Pre-Lab

Our initial planning strategy was to plan in the configuration space, as this would allow us to freely select random points in the workspace to sample.

Next, we planned on mapping all joint limits, robot linkages, and obstacles into the configuration space by discretizing the workspace, calculating IK, and marking the discretized point in the configuration space as in collision or not. Furthermore, because the robot linkages continue to move as the robot itself moves, we would need to remap the configuration space for every instant in time.

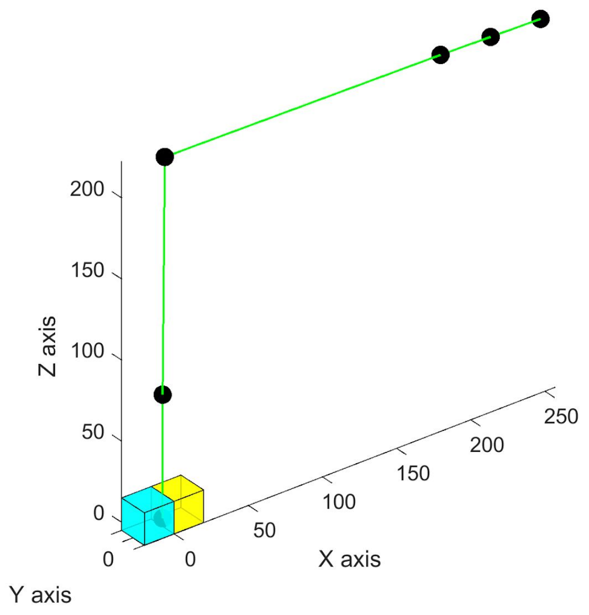
- Will you plan in the workspace or config space?
  - I will plan in the configuration space since. Although solutions using configuration space planning can lead to high joint velocities/accelerations/jerk around singularities, since we are not dealing with a real robot, this is not much of an issue.
  - Additionally, path planning in the configuration space is less computationally intensive since you do not need to solve for IK at every node in the path. Lastly, the joint space trajectory will be much smoother despite taking a longer route.
- How will you represent the space that you are planning in?
  - To start off, we will need to discretize each of the joint angles by a reasonably large number n
  - We will have a 6xn matrix, where each of the joint variables is a row and the joint discretizations is a column
  - Lastly, we will represent joint limits and obstacles in the configuration space as 0 and free configuration space as 1
- How will you check for collisions (with the end effector? w/ the rest of the robot?
  - For simplicity, we will not check collision of link n with link n+1. However, all other objects should be compared with each other
  - Additionally, since we can simplify the representation of each link in the robot arm as a rectangular prism, we can project each shape onto the plane perpendicular to each of the shapes' flat faces. we then further project these 2D shapes in the plane onto a line. if the shape projections for every combination are overlapping, then the objects are in collision.
  - After a trajectory has been computed, we can compute the forward kinematics and do a collision detection based using the projection method described above
- How will you account for the geometry (volume) of the robot?
  - We will represent each link as a separate rectangular prism in 3D space since modeling the complex object would be too computationally expensive for collision detection.
- How will you compute waypoints?

    - By using the RRT algorithm, we first check to see if the trivial solution exists. We check if the straight line path to the goal is collision-free.
    - If not, we then pick a random point in the configuration space that's a specified max distance (defined as 6-dimensional norm) from the start node.
    - We then use the trapezoidal method for interpolation to generate a path from the initial to current node. We can check if it has collided with an object or the robot arm or not using the method described above for every n nodes to save on computation.
    - If it is not colleded, then we connect it to the start node as a waypoint.
    - Next, we pick another random point in the configuration space. If it's within the max distance and also collision-free, we connect it to the closest node.
    - If it is not collided, then we repeat until we reach the goal pose.
- How will you interpolate b/w waypoints?
    - Because we're planning in the configuration space, we only need to solve inverse kinematics at the waypoints
    - For position, I prefer to use trapezoidal trajectory planning due to the simplicity and also easy to validate
    - For orientation, I will interpolate using quaternions using SLERP method

Methods

In retrospect, our planning strategy outlined in the Pre-Lab seemed to line up much more closely with A* than with RRT, as it involved discretizing and mapping the entire space instead of sampling the space. Additionally, it would prove to be extremely computationally expensive to remap the entire workspace at each step.

Instead, for self collisions, the observation was made that there were very few configurations which caused the robot to collide with itself. As a result, the ranges of these self-colliding configurations were separately analyzed in the ROS simulator and then hard-coded such that they fell outside the configuration space. This allowed us to save on calculating whether each discretization in our candidate waypoint had collided with itself. In practice this meant adding a box obstacle around the base of our robot because that was the only part of the robot that consistent collisions with the lynx. We do not show these obstacles in the plots below but they are there nevertheless, and were crucial in preventing the end effector from colliding with the base. We had them in the obstacle array as they went into RRT. A schematic diagram is shown below.



Our RRT path planning strategy stayed centered around planning in the configuration space. Since the robot (including the gripper) was a total of 6-DOF, our configuration space was 6-dimensional. Planning in the configuration space was chosen as it was more computationally expensive to solve for each joint angle using inverse kinematics if planning in the workspace. Additionally, an extra layer of complexity is added when deciding between the different IK solutions. This is because extra computation power would be required to evaluate each IK solution along each path discretization between waypoints. Granted, calculating forward kinematics equations is computationally expensive in its own right. However, because we were programming in MATLAB, we wanted to leverage its strength doing matrix calculations.

Regarding the implementation of the path planning trees, we decided to represent our nodes as an $n\,x\,6$ matrix, with n being the number of nodes and 6 being the number of dimensions in the configuration space. We decided to represent our edges as an singly-linked adjacency matrix, with $E_{ij} = 1$ for an edge connecting node i to node j. Conversely, $E_{ji} = -1$ for the same edge connecting node j to node i in reverse, which is useful when backtracking the path. When backtracking, we decided to start from the last node in the tree and go back to the root, as we are guaranteed completion in O(1) time since there can only be a single possible path from end to start. If we decided to track from start to end, we'd need to implement a version of BFS or DFS since we would have to search all branches in the worst case.

To interpolate between waypoints, we decided that a linear interpolation would suffice. As long as the interpolation steps were small enough, the mapped path in the workspace from the configuration space would linearize to a straight line.

Additionally, we decided to generate trees from the start node as well as the end node simultaneously, checking the validity of each candidate waypoint with both trees. Not only did this allow us to save on iterations (progress in both trees could be made during a single iteration), but it also guaranteed that any candidate waypoint that was valid for both trees meant that the algorithm had found a path. Lastly, it prevented issues with the algorithm venturing too far in the wrong direction.

In order to represent the robot in our algorithm, we decided to approximate the links of the robot as cylinders, each with radius equal to the width of the largest link. This gives a conservative collision detection boundary as the robot arm links with smaller radii may not require such a large collision boundary. Given additional time, it would be prudent to change the collision boundary such that it equals each link's respective radius.

The following pseudocode outlines our algorithm:

```
1. Initialize node matrices nodesStart and nodesEnd, and adjacency matrices
   edgesStart and edgesEnd for start and end poses, respectively
    a. Select a random point $P$ in the configuration space
    b. Using the configuration space norm in 6-dim, find the closest node to $P$ in the
       start tree (nodeStart) and closest node to $P$ in the end tree (nodeEnd)
    c. Compute the lines between the $P$ and nodeEnd, as well as $P$ and nodeStart
        i. Line between $P$ and nodeStart is called $l_{start}$
        ii. Line between $P$ and nodeEnd is called $l_{end}$
    d. Check if any of robot links collide with obstacle while traveling along $l_{start}$ and
       $l_{end}$
        i. Increase number of discretizations step along $l_{start}$ and $l_{end}$ until the
           step size is less than or equal to specified step size radiicheck
        ii. Discretize $l_{start}$ and $l_{end}$ based on step
        iii. Expand each map obstacle's dimension by the largest cylindrical radius
             $r_{max}$ out of all robot arm links.
        iv. Initialize collision check flag for start Scheck = true and collision
            check flag for end Echeck = true
        v. Walk along each step in $l_{start}$ and $l_{end}$
            a. Compute the joint positions for robot arm using CalculateFK
               function
            b. Construct line segments representing each link $link_i$ in robot arm
               from joint position $j_i$ and next joint position $j_{i+1}$
            c. For each $link_1$ to $link_5$, calculate whether $link_i$ has collided with
               an obstacle yet
                a. If $link_i$ passes through the expanded obstacle in all three
                   planes (x-y, y-z, and x-z)
```

    d. If $link_i$ has collided with an obstacle for either l_{start} or l_{end}, set `Scheck = false` or `Echeck = false`, respectively
  e. If `Scheck = true` but `Echeck = false`, then add `P` to only `nodesStart`. Additionally, add another row and column to `edgesStart`, with a `1` at (index of `nodeStart` in `nodesStart`, end) and a `-1` at (end, index of `nodeStart` in `nodesStart`).
  f. If 1e) is not true, then check if `Echeck = true` and `Scheck = false`, then add `P` to only `nodesEnd`. Additionally, add another row and column to `edgesEnd`, with a `1` at (index of `nodeEnd` in `nodesEnd`, end) and a `-1` at (end, index of `nodeEnd` in `nodesEnd`).
  g. If 1e) and 1f) are both not true, then check if both `Echeck` and `Scheck = true`. If this is the case, then we know that the start and end trees have connected at a common node. This means that we have found a valid path
2. To aggregate the path `Waypoints`, we backtrack through the both start and end adjacency matrices
  a. We first begin with start tree `edgeStart`, starting at the last row.
  b. We find the column index `Cindex` with value =`-1.` Retrieve the `node` from `nodeStart` with index = `Cindex` and prepend it to `Waypoints` since we need to reverse the node ordering.
  c. Once we have found `Cindex,` we navigate to the row with index = `Cindex`. For the new row, we repeat step 2b) until `Cindex = 1.` This means that we've reached the root of `nodesStart` and have finished aggregating the path for the start tree.
  d. We repeat steps 2a) - 2c) for the end tree `edgeEnd`. However, for 2b), we append to `Waypoints` since we do not need to reverse the node ordering. Additionally, we skip adding the common node to `Waypoints` as it has already been added when traversing the start tree.

