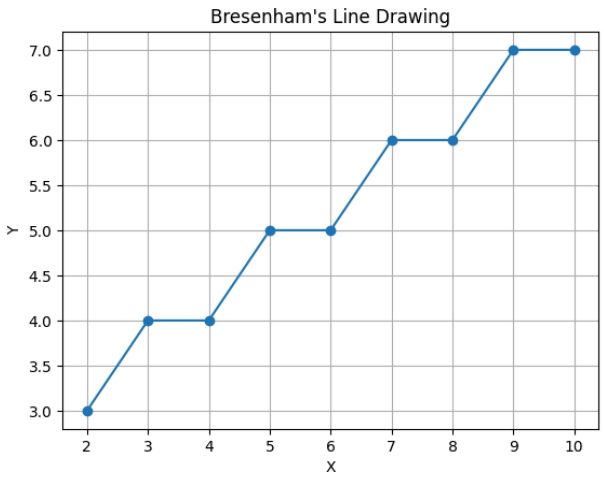
Objectives: To draw a line efficiently using integer calculations.

1. To convert a geometric line into discrete pixels on a screen.
2. To understand rasterization and pixel plotting.
3. To compare efficiency with the DDA algorithm.
4. To build a foundation for other graphics algorithms.

Code:

```python
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    d = 2 * dy - dx
    dS = 2 * dy
    dT = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    while x < x2:
        if d < 0:
            d += dS
        else:
            d += dT
            y += 1
        x += 1
        x_points.append(x)
        y_points.append(y)

    # Plot the line
    plt.plot(x_points, y_points, marker='o')
    plt.title("Bresenham's Line Drawing")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

# Example Run
bresenham_line(2, 3, 10, 7)
```

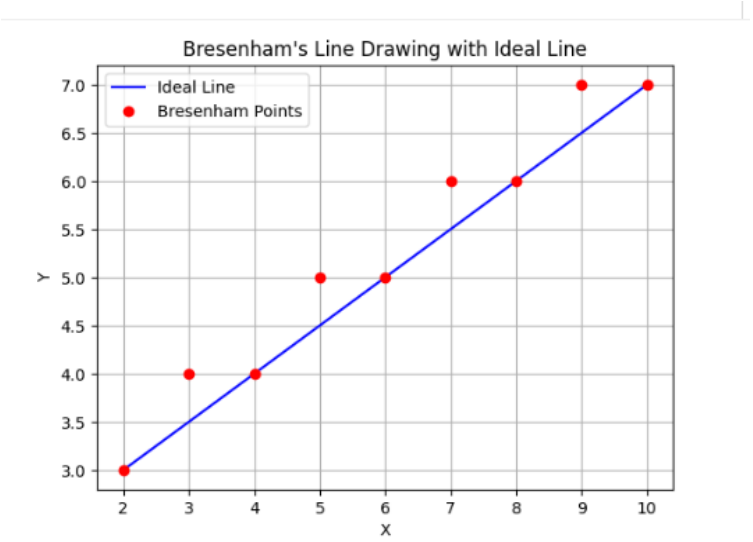
Output:



Code2:

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

def bresenham_line(x1, y1, x2, y2):
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    d = 2 * dy - dx
    dS = 2 * dy
    dT = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    while x < x2:
        if d < 0:
            d += dS
        else:
            d += dT
            y += 1
        x += 1
        x_points.append(x)
        y_points.append(y)

    # Draw ideal straight line
    x_vals = np.linspace(x1, x2, 100)
    slope = dy / dx if dx != 0 else 0
    y_vals = slope * (x_vals - x1) + y1

    plt.plot(x_vals, y_vals, 'b-', label='Ideal Line')          # Ideal line in blue
    plt.plot(x_points, y_points, 'ro', label='Bresenham Points')  # Bresenham points in red circles

    plt.title("Bresenham's Line Drawing with Ideal Line")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()

# Example Run
bresenham_line(2, 3, 10, 7)
```

Output:



Lab2.2

Lab Exercises

Exercise 1:

Draw a line from (0, 0) to (8, 5).

● Hint: Check the values of dx, dy, and verify each value of d after every step.

Exercise 2:

Modify the program to accept user input for coordinates.

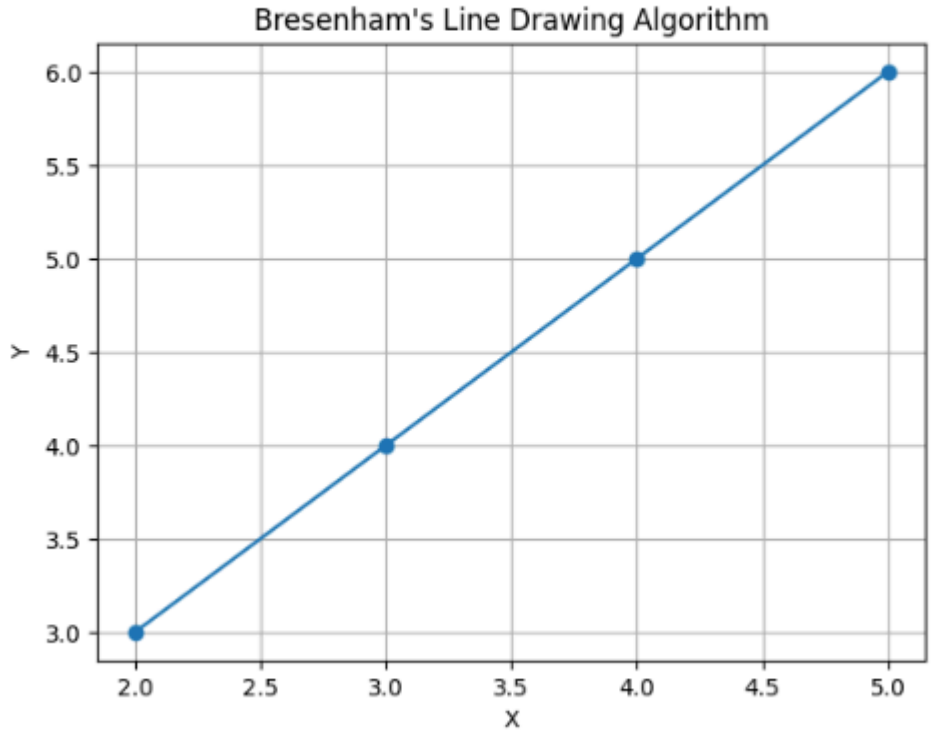
● Hint: Use input() function in Python and cast to int.

Code:

```
#Exercise 1 and 2 in one code

import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    d = 2 * dy - dx
    dE = 2 * dy
    dNE = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    while x < x2:
        if d < 0:
            d += dE
        else:
            d += dNE
            y += 1
        x += 1
        x_points.append(x)
        y_points.append(y)

    # Plot the line
    plt.plot(x_points, y_points, marker='o')
    plt.title("Bresenham's Line Drawing Algorithm")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

    return list(zip(x_points, y_points))

# User Input
x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))

pixels = bresenham_line(x1, y1, x2, y2)
print("pixels:", pixels)
```

Output:

