CS 4610/5335: Robotic Science and Systems (Spring 2023) Robert Platt
Northeastern University

HW 3: Motion Planning, Sample Based

Please remember the following policies:

- Submissions should be made electronically via the Canvas. Please ensure that your solutions for both the written and/or programming parts are present and zipped into a single file.
- Solutions may be handwritten or typeset. For the former, please ensure handwriting is legible.
- You are welcome to discuss the programming questions (but *not* the written questions) with other students in the class. However, you must understand and write all code yourself. Also, you must list all students (if any) with whom you discussed your solutions to the programming questions.

Sampling-based Motion Planning

Download the starter code from Piazza (hw3.zip). In this file, you will find:

- hw3_motion.m: Main function. Call hw3_motion(<questionNum>) with an appropriate question number (0–5) to run the code for that question. Do not modify this file! (Feel free to actually make changes, but check that your code runs with an unmodified copy of hw3_motion.m.)
- M0.m – M5.m: Code stubs that you will have to fill in for the respective questions.
- check_collision.m, check_edge.m: Helper functions to perform collision checking.

In this part of the assignment, you will implement two sampling-based motion planning algorithms, the probabilistic roadmap (PRM) and the rapidly exploring random tree (RRT). A 4-DOF 2-link arm has been defined, together with a spherical obstacle. The cylinders depicting the arm's links should not collide with the obstacle, but the cylinders located at the joints are just markers visualizing the axes of rotation and do not contribute to potential collisions.

M0. **2 points.** Familiarize yourself with the provided code in hw3_motion.m and the robot that has been defined. Use the code in M0.m to visualize the robot in various configurations. The code in hw3_motion.m that calls M0 also checks whether the configuration is within joint limits, and whether the configuration is in collision.

Look at the code in check_collision.m and check_edge.m. Explain what each function is doing to check for collisions, for individual configurations and for configuration-space line segments respectively. Identify and describe two potential issues in the collision-checking algorithm (but do not try to fix them).

hw3_motion.m (main driver code)

- The provided code in this file is using the function M0.m to set the initial configuration of the robot, and then it is using the inverse kinematics function to find the joint angles that result in a given end-effector pose. The code then checks whether the resulting joint angles are within joint limits and whether the robot is in collision with any obstacles.

check_collision.m

- This helper function is used to check whether the robot is in collision with any of the obstacles in the environment for a single configuration of the robot. It does this by computing the position of each link of the robot using forward kinematics and then checking if any of these positions are inside any of the obstacles.

check_edge.m

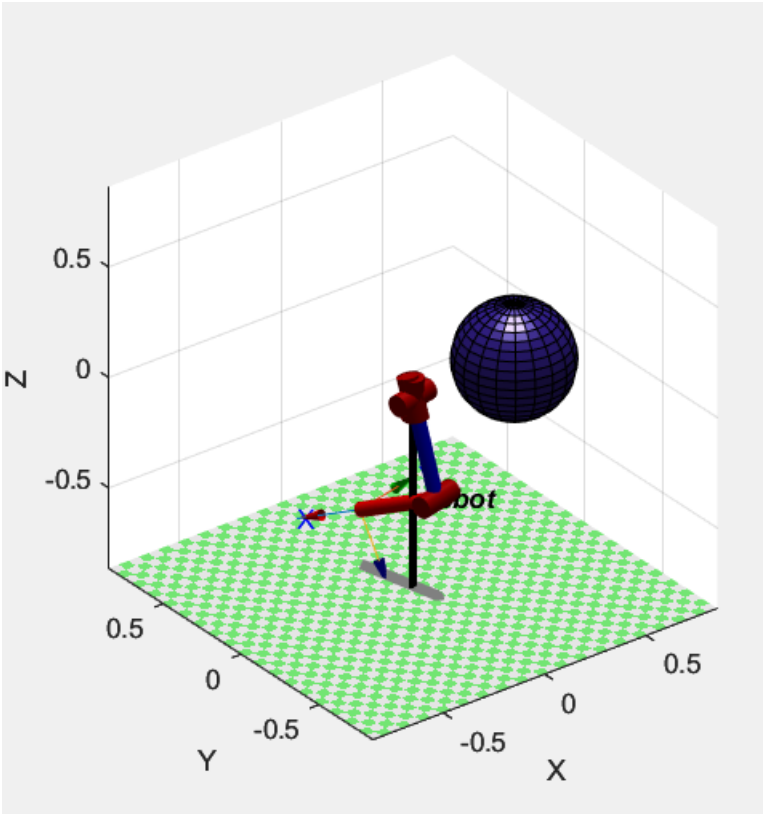
- This helper function is used to check for collisions between two configurations of the robot. It does this by generating a sequence of configurations between the two given configurations and then checking for collisions for each of these configurations using the check_collision function.

Two potential issues in the collision-checking algorithm:

1. Discretization error: The check_edge function discretizes the configuration space by generating a sequence of configurations between the two given configurations. If the discretization is too coarse, it is possible to miss collisions that occur between the generated configurations. Conversely, if the discretization is too fine, the algorithm will be slower.

2. Overestimation of collision: The check_collision function checks if any link of the robot is inside any of the obstacles. This can lead to overestimation of collision, as the robot might only be close to the obstacle without actually colliding with it. This can lead to unnecessary collisions being reported, and can also slow down the algorithm.

M1. **1 points.** Despite the issues you may have identified in M0, we will proceed with the provided collision checking functions. Given the provided joint limits, generate uniformly distributed samples from configuration space. The configurations should be within the joint limits, but they can be in collision.

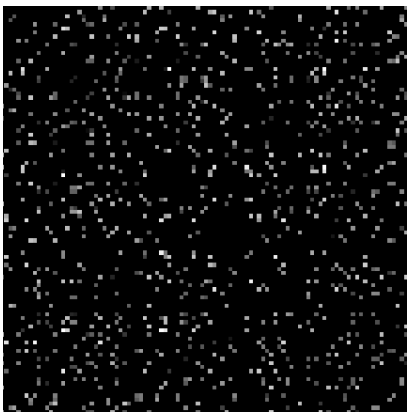



```
>> hw3_motion(1);
Number of samples: 100
Number within bounds: 100
Number in collision: 23
```

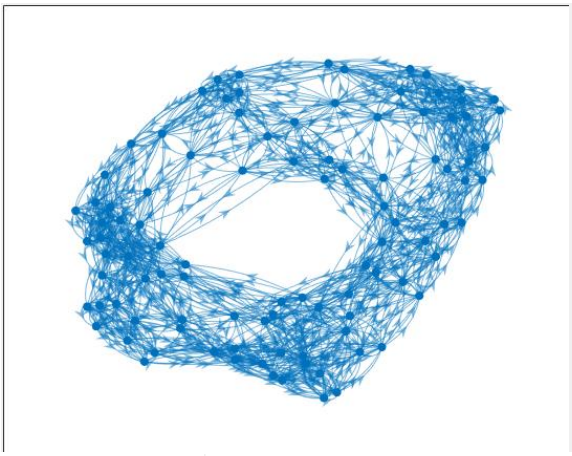
M2. **2 points.** Implement the PRM algorithm to construct a roadmap for this environment. Your graph should contain num samples collision-free configurations within joint limits. For each vertex, consider the nearest num neighbors neighbors, and add an edge if the segment between them is collision-free. Edges are specified using the output adjacency matrix, where adjacency(i,j) is the distance between vertices v_i and v_j if an edge exists between them, and is 0 otherwise. Note that the adjacency matrix should be symmetric (edge between v_i and v_j implies edge between v_j and v_i), and that each vertex may have ≤num neighbors neighbors (if some edges are in collision) or ≥num_neighbors neighbors (v_j may be closest to v_i, but there may be other vertices closer to v_j).

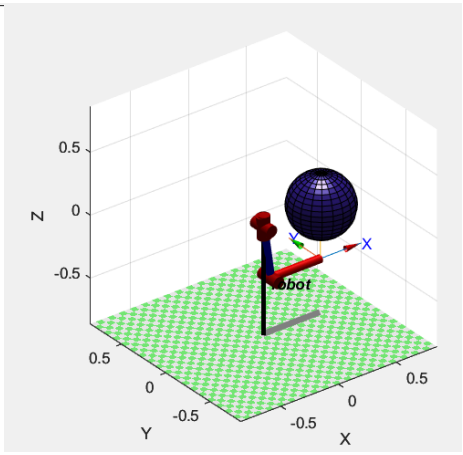
Hint: The next question relies on the output of M2.m, which may take a while to compute. To avoid recomputing it in future questions, we have provided functionality to save it in your MATLAB workspace, and pass it in on future calls:

