Lab 4: Planning

6.1 Methods

To start implement the functional code, we first introduced the RRTGraph as a class. The graph is utilized to help us track backward from node to node, so that we could obtain the clear path which connects the starting point and the end point. When initializing an instance of the Class, we first define the initial condition: the starting and end point, a flag if the goal succeeds, and the map. We also defined the data structure to store the lists of all the configurations and parents produced during the reiteration and assigned the initial value to the lists. Then we added the boundary conditions, which is the upper and lower limits to generate new nodes, as well as the obstacles in the maps.

The algorithm of main loop is very similar to the Prelab. First, we randomly sample nodes in the configuration space, which is implemented by the **expand ()** function in the code. Here we have two separate approaches for sampling the nodes. The first method we implement is to sample with steps, which is to sample each point within a certain range of the last configuration. For example, in the code, we planned to sample in next configuration within 10% of the entire range (upper limit – lower limit), and in this way we could increase the successful rate of connecting the newly generated nodes with the previous configurations. However, we found this method unnecessary because the obstacle in the map is not so complicated that we must make such tradeoffs between successful rate and efficiency, so we implemented the second sampling method, which is the normal sampling method within the upper and lower limit, and it improves the efficiency of finding the path to the target. Then we added this new node into the configuration list using **addNode (n, q) function**, which takes in the index of the node and the configuration of the node, so that we could obtain its notation and conduct distance calculation later. To find the nearest node to the new node, we planed the compute the total distance between each joint in a particular configuration to the new configuration because we think other way of defining the 'nearest' distance, for example, considering the normalization, is biased since the length of each link is different, which should be retained in calculation, and normalization erase the difference. We also considered calculating the distance directly using configuration. But since the calculation in this way was done in configuration space, it makes little sense to me compared with calculating in word space. After we find the nearest node to the new node with the **nearest (n)** function, which takes in the index of the added node, now we tried to connect these two nodes and figure out if there's obstacle between them. In the **connect (n1, n2)** function, which takes in two adjacent nodes, first we obtained the configuration of the two node we try to connect, then we input the configurations of the nearest node we found and the newly generated node into the given **isRobotCollided (q1, q2).** If there is obstacle between the nodes and it returned FALSE, we remove the new node we generated from the configuration list as well as the node list and repeat the process starting from the adding a new node until the return becomes TRUE; if there is no obstacle detected, we would like to connect these two nodes by adding an edge between them with the **addEdge (parent, child)** function, which keeps track of the parent node of each generated node to help us traverse a valid path from the entire graph. After connecting the nodes into our 'tree', we also want to test the connecting status between the new node and our end node, the goal, by implementing the collision check function again. If the function returns TRUE, that means our task is completed and the path that connect the stating node and the goal is found. For further analysis, we added the function **pathToGoal ()**, which returns nodes in a valid path, and **getPathConfig ()**, which returns the configurations in the path based on the nodes in the valid path, to print out each configuration we walked through on the path that connects the start and goal. Since the 'tree' has multiple branches which is hard to track, here we applied the **parent ()** in the class setup, which could find the parent node of our current node. It is also the nearest among the nodes ahead of the current node based on our definition. Starting from the last node and tracking backward, what we did is to put a node into the function to find its parent node, put the parent node into the function again and obtained the new parent node, etc. By doing so, we can easily track from end to start and find the path that successfully connects the start and the goal. If it detects that there are obstacles between the current node we stepped on and the goal, the **goalFlag** will return FALSE and we will keep adding new nodes, transforming it into configuration and adding it to the list of q(n), finding the nearest node, trying to connect, detecting obstacles, adding edges. We will repeat this method until the obstacle detection function tells there is nothing between the new-added node and our goal, then our task is completed. Besides the functions for controlling the RRT algorithm, two other functions in the class are **visualize (n1, n2)**, which takes in two nodes and visualize using matplotlib, and **addVisualizedObstacles (ax)**, which takes in the axis from visualize function and plot the obstacles. These two functions help the development and debug of all other functions.

6.2 Evaluation

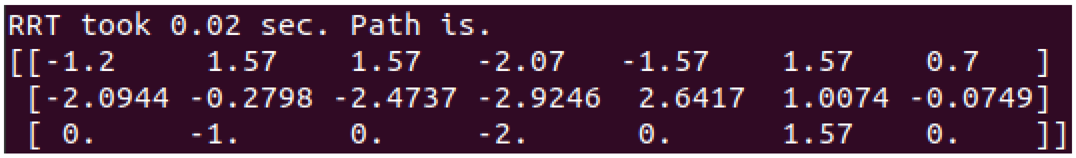
Test 1

Map: 1

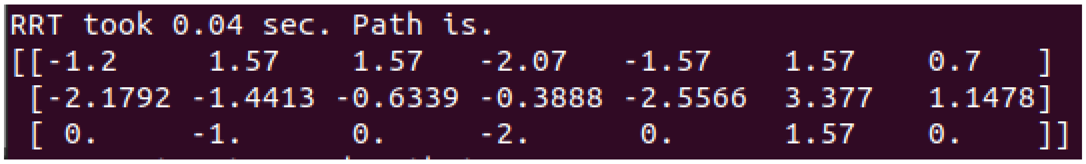
Starting Configuration: [-1.2, 1.57, 1.57, -2.07, -1.57, 1.57, 0.7]

Goal Configuration: [0, -1, 0, -2, 0, 1.57, 0]

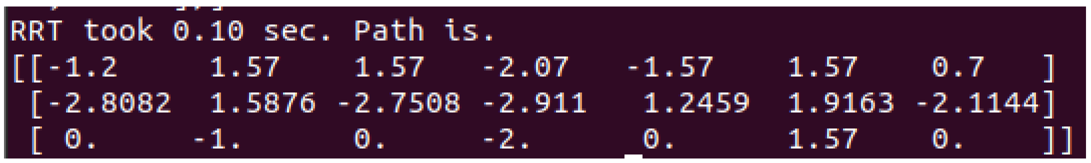
RRT**:**



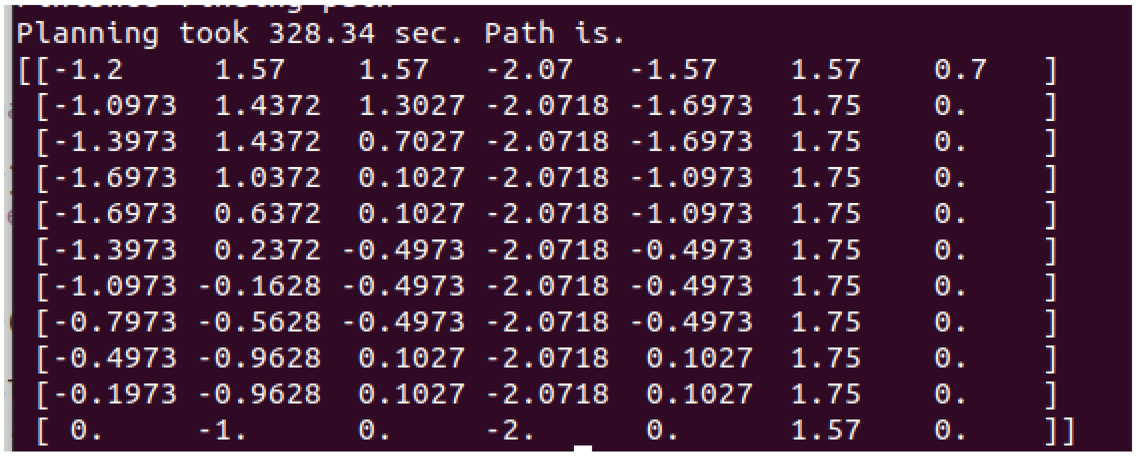
```
RRT took 0.02 sec. Path is.
[[-1.2      1.57     1.57    -2.07    -1.57     1.57     0.7    ]
 [-2.0944 -0.2798 -2.4737 -2.9246  2.6417  1.0074 -0.0749]
 [ 0.     -1.      0.     -2.      0.      1.57    0.    ]]
```



```
RRT took 0.04 sec. Path is.
[[-1.2      1.57     1.57    -2.07    -1.57     1.57     0.7    ]
 [-2.1792 -1.4413 -0.6339 -0.3888 -2.5566  3.377   1.1478]
 [ 0.     -1.      0.     -2.      0.      1.57    0.    ]]
```


```
RRT took 0.10 sec. Path is.
[[-1.2      1.57     1.57    -2.07    -1.57     1.57     0.7    ]
 [-2.8082  1.5876 -2.7508 -2.911    1.2459   1.9163 -2.1144]
 [ 0.     -1.       0.     -2.       0.       1.57     0.     ]]
```

A*:


```
Planning took 328.34 sec. Path is.
[[-1.2      1.57     1.57    -2.07    -1.57     1.57     0.7    ]
 [-1.0973  1.4372  1.3027 -2.0718 -1.6973  1.75     0.     ]
 [-1.3973  1.4372  0.7027 -2.0718 -1.6973  1.75     0.     ]
 [-1.6973  1.0372  0.1027 -2.0718 -1.0973  1.75     0.     ]
 [-1.6973  0.6372  0.1027 -2.0718 -1.0973  1.75     0.     ]
 [-1.3973  0.2372 -0.4973 -2.0718 -0.4973  1.75     0.     ]
 [-1.0973 -0.1628 -0.4973 -2.0718 -0.4973  1.75     0.     ]
 [-0.7973 -0.5628 -0.4973 -2.0718 -0.4973  1.75     0.     ]
 [-0.4973 -0.9628  0.1027 -2.0718  0.1027  1.75     0.     ]
 [-0.1973 -0.9628  0.1027 -2.0718  0.1027  1.75     0.     ]
 [ 0.     -1.       0.     -2.       0.       1.57     0.     ]]
```

RRT Attempts	Time Usage (seconds)	Success in Gazebo (Yes or No)
1	0.02	Yes
2	0.04	Yes
3	0.1	Yes
Average	0.053	100%
A* Attempt	328.34	Yes

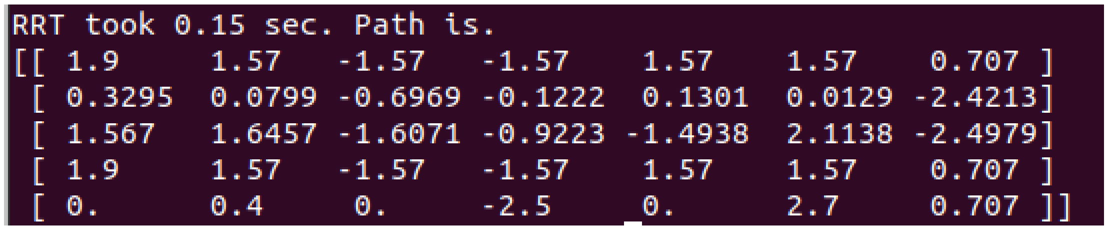
Test 2

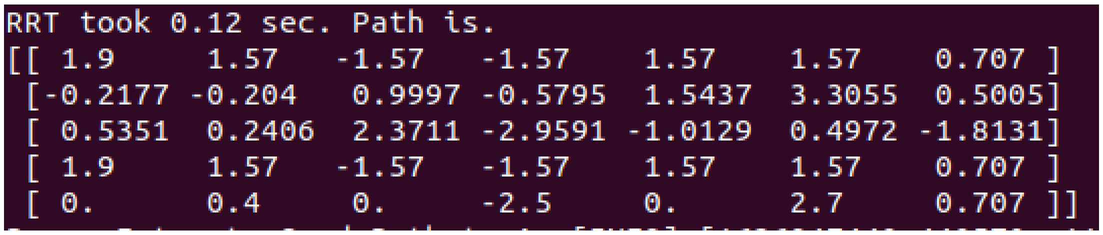
Map: 2

Starting Configuration: [1.9, 1.57, -1.57, -1.57, 1.57, 1.57, 0.707]

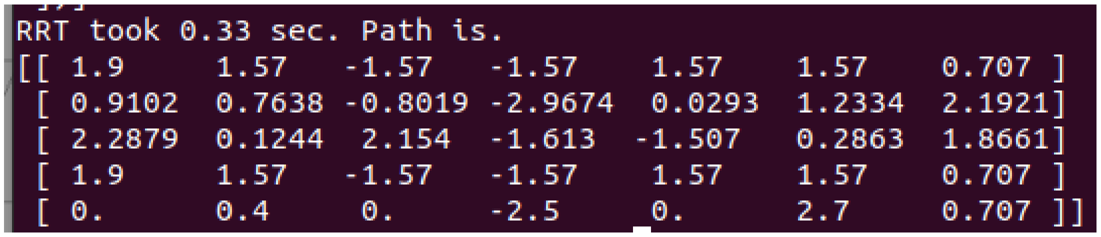
Goal Configuration: [0, 0.4, 0, -2.5, 0, 2.7, 0.707]

RRT:


```
RRT took 0.15 sec. Path is.
[[ 1.9      1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [ 0.3295  0.0799 -0.6969 -0.1222  0.1301  0.0129 -2.4213]
 [ 1.567   1.6457 -1.6071 -0.9223 -1.4938  2.1138 -2.4979]
 [ 1.9     1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [ 0.      0.4      0.      -2.5      0.       2.7      0.707 ]]
```



```
RRT took 0.12 sec. Path is.
[[ 1.9      1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [-0.2177 -0.204    0.9997 -0.5795   1.5437   3.3055   0.5005]
 [ 0.5351  0.2406   2.3711 -2.9591  -1.0129   0.4972  -1.8131]
 [ 1.9      1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [ 0.       0.4      0.      -2.5      0.       2.7      0.707 ]]
```



```
RRT took 0.33 sec. Path is.
[[ 1.9      1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [ 0.9102  0.7638  -0.8019  -2.9674   0.0293   1.2334   2.1921]
 [ 2.2879  0.1244   2.154   -1.613   -1.507    0.2863   1.8661]
 [ 1.9      1.57    -1.57    -1.57     1.57     1.57     0.707 ]
 [ 0.       0.4      0.      -2.5      0.       2.7      0.707 ]]
```

A*: Time usage > 10 min, no result.

RRT Attempts	Time Usage (seconds)	Success in Gazebo (Yes or No)
1	0.15	Yes
2	0.12	Yes
3	0.33	Yes
Average	0.2	100%
A* Attempt	>10 min	No (could be Yes, takes too long)

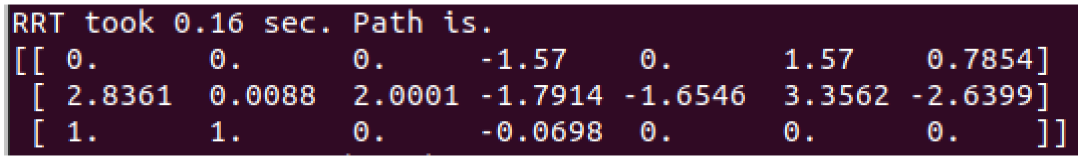
Test 3

Map: 2

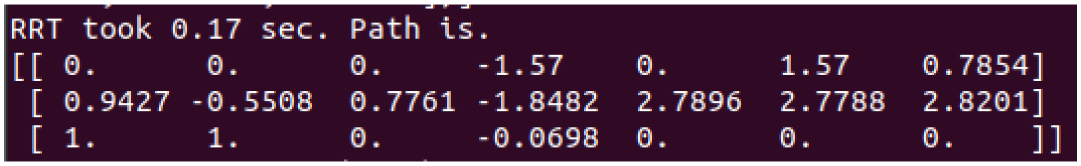
Starting Configuration: [0, 0, 0, -1.57, 0, 1.57, 0.7854]

Goal Configuration: [1, 1, 0, -0.0698, 0, 0, 0]

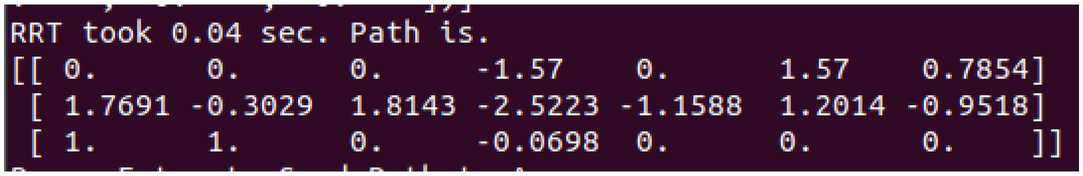
RRT:



```
RRT took 0.16 sec. Path is.
[[ 0.       0.       0.      -1.57     0.       1.57     0.7854]
 [ 2.8361  0.0088   2.0001  -1.7914  -1.6546   3.3562  -2.6399]
 [ 1.       1.       0.      -0.0698   0.       0.       0.    ]]
```



```
RRT took 0.17 sec. Path is.
[[ 0.       0.       0.      -1.57     0.       1.57     0.7854]
 [ 0.9427 -0.5508   0.7761  -1.8482   2.7896   2.7788   2.8201]
 [ 1.       1.       0.      -0.0698   0.       0.       0.    ]]
```



```
RRT took 0.04 sec. Path is.
[[ 0.       0.       0.      -1.57     0.       1.57     0.7854]
 [ 1.7691 -0.3029   1.8143  -2.5223  -1.1588   1.2014  -0.9518]
 [ 1.       1.       0.      -0.0698   0.       0.       0.    ]]
```

A*: Time usage > 10 min, no result.

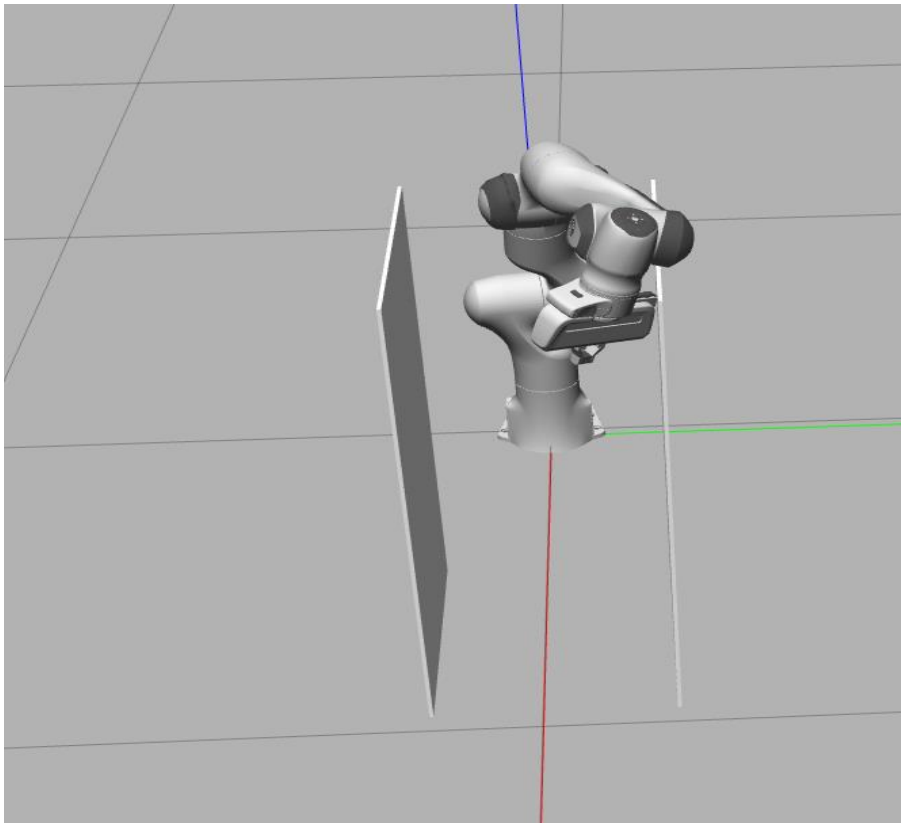


Figure 6.2.1 Final State of Test 3 (Collision)

RRT Attempts	Time Usage (seconds)	Success in Gazebo (Yes or No)
1	0.16	No
2	0.17	No
3	0.04	No
Average	0.123	100%
A* Attempt	>10 min	No (could be Yes, takes too long)

To test out the RRT algorithm we implemented, we designed and conducted three sets of different tests and gave each test three attempts. The purpose of such design is to collect enough data to calculate the average time of path-finding process in each test as well as the successful rate among all attempts.

Before conducting the self-designed test, we passed the test in map 1 and 2 with the starting and goal configurations provided already. When it comes to the tests designed by ourselves, we switched the position of the given start and goal configuration, now that the initial goal becomes our starting point and the initial start becomes the goal configuration, both in map 1 and 2. The purpose of such design is that we have had an idea of what is the path the robot passed to the final destination in the initial test and how long it takes to find such path, we wonder if such path will look similar if we switch the start/goal position and if such change will affect the successful rate in the Gazebo Simulation. We designed such test in both map 1 and map 2 because the environment in map 1 is relatively simple compared to map 2 (1 piece of obstacle vs 2 pieces), so we would like to see what the result is going to be in both simple and more complicated cases.

In test 3, we chose the start and goal configuration on purpose so that the robot would start within the obstacle and end up with a configuration which is completely outside of the obstacles. This task is the most complicated one among the three tests and we want to see how exactly self-collision and the link volume will affect the result. The testing result verifies what we thought: though the time of pathfinding takes very little time and succeeds for all three attempts, the robot cannot ideally find its way out of the two pieces of obstacles and collides with the obstacle near the goal configuration in the Gazebo simulation. Figure 6.2.1 shows such collision, and it is also the final configuration of the robot on its way to the goal. One potential explanation we could come up with is we need to take more of self-collision and link volume into consideration so that we could set a more restricted range for the arm to pass the collision check. In this way we could bring out the arm out and way from the obstacles more safely.

We have 3 tests and 9 attempts in total, and the rate of finding a connection path successfully is 100%, which means in every single case the code we implemented could find a potential path from the start to the goal; the rate of arm finds its way to the targeted configuration in Gazebo simulation is 66.7%. The case of failure happens only in test 3, which is a more complicated test than the previous two.

The average time usage of finding a path in each of three tests are: 0.053s, 0.2s, 0.123s. The total average time usage for all the attempts in all three cases is 0.1255s.

From the different attempts in the same test, we could easily tell that every time when the path planner finds a path, the routes will highly possibly not be the same over the multiple runs. We did notice there is some configurations right before success that looks similar in different attempts. It makes sense to us since that we are using random sampling instead of sampling with a certain step size as we mentioned in the Method, the probability of finding two exact same paths over multiple runs is rather low. However, when it comes near the goal between obstacles, the range of node sampling is narrowed down a lot and we will have a higher chance to obtain some similar configurations near the goal.

According to our tests, the time usage goes up when the map becomes more complicated and that is why our attempts in test 2 and 3 usually takes much longer to complete; besides that, the relative position of the start and goal configurations about the location of obstacles will also influence the time we use since it determines the successful sampling rate. At the same time, when the map environment becomes more complicated, the rate of path planner succeeds decreases, for instance, all the attempts failed in test 3 when we tried to create a hard scenario for the path finder. In other words, we must take more into consideration when implement our code so that our success rate could increase in more complicated cases. Regardless of the differences in environments and variations, the path found by the planner will hardly be the same according to all our attempts.

We also implemented A* algorithm provided to compare with the RRT planner we implement earlier. In general, RRT planner has a much higher working efficiency than the A* in the condition that path can be found for both approaches. When the map contains complicated environment, or the relative location of the start and goal configuration about the obstacle is hard for the arm to find its way out, A* approach will take a much longer time to complete. In test 3 we terminated its process after 10 min period if no path was found. It might finally find the path, but just too slow to be included.

6.3 Analysis

According to the success rate of the tests, our planner worked well in the map with simple environment and fewer obstacles, for example test 1. Our planner also has a decent performance when the relative location of the start and the goal configurations about the obstacles is not so complicated. For example, in test 2, even the environment becomes more complicated than test 1, our planner could still find the path and pass the test without collision. However, our planner does not do a good job when the path it followed is too close to the obstacles. It is because that we did not take enough consideration of the link volume and basically treated the link as a 'line'. When it comes near the obstacle, even the planner shows an accessible path, we might still have collisions between the links and the obstacles in the environment. This is a significant point that our planner is currently bad at and deserves more considerations in our code implementation.

Based on our expectations, as long as the planner finds a potential way to connects the start and goal configuration, the arm should be able to generate such access in the real simulation. Test 3 proves that such assumption is wrong. Even if a path has been found, the arm still ended up colliding with the obstacles on its way. Although there's multiple ways to avoid reduce such difference between the expected performance and the experimental performance, for example, create and implement link volume in the code of checking collisions, we could only make our planner as close to the real scenario as possible, but we still need to make tradeoffs between the processing efficiency and the general performance.

If we have more time in the lab, we might implement the RRT* algorithm against our RRT planner and make comparisons between them. We could also add some aspects we currently ignored, for example, to solve the problem of self-collision and take link volume into account, which will increase the planner's performance when handling more complicated environments. Also, expanding the RRT algorithm to heuristic RRT by adding guided node sampling may also help the convergence speed.

Experiment wise, the most obvious phenomenal is that A* takes much more time than RRT to find the path which connects the start and the goal in the same map. In addition, the path that A* found will normally have much more intermediate nodes than that for RRT. Algorithm wise, A* divides the entire space into voxels and calculate the distances and check for collisions between voxels. Conducting such operations is very computationally intensive, and A* approach requires such operations much more frequently than the RRT approach. It is one of the potential reasons that A* has a relatively bad performance in our test.