```
Enter x1: 2
Enter y1: 3
Enter x2: 5
Enter y2: 7
```




```
pixels: [(2, 3), (3, 4), (4, 5), (5, 6)]
```

Exercise 3:

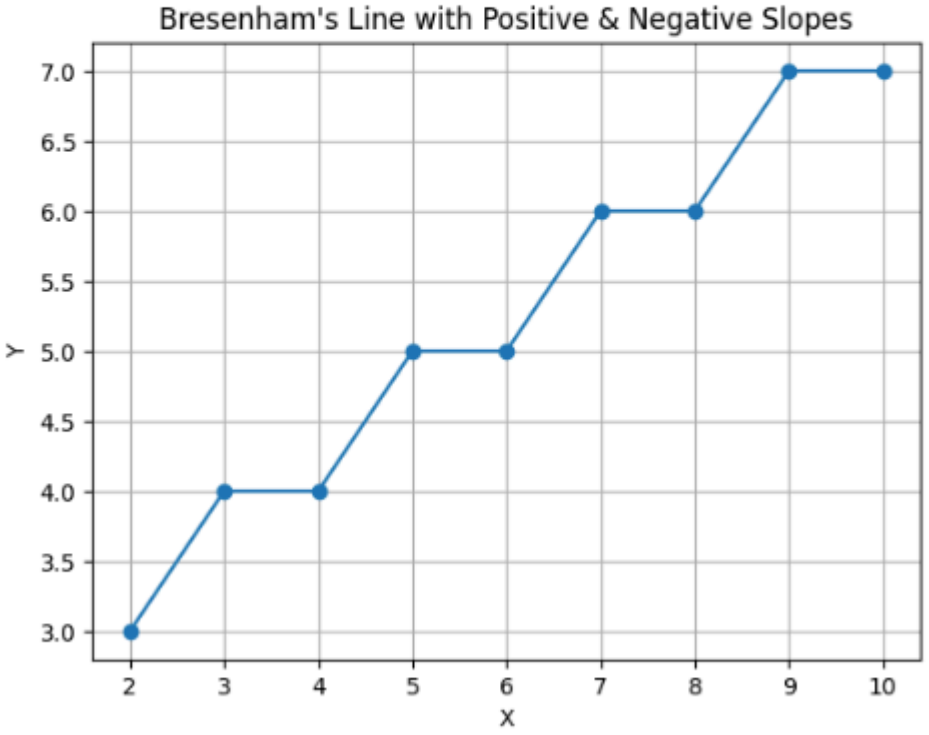
Implement the line drawing algorithm for lines with a negative slope.

● Hint: Modify the y-increment direction based on slope.

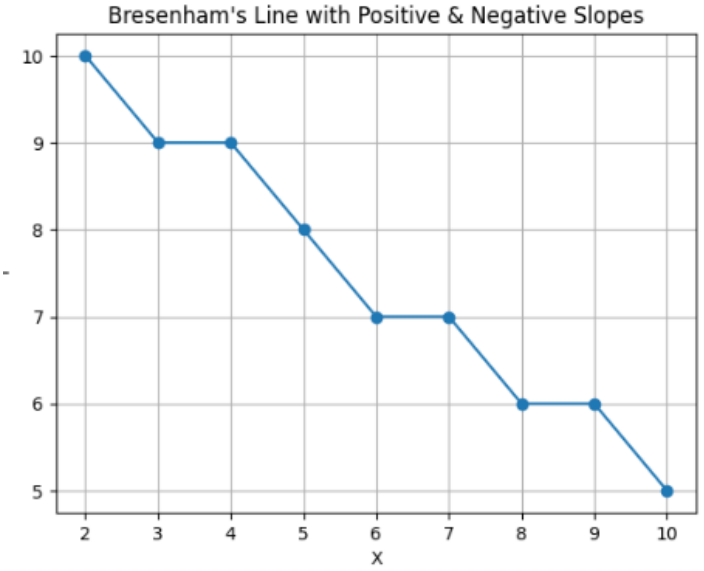
```python
#exercise 3
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    # Ensure we always draw left to right
    if x1 > x2:
        x1, y1, x2, y2 = x2, y2, x1, y1

    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    # Decide slope direction
    y_step = 1 if dy >= 0 else -1
    dy = abs(dy)

    d = 2 * dy - dx
    dE = 2 * dy
    dNE = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    while x < x2:
        if d < 0:
            d += dE
        else:
            d += dNE
            y += y_step   # Handles both positive and negative slopes
        x += 1
        x_points.append(x)
        y_points.append(y)

    # Plot the line
    plt.plot(x_points, y_points, marker='o')
    plt.title("Bresenham's Line with Positive & Negative Slopes")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

    return list(zip(x_points, y_points))  # Pixel activation list
```

```
# --- Example Runs ---
print("Positive slope line:")
print(bresenham_line(2, 3, 10, 7))  # slope ≈ +0.5

print("Negative slope line:")
print(bresenham_line(2, 10, 10, 5))  # slope ≈ -0.625
```

```
Positive slope line:
```




```
[(2, 3), (3, 4), (4, 4), (5, 5), (6, 5), (7, 6), (8, 6), (9, 7), (10, 7)]
Negative slope line:
```




```
2, 10), (3, 9), (4, 9), (5, 8), (6, 7), (7, 7), (8, 6), (9, 6), (10, 5)]
```

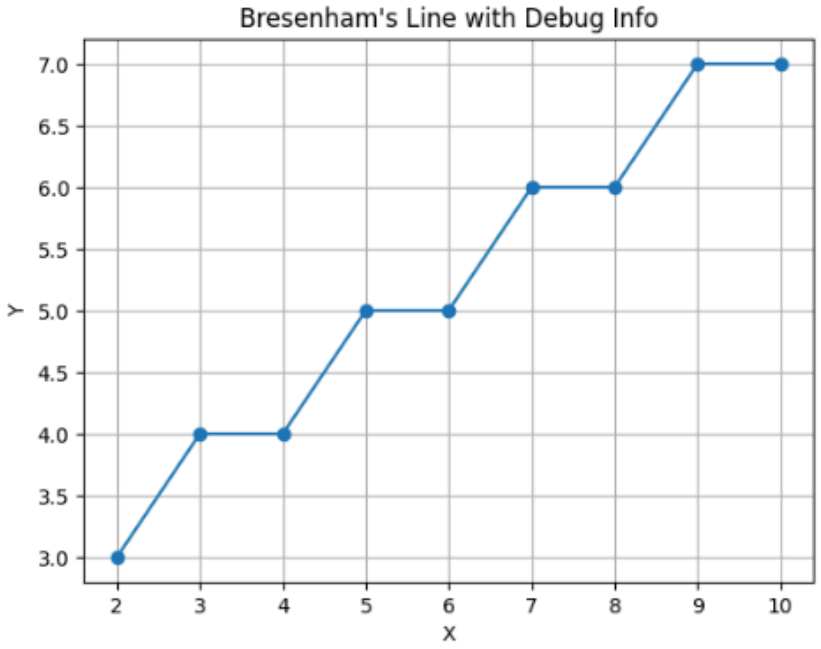
Exercise 4:
Print all values of d, x, and y in each iteration.
● Hint: Use print() inside the loop.

Code:

```python
#exercise 4
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    # Ensure we always draw left to right
    if x1 > x2:
        x1, y1, x2, y2 = x2, y2, x1, y1

    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    # Decide slope direction
    y_step = 1 if dy >= 0 else -1
    dy = abs(dy)

    d = 2 * dy - dx
    dE = 2 * dy
    dNE = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    print(f"Initial: x={x}, y={y}, d={d}")

    while x < x2:
        if d < 0:
            d += dE
        else:
            d += dNE
            y += y_step   # Handles positive & negative slopes
        x += 1

        x_points.append(x)
        y_points.append(y)

        # Print after each step
        print(f"Iteration: x={x}, y={y}, d={d}")
```

```
    # Plot the line
    plt.plot(x_points, y_points, marker='o')
    plt.title("Bresenham's Line with Debug Info")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

    return list(zip(x_points, y_points))  # Pixel activation list


# Example Run
pixels = bresenham_line(2, 3, 10, 7)
print("Activated pixels:", pixels)
```

```
Initial: x=2, y=3, d=0
Iteration: x=3, y=4, d=-8
Iteration: x=4, y=4, d=0
Iteration: x=5, y=5, d=-8
Iteration: x=6, y=5, d=0
Iteration: x=7, y=6, d=-8
Iteration: x=8, y=6, d=0
Iteration: x=9, y=7, d=-8
Iteration: x=10, y=7, d=0
```




```
Activated pixels: [(2, 3), (3, 4), (4, 4), (5, 5), (6, 5), (7, 6), (8, 6), (9, 7), (10, 7)]
```

Home Tasks

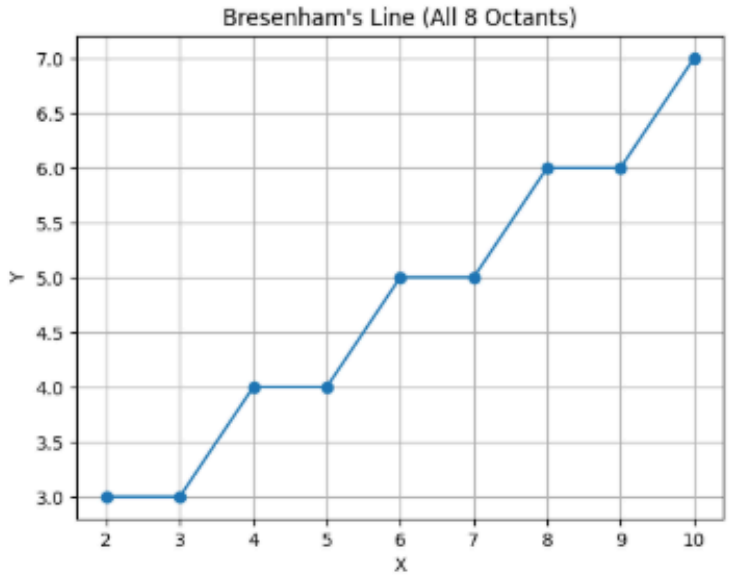
1. Modify the algorithm to draw lines in all 8 octants.
2. Compare output with DDA algorithm.
3. Count the number of pixels drawn for a diagonal line.
4. Extend the code to draw multiple lines using loops.

Home task are in next page

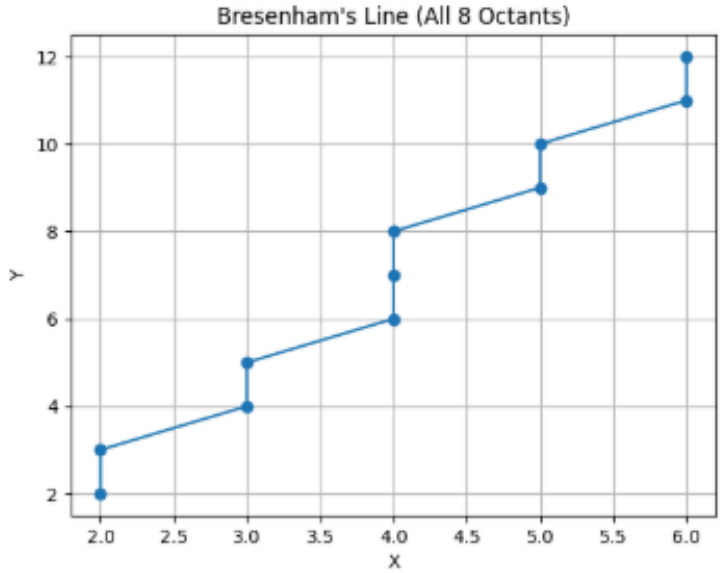
Task 1:

