Robotics 2017

MEAM 520
University of Pennsylvania

Lab 3 – Trajectory Planning

Lab Due Date: 10/18/2017

Jeremy Saslaw

Jeremy Saslaw

Planning Trajectories for the Lynx Robot

This lab is about planning collision-free trajectories through a cluttered environment using the Lynx robot.

Method

Consider the Lynx robot trying to move its end effector from a point p_{start} in its workspace to another point p_{end}. The environment has i spherical obstacles in it, which are centered at positions $[x_i, y_i, z_i]^T$, have radius r_i, and are attached to the boundaries of the environment using straight vertical wires from the top (line segments). Devise a planner that will allow the robot to perform this movement without colliding with any of the obstacles. Questions to consider include:

- **How to represent the workspace**
- **How to check for collisions (with the end effector? With the rest of the robot?)**
- **How to account for the geometry (volume) of the robot**
- **How to compute waypoints**
- **How to interpolate between waypoints**

Explain your choices. Include your pseudocode in the report.

In order to select the optimal path planning technique, the final application of the robot would need to be given. For instance, different techniques prioritize different characteristics. For example, one technique may prioritize faster robot speed from a given start point to a given end point, while another technique may prioritize faster computer processing to solve for any given path. With the design requirements provided, the most logical approach to achieve proper path planning seemed to be a Rapidly-exploring Random Trees (RRTs) method. It was decided that the goal would be to solve for the most efficient program as possible, which would solve for a given path that adhered to all the requirements, in as fast a manner as possible. This would be ideal for robot applications where rapid movements between different waypoints would be necessary, as computation time in between points would be minimized.

The process by which the final path planning code was determined included creating various functions, which would be used together to create a final path planning function. Each of these functions serve a specific purpose, where each was determined to be a valuable application that would be useful to have in a concise function where they can be called at any time. Each of these functions will be introduced below, where the name will be shown, followed by a brief summary of its use, and finally its pseudocode. Following this, the final path planning function that operates on the RRT method will be discussed by walking through its pseudocode. Finally, after this process of how the path planning was undertaken has been introduced, the questions to consider will be answered, and explanations will be given for each important concept.

function [workspaceQ] = randomQ()

This function outputs a random Q = [q1, q2, q3, q4, q5, 0] that is anywhere in the robot's reachable workspace. The only limiting factor to what Q may be is by the Lynx robot's joint limits.

- Input max and min joint limits for each joint.
- For each q
    - Determine random float from 0 → 1
    - Multiple this random number by the allowable range of joint angles
    - Add the minimum joint angle to this number to create the q for this random configuration
- Output workspaceQ = [q1, q2, q3, q4, q5, 0]

function [tester] = isValidQ(Q,R,P)

This function takes in any Q = [q1, q2, q3, q4, q5, 0], an array R = [r1, r2, r3, ...] which represents the radius of each given spherical obstacle, as well as an array P = [x1, y1, z1; x2, y2, z2; ...] which represents the center of each position of each obstacle. This function will output a bool "tester" which equals 1 if the Q is a valid position in the free space, and will equal 0 if the Q is a non-valid position in the freespace.

- Increase radius of each obstacle by 1.5 to account for robot joints in 3-D space
- Assume position is valid, and test until found otherwise, or all tests prove true
- Create array "position" which is a 6 x 3 matrix of each joint position
- While tester is true
    - For each joint position 3, 4, and 6
        - For each obstacle
            - Test if x position is less than the (obstacle x center + its radius) AND greater than the (obstacle x center – its radius)
            - Test if y position is less than the (obstacle y center + its radius) AND greater than the (obstacle y center – its radius)
            - Test if z position is less than the (obstacle z center + its radius) AND greater than the (obstacle z center – its radius)
            - If all three tests are true, position is not valid, and tester is false
    - For each link
        - Create position vector for the link

- Determine which side of the link has a greater x
- Create a range of x values which increases from the minimum x to the maximum x value of the link, where the step size is determined by a predetermined "steps" variable, which is currently 100
- Create corresponding y and z ranges
- For each point (x, y, z)
    - For each obstacle check if link hits obstacle
        - Test if x position is less than the (obstacle x center + its radius) AND greater than the (obstacle x center – its radius)
        - Test if y position is less than the (obstacle y center + its radius) AND greater than the (obstacle y center – its radius)
        - Test if z position is less than the (obstacle z center + its radius) AND greater than the (obstacle z center – its radius)
        - If all three tests are true, position is not valid, and tester is false
    - For each obstacle check if link hits string (which is modeled as a rectangular prism of width "wide")
        - Test if x position is less than the (obstacle x center + wide) AND greater than the (obstacle x center – wide)
        - Test if y position is less than the (obstacle y center + wide) AND greater than the (obstacle y center – wide)
        - Test if z position is greater than the obstacle z center
        - If all three tests are true, position is not valid, and tester is false
- Break the while loop because all tests passed

function [Q] = freespaceQ(R,P)

This function takes in an array R = [r1, r2, r3, ...] which represents the radius of each given spherical obstacle, as well as an array P = [x1, y1, z1; x2, y2, z2; ...] which represents the center of each position of each obstacle. This function will output a random Q = [q1, q2, q3, q4, q5, 0] that is valid for the given freespace.

- Generate random Q via randomQ() function

- While this random Q isn't valid
    - Generate new randomQ until valid one is found
- Output the valid Q

function [tester] = isCollide(Q1, Q2, R, P)

This function takes in any Q1 = [q1, q2, q3, q4, q5, 0] and Q2 = [q1, q2, q3, q4, q5, 0], an array R = [r1, r2, r3, ...] which represents the radius of each given spherical obstacle, as well as an array P = [x1, y1, z1; x2, y2, z2; ...] which represents the center of each position of each obstacle. This function will output a bool "tester" which equals 1 if there is a collision between the two configurations, and equals 0 if there is no collision between the two configurations.

- Increase radius of each obstacle by 1.5 to account for robot joints in 3-D space
- Create an array "qArray" of intermediate Q configurations between Q1 and Q2 which are spaced linearly in "dt" steps
- Check if each Q in qArray produced a valid position in the workspace by calling isValidQ(Q,R,P) for each row and checking output
- While tester = 0 (meaning there is no collision)
    - For each intermediate configuration between Q1 to Q2
        - Determine array "positionSet" which is a 6 x 3 matrix of each joint position
        - For each link
            - Create position vector for the link
            - Determine which side of the link has a greater x
            - Create a range of x values which increases from the minimum x to the maximum x value of the link, where the step size is determined by a predetermined "steps" variable, which is currently 100
            - Create corresponding y and z ranges
            - For each point (x, y, z)
                - For each obstacle check if link hits obstacle
                    - Test if x position is less than the (obstacle x center + its radius) AND greater than the (obstacle x center – its radius)
                    - Test if y position is less than the (obstacle y center + its radius) AND greater than the (obstacle y center – its radius)

        - Test if z position is less than the (obstacle z center + its radius) AND greater than the (obstacle z center – its radius)
        - If all three tests are true, position is not valid, and tester is false
    - For each obstacle check if link hits string (which is modeled as a rectangular prism of width "wide")
        - Test if x position is less than the (obstacle x center + wide) AND greater than the (obstacle x center – wide)
        - Test if y position is less than the (obstacle y center + wide) AND greater than the (obstacle y center – wide)
        - Test if z position is greater than the obstacle z center
        - If all three tests are true, position is not valid, and tester is false
- Break the while loop because all tests passed

function row = closestNode(Qnew, Qarray)

This function takes in any Qnew = [q1, q2, q3, q4, q5, 0] and Qarray = [q1, q2, q3, q4, q5; q1, q2, q3, q4, q5; ...]. This function outputs a int "row" which specifies which row of the Qarray the closest node to Qnew exists at.

