西北工業大學

Assignment Report

Score:

Student ID: Radiv Sarwar

Student Name: 2019380138

Subject: Computational Geometry

Submitted Date: 2024.1.11

Northwestern Polytechnical University

Assignment

Computational Geometry Algorithms and Applications, 2023

Distributed on Thursday, December 7.

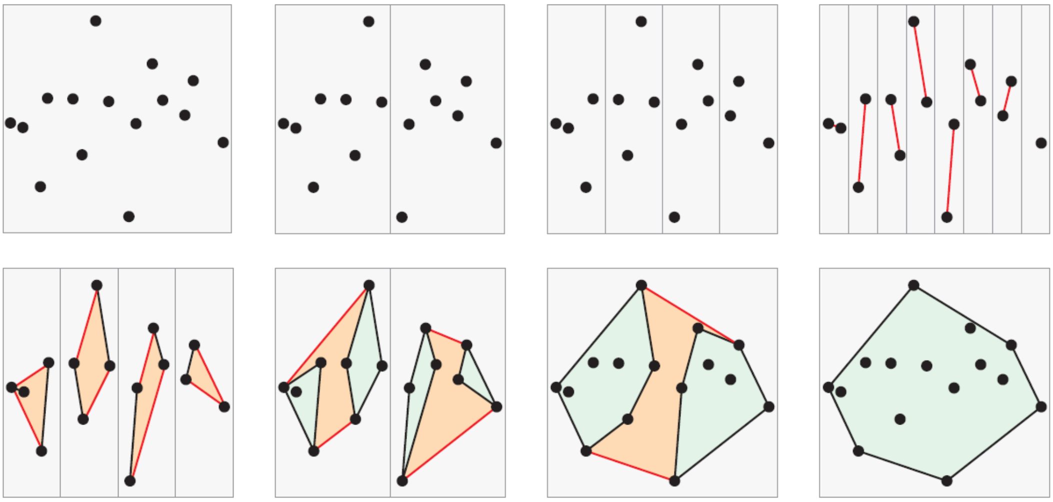
Submit the digital copy (Word or PDF with this template) of your solutions (with your name and ID!) to yujf@nwpu.edu.cn as an attachment of email, NO LATER THAN Thursday, January 11, 2024.

Programming results need to be explained with "running instructions", which means the corresponding steps to run the interface (with screenshots) are described. If the program requires certain system configuration and auxiliary program installation, it should also be explained in the specific configuration steps. The running instructions should be printed in the answer booklet together with the first question as the text of the second question, converted into pdf format and named uniformly: Student-ID_2023CGAssignment.pdf.

Programming submissions should consist of “source code”, “executable program that can be run” and should be submitted in a single package using a consistent naming format: Student-ID_2023CG_Program.zip.

A-1: Divide-and-conquer algorithm for convex hull computation

Whereas the incremental algorithm uses induction, the divide-and-conquer algorithm uses the technique of recursion, a powerful algorithm paradigm. The divide-and-conquer paradigm partitions the problem into two parts, solves each of the parts recursively, and then "merges" the two solutions to obtain the full solution.



(a.) Let P_1 and P_2 be two disjoint convex polygons with n vertices in total. Give an $O(n)$ time algorithm that computes the convex hull of $P_1 \cup P_2$.

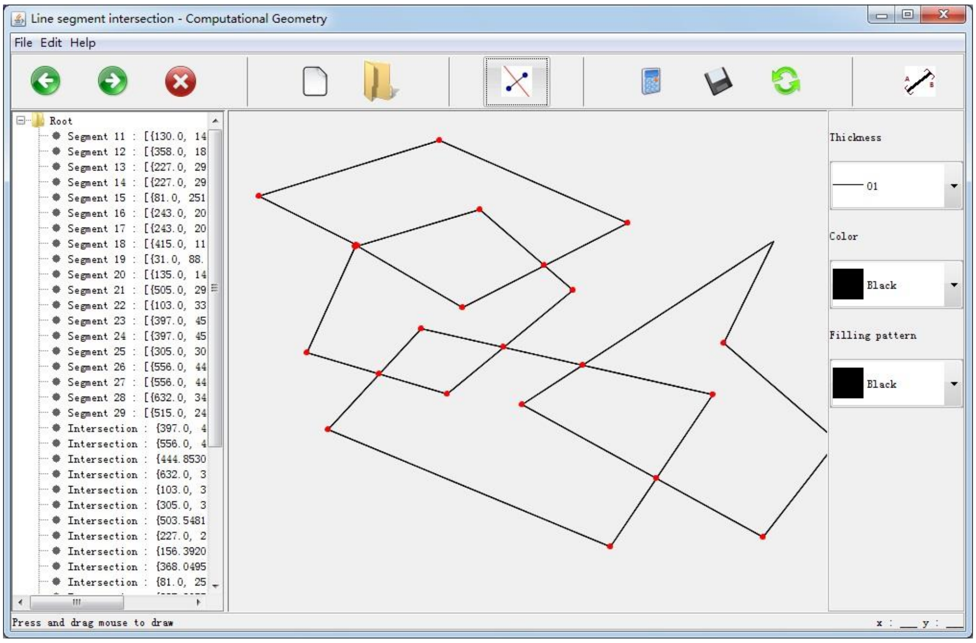
(b.) Use the algorithm from part a to develop an $O(n\log n)$ time divide-and-conquer algorithm to compute the convex hull of a set of n points in the plane.

A-2: Programming

Using C/C++, C#, Java, Java Script, Python and other programming languages (as you like) to implement the corresponding algorithm. The user interface of the implementation could be designed in the way shown in the samples.