```python
#1. Modify the algorithm to draw lines in all 8 octants.
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    # Store points
    x_points = []
    y_points = []

    # Calculate dx, dy
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    # Step directions
    x_step = 1 if x2 > x1 else -1
    y_step = 1 if y2 > y1 else -1

    # Case 1: Gentle slope (|slope| ≤ 1) → iterate over x
    if dx >= dy:
        d = 2*dy - dx
        x, y = x1, y1
        for _ in range(dx + 1):
            x_points.append(x)
            y_points.append(y)
            if d > 0:
                y += y_step
                d -= 2*dx
            d += 2*dy
            x += x_step

    # Case 2: Steep slope (|slope| > 1) → iterate over y
    else:
        d = 2*dx - dy
        x, y = x1, y1
        for _ in range(dy + 1):
            x_points.append(x)
            y_points.append(y)
            if d > 0:
                x += x_step
                d -= 2*dy
            d += 2*dx
            y += y_step
```

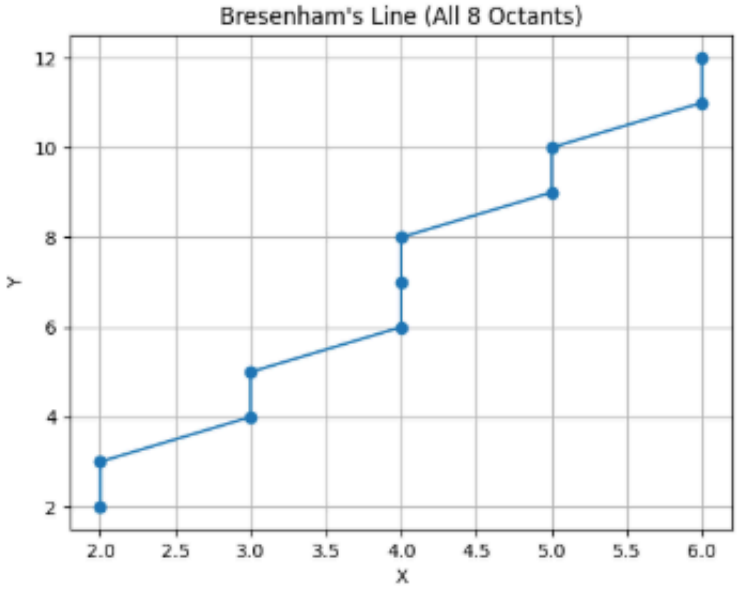
[+]



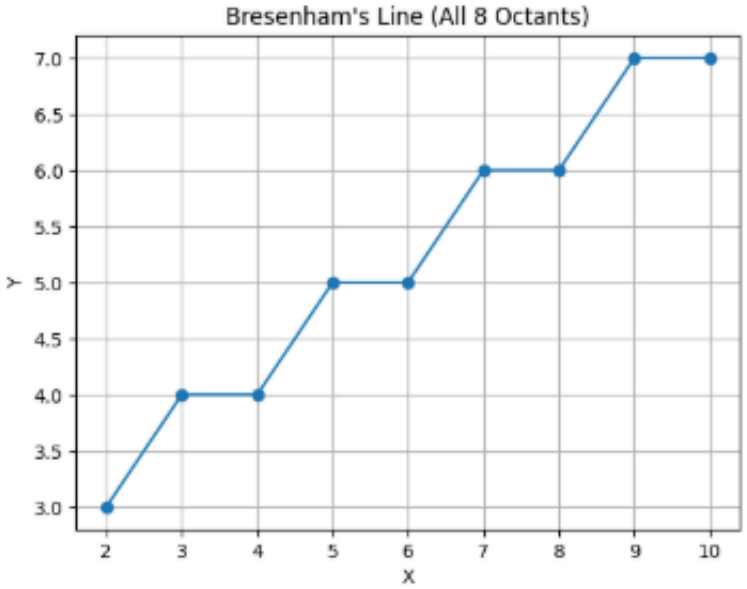

```
Octant 1 (gentle positive slope): [(2, 3), (3, 3), (4, 4), (5, 4), (6, 5), (7, 5), (8, 6), (9, 6), (10, 7)]
```




```
Octant 2 (steep positive slope): [(2, 2), (2, 3), (3, 4), (3, 5), (4, 6), (4, 7), (4, 8), (5, 9), (5, 10), (6, 11), (6, 12)]
```




```
Octant 3 (steep negative slope): [(6, 12), (6, 11), (5, 10), (5, 9), (4, 8), (4, 7), (4, 6), (3, 5), (3, 4), (2, 3), (2, 2)]
```



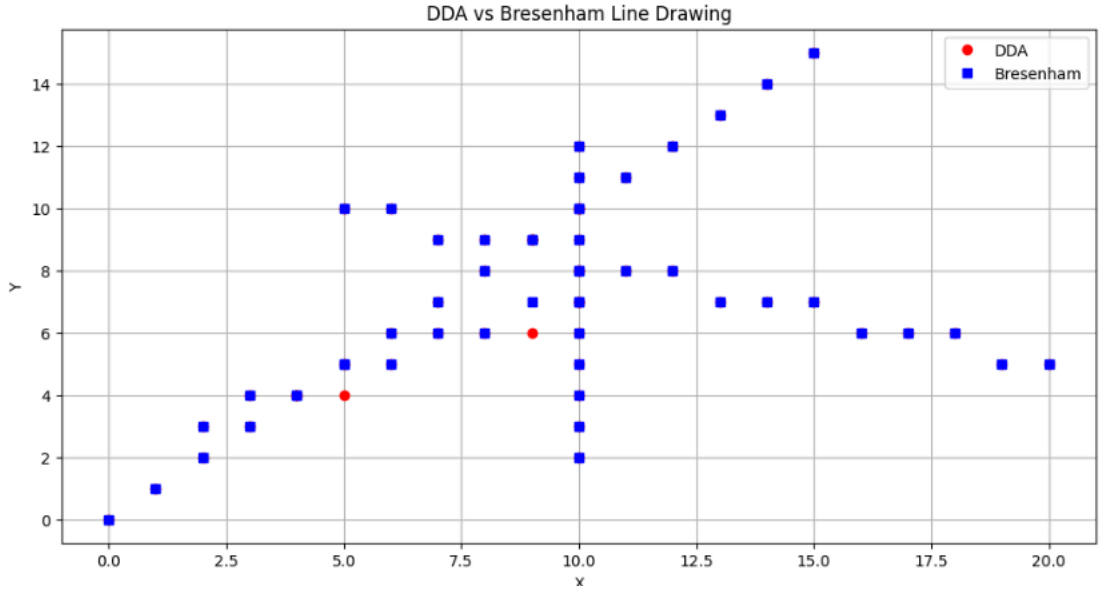

```
Octant 4 (gentle negative slope): [(10, 7), (9, 7), (8, 6), (7, 6), (6, 5), (5, 5), (4, 4), (3, 4), (2, 3)]
```

Home task 2:

