Lab 3: Trajectory Planning for the Lynx

MEAM 520, University of Pennsylvania

October 23rd, 2020

Venkata Gurrala, Sheil Sarda

Table of Contents

Introduction

The objective of this lab was to plan collision free trajectories through a cluttered environment using the lynx robot. Artificially made obstacles within a boundary space were put into a txt file as a map for the python code to interpret before determining a feasible trajectory from a specified start and end position (q). The input q is a list of angles from theta 1 through 5.

Method

The planning was done within the configuration space rather than the workspace. The configuration space can be split into obstacle space and free space which can be considered when planning for all the joints. Additionally, it enables us to consider the joint limits easier without much back and forth kinematic calculations. The space that we are working with can be discretized to represent the volume of the robot and obstacles. From the map input given which represents the location and size of obstacles in addition to the boundaries of the discretized space, the obstacle collision space can be modelled. Since the joint motors are moving at a fixed angular velocity, there would be less benefit to having cubic or quintic trajectories compared to linear trajectories. Additionally, solving for cubic and quintic trajectories can increase computation time significantly. The trade off due to this linear interpolation method is that there would be higher jerk associated.

Obstacle detection:

If the robot reaches the obstacle configuration space a collision is detected. However, this assumes that the robot has zero thickness in space which it obviously does not. Therefore, it becomes necessary to model the lynx robot with volume to ensure that collisions with obstacles can be clearly identified. A simple approach that can be used to make this happen is by assuming the 0-thickness links with the robot while artificially modifying the dimensions of each block. Figure 1 shows a 2D representation of the obstacle and the link right before collision. Both the obstacle and the link are modelled with volume in the discretized space. The exact point at which the collision occurs can be similarly modelled as shown in figure 1 by inflating the size of the obstacle while still representing the link to have 0 thickness. This will remove the necessity to model the entire robot's volume while getting relatively accurate yet conservative collision detections. Additionally, it allows us to use the obstacle detection algorithm that was already developed which assumed a 0-thickness line path. The inflation of the obstacle was decided based on a conservative estimate of the largest dimension that the end effector had. The opening of the end effector was limited to 30 mm so a conservative overestimate was chosen (40mm). This was added to each side of the obstacle to inflate its size. This will help the RRT planner to plan paths that help create a larger clearance between the robot and the obstacles.

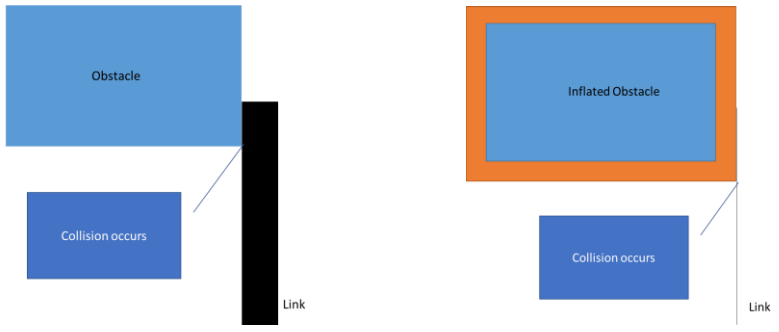


Figure 1. Visualizing the simplification of obstacle detection algorithm

The robot can self-collide with itself at its base. To prevent this collision, 2 blocks were created around the origin to represent the volume of the base. Creating a single block that is in the origin would have the algorithm assume that the robot has already collided so two blocks were created that were close to the origin of the base. Figure 2 shows the top view of the actual base in green. The two rectangular shapes represent the obstacles that were created in the space to give the picture of the robot to prevent collision with itself.

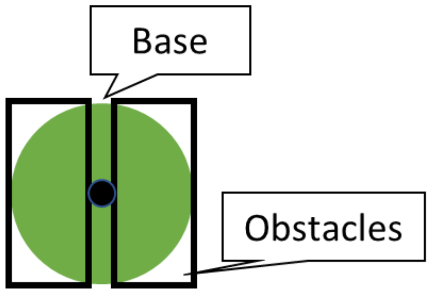


Figure 2. Simplified obstacle to prevent self-collision

Planning strategy

An RRT planner was implemented in this lab which in principle, samples random points till a viable path is found from a specified start and end position. After loading the given map where the obstacles are known, the RRT planner can first use the specified start and end goal to determine if a straight path already exists or if the start and end positions are the same. While checking for a straight path between the start and end point, the algorithm can check for obstacles' position and size in the discretized space. If it detects that an obstacle exists, this path will be rejected, and the main algorithm will start.

A variable named "current position" is initialized to be start position. The implemented RRT algorithm mainly consists of a while loop within which each joint position is randomly sampled within their respective joint limits. Using the new random configuration, forward kinematics can be performed to determine the end effector's randomly sampled position in the discretized space. A line segment can be drawn from the current position to this randomly sampled position. If this line segment intersects an obstacle, a collision can be assumed. Similarly, all the current joint positions and the randomly sampled joint positions can be calculated in the discretized space. The path from their current positions and new position can be checked for collisions. If any collisions are predicted, the new randomly sampled position is rejected, and a new point is sampled. The maximum iteration limit was set to 5,000 as a limit to the computation time.

If a feasible position is found, it is added to an array which stores waypoints. When a new point is added to the waypoints array, it is stored under the variable, current position. A direct path is then checked for collisions between this position and the goal position. If the path doesn't exist, the cycle of finding a randomly sampled point continues till the end goal is reached.

A common issue we experienced with Gazebo was that many positions were not given enough time for the RRT planner to reach. While some positions can be reached relatively quickly, some positions took much longer before the next waypoint was given. This would often cause collisions. To solve this issue, the next waypoint was not given till each joint reached 0.01 rad of the position specified. This was successful in giving enough time for the robot to navigate to the desired position.

Post processing

The disadvantage of choosing randomly sampled points is that there could be redundant paths can be taken to reach the goal position. To eliminate this, we implement a post-processing strategy of pruning the RRT using a randomized algorithm with the following steps:

1) Generate two indices a and b which point to different poses in the path, maintaining the invariant that a < b
2) Check for any collisions between point a and b in the workspace using forward kinematics
3) If no collisions, join eliminate the poses contained in the subarray [a + 1, a + 2, … , b – 1], placing a and b adjacent to each other in the post-processed path
4) Repeat the procedure from (1) at every iteration

We also scaled the number of iterations with the number of points in the path to achieve a relatively uniform post-processing path length. After the post-processing is complete, we also validate the shortened path maintains the following invariants:

- First pose in the post-processed path is the start pose, and final pose is the goal pose
- The pose at index i in the post-processed array has a collision-free path to the pose at index i + 1

Suggested next steps to improve the post-processing methodology:

1) Connect the start pose to a pose which has the highest index in the array (j) and has a collision-free path from the start to this intermediate pose
2) In the modified path after (1), connect the goal pose to the pose with the lowest index (i) in the array and has a collision-free path from the intermediate to the goal pose
3) Eliminate the subarrays
    a. [start + 1, start + 2, …, j – 1]
    b. [i + 1, i + 2, …, goal - 1]

Evaluation

Map 1:

Goal: (1,0,1.05,0,0,0), (-1.2,0.3,0.8,-0.3,0,0)

From the start position, two goal positions were chosen which placed the end effector below the table. The robot had taken very different paths with the same start and goal positions given. The robot collided during some of its routes as the goal point was deliberately set close to the obstacle. For most runs, the robot found anywhere between 3-7-way points before getting to the goal position and took less than 30 seconds. When the robot was set to a position under the obstacle, it had a harder time navigating out of the obstacle and sometimes collided while on its way to the zero position. The A* code took longer to compute a path but eventually got to the same position and had less collisions than the RRT algorithm. When re-running the code to get the robot back into the zero configuration, the robot always got stuck below the table and had collided.

Map 2:

Goal: (1,0,1.05,0,0,0), (-1.2,0.3,0.8,-0.3,0,0)

Using the same start and end positions in this map, the robot was able to get out of the enclosure on some runs. The same goal positions were chosen since they successfully tested the robot's ability to navigate out of the encloser on both sides by varying theta 1. This environment had similar behaviors observed from map 1. The wrist tended to collide within the tight obstacle space and could not get out. However, it still had similar number of waypoints needed to get to the goal position. Its path was still random, but it was relatively limited compared to the first map since the arm was enclosed by obstacles in two directions which limited the number of feasible paths. The A* algorithm took significantly longer to compute a path for, and a few collisions had occurred on the way to the final goal position.

Map 3:

Goal: (1.4,0,1,0,0,0)

From the 0 configuration, the goals chosen tested the robots ability to move in a tight encloser in 3 directions. In this map, the RRT planner had determined a feasible path faster than the A* code and had fewer random and erratic waypoints. This was mainly due to the volume covered by the obstacles itself which minimized the number of feasible waypoints. The A* code at the same goal position given had also collided with the column as shown in figure 5. The RRT planner had taken different paths in several instances whereas the A* code chose one path. There were more collision that had occurred with both planners compared to the other maps. But the RRT planner had more collisions than the A* algorithm did.

Map 4:

Goal: (0,0,-pi/2,0,0,0)

The goal of this environment was to use joint 3 to navigate the arm out of the enclosure. In this map, the RRT performed much quicker while often taking more direct routes compared to the A* algorithm. The A* algorithm took over 5 minutes to find a path whereas the RRT algorithm took less than 20 seconds to find and navigate a path. The RRT algorithm took slightly different paths each time it was run but the paths were very similar. In a couple iterations, the RRT planner led the robot to collide with the obstacles.

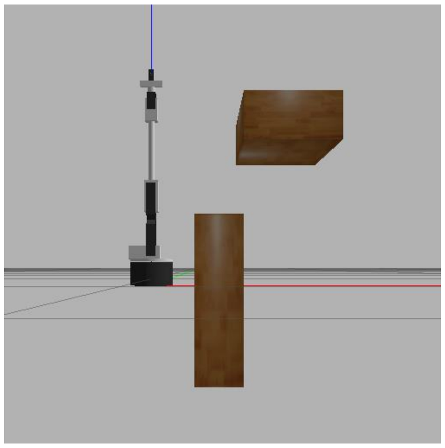


Figure 3. Correct position of given goal point in map 4

Map 5:

Goal: (0.5,-0.5,-pi/2,-1,0,0)

Map 5 was created to test the RRT planner to use joints 3-e from one tight space to another. Previous maps predominantly involved the rotation of joints 1 and 2 to move the arm away from enclosed obstacles. Some goal positions that were tested had sampled over 2000 points in space before finding a path with 500 feasible waypoints. This was much more than most other maps which had anywhere between 3-7 waypoints to get to the goal position. For this reason, the RRT planner took over 1 minute to compute a path before it was post-processed. The A* planner took even longer at 6-8 minutes at certain goal positions. Eventually, both planners were able to compute a relatively short path and their paths were always different. Sometimes the robot had collided while getting out of the zero configuration.

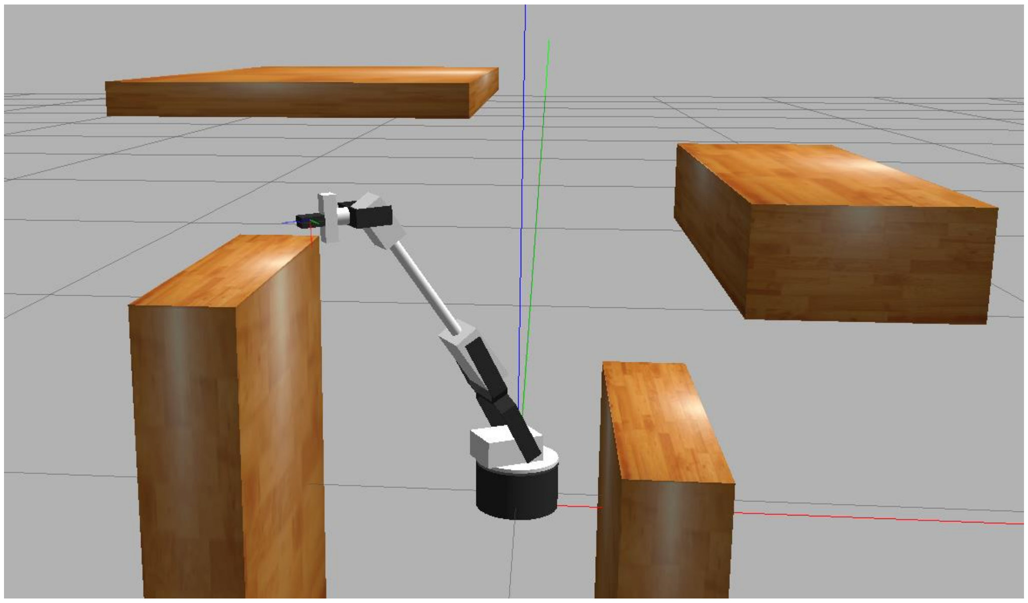


Figure 4. Correct positioning of robot to goal position in map 5

Analysis

Common issues:

Figures 5 and 6 show the collision that occurred with the A* code while returning back to the 0 configuration. This was quite common for many of the maps even though each of the obstacles were inflated to prevent these forms of collisions. The collisions were also present with the RRT planner. The collisions mostly occurred with the wrist and end effectors. Figure 6 was also stuck on the way to get to the zero configuration. The collision of the wrist could be explained by its relative size compared to the links. However, inflating the obstacles to account for path planning would not be feasible since it would not be realistic.

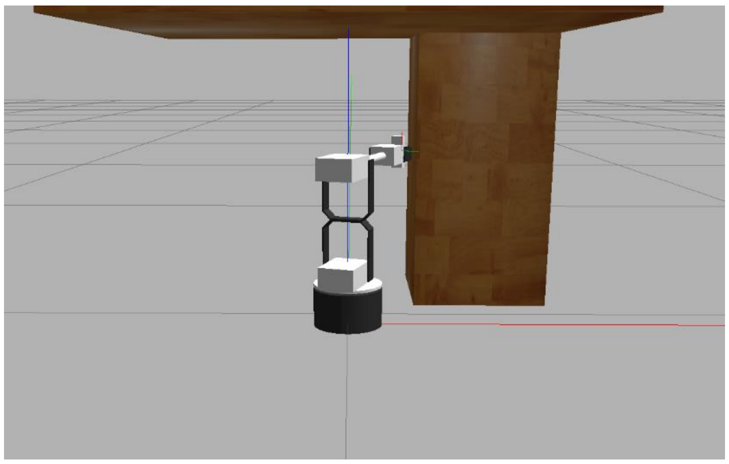
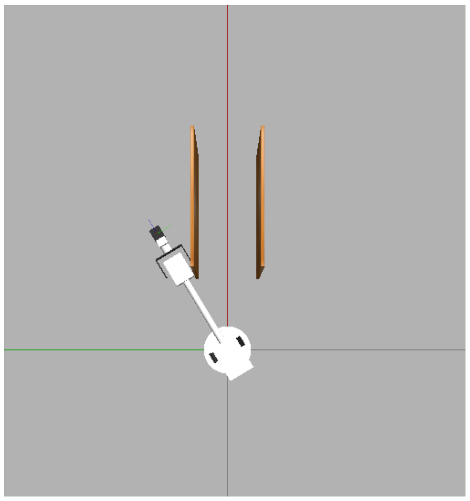



Figure 5. Instances where wrist is colliding with obstacles in maps 3 and 2 respectively

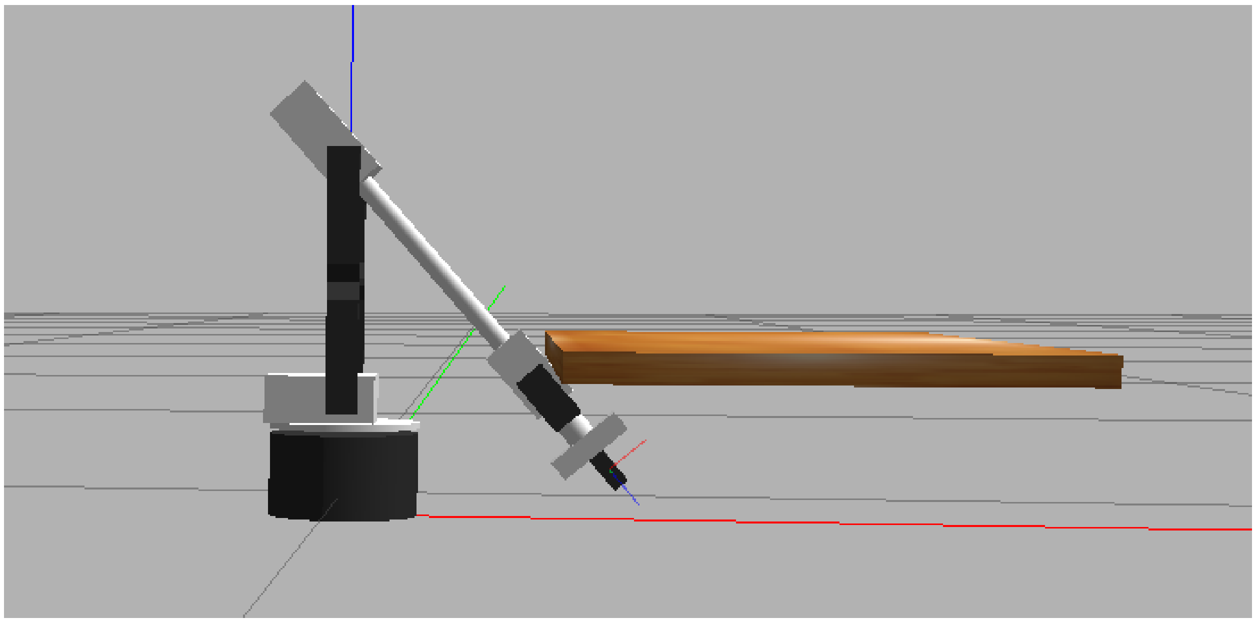


Figure 6. Robot stuck under obstacle while going to the 0 configuration

A problem that can be identified with the current path is that joints will never take a straight-line path from the current position to its final position. So, while a path is feasible in a straight line, the joints will often have a different trajectory and our planner does not detect collisions during its path. For this reason, if there was more time, the trajectory of the joints can be planned while considering the configuration space in more detail to prevent collisions between waypoints. Using a cubic or quintic trajectory will additionally help prevent jerk in certain configurations which could make the robot collide in tight spaces. In enclosed spaced, varying the speed can potentially avoid collisions.

The A* implementation had taken more time to determine a path compared to the RRT planner. In certain, compact environments, the RRT planner had determined a feasible path and had successfully reached the path all before the A* planner could even determine the path itself. The post processing used along with the RRT planner helped the robot compute a path as fast or faster than the A* method for most configurations. However, the A* had a much lower rate of collision compared to the RRT planner.

If we had more time with the lab, we would try to model the volume of the robot more accurately to determine paths that were feasible without collisions. One of the ways this could be done was to deliberately overestimate the length of link between the rotating wrist and end effector. This would reduce collisions with the end effector specifically. The large wrist could also have been modelled more accurately by understanding its dimensions with respect to the joint position of joint 4. For example, given the dimensions of the wrist and their positions with respect to the joint position, several line segments can represent the boundary between the free configuration and wrist 4's volume. The series of line segments can be used with the obstacle collision detection to find paths that avoid the wrist from coming too close to the obstacle.

Implementation details

We split up the functionality of our codebase into the following 6 functions:

Brief description of each of the functions

- `boundaryCollision(pose, boundary)`

  Takes as input a robot pose (q-array) and the boundary list, and detects if any of the joint positions, computed using forward kinematics, exceed the boundary constraints.

  This function is used to evaluate the feasibility of randomly sampled points in the configuration space.

- `obstacleCollision(start, goal, obstacles)`

  Takes as input the start and the goal robot pose (q-array), as well as the list of obstacles, and detects if any of the joint positions collide with any obstacle.

  This function is used to evaluate the feasibility of randomly sampled points in the configuration space, and in the pre- and post-processing steps to evaluate if obstacle-free paths exist between any points in the paths list.

- `graphTrajectory(path)`

  Takes as input a path of q-arrays and creates a 3-D line plot for easier visualization. Built using matplotlib's pyplot library.

- `postProcessing(path, obstacles)`

  Takes as input the path (q-arrays) and prunes the RRT using a randomized algorithm described in the previous section.

- `RRT(map, start, goal)`

  Randomly samples points in configuration space to create an RRT, and invokes all the above functions to return a path from the start to the goal pose.

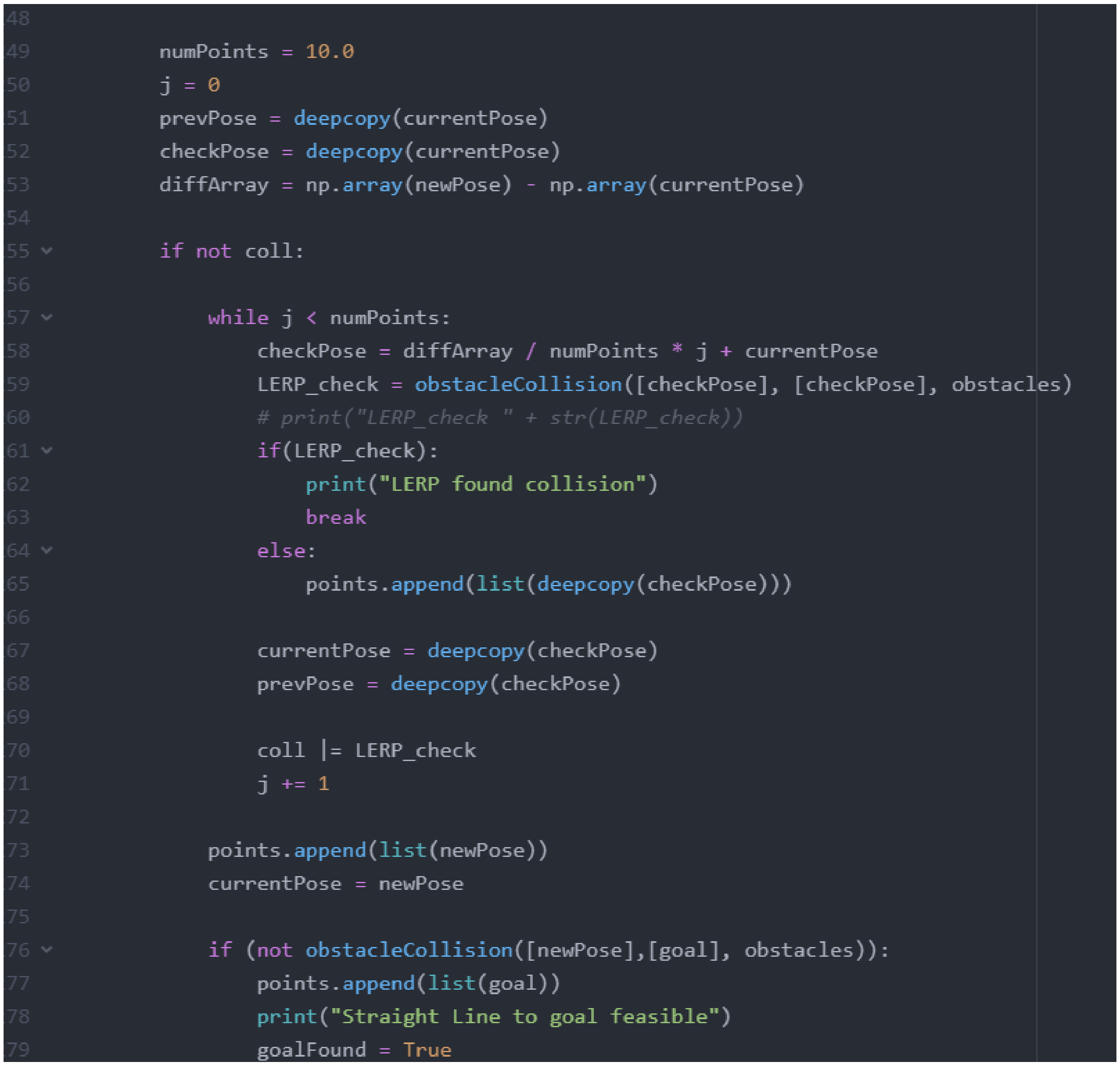
Appendix

Pseudocode for RRT function

Pseudocode for `rrt(map, start, goal)`

```
1. if goal pose is equal to start pose, OR if no obstacles between start and goal
    1. return [start, goal]
2. Set current position as start position
    1. While current pose not equal to goal and loop counter is less than max iterations
        1. Generate random pose using joint limits as the allowed range
        2. Check if line segment from current pose to randomly generated pose collides
            1. If collision detected, break out and start for loop again
        3. If the newly generated point has a collision-free path to the goal pose, append both the random pose and goal pose to the path
        4. Else, store the random pose in the path array, set the current pose to the random pose, and continue to the next iteration of the while loop.
3. Post-process the path to prune the generated RRT.
4. Return the path.
```

Code used to interpolate between waypoints in line 250:



```
48
49          numPoints = 10.0
50          j = 0
51          prevPose = deepcopy(currentPose)
52          checkPose = deepcopy(currentPose)
53          diffArray = np.array(newPose) - np.array(currentPose)
54
55          if not coll:
56
57              while j < numPoints:
58                  checkPose = diffArray / numPoints * j + currentPose
59                  LERP_check = obstacleCollision([checkPose], [checkPose], obstacles)
60                  # print("LERP_check " + str(LERP_check))
61                  if(LERP_check):
62                      print("LERP found collision")
63                      break
64                  else:
65                      points.append(list(deepcopy(checkPose)))
66
67                  currentPose = deepcopy(checkPose)
68                  prevPose = deepcopy(checkPose)
69
70                  coll |= LERP_check
71                  j += 1
72
73              points.append(list(newPose))
74              currentPose = newPose
75
76              if (not obstacleCollision([newPose],[goal], obstacles)):
77                  points.append(list(goal))
78                  print("Straight Line to goal feasible")
79                  goalFound = True
```