A-2-1: Computing the overlap of planar subdivisions:

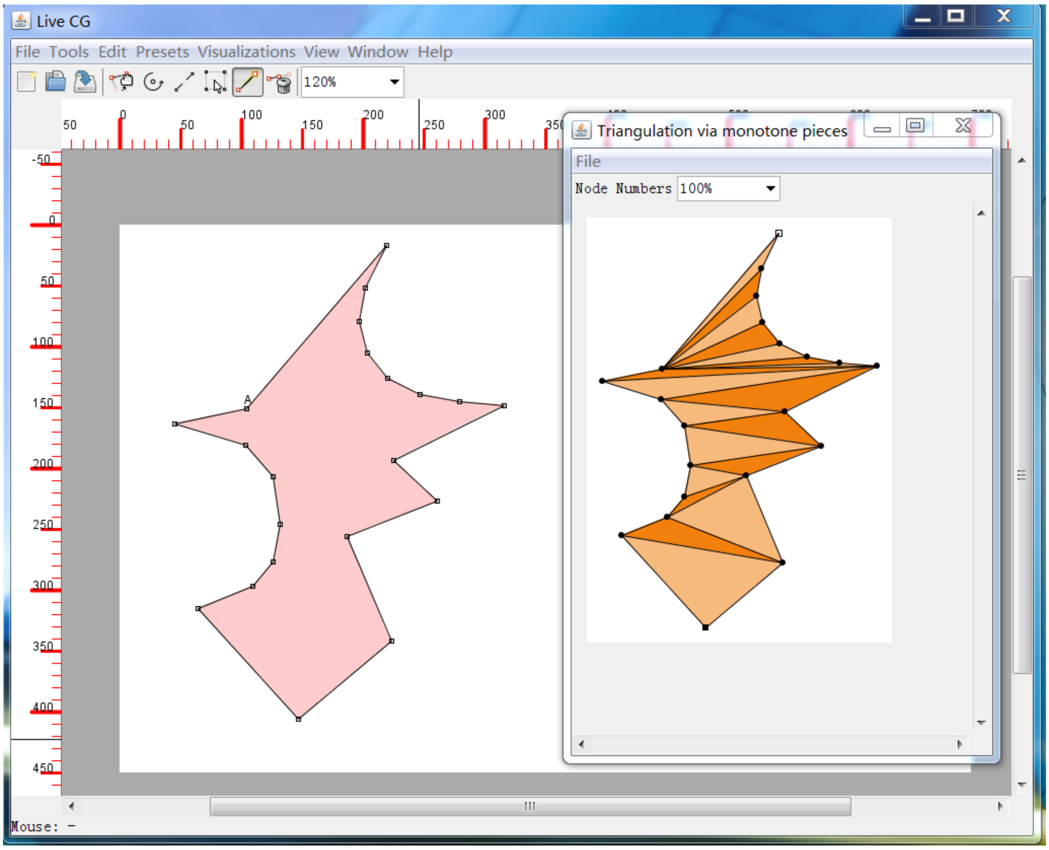
Input:	Vertices are defined by the mouse click interactively, or by importing an external data files. These subdivisions are stored in DCEL.	
Output:	The overlap of these subdivisions, also stored in DCEL.	
Requirements:	To show the dynamic process of updating the plan sweep	
Group:	2020300046 马岳言	2020301426 吴俊杰
	2019380138 RADIV SARWAR	2021380069 SIEWE SIEMENI BLUCHER



A-2-2: Triangulation of polygons:

Input:	Monotonic polygon P, whose vertices are defined by the mouse click interactively, or by importing an external data files. The monotony of the polygon should be tested as the vertex is input, non-monotonic error should be prompted.
Output:	Adjacent triangles after triangulation should be filled by different colors and clearly identified.

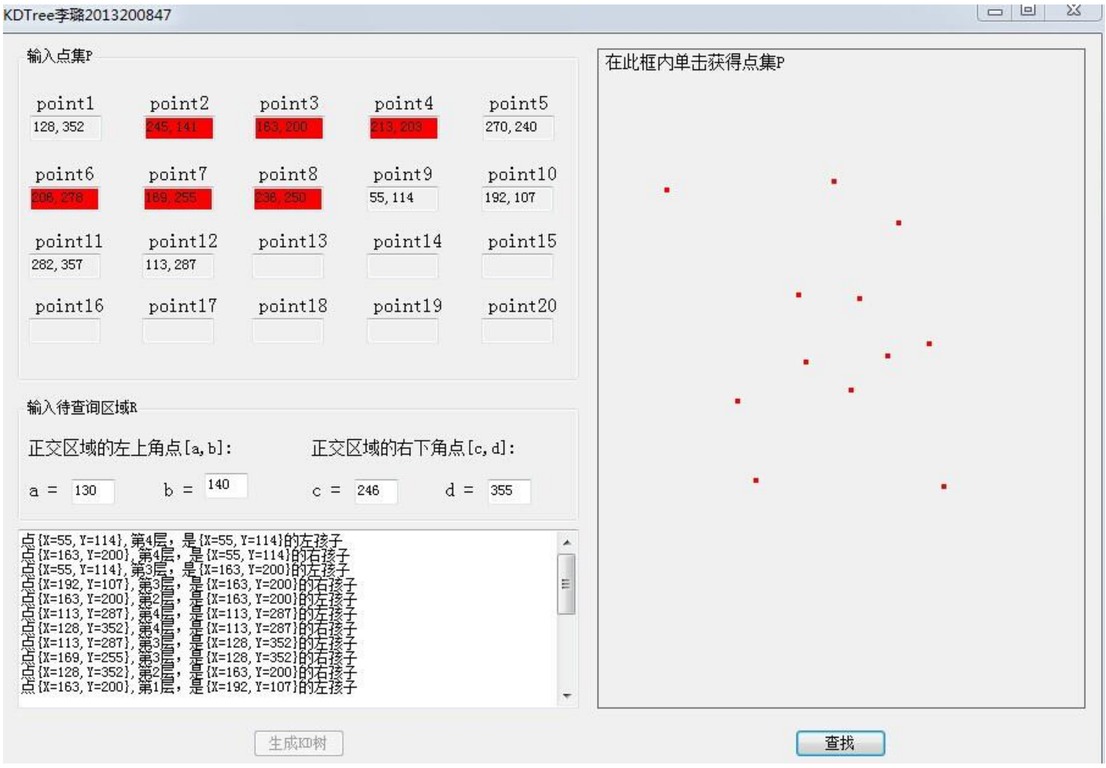
Requirements:	To show the dynamic process of updating the plan sweep, the greedy manipulation as well.	
Group:	2020301464 赵语婕	2020301513 王彬全
	2020301518 靳铠豪	2020301632 李成林



A-2-3: KD tree construction and query:

Input:	N points on the plane, defined by the mouse click interactively, or by importing an external data files. Query area R, defined by the mouse click interactively, or by importing an external data files.
Output:	The traverse of the *k*d tree, and the points contained in the query area are reported.