- Find position of end effector of Qnew configuration
- Assume closest configuration in Qarray is the first row of the matrix, and test further to verify
- Distance "d" between end effector of Qnew and end effector of current closest configuration is determined by comparing two points in 3-D space
- For each row of the Qarray
    - If the distance between the end effector of the given configuration is closer to the Qnew end effector than the distance currently set as "d," change "d" to this new distance and modify the row of the closest configuration to this given row
- Output the row of Qarray that contains the configuration with an end effector closest to that of Qnew

function Qfinal = pathPlanning(Qstart, Qend, R, P)

This function takes in an array Q1 = [q1, q2, q3, q4, q5, 0] from the freespace and Q2 = [q1, q2, q3, q4, q5, 0] from the freespace, an array R = [r1, r2, r3, ...] which represents the radius of each given spherical obstacle, as well as an array P = [x1, y1, z1; x2, y2, z2; ...] which represents the center of each position of each obstacle. This is the main path planning function which will use the previous functions as well as some new logic to determine the path from Qstart to Qend. This function will output Qfinal = [qstart1, qstart2, qstart3, qstart4, qstart5, 0; ... ; ... ; ... ; qend1, qend2, qend3, qend4, qend5, 0], which represents the Q steps to get from Qstart to Qend. Additionally, this function has the functionality to have the robot move automatically from Qstart to Qend, but this portion is commented out to allow the user to see the Qfinal, and then make the movements manually in order to allow for systematic analysis.

- Create "Q1array" with Qstart as the current first and only row, which will grow to track all Q configurations with reference to the starting position
- Create "T1" (tree 1) as [1], which will grow to track all possible branches from Qstart to new configurations that are found in Q1array
- Create int "aTracker" set as 2, which will track where the next row in Q1array should be placed
- Create "Q2array" with Qend as the current first and only row, which will grow to track all Q configurations with reference to the ending position
- Create "T2" (tree 2) as [1], which will grow to track all possible branches from Qend to new configurations that are found in Q2array
- Create int "bTracker" set as 2, which will track where the next row in Q2array should be placed
- While a path isn't found from Qstart to Qend
    - Use freespaceQ(R,P) to find "q," a random configuration in the freespace
    - Use closestNode(q,Q1array) to find the row of the configuration in Q1array that has an end effector closest to the end effector of configuration q, set as "qa"
    - If q and qa don't collide
        - Add q to Q1 array
        - Find what branch in T1 has qa as an end of the branch (last column of the matrix) using ismember() Matlab function
        - Add an extra row to T1 at the bottom of the array
        - Add a column of all zeros to the left of T1 array
        - Modify the final row of T1 to show the newest branch
        - Increment aTracker
    - If q and qb don't collide
        - Add q to Q2array

        - Find what branch in T2 has qb as an end of the branch (last column of the matrix) using ismember() Matlab function
        - Add an extra row to T2 at the bottom of the array
        - Add a column of all zeros to the left of T2 array
        - Modify the final row of T2 to show the newest branch
        - Increment bTracker
    - If (q and qa don't collide) AND (q and qb don't collide)
        - Use T1 to find the steps in Q1array from Qstart configuration to the current test configuration
        - Use T2 to find the steps in Q2array from Qend configuration to the current test configuration
        - Set Qfinal as the steps from Qstart to the current test configuration and then from the current test configuration to Qend
- COMMENTED – If desired this can be uncommented – Movement can be performed by running lynxServo() for each of the configurations in Qfinal, with a pause() function in between each step to allow for proper movement specified at the desired speed of the user – COMMENTED

Questions

The workspace itself is never solved for as a whole, as the Rapidly-exploring Random Tree (RRT) strategy never calls for this. The reachable workspace is tested for using the randomQ() function, which only allows for points within the joint limits. The freespace is tested for via the isValidQ(Q,R,P) function, which only allows for configurations Q that adhere to the space created by spherical obstacles with radii R centered at points set in P.

Collisions are checked for using the isCollide(Q1, Q2, R, P) function, which takes configurations Q1 and Q2 and creates a linearly spaced array of Qs that range from Q1 to Q2. By decreasing the size of the step between these two points, an accurate representation of what will collide and what will not collide can be produced. This is accomplished by testing the position of each joint and link at each of these intermediate points with respect to the spherical obstacles themselves, as well as the vertical string above them which hold them up.

The geometry (volume) of the robot is accounted for when setting the radius of the obstacles. In various functions, the input R refers to the explicit radius of the spherical obstacle itself. Although, inside of these functions, wherever necessary a new array titled "Rnew," "R1," etc., is created which increases this radius by 1.5. By increasing the radius that the robot is not allowed to travel in, this takes into account the three-dimensional nature of the robot's joints. Additionally, in order to account for the strings, they are given a rectangular prism geometry. In reality, these strings have an

extremely small radius, but in order to account for when the robot moves from one side of a string to another, this 3D approximation is created.

Waypoints are computed using the freespaceQ(R, P) method. This function, as described above, creates random points that are valid in the freespace, meaning they don't interfere with any of the spherical obstacles or strings.

Waypoints are interpolated between via the RRT method in the pathPlanning(Qstart, Qend, R, P) function. This function begins with a tree from the starting position, as well as a tree from the ending position. While adding test waypoints using the freespaceQ(R, P) method, each tree adds a branch when it contains a node that connects to the new test waypoint. Once both trees reach the same waypoint, a Qfinal path is created which represents how to move the robot along a path from Qstart to Qend.