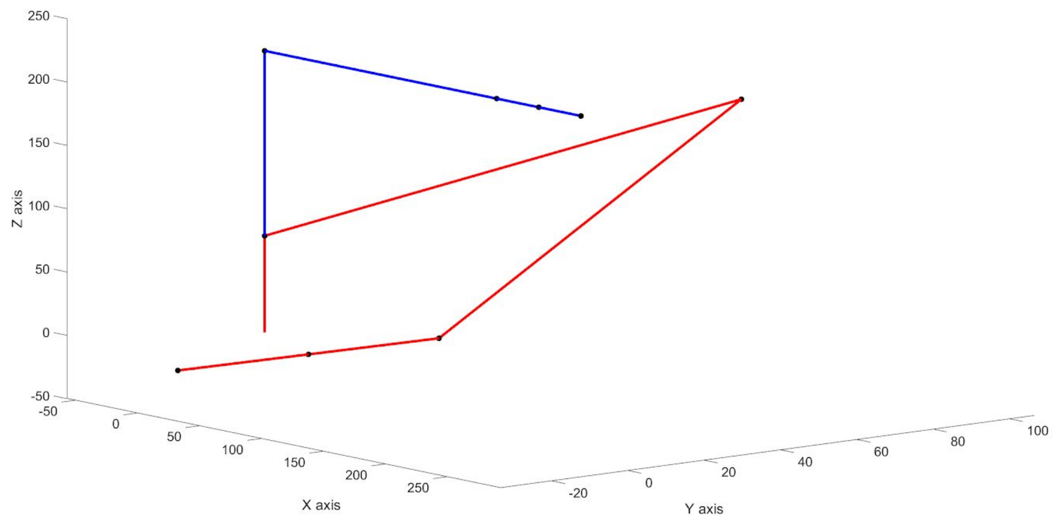
Evaluation

Our path planner works fairly often. There were few cases if any when our rrt algorithm was not able to find a path in the given maps.To be sure our rrt was working we developed various plotting systems for evaluating rrt. The first plotting system involved plotting the configuration of each valid node added to each tree. We set the blue nodes to be the configuration nodes in the start tree and the red nodes to be the configuration nodes of the goal tree. These are not the complete paths but rather the lynx configurations that had a valid linear interpolation through config space to either the start or end tree.

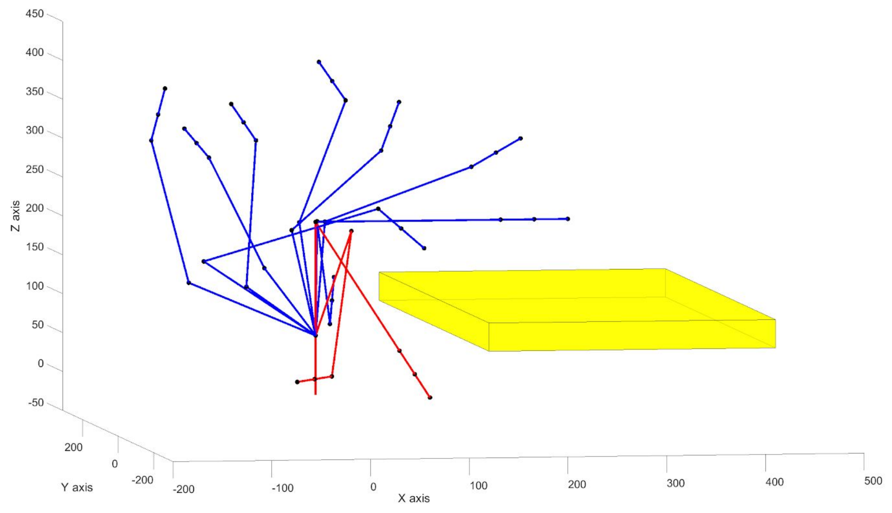
Obstacle	**Start Configuration**	**End Configuration**

emptyMap.txt	q=[0 0 0 0 0 0]	q=[1,1,1,1,1,0]
map1.txt	q=[0 0 0 0 0 0]	q=[0,0,1.1,0,0,0]
map2.txt	q=[0 0 0 0 0 0]	q=[1 0 0 0 0 0]
map3.txt	q=[0,0,0,0,0,0]	q=[1.4,0,0,0,0,0]
map4.txt	q=[0,0,0,0,0,0]	q=[0 0 1.4 0 0 0]
map5.txt	q=[0,0,0,0,0,0]	q=[1,1,1.1,0,0,0];
map6.txt	q=[0,0,0,0,0,0]	q=[0,1,1,1,1,1]
Map7.txt (this is an extremely dense map with 5 small objects)	q=[1.3, 0.7,1.7,0.2,-0.91,15]	q=[1,1.3,-1.5,1.5,0.0,0]

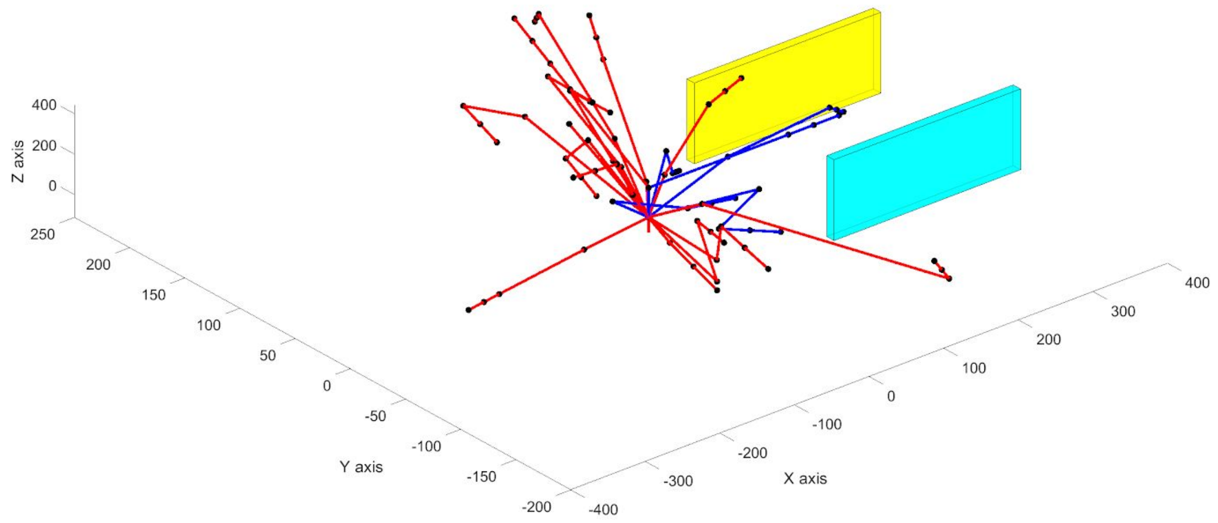
For the trivial empty map case we set up our algorithm to simply output a linear interpolation from the start to end node giving a tree that looks like



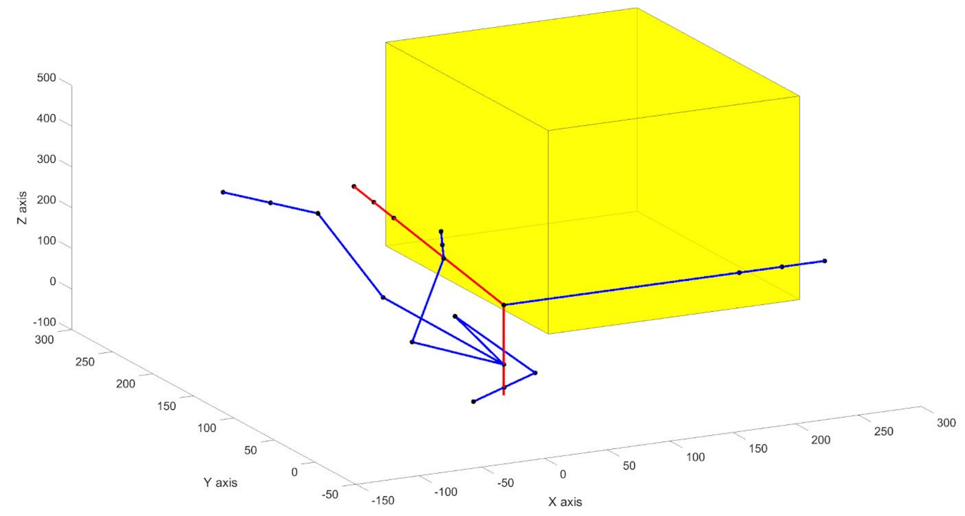
A sample tree in rrt for empty map red is goal tree blue is start the start and end are defined in table.



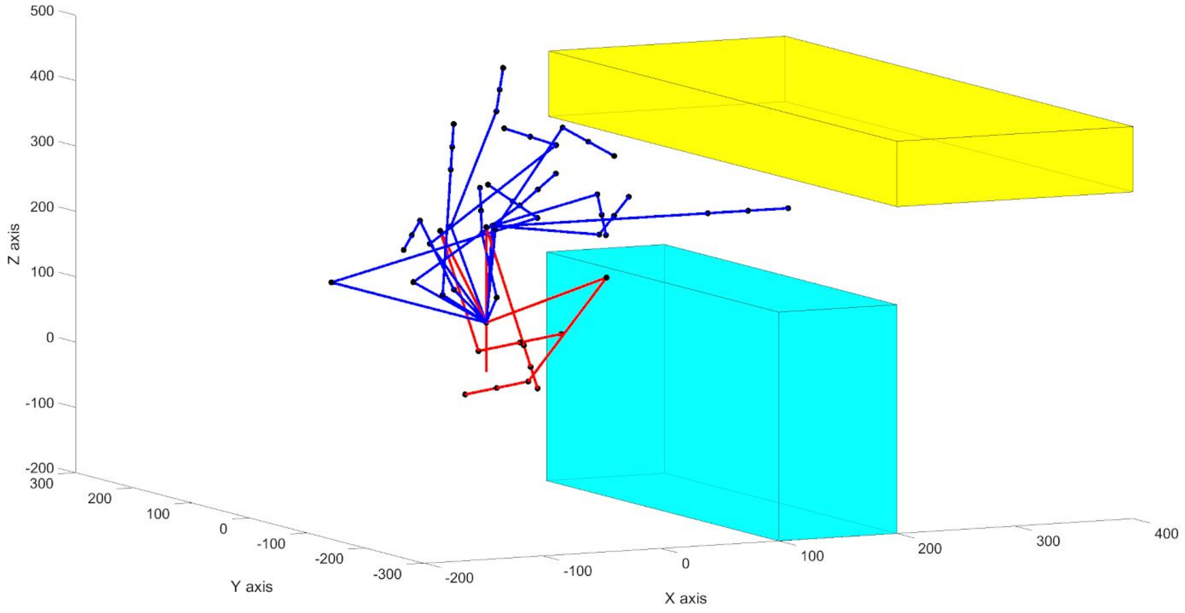
A sample tree in rrt for map1 red is goal tree blue is start the start and end are defined in table 1



A sample tree in rrt for map2 red is goal tree blue is start the start and end are defined in table 1