```
# ------------------ DDA Algorithm ------------------
def dda_line(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    pixels = []

    if dx == 0:  # Vertical line
        m = float('inf')
    else:
        m = dy / dx

    if abs(m) <= 1:
        if x1 > x2:  # Swap points if needed
            x1, y1, x2, y2 = x2, y2, x1, y1
            dx = x2 - x1
            dy = y2 - y1
            m = dy / dx if dx != 0 else float('inf')
        x = x1
        y = y1
        while x <= x2:
            pixels.append((round(x), round(y)))
            x += 1
            y += m
    else:
        if y1 > y2:  # Swap points if needed
            x1, y1, x2, y2 = x2, y2, x1, y1
            dx = x2 - x1
            dy = y2 - y1
            m = dy / dx if dx != 0 else float('inf')
        x = x1
        y = y1
        while y <= y2:
            pixels.append((round(x), round(y)))
            y += 1
            x += 1/m

    return pixels

# ------------------ Bresenham Algorithm ------------------
def bresenham_line(x1, y1, x2, y2):
    pixels = []

    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    x, y = x1, y1

    sx = 1 if x2 > x1 else -1
    sy = 1 if y2 > y1 else -1
```

```
plt.title("DDA vs Bresenham Line Drawing")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.legend(['DDA','Bresenham'])
plt.show()
```

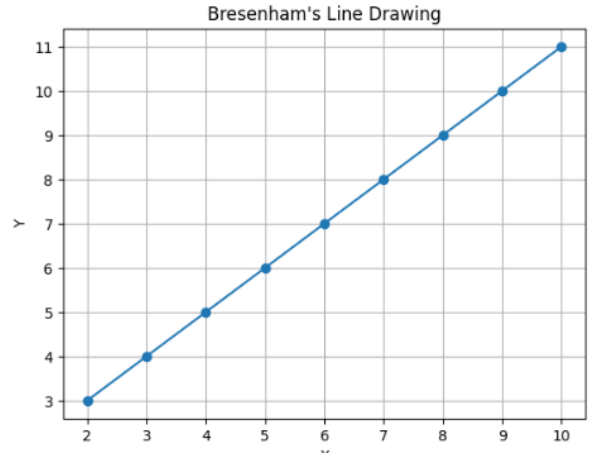
```
Line 1: DDA pixels=9, Bresenham pixels=9
Line 2: DDA pixels=16, Bresenham pixels=16
Line 3: DDA pixels=16, Bresenham pixels=16
Line 4: DDA pixels=11, Bresenham pixels=11
```



Home task 3:

```python
#3. Count the number of pixels drawn for a diagonal line.
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    d = 2 * dy - dx
    dS = 2 * dy
    dT = 2 * (dy - dx)

    x_points = [x]
    y_points = [y]

    while x < x2:
        if d < 0:
            d += dS
        else:
            d += dT
            y += 1
        x += 1

        x_points.append(x)
        y_points.append(y)

    # Count pixels
    pixel_count = len(x_points)
    print(f"Number of pixels drawn: {pixel_count}")

    # Plot the line
    plt.plot(x_points, y_points, marker='o')
    plt.title("Bresenham's Line Drawing")
    plt.grid(True)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

# Example Run: diagonal line
bresenham_line(2, 3, 10, 11)
```

```
Number of pixels drawn: 9
```

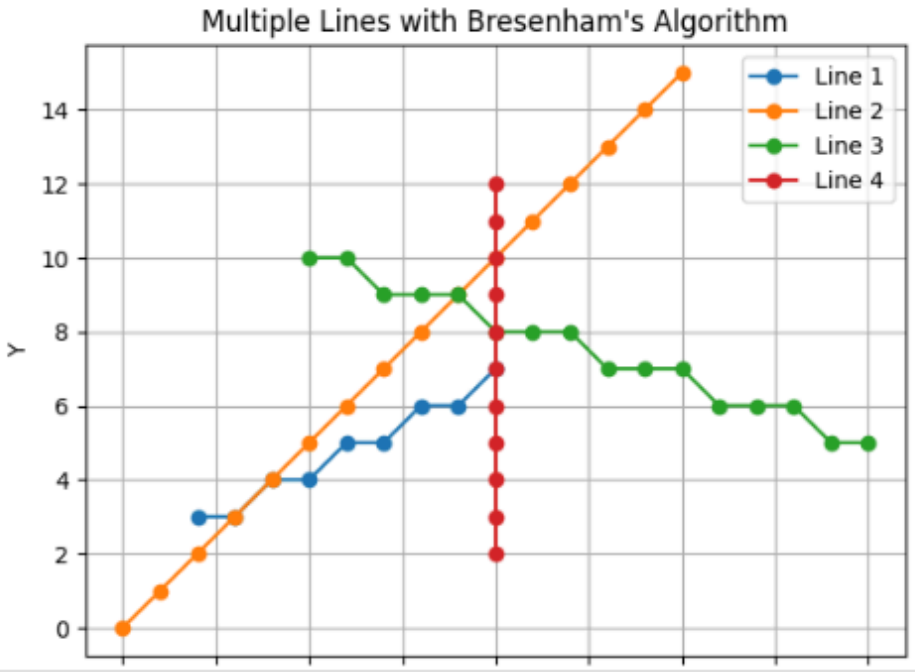


Home task4

```python
#4. Extend the code to draw multiple lines using loops.
import matplotlib.pyplot as plt

def bresenham_line(x1, y1, x2, y2):
    """Draw a line from (x1, y1) to (x2, y2) using Bresenham's algorithm."""
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1

    # Handle steep lines
    steep = abs(dy) > abs(dx)
    if steep:
        x, y = y, x
        dx, dy = dy, dx
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1

    dx = x2 - x1
    dy = y2 - y1
    d = 2*abs(dy) - dx
    y_step = 1 if y2 > y1 else -1

    x_points = []
    y_points = []

    y = y1
    for x in range(x1, x2 + 1):
        if steep:
            x_points.append(y)
            y_points.append(x)
        else:
            x_points.append(x)
            y_points.append(y)

        if d > 0:
            y += y_step
            d -= 2*dx
        d += 2*abs(dy)

    return x_points, y_points
```

```python
# Example: Draw multiple lines
lines = [
    (2, 3, 10, 7),
    (0, 0, 15, 15),   # diagonal
    (5, 10, 20, 5),   # negative slope
    (10, 2, 10, 12)   # vertical line
]

plt.figure()
for idx, (x1, y1, x2, y2) in enumerate(lines):
    x_points, y_points = bresenham_line(x1, y1, x2, y2)
    plt.plot(x_points, y_points, marker='o', label=f'Line {idx+1}')
    print(f'Line {idx+1} pixels: {len(x_points)}')

plt.title("Multiple Lines with Bresenham's Algorithm")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.legend()
plt.show()
```

```
Line 1 pixels: 9
Line 2 pixels: 16
Line 3 pixels: 16
Line 4 pixels: 11
```



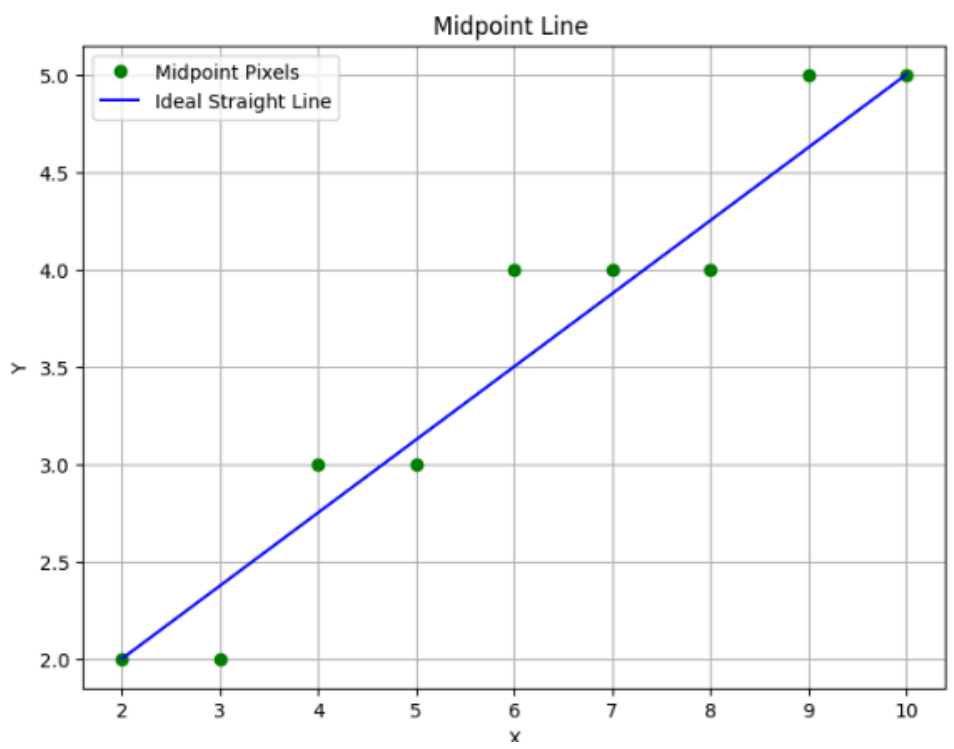
Lab3:Mid point line drawing algorithm

Draw the ine using Midpoint Line Generation Algorithm.