Evaluation

Design a few tests (environments, start and end positions) to check whether your algorithms work. Explain why you have chosen those tests. Run each test several times to evaluate the performance of your planner. Consider questions such as:

- **How often does your path planner succeed?**
- **How long does it take for your planner to find a path (running time)? MATLAB's in-built functions *tic* and *toc* may be useful here.**
- **When your planner finds a path, is the path the same over multiple runs?**
- **How do the answers to these questions change for different environments or variations in your implementation?**

As an initial test, the freespaceQ(R, P) function was used to create allowable points in the free space for Qstart and Qend. The path planner function was them implemented given these start and end points to see if it would successfully solve for the path between them, how this would vary with different positions, and how long this would take. It is first noted that the R and P arrays were developed from measurements taken on Station 1, and existed as follows:

$$R = [1.5, 1.5, 1.5, 1.5]$$

$$P = transpose([5, -7.75, -3\ ;\ 11, -5.25, 4\ ;\ 4, 3.6, 10.5\ ;\ 12, 3, -7.25\ ])$$

15 tests were performed, five of them will be shown below. For each, Qfinal will be displayed, where the first row is randomly determined Qstart as described above, and Qend will be the final row. The intermediate rows are the steps determined to be taken in between these points. For example, a Qfinal matrix of 3 rows has 1 intermediate step, while a Qfinal matrix of 5 rows has 3 intermediate steps.

Test 1

$$Q_{final} = \begin{bmatrix} 0.6421 & 1.0356 & 0.3943 & -1.286 & 0.0066 & 0 \\ -1.2004 & 0.0717 & 1.1746 & 1.4899 & -1.9846 & 0 \\ -0.0693 & 0.8938 & 0.0577 & -1.1271 & 0.4832 & 0 \end{bmatrix}$$

$$time = .2131\ seconds$$

Test 2

$$Q_{final} = \begin{bmatrix} 0.7184 & 0.9925 & 0.8955 & 1.4274 & -1.6102 & 0 \\ -0.9584 & 0.3613 & 1.4811 & -1.5336 & 1.1499 & 0 \\ -0.7009 & 0.0503 & 1.2829 & -0.9457 & 0.0970 & 0 \\ 0.0918 & -0.2779 & 0.9059 & 1.0977 & 0.5693 & 0 \end{bmatrix}$$

$$time = .540645\ seconds$$

Test 3

$$Q_{final} = \begin{bmatrix} 0.1413 & -0.0889 & -1.2667 & -1.0583 & -0.4342 & 0 \\ -0.1572 & -0.9650 & 0.8053 & -1.7891 & -0.4959 & 0 \\ 0.0324 & 0.2626 & 0.5968 & -1.5859 & 1.0540 & 0 \\ 0.0918 & -0.2779 & 0.9059 & 1.0977 & 0.5693 & 0 \end{bmatrix}$$

$$time = .296746\ seconds$$

Test 4

$$Q_{final} = \begin{bmatrix} 0.8914 & 1.2392 & -1.7989 & 0.2773 & -1.9743 & 0 \\ 0.2739 & 0.9118 & 1.0510 & -1.5959 & 0.5595 & 0 \\ 0.2978 & 0.9234 & 1.1052 & -0.2373 & 0.6625 & 0 \\ -1.1625 & -0.4249 & 1.4100 & -0.3002 & -1.0569 & 0 \\ -0.5235 & -1.0013 & 0.9700 & -0.6577 & 0.0478 & 0 \\ -0.7011 & -0.1936 & -0.3264 & 0.2763 & 0.7564 & 0 \\ -1.1020 & -0.9223 & -0.5151 & -1.0853 & -0.7885 & 0 \end{bmatrix}$$

$$time = .568184\ seconds$$

Test 5

$$Q_{final} = \begin{bmatrix} -0.9612 & -1.0489 & -0.6110 & 0.8784 & -0.6786 & 0 \\ -0.6365 & -0.7573 & 0.5241 & -0.0781 & 0.9019 & 0 \\ 0.9820 & -0.5323 & -0.8008 & 0.7518 & 0.4549 & 0 \end{bmatrix}$$

$$time = .231643\ seconds$$

As mentioned above, 15 total tests were run, although only 5 samples are shown above. These averages of all 15 were found to be:

- Run time: **.339 seconds**
- Intermediate steps: **2.13 configurations**
- Success rate of avoidance of all obstacles and strings: **100%**

Additionally, for test 4, a video was taken of the corresponding movement on the Lynx, and is attached as "Test4.MOV." Screenshots of the starting configuration, each intermediate step, and the ending configuration are shown below to model a plot of what takes place.

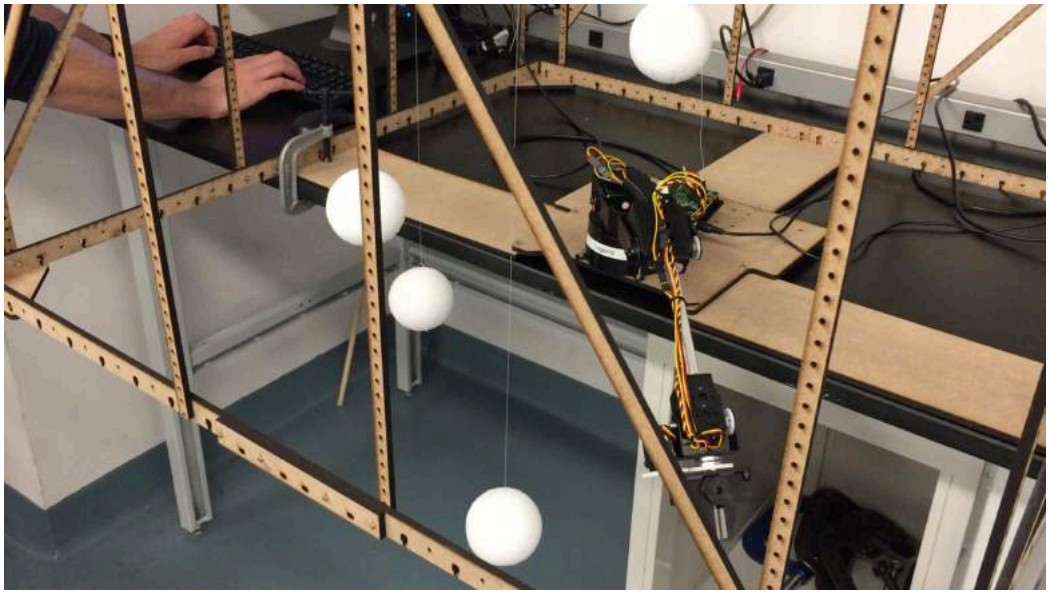

Figure 1 - Test 4 starting configuration 1.

