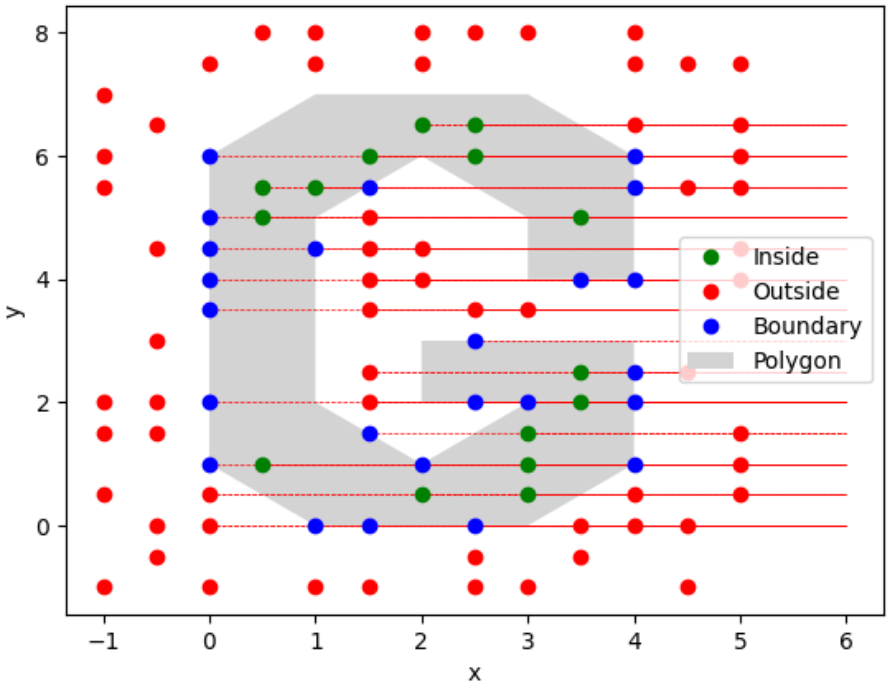
Point-in-Polygon Test Algorithm



Introduction

This algorithm is to do a Point-in-Polygon Test, which includes three parts:

Write a main_from_file.py program

The program can:

1) Read a list of coordinates from a CSV file and create a polygon

2) Read a list of coordinates from a file and create a list of testing points;

3) Categorize these points into: “inside”, “outside” and “boundary”;

4) Output the result of each point in a CSV file;

5) Plot the points and polygon in a plot window.

Write a mian_from_user.py program

The program can:

1) Read a list of coordinates from a CSV file and create a polygon;

2) Read a point from keyboard;

3) Categorize this point into “inside”, “outside” and “boundary”;

4) Plot the point and polygon in a plot window.

This report would present what I have done the work in each task. It can be used a guideline as well as a work journal

Software

1. Prerequisites of the software

1) Operating system: Windows 11 with 16GB RAM

2) Anaconda 3

3) Pycharm 2022.2.2

4) Python 3.8

5) Libraries: Numpy 1.23.4, Matplotlib 3.3.3

Task Description

Task 1. The MBR Algorithm

```
min_x = min(polygon_points.x)
max_x = max(polygon_points.x)
min_y = min(polygon_points.y)
max_y = max(polygon_points.y)
for q in range(len(testing_points.x)):
    # check if testing_point is inside MBR
    if min_x <= testing_points.x[q] <= max_x and min_y <= testing_points.y[q] <= max_y:
        testing_points.position[q] = 'inside'
    else:
        testing_points.position[q] = 'outside'
```

1.Interpretations of variables:

min_x: the min x value of MBR (left boundary)

max_x: the max v value of (right boundary)

min_y: the min y value of MBR (bottom boundary)

max_y: the max y value of MBR (upper boundary)

2.How it works:

1) Find the boundary of the MBR of the polygon

2) Check the position of each testing point

Task 2. The RCA Algorithm

1.I intended to draw a line from the testing point to “infinity” in the horizontal direction as the ray (i.e.,the ray is parallel to the x-axis). As assuming that the other point of the ray is at infinity, I didn’t define such a point with simply using the y-value of the testing point to calculate afterwards.

2.I firstly checked if the ray is parallel but not coincide with the line. If that is the case, there will be no intersecting point, which means that counting number would be zero.

```
count = 0

# the ray is parallel with the line but not coincide with it
if polygon_points.y[m] == polygon_points.y[m+1] and testing_points.y[k] != polygon_points.y[m]:
    count = count
```

3.Then I focused on the cases that the ray and the line are neither parallel nor coincided (coincided cases will be discussed in the special case part). I defined the boundaries of MBR (line) at the beginning for later intersecting-point-checking, by comparing two y-values of vertices of the line:

```
# find MBR(line) for later checking the intersecting point
line_y_max = max(polygon_points.y[m], polygon_points.y[m+1])
line_y_min = min(polygon_points.y[m], polygon_points.y[m+1])
```

4.I considered the above cases as two situations:

1) The line is vertical

```
if polygon_points.x[m] == polygon_points.x[m+1]:
  # The intersecting point is on the right of the testing point (counted)
    if polygon_points.x[m] > testing_points.x[k]:
        intersecting_point_x = polygon_points.x[m]
        intersecting_point_y = testing_points.y[k]
        if testing_points.x[k] < intersecting_point_x <= max_x and \
                line_y_min < intersecting_point_y < line_y_max:
            count = count + 1
        else:
            count = count
    # the intersecting point is on the left of the testing point (not counted)
    else:
        count = count
```

2) The line is neither vertical nor horizontal (the horizontal situation has been considered before)

```
# calculate the y-intercept and slope of the line
b = polygon_points.y[m] - (((polygon_points.y[m+1] - polygon_points.y[m]) /
                            (polygon_points.x[m+1] - polygon_points.x[m])) *
                           polygon_points.x[m])
s = (polygon_points.y[m+1] - polygon_points.y[m]) / \
    (polygon_points.x[m+1] - polygon_points.x[m])
if s != 0:
    # calculate coordinates of the intersecting point
    intersecting_point_x = (testing_points.y[k] - b) / s
    intersecting_point_y = testing_points.y[k]
    # check the intersecting point is inside MBR(polygon), MBR(ray) and MBR(line)
    if testing_points.x[k] <= intersecting_point_x <= max_x and \
```

```
            line_y_min < intersecting_point_y < line_y_max:
        count = count + 1
    else:
        count = count
```

5.If the total count number is even, the testing point is outside the polygon. Otherwise (i.e.,the total count number is odd), the testing point is inside the polygon.

However, some special cases are needed to be considered as their counting-process is different, which would influence the total count number. I will show the codes of determining whether the total count number is even or odd at the end of the next task.

Task 3. The Categorisation of Special Cases

There are three types of special cases mentioned in the teaching slides: boundary points, points crossing vertices, and rays coincide with line. While I mainly considered two types of special cases as I integrated the third case into the second one, the details are as follows.

1.Boundary points

1) To determine whether the testing point is on the polygon-line, other than calculating the point-on-a-line equation in the teaching slides, it is also necessary to check if the testing point is on the segment defined by two points, which means that the testing point is inside the MBR (line).

2) A check_mbr() function is created to simplify the checking MBR(line) process.

```
# create a function that checks the intersecting point is inside MBR(line) and MBR(ray)
def check_mbr(testing_point_x, testing_point_y, line_x_range, line_y_range):
    mbr_x_min = min(line_x_range)
    mbr_x_max = max(line_x_range)
    mbr_y_min = min(line_y_range)
    mbr_y_max = max(line_y_range)
    if mbr_x_min <= testing_point_x <= mbr_x_max and mbr_y_min <= testing_point_y <=
mbr_y_max:
        return True
```

3) Checking boundary:

```
# use a "for loop" to check the testing points one by one
for p in range(len(testing_points.x)):
    # check testing_point is inside MBR(polygon)
    if min_x <= testing_points.x[p] <= max_x and min_y <= testing_points.y[p] <= max_y:
        # special case 1--boundary points--Point on a Line Algorithm
```

```
        for n in range(len(polygon_points.x) - 1):  # start from the first line
            # the line is vertical or horizontal(special cases)
            if testing_points.x[p] == polygon_points.x[n] == polygon_points.x[n+1] or \
                    testing_points.y[p] == polygon_points.y[n] == polygon_points.y[n+1]:
                # check the testing point is on the segment defined by two points -- MBR(line)
                x_range = [polygon_points.x[n], polygon_points.x[n+1]]  # parameter of "the
check_mbr()" function
                y_range = [polygon_points.y[n], polygon_points.y[n+1]]  # parameter of "the
check_mbr()" function
                if check_mbr(testing_points.x[p], testing_points.y[p], x_range, y_range):
                    testing_points.position[p] = 'boundary'
            # the line is neither vertical nor horizontal(normal cases)
            # condition 1 and condition 2 check the line is neither vertical nor horizontal
            # condition 3 checks y-values of the testing point and the intersecting point are equal
            elif polygon_points.x[n] != polygon_points.x[n+1] and polygon_points.y[n] !=
polygon_points.y[n+1] and \
                    testing_points.y[p] == (((testing_points.x[p] - polygon_points.x[n]) *
                                             (polygon_points.y[n+1] - polygon_points.y[n])) /
                                            (polygon_points.x[n+1] - polygon_points.x[n])) +
polygon_points.y[n]:
                # check the testing point is on the segment defined by two points -- MBR(line)
                x_range = [polygon_points.x[n], polygon_points.x[n+1]]
                y_range = [polygon_points.y[n], polygon_points.y[n+1]]
                if check_mbr(testing_points.x[p], testing_points.y[p], x_range, y_range):
                    testing_points.position[p] = 'boundary'
        else:
            testing_points.position[p] = 'outside'
```

2.Rays crossing vertice and rays coincide with lines

I put a premise for this special case: the ray intersects with the first vertex of the line. If that is not the case, it would be considered as the normal cases for RCA (task 2).

```
# special case 2 -- rays crossing vertices and rays coincide with lines
# condition 1: check the ray intersecting with the first vertex of the line (clockwise)
```

```
# condition 2: check the intersecting point is on the right of the testing point
elif testing_points.y[k] == polygon_points.y[m] and polygon_points.x[m] > testing_points.x[k]:
```

After checking the premise, I need to think about current and previous lines at the same time.

I firstly split this special case into two situations:

1) the current line coincides with the ray

2) the current line doesn't coincide with the ray. For each situation, I then checked if the previous line coincides with the ray. Therefore, until this point, special case two can be generally split into four scenarios:

Current line	**Previous line**
Situation 1: coincides with the ray	Scenario 1: Coincides with the ray
	Scenario 2: Doesn't coincides with the ray
Situation 2: doesn't coincides with the ray	Scenario 1: Coincides with the ray
	Scenario 2: Doesn't coincides with the ray

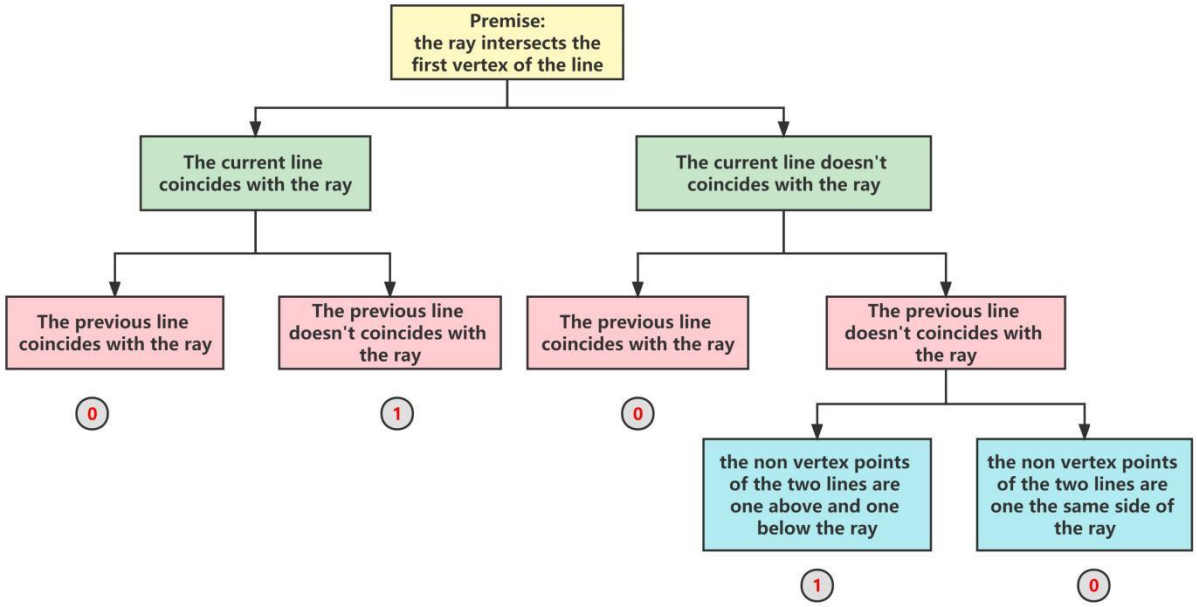
For Situation 2 - Scenario 2, two more cases need to be considered:

1) the non-vertex points of the two lines are one above and one below the ray

2) the non-vertex points of the two lines are on the same side of the ray.

A table and a workflow containing the count number are shown as follows to make it easier to understand:

Current line	**Previous line**	**Counting number**
Situation 1 : coincides with the ray	Scenario 1: Coincides with the ray	0
	Scenario 2: Doesn't coincides with the ray	1
Situation 2 : doesn't coincides with the ray	Scenario 1: Coincides with the ray	0
	Scenario 2: Doesn't coincides with the ray	1) 1 2) 0



Situation 1:

```
elif testing_points.y[k] == polygon_points.y[m] and polygon_points.x[m] > testing_points.x[k]:
    # think about current and the previous lines:
    # special case 2 -- situation 1 -- current line coincides with the ray
    if polygon_points.y[m] == polygon_points.y[m+1]:
        # special case 2 -- situation 1 -- scenario 1 -- previous line is vertical
        if polygon_points.x[m] == polygon_points.x[m-1]:
            count = count
        # special case 2 -- situation 1 -- scenario 2
        # previous line is oblique, for one special testing point: (0,0)
        elif polygon_points.y[m] < polygon_points.y[m-1]:
            count = count
        # special case 2 -- situation 1 -- scenario 2 -- previous line is oblique (normal case)
        else:
            count = count + 1
```

Situation 2:

```
# special case 2 -- situation 2 -- current line doesn't coincide with the ray
else:
    # special case 2 -- situation 2 -- scenario 1 -- previous line coincides with the ray
    if polygon_points.y[m] == polygon_points.y[m-1]:
```

```
            count = count
        # special case 2 -- situation 2 -- scenario 2 -- previous line doesn't coincide with the ray
        else:
            # scenario 2.1 --the non vertex points of the two lines are one above and one below the
ray
            if (polygon_points.y[m-1] - polygon_points.y[m]) * \
                    (polygon_points.y[m+1] - polygon_points.y[m]) < 0:
                count = count + 1
            # scenario 2.2 -- the non vertex points of the two lines are on the same side of the ray
            else:
                count = count
```

Since I have considered all the cases, which include the normal cases in task 2 and special cases in task 3, now I can check whether the total count number is even (outside the polygon) or odd (inside the polygon).

```
if count % 2 == 0:  # counting number is even
    testing_points.position[k] = 'outside'
else:  # counting number is an odd
    testing_points.position[k] = 'inside'
```

Task 5. Object-Oriented Programming

1.I created a Point class, whose attributes include point id, x-value, y-value and point position.

```
class Point:
    def __init__(self, i_d, x, y, position):
        self.i_d = i_d
        self.x = x
        self.y = y
        self.position = position
```

2.Then I created two objects of the Point class--the polygon points and the testing points, using a create_points() function that takes a CSV file and creates a list of points.

```
# create a polygon
polygon_points = create_points('polygon.csv')
```

```
# create a list of testing points
testing_points = create_points('input.csv')
```

Task 6. Plotting

I used Plotter class and three additional plotting functionalities for plotting:

1) plt.xlabel(): plot x label

2) plt.ylabel : plot y labe

3) plt.hlines(): plot rays

```
# plot x and y label
plt.xlabel('x')
plt.ylabel('y')
```

```
# plot rays
plt.hlines(y=testing_points.y[k], xmin=testing_points.x[k], xmax=6, color='r', linestyle='--',
lw=0.5)
```

```
# plot polygon points
plotter.add_polygon(polygon_points.x, polygon_points.y)
# plot testing points
for r in range(len(testing_points.x)):
    plotter.add_point(testing_points.x[r], testing_points.y[r], testing_points.position[r])
# show the graph
plotter.show()
```

Task 7. Error Handling

In the mian_from_user.py, the program asks the user to input a pair of coordinates. While the user may input a value with errors such as extra blank paces or other characters. To handle the invalid character error, I created two initial variables (start_x, start_y) to start the loop and used try-except statements to identify input errors.

```
while True:
    start_x = True
    start_y = True
```

```
while start_x:
    try:  # valid x value
        x = float(input('x coordinate: '))
        while start_y:
            try:  # valid y value
                y = float(input('y coordinate: '))
                ...(Point-in-Polygon Algorithm)...
                # stop the loop and start over from the top
                start_y = False
                start_x = False
            except ValueError:  # invalid x value
                print('Please delete irrelevant characters and input a valid y value')
    except ValueError:  # invalid x value
        print('Please delete irrelevant characters and input a valid x value')
```

Reference

The “Plotter” class, the input polygon and testing points are provided by Dr. Aldo Lipani at UCL for the taught master module Geospatial Programming.