Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 2
[Code No: COMP 342]

Submitted by:

Nischal Subedi (53)

Group: CS60 (III/I)

Submitted to:
Mr. Dhiraj Shrestha
Department of Computer Science and Engineering

Submission Date: 01/01/2026

Q1. Implement Digital Differential Analyzer Line drawing algorithm.

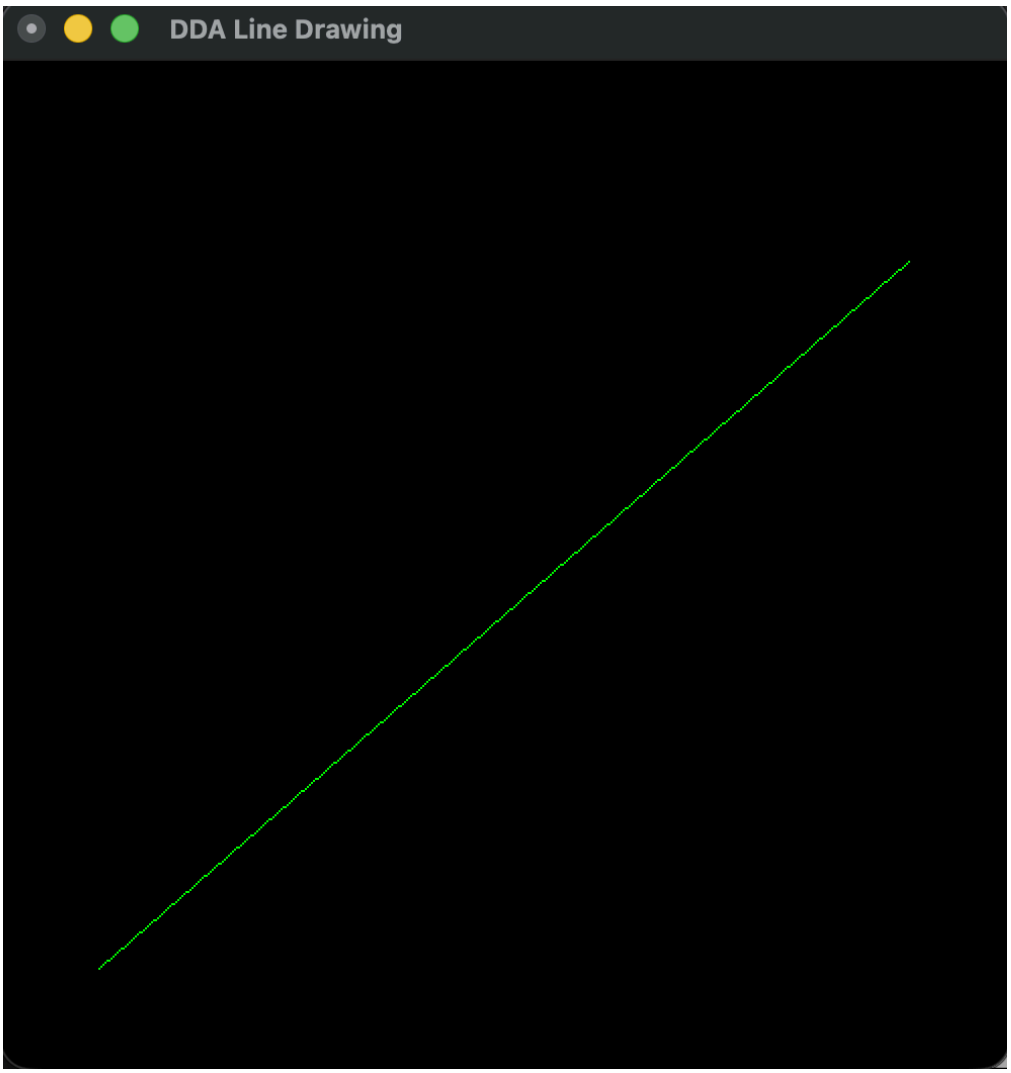
The Digital Differential Analyzer (DDA) algorithm draws a straight line by calculating incremental steps in x and y directions. It uses floating-point arithmetic and rounds the calculated points to the nearest pixel.

Algorithm

1. Start with the two endpoints (x0, y0) and (x1, y1).
2. Compute differences: dx = x1 - x0 and dy = y1 - y0.
3. Determine number of steps: step_size = max(|dx|, |dy|).
4. Compute the increments: x_inc = dx / step_size and y_inc = dy / step_size.
5. Initialize: x = x0, y = y0.
6. Plot initial point: plot(round(x), round(y)).
7. Repeat for "step_size" times:
    - x = x + x_inc
    - y = y + y_inc
    - Plot (round(x), round(y))

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def dda_line(x1, y1, x2, y2):
   dx, dy = x2 - x1, y2 - y1
   steps = int(max(abs(dx), abs(dy)))
   x_inc = dx / steps
   y_inc = dy / steps

   x, y = x1, y1
   glBegin(GL_POINTS)
   for _ in range(steps + 1):
       glVertex2i(round(x), round(y))
       x += x_inc
       y += y_inc
   glEnd()

def display():
   glClear(GL_COLOR_BUFFER_BIT)
   glColor3f(0.0, 1.0, 0.0) # Green Line
```

```
    dda_line(50, 50, 450, 400)
    glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"DDA Line Drawing")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



Q2. Implement Bresenham Line Drawing algorithm for both slopes(|m|<1 and |m|>=1).

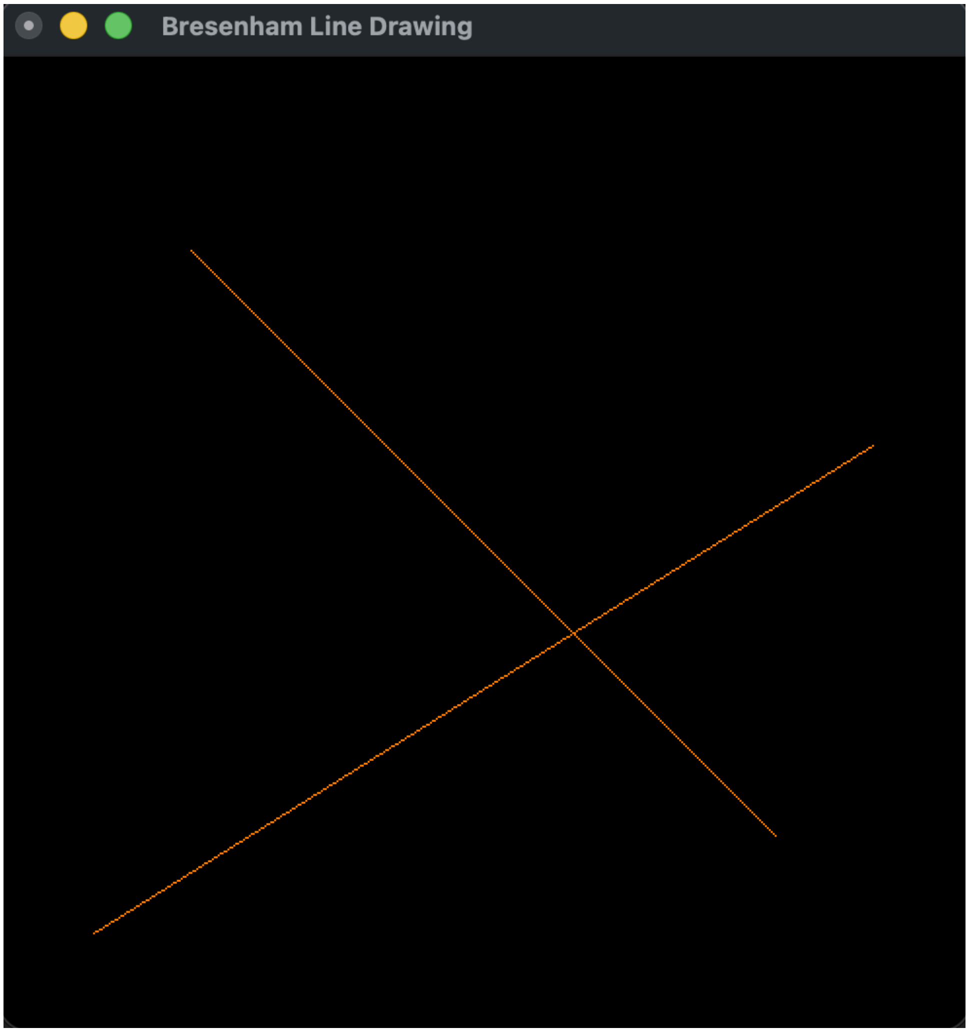
Bresenham's algorithm is an efficient line drawing technique that uses only integer calculations. It selects the nearest pixel based on a decision parameter, making it faster than DDA.

Algorithm

1. Start with two endpoints: (x0, y0) and (x1, y1).
2. Compute the differences: dx = |x1 - x0| and dy = |y1 - y0|.
3. Determine direction: If x1 >= x0 then sx = 1 else sx = -1; If y1 >= y0 then sy = 1 else sy = -1.
4. Initialize starting points: x = x0, y = y0.
5. Case 1: |m| < 1 (i.e., dx > dy):
    - Initialize decision parameter: p = 2dy - dx.
    - For each step from 0 to dx:
        - Plot (x, y).
        - If p >= 0: y = y + sy, p = p + 2dy - 2dx.
        - Else: p = p + 2dy.
        - Update x = x + sx.
6. Case 2: |m| >= 1 (i.e., dy >= dx):
    - Initialize decision parameter: p = 2dx - dy.
    - For each step from 0 to dy:
        - Plot (x, y).
        - If p >= 0: x = x + sx, p = p + 2dx - 2dy.
        - Else: p = p + 2dx.
        - Update y = y + sy.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def bresenham_line(x1, y1, x2, y2):
   dx, dy = abs(x2 - x1), abs(y2 - y1)
   sx = 1 if x1 < x2 else -1
```

```
    sy = 1 if y1 < y2 else -1
    err = dx - dy

    glBegin(GL_POINTS)
    while True:
        glVertex2i(x1, y1)
        if x1 == x2 and y1 == y2: break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x1 += sx
        if e2 < dx:
            err += dx
            y1 += sy
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 0.5, 0.0) # Orange Line
    bresenham_line(100, 400, 400, 100) # Negative slope
    bresenham_line(50, 50, 450, 300)   # Positive slope
    glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"Bresenham Line Drawing")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



Q3. Write a Program to implement mid- point Circle Drawing Algorithm

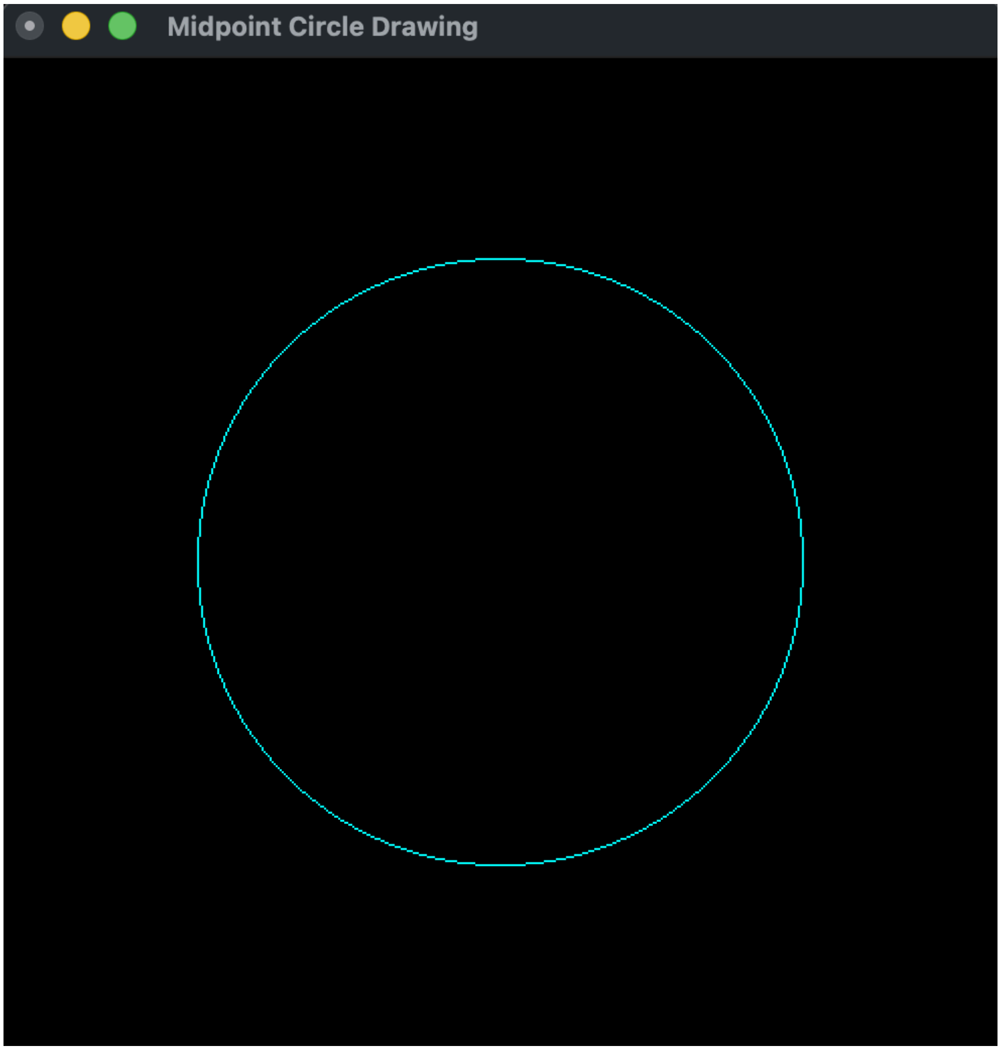
The midpoint circle drawing algorithm determines the points needed for rasterizing a circle. It calculates perimeter points in the first octant and uses symmetry to print mirror points in the other seven octants.

Algorithm

1. Input center (xc, yc) and radius r.
2. Initialize x = 0, y = r.
3. Initialize decision parameter: p = 1 - r.
4. Repeat while x <= y:

- Plot the eight symmetric points: (xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y), (xc+y, yc+x), (xc-y, yc+x), (xc+y, yc-x), (xc-y, yc-x).
- If decision parameter p < 0:
    - p = p + 2x + 3.
- Else:
    - p = p + 2(x - y) + 5, y = y - 1.
- Increment x = x + 1.

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def midpoint_circle(xc, yc, r):
   x, y = 0, r
   p = 1 - r
   glBegin(GL_POINTS)
   while x <= y:
       # 8-way symmetry
       for dx, dy in [(x,y), (y,x), (y,-x), (x,-y), (-x,-y), (-y,-x), (-y,x), (-x,y)]:
           glVertex2i(xc + dx, yc + dy)
       x += 1
       if p < 0:
           p += 2 * x + 1
       else:
           y -= 1
           p += 2 * (x - y) + 1
   glEnd()

def display():
   glClear(GL_COLOR_BUFFER_BIT)
   glColor3f(0.0, 1.0, 1.0) # Cyan Circle
   midpoint_circle(250, 250, 150)
   glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"Midpoint Circle Drawing")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



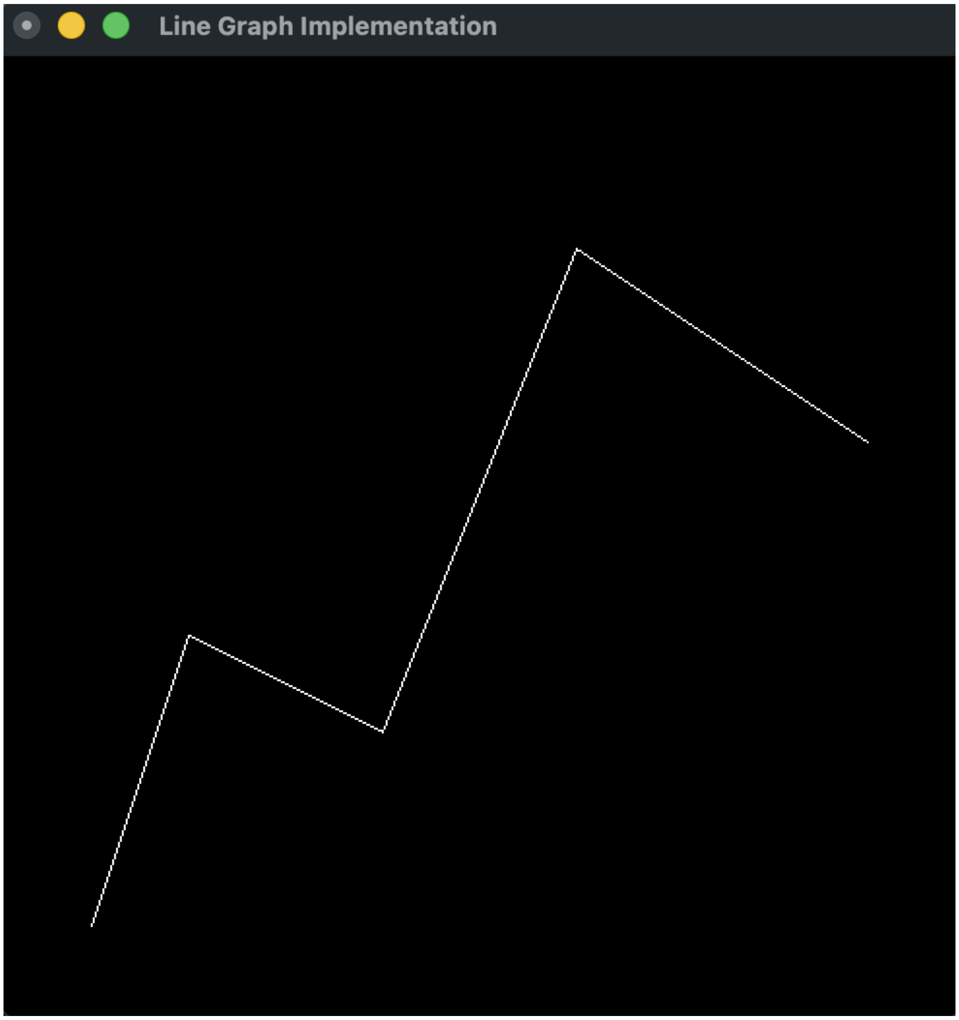
Q4. Implement the Line Function (DDA/BLA) for generating a line graph of a given set of data

Algorithm

1. Define dataset size n.
2. Calculate horizontal gap between points: Gap = (Window_width - 2 * margin) / (n - 1).
3. For each index i from 0 to n-1:
    - Assign xi = margin + i * gap.
    - Generate random or given vertical value yi.

   ○ Store points as pairs (xi, yi).
4. Draw lines between consecutive pairs (xi, yi) and (xi+1, yi+1) using the DDA algorithm steps.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

# Sample Data: (x, y)
dataset = [(50, 50), (100, 200), (200, 150), (300, 400), (450, 300)]

def bresenham_line(x1, y1, x2, y2):
   dx, dy = abs(x2-x1), abs(y2-y1)
   sx, sy = (1 if x1<x2 else -1), (1 if y1<y2 else -1)
   err = dx - dy
   glBegin(GL_POINTS)
   while True:
       glVertex2i(x1, y1)
       if x1 == x2 and y1 == y2: break
       e2 = 2 * err
       if e2 > -dy: err -= dy; x1 += sx
       if e2 < dx: err += dx; y1 += sy
   glEnd()

def display():
   glClear(GL_COLOR_BUFFER_BIT)
   glColor3f(1.0, 1.0, 1.0) # White lines
   for i in range(len(dataset) - 1):
       p1, p2 = dataset[i], dataset[i+1]
       bresenham_line(p1[0], p1[1], p2[0], p2[1])
   glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"Line Graph Implementation")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



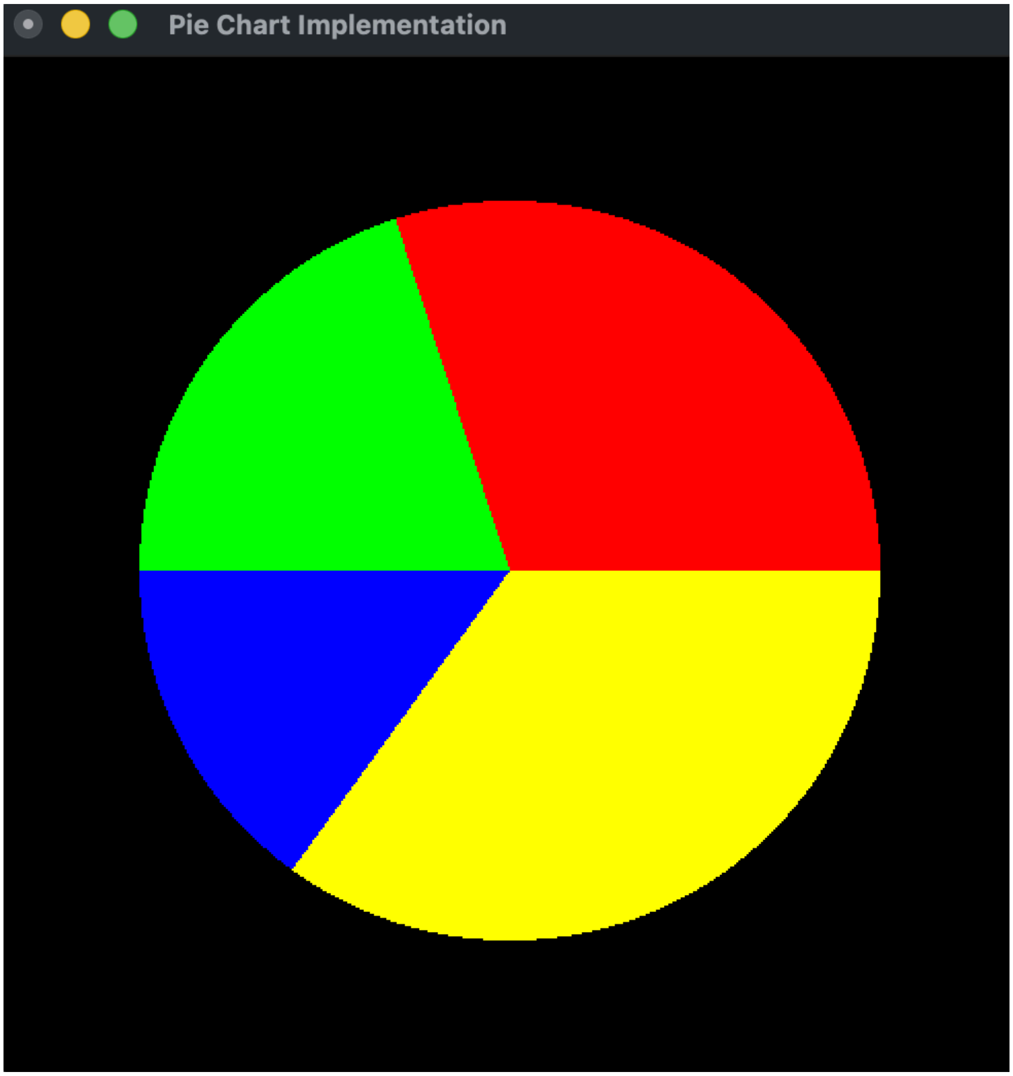
Q5. Implement the Pie chart

Algorithm

1. Initialize parameters: center (cx, cy), radius r, and data values.
2. Generate data values and compute total sum T.
3. Convert each value to an angle: angle = 360 * value / T.
4. Initialize start_angle = 0.
5. For each data slice:
    - Calculate end_angle = start_angle + angle.

- Generate boundary points for the arc using: x = cx + r * cos(theta) and y = cy + r * sin(theta).
- Draw a filled wedge using a triangle fan from the center (cx, cy) to the arc points.
- Set start_angle = end_angle for the next slice.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import math

data = [30, 20, 15, 35] # Percentages
colors = [(1,0,0), (0,1,0), (0,0,1), (1,1,0)]

def draw_pie(xc, yc, r):
   total = sum(data)
   start_angle = 0
   for i, val in enumerate(data):
       sweep = (val / total) * 2 * math.pi
       glColor3fv(colors[i % len(colors)])
       glBegin(GL_TRIANGLE_FAN)
       glVertex2f(xc, yc)
       for j in range(51):
           theta = start_angle + (sweep * j / 50)
           glVertex2f(xc + r * math.cos(theta), yc + r * math.sin(theta))
       glEnd()
       start_angle += sweep

def display():
   glClear(GL_COLOR_BUFFER_BIT)
   draw_pie(250, 250, 180)
   glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"Pie Chart Implementation")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



Q6. Implement midpoint Ellipse drawing Algorithm

The midpoint ellipse algorithm draws an ellipse using symmetry and region-based decision parameters.

Algorithm

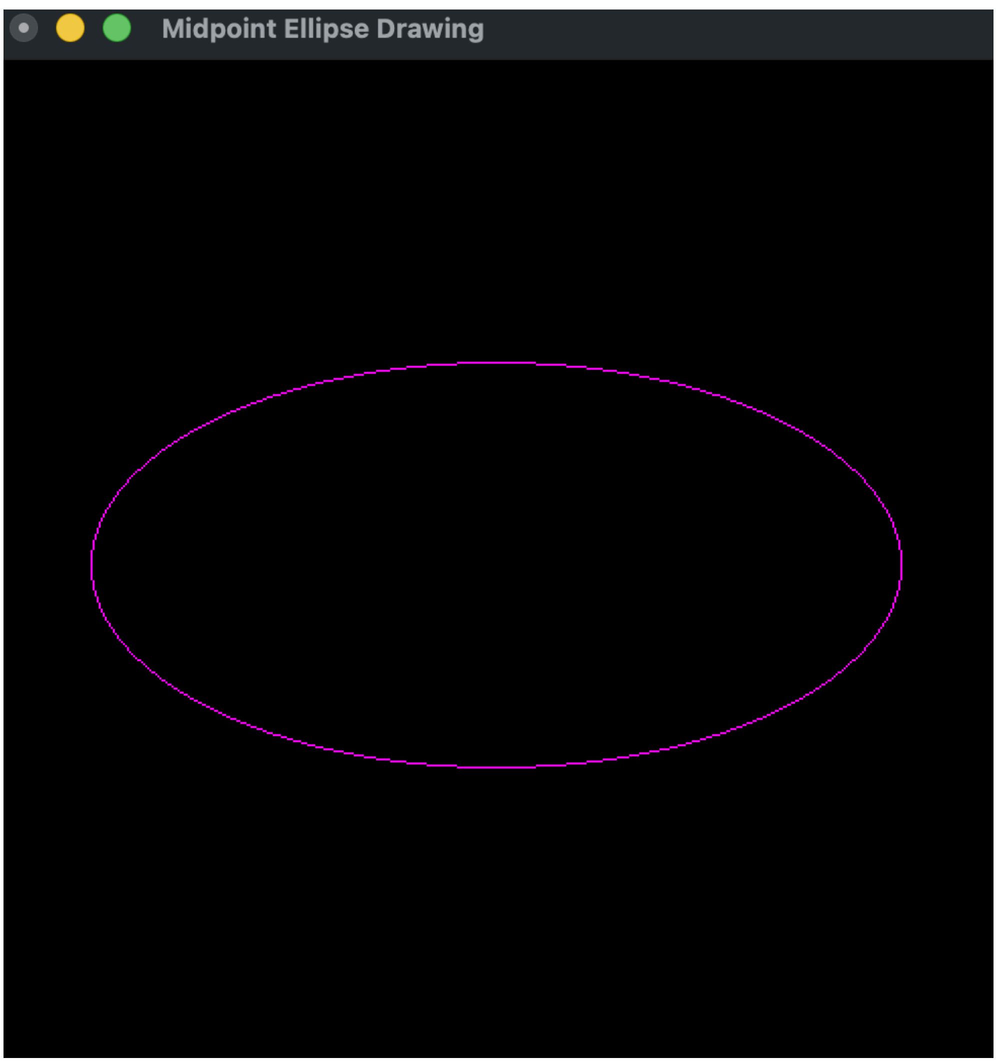
1. Input center (xc, yc) and radii rx, ry.
2. Initialize x = 0, y = ry.
3. Region 1 Processing (slope < 1):
    - Initialize decision parameter: p1 = ry*ry - rx*rx*ry + 0.25 * rx*rx.
    - While (2*ry*ry*x < 2*rx*rx*y):

        - Plot four symmetric points: (xc+x, yc+y), (xc-x, yc+y), (xc+x, yc-y), (xc-x, yc-y).
        - x = x + 1.
        - If p1 < 0: p1 = p1 + 2*ry*ry*x* + *ry*ry.
        - Else: y = y - 1, p1 = p1 + 2*ry*ry*x* - 2rx*rx*y + ry*ry.

4. Region 2 Processing (slope >= 1):
    - Initialize decision parameter: p2 = ry*ry*(x+0.5)*(x+0.5)* + *rx*x*(y-1)*(y-1)* - *rx*rx*ry*ry.
    - While (y >= 0):
        - Plot four symmetric points.
        - y = y - 1.
        - If p2 > 0: p2 = p2 - 2*rx*rx*y* + *rx*rx.
        - Else: x = x + 1, p2 = p2 + 2*ry*ry*x* - 2rx*rx*y + rx*rx.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def midpoint_ellipse(xc, yc, rx, ry):
   x, y = 0, ry
   # Region 1
   p1 = (ry**2) - (rx**2 * ry) + (0.25 * rx**2)
   glBegin(GL_POINTS)
   while (2 * ry**2 * x) <= (2 * rx**2 * y):
       for dx, dy in [(x,y), (-x,y), (x,-y), (-x,-y)]:
           glVertex2i(xc + dx, yc + dy)
       x += 1
       if p1 < 0:
           p1 += 2 * ry**2 * x + ry**2
       else:
           y -= 1
           p1 += 2 * ry**2 * x - 2 * rx**2 * y + ry**2
   # Region 2
   p2 = (ry**2 * (x + 0.5)**2) + (rx**2 * (y - 1)**2) - (rx**2 * ry**2)
   while y >= 0:
       for dx, dy in [(x,y), (-x,y), (x,-y), (-x,-y)]:
           glVertex2i(xc + dx, yc + dy)
       y -= 1
       if p2 > 0:
           p2 += rx**2 - 2 * rx**2 * y
       else:
```

```
            x += 1
            p2 += 2 * ry**2 * x - 2 * rx**2 * y + rx**2
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 0.0, 1.0) # Magenta Ellipse
    midpoint_ellipse(250, 250, 200, 100)
    glFlush()

glutInit()
glutInitWindowSize(500, 500)
glutCreateWindow(b"Midpoint Ellipse Drawing")
gluOrtho2D(0, 500, 0, 500)
glutDisplayFunc(display)
glutMainLoop()
```



Conclusion:

In this lab, fundamental computer graphics algorithms were implemented using OpenGL and Python. Line drawing algorithms like DDA and Bresenham were studied for efficient rasterization, while midpoint algorithms were used for circles and ellipses using symmetry and decision parameters. Graphical representations like line graphs and pie charts were also generated. These implementations provided practical exposure to converting mathematical models into pixel-based 2D graphics.