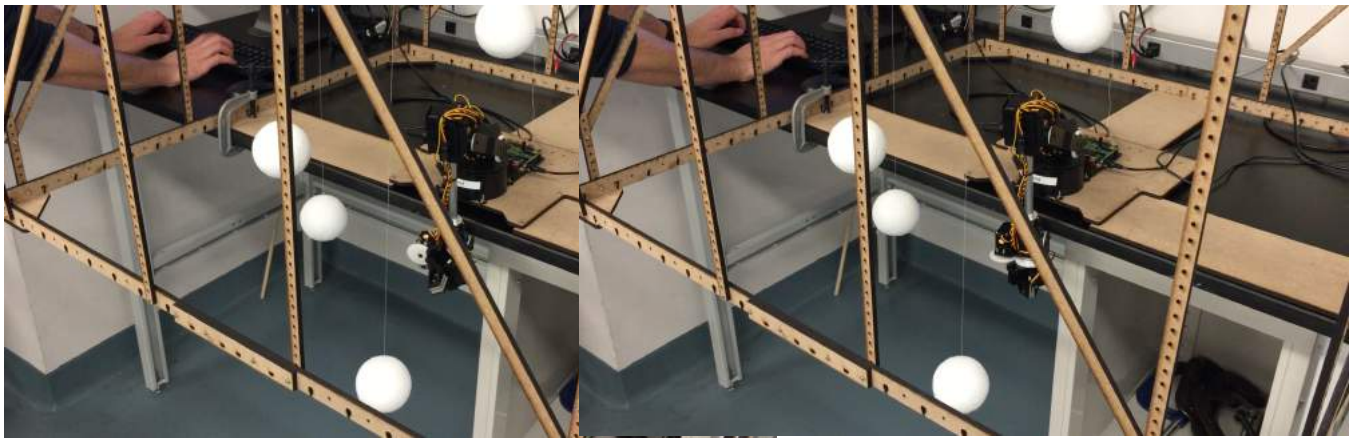

Figure 2 - Test 4 configurations 2 (left) and 3 (right)

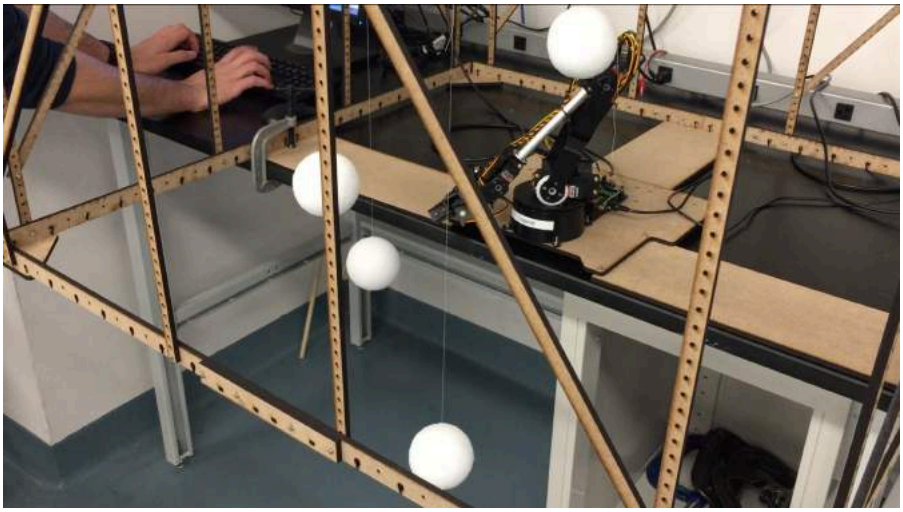

Figure 3 - Test 4 configuration 4.

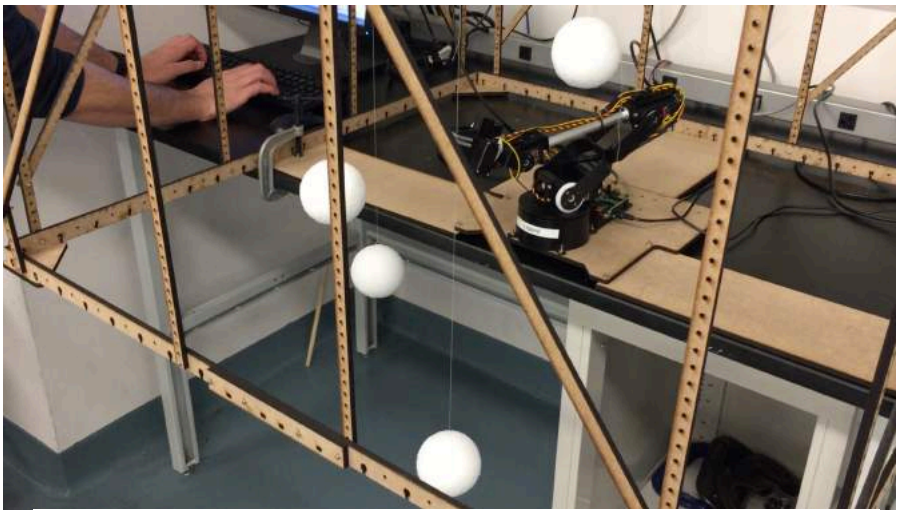

Figure 4 - Test 4 configuration 5.

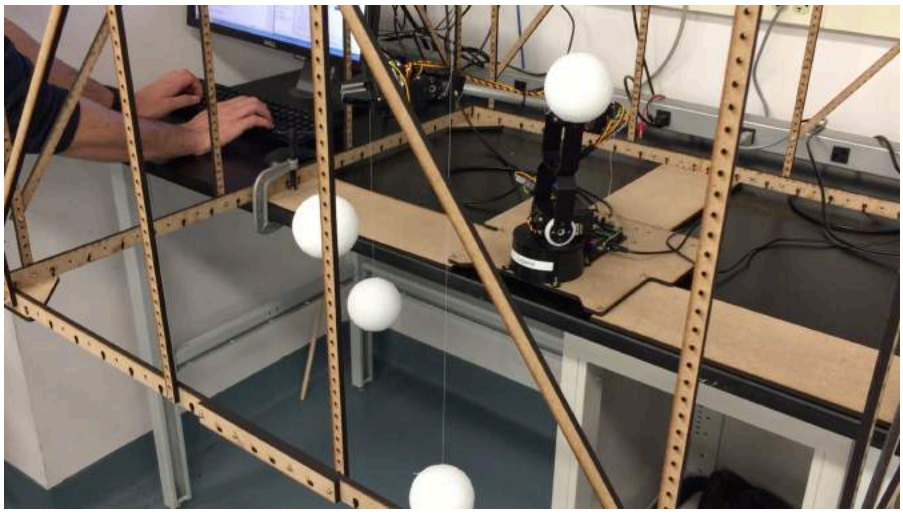

Figure 5 - Test 4 configuration 6.

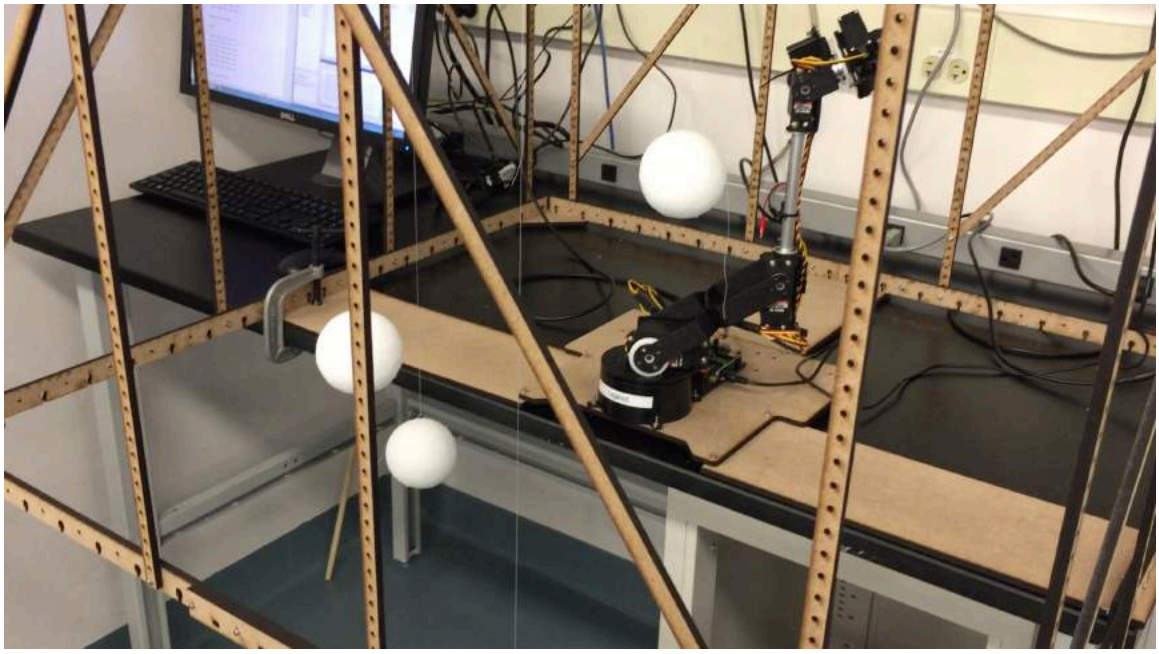

Figure 6 - Test 4 final ending configuration 7.

As can be seen above, the robot is able to move from the starting position to the ending position without hitting any of the obstacles. The attached video provides a view of the continuous motion. Furthermore, a video of test 5 was taken as well, and is attached as a file entitled "Test5.MOV." Screenshots which plot the motion can be seen below. This test was particularly useful because it modeled the robot coming near an obstacle, but passing it without touching it, which can be seen in Figure 9 below.

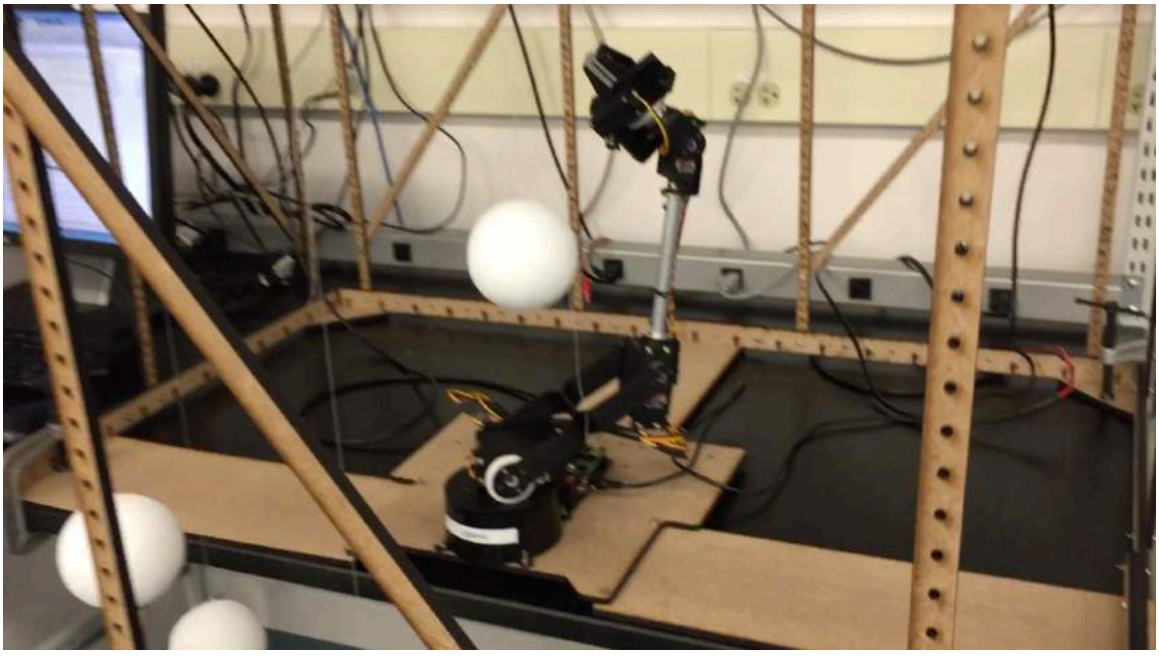

Figure 7 - Test 5 starting configuration 1.

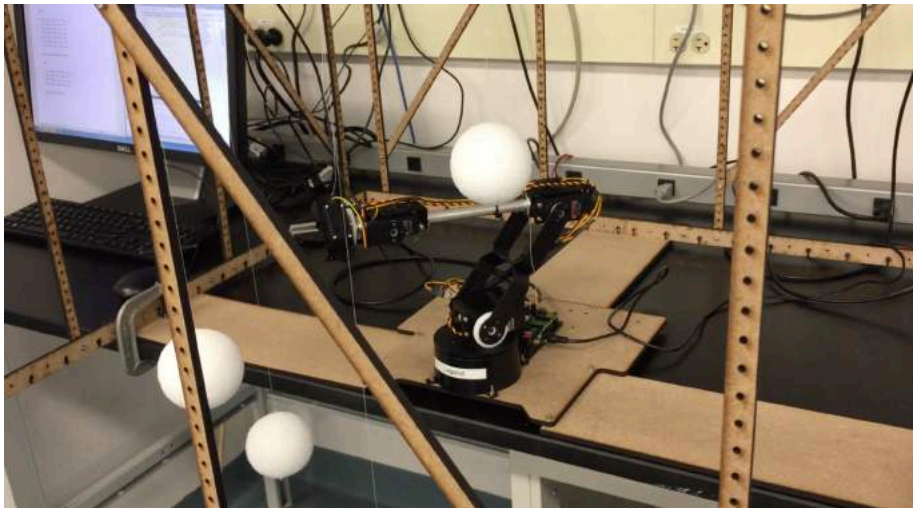

Figure 8 - Test 5 intermediate configuration 2.

