ME145 Robotic Planning and Kinematic: Lab 4

BFS algorithm + Sweeping trap algorithm

Elijah Perez

Winter 2025, Jan 21

Pseudocode form notes:

breadth-first search (BFS) algorithm

Input: a graph G, a start node v_{start} and goal node v_{goal}
Output: a path from v_{start} to v_{goal} if it exists, otherwise a failure notice

for each node v in G :
    `parent`(v) := `NONE`
`parent`(v_{start}) := `SELF`
create an empty queue Q and insert(Q, v_{start})
while Q is not empty :
    v := retrieve(Q)
    for each node u connected to v by an edge :
        if `parent`(u) == `NONE` :
            set `parent`(u) := v and insert(Q, u)
        if $u == v_{\text{goal}}$:
            run *extract-path* algorithm to compute the path from start to goal
            return *success* and the path from start to goal
return *failure* notice along with the `parent` values.

A Q array and parent array are initialized, with the Q array containing the starting node and the parent array filled with NaNs, matching the number of nodes in the workspace. A while loop runs as long as the Q array isn't empty, removing the first node each iteration and adding unvisited adjacent nodes. The parent array updates any NaN slots with the current node, marking edges as active. This process continues until all reachable adjacent edges are explored. Below is the code for the computeBFS tree.

extract-path algorithm

Input: a goal node v_{goal}, and the `parent` values
Output: a path from v_{start} to v_{goal}

create an array $P := [v_{\text{goal}}]$
set $u := v_{\text{goal}}$
while `parent`$(u) \neq$ `SELF` :
    u := `parent`(u)
    insert u at the beginning of P
return P

The "Extract-path" algorithm retrieves the path from the parent vector by traversing backward from the goal node. Once the goal is reached, a while loop initiates, filling the path array as it traces back to the start node. This is done by setting the current node to its parent, using the parent vector (where the column index corresponds to the node number), and adding each node to the path array until reaching the start. This logic is integrated into the previous code to form the computeBFSpath function.

algorithm implementation:

1) function -> (ComputeBFSTree)

```
function [Parent,G] = computeBFSTree(AdjTable,vStart)
%% Intilize

Parent = nan(1,length(AdjTable));
Parent(vStart) = 0;

Que = [];
Que(1) = vStart;

Node   = [];
Branch = [];

%% main looop
while ~isempty(Que)
    vCurrent = Que(1);
    Que(1) = [];
    AttatchedNodes = AdjTable{vCurrent};

    for i = 1:length(AttatchedNodes)
        visit = AttatchedNodes(i);


        if isnan(Parent(visit(1)))
            Parent(visit(1)) = vCurrent;
            Que(end+1) = visit(1);


            Node(end+1)   = vCurrent;
            Branch(end+1) = visit;
        end
    end
end

G = digraph(Node,Branch);
end
```

This function accepts an arbitrary adjacency table, a start node, and an end node as inputs. It outputs the parent vector and G, where G represents the directed graph (digraph) of the nodes and branches. G serves as a visual representation of the graph structure.

Extract Path algorithm implementation:

2) function -> (ComputeBFSPath)

```
function [Path,G,Parent] = computeBFSPath(AdjTable, vstart, vgoal)
    [Parent, G] = computeBFSTree(AdjTable, vstart);
    Path = [];
    current = vgoal;
    while current ~= vstart
        if Parent(current) == 0
            error('No path exists between vstart and vgoal.');
        end

        Path = [current, Path];
        current = Parent(current);
    end

    Path = [vstart, Path];
end
```

This function takes inputs from the adjacency table, Vstart, and VGoal. It outputs the Path, the Parent Vector, and G. The function calls “computeBFSTree” to compute the Parent Vector and G. It then uses the Parent Vector to determine the Path from the start node to the goal node.

Plotting Function 1 :

Extra) Function -> (plotPathOnBFSTree)

```
function plotPathOnBFSTree(AdjTable, Parent, Path)
% Initialize
    numNodes = numel(AdjTable);
    source = [];
    target = [];
    for i = 1:numNodes
        if Parent(i) ~= 0
            source = [source, Parent(i)];
            target = [target, i];
        end
    end

    BFS_Tree = digraph(source, target);
    pathEdgesSource = Path(1:end-1);
    pathEdgesTarget = Path(2:end);
    isPathEdge = ismember([source', target'], [pathEdgesSource', pathEdgesTarget'], 'rows');

  %% Plotting
    figure;
    h = plot(BFS_Tree, 'Layout', 'layered', 'ArrowSize', 10);
    hold on;
    highlight(h, pathEdgesSource, pathEdgesTarget, 'EdgeColor', 'r', 'LineWidth', 2);
    labelnode(h, 1:numNodes, string(1:numNodes));
    title('BFS Tree with Highlighted Path');
    xlabel('X-axis');
    ylabel('Y-axis');
    hold off;
end
```

This function takes in the adjacency table, the parent vector, and the path vector as inputs. The main purpose of the function is to compute PathEdgesSource and PathEdgesTarget, which define the edges along the path from the start node to the goal node. The PathEdgesSource is the list of nodes in the path, excluding the final goal node, and is extracted using Path(1:end-1). The PathEdgesTarget represents the list of nodes that directly follow each source node along the path and is extracted using Path(2:end). Together, these vectors represent the sequence of edges that form the path from the start to the end node. This allows the function to overlay the computed path onto the BFS tree, visually highlighting the route from the start node to the goal node.

Plotting Function 1 :

Extra) Function -> (plotWorkspaceWithPath)

```
function W = plotWorkspaceWithPath(AdjTable, Parent, Path)

% Initilization
    numNodes = numel(AdjTable);
    source = [];
    target = [];


    for i = 1:numNodes
        for j = 1:numel(AdjTable{i})
            if AdjTable{i}(j) ~= i
                source = [source, i];
                target = [target, AdjTable{i}(j)];
            end
        end
    end

    edgeTable = table(source', target', 'VariableNames', {'Source', 'Target'});
    edgeTable = unique(sortrows(sort(edgeTable.Variables, 2), 1), 'rows');
    W = graph(edgeTable(:, 1), edgeTable(:, 2));

    %% Plottingggg
    figure;
    h = plot(W, 'Layout', 'force', 'NodeLabel', 1:numNodes);
    hold on;

    if ~isempty(Path)
        pathEdgesSource = Path(1:end-1);
        pathEdgesTarget = Path(2:end);
        highlight(h, pathEdgesSource, pathEdgesTarget, 'EdgeColor', 'r', 'LineWidth', 2);
    end
    title('Workspace Graph with Highlighted Path');
    xlabel('X-axis');
    ylabel('Y-axis');
    hold off;
end
```

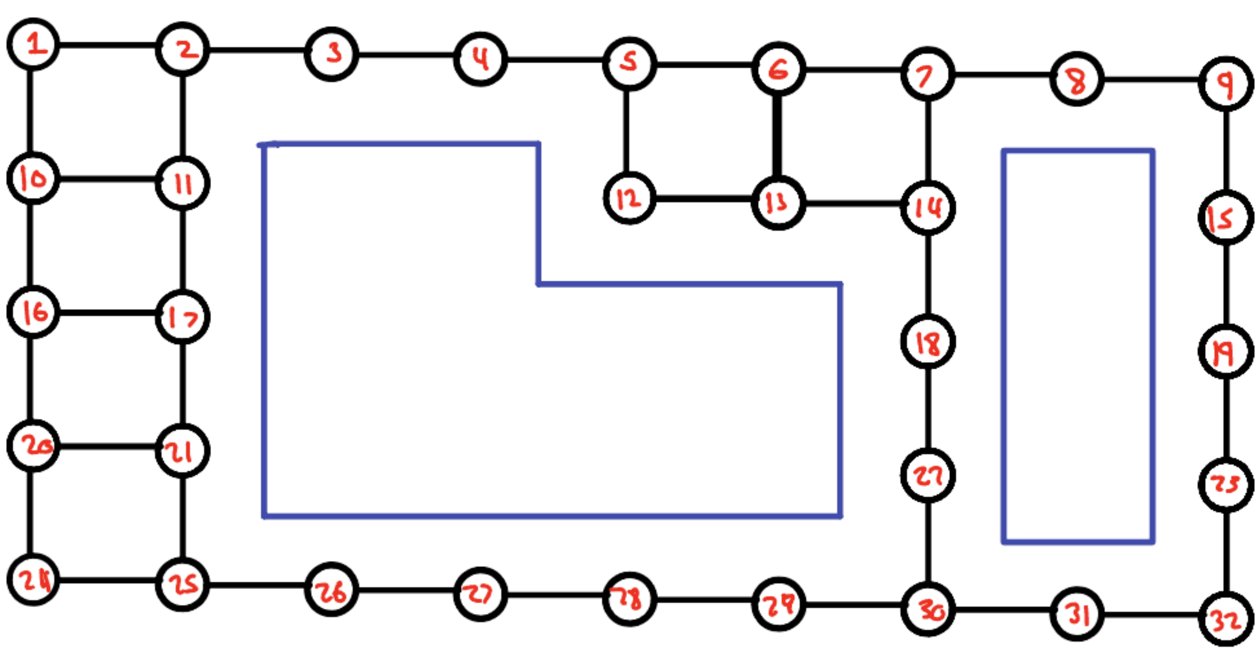
This function is very similar to the previous one, with most of the logic copied and pasted. The key difference here is that instead of plotting the BFS tree, this function plots the graph G directly in the workspace. It takes the adjacency table, parent vector, and path as inputs and constructs the graph by iterating through the adjacency table to create source and target node pairs, which are then used to build an edge table. The graph G is generated using these edges, and the plot is created using a force-directed layout for better visualization.

Running Algo:
Script->(TestFunc)

```
clc
clear
close all
%% Inputs
AdjTable  ={[2 10],[3 11 1],[4 2],[5 3],[6 12 4], [7 13 5],[8 14 6], [9 7], ...
            [15 8], [16 11 1],[2 10 17],[13 6 5],[14 6 12],[8 13 7],[19 9],[20 17 10],...
            [21 16 11],[22 14],[23 15],[24 21 16],[25 20 17],[30 18],[32 19],[25 20],...
            [26 24],[27 25],[28 26],[29 27],[30 28],[31 22 29],[32 30],[23 31]};

vstart = 1;
vGoal  = 32;
%% Run functions

[Path,G,Parent] = computeBFSPath(AdjTable,vstart,vGoal);

plotPathOnBFSTree(AdjTable, Parent, Path);

plotWorkspaceFromAdjTable(AdjTable,Parent, Path);

```

This script defines the main inputs: AjdTable, vstart, and goal. This script then runs all the functions discussed above.
Note: “computeBFStree” is nested inside “ComputeBFSPath”.

The adjacency table was defined by the labeling seen below. (I just counted up each node in rows.)



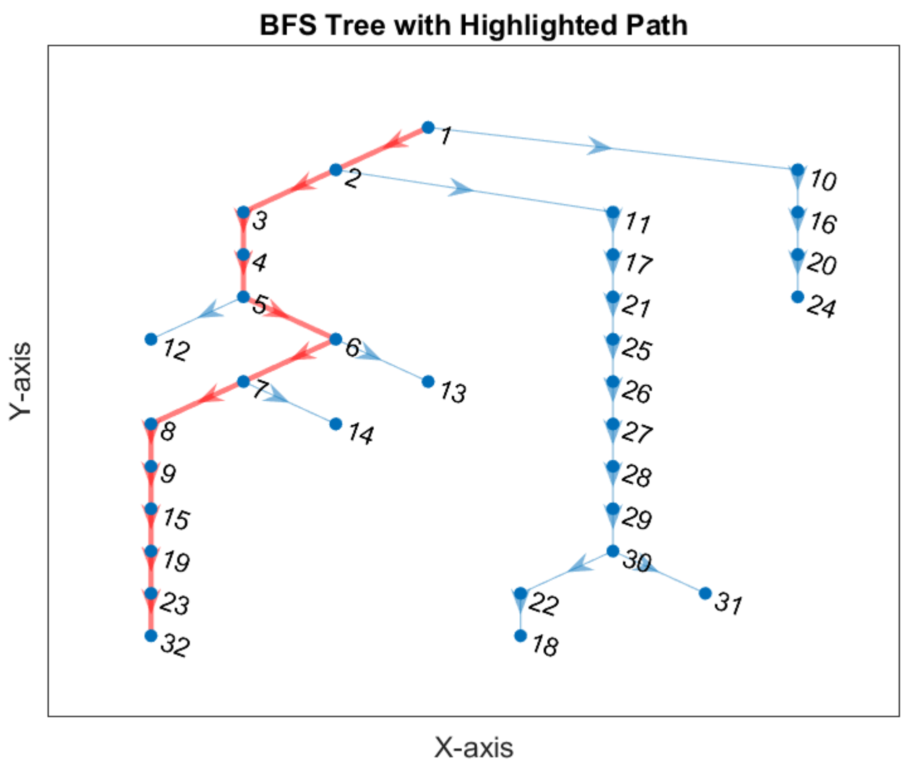
Output

Name	Value
AdjTable	1x32 cell
ans	1x1 graph
G	1x1 digraph
Parent	1x32 double
Path	1x13 double
vGoal	32
vstart	1