Requirements:	Points in the query area are highlighted	
Group:	2021301477 高铜	2021301628 张健
	2021301639 刘家伊	2021301664 高羽凡



Ans to the question no. 1

Ans a: To find the convex hull of the union of two disjoint convex polygons P1 and P2, which together have a total of n vertices, we can use the Graham's scan algorithm to accomplish this task in O(n) time. Here is the algorithm:

```
function calculate_convex_hull(P1, P2):
    n1 = length(P1)
    n2 = length(P2)
    n = n1 + n2

    Union = concatenate(P1, P2)

    bottom_vertex = find_bottom_vertex(Union)

    sort(Union, by the polar angle with respect to bottom_vertex)

    stack = empty stack

    push(Union[0], stack)
    push(Union[1], stack)
    push(Union[2], stack)

    for i = 3 to n-1:
        while size(stack) >= 2 and is_nonleft_turn(second_top(stack), top(stack), Union[i]):
            pop(stack)
        push(Union[i], stack)

    convex_hull = empty list

    while stack is not empty:
        vertex = pop(stack)
```

```
        convex_hull.append(vertex)

    return convex_hull
```

Ans b: One approach to constructing a divide-and-conquer algorithm with an O(nlogn) time complexity for computing the convex hull of a set of n points in the plane is by leveraging the principles of Graham's scan algorithm. By combining these two techniques, we can efficiently solve the problem. The following algorithm attempts to do just that:

```
function calculate_convex_hull(points):
    n = length(points)

    if n <= 3:
        return points

    mid = n // 2

    left_points = points[0:mid]
    right_points = points[mid:n]

    left_hull = compute_convex_hull(left_points)
    right_hull = compute_convex_hull(right_points)

    upper_tangent = find_upper_tangent(left_hull, right_hull)
    lower_tangent = find_lower_tangent(left_hull, right_hull)

    convex_hull = merge_hulls(left_hull, right_hull, upper_tangent, lower_tangent)

    return convex_hull
```

Ans to the question no. 2

Ans: I will be beginning with the prerequisites for running the program. For this program to run from the source code one will need to install the following python packages:

1) Matplotlib:

This will be the main package we will be using to plot the points in the graph.

2) Mplcursors:

This is a package that helps us interact with the graph. This allows us to place points on the graph by just clicking.

3) Shapely:

We will be using this to perform operations on geometrical entities.

These packages can be installed through pip package manager. An example of the installation can be seen here:

```
pip install matplotlib
```

We will now be going into the program that will measure the overlap of the subdivisions by taking in input points on a scatter graph.

```python
import matplotlib.pyplot as plt
import mplcursors
from matplotlib.widgets import Button
from shapely import Polygon

```

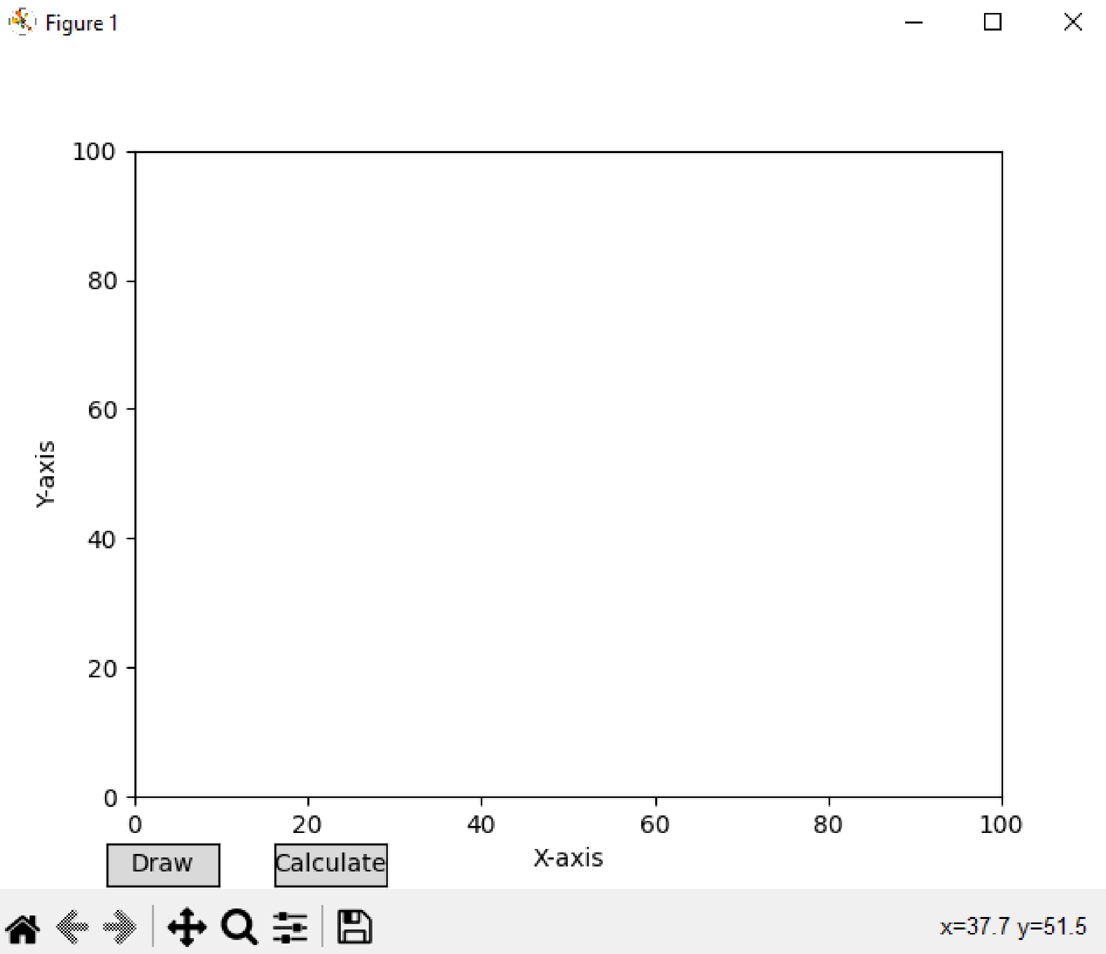
We start by importing all the packages that we will be using. We choose the alias for matplotlib as plt as per convention and import the Button feature from matplotlib.widgets library. We also import Polygon from the shapely library.

```python
# Creating a scatter plot
fig, ax = plt.subplots()
```

```
scatter = ax.scatter([], [])
ax.set_xlim(left = 0, right = 100)
ax.set_ylim(bottom = 0, top = 100)
```