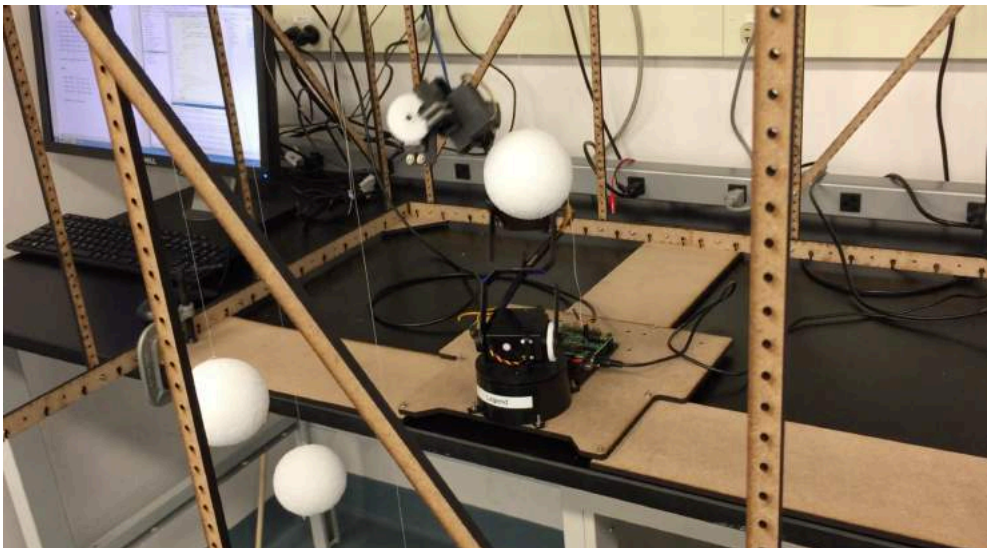

Figure 9 – Test 5 Moving between intermediate configuration 2 (Figure 8) and final configuration (Figure 10). It is seen how close the robot comes to the obstacle, but does not touch it.

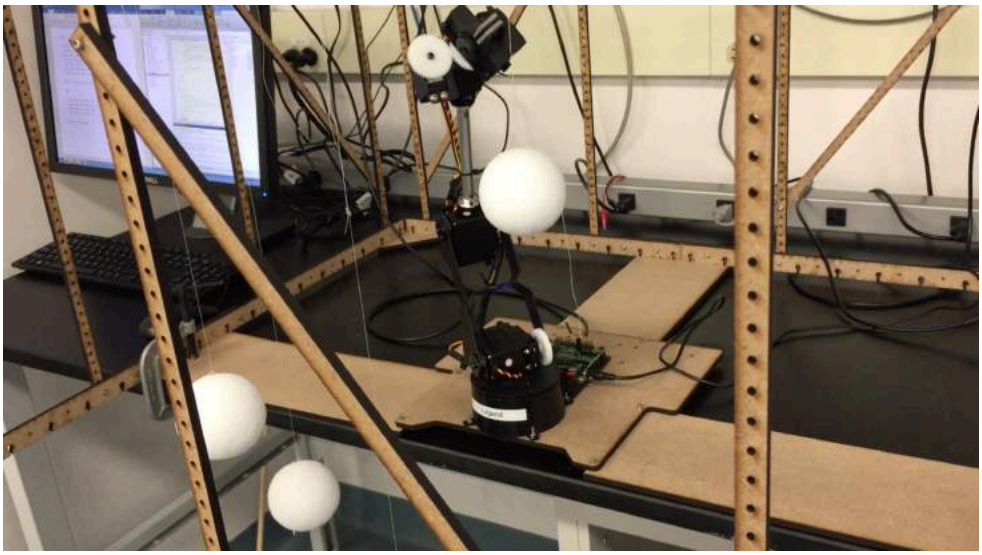

Figure 10 - Test 5 final configuration 3.

Additionally, two start/end configuration combinations were deliberately setup, and tested, as these were especially difficult points to maneuver a path between. If the robot successfully passed these tests, it was decided that the path planning technique was in fact, working properly. First, test 6's Q array is seen below, followed by the start and ending configurations to show the difficult path that needs to be taken.

Test 6

$$Q_{final} = \begin{bmatrix} -0.2000 & 1.0000 & 0 & 0 & 0 & 0 \\ -1.1952 & 0.3466 & 1.2169 & -0.3741 & 0.2840 & 0 \\ -1.3388 & -0.8386 & 0.8933 & 1.3972 & -0.6462 & 0 \\ -1.2178 & 0.1246 & 0.3728 & 0.5941 & -1.1950 & 0 \\ -1.4000 & -0.9000 & -1.0000 & 0 & 0 & 0 \end{bmatrix}$$

$$time = .331106\ seconds$$

Figure 11 - Start configuration of test 6.

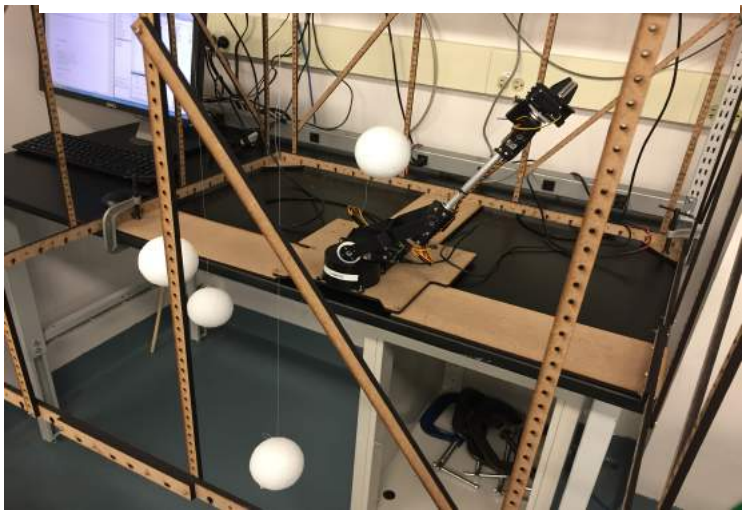

Figure 12 - End configuration of test 6.

Furthermore, test 7's Q array can be seen below, followed by the start and ending configurations to show the difficult path that needs to be taken. Video titled "Test7.MOV" attached shows this path in real time as a video.

$$Q_{final} = \begin{bmatrix} 0.6000 & 0.9000 & -1.0000 & 0 & 0 & 0 \\ .8703 & 1.2151 & 0.5545 & -0.6341 & -0.5801 & 0 \\ -0.1713 & 0.5645 & -0.1723 & 1.3411 & -0.7586 & 0 \\ -0.1309 & -1.0057 & 0.5217 & 0.4924 & 1.2163 & 0 \\ -1.3126 & -0.7131 & 1.5026 & 1.3229 & -0.4146 & 0 \\ -1.4000 & .9000 & -1.0000 & 0 & 0 & 0 \end{bmatrix}$$

$$time = .298198\ seconds$$

Figure 13 – Starting configuration of test 7.

Figure 14 - Ending configuration of test 7.

In summary, the path planner succeeds 100% of the time. This is assuming that the starting and ending positions are in the free space, which is a given of the design requirements. As is the nature of a random path planning strategy, the path taken for a given starting and ending position is not the same each time the code is executed. As seen above, the average run time of the program is only .34 seconds to find an acceptable path. Additionally, as seen by the above tests, various different implementations, such as changes in starting and ending positions, does not change the success of this technique. Modifications to the obstacles were additionally undertaken, and this still resulted in 100% success.

Analysis

Discuss any conclusions on your planner based on the evaluations. What kinds of environments/situations did your planner work well for? What kinds of environments/situations was it bad at? What issues did you run into or what differences were there between your expected and experimental performance? What changes would you make to your implementation if you had more time with the lab?

In conclusions, it is determined that the Rapidly-exploring Random Trees (RRT) strategy works extremely well given the desired requirements of the path planner. 100% success is achieved with regards to clearing obstacles. All environments and situations worked well for the planner, assuming all of the positions were within the robot's configuration space, and the starting and ending positions were within the robot's free space cleared of the spherical obstacles and strings. Occasionally, the robot would run into the table, but this was determined to not be a failed trial as the robot could still continue onto the next step, and the table was not given as a design requirement as an obstacle. Initially, an issue that was run into was that while the robot would clear the obstacles during its paths taken, it would not clear the strings. This was fixed by adding rectangular prisms that model the strings and created them as obstacles. The attached video "StringWithoutPathPlanner.MOV" shows what happened before the modification to the code was implemented, while the attached video "StringWithPathPlanner.MOV" shows the proper path with the current code. Finally, if given more time for the lab, the only change that would be made would be further testing with more difficult obstacles, as the current setup worked 100% of the time.

Code Appendix

```
%% FUNCTION [workspaceQ] = randomQ()
% Outputs Q = [q1, q2, q3, q4, q5, 0] randomly from reachable workspace
% Q is only limited by joint limits
% Q does not care about any obstacles that may be in the workspace

function [workspaceQ] = randomQ()

% Joint Limits
q1Min = -1.4;
q1Max = 1.4;
q2Min = -1.2;
q2Max = 1.4;
q3Min = -1.8;
q3Max = 1.7;
q4Min = -1.9;
q4Max = 1.5;
q5Min = -2;
q5Max = 1.5;

q1 = q1Min + (q1Max-q1Min)*rand();
q2 = q2Min + (q2Max-q2Min)*rand();
q3 = q3Min + (q3Max-q3Min)*rand();
q4 = q4Min + (q4Max-q4Min)*rand();
q5 = q5Min + (q5Max-q5Min)*rand();

workspaceQ = [q1,q2,q3,q4,q5,0];
```

```
%% FUNCTION [tester] = isValidQ(Q)
% Input Q = [q1, q2, q3, q4, q5, 0] which is random in workspace
% Input R = [r1,r2,r3 ...] of each radius of each obstacle
% Input P = [x1,y1,z1; x2,y2,z2; ...] of each position of each obstacle
% Outputs bool that is dependent on freespace
% tester = 1 if valid posiiton in freespace
% tester = 0 if non-valid position in freespace

function [tester] = isValidQ(Q,R,P)

rNew = R + 1.5;

tester = true;      % Assume position is valid, test below to see if not

position = updateQ(Q);          % Create array of 6:3 x,y,z positions

obstacles = length(rNew);          % Amount of obstacles

steps = 100;         % How many steps in each link to check

wide = 1.5;       % How wide is "rectangle" above obstacle for string


% Only test until a bad point is found
while (tester == true)

    %%% Check the joints themselves
    for row = [3,4,6]

        for obstacle = 1:obstacles             % For each obstacle

            flag = 0;       % Set flag to 0

            % Check x of obstacle
            if ((position(row,1) < (P(1,obstacle)+rNew(obstacle))) ...
                    && ((position(row,1) > (P(1,obstacle)-rNew(obstacle)))))
                flag = flag + 1;
            end

            % Check y of obstacle
            if ((position(row,2) < (P(2,obstacle)+rNew(obstacle))) ...
                    && ((position(row,2) > (P(2,obstacle)-rNew(obstacle)))))
                flag = flag + 1;
            end

            % Check z of obstacle
            if ((position(row,3) < (P(3,obstacle)+rNew(obstacle))) ...
                    && ((position(row,3) > (P(3,obstacle)-rNew(obstacle)))))
                flag = flag + 1;
            end

            % If x, y, AND z all are issues, position not valid
            if (flag == 3)
                tester = false;
            end

        end

    end

    %%% Check the links
    for rowL = [3,4,6]

        % Create link position vector
        d = position(rowL,:)-position(rowL-1,:);
```

```
l = d(1);
m = d(2);
n = d(3);

% Find which x is bigger for link
if (position(rowL,1) > position(rowL-1,1))
    max = position(rowL,1);
    min = position(rowL-1,1);
else
    max = position(rowL-1,1);
    min = position(rowL,1);
end

% Create range of x values to iterate through for line
xRange = zeros(1,steps+1);
xRange(1) = min;
for i = 2:(steps+1)
    xRange(i) = xRange(i-1) + (max-min)/steps;
end

% Create corresponding range of y and z
yRange = m.*((xRange-position(rowL,1))./l)+position(rowL,2);
zRange = n.*((xRange-position(rowL,1))./l)+position(rowL,3);

% For each point x,y,z on the link line check obstacle / string
for point = 1:length(xRange)

    % For each obstacle
    for obstacle = 1:obstacles

        % Count if error in x, y, and z
        errorCount = 0;

        % Check x of obstacle
        if ((xRange(point) < P(1,obstacle)+rNew(obstacle)) && ...
                (xRange(point) > P(1,obstacle)-rNew(obstacle)))
            errorCount = errorCount + 1;
        end

        % Check y of obstacle
        if ((yRange(point) < P(2,obstacle)+rNew(obstacle)) && ...
                (yRange(point) > P(2,obstacle)-rNew(obstacle)))
            errorCount = errorCount + 1;
        end

        % Check z of obstacle
        if ((zRange(point) < P(3,obstacle)+rNew(obstacle)) && ...
                (zRange(point) > P(3,obstacle)-rNew(obstacle)))
            errorCount = errorCount + 1;
        end

        % See if all 3 x,y,z failed
        if (errorCount == 3)
            tester = 0;
        end

    end

    % For each "rectangle" string above each obstacle
    for obstacle = 1:obstacles

        % Count if error in x, y, and z
        errorCount = 0;

        % Check x of obstacle
```

```
                if ((xRange(point) < P(1,obstacle)+wide) && ...
                        (xRange(point) > P(1,obstacle)-wide))
                    errorCount = errorCount + 1;
                end

                % Check y of obstacle
                if ((yRange(point) < P(2,obstacle)+wide) && ...
                        (yRange(point) > P(2,obstacle)-wide))
                    errorCount = errorCount + 1;
                end

                % Check z of obstacle
                if (zRange(point) > P(3,obstacle))
                    errorCount = errorCount + 1;
                end

                % See if all 3 x,y,z failed
                if (errorCount == 3)
                    tester = 0;
                end

            end

        end
    end

    break
end
```

```
%% FUNCTION [Q] = freespaceQ(R,P)
% Input R = [r1,r2,r3 ...] of each radius of each obstacle
% Input P = [x1,y1,z1; x2,y2,z2; ...] of each position of each obstacle
% Outputs Q = [q1, q2, q3, q4, q5, 0] randomly from freespace

function [Q] = freespaceQ(R,P)

Q = randomQ();
while (isValidQ(Q,R,P) == 0)
    Q = randomQ();
end
```

```
%% FUNCTION [tester] = isCollide(Q1,Q2,R,P)
% Input Q1 = [q1, q2, q3, q4, q5, 0] which is random point in freespace
% Input Q2 = [q1, q2, q3, q4, q5, 0] which is random point in freespace
% Input R = [r1,r2,r3 ...] of each radius of each obstacle
% Input P = [x1,y1,z1; x2,y2,z2; ...] of each position of each obstacle
% Outputs bool tester based on if collision between parts
% tester = 1 if there is collision
% tester = 0 if there is no collision, THIS IS GOOD

function [tester] = isCollide(Q1,Q2,R,P)

%%% Determine amount of obstacles
obstacles = length(R);

%%% Determine modified radius
R1 = R+1.5;

%%% Create a Q array
dt = 100;                   % steps between angles
qArray = zeros(dt+1,6);     % create array for Qs
qArray(1,:) = Q1;
for column = 1:5
    if (Q1(column) > Q2(column))
        max = 1;
    else
        max = 2;
    end

    if (max == 1)
        dif = (Q1(column) - Q2(column))/dt;
        for i= 2:(dt+1)
            qArray(i,column) = qArray(i-1,column) - dif;
        end
    else
        dif = (Q2(column) - Q1(column))/dt;
        for i= 2:(dt+1)
            qArray(i,column) = qArray(i-1,column) + dif;
        end
    end
end

%%% Check if each Q array produces a valid position in workspace
% If test1 = 0, then can stop performing all tests, FAILED
% If test1 = 1, then so far valid, continue testing
flag = 0;
while (flag == 0)
    for i = 1:length(qArray)
        test1 = isValidQ(qArray(i,:),R,P);
        if (test1 == 0)
            flag = 1;
        end
    end
    flag = 1;
end

% Set overall tester
if (test1 == 0)
    tester = 1;
else
    tester = 0;
end

%%% Check links
% How many steps in each link to check
steps = 100;
```

```
% How wide is "rectangle" above obstacle for string
wide = 1.5;

% If tester = 1, or there is a collision, STOP TESTING
while (tester ~= 1)

    % For each row in array of Q=q1:q5 to get from Qstart to Qend
    for rowq = 1:length(qArray)

        % Set of joint positions for given Q=q1:q5
        positionSet = updateQ(qArray(rowq,:));
        % For each link
        for rowp = [3,4,6]

            % Create link position vector
            d = positionSet(rowp,:)-positionSet(rowp-1,:);
            l = d(1);
            m = d(2);
            n = d(3);

            % Find which x is bigger for link
            if (positionSet(rowp,1) > positionSet(rowp-1,1))
                max = positionSet(rowp,1);
                min = positionSet(rowp-1,1);
            else
                max = positionSet(rowp-1,1);
                min = positionSet(rowp,1);
            end

            % Create range of x values to iterate through for line
            xRange = zeros(1,steps+1);
            xRange(1) = min;
            for i = 2:(steps+1)
                xRange(i) = xRange(i-1) + (max-min)/steps;
            end

            % Create corresponding range of y and z
            yRange = m.*((xRange-positionSet(rowp,1))./l)+positionSet(rowp,2);
            zRange = n.*((xRange-positionSet(rowp,1))./l)+positionSet(rowp,3);

            % For each point x,y,z on the link line check obstacle
            for point = 1:length(xRange)

                % For each obstacle
                for obstacle = 1:obstacles

                    % Count if error in x, y, and z
                    errorCount = 0;

                    % Check x of obstacle
                    if ((xRange(point) < P(1,obstacle)+R1(obstacle)) && ...
                            (xRange(point) > P(1,obstacle)-R1(obstacle)))
                        errorCount = errorCount + 1;
                    end

                    % Check y of obstacle
                    if ((yRange(point) < P(2,obstacle)+R1(obstacle)) && ...
                            (yRange(point) > P(2,obstacle)-R1(obstacle)))
                        errorCount = errorCount + 1;
                    end

                    % Check z of obstacle
                    if ((zRange(point) < P(3,obstacle)+R1(obstacle)) && ...
                            (zRange(point) > P(3,obstacle)-R1(obstacle)))
```

```
                    errorCount = errorCount + 1;
                end

                % See if all 3 x,y,z failed
                if (errorCount == 3)
                    tester = 1;
                end

            end

            % For each "rectangle" string above each obstacle
            for obstacle = 1:obstacles

                % Count if error in x, y, and z
                errorCount = 0;

                % Check x of obstacle
                if ((xRange(point) < P(1,obstacle)+wide) && ...
                        (xRange(point) > P(1,obstacle)-wide))
                    errorCount = errorCount + 1;
                end

                % Check y of obstacle
                if ((yRange(point) < P(2,obstacle)+wide) && ...
                        (yRange(point) > P(2,obstacle)-wide))
                    errorCount = errorCount + 1;
                end

                % Check z of obstacle
                if (zRange(point) > P(3,obstacle))
                    errorCount = errorCount + 1;
                end

                % See if all 3 x,y,z failed
                if (errorCount == 3)
                    tester = 1;
                end
            end

        end
    end
end

    break
end
```

```
%% FUNCTION row = closestNode(Qnew,Qarray)
% Input Qnew = [q1, q2, q3, q4, q5, 0]
% Input Qarray = [q1, q2, q3, q4, q5, 0; ...]
% Outputs row of which Q in Qarray is closest to Qnew

function row = closestNode(Qnew,Qarray)

% Find position of end effector of Qnew
positionQnew = updateQ(Qnew);
endQnew = positionQnew(6,:);

% Assume closest is first Q in Qarray, and then test the rest
position1 = updateQ(Qarray(1,:));
endPosition1 = position1(6,:);
d = ((endQnew(1)-endPosition1(1))^2 + (endQnew(2)-endPosition1(2))^2 ...
    + (endQnew(3)-endPosition1(3))^2)^.5;
row = 1;

% For each Q in Qarray test if closer to Qnew
for i = 2:size(Qarray,1)
    position = updateQ(Qarray(i,:));
    endPosition = position(6,:);
    d1 = ((endQnew(1)-endPosition(1))^2 + (endQnew(2)-endPosition(2))^2 ...
    + (endQnew(3)-endPosition(3))^2)^.5;
    if (d1 < d)
        row = i;
        d = d1;
    end
end
```

```
%% FUNCTION Qfinal = pathPlanning(Qstart,Qend,R,P)
% Input Qstart = [q1, q2, q3, q4, q5, 0] to start at
% Input Qend = [q1, q2, q3, q4, q5, 0] to end at
% Input R = [r1,r2,r3 ...] of each radius of each obstacle
% Input P = [x1,y1,z1; x2,y2,z2; ...] of each position of each obstacