Guidelines – Read Carefully! Please check each problem for problem-specific instructions. You are provided a zip file containing a single folder named xyz007 with the following folders and files:

xyz007
xyz007/RRT
xyz007/RRT/rrt.py
xyz007/group.txt

You should first rename the folder name xyz007 with your NETID. If you are working in a team, the folder should be named with both group members' NETIDs, e.g., NETID01NETID02. For assurance, also put the NETID(s) in the file group.txt. When you are ready to submit, remove any extra files (e.g., some python interpreter will create .pyc files) that are not required and zip the entire folder. The zip file should also be named with your NETID as NETID.zip (or NETID01NETID02.zip for groups with two members).

You are required to write your program adhering to Python 2.7 standards. In particular, DO NOT use Python 3.x. Beside the default libraries supplied in the standard Python distribution, you may use ONLY numpy and matplotlib libraries.

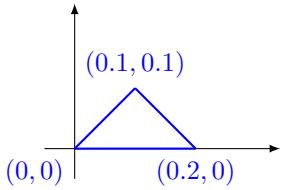
As mentioned in class, you may form groups of up to two people. Only a single student needs to submit per group.

Problem 1 [100 + 100 points]. Implementation of a complete RRT planner for 2D. For grading purposes, this assignment counts as two separate MP assignments. The first three tasks will count as one assignment and the last two tasks will count as a second assignment. This is relevant when it comes to the computation of the "drop one lowest score" option. For example, if you get 50 out of the first three tasks and 60 out of the last two tasks, and your other HW/MP are all higher than 50, then the 50 will become 75 after applying the "drop one lowest score" option. Another scenario: if you did well on other assignments and did well on first three tasks and get 100 + 0 overall for MP3, then the 0 will become 75 after applying the "drop one lowest score" option.

Robot. You will be implementing the Rapidly-exploring Random Tree (RRT) algorithm for a *convex polygonal translating robot*. That is, the robot will NOT rotate. The robot, in its *local coordinate system*, will be defined as a list of clockwise arranged points, e.g., the three points

$$(0, 0), (0.1, 0.1), (0.2, 0)$$

would define the triangle as shown below.

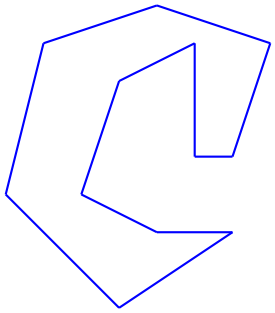


The robot's configuration in the (global) workspace will be specified using a single point $q = (q_x, q_y)$, which is the origin of the robot's local frame. That is, the robot will be occupying the space bounded by the points

$$(q_x, q_y), (q_x + 0.1, q_y + 0.1), (q_x + 0.2, q_y)$$

Note that the robot is not fixed to be the triangle provided above. It may be any convex polygon with up to six vertices.

Environment/workspace. The robot resides in a 10×10 region with the lower left corner being $(0, 0)$ and the upper right corner being $(10, 10)$. There are multiple polygonal obstacles in the region. The obstacles may be non-convex, e.g., you may have obstacles that look like the following non-convex polygon

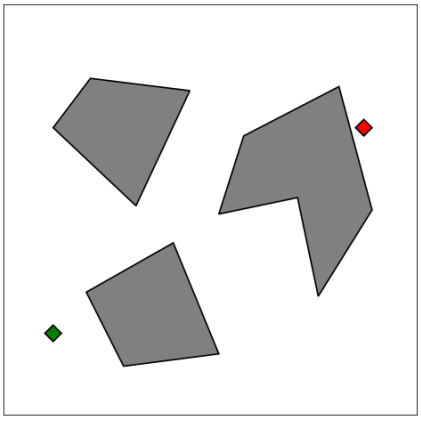


The environment will contain 0+ such obstacles. You may assume that two obstacles will not intersect with each other and no obstacle will intersect the boundary of the environment.

The problem. The robot and obstacles are specified in an input file (e.g., `robot_env_01.txt`). Each line in the file represents a list of clockwise arranged x-y coordinates that defines a polygon.The first line of the file defines the robot and the rest of the lines are the obstacles, one obstacle per line. We have already parsed these for you in `rrt.py`. A full problem is then specified by such a file plus the start and goal configuration of the robot, which is passed on the command line as

```
python rrt.py robot_env_01.txt 1.0 2.0 8.5 7
```

This will visualize the problem as well as printing out problem on the command line. You will of course be given different inputs to test your program. The visualization should give you the figure on the right.



The tasks. You are given a skeleton file `rrt.py`, as already mentioned above, to work with. As said, the current code takes in as arguments a file describing the robot and the polygonal obstacles, and coordinates for the start and goal configurations. You are to implement the RRT algorithm following the steps listed below.

1. **Generate an RRT without obstacles [50 points]**. You are to implement the function

   ```
   growSimpleRRT(points)
   ```

   The argument `points` is a map of point ids to tuples of xy-coordinates that you will use as samples, i.e., instead of generating your own random samples, you should use these as the random samples. A provided `points` may look like

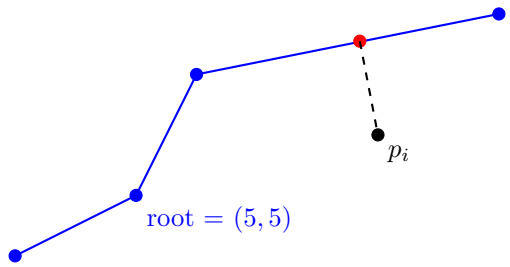
   ```
   {1: (5.2,6.7), 2: (9.2,2.3), ...}
   ```

   In the function `growSimpleRRT(points)` , you are to realize the following:

   (a) Grow an RRT with the root $(5, 5)$ for `len(points)` iterations. The RRT should be stored as an adjacency list (map) that looks like

   ```
   {1: [2, 5], 2: [3, 4], ...}
   ```


(b) For each iteration, for the given point i from `len(points)`, which has coordinates $p_i = (x_i, y_i) =$ `points`$[i]$, you are to find the nearest point on the existing tree to p_i. This means that sometimes the nearest point can be somewhere on an edge of the existing RRT instead of at an end point, which then requires the addition of a new point to the RRT. When this happens, the existing RRT data structure needs to be updated to reflex this. The map `points` should also be updated to add the new point. As an example, in the figure given below, the closest point to p_i on the existing (blue) RRT tree is the (red) point. This (red) point, in addition to p_i must be added to the RRT data structure as well as the map `points`.



(c) Return the updated map of `points` and also the RRT that you just build. Note that you can return multiple items in python simply with

```
return a, b
```

Note that you do not need to consider the size of robot in this task. That is, you may treat the robot as a point robot for this task.

2. **Implement a basic BFS/DFS search routine [25 points]**. You are to implement a basic breadth first or depth first search function (your choice) with the signature

   ```
   basicSearch(tree, start, goal)
   ```

   in which `tree` is an adjacency list (map). `start` and `goal` are the indices of two points in the tree. Your function should return a list of points that is a path from `start` to `goal`, e.g.,

   ```
   [start, ..., goal]
   ```

3. **Visualization [25 points]**. You are to implement a visualization function

   ```
   displayRRTandPath(points, tree, path, robotStart, robotGoal, polygons)
   ```

   that draws the tree that you have just grown and a path on the tree. This can be a straightforward modification from the visualization code from MP2. It should display the RRT that you've just grown. In fact, this task is probably best done before you proceed with the first task as it helps you check whether you have done the first task correctly. Please draw the tree in black and the path in orange. You should check whether the path is empty. Your function should also draw the problem if the arguments `robotStart`, `robotGoal`, `polygons` are not set to `None`.

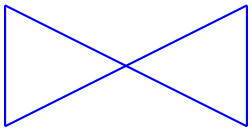
4. **Implement basic collision detection [30 + 20 points]**. You are to implement a basic collision checking routine that detects whether the robot, at a given global coordinate, would collide with the boundary and obstacles in the environment. The function should have the following signature

   ```
   isCollisionFree(robot, point, obstacles)
   ```


in which `robot` is the polygon (as a list of tuples) for the robot, `point` is where the robot is at, and `obstacles` is a list of obstacle polygons. All polygons are given as clockwise oriented points, e.g., a polygon will look like

`[(`x_1`, `y_1`), (`x_2`, `y_2`), ...]`

You may assume the boundary of the workspace as given in the beginning, i.e., the workspace is a 10×10 area. You will be given 60% of credits on this task if your code works with convex obstacles and full credits if your code works with non-convex polygonal obstacles. You do not need to consider self-intersecting obstacles like the one below. Your function should return either `True` or `False`.



5. **Solve a real problem using RRT [50 points]**. You are now to put everything together and implement a simplified RRT algorithm with the signature

   `RRT(robot, obstacles, startPoint, goalPoint)`

   Here, `robot` and `obstacles` are the same as in the function `isCollisionFree`. `startPoint` and `goalPoint` have the format (as a tuple)

   `(`x`, `y`)`

   You will now need to sample random points yourself for growing the RRT. By *simplified RRT*, we mean that you do not need to grow a partial edge; if a new edge that is generated causes collision with an obstacle, you may simply discard that new point and edge, and try another random point. Your function should display the RRT that you created and draw the path that you found, using your function `displayRRTandPath`. Your function should also return the map of points, the adjacency list (map), and the path, i.e.,

   ```
   return points, tree, path
   ```

Note that you should not modify the main function in `rrt.py`. Of course, you may modify it during your implementation and testing but it should not be changed to contain any additional logic that you implemented. You may add helper functions as needed inside `rrt.py`.

Good luck!