We then create the scatter plot that will serve as our interactive window.

This will create the following cartesian coordinate graph to contain all our work:



Notice that the x and y axis are limited to 100 units each. There are two buttons that are implemented in this graph that we will discuss in the next sections. Before that we will define all the variables that we will be needing for the calculation for the overlapped subdivisions.

The method of input chosen for this program is by clicking interactively on the graph that is shown above.

```
#creating polygon variables
poly= []
all_polygons= []
all_intersections= []

```

We create the variable we will be needing for the calculation throughout the program. These are of the list data structure in python.

Next we create the buttons:

```
#creating the buttons
button1_ax = plt.axes([0.1, 0.005, 0.1, 0.05])
done = Button(button1_ax,"Draw")
button2_ax = plt.axes([0.25,0.005, 0.1, 0.05])
calculate = Button(button2_ax, "Calculate")
```

The button’s dimensions and positions are given in the bracket as (x,y,width,height). The first button is called done and named “Draw”. This button’s function is to stop taking mouse clicks as inputs and complete the polygon with all the input points. When clicked it calls on the function done_clicked(). This function is defined as:

```

#defining what happens when we click the done button
def done_clicked(event):
    poly.pop()
    poly_drawn = Polygon(poly)
    x,y = poly_drawn.exterior.xy
    ax.plot(x,y)
    plt.draw()
    all_polygons.append(poly.copy())
    poly.clear()

```

The next button is called calculate. When clicked it calls on the function calculate_clicked(). It is defined as:

```python
def calculate_clicked(event):
    # we stop drawing polygons
    fig.canvas.mpl_disconnect(start_graph)
    # we make all the lists in all polygons into shapely polygons
    shapely_polys= []
    for p in all_polygons:
        shapely_polys.append(Polygon(p))

    #we now need to produce all intersection polys and put them in intersection_polys[]
    shapely_polys_list= list(shapely_polys)

    intersections = []
    for i in range(len(shapely_polys)):
        for j in range(i + 1, len(shapely_polys)):
            intersection = shapely_polys[i].intersection(shapely_polys[j])
            if not intersection.is_empty:
                intersections.append(intersection)


    for i in intersections:
        inter_draw = i.exterior.coords
        patch = plt.Polygon(inter_draw, alpha=0.8)
        ax.add_patch(patch)
        plt.draw()

    print(intersections)
    # Making the output in DCEL format in a txt file
    # Open the file in write mode
    filename = "output_in_DCEL.txt"
    with open(filename, "w") as file:
        vertex_id = 0
        for polygon_id, intersections in enumerate(intersections):
            # Write the vertices
            vertices = intersections.exterior.coords
            for i, (x, y) in enumerate(vertices):
                file.write(f"v {vertex_id} {round(x,2)} {round(y,2)}\n")
                vertex_id += 1

            # Write the half-edges
            num_edges = len(vertices)
            for i in range(num_edges):
                file.write(f"e {polygon_id*num_edges + i} {polygon_id*num_edges + i} {polygon_id*num_edges + (i+1) % num_edges}\n")

            # Write the face
            file.write(f"f {polygon_id*num_edges}\n")
```

This is a complicated function that we will get back to when I explain the workflow of the program in the following sections.

For now, we have a scatter plot that needs to take in inputs as mouse clicks. To update the plot we will define the following function:

```python
# Function to update the scatter plot with a new point
def update_plot(x, y):
    scatter.set_offsets([[x, y]])
    poly.append((x,y))

    plt.draw()
```