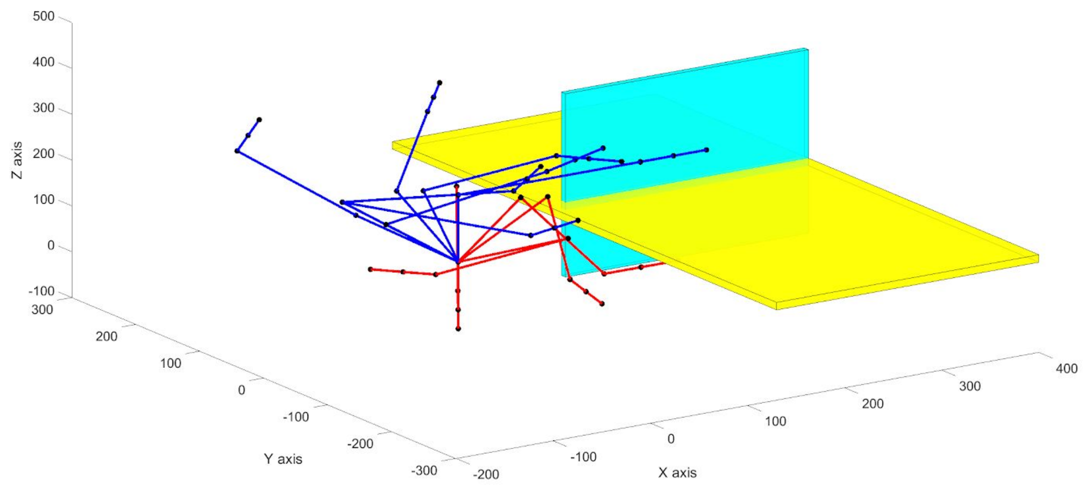


A sample tree in rrt for map3 red is goal tree blue is start the start and end are defined in table 1

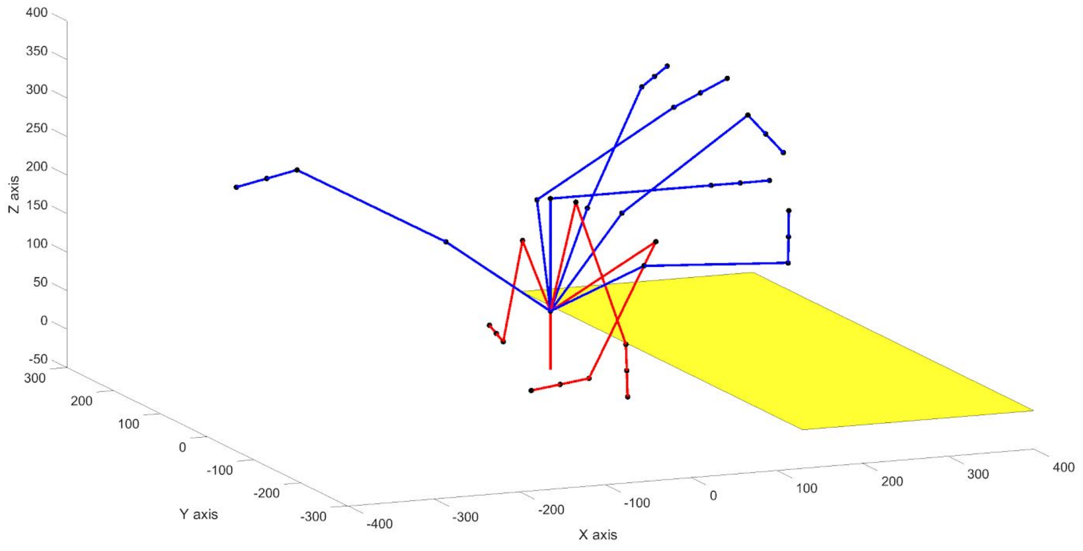


A sample tree in rrt for map3 red is goal tree blue is start the start and end are defined in table 1

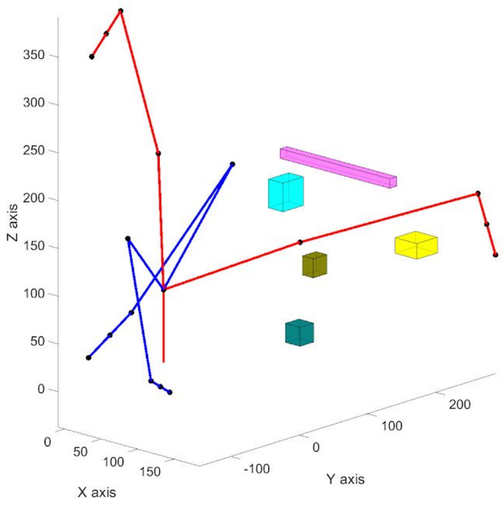
After running this plot script several times on the given maps with no link collisions in the start or goal trees we attempted to break our rrt code with our own suite of maps. The first attempt involved intersecting obstacles; we wanted to be sure that two obstacles overlapping did not cause an error in the given detect collisions file. After running our debug algorithm several times we were unable to get any trees that intersected with our obstacle. This communicated that the random nodes were indeed free of collision.



The second attempt involved seeing how our algorithm handled purely 2d obstacles . At first these obstacles gave us a great deal of trouble. However, by rethinking collisions with obstacles we were able to complete this test suite. The principle we used was the nonzero thickness of the link arms. By making a conservative estimate that incorporated the thickest arm into the dimensions of our obstacle we were able to detect collisions with 2d objects. The function that did this was expand obstacles which set the obstacle variable in our checkLine function to equal an expanded cube, which we lovingly called thiccObstacle, more on the checkLine function can be found in our code explanation.



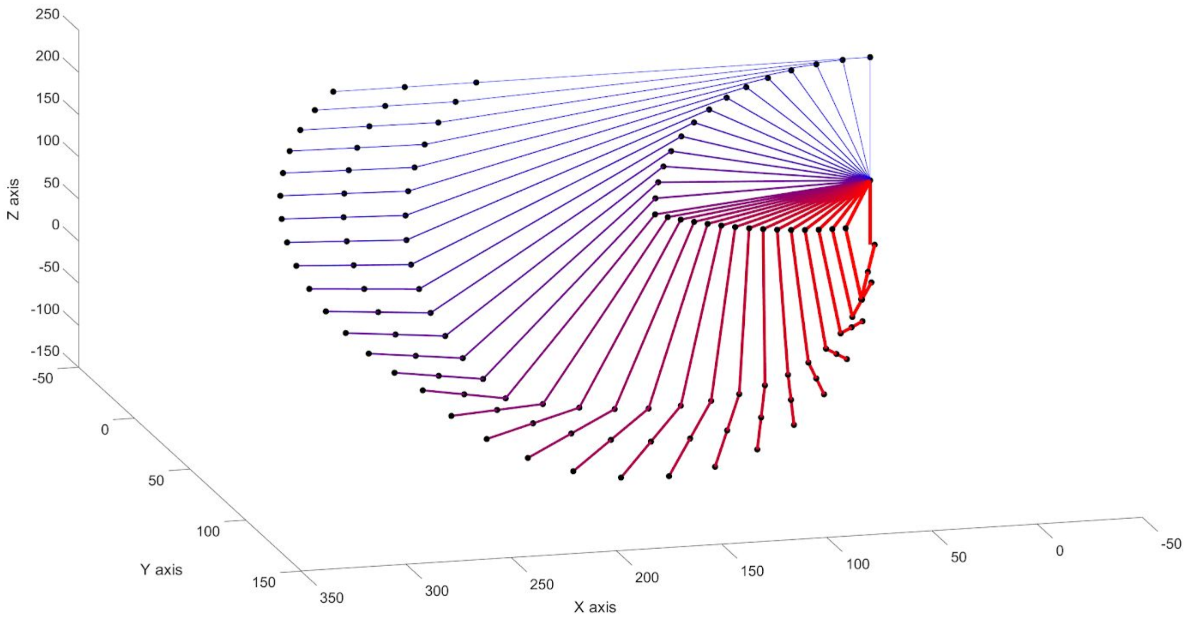
The final custom map attempted to break RRT and A* by providing an intensely populated map. We placed the end config in an almost impossible configuration but surprisingly the sample start and end trees were not very populated. RRT was able to find a solution with 2 to 4 random nodes typically.



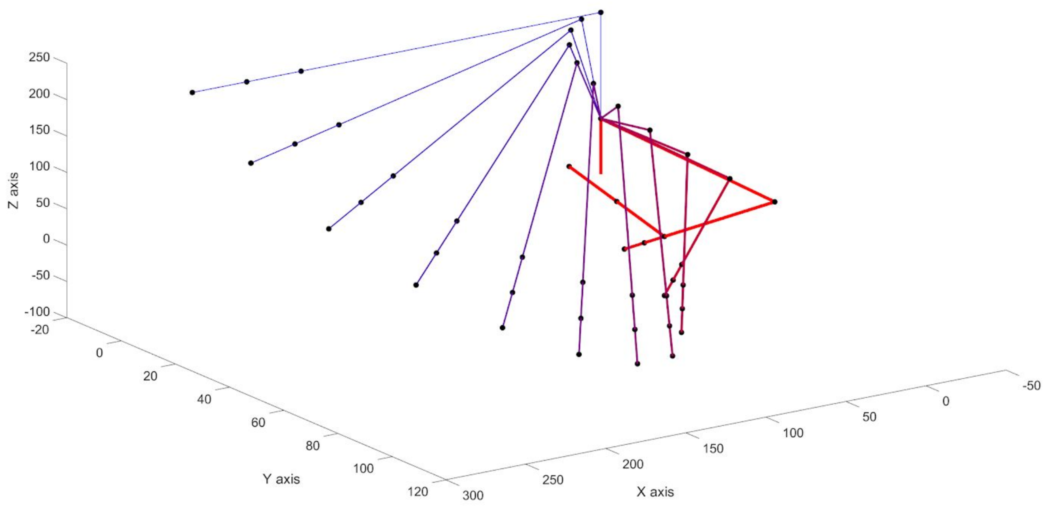
Once we were sure that the random configurations in our start and goal trees were valid we needed to be sure that the linear interpolation between these waypoints were indeed valid. We did this in two ways. The first was running a linear interpolation in config space in gazebo and the second was creating a plotting system to plot the trajectory of our path. We will discuss the latter technique first.

Custom Plot Trajectories

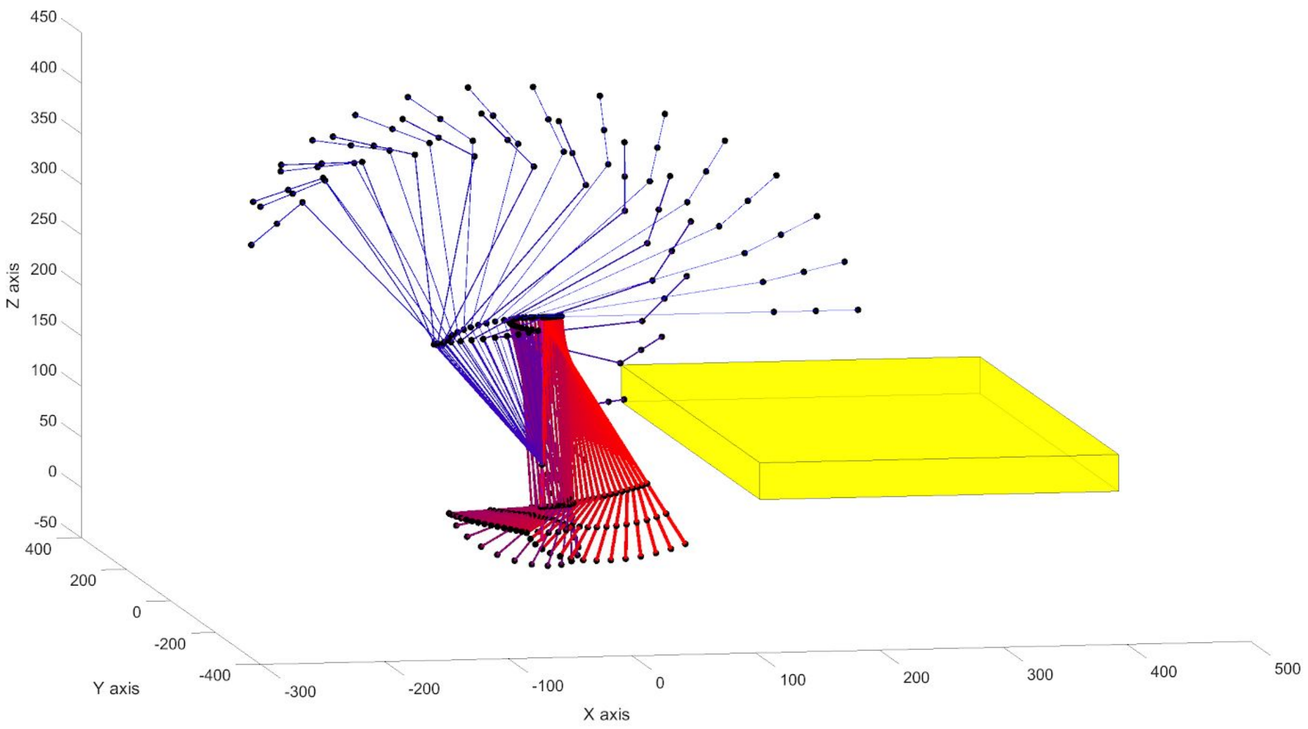
We modified the code used to output the start and goal trees of rrt to output the actual path of rrt because the paths for rrt can be quite complicated. We did our best to make the representation of the outputted path clear to see. We kept the blue, red convention of the start and end tree but now instead of indicating which tree a configuration belongs to it red indicates the amount the configuration has traveled along the path. For example a completely red config indicates the end goal configuration while a completely blue config represents the start configuration. We also vary the thickness of the link lines so that those at the start of the trajectory are thinner and those at the end are thicker as another method to distinguish between the configs in time.



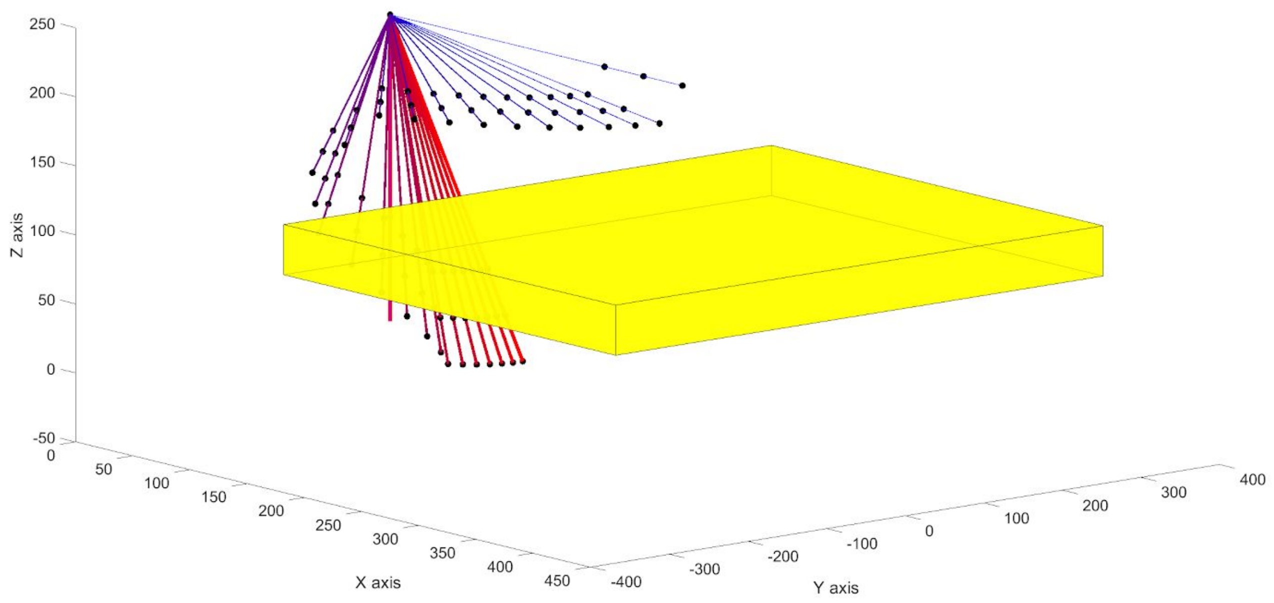
An RRT path in the empty map. The elapsed time was 0.007787 seconds.



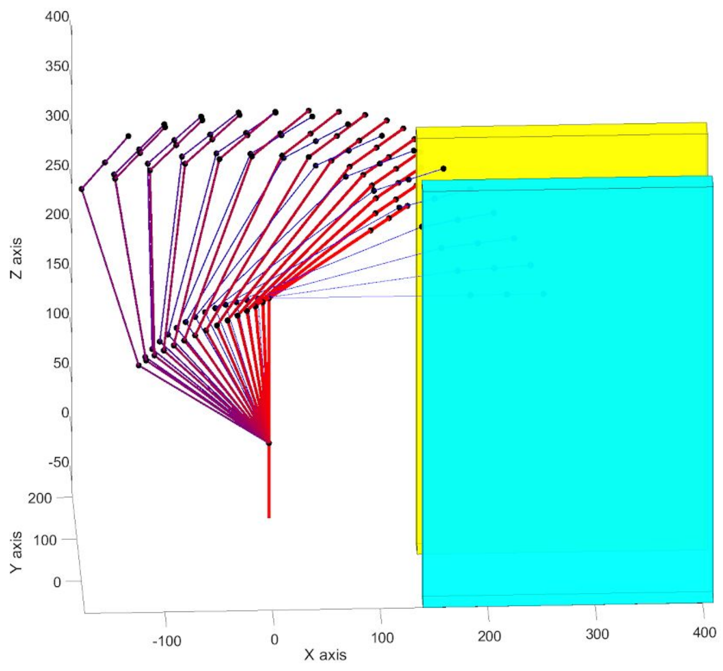
An A* path in the empty map. The elapsed time was 0.115732 seconds



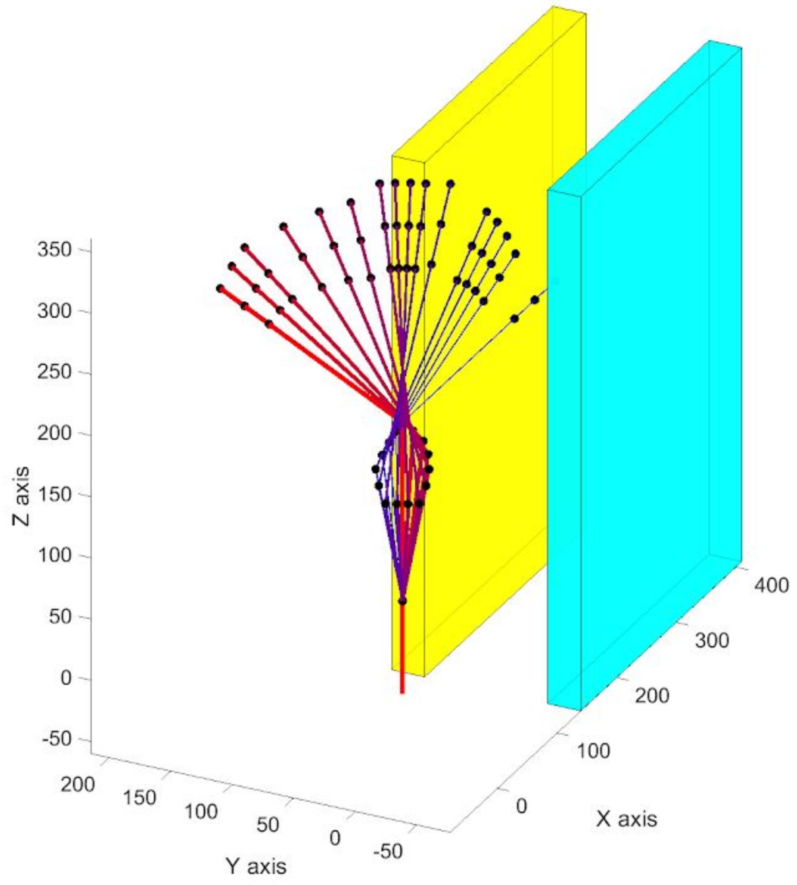
An RRT path in map1,elapsed time: 0.079194 seconds.



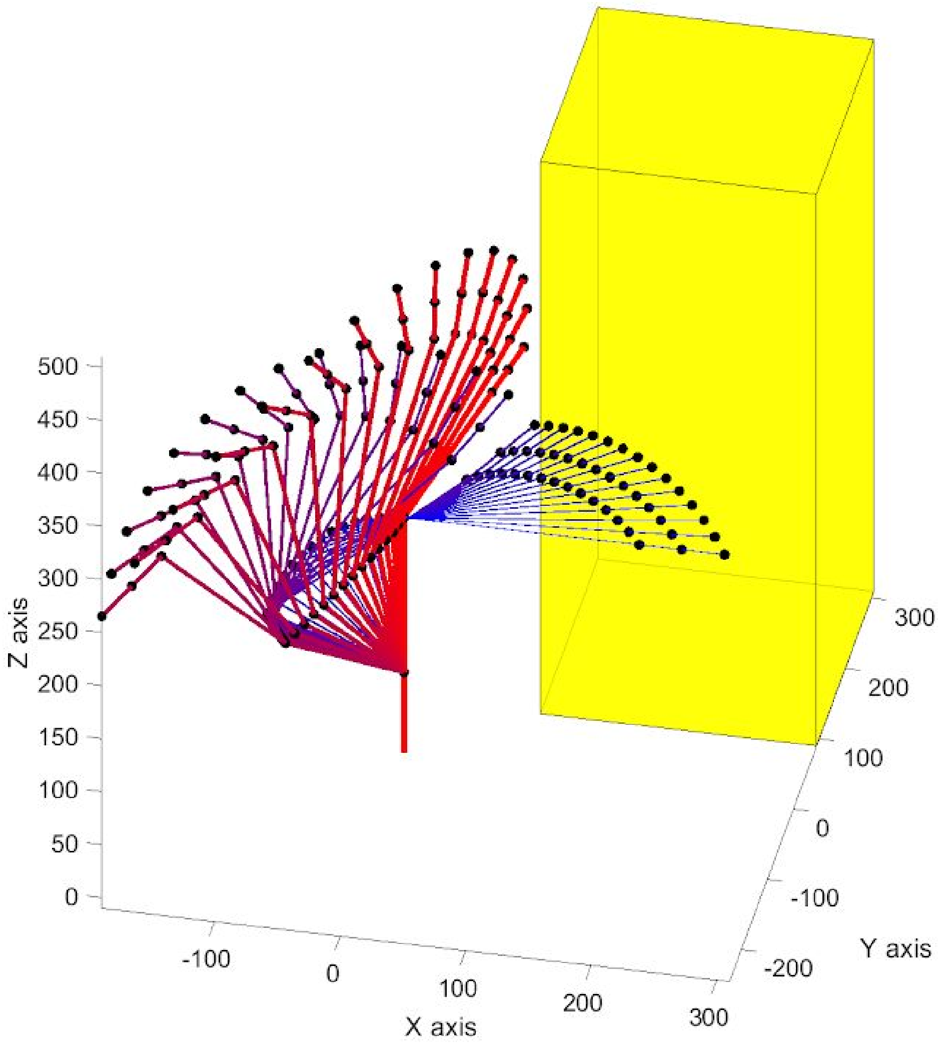
An A* path in map 1, elapsed time: 0.280332 seconds.