[samples, adjacency] = hw3 motion(2); hw3
motion(3, samples, adjacency);



M3. **2 points.** Use the roadmap to find a collision-free path from the start configuration to the goal configuration. You will need to appropriate points to get "on" and "off" the roadmap from the start and goal respectively. *Useful functions*: shortestpath

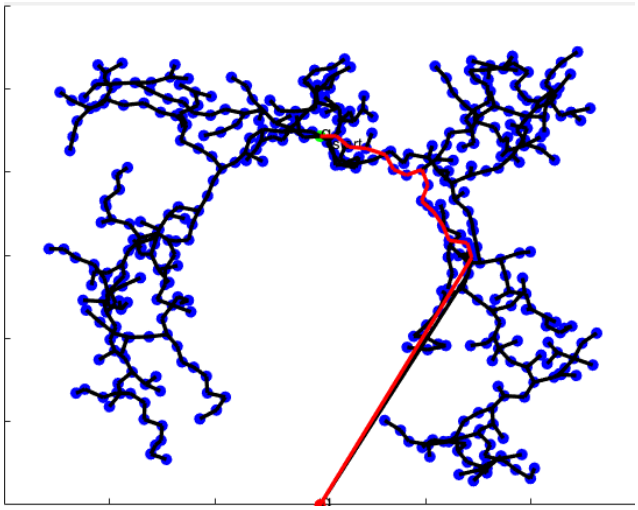





```
>> hw3_motion(3, samples, adjacency);
Path found with 4 intermediate waypoints:
         0   -0.7854         0   -0.7854
   -0.6605   -1.1565         0   -1.1874
   -0.7959   -1.4360         0   -1.5021
   -0.9953   -1.7358         0   -2.2625
   -0.7844   -2.1742         0   -2.4856
         0   -3.0000         0   -3.0000
```

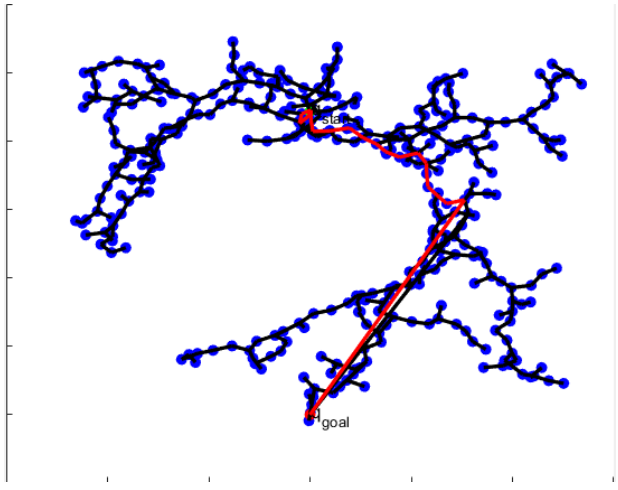
M4. **2 points.** Implement the RRT algorithm to find a collision-free path from the start configuration to the goal configuration. Choose appropriate hyperparameter values for the step size and frequency of sampling q_{goal}. Remember to traverse the constructed tree to recover the actual path.

ONLY ALPHA:



```
>> hw3_motion(4);
Path found with 15 intermediate waypoints:
         0   -0.7854         0   -0.7854
         0   -0.7854         0   -0.7854
    0.0778   -0.7841         0   -0.8482
    0.1355   -0.8490         0   -0.8979
    0.2181   -0.8589         0   -0.8424
    0.3107   -0.8964         0   -0.8465
    0.3482   -0.9849         0   -0.8742
    0.4105   -1.0191         0   -0.9446
    0.4832   -0.9858         0   -1.0045
    0.5115   -1.0784         0   -1.0295
    0.4817   -1.1739         0   -1.0289
    0.5381   -1.2391         0   -0.9783
    0.5816   -1.3195         0   -0.9378
    0.6125   -1.4112         0   -0.9630
    0.6984   -1.4260         0   -1.0121
    0.7196   -1.5148         0   -0.9713
         0   -3.0000         0   -3.0000
```

WITH BETA:




```
>> hw3_motion(4);
Path found with 20 intermediate waypoints:
         0   -0.7854         0   -0.7854
         0   -0.7854         0   -0.7854
   -0.0047   -0.8159         0   -0.8805
   -0.0513   -0.8713         0   -0.9495
   -0.0410   -0.8021         0   -1.0210
    0.0049   -0.7751         0   -1.1057
    0.0047   -0.8512         0   -1.1705
    0.0144   -0.9303         0   -1.2309
    0.0975   -0.9276         0   -1.2864
    0.1946   -0.9050         0   -1.2782
    0.2609   -0.9764         0   -1.2555
    0.3272   -1.0221         0   -1.3148
    0.3738   -1.0789         0   -1.2470
    0.4424   -1.1190         0   -1.1863
    0.5336   -1.0885         0   -1.1587
    0.5746   -1.1638         0   -1.1072
    0.5760   -1.2387         0   -1.0410
    0.5824   -1.3352         0   -1.0666
    0.6274   -1.3974         0   -1.1306
    0.6754   -1.4577         0   -1.0669
    0.7632   -1.4310         0   -1.0272
         0   -3.0000         0   -3.0000
```

M5. **2 points.** The path found by RRT likely has many unnecessary waypoints and motion segments in it. Smooth the path returned by the RRT by removing unnecessary waypoints. One way to accomplish this is to check non-consecutive waypoints in the path to see if they can be connected by a collision-free edge.

Before Smoothing:

```
>> hw3_motion(5);
Path found with 18 intermediate waypoints:
         0   -0.7854         0   -0.7854
         0   -0.7854         0   -0.7854
    0.0502   -0.8407         0   -0.8520
    0.1004   -0.8285         0   -0.9376
    0.0970   -0.9010         0   -1.0064
    0.1717   -0.8742         0   -1.0673
    0.1480   -0.9258         0   -1.1496
    0.2220   -0.9448         0   -1.2141
    0.2827   -0.9864         0   -1.2818
    0.3542   -1.0486         0   -1.3136
    0.3883   -1.0149         0   -1.4013
    0.4757   -1.0625         0   -1.3918
    0.4679   -1.1501         0   -1.4395
    0.4921   -1.2239         0   -1.5025
    0.5162   -1.2508         0   -1.5957
    0.5623   -1.3393         0   -1.6030
    0.6140   -1.3805         0   -1.5279
    0.6666   -1.4655         0   -1.5283
    0.7427   -1.5205         0   -1.4938
         0   -3.0000         0   -3.0000
```

After Smoothing:

```
Smoothed path found with 2 intermediate waypoints:
         0   -0.7854         0   -0.7854
    0.4679   -1.1501         0   -1.4395
    0.7427   -1.5205         0   -1.4938
         0   -3.0000         0   -3.0000
```

M6. **0 points.** If M4 and M5 have been implemented correctly, this question should require no further implementation. Watch your algorithm work on a more challenging motion planning problem. Can you make it even harder

```
>> hw3_motion(6);
Path found with 26 intermediate waypoints:
         0   -0.7854         0   -0.7854
         0   -0.7854         0   -0.7854
    0.0102   -0.7689         0   -0.8835
    0.0099   -0.8414         0   -0.9523
    0.1046   -0.8670         0   -0.9715
    0.1610   -0.9031         0   -1.0458
    0.2464   -0.9417         0   -1.0807
    0.2273   -0.9520         0   -1.1783
    0.2830   -0.9724         0   -1.2588
    0.3145   -0.9757         0   -1.3536
    0.2489   -1.0113         0   -1.4203
    0.2763   -0.9731         0   -1.5085
    0.3239   -1.0427         0   -1.5623
    0.3702   -1.0940         0   -1.6346
    0.4423   -1.0584         0   -1.6940
    0.5200   -1.0916         0   -1.6405
    0.5294   -1.1763         0   -1.6928
    0.5876   -1.1502         0   -1.7697
    0.5672   -1.2068         0   -1.8496
    0.6389   -1.2111         0   -1.9192
    0.6181   -1.2736         0   -1.9945
    0.6235   -1.3695         0   -2.0224
    0.6112   -1.4500         0   -2.0804
    0.5978   -1.5398         0   -2.1224
    0.5941   -1.6382         0   -2.1052
    0.5850   -1.7269         0   -2.1505
    0.5843   -1.7909         0   -2.0737
         0   -3.0000         0   -3.0000

Smoothed path found with 2 intermediate waypoints:
         0   -0.7854         0   -0.7854
    0.6181   -1.2736         0   -1.9945
    0.5843   -1.7909         0   -2.0737
         0   -3.0000         0   -3.0000

>>
```