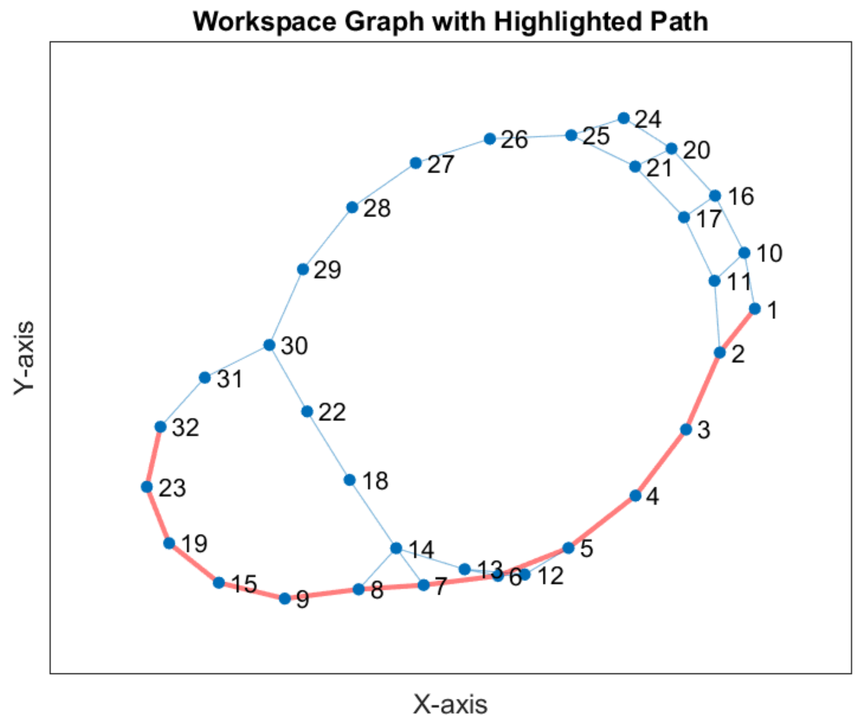
```
>> Path'

ans =

     1
     2
     3
     4
     5
     6
     7
     8
     9
    15
    19
    23
    32
```

Here is the Path from point 1 to point 32 (Vstart - Vgoal).

Below are the graphed outputs.



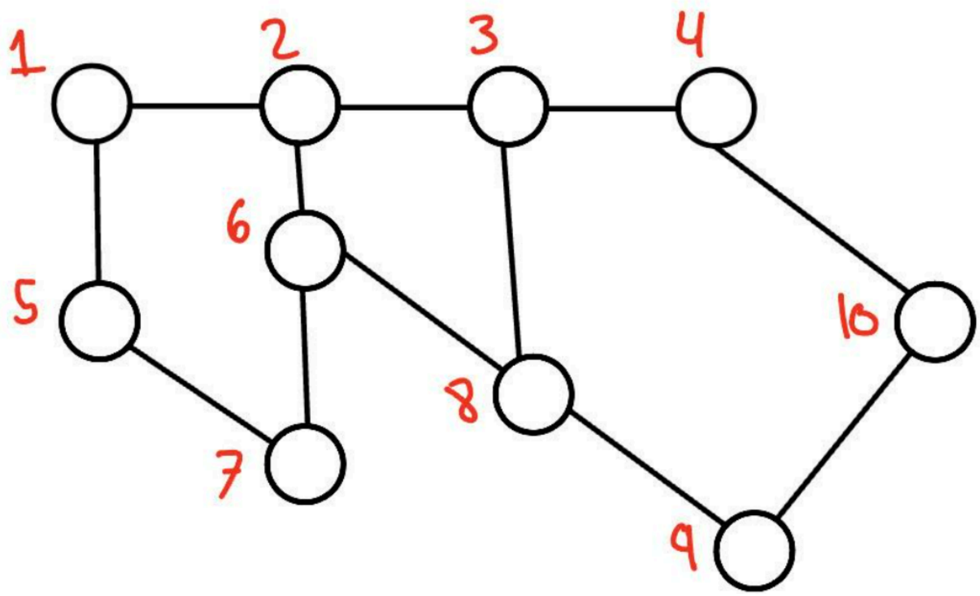
Here is the graph from “(plotPathOnBFSTree)”. This plot is in the form of a tree, given the adjacency table. Overlaid in red is the path from Node 1 to Node 32.



Here is a workspace representation of the path adjacency table created from "Function -> (plotWorkspaceWithPath)". Overlayed is the red line showing the workspace representation of the path from Node 1 to Node 32.

Another example for robustness:

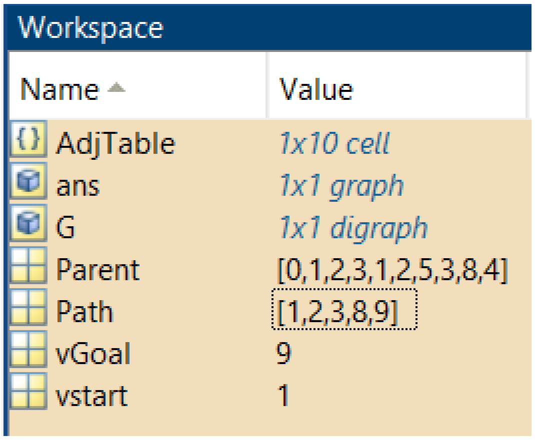
Let the start be, vStart = 1, and goal be, vGoal = 9

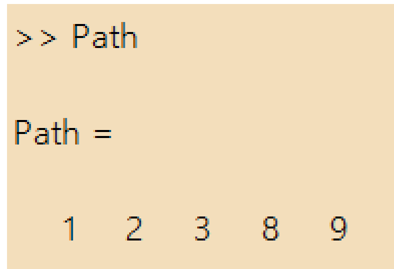


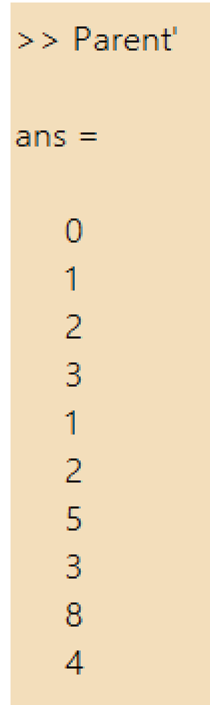
Inputs:

```
%% Inputs

AdjTable = { [2, 5],[1, 3, 6],[2, 4, 8],[3, 10],[1, 7]...
    ,[2,7, 8],[5, 6],[3, 6, 9],[8, 10],[4, 9] };

vstart = 1;
vGoal  = 9;
%% Run functions

[Path,G,Parent] = computeBFSPath(AdjTable,vstart,vGoal);

plotPathOnBFSTree(AdjTable, Parent, Path);

plotWorkspaceFromAdjTable(AdjTable,Parent, Path);

```

Output:

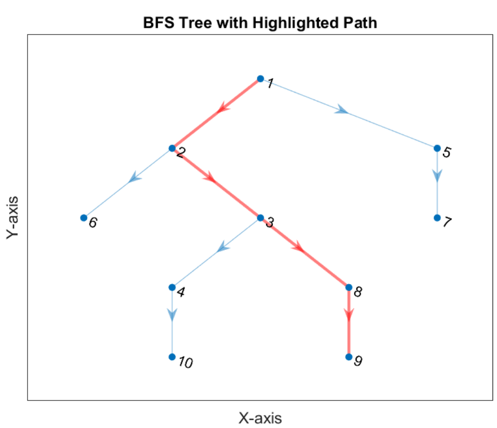




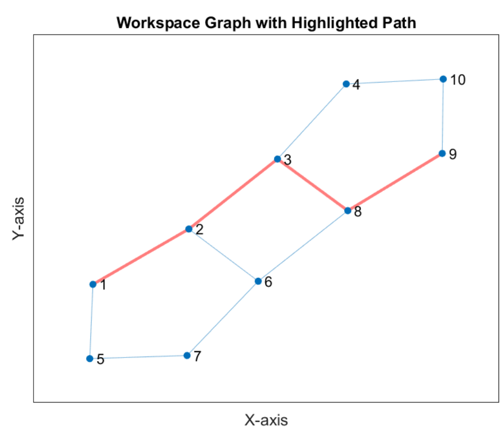


Visual plots:

Tree with colored path:



Workspace with colored path



Sweeping trapezoid extra credit:

Part 1) Pseudo Code

Sweeping trap: pseudo code

```
• Input → [workspace, obsticles]

• output → [ collected trapazoids]


% Initilize
collected vertex = [ ]
for each poly in obsticle
    for each vertex in obsticle
        V last = Prevouse Vertex in couter clock wise order
        V next = next vertex in counter clock wise order
        compute angle between edges

        if V is a local min (Both edges point up)
            if inner angle < 180°
                Label V as "start v"
            else
                Label V as "split v"

        else if V is a local max (Both edges go down)
            if the inner angle < 180°
                Lable V as "end v"
            else
                Lable V as "merged V"

        else
            if Vnext(y) > Vy and Vlast(y) < V.y
                Label V as "Left V"
            else
                Label V as "right V"
        return (Vcurrent) → collected vertex
```

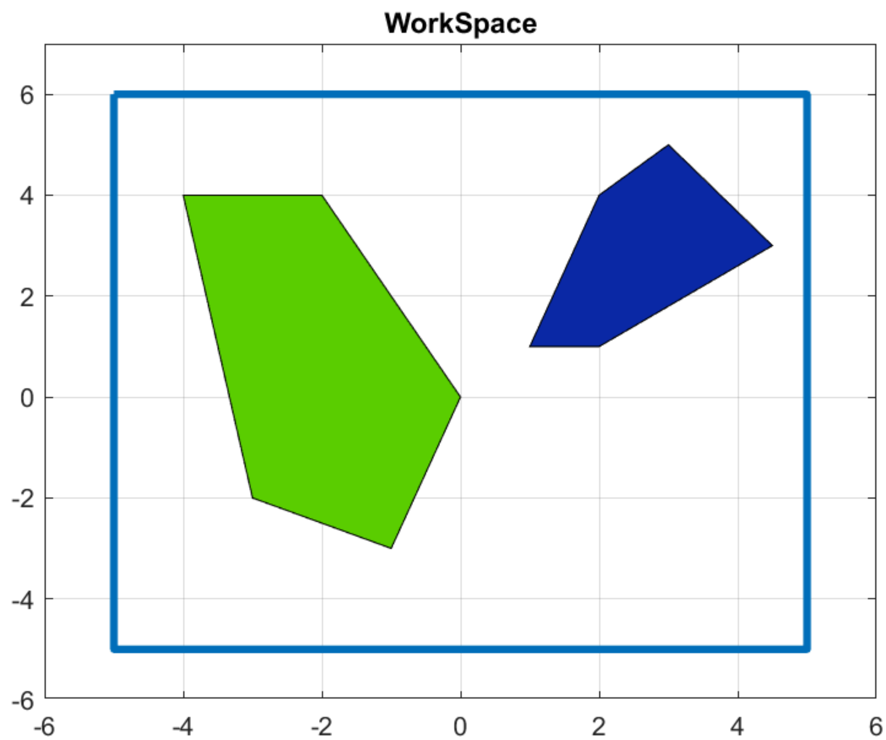
Part 2)
Implementation:

Step by step (1): Inputs

```
clc
close all
clear
%% Input

%Workspace

Ymax   = 6;
Ymin   =-5;
Xwidth =10;

W = [Ymax Ymin Xwidth]; % Assume (0,0) is origin


%extracting work space
w1 = [-W(3)/2 W(1)];
w2 = [W(3)/2 W(1)];
w3 = [W(3)/2 W(2)];
w4 = [-W(3)/2 W(2)];
w5 = [-W(3)/2 W(1)];


WorkSpace = [w1; w2; w3; w4; w5];

% Obsticles

Q1 = [-4 4 ;-2 4 ;0 0 ; -1 -3;-3 -2];
Q2 = [1 1; 2 4; 3 5; 4.5 3; 2 1];

P(:,:,1) = Q1;
P(:,:,2) = Q2;
```

Here P and W will be the inputs for the sweeping trapezoid function.
P is a 3D matrix with dimensions ({x},{y},n) where n is the obstacle number. Q = obstacle, So Q_n will denote which obstacle. Here we are inputting two obstacles inside P.

With the P and W, we can define the workspace and obstacle locations. Shown in the figure below. For fun, I have the colors of the obstacle chosen at random.



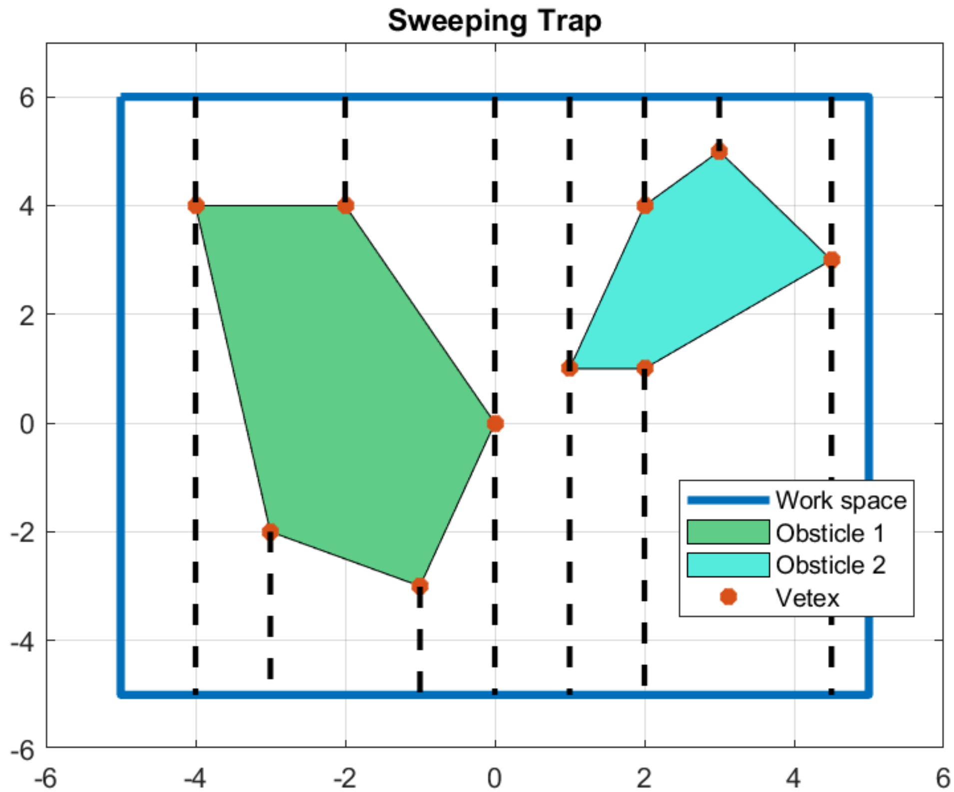
Step by step (2):

Labeling vertices and drawing vertical lines that stop at work space and obstacle boundaries.

```
%% algo, vertex and lines

% Getting all x and y Vales
n=0;
for i =1:length(P(1,1,:))
    for j =1:length(P)
    n = n+1;
    Px(n) = P(j,1,i);
    Py(n) = P(j,2,i);
    end
end
Xobj = [Px;Px];

% getting points from left to right

for i =1:length(Px)
    xmin = min(Px);
    finder = find(Px ==xmin);

    if length(finder)>=2
        for j = 2:length(finder)
            finder(j) =[];
        end
    else
    end

    Px(finder) = [];
    STGx(i) = xmin;
    STGy(i) = Py(finder);
    Py(finder) = [];
end

% Creaing verticle line

for i = 1:length(STGx)
VLine(:,1,i) =  [STGx(i) Ymax];
VLine(:,2,i) =  [STGx(i) Ymin];

Vlinex(:,i) = [STGx(i) STGx(i)];
Vliney(:,i) = [Ymax Ymin];
end
```

```

% stoping the vertivcle line from going through the obsticle

n=0;
for i =1:length(P(1,1,:))

     Que = 1:length(P);
     Que = [length(P) Que 1];
    for j =1:length(P)
        n = n+1;

        P_current    = P(j,1,i);
        P_node_back  = P(Que(j),1,i);
        P_node_front = P(Que(j+2),1,i);

       if (P_current < min(P_node_front, P_node_back)) || (P_current > max(P_node_front, P_node_back))

       else
           if mean(P(:,2,i)) > P(j,2,i)
                Vliney(:,n) = [P(j,2,i) Ymin];
           else
                Vliney(:,n) = [Ymax P(j,2,i)];
           end

       end
    end
end
```

At this point, we have the vertices labeled from smallest to greatest (Vlinex(:,:)), as well as the y coordinates for drawing the lines (Vliney). At this point, we can add the labeled vertices and lines to the workspace. This is shown below.



Step by step (3): Extracting created segments by the dashed lines.

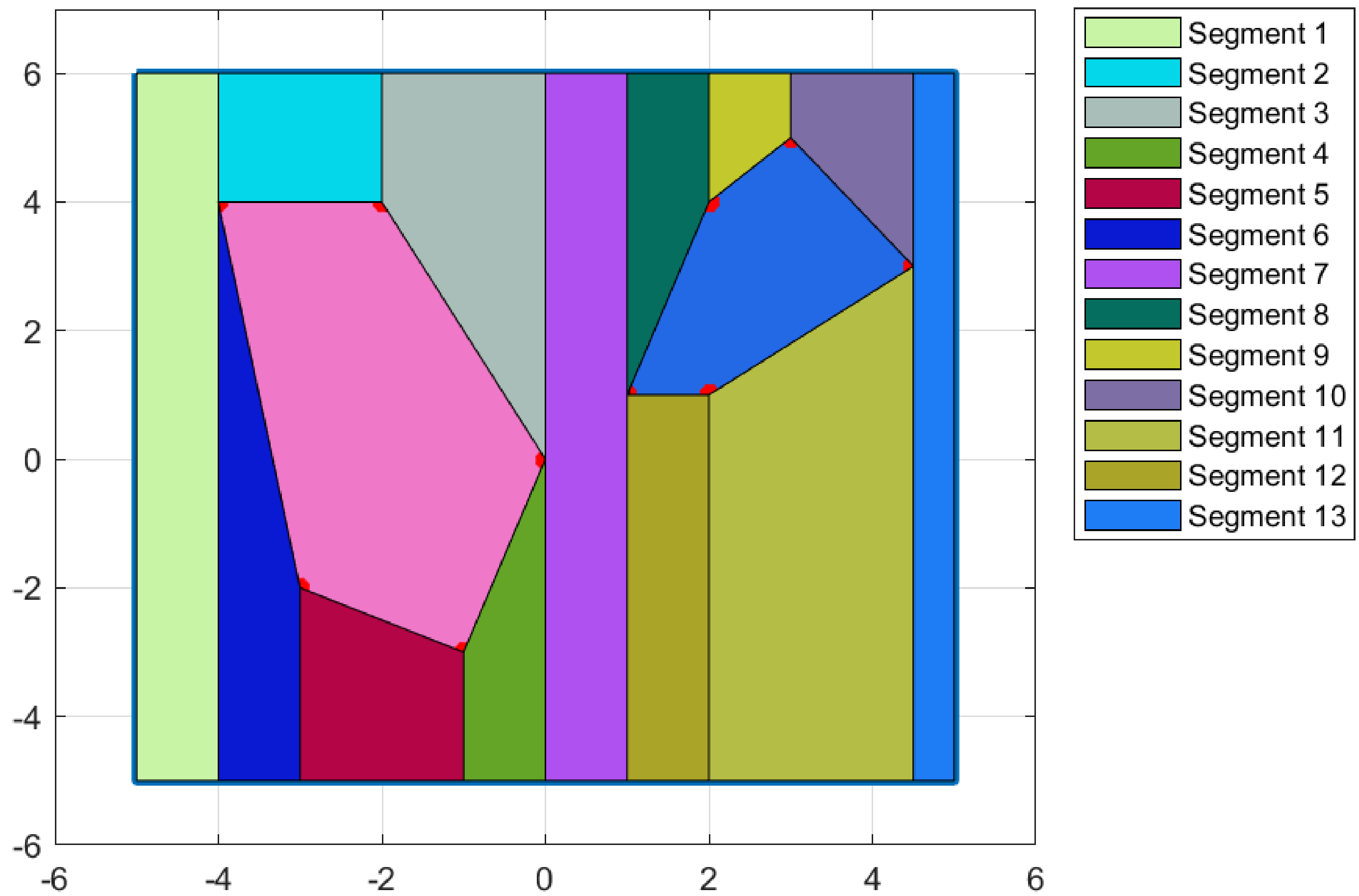
Since we know where the vertices are and the max and min points from Vliney. This just becomes a checking game to properly label all the vertices correctly to create the segment of each interval. The code is shown below.

```
%% Getting Segments

%sweeping from left to right

n=0;
m=0;
k=0;

for i =1:length(P(1,1,:))
    xlast  = -W(3)/2;
    ylast1 = Ymax;
    ylast2 = Ymin;

    if i>1
    xlast  = max(P(:,1,i-1));
    else
    end

    Que = 1:length(P);
    Que = [length(P) Que 1];

    for j =1:length(P)+1
        n = n+1;
        m = m+1;
        k = k+1;

         if j == length(P)+1
            k = k-1;
            P_current    = [P(1,1,i) P(1,2,i)];
            P_node_back  = [P(Que(1),1,i) P(Que(j),2,i)];
            P_node_front = [P(Que(1+2),1,i) P(Que(1+2),2,i)];

            x1 = P(1,1,i);
            x4 = x1;
            x2 = xlast;
            x3 = x2;

            y1 =  P(1,2,i);
            y4 =  Vliney(2,k);

            if x1 <= x2

            y2 = P(Que(1),2,i);
            y3 = Ymin;
            else

            y2 = ylast1;
            y3 = P(Que(1),2,i);
            end

            Poly(:,:,m) = [x1 x2 x3 x4; y1 y2 y3 y4];
         else
            P_current    = [P(j,1,i) P(j,2,i)];
            P_node_back  = [P(Que(j),1,i) P(Que(j),2,i)];
            P_node_front = [P(Que(j+2),1,i) P(Que(j+2),2,i)];

            if (P_current(1) < min(P_node_front(1), P_node_back(1))) || (P_current(1) > max(P_node_front(1), P_node_back(1)))
                y3 = ylast2;

                x1 = P(j,1,i);
                x4 = x1;
                x2 = xlast;
                x3 = x2;
                y1 =  Ymax;

                if y3 == Ymin
                   y4 =  Ymin;
                else
                   y4 = P(j,2,i);
                end
                y2 = ylast1;
                y3 = ylast2;
                Poly(:,:,m) = [x1 x2 x3 x4; y1 y2 y3 y4];

```

```
                else
                    x1 = P(j,1,i);
                    x4 = x1;
                    x2 = xlast;
                    x3 = x2;
                    y1 =  Vliney(1,k);
                    y4 =  Vliney(2,k);

                    if x1 <= x2
                    y2 = P(Que(j),2,i);
                    y3 = Ymin;
                    else
                    y2 = ylast1;
                    y3 = P(Que(j),2,i);
                    end

                 Poly(:,:,m) = [x1 x2 x3 x4; y1 y2 y3 y4];
                end

                xlast  = x1;
                Ylast1 = y1;
                ylast2 = y4;
            end
        end
    end
```

```

    % last segment
    xlast  = max(P(:,1,length(P(1,1,:))));
    x1 = xlast;
    x4 = x1;
    x2 = Xwidth/2;
    x3 = x2;

    y1 = Ymax;
    y2 = Ymax;
    y4 = Ymin;
    y3 = Ymin;
    Poly(:,:,m+1) = [x1 x2 x3 x4; y1 y2 y3 y4];

```

That code seems long, but all it's doing is checking for the vertex of the 6 possible vertices configurations to properly label the vertices, hence all of the if statements. At this point, the code is done and we can output the results of the segments. The segments all get stored in a 3d matrix called Poly({x}{y},n,m). m = segment number. The segment results are shown below:
N = number of points in a segment


6
4
2
0
-2
-4
-6
-6
-4
-2
0
2
4
6
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
Segment 6
Segment 7
Segment 8
Segment 9
Segment 10
Segment 11
Segment 12
Segment 13

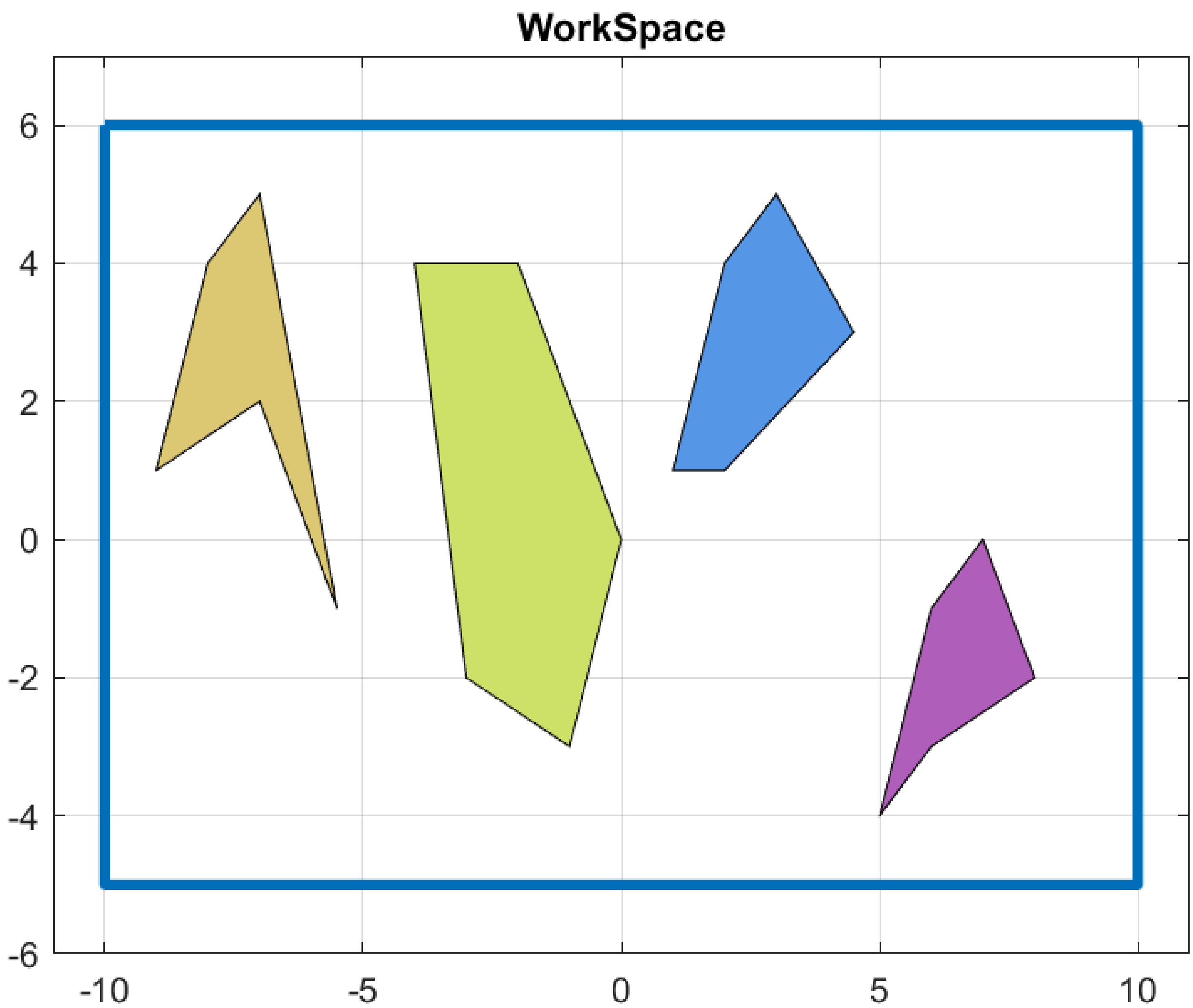
```
val(:,:,1) =

    -4    -5    -5    -4
     6     6    -5    -5


val(:,:,2) =

    -2    -4    -4    -2
     6     6     4     4


val(:,:,3) =

     0    -2    -2     0
     6     6     4     0


val(:,:,4) =

    -1     0     0    -1
    -3     0    -5    -5


val(:,:,5) =

    -3    -1    -1    -3
    -2    -3    -5    -5


val(:,:,6) =

    -4    -3    -3    -4
     4    -2    -5    -5


val(:,:,7) =

     1     0     0     1
     6     6    -5    -5


val(:,:,8) =

     2     1     1     2
     6     6     1     4


val(:,:,9) =

     3     2     2     3
     6     6     4     5


val(:,:,10) =

    4.5000    3.0000    3.0000    4.5000
    6.0000    6.0000    5.0000    3.0000


val(:,:,11) =

    2.0000    4.5000    4.5000    2.0000
    1.0000    3.0000   -5.0000   -5.0000


val(:,:,12) =

     1     2     2     1
     1     1    -5    -5


val(:,:,13) =

    4.5000    5.0000    5.0000    4.5000
    6.0000    6.0000   -5.0000   -5.0000
```

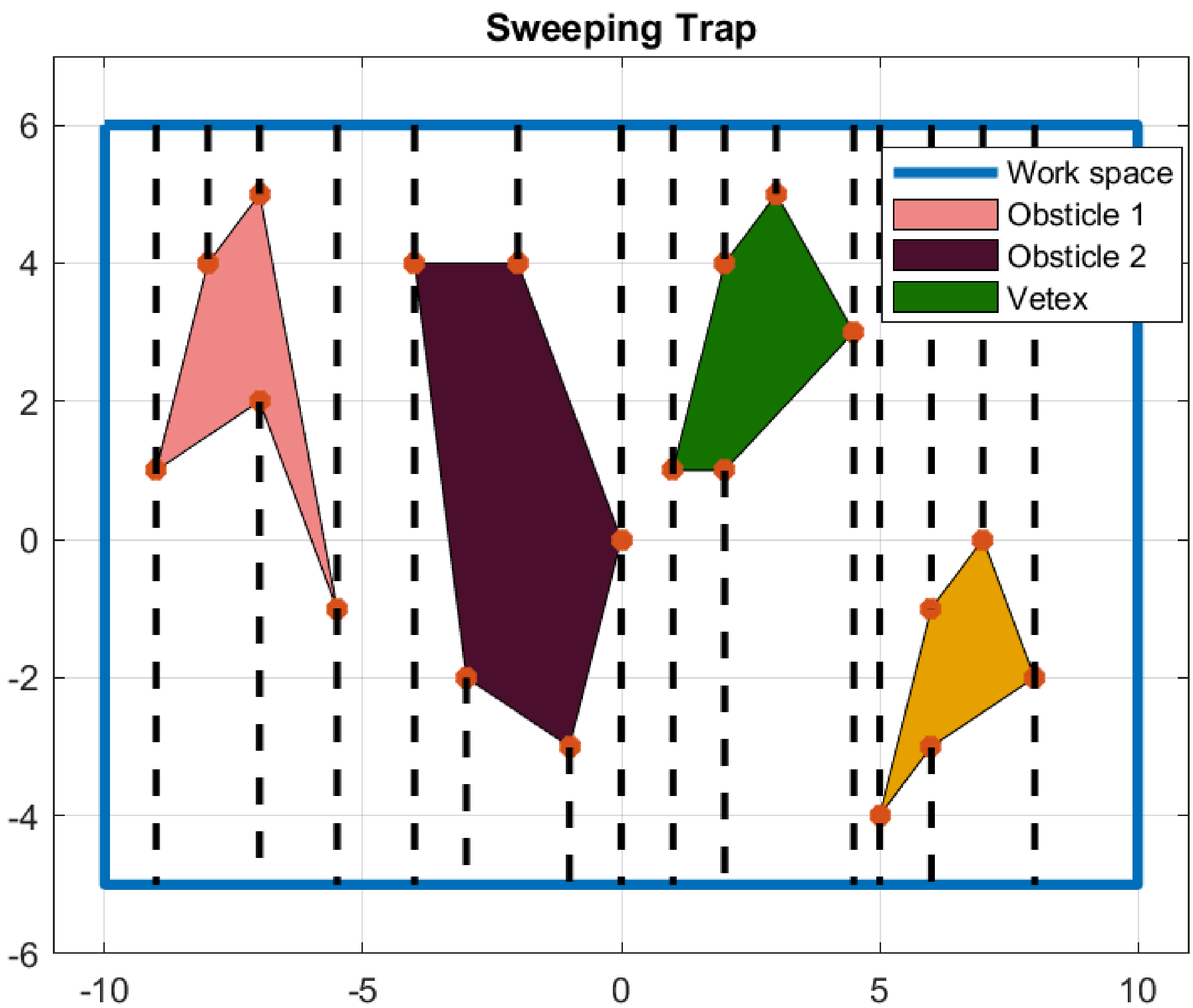
Segment Output:
Row 1 = X vals
Row 2 = Y vals

The Last thing there is for us to do is make it an office function and test new inputs for robustity. The function is colled “SweepingAlgo(W,P)” Takes in the workspace and obstacles P.
The function has all the code above. Let's try adding two more shapes and expanding the work space!

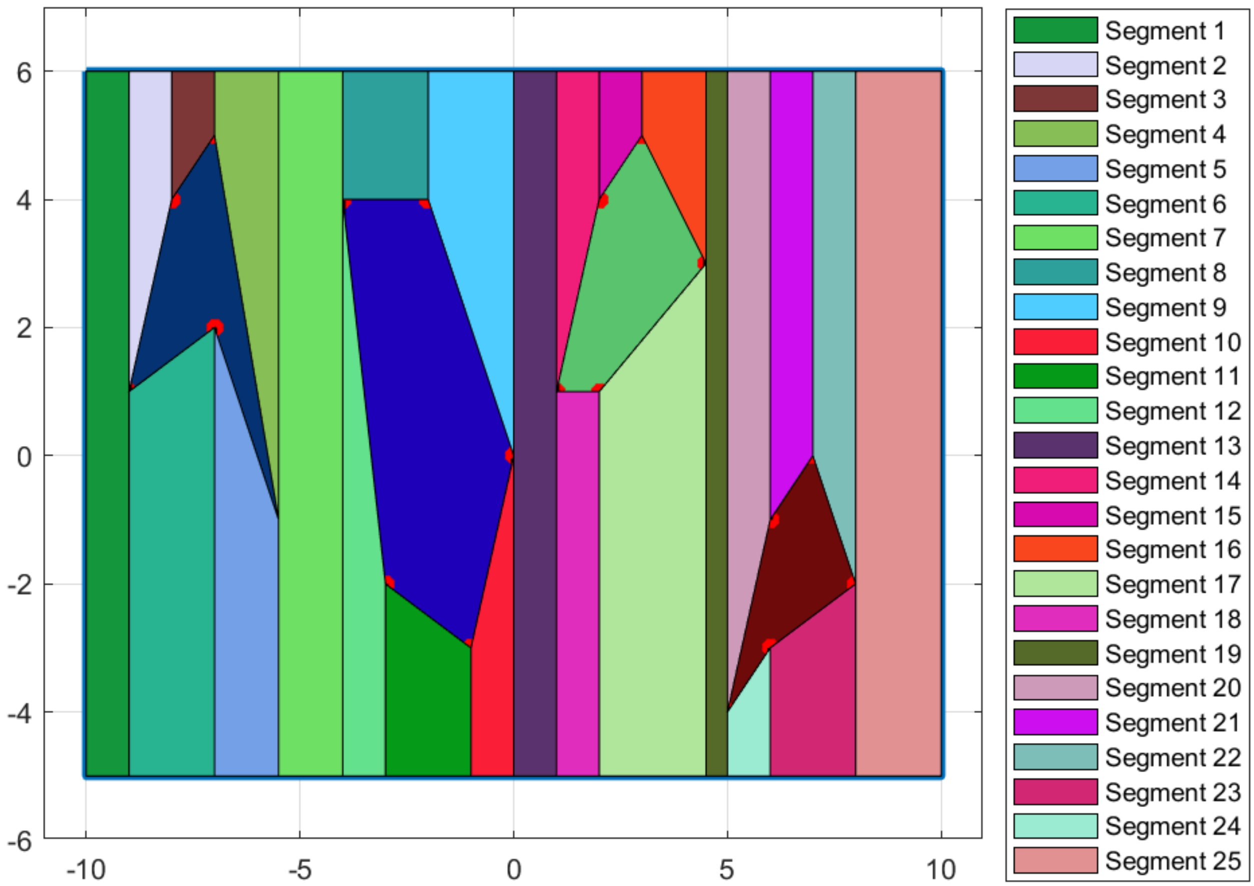
```
%% Runing sweepig trap

% Inputa

%Workspace

Ymax   = 6;
Ymin   =-5;
Xwidth =20;
W = [Ymax Ymin Xwidth]; % Assume (0,0) is origin

% Obsticles

Q1 = [-4 4 ;-2 4 ;0 0 ; -1 -3;-3 -2];
Q2 = [1 1; 2 4; 3 5; 4.5 3; 2 1];
Q3 = [5 -4; 6 -1; 7 0; 8 -2; 6 -3];
Q4 = [-9 1; -8 4; -7 5; -5.5 -1; -7 2];

P(:,:,2) = Q1;
P(:,:,3) = Q2;
P(:,:,4) = Q3;
P(:,:,1) = Q4;


% Running function
segment = SweepingAlgo(W,P);
```

Here are the results:


WorkSpace
6
4
2
0
-2
-4
-6
-10
-5
0
5
10



Sweeping Trap
Work space
Obsticle 1
Obsticle 2
Vetex
6
4
2
0
-2
-4
-6
-10
-5
0
5
10



6
4
2
0
-2
-4
-6
-10
-5
0
5
10
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
Segment 6
Segment 7
Segment 8
Segment 9
Segment 10
Segment 11
Segment 12
Segment 13
Segment 14
Segment 15
Segment 16
Segment 17
Segment 18
Segment 19
Segment 20
Segment 21
Segment 22
Segment 23
Segment 24
Segment 25

```
>> segment

segment(:,:,1) =

    -9   -10   -10    -9
     6     6    -5    -5


segment(:,:,2) =

    -8    -9    -9    -8
     6     6     1     4


segment(:,:,3) =

    -7    -8    -8    -7
     6     6     4     5


segment(:,:,4) =

   -5.5000   -7.0000   -7.0000   -5.5000
    6.0000    6.0000    5.0000   -1.0000


segment(:,:,5) =

   -7.0000   -5.5000   -5.5000   -7.0000
    2.0000   -1.0000   -5.0000   -5.0000


segment(:,:,6) =

    -9    -7    -7    -9
     1     2    -5    -5


segment(:,:,7) =

   -4.0000   -5.5000   -5.5000   -4.0000
    6.0000    6.0000   -5.0000   -5.0000


segment(:,:,8) =

    -2    -4    -4    -2
     6     6     4     4


segment(:,:,9) =

     0    -2    -2     0
     6     6     4     0


segment(:,:,10) =

    -1     0     0    -1
    -3     0    -5    -5


segment(:,:,11) =

    -3    -1    -1    -3
    -2    -3    -5    -5


segment(:,:,12) =

    -4    -3    -3    -4
     4    -2    -5    -5


segment(:,:,13) =

     1     0     0     1
     6     6    -5    -5
```

```
segment(:,:,16) =

    4.5000    3.0000    3.0000    4.5000
    6.0000    6.0000    5.0000    3.0000


segment(:,:,17) =

    2.0000    4.5000    4.5000    2.0000
    1.0000    3.0000   -5.0000   -5.0000


segment(:,:,18) =

     1     2     2     1
     1     1    -5    -5


segment(:,:,19) =

    5.0000    4.5000    4.5000    5.0000
    6.0000    6.0000   -5.0000   -5.0000


segment(:,:,20) =

     6     5     5     6
     6     6    -4    -1


segment(:,:,21) =

     7     6     6     7
     6     6    -1     0


segment(:,:,22) =

     8     7     7     8
     6     6     0    -2


segment(:,:,23) =

     6     8     8     6
    -3    -2    -5    -5


segment(:,:,24) =

     5     6     6     5
    -4    -3    -5    -5


segment(:,:,25) =

     8    10    10     8
     6     6    -5    -5
```

A total of 25 segments where created in this run