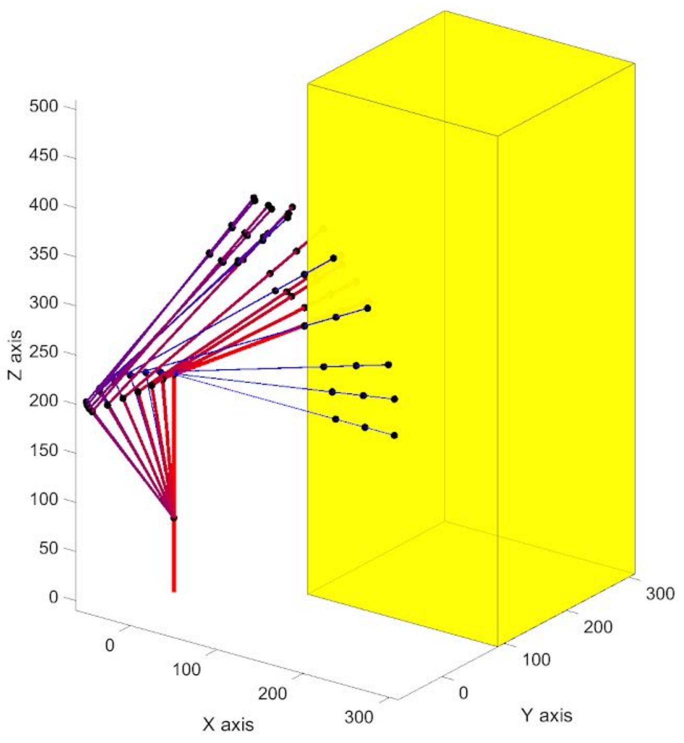
An RRT path in map2, elapsed time: 0.027028 seconds.



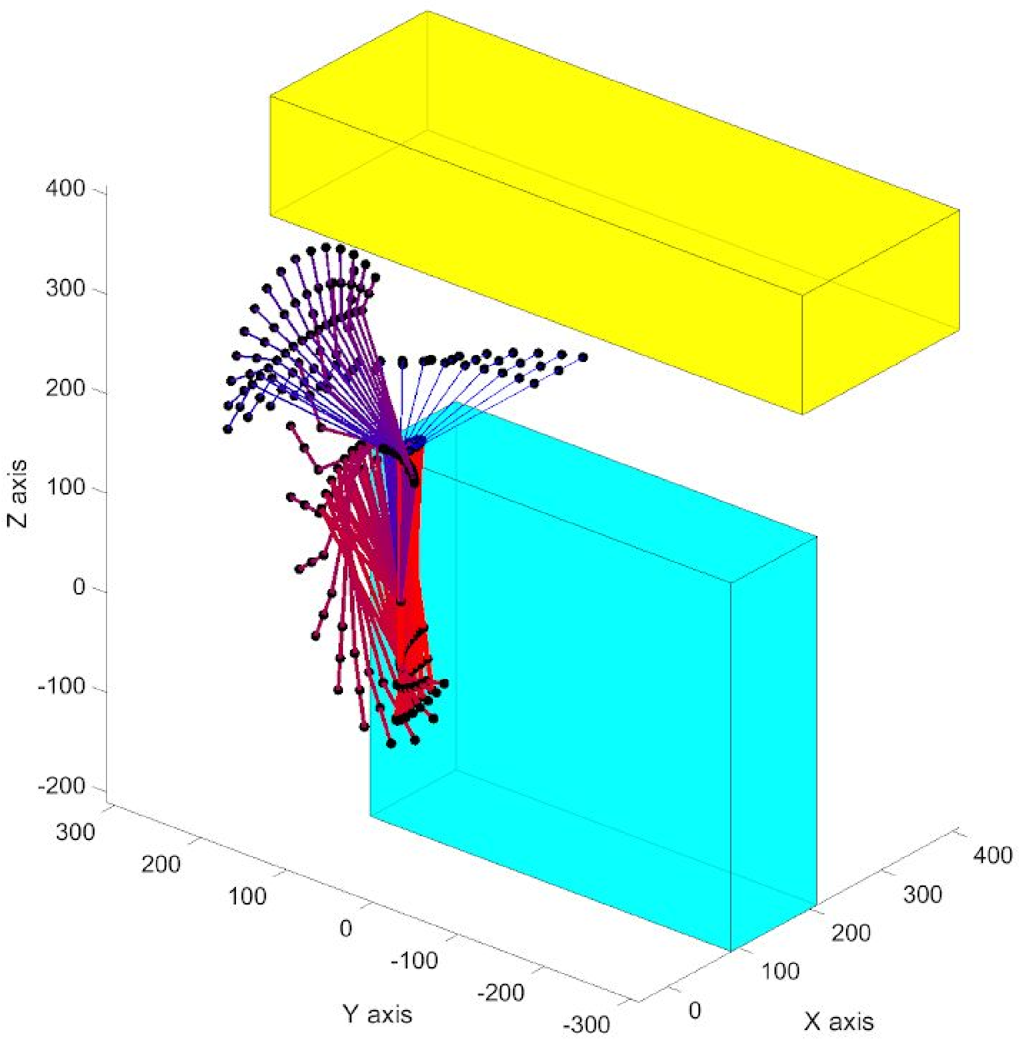
An A* path in map 2, elapsed time: 0.087618 seconds.



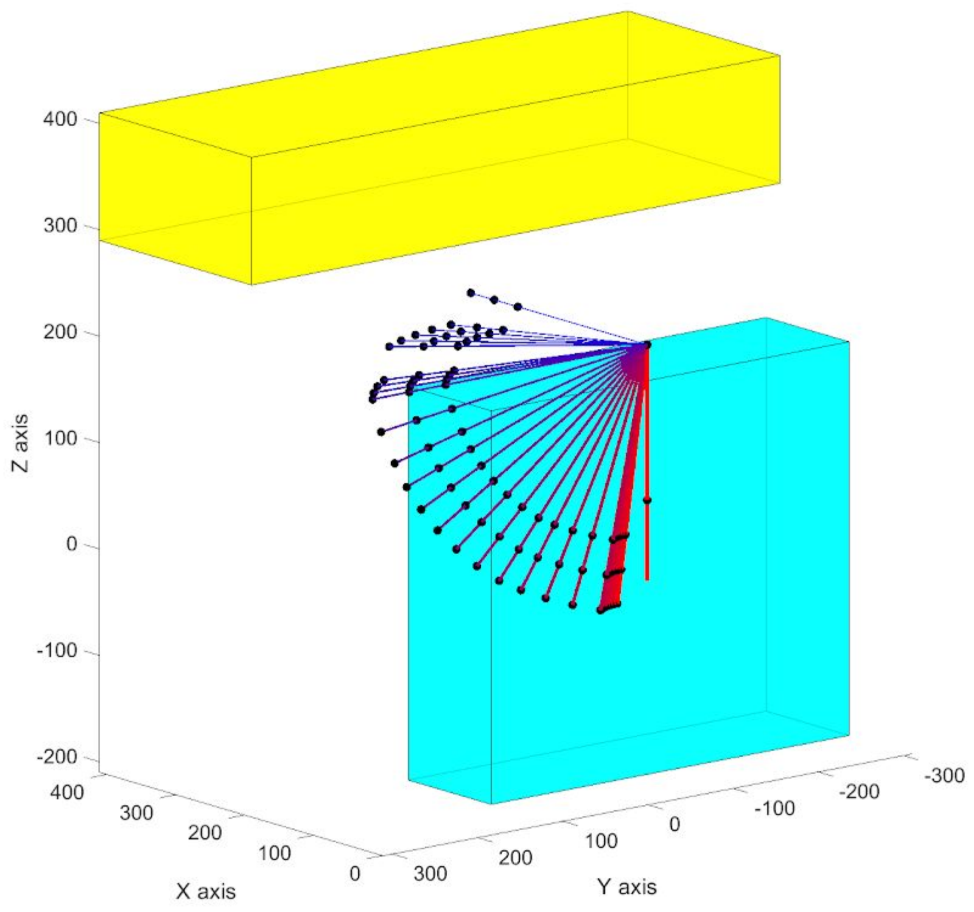
An RRT path in map3, elapsed time: 0.087788 seconds.



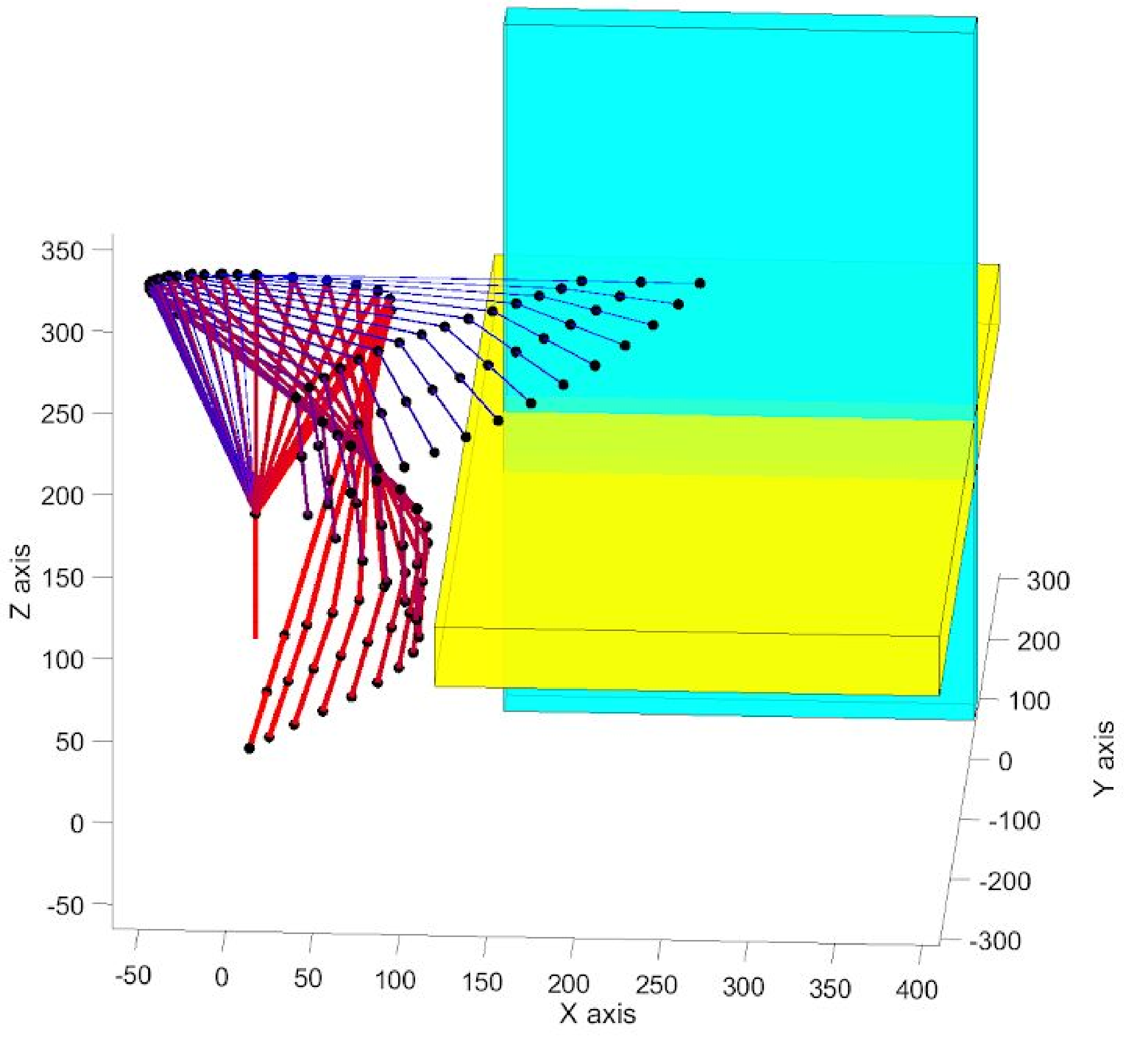
An A* path in map 3, elapsed time: 0.087026 seconds.



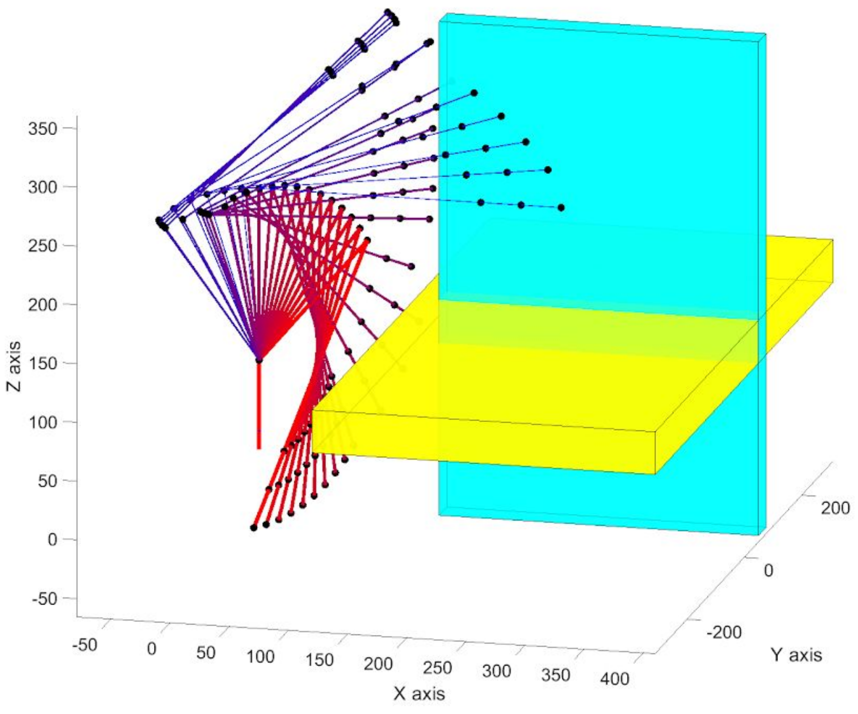
An RRT path in map4, elapsed time: 0.191051 seconds.



An A* path in map 4, elapsed time: 0.142068 seconds.

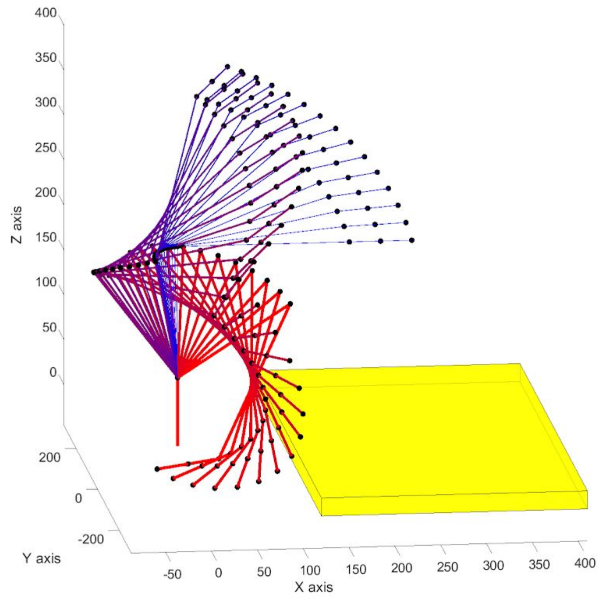


An RRT path in map5, elapsed time: 0.013798 seconds.

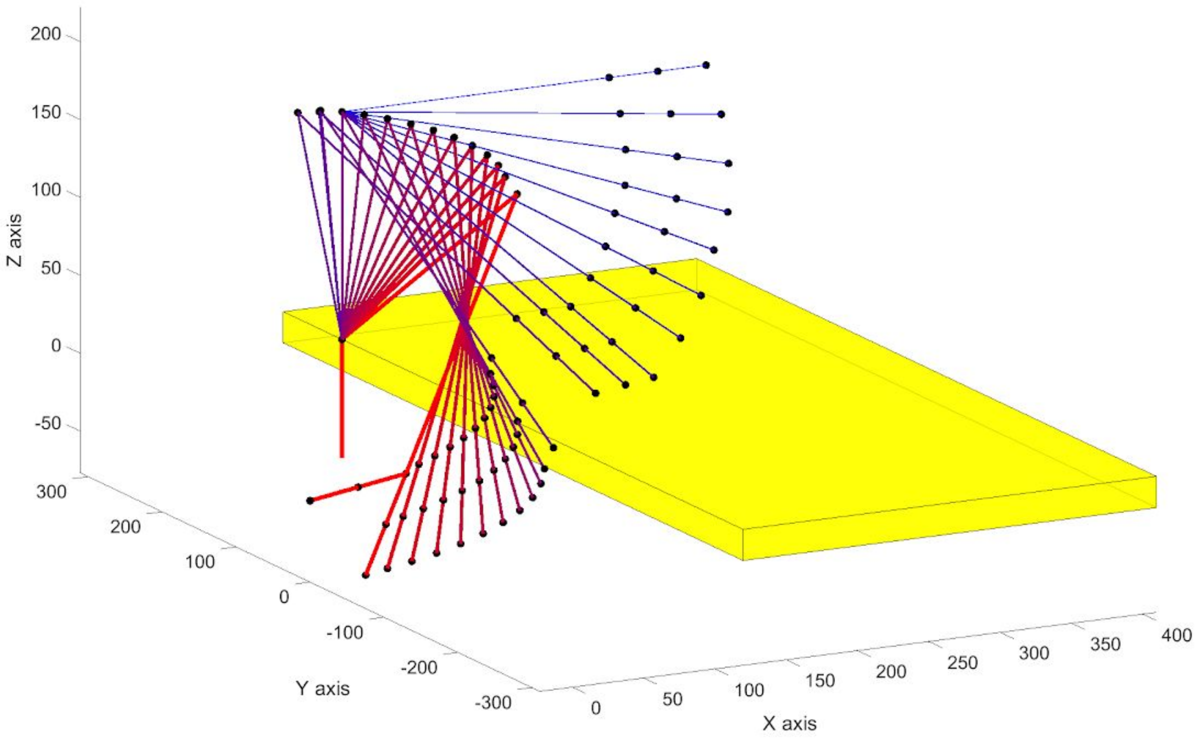


An A* path in map 5, elapsed time: 0.092623 seconds.

It is worth noting that map 6 is actually a 2d obstacle here. I chose to plot the thick obstacles rather than the original ones displayed in the previous section, so that you can see the conservatice estimate we made for the volume of the lynx arms.

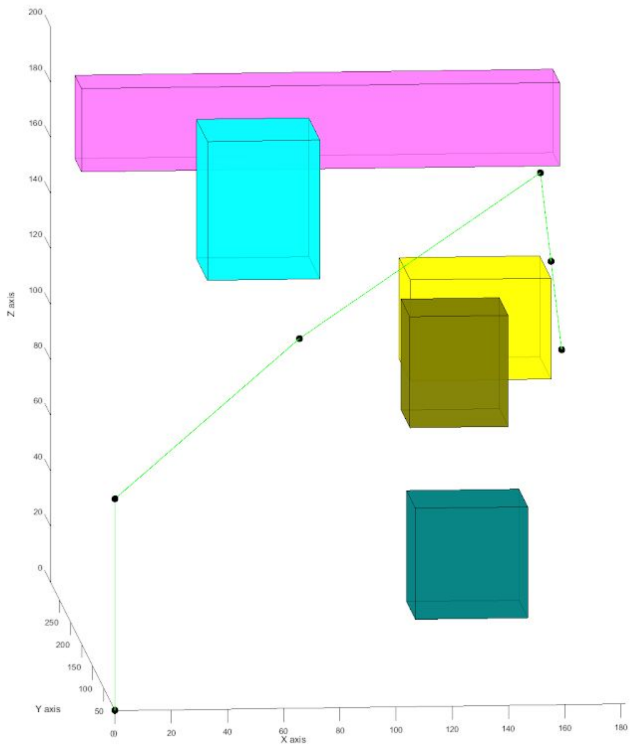


An RRT path in map6, elapsed time: 0.142666 seconds.