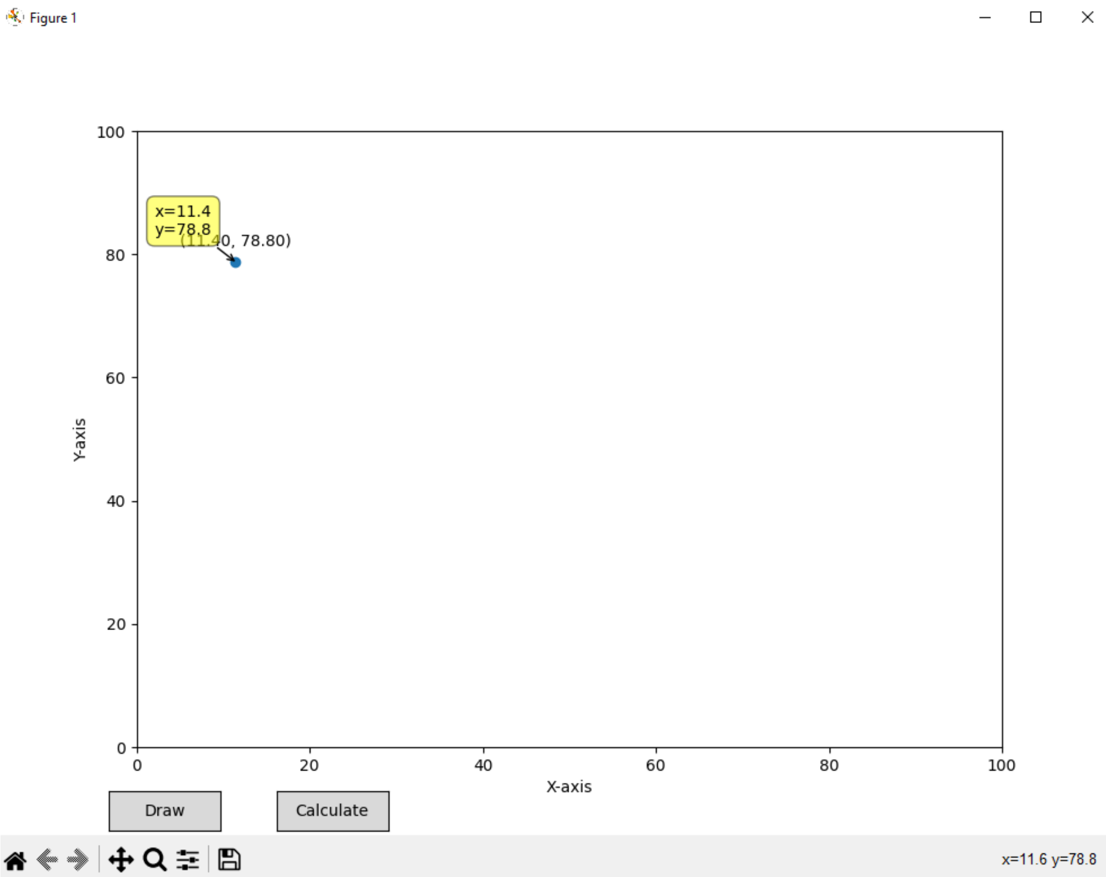
This function draws the points when we click on the canvas. The function that handles the clicking event is done by the following function:

```python
# Function to handle mouse click events
def on_click(event):
    if event.inaxes:
        x, y = round(event.xdata,1), round(event.ydata,1)
        update_plot(x, y)
        annotate_text = f'({x:.2f}, {y:.2f})'
        ax.annotate(annotate_text, (x, y), textcoords="offset points", xytext=(0,10), ha='center')
        plt.draw()
```

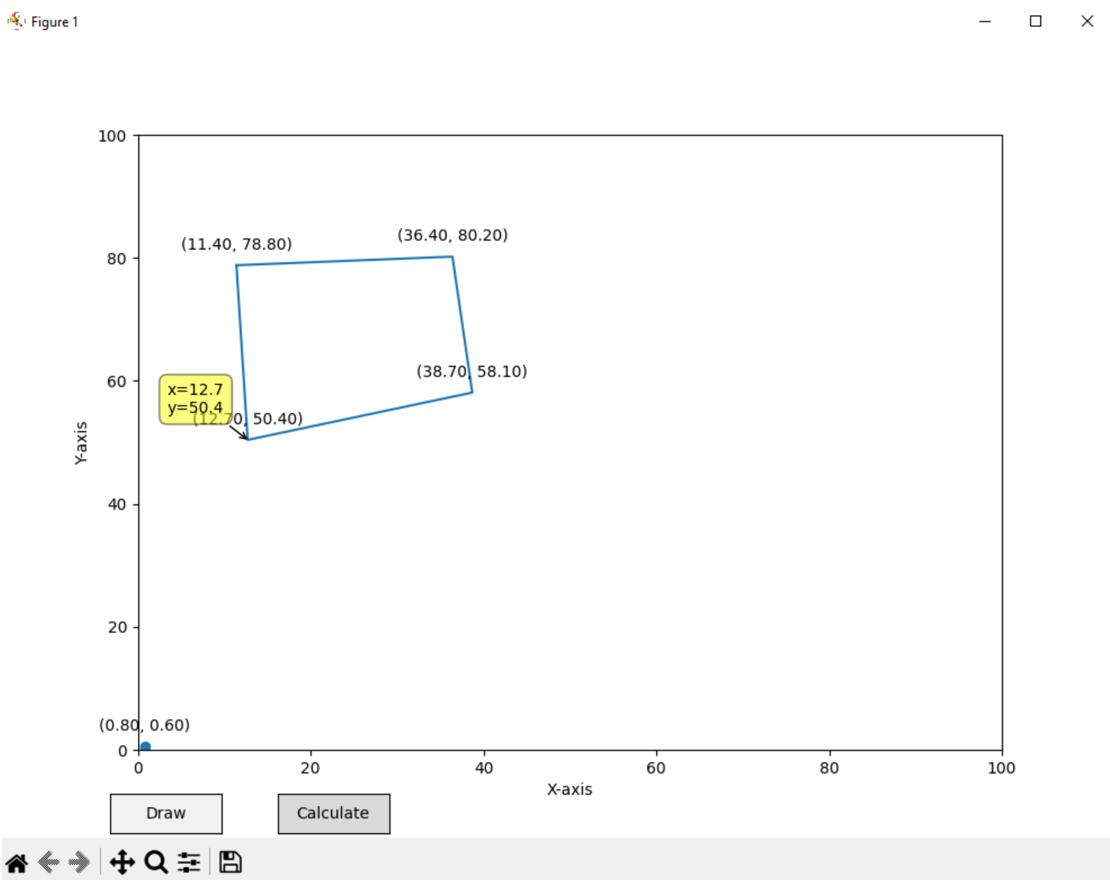
Finally, we have the main part of the program where we use all the functions to work out the required solution and create an output file called “output_in_DCEL.txt” that carries the answers in DCEL format.

Flow of the program:

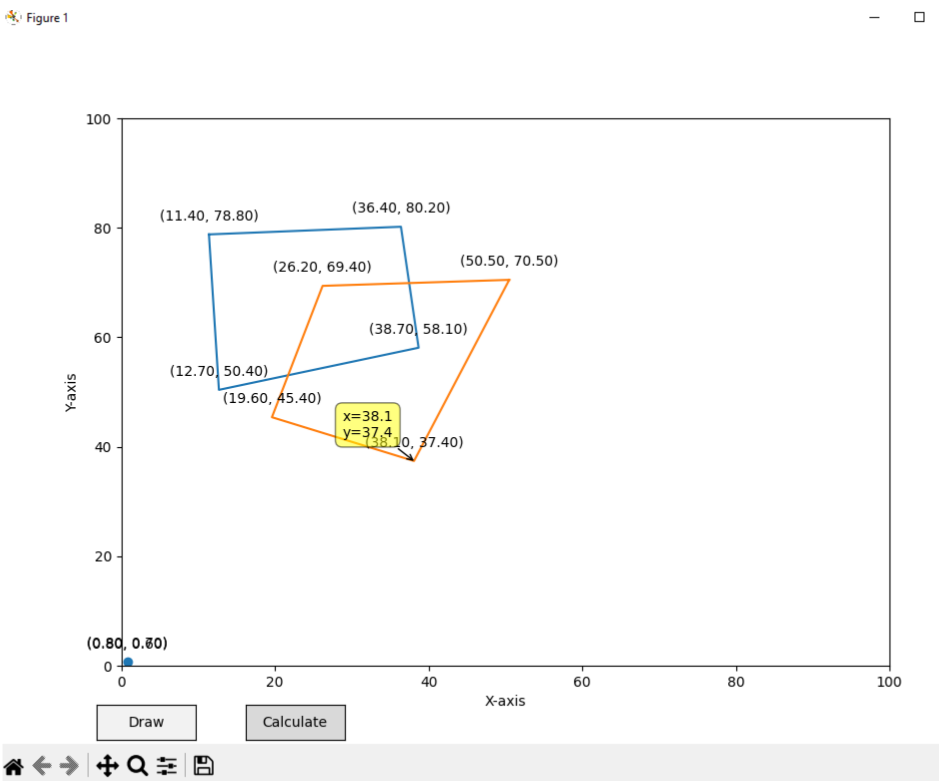
When we run the program we will receive an interactive matplotlib scatter plot on the screen as shown before. We can then start providing inputs by clicking on the canvas. If we click on the scatter plot the program will plot the corresponding point on the plot like this:



We then place more points around to make a polygon.
When we are done placing all the desired points on the plot we click on the draw button. This will join all the points and create our desired polygon as such:

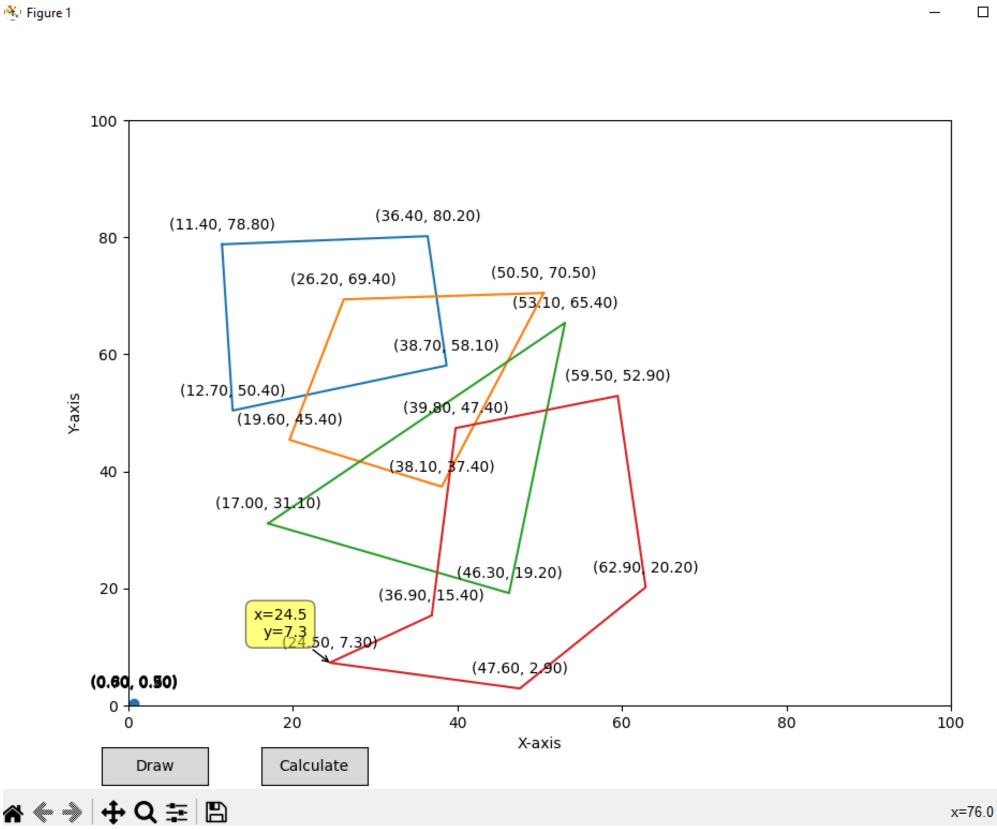


Afterwards we will be able to add more points for the second polygon on the graph. Clicking the draw button will also then complete the second polygon as such:



We can keep adding more and more polygons. After inputting the points by clicking all we have to do is press the draw button.

We will then have a scatter plot where we will have something that looks like this:




```
# Function to handle mouse click events
def on_click(event):
    if event.inaxes:
        x, y = round(event.xdata,1), round(event.ydata,1)
        update_plot(x, y)
        annotate_text = f'({x:.2f}, {y:.2f})'
        ax.annotate(annotate_text, (x, y), textcoords="offset points", xytext=(0,10), ha='center')
        plt.draw()
```

Every click is being registered through this function. When the event (in this case a mouse click) occurs the xdata and ydata from the event is taken and plotted on the graph. This gives us the inputs. Updating the plot is done by this function:

```
# Function to update the scatter plot with a new point
def update_plot(x, y):
    scatter.set_offsets([[x, y]])
    poly.append((x,y))

    plt.draw()
```

When every new point is being added, the poly list is being updated with the new input.

When the “Draw” button is pressed the following code is called:

```
#defining what happens when we click the done button
def done_clicked(event):
    poly.pop()
    poly_drawn = Polygon(poly)
    x,y = poly_drawn.exterior.xy
    ax.plot(x,y)
    plt.draw()
    all_polygons.append(poly.copy())
    poly.clear()

```

The latest faulty value from poly[] is deleted and the list is turned into a shapely Polygon data type. We then take the x,y values from the polygon and plot it on the graph. Thus creating the polygon from the input points on the graph. Lastly we add a copy of the list poly to the all_polygons list that collects all the values of the polygons that are being input.

When we are happy with the input we will press the calculate button. This will call on the function calculate_clicked().

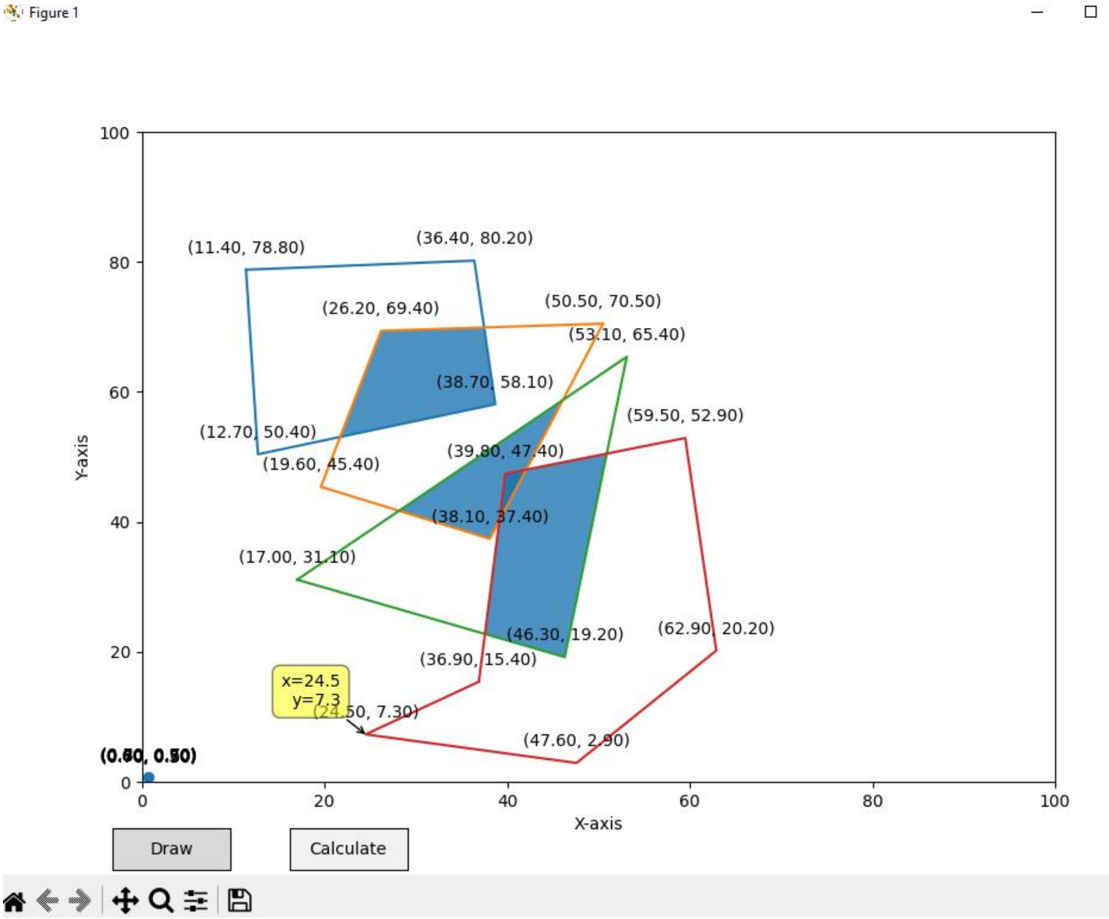
```
def calculate_clicked(event):
    # we stop drawing polygons
    fig.canvas.mpl_disconnect(start_graph)
    # we make all the lists in all polygons into shapely polygons
    shapely_polys= []
    for p in all_polygons:
        shapely_polys.append(Polygon(p))
```

The first line of the function ceases the function of the input on the graph. After this point the program will not take any more inputs. We create a list called shapely_polys[]. We then convert all the collected polygons in all_polygons into Polygon object from shapely and store them in this list.

```
#we now need to produce all intersection polys and put them in intersection_polys[]
intersections = []
for i in range(len(shapely_polys)):
    for j in range(i + 1, len(shapely_polys)):
        intersection = shapely_polys[i].intersection(shapely_polys[j])
        if not intersection.is_empty:
            intersections.append(intersection)


for i in intersections:
    inter_draw = i.exterior.coords
    patch = plt.Polygon(inter_draw, alpha=0.8)
    ax.add_patch(patch)
    plt.draw()
```

With the next section we will calculate the overlap of the subdivisions through the intersect function. We first define a list called intersections[].
The next nested for loop calculates all the possible intersections between the input polygons and stores them in the list intersection[].

With the next for loop we draw all the intersections between the polygons and create a patch to signify the overlaps on the graph. After clicking the calculate button the intersections will be highlighted as such:



The next section will take the intersections[] and output them to a .txt file called “output_in_DCEL.txt”:

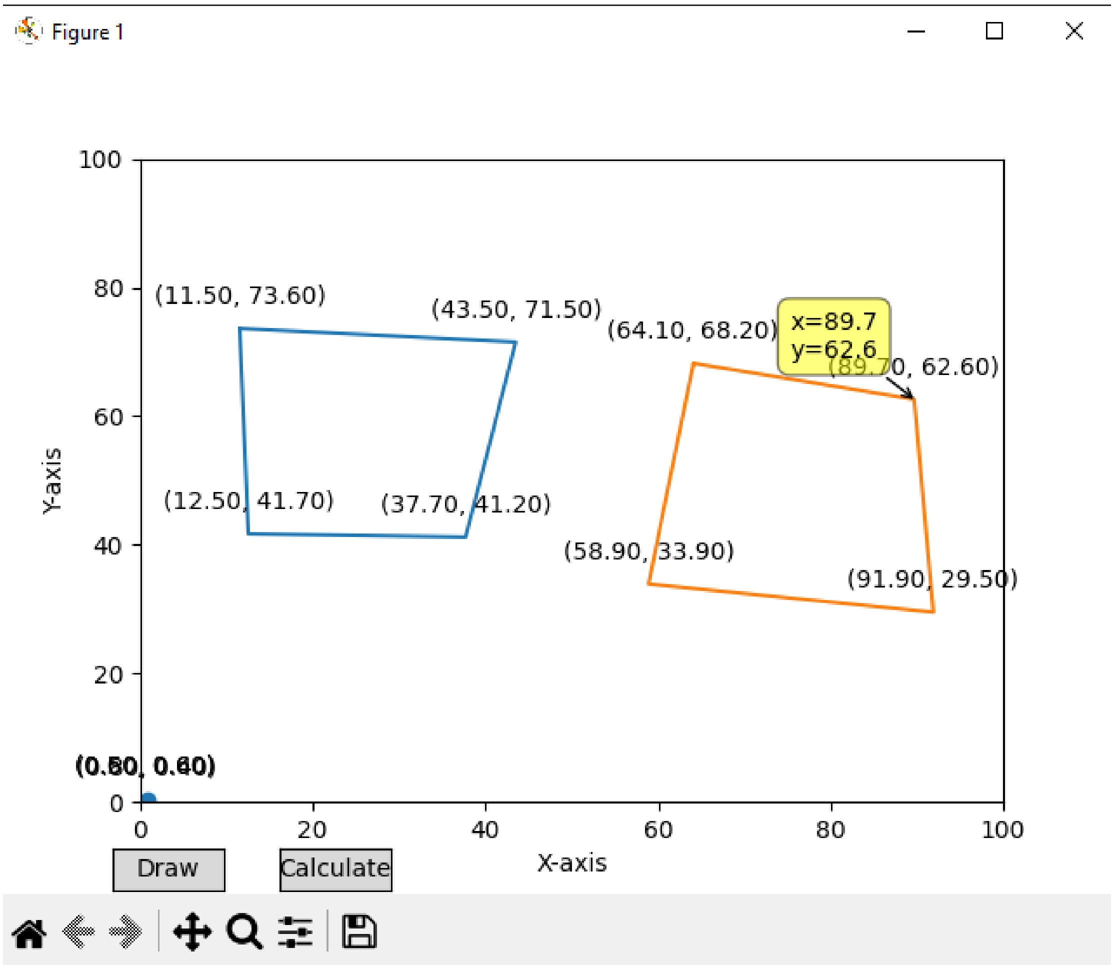
```python

    # Making the output in DCEL format in a txt file
    # Open the file in write mode
    filename = "output_in_DCEL.txt"
    with open(filename, "w") as file:
        vertex_id = 0
        for polygon_id, intersections in enumerate(intersections):
            # Write the vertices
            vertices = intersections.exterior.coords
            for i, (x, y) in enumerate(vertices):
                file.write(f"v {vertex_id} {round(x,2)} {round(y,2)}\n")
                vertex_id += 1

            # Write the half-edges
            num_edges = len(vertices)
            for i in range(num_edges):
                file.write(f"e {polygon_id*num_edges + i} {polygon_id*num_edges + i} {polygon_id*num_edges + (i+1) % num_edges}\n")

            # Write the face
            file.write(f"f {polygon_id*num_edges}\n")

```

The values will be rounded and this is the output file that we will get in the same folder:

output_in_DCEL.txt

```
v 0 38.7 58.1
v 1 21.71 53.07
v 2 26.2 69.4
v 3 37.47 69.91
v 4 38.7 58.1
e 0 0 1
e 1 1 2
e 2 2 3
e 3 3 4
e 4 4 0
f 0
v 5 38.1 37.4
v 6 28.16 41.7
v 7 46.1 58.75
v 8 38.1 37.4
e 4 4 5
e 5 5 6
e 6 6 7
e 7 7 4
f 4
v 9 39.15 40.19
v 10 39.8 47.4
v 11 42.09 48.04
v 12 39.15 40.19
e 8 8 9
e 9 9 10
e 10 10 11
e 11 11 8
f 8
v 13 46.3 19.2
v 14 37.57 22.75
v 15 39.8 47.4
v 16 50.91 50.5
v 17 46.3 19.2
e 15 15 16
e 16 16 17
e 17 17 18
e 18 18 19
e 19 19 15
f 15
```

For cases where there will be not overlap between the subdivisions like:



We will receive an empty DCEL file: