Solutions for HW5

Yunhai Han

February 15, 2020

1 E4.1 Minkowski difference

1.1 i

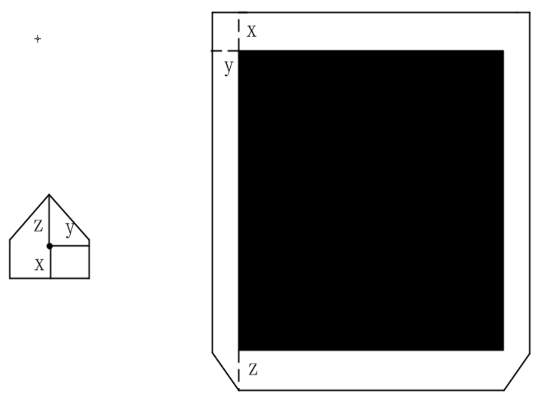

Figure 1: The trajectory of the robot's reference point

In Fig 1, I assume the shape of the robot is symmetric along the the vertical x-axis. Due to its ship-like shape, there are two triangles at the both sides of the bottom line of the obstacle.

1.2 ii

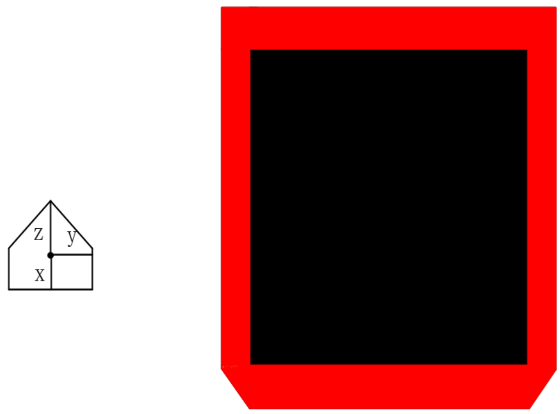

Figure 2: The configuration space obstacle for a translating robot

In Fig 2, the red space represents the new configuration space obstacle caused by the shape of the robot. For different robot, the shape or the area of the red space could be different, but the black space which represent the original obstacle could be the same.

1.3 iii

From my perspective, there are two main reasons:

- The robot body and the obstacle could both be considered as polygons with n and m vertices and the resulting configuration space obstacle at most has $n + m$ vertices. In this sense, the most efficient algorithm could be implemented to run in $O(n+m)$. besides, if we don't use this property, we have in total nm difference points and we would extract a convex hull from these points. The runtime of such algorithm is $O(nmlog(nm))$ and I think it is affordable on most computing machines because in fact, it is not necessary to use lots of vertices to represent any obstacles and the robot body(nm is not very large).
- It is also a natural way to represent robots and obstacles using their vertices. For a computer program, it doesn't have the visual ability to distinguish between different shapes but only compute with the numbers. The Minkowski difference only involves the vertices coordinates, which is the most suited way for computer program.

2 E4.3 Programming:Sampling algorithms

2.1 i

In this part, we are required to draw sample points in the uniform center grid. From the title, the d is given and it is 2. There are two steps:

- Along each of the 2 dimensions, divide the [0,1] into k subintervals of equal length and therefore compute k^2 sub-cubes of $X = [0, 1]^2$.
- Place one grid point at the center of each sub-cube

2.2 ii

In this part, we are required to draw sample points in the uniform corner grid. From the title, the d is given and it is 2. There are two steps:

- Along each of the 2 dimensions, divide the [0,1] into $k - 1$ subintervals of equal length and therefore compute $(k - 1)^2$ sub-cubes of $X = [0, 1]^2$.
- Place one grid point at each vertex of each sub-cube

2.3 iii

2.3.1 computeGridSukharev

Just as I describe in the previous two sections, I draw sample points along each axis. There are no special cases.

2.3.2 computeGridRandom

It is very easy to generate n random sample points: in Python, we could import numpy module and run $numpy.random.sample(n)$ 2 times. There are no special cases.

2.3.3 computeGridHalton

Halton sequences are generated by prime numbers and we could follow a rule to generate each Halton sequence. Since it is deterministic with respect to each prime number, we could enter arbitrary prime number. Also, the lecture notes have already provided with the algorithm to generate Halton sequence. The special case is when the two prime numbers are the same. In this case, all the sample points are on the diagonal line.

2.3.4 Verification

We are required to verify the correctness of the function by plotting the three grids for $n = 100$. Here, I select 2 and 3 as the two prime numbers for the Halton grid.

```
if __name__ == '__main__':
    title=["Sukharev Grid","Random Grid","Halton Grid"]
    number_samples = 10**2
    if number_samples < 0:
        raise ValueError("The number of samples must be larger than zero")
    X,Y = computeGridSukharev(number_samples)
    show_results(X,Y,title[0])
    X,Y = computeGridRandom(number_samples)
    show_results(X,Y,title[1])
    X,Y = computeGridHalton(number_samples,2,3)
    show_results(X,Y,title[2])
```

Figure 3: Python codes

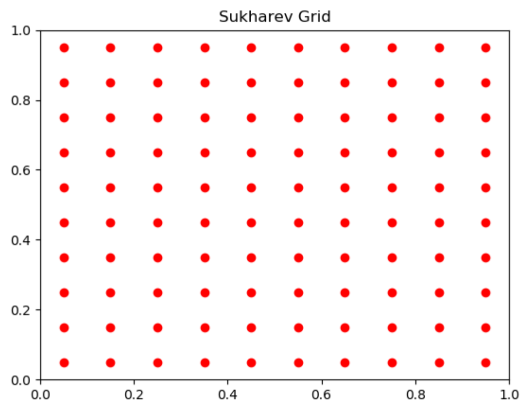

Figure 4: Center grid

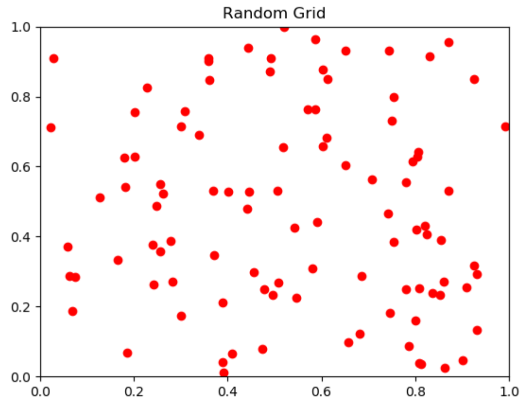

Figure 5: Random grid

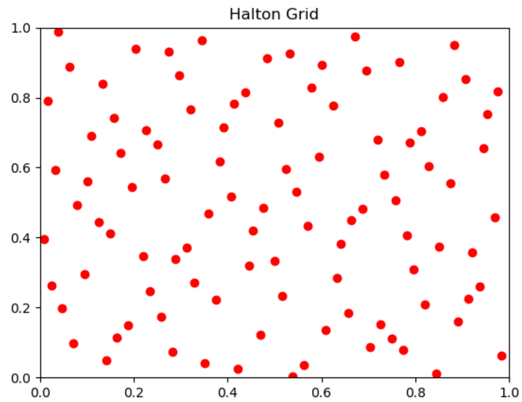

Figure 6: Halton grid

For the special cases of Halton grid, I enter two prime numbers which are the same.

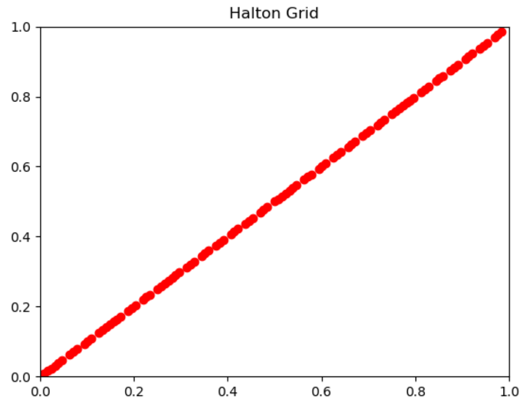

Figure 7: Halton grid with two same prime numbers

3 Programming exercise:RRT planner

3.1 i

First of all, we need to check the random whether the sample point is in the obstacle or not. If it is in the obstacle, we need to generate another one. This could be done by the function $isPointInConvexPolygon$ in **auxiliaryfunctions**.

In order to find the nearest node in the tree to the sampled point, we have to first define the distance function. Since the configuration space here is R^2, the distance function is simple:

$$\textbf{dist}_{R^2} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

We can use a list V to store all the nodes in the tree and compare the distance between each of them with the random sample point to find the nearest one. We return the index of the nearest node by the function *getNearestNode*.

3.2 ii

From the previous part, we already know the index of the nearest node in the tree to the sample point. Then, we compare the smallest distance with 0.25(it could be any constant or even a variable) as the problem required. If the distance is smaller, we just add the sample point into the tree list and its parent node is the nearest node. Otherwise, we have to generate another node according to the sample point and the nearest node as given in the title. Also, its parent node is the nearest node. Still, we have to check whether the new node is inside the obstacle or not.

3.3 iii

Check if the new point connects to the tree you are building without hitting an obstacle. We could use the function $inCollision$ in **auxiliaryfunctions**. In the function, the number of samples on each segment is 3, but the python function $range(1, 3)$ would only contain 1 and 2. Too few samples are not enough to guarantee the accuracy because in some cases, there may exist some thin obstacles. If the samples are sparse on the segment, we would not detect these thin obstacles. To be honest, it is a trade-off between running time and accuracy. If the performance of the algorithm is poor, we could tune these parameters to improve it.

3.4 iv

If we can connect the tree to the new sample point by a collision-free path, the sample point could be added into the tree list and set its parent node. This could be easily done in the codes. You could see how I implement this from the codes(in **Appendix**).

3.5 Results

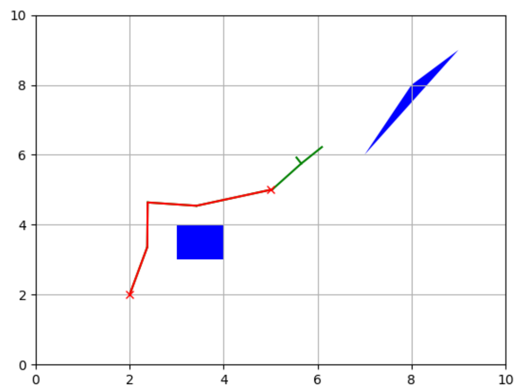

Figure 8: The tree roadmap

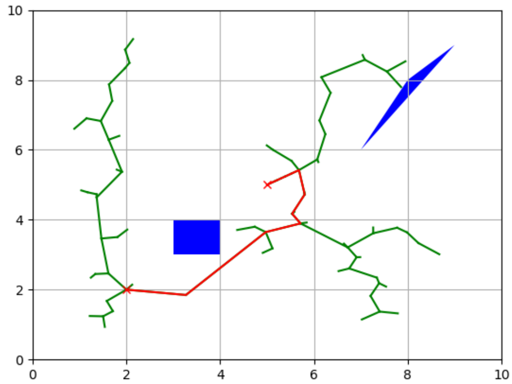

Figure 9: The tree roadmap

The above two figures are drew with the same set of parameters.

I tune the choice (0.75,0.25) to other values (0.65,0.35). Intuitively, since the random sample points share larger weights, the algorithm should find the path quickly, because the node added into the tree-list is closer to the goal instead of its parent node. On the other side, the length of the segment grows larger but the number of sample points on the segment remain unchanged, which means the distance between any two sample points increases. And this could increase the possibility of some thin obstacles not being detected. In other words, it makes the algorithm less conservative. You could see the result in the following figure.

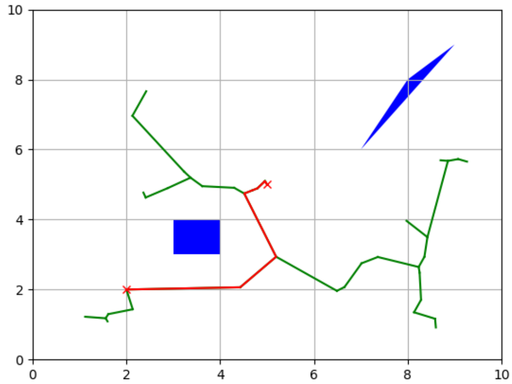

Figure 10: The tree roadmap

Besides, I rewrite the function $isPointInConvexPolygon$ as $checkPointInsidePolygon$, which I wrote in homework2. And you could see my function also works! I add two more obstacles into the workspace(in this case, also the configuration space) and add more sample points on each segment.

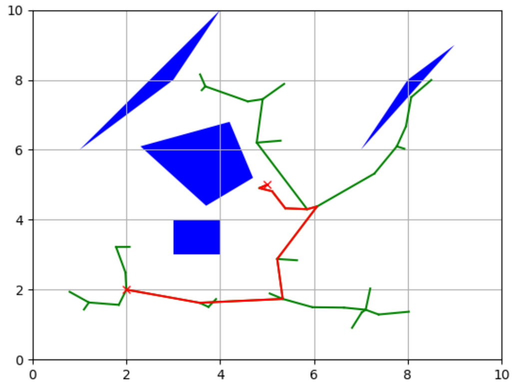

Figure 11: The tree roadmap

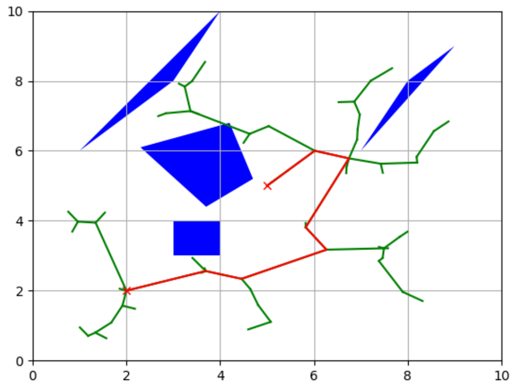

Figure 12: The tree roadmap

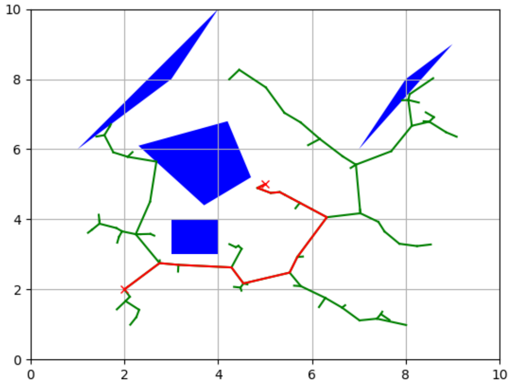

Figure 13: The tree roadmap

To sum up, it is an art to tune all these parameters and find the best solution.

3.6 Appendix

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# auxiliary_functions.py
import math
import copy
import numpy as np

"""
@author:Yunhai Han

This function returns the ith Halton value with p1 as prime number
    and
n as index
"""
def computeGridHalton(n,p1):
    X=0
    p1_tmp = n
    f1 = 1/p1
    while p1_tmp > 0:
        q=p1_tmp//p1
        r=p1_tmp%p1
        X=X+f1*r
        p1_tmp=q
        f1=f1/p1
    return X
```

```
"""
@author: sonia

This function samples a segment in the plane according to
a Halton sequence by sampling over [0, 1] and then using
the parametrization of the segment to map to the
corresponding point in the segment.

The function can be extended to any parametrized curve.
"""


def haltonPointInSegment(n, base, q1, q2):
    # parameters:
    # n = the nth point in the Halton sequence
    # base = the prime number basis of the Halton sequence
    # q1 = first end point of segment, entered as a list
    # q2 = second end point of segment, entered as a list

    # convert to np arrays
    n = int(n)
    base = float(base)

    # nth halton sequence point on [0, 1]
    # To Do: obtain pB the nth point of the Halton sequence
    # associated with the prime 'base'
    pB=computeGridHalton(n,base)
    # Obtain point in segment
    q = (pB*q1[0] + (1 - pB)*q2[0], pB*q1[1] + (1 - pB)*q2[1])
    return q


"""
Patrick Therrien, May 5,2015
This code takes a point and a polygon and checks whether or not the
    point is
inside the polygon by created normal vectors to each segment and
    checking the
dot product with a vector from the initial point of the segment to
    the given
point q
"""


def dist(p1, p2):
    return np.sqrt(np.square(p1[0] - p2[0]) + np.square(p1[1] -
        p2[1]))
        return 0,1,-y1
```

```
"""
Author:Yunhai Han
"""
def computeLineThroughTwoPoints(p1,p2): #from homework1
    x1=p1[0]
    y1=p1[1]
    x2=p2[0]
    y2=p2[1]
    if y1-y2!=0:
        a=1/(math.sqrt(1+np.square((x1-x2)/(y1-y2))))
        b=-((x1-x2)/(y1-y2))*a
        c=-x1*a-y1*b
        return a,b,c
    elif x1==x2:
        return 0,0,0
    else:
        return 0,1,-y1
"""
Author:Yunhai Han
Function:Point inside Polygon or not
"""
def checkPointInsidePolygon(q,mypolygon):
    if(q in mypolygon):
        return 1
    else:
        vertices_number=len(mypolygon)
        number_intersection=0
        for index in range(0,vertices_number):
            if(index!=vertices_number-1):
                d_min=min(mypolygon[index][0],mypolygon[index+1][0])
                d_max=max(mypolygon[index][0],mypolygon[index+1][0])
                if(q[0]<=d_max and q[0]>d_min):
                    (a,b,c)=computeLineThroughTwoPoints(mypolygon[index],mypolygon[index+1
                    value_y=(-a*q[0]-c)/b
                    if(q[1]>=value_y):
                        number_intersection+=1
            else:
                d_min=min(mypolygon[index][0],mypolygon[0][0])
                d_max=max(mypolygon[index][0],mypolygon[0][0])
                if(q[0]<=d_max and q[0]>d_min):
                    (a,b,c)=computeLineThroughTwoPoints(mypolygon[index],mypolygon[0])
                    value_y=(-a*q[0]-c)/b
                    if(q[1]>=value_y):
                        number_intersection+=1
        if(number_intersection % 2 == 0):
            return 0
        else:
            return 1
def isPointInConvexPolygon(q, P):
```

```python
    """
    :param q: a point
    :param P: a list of points to define a polygon(obstacle)
    :return: 0 if point is outside of polygon, 1 if point is on or
       inside
    the polygon
    """
    polyPList = copy.deepcopy(P) # this ensures the changes stay local

    polyPList.append(polyPList[0])
    # Initialize relevant variables
    pV = [0, 0]
    fail = 0
    qV = [0, 0]
    pPHV = [0, 0]
    passVar = 0

    for i in range(len(polyPList) - 1): # If the point is a vertex,
       autopass
        if q == polyPList[i]:
            passVar = 1
    if passVar == 0:
        for j in range(len(polyPList) - 1):
            p1 = polyPList[j]
            p2 = polyPList[j + 1]
            # create vector along segment
            pPHV[0] = (p2[0] - p1[0]) / dist(p1, p2)
            pPHV[1] = (p2[1] - p1[1]) / dist(p1, p2)
            # rotate vector 90deg so it is normal to segment
            pV[0] = pPHV[1] * -1
            pV[1] = pPHV[0]
            # create vector from q point to first segment point
            qV[0] = (q[0] - p1[0]) / dist(q, p1)
            qV[1] = (q[1] - p1[1]) / dist(q, p1)
            # take dot product and if it is negative then q is outside
            # respective plane
            dotP = np.dot(pV, qV)
            if dotP < 0:
                fail = 1

    del polyPList # make sure variable is not reused

    if fail:
        return 0
    else:
        return 1


def inCollision(nearest_node, new_point, obstacleList):
```

```python
    # try different values on n of collisions to improve your
        collision checker
    collision_points_on_segment = 10 #the number of samples on each
        segment
    prime_number = 2
    for i in range(1, collision_points_on_segment):
        point = haltonPointInSegment(i, prime_number,
                                     (nearest_node.x, nearest_node.y),
                                     (new_point[0], new_point[1]))
        for i in range(0,len(obstacleList)):
            val = checkPointInsidePolygon(point, obstacleList[i][::-1])
            if val == 1:
                return 1 # collision
    return 0


if __name__ == '__main__':
    # check halton point
    point = haltonPointInSegment(2, 2, (0, 0), (1, 1))
    # check collision function
    obstacle = [[3, 3], [4, 3], [4, 4], [3, 4]]
    q = (3.5, 3.5) #whether q in the obstacle
    val = isPointInConvexPolygon(q, obstacle)
```

```python
# A53307224
# main.py
import matplotlib.pyplot as plt
import numpy as np
from auxiliary_functions import inCollision
from auxiliary_functions import dist
from auxiliary_functions import checkPointInsidePolygon
# node class which has the x and y position along with the parent
# node index
class Node():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None

# RRT class implementation
class RRT():
    def __init__(self, start, goal, obstacle_list):
        self.start = Node(start[0], start[1]) # start node for the RRT
        self.goal = Node(goal[0], goal[1]) # goal node for the RRT
        self.obstacle_list = obstacle_list # list of obstacles
        self.node_list = [] # list of nodes added while creating the
            RRT
```

```
# You need to complete this planning part of the code, note that it
    has a
# random sampling part that outputs a random_point, but still you
    need
# to try to connect this random_point to the tree, by finding the
# closest node, and verify that the segment connecting to the tree
# is collision free. To check for collisions use an imported
    auxiliary
# function.
   def planning(self, animation=True):
      self.node_list = [self.start]
      while self.goal.parent is None: #the initial value is None
         # Random Sampling
         # We are choosing the goal node with 0.1 probability, this
         # gives a bias to RRT to search towards the goal. Increasing
         # the bias may take longer time to converge to goal if the
         # path has lot of obstacles in its path. Tune this
             parameter to
         # see the differences
         if np.random.rand() > 0.1:
            random_point = np.random.sample((2, 1))*10.1
         else:
            random_point = np.asarray([self.goal.x, self.goal.y],
                                       dtype=float)
         for i in range(0,len(self.obstacle_list)):
            val =
                checkPointInsidePolygon([random_point[0],random_point[1]],self.obstac
            if val == 1:
               continue
         # creating a node from the point
         new_node = Node(random_point[0], random_point[1])
         # set the parent as index no of the node in the
             self.node_list
         index = self.getNearestNode(random_point)
         dist_nearest =
             self.calcDistNodeToPoint(self.node_list[index],random_point)
         if(dist_nearest < 0.25):
            new_node.parent = index # setting the parent of new node
                to start node
         else:
            new_sample_node =
                Node(0.75*self.node_list[index].x+0.25*new_node.x,0.75*self.node_list
            for i in range(0,len(self.obstacle_list)):
               val =
                   checkPointInsidePolygon([new_sample_node.x,new_sample_node.y],self
               if val == 1:
                  continue
            new_sample_node.parent = index
            new_node = new_sample_node
```

```
        val =
           inCollision(self.node_list[index],[new_node.x,new_node.y],self.obstacle_l
        if val == 1: #collision
           continue
        if new_node.x==self.goal.x and new_node.y==self.goal.y:
           self.goal = new_node
           self.node_list.append(self.goal)
        else:
           self.node_list.append(new_node) # storing the nodes in a
              list
        if animation:
           self.drawGraph(random_point)
     # once the goal node has a parent this means the tree has a
        path
     # to the start node.
     # Edit below this line at your own risk. This will take care
        of creating a
     # path from goal to start.
     path = [[self.goal.x, self.goal.y]]
     prev_node_index = len(self.node_list) - 1
     while self.node_list[prev_node_index].parent is not None:
        node = self.node_list[prev_node_index]
        path.append([node.x, node.y])
        prev_node_index = node.parent
     path.append([self.start.x, self.start.y])

     return path

  # input: node as defined by node class
  # input: point defined as a list (x,y)
  # output: distance between the node and point
  def calcDistNodeToPoint(self, node, point):
     di = dist([node.x,node.y],point)
     return di

  # input: random_point which you sampled as (x,y)
  # output: index of the node in self.node_list
  def getNearestNode(self, random_point):
     min_dist = 10000000
     iindex = 0
     index = 0
     for i in self.node_list:
        di = self.calcDistNodeToPoint(i,random_point)
        if di < min_dist:
           index = iindex
           min_dist = di
        iindex += 1
     return index
```

```python
    # edit this function at your own risk
    def drawGraph(self, random_point=None):
        plt.clf()
        # draw random point
        if random_point is not None:
            plt.plot(random_point[0], random_point[1], "^k")
        # draw the tree
        for node in self.node_list:
            if node.parent is not None:
                plt.plot([node.x, self.node_list[node.parent].x], [
                         node.y, self.node_list[node.parent].y], "-g")
        # draw the obstacle
        for obstacle in self.obstacle_list:
            obstacle_draw = plt.Polygon(obstacle, fc="b")
            plt.gca().add_patch(obstacle_draw)
        # draw the start and goal points
        plt.plot(self.start.x, self.start.y, "xr")
        plt.plot(self.goal.x, self.goal.y, "xr")
        plt.axis([0, 10, 0, 10])
        plt.grid(True)
        plt.pause(0.01)

if __name__ == '__main__':
    # Define obstacle polygon in the counter clockwise direction
    obstacle_list = [[[3, 3], [4, 3], [4, 4], [3, 4]],
                     [[8, 8], [7, 6], [9, 9]],
                     [[1, 6], [3, 8], [4, 10]],
                     [[3.7,4.4], [4.7, 5.2], [4.2, 6.8], [2.3, 6.1]]]
    # Set Initial parameters
    rrt = RRT(start=[2, 2], goal=[5, 5], obstacle_list=obstacle_list)
    show_animation = True
    path = rrt.planning(animation=show_animation)
    # Draw final path
    if show_animation:
        rrt.drawGraph()
        plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
        plt.grid(True)
        plt.show()
```