```
import matplotlib.pyplot as plt

def midpoint_line(x1, y1, x2, y2):
    x_points = []
    y_points = []

    dx = x2 - x1
    dy = y2 - y1

    x, y = x1, y1
    x_points.append(x)
    y_points.append(y)

    if abs(dy) <= abs(dx):
        d = dy - (dx / 2)
        while x < x2:
            x += 1
            if d < 0:
                d = d + dy
            else:
                d = d + (dy - dx)
                y += 1
            x_points.append(x)
            y_points.append(y)
    else:   # dy > dx
        d = dx - (dy / 2)
        while y < y2:
            y += 1
            if d < 0:
                d = d + dx
            else:
                d = d + (dx - dy)
                x += 1
            x_points.append(x)
            y_points.append(y)

    return x_points, y_points


# Example usage
x1, y1 = 2, 2
x2, y2 = 10, 5

x_points, y_points = midpoint_line(x1, y1, x2, y2)

# Plotting
plt.figure(figsize=(8, 6))
```

```
# Midpoint pixel points (red dots)
plt.plot(x_points, y_points, "go", label="Midpoint Pixels")

# Ideal straight line (blue line connecting start to end)
plt.plot([x1, x2], [y1, y2], "b-", label="Ideal Straight Line")

plt.title("Midpoint Line ")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.grid(True)
plt.show()
```

[4]



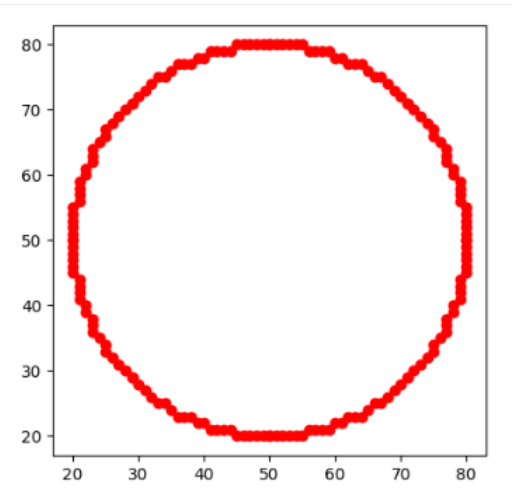
Lab4:Bresenham's Circle Drawing Algorithm

Classtask:

Bresenham's Circle Drawing Algorithm

```
import matplotlib.pyplot as plt

# Function to plot eight symmetric points
def drawCircle(xc, yc, x, y):
    plt.plot(xc + x, yc + y, 'ro')  # Octant 1
    plt.plot(xc - x, yc + y, 'ro')  # Octant 2
    plt.plot(xc + x, yc - y, 'ro')  # Octant 3
    plt.plot(xc - x, yc - y, 'ro')  # Octant 4
    plt.plot(xc + y, yc + x, 'ro')  # Octant 5
    plt.plot(xc - y, yc + x, 'ro')  # Octant 6
    plt.plot(xc + y, yc - x, 'ro')  # Octant 7
    plt.plot(xc - y, yc - x, 'ro')  # Octant 8

def bresenhamCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r  # Decision parameter

    while x <= y:
        drawCircle(xc, yc, x, y)
        if d < 0:  # Select N (x+1, y)
            d = d + (4 * x) + 6
        else:  # Select S (x+1, y-1)
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage
if __name__ == "__main__":
    xc, yc = 50, 50  # Circle center
    r = 30           # Circle radius
    bresenhamCircle(xc, yc, r)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()
```

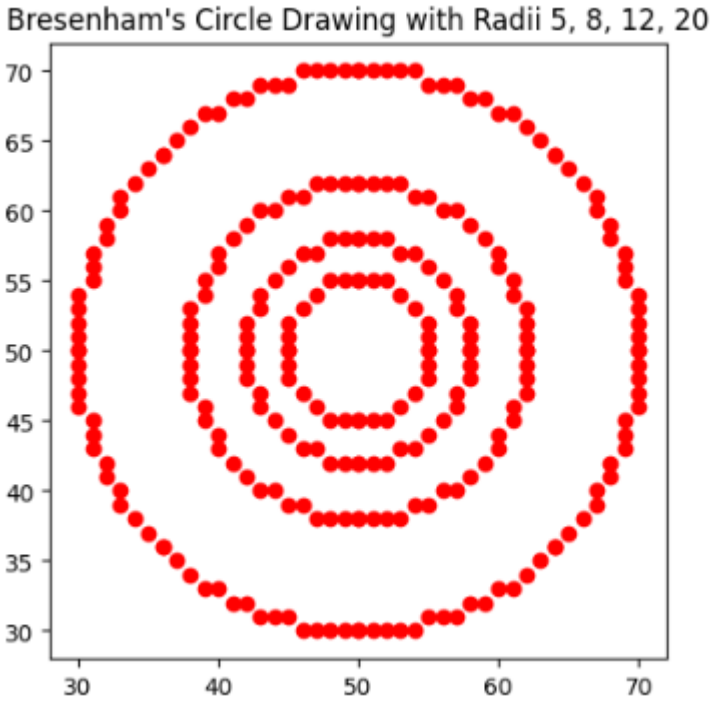
Output:



More Tasks:

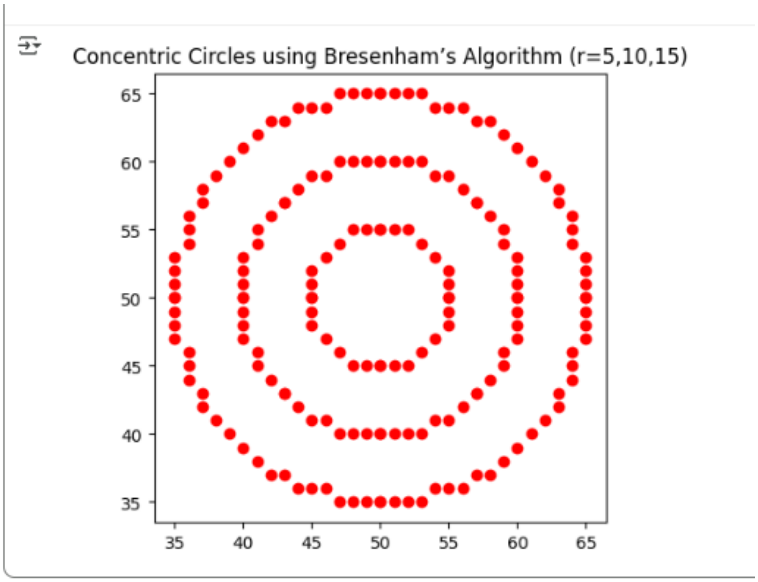
1. Circle Drawing with Different Radii Draw circles using Bresenham's algorithm with radii 5, 8, 12, and 20.

```python
import matplotlib.pyplot as plt

# Function to plot eight symmetric points
def drawCircle(xc, yc, x, y):
    plt.plot(xc + x, yc + y, 'ro')  # Octant 1
    plt.plot(xc - x, yc + y, 'ro')  # Octant 2
    plt.plot(xc + x, yc - y, 'ro')  # Octant 3
    plt.plot(xc - x, yc - y, 'ro')  # Octant 4
    plt.plot(xc + y, yc + x, 'ro')  # Octant 5
    plt.plot(xc - y, yc + x, 'ro')  # Octant 6
    plt.plot(xc + y, yc - x, 'ro')  # Octant 7
    plt.plot(xc - y, yc - x, 'ro')  # Octant 8

def bresenhamCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r  # Decision parameter

    while x <= y:
        drawCircle(xc, yc, x, y)
        if d < 0:  # Select N (x+1, y)
            d = d + (4 * x) + 6
        else:  # Select S (x+1, y-1)
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage: draw multiple circles
if __name__ == "__main__":
    xc, yc = 50, 50  # Circle center
    radii = [5, 8, 12, 20]  # Different radii

    for r in radii:
        bresenhamCircle(xc, yc, r)

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Bresenham's Circle Drawing with Radii 5, 8, 12, 20")
    plt.show()
```



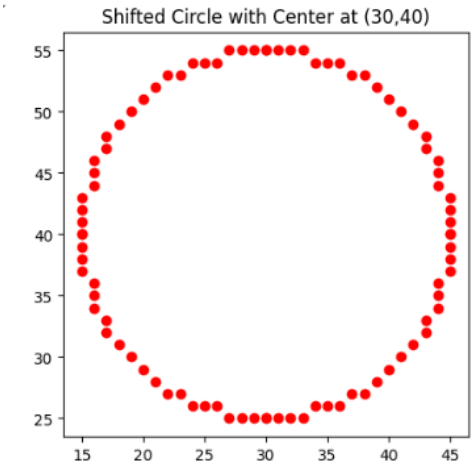
2.Multiple Circles from Same Center Implement a program that draws concentric circles (e.g., radii = 5, 10, 15) using Bresenham’s algorithm.

```python
import matplotlib.pyplot as plt

# Function to plot eight symmetric points
def drawCircle(xc, yc, x, y):
    plt.plot(xc + x, yc + y, 'ro')  # Octant 1
    plt.plot(xc - x, yc + y, 'ro')  # Octant 2
    plt.plot(xc + x, yc - y, 'ro')  # Octant 3
    plt.plot(xc - x, yc - y, 'ro')  # Octant 4
    plt.plot(xc + y, yc + x, 'ro')  # Octant 5
    plt.plot(xc - y, yc + x, 'ro')  # Octant 6
    plt.plot(xc + y, yc - x, 'ro')  # Octant 7
    plt.plot(xc - y, yc - x, 'ro')  # Octant 8

def bresenhamCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r  # Decision parameter

    while x <= y:
        drawCircle(xc, yc, x, y)
        if d < 0:  # Select N
            d = d + (4 * x) + 6
        else:  # Select S
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage: draw concentric circles
if __name__ == "__main__":
    xc, yc = 50, 50   # Same center
    radii = [5, 10, 15]  # Concentric circles

    for r in radii:
        bresenhamCircle(xc, yc, r)

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Concentric Circles using Bresenham’s Algorithm (r=5,10,15)")
    plt.show()
```



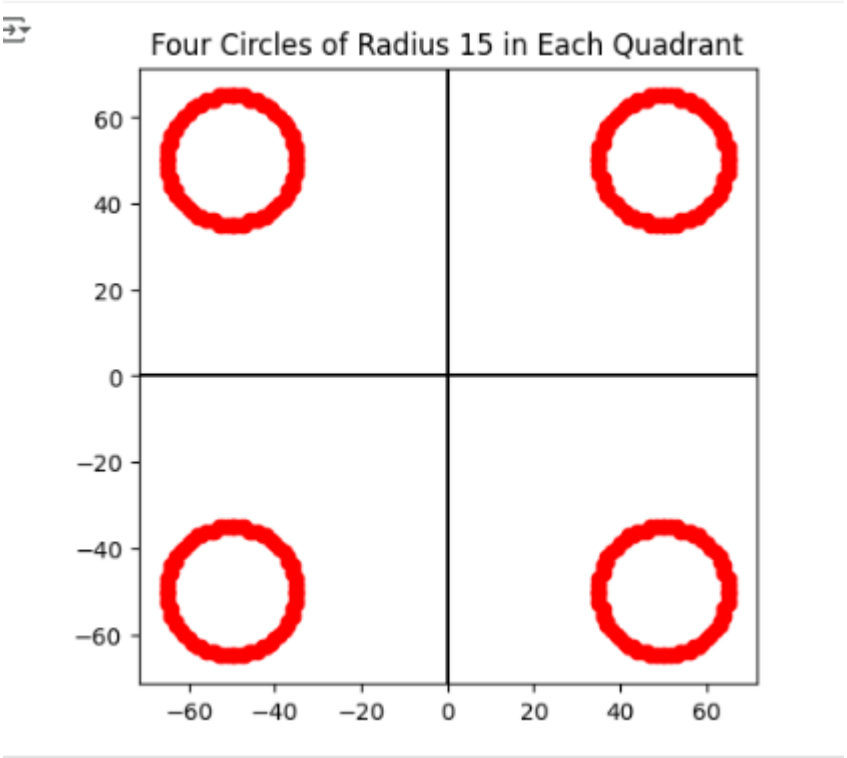
3. Shifted Circle Modify the algorithm to draw a circle with a center at (30, 40) instead of the origin.

```python
import matplotlib.pyplot as plt

# Function to plot eight symmetric points with given center
def drawCircle(xc, yc, x, y):
    plt.plot(xc + x, yc + y, 'ro')  # Octant 1
    plt.plot(xc - x, yc + y, 'ro')  # Octant 2
    plt.plot(xc + x, yc - y, 'ro')  # Octant 3
    plt.plot(xc - x, yc - y, 'ro')  # Octant 4
    plt.plot(xc + y, yc + x, 'ro')  # Octant 5
    plt.plot(xc - y, yc + x, 'ro')  # Octant 6
    plt.plot(xc + y, yc - x, 'ro')  # Octant 7
    plt.plot(xc - y, yc - x, 'ro')  # Octant 8

def bresenhamCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r  # Decision parameter

    while x <= y:
        drawCircle(xc, yc, x, y)
        if d < 0:  # Select N
            d = d + (4 * x) + 6
        else:  # Select S
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage: draw a shifted circle
if __name__ == "__main__":
    xc, yc = 30, 40   # Shifted center
    r = 15            # Radius
    bresenhamCircle(xc, yc, r)

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Shifted Circle with Center at (30,40)")
    plt.show()
```



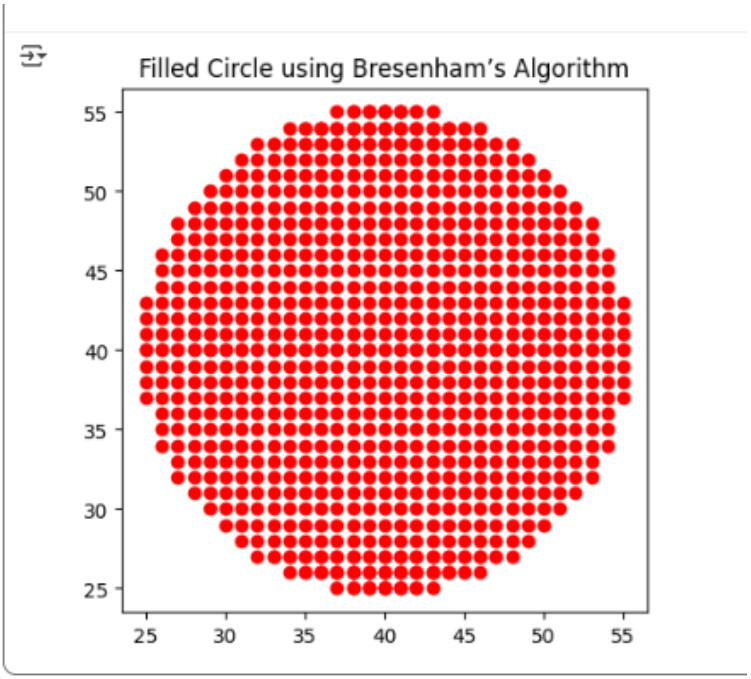
4. Circle in Each Quadrant Draw four circles of radius 15, each centered in one quadrant of the screen.

```python
import matplotlib.pyplot as plt

# Function to plot eight symmetric points with given center
def drawCircle(xc, yc, x, y):
    plt.plot(xc + x, yc + y, 'ro')  # Octant 1
    plt.plot(xc - x, yc + y, 'ro')  # Octant 2
    plt.plot(xc + x, yc - y, 'ro')  # Octant 3
    plt.plot(xc - x, yc - y, 'ro')  # Octant 4
    plt.plot(xc + y, yc + x, 'ro')  # Octant 5
    plt.plot(xc - y, yc + x, 'ro')  # Octant 6
    plt.plot(xc + y, yc - x, 'ro')  # Octant 7
    plt.plot(xc - y, yc - x, 'ro')  # Octant 8

def bresenhamCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r  # Decision parameter

    while x <= y:
        drawCircle(xc, yc, x, y)
        if d < 0:  # Select N
            d = d + (4 * x) + 6
        else:  # Select S
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage: draw circles in each quadrant
if __name__ == "__main__":
    r = 15
    centers = [(50, 50), (-50, 50), (-50, -50), (50, -50)]  # Quadrant centers

    for (xc, yc) in centers:
        bresenhamCircle(xc, yc, r)

    plt.axhline(0, color='black')  # X-axis
    plt.axvline(0, color='black')  # Y-axis
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Four Circles of Radius 15 in Each Quadrant")
    plt.show()
```



5. Filled Circle (Extension Task) Using the points generated by Bresenham's circle algorithm, implement a method to fill the circle with pixels.

```python
import matplotlib.pyplot as plt

# Function to draw a horizontal line (fill between two symmetric points)
def drawHorizontalLine(xc, yc, x, y):
    # Fill horizontally across the circle
    plt.plot(range(xc - x, xc + x + 1), [yc + y] * (2 * x + 1), 'ro')
    plt.plot(range(xc - x, xc + x + 1), [yc - y] * (2 * x + 1), 'ro')
    plt.plot(range(xc - y, xc + y + 1), [yc + x] * (2 * y + 1), 'ro')
    plt.plot(range(xc - y, xc + y + 1), [yc - x] * (2 * y + 1), 'ro')

def bresenhamFilledCircle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r

    while x <= y:
        drawHorizontalLine(xc, yc, x, y)
        if d < 0:  # Select N
            d = d + (4 * x) + 6
        else:  # Select S
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Example usage: draw a filled circle
if __name__ == "__main__":
    xc, yc = 40, 40   # Center
    r = 15            # Radius
    bresenhamFilledCircle(xc, yc, r)

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Filled Circle using Bresenham's Algorithm")
    plt.show()
```


Filled Circle using Bresenham’s Algorithm
55
50
45
40
35
30
25
25
30
35
40
45
50
55

6.Using your circle-drawing algorithm, design a simple “Olympic Rings” logo (five circles intersecting).

```python
import matplotlib.pyplot as plt

# Function to plot eight symmetric points with color
def drawCircle(xc, yc, x, y, color):
    plt.plot(xc + x, yc + y, color)
    plt.plot(xc - x, yc + y, color)
    plt.plot(xc + x, yc - y, color)
    plt.plot(xc - x, yc - y, color)
    plt.plot(xc + y, yc + x, color)
    plt.plot(xc - y, yc + x, color)
    plt.plot(xc + y, yc - x, color)
    plt.plot(xc - y, yc - x, color)

def bresenhamCircle(xc, yc, r, color):
    x = 0
    y = r
    d = 3 - 2 * r

    while x <= y:
        drawCircle(xc, yc, x, y, color)
        if d < 0:
            d = d + (4 * x) + 6
        else:
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1

# Olympic Rings
if __name__ == "__main__":
    r = 20  # Radius
    gap = 50  # Horizontal distance between centers

    # Top row (Blue, Black, Red)
    bresenhamCircle(30, 80, r, 'bo')   # Blue
    bresenhamCircle(30 + gap, 80, r, 'ko')   # Black
    bresenhamCircle(30 + 2 * gap, 80, r, 'ro')   # Red

    # Bottom row (Yellow, Green)
    bresenhamCircle(30 + gap // 2, 60, r, 'yo')   # Yellow
    bresenhamCircle(30 + 3 * gap // 2, 60, r, 'go')   # Green

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Olympic Rings using Bresenham’s Algorithm")
    plt.axis('off')
    plt.show()
```

Lab 5: 2D transformtion

🔹 Experiment Setup

We defined shapes (a triangle and a rectangle) in **homogeneous coordinates** $[x, y, 1]$.
Transformation matrices used:

- **Translation:**

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- **Scaling:**

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Rotation (θ degrees):**

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Shear:**

$$Sh = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Cell 1: Imports + Helper Functions

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

# ----- Helper Functions -----
def translate(points, tx, ty):
    """Translation of points by tx, ty"""
    T = np.array([[1, 0, tx],
                  [0, 1, ty],
                  [0, 0, 1]])
    return (T @ points.T).T

def scale(points, sx, sy):
    """Scaling of points by sx, sy"""
    S = np.array([[sx, 0, 0],
                  [0, sy, 0],
                  [0, 0, 1]])
    return (S @ points.T).T

def rotate(points, angle):
    """Rotation of points by given angle (in degrees)"""
    rad = np.radians(angle)
    R = np.array([[np.cos(rad), -np.sin(rad), 0],
                  [np.sin(rad),  np.cos(rad), 0],
                  [0, 0, 1]])
    return (R @ points.T).T
```

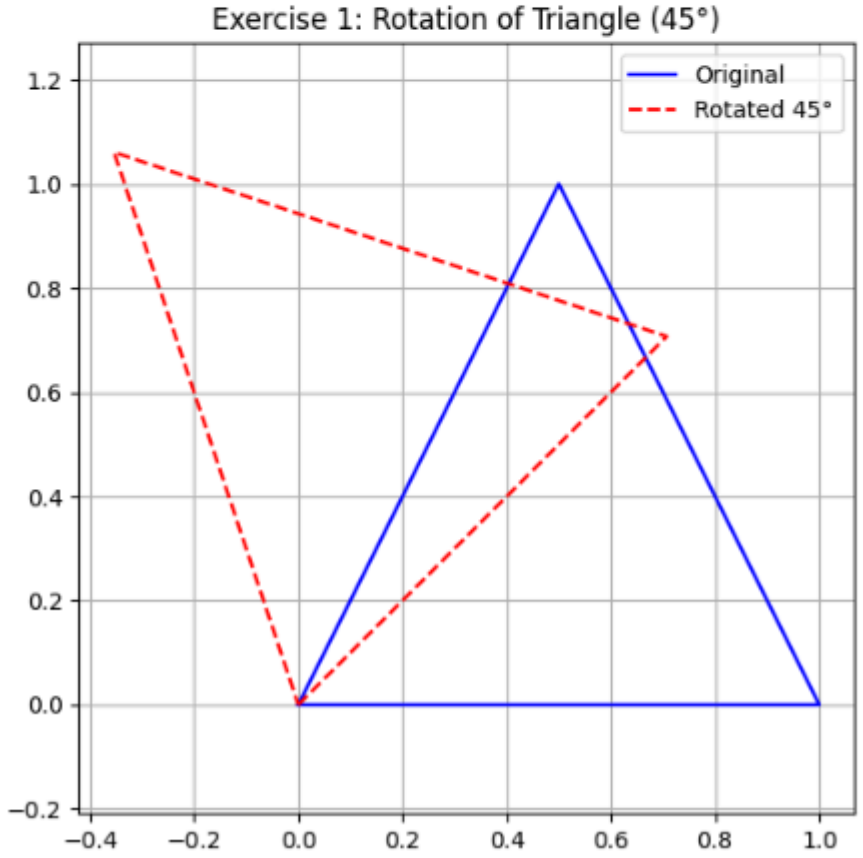
Cell 2: Define Shapes

```
# Example Shapes
triangle = np.array([[0, 0, 1],
                     [1, 0, 1],
                     [0.5, 1, 1],
                     [0, 0, 1]]) # closed triangle

rectangle = np.array([[0,0,1],
                      [2,0,1],
                      [2,1,1],
                      [0,1,1],
                      [0,0,1]]) # closed rectangle
```

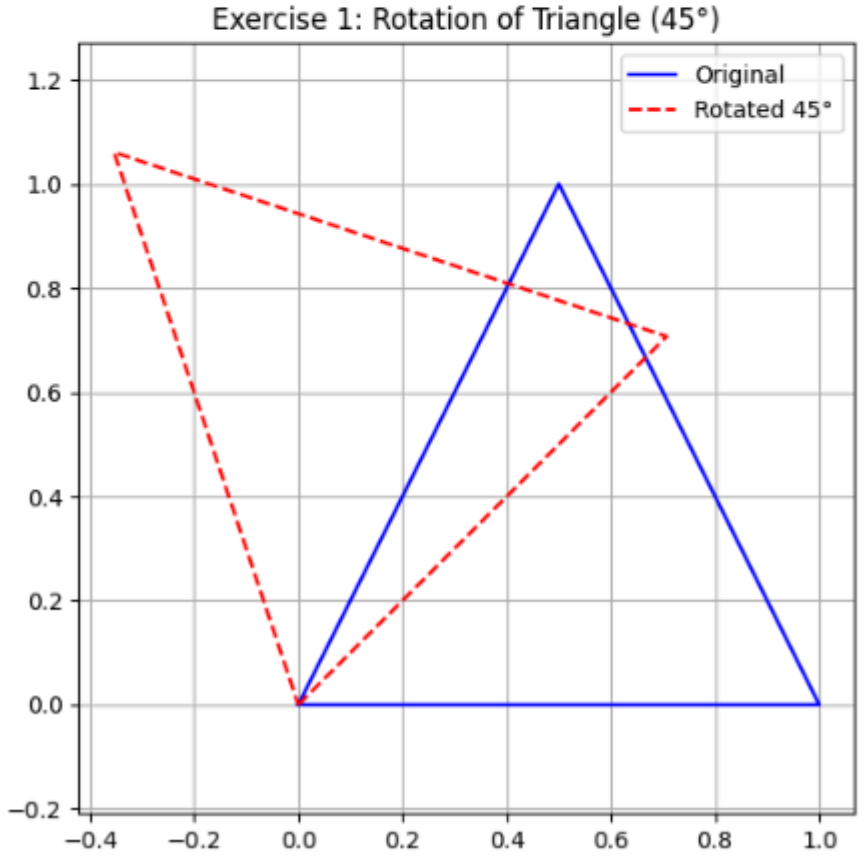
Exercise 1: Single Transform (Rotation 45° around origin)

```python
# Rotation function with explicit matrix return
def rotation_matrix(angle):
    rad = np.radians(angle)
    return np.array([[np.cos(rad), -np.sin(rad), 0],
                     [np.sin(rad),  np.cos(rad), 0],
                     [0, 0, 1]])

# Define angle
angle = 45
R = rotation_matrix(angle)

print("Rotation Matrix (45°):\n", R)

# Apply rotation
triangle_rot = (R @ triangle.T).T

# Show step-by-step multiplication for one vertex (example: [1,0,1])
vertex = triangle[1]   # second vertex (1,0,1)
print("\nOriginal Vertex:", vertex)
print("Matrix Multiplication:\n", R, " @ ", vertex)

result = R @ vertex
print("Resulting Vertex:", result)

# Print all coordinates
print("\nOriginal Triangle Coordinates:\n", triangle)
print("\nRotated Triangle (45°):\n", triangle_rot)

# Plotting
plt.figure(figsize=(6,6))
plt.plot(triangle[:,0], triangle[:,1], 'b-', label="Original")
plt.plot(triangle_rot[:,0], triangle_rot[:,1], 'r--', label="Rotated 45°")
plt.axis("equal")
plt.grid(True)
plt.legend()
plt.title("Exercise 1: Rotation of Triangle (45°)")
plt.show()
```

```
Rotation Matrix (45°):
 [[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]]

Original Vertex: [1. 0. 1.]
Matrix Multiplication:
 [[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]] @ [1. 0. 1.]
Resulting Vertex: [0.70710678 0.70710678 1.        ]

Original Triangle Coordinates:
 [[0.  0.  1. ]
 [1.  0.  1. ]
 [0.5 1.  1. ]
 [0.  0.  1. ]]

Rotated Triangle (45°):
 [[ 0.          0.          1.        ]
 [ 0.70710678  0.70710678  1.        ]
 [-0.35355339  1.06066017  1.        ]
 [ 0.          0.          1.        ]]
```



```
Rotation Matrix (45°):
 [[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]]

Original Vertex: [1. 0. 1.]
Matrix Multiplication:
 [[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]] @ [1. 0. 1.]
Resulting Vertex: [0.70710678 0.70710678 1.        ]

Original Triangle Coordinates:
 [[0.  0.  1. ]
 [1.  0.  1. ]
 [0.5 1.  1. ]
 [0.  0.  1. ]]

Rotated Triangle (45°):
 [[ 0.          0.          1.        ]
 [ 0.70710678  0.70710678  1.        ]
 [-0.35355339  1.06066017  1.        ]
 [ 0.          0.          1.        ]]
```



Exercise 2: Order Matters (Scaling vs Translation)

```
# Scale then Translate
rect_scaled = scale(rectangle, 2, 1.5)
rect_scaled_translated = translate(rect_scaled, 2, 1)

# Translate then Scale
rect_translated = translate(rectangle, 2, 1)
rect_translated_scaled = scale(rect_translated, 2, 1.5)

print("Scale -> Translate:\n", rect_scaled_translated)
print("\nTranslate -> Scale:\n", rect_translated_scaled)

# Plot
plt.figure(figsize=(8,6))
plt.plot(rectangle[:,0], rectangle[:,1], 'k-', label="Original")
plt.plot(rect_scaled_translated[:,0], rect_scaled_translated[:,1], 'r--', label="Scale then Translate")
plt.plot(rect_translated_scaled[:,0], rect_translated_scaled[:,1], 'g--', label="Translate then Scale")
plt.axis("equal")
plt.grid(True)
plt.legend()
plt.title("Exercise 2: Order of Transformations")
plt.show()
```

```
Scale -> Translate:
 [[2.  1.  1. ]
 [6.  1.  1. ]
 [6.  2.5 1. ]
 [2.  2.5 1. ]
 [2.  1.  1. ]]

Translate -> Scale:
 [[4.  1.5 1. ]
 [8.  1.5 1. ]
 [8.  3.  1. ]
 [4.  3.  1. ]
 [4.  1.5 1. ]]
```

Exercise 3: Combined Transform with Sliders

We'll use ipywidgets.

```
import ipywidgets as widgets
from ipywidgets import interact

def transform_and_plot(tx=0.0, ty=0.0, sx=1.0, sy=1.0, angle=0.0):
    # Apply transformations in order: Scale -> Rotate -> Translate
    transformed = scale(triangle, sx, sy)
    transformed = rotate(transformed, angle)
    transformed = translate(transformed, tx, ty)

    plt.figure(figsize=(6,6))
    plt.plot(triangle[:,0], triangle[:,1], 'b-', label="Original")
    plt.plot(transformed[:,0], transformed[:,1], 'r--', label="Transformed")
    plt.axis("equal")
    plt.grid(True)
    plt.legend()
    plt.show()

    print("Final Coordinates:\n", transformed)
```

```
interact(transform_and_plot,
         tx=widgets.FloatSlider(min=-5, max=5, step=0.5, value=2),
         ty=widgets.FloatSlider(min=-5, max=5, step=0.5, value=-1),
         sx=widgets.FloatSlider(min=0.5, max=3, step=0.5, value=2),
         sy=widgets.FloatSlider(min=0.5, max=3, step=0.5, value=2),
         angle=widgets.FloatSlider(min=-180, max=180, step=5, value=-30))
```

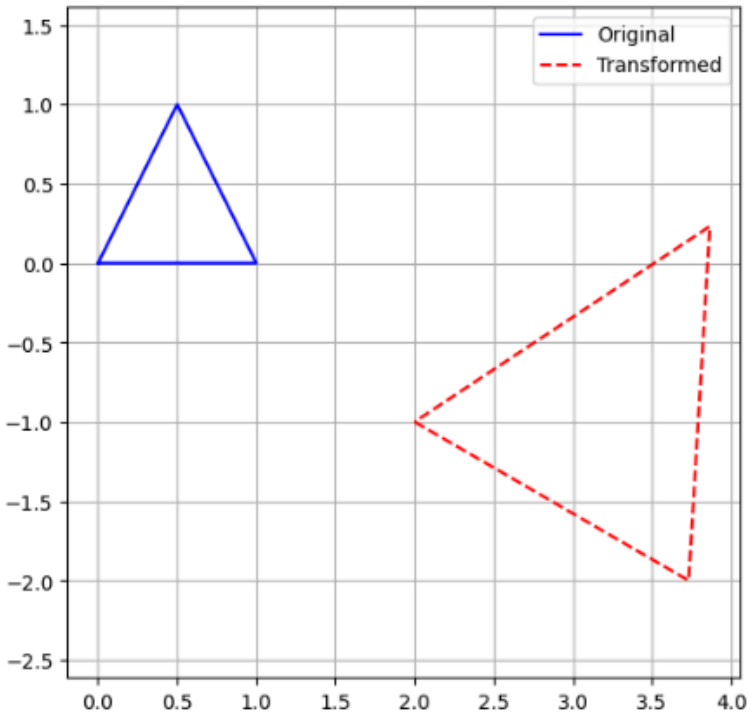
tx 2.00

ty -1.00

sx 2.00

sy 2.00

angle -30.00




```
Final Coordinates:
 [[ 2.         -1.          1.        ]
 [ 3.73205081 -2.          1.        ]
 [ 3.8660254   0.23205081  1.        ]
 [ 2.         -1.          1.        ]]
```

transform_and_plot

```
def transform_and_plot(tx=0.0, ty=0.0, sx=1.0, sy=1.0, angle=0.0)
```

<no docstring>

Exercise 4: Extra (Shear Transform)

```
def shear(points, shx, shy):
    """Shear transformation"""
    Sh = np.array([[1, shx, 0],
                   [shy, 1, 0],
                   [0, 0, 1]])
    return (Sh @ points.T).T

triangle_sheared = shear(triangle, 0.5, 0.2)

print("Sheared Triangle Coordinates:\n", triangle_sheared)

# Plot
plt.figure(figsize=(6,6))
plt.plot(triangle[:,0], triangle[:,1], 'b-', label="Original")
plt.plot(triangle_sheared[:,0], triangle_sheared[:,1], 'r--', label="Sheared")
plt.axis("equal")
plt.grid(True)
plt.legend()
plt.title("Exercise 4: Shear Transformation")
plt.show()
```

```
Sheared Triangle Coordinates:
 [[0.  0.  1. ]
 [1.  0.2 1. ]
 [1.  1.1 1. ]
 [0.  0.  1. ]]
```