Bangladesh University of Business and Technology

Lab report - 1

Lab topic: Implementation of DDA Line Drawing Algorithm in Python .Here in the report lab task and home task is included

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 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.

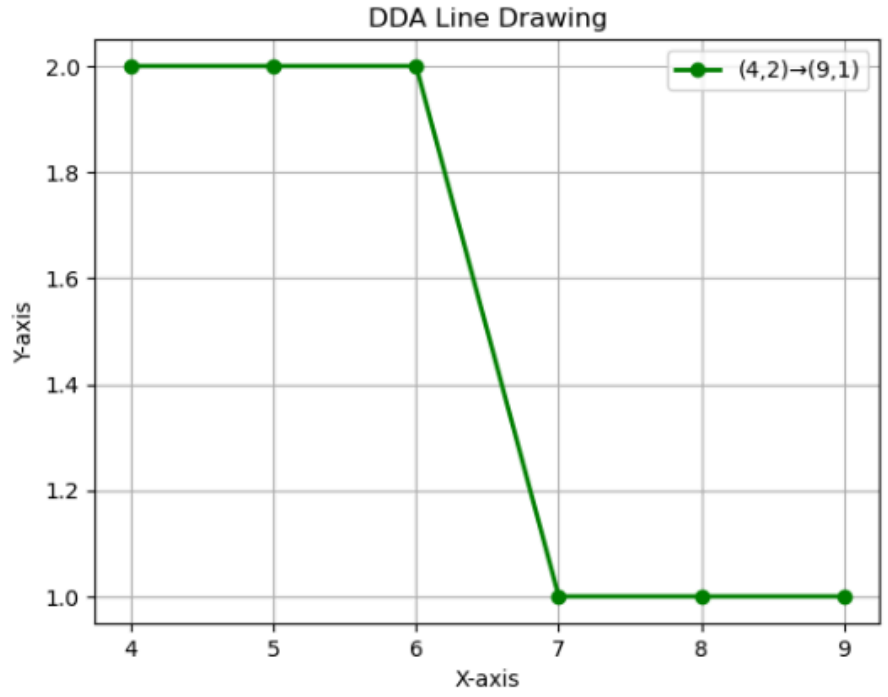
3. Handling the Plotting & Visualization

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

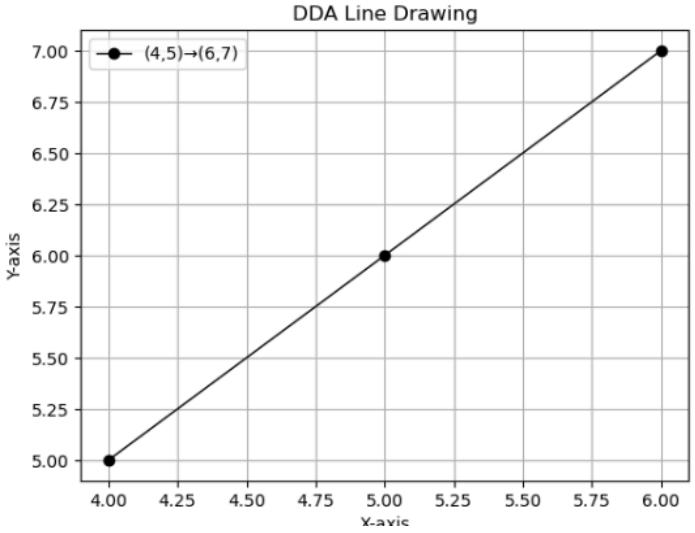
- `plt.title()` sets the chart title.
- `plt.xlabel()` and `plt.ylabel()` label the axes.
- `plt.legend()` creates a label for each line, showing start → end coordinates.
- `plt.grid(True)` shows a background grid (like raster pixels).
- `plt.show()` finally renders the plot with all drawn lines.

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
```

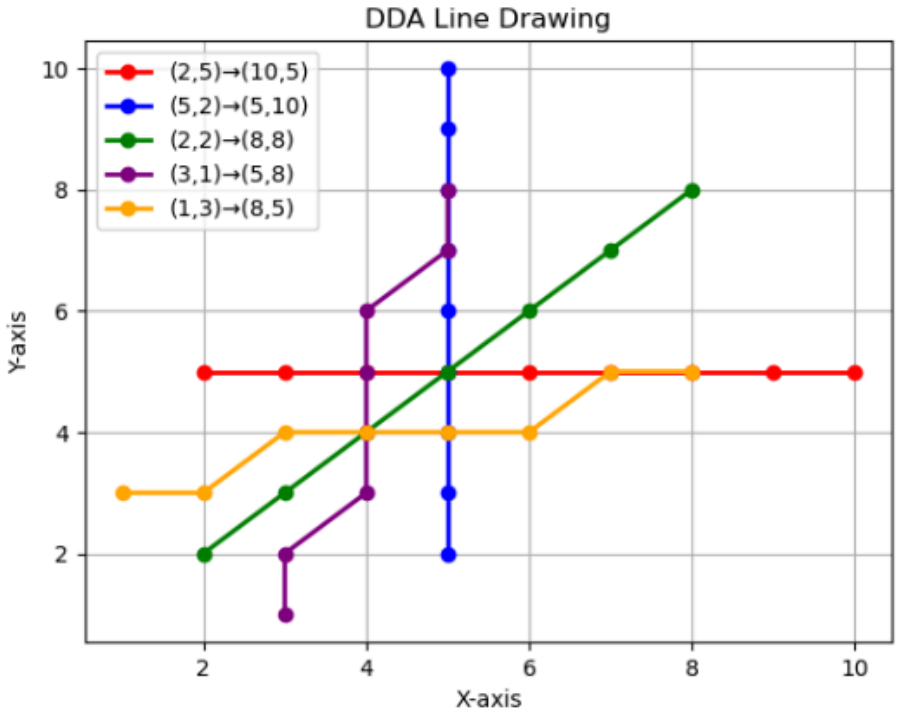



```
Choose mode:
1. Draw line with user input applying DDA
2. Test with different types of lines like Horizontal ,Vertical , Diagonal ,Steep and shallow slopes
Enter choice:  1
Enter x1:  4
Enter y1:  5
Enter x2:  6
Enter y2:  7
Enter line color (e.g., red, blue, green):  black
Enter line thickness (e.g.,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



<u>Practice Exercises (Home Tasks)</u>

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.

<u>Solution of home task(code):</u>

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')
```

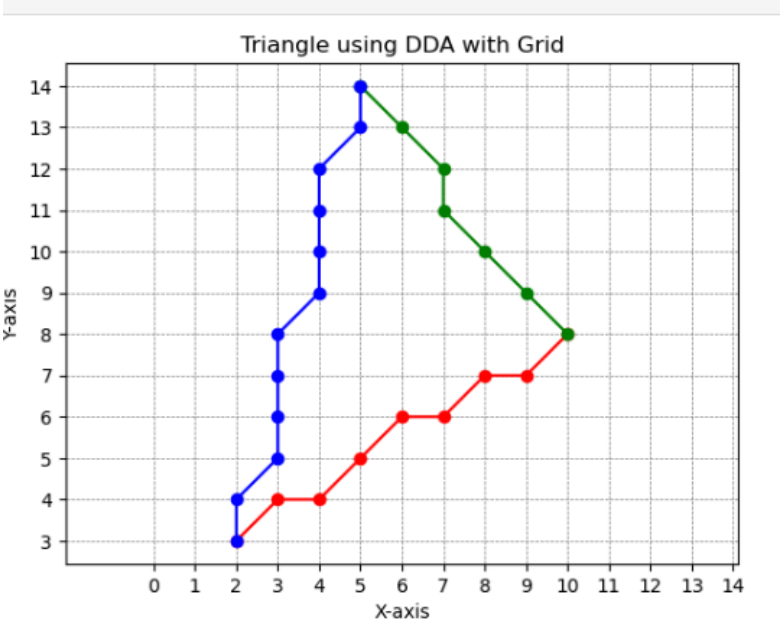
```
    plt.title('Triangle using DDA with Grid')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.axis('equal')
    plt.show()

# 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:

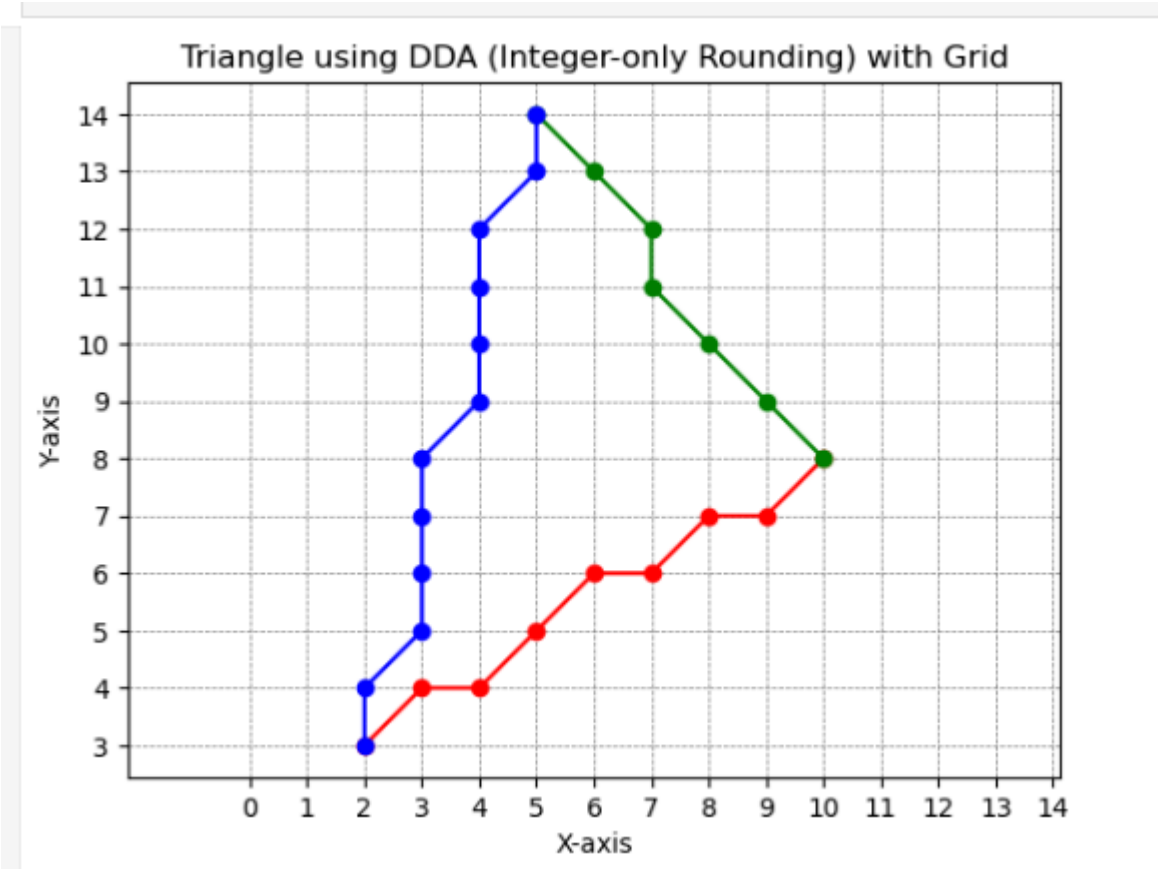


Task 2 code is given below adding with task 3 (Draw a grid and visualize line drawing over it using matplotlib.)in it

```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.