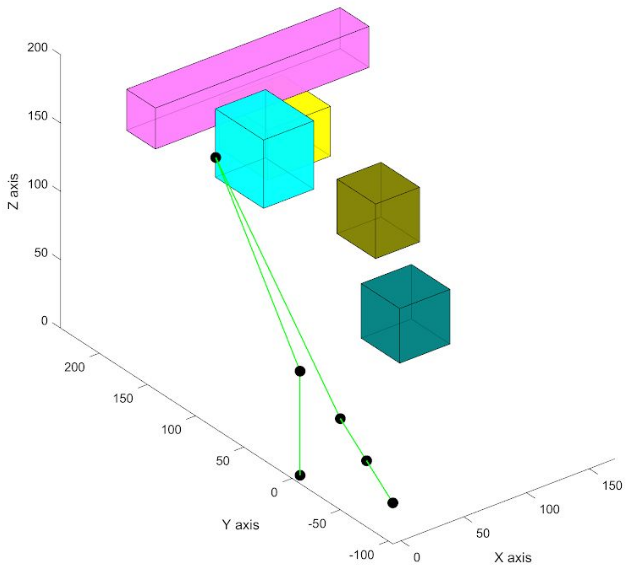


An A* path in map 6 , elapsed time: 0.235909 seconds.

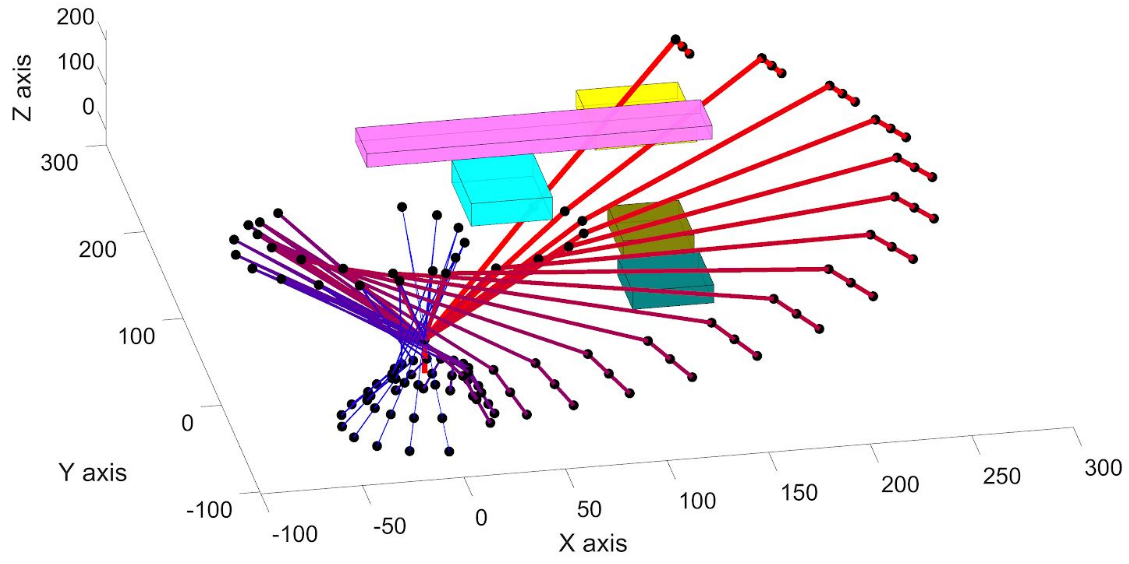
Map 7 was by far the most interesting case. Not only did RRT find the path extremely quickly and with few samples nodes, but A* was unable to complete this map. This is despite the fact that we made sure that the start and end configs did not touch any obstacles and the path itself was possible. A* thought the goal config touched an obstacle when it clearly does not. We assume this happened because A* did not discretize the space finely enough resulting in it registering valid free space as an obstacle. We also chose this time to point out that we have been choosing our start and end configs so that they are trackable in the plots. We ran similar tests with random,valid start configs and rrt worked fine. We choose to demonstrate this for the most interesting case of Map7 where the start config is clearly not the base configuration.



Goal config for map 7



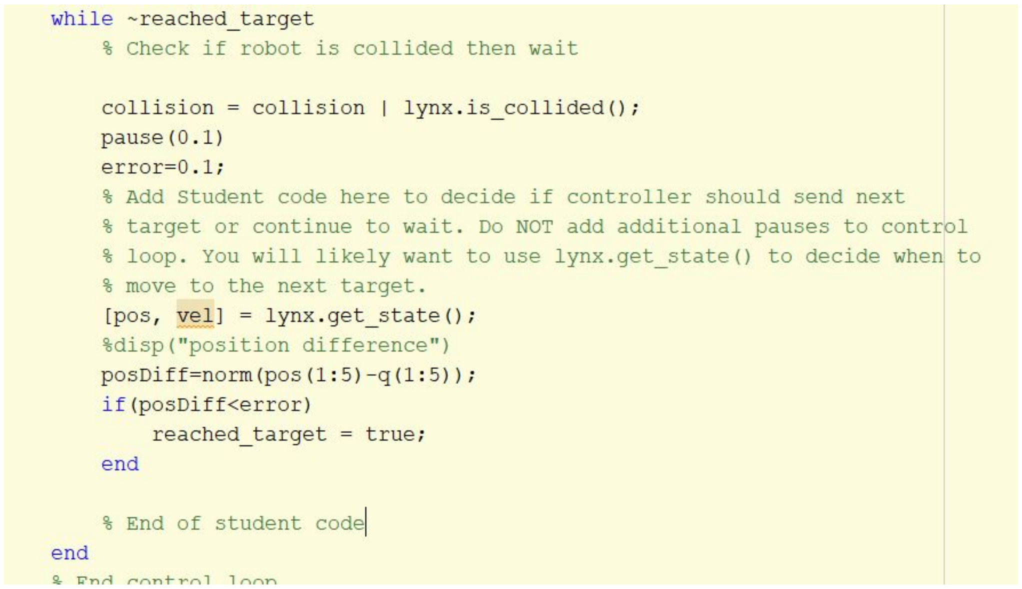
Start config for map 7



An RRT path in map7, elapsed time: 0.155009 seconds. When including a more conservative estimate for self collision with the base this map took significantly longer to run, about 10-30 seconds.

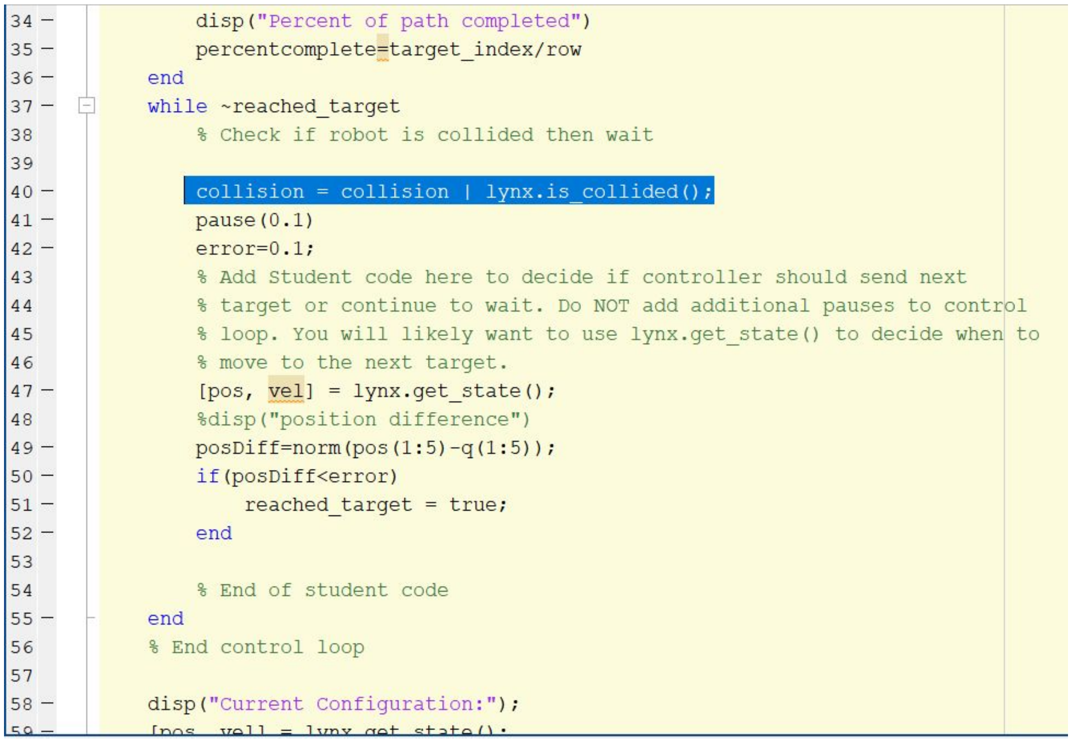
Gazebo

We noticed that imputed q's do not get linearly interpolated in ROS;instead, ros advances through the first joint, then the second, then the third, and so forth. As a result we had to force ROS to linearly interpolate our path in config space by creating a function called pathExpand. This function uses the waypoints that were outputted by RRT and interpolates through each waypoint based on a given step size. Because we set up RRT to only allow waypoints that have a valid linear interpolation to their parent node, then this results in pathExpand producing the equivalent path that the original RRT program outputted. We also ran into an issue that ros would go to the next waypoint before it reached the current waypoint. To solve this issue, we created an error function that only allowed ros to advance to the next waypoint once it was close enough to the current waypoint.



```
while ~reached_target
    % Check if robot is collided then wait

    collision = collision | lynx.is_collided();
    pause(0.1)
    error=0.1;
    % Add Student code here to decide if controller should send next
    % target or continue to wait. Do NOT add additional pauses to control
    % loop. You will likely want to use lynx.get_state() to decide when to
    % move to the next target.
    [pos, vel] = lynx.get_state();
    %disp("position difference")
    posDiff=norm(pos(1:5)-q(1:5));
    if(posDiff<error)
        reached_target = true;
    end

    % End of student code
end
% End control loop
```

After performing this fix, ros was able to take the path outputted from rrt and advance along the desired trajectory. Videos of our trajectories in ros can be found in the zip folder labeled videos. We did not complete our personal test suite in Ros because the custom trajectory plotting gave a more convenient way to analyze more interesting paths;moreover, using ros proved to be unreliable resulting in us not having enough time to test our personal maps in Ros. We were not able to get Ros to work until about 2 days before this deadline because of the error in line 40 of the provided test_pathsim.



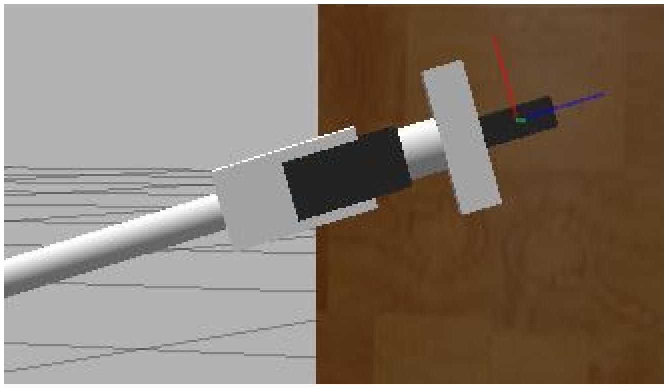
```
        disp("Percent of path completed")
        percentcomplete=target_index/row
    end
    while ~reached_target
        % Check if robot is collided then wait

        collision = collision | lynx.is_collided();
        pause(0.1)
        error=0.1;
        % Add Student code here to decide if controller should send next
        % target or continue to wait. Do NOT add additional pauses to control
        % loop. You will likely want to use lynx.get_state() to decide when to
        % move to the next target.
        [pos, vel] = lynx.get_state();
        %disp("position difference")
        posDiff=norm(pos(1:5)-q(1:5));
        if(posDiff<error)
            reached_target = true;
        end

        % End of student code
    end
    % End control loop

    disp("Current Configuration:");
```

The process to run the ros test was also time intensive because they required us to restart ros on every interaction. The record feature in ros also has its limitations and we were forced to use a windows screen recording service that frequently failed to detect gazebo. Despite these issues we were still able to have time to run ros on the given suite of maps. And the videos can be found in the provided video zip file.

Analysis

If we had more time with this lab, we would additionally implement either the RRT* algorithm, or do additional post-processing after a valid path has been found to shorten the path. This would drastically reduce the large discrepancy in total path distance between the A* algorithm and our RRT algorithm at the slight cost of processing time. Additionally, we would increase our sample size when evaluating the average processing times to get more accurate data.

An issue that we ran into were collisions in the Gazebo workspace that were not considered collisions in our planning environment. This was due to the geometry of the end effector, which when θ_5 is in the zero configuration (as seen in the image below), creates a geometry whose radius is much larger than the radius of the end effector. To remedy this problem, we could have increased the collision radius in our planning space to be greater than the end effector's radius; however, this would have caused the environment to be much too conservative.



Another potential reason why we ran into unexpected collisions could have been due to differences between our waypoint interpolation method and Gazebo's waypoint interpolation method. Because our interpolation method was linear, this difference would mean that the route which Gazebo had planned would be slightly different than ours. As a few of our algorithm's planned paths were quite close to the obstacles, this slight difference would have caused collisions.

To quantify the differences between our RRT implementation and the provided A* implementation, both algorithms were run 50 times for each of the four provided map obstacles and the start to end configurations below. The number of iterations was selected to be 50 as it seemed to be a large enough sample size while placing a reasonably low workload on the computer running the simulation. Start and end configurations were selected such that the robot arm would have to move a significant amount in order to avoid the obstacles.

The average processing time and the configuration path length were recorded and displayed below. The path lengths for the configuration space was computed as

$$\sum_{k=1}^{n}\sum_{i=1}^{5}\|JointPos_{i+1} - JointPos_i\|$$

where n = number of waypoints. Thus, the average workspace path length is the summation of norms of the vector from one waypoint to the next for each joint.

A* Algorithm			
Obstacle	**Avg Processing Time (sec)**	**Avg Workspace Path Length (m)**	**Avg Configuration Path Length (rad)**
emptyMap.txt	0.0684	520.8246	1.6667
map1.txt	0.2441	557.5379	1.6667
map2.txt	0.1040	482.1315	0.8000

map3.txt	0.1453	581.6374	1.0667
map4.txt	0.2152	530.3679	1.200
map5.txt	0.1905	840.9029	1.5667
map6.txt	0.3359	553.5173	2.400
map7.txt	N/A[1]	N/A	N/A

RRT Algorithm			
Obstacle	**Avg Processing Time (sec)**	**Avg Workspace Path Length (m)**	**Avg Configuration Path Length (rad)**
emptyMap.txt	0.0029	966.4880	11.1007
map1.txt	0.0774	1,430.6	27.5104
map2.txt	0.0363	1,198.3	19.6802
map3.txt	0.0550	1,519.4	24.0476
map4.txt	0.1516	1,390.8	28.6654
map5.txt	0.1467	1,473.3	28.0807
map6.txt	0.0349	1,020.4	17.0206
map7.txt	0.1365	1,466.2	21.1975

As can be observed from the figures above, the main differences that can be noted between the A* algorithm and RRT are between average processing time, average workspace path length, and average configuration path length. The A* algorithm excels at finding the optimal path between start and end configurations, as can be noted by the very small configuration path lengths compared to the RRT algorithm. This makes sense, as A* is a search-based algorithm, where the planner first discretizes and maps the entire workspace including obstacles, then finds the shortest path between those obstacles. This means that depending on the heuristic, A* is guaranteed to find the shortest path within the workspace, provided its map of the sampled space is accurate and its discretizations small enough.

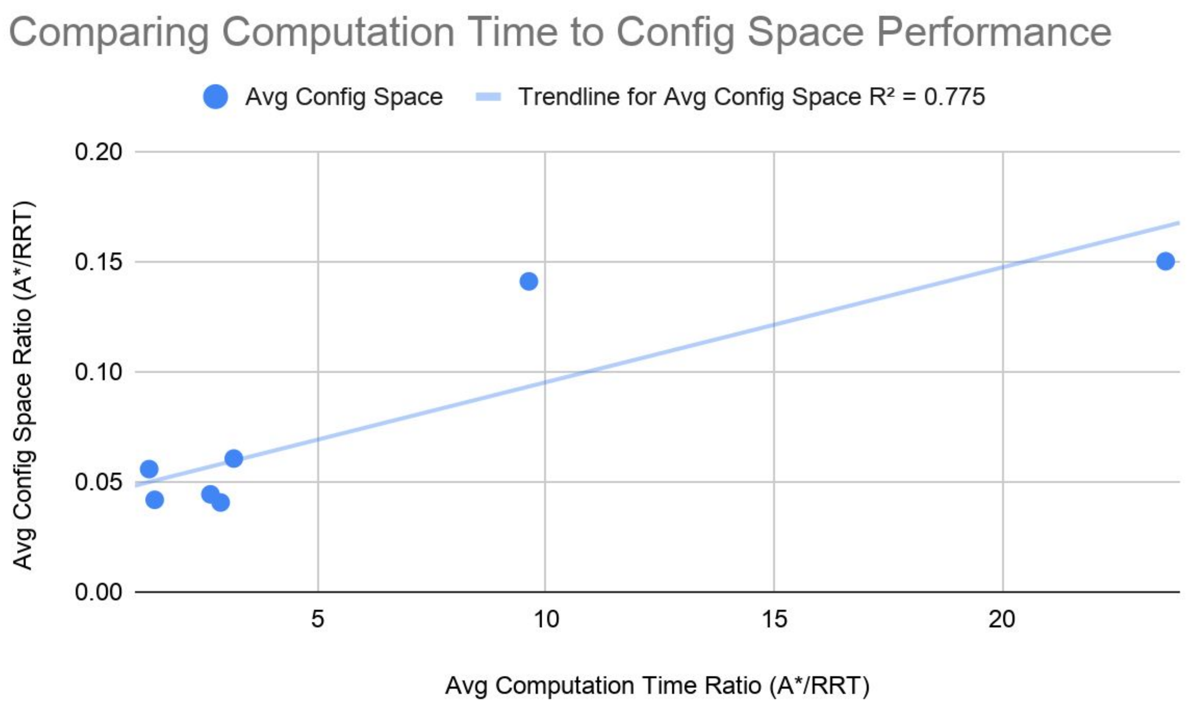
On the other hand, RRT is a sampling-based algorithm, where the planner selects random nodes in the configuration space and does not know a priori whether a given sample node is in

[1] The A* algorithm was not able to find a path for the map7.txt test case as it detected that the goal configuration was in an obstacle, which was most likely due to the proximity of the goal configuration to an obstacle. However, our RRT algorithm was able to find a path. If the A* algorithm does not discretize the space finely enough, the algorithm is not able to detect potential candidate configurations that may actually exist.

collision or not. Thus, although it is guaranteed to find a path as time approaches infinity, there may be some instances where RRT takes significantly longer than a search based algorithm although this is not the norm as can be observed by the data above. Additionally, our RRT algorithm did not post-process computed waypoints to find shorter paths, so this explains the blatantly excessive path lengths.

However, A*'s ability to find an optimal path comes at the cost of processing time. As can be seen above, the average processing time for RRT is significantly lower for all maps.

Map	**Ratio of Avg Times (A*/RRT)**	**Ratio of Avg Config Space Path Lengths (A*/RRT)**
map1	23.5862069	0.1501436846
map2	3.15374677	0.06058436082
map3	2.865013774	0.04064999339
map4	2.641818182	0.04435785692
map5	1.419525066	0.04186231485
map6	1.298568507	0.0557927687



The differences between the maps correlates with the number of objects and configuration of said objects in each map. For maps with one or fewer obstacles, our RRT algorithm was able to perform quite well with respect to computation time, as noted above. However, for maps with two obstacles and narrow corridors, our RRT algorithm performed worse than A* with respect to time. This makes sense as RRT's use of sampling utilizes probabilistic sampling with a normal distribution. As such, if a goal configuration requires the robot to navigate into or out of a narrow corridor, it is unlikely that RRT's random sample will select a configuration that is a) outside of the obstacle, or b) part of a path that is able to navigate the robot into or out of said narrow corridor.

Description of Functions

- addEdge - adds a new edge to the tree adjacency matrix
- checkLine - checks whether the line connecting each new candidate waypoint has collided with an object or not. Additionally, it discretizes the number of steps such that each discretization is within a certain step size.
- closeNode - gets the closest node already in the tree to the candidate waypoint
- ConnectTrees - walks through both matrices after a path has been found and generates the path from start to goal configuration
- createA - calculates FK matrix using DH parameters
- expandObstacles - increases the dimensions of each obstacle by a radius = to the largest link's radius
- ExpandPath - interpolates between path points in real space
- ItWerks - main driver function for RRT algorithm
- makeLine - creates a discretized line from one point to another based on a given number of steps
- plotJointPos - plots joint positions for a given instance in time in 3D space
- randpoint - generates a random point in configuration space using normal distribution
- calculatePathDistance - computes the computation time, workspace path distance, and configuration space path distance for a given path from start to end configuration
- compareAstartToRRT - driver function for calculatePathDistance

Conclusion

To conclude, in this lab we were able to compare the performance of the A* algorithm to a version of the RRT algorithm that we implemented. We found that the A* algorithm was able to generate optimal paths at the cost of computation time, and that the RRT algorithm was able to compute sub-optimal paths but at a fraction of A* algorithm's time.

In summary, the RRT planner is most likely best used for situations where navigation within cluttered or confined workspaces is not present and extremely quick calculation times or limited processing power is needed. The most obvious example is a robot in an open space, such as outdoors or in a large room where the robot needs to continuously update its trajectory to avoid relatively small moving obstacles. Conversely, the A* planner is most likely best used for situations where navigation within cluttered and confined spaces is necessary, but it is not required to have extremely quick calculations and reaction times.