Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 4

[Course Code: COMP 342]

Submitted by:
Jatin Madhikarmi
Roll No. 32

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

1 Introduction

Line clipping is the process of removing portions of a line segment that lie outside a specific rectangular area, known as the clip window. The Cohen-Sutherland algorithm is one of the most popular line-clipping algorithms. it works by dividing the 2D plane into nine regions. Each region is assigned a 4-bit binary code (Outcode) based on its position relative to the clipping window boundaries (Left, Right, Bottom, Top).

Algorithm Steps

The algorithm follows these primary steps:

1. Assign a 4-bit region code to both endpoints of the line.
2. **Case 1 (Trivial Accept):** If both codes are 0000, the line is entirely inside the window.
3. **Case 2 (Trivial Reject):** If the bitwise AND of the two codes is not zero, the line is entirely outside.
4. **Case 3 (Clip):** If neither of the above, find the intersection of the line with a window boundary, replace the outside point with the intersection point, and repeat.

Code Implementation (PyOpenGL)

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

# Define region codes
INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000

# Clipping window boundaries
x_max, y_max = 0.5, 0.5
x_min, y_min = -0.5, -0.5

def compute_code(x, y):
    code = INSIDE
    if x < x_min: code |= LEFT
    elif x > x_max: code |= RIGHT
    if y < y_min: code |= BOTTOM
    elif y > y_max: code |= TOP
    return code
```

Figure 1: Implemenation of Cohen Sutherland Line Clipping Algorithm Source Code 1

```python
def cohen_sutherland_clip(x1, y1, x2, y2):
    code1 = compute_code(x1, y1)
    code2 = compute_code(x2, y2)
    accept = False

    while True:
        if code1 == 0 and code2 == 0:
            accept = True
            break
        elif (code1 & code2) != 0:
            break
        else:
            # Line needs clipping
            x, y = 0.0, 0.0
            code_out = code1 if code1 != 0 else code2

            if code_out & TOP:
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
                y = y_max
            elif code_out & BOTTOM:
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
                y = y_min
            elif code_out & RIGHT:
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
                x = x_max
            elif code_out & LEFT:
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
                x = x_min

            if code_out == code1:
                x1, y1 = x, y
                code1 = compute_code(x1, y1)
            else:
                x2, y2 = x, y
                code2 = compute_code(x2, y2)

    if accept:
        return [(x1, y1), (x2, y2)]
    return None
```

Figure 2: Implemenation of Cohen Sutherland Line Clipping Algorithm Source Code 2

```python
def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Draw Clipping Window (White)
    glColor3f(1.0, 1.0, 1.0)
    glBegin(GL_LINE_LOOP)
    glVertex2f(x_min, y_min); glVertex2f(x_max, y_min)
    glVertex2f(x_max, y_max); glVertex2f(x_min, y_max)
    glEnd()

    # Original line coordinates
    p1, p2 = (-0.8, -0.2), (0.7, 0.9)

    # Draw original line (Red - faint)
    glColor3f(1.0, 0.0, 0.0)
    glBegin(GL_LINES)
    glVertex2f(*p1); glVertex2f(*p2)
    glEnd()

    # Apply clipping
    clipped_line = cohen_sutherland_clip(p1[0], p1[1], p2[0], p2[1])

    # Draw clipped line (Green)
    if clipped_line:
        glColor3f(0.0, 1.0, 0.0)
        glBegin(GL_LINES)
        glVertex2f(*clipped_line[0]); glVertex2f(*clipped_line[1])
        glEnd()

    glFlush()

def main():
    glutInit()
    glutCreateWindow(b"Cohen-Sutherland Clipping")
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 3: Implemenation of Cohen Sutherland Line Clipping Algorithm Source Code 3

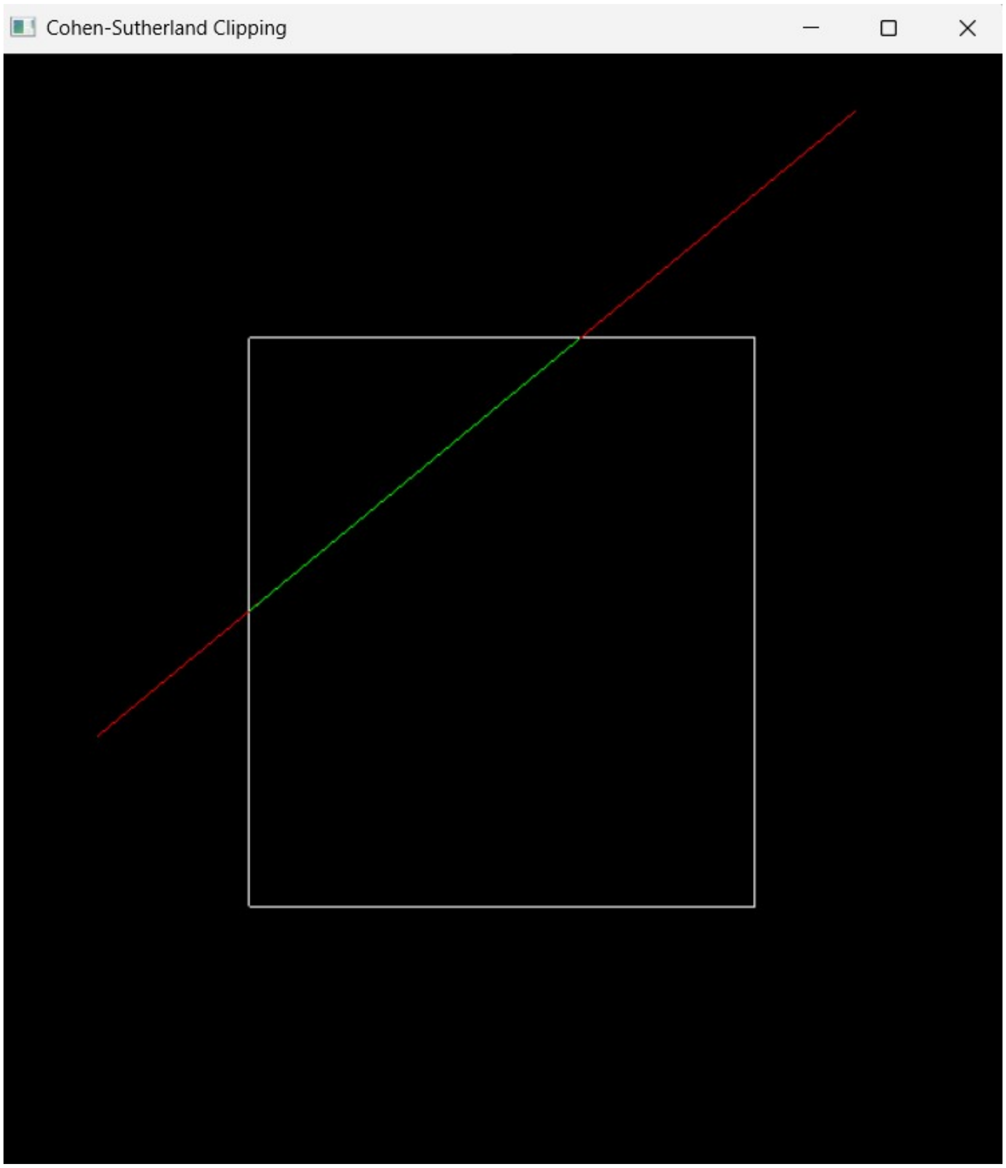

Figure 4: Output of Cohen Sutherland Line Clipping Algorithm

Conclusion

The Cohen-Sutherland algorithm is highly efficient because it uses bitwise operations to quickly perform trivial accepts and rejects. While it is excellent for simple rectangular clipping, other algorithms like Liang-Barsky may be faster for more complex scenarios. However, Cohen-Sutherland remains a fundamental concept in computer graphics education.

2 Introduction

The Liang-Barsky algorithm is a line-clipping algorithm that uses the parametric equation of a line and inequalities to determine the visible portion of a line segment. It is significantly faster than the Cohen-Sutherland algorithm because it avoids the repetitive clipping process by calculating the entry and exit points of the line relative to the clipping boundaries.

Algorithm Steps

A line is defined parametrically as: $x = x_1 + t \cdot \Delta x$
$y = y_1 + t \cdot \Delta y$
where $0 \leq t \leq 1$.

The algorithm defines four values of p_k and q_k for the boundaries:

1. Define $p_1 = -\Delta x, q_1 = x_1 - x_{min}$ (Left)
2. Define $p_2 = \Delta x, q_2 = x_{max} - x_1$ (Right)
3. Define $p_3 = -\Delta y, q_3 = y_1 - y_{min}$ (Bottom)
4. Define $p_4 = \Delta y, q_4 = y_{max} - y_1$ (Top)
5. For each k, if $p_k < 0$, calculate t as a potential entry point; if $p_k > 0$, calculate t as a potential exit point.
6. If $t_{entry} < t_{exit}$, the line segment is visible.

Code Implementation (PyOpenGL)

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

# Clipping window boundaries
x_min, y_min = -0.5, -0.5
x_max, y_max = 0.5, 0.5

def liang_barsky_clip(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1

    # p[k] and q[k] values for Left, Right, Bottom, Top
    p = [-dx, dx, -dy, dy]
    q = [x1 - x_min, x_max - x1, y1 - y_min, y_max - y1]

    t1 = 0.0
    t2 = 1.0

    for i in range(4):
        if p[i] == 0:
            if q[i] < 0:
                return None  # Parallel and outside
        else:
            t = q[i] / p[i]
            if p[i] < 0:
                if t > t1: t1 = t
            else:
                if t < t2: t2 = t

    if t1 < t2:
        nx1 = x1 + t1 * dx
        ny1 = y1 + t1 * dy
        nx2 = x1 + t2 * dx
        ny2 = y1 + t2 * dy
        return [(nx1, ny1), (nx2, ny2)]

    return None
```

Figure 5: Implemenation of Liang Barsky Line Clipping Algorithm Source Code 1

```python
def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Draw Clipping Window (White)
    glColor3f(1.0, 1.0, 1.0)
    glBegin(GL_LINE_LOOP)
    glVertex2f(x_min, y_min); glVertex2f(x_max, y_min)
    glVertex2f(x_max, y_max); glVertex2f(x_min, y_max)
    glEnd()

    # Original line coordinates
    p1, p2 = (-0.9, -0.4), (0.8, 0.6)

    # Draw original line (Red - faint/rejected part)
    glColor3f(1.0, 0.0, 0.0)
    glBegin(GL_LINES)
    glVertex2f(*p1); glVertex2f(*p2)
    glEnd()

    # Apply clipping
    clipped_line = liang_barsky_clip(p1[0], p1[1], p2[0], p2[1])

    # Draw clipped line (Cyan for Liang-Barsky)
    if clipped_line:
        glColor3f(0.0, 1.0, 1.0)
        glBegin(GL_LINES)
        glVertex2f(*clipped_line[0]); glVertex2f(*clipped_line[1])
        glEnd()

    glFlush()
```

Figure 6: Implemenation of Liang Barsky Line Clipping Algorithm Source Code 2

```python
def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow(b"Liang-Barsky Line Clipping")
    glClearColor(0.0, 0.0, 0.0, 1.0)
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 7: Implemenation of Liang Barsky Line Clipping Algorithm Source Code 3

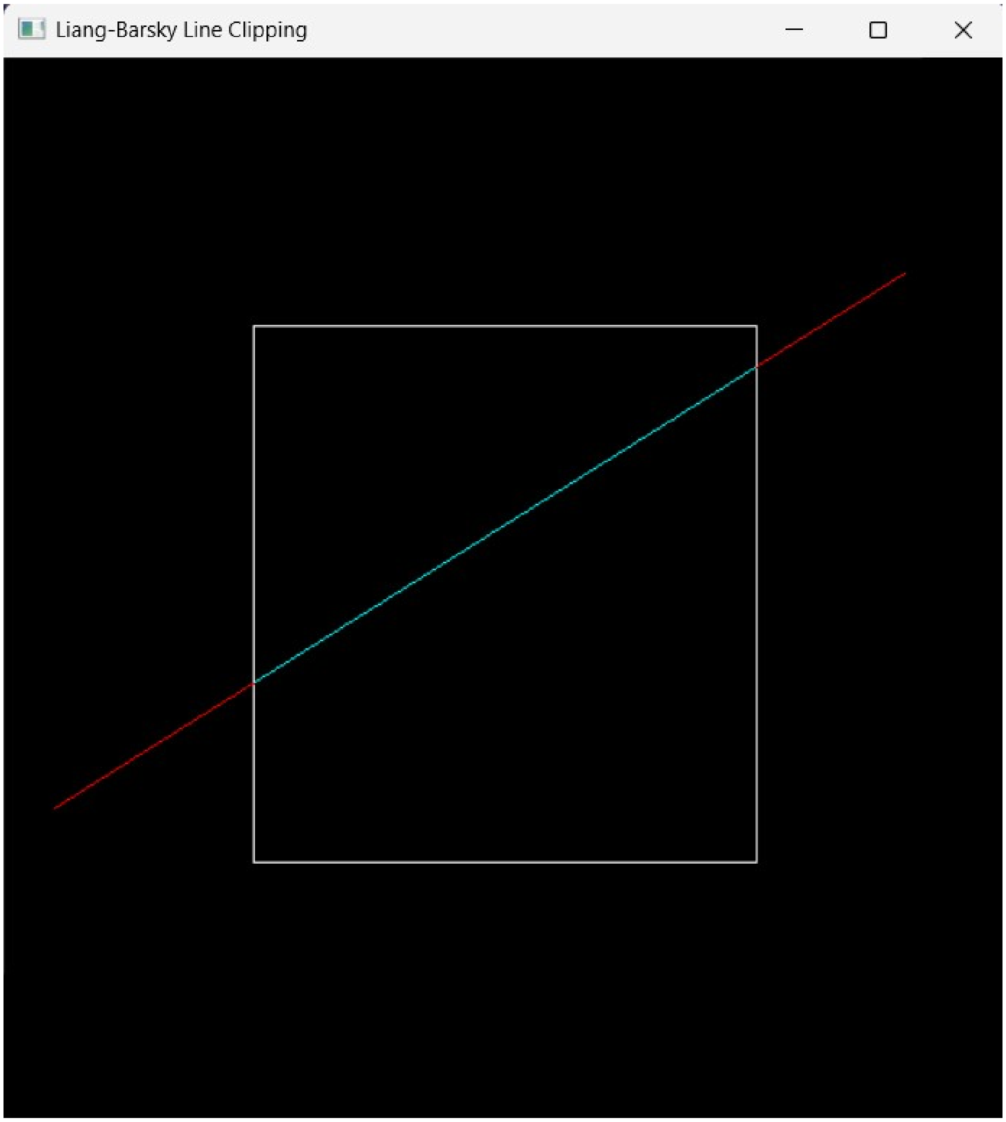

Figure 8: Output of Liang Barsky Line Clipping Algorithm

Conclusion

The Liang-Barsky algorithm is more efficient than Cohen-Sutherland because it minimizes the number of intersections that need to be calculated. By comparing parametric values t, it can determine if a line is entirely outside the window earlier in the computation process, making it a preferred choice for high-performance graphics systems.

3 Introduction

The Sutherland-Hodgeman algorithm is used for clipping polygons against a convex clipping window. Unlike line clipping, which treats segments independently, this algorithm processes the polygon as a sequence of vertices. It outputs a new list of vertices that define the portion of the polygon lying inside the clipping region.

Algorithm Steps

The algorithm processes the polygon by clipping it against each of the four window boundaries (Left, Right, Top, Bottom) sequentially:

1. For each boundary, iterate through the polygon edges (v_1, v_2).
2. **Case 1:** Both vertices are inside $\rightarrow$ Add v_2 to the output list.
3. **Case 2:** v_1 inside, v_2 outside $\rightarrow$ Add the intersection point to the output list.
4. **Case 3:** v_1 outside, v_2 inside $\rightarrow$ Add the intersection point AND v_2 to the output list.
5. **Case 4:** Both outside $\rightarrow$ Add nothing.

Code Implementation (PyOpenGL)

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

# Clipping window boundaries
x_min, y_min = -0.4, -0.4
x_max, y_max = 0.4, 0.4

def get_intersection(p1, p2, edge_type):
    x1, y1 = p1
    x2, y2 = p2

    if edge_type == "LEFT":
        x = x_min
        y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
    elif edge_type == "RIGHT":
        x = x_max
        y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
    elif edge_type == "BOTTOM":
        y = y_min
        x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
    elif edge_type == "TOP":
        y = y_max
        x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
    return (x, y)

def is_inside(p, edge_type):
    if edge_type == "LEFT": return p[0] >= x_min
    if edge_type == "RIGHT": return p[0] <= x_max
    if edge_type == "BOTTOM": return p[1] >= y_min
    if edge_type == "TOP": return p[1] <= y_max
    return False
```

Figure 9: Sutherland Hodgemann Polygon Clipping Algorithm Source Code 1

```python
def clip(polygon, edge_type):
    new_polygon = []
    for i in range(len(polygon)):
        p1 = polygon[i]
        p2 = polygon[(i + 1) % len(polygon)]

        in1 = is_inside(p1, edge_type)
        in2 = is_inside(p2, edge_type)

        if in1 and in2:
            new_polygon.append(p2)
        elif in1 and not in2:
            new_polygon.append(get_intersection(p1, p2, edge_type))
        elif not in1 and in2:
            new_polygon.append(get_intersection(p1, p2, edge_type))
            new_polygon.append(p2)

    return new_polygon
```

Figure 10: Sutherland Hodgemann Polygon Clipping Algorithm Source Code 2

```python
def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Draw Clipping Window (White)
    glColor3f(1.0, 1.0, 1.0)
    glBegin(GL_LINE_LOOP)
    glVertex2f(x_min, y_min); glVertex2f(x_max, y_min)
    glVertex2f(x_max, y_max); glVertex2f(x_min, y_max)
    glEnd()
    # Original Polygon (Red Outline)
    poly = [(-0.6, -0.2), (0.0, 0.7), (0.5, -0.3)]
    glColor3f(1.0, 0.0, 0.0)
    glBegin(GL_LINE_LOOP)
    for v in poly: glVertex2f(*v)
    glEnd()

    # Apply Clipping for all 4 edges
    clipped_poly = poly
    for edge in ["LEFT", "RIGHT", "BOTTOM", "TOP"]:
        clipped_poly = clip(clipped_poly, edge)

    # Draw Clipped Polygon (Yellow Solid)
    if clipped_poly:
        glColor3f(1.0, 1.0, 0.0)
        glBegin(GL_POLYGON)
        for v in clipped_poly: glVertex2f(*v)
        glEnd()

    glFlush()

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"Sutherland-Hodgeman Polygon Clipping")
    glClearColor(0.0, 0.0, 0.0, 1.0)
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 11: Sutherland Hodgemann Polygon CLipping Algorithm Source Code 3

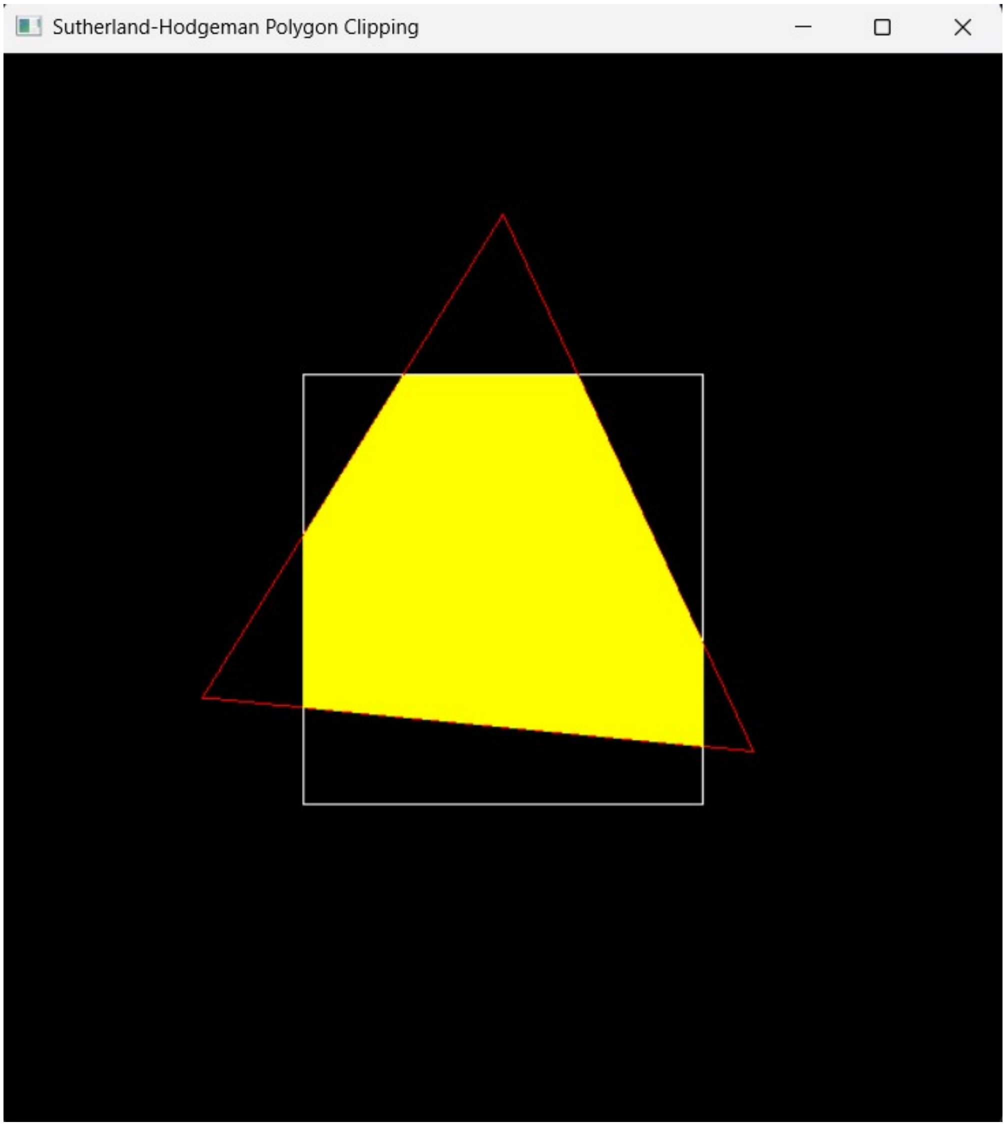

Figure 12: Output of Sutherland Hodgemann Polygon CLipping Algorithm

Conclusion

Sutherland-Hodgeman is highly effective for convex clipping windows. However, it has a limitation: when clipping a concave polygon, it may produce ”ghost” lines (extra edges) connecting separate regions. For such cases, the Weiler-Atherton algorithm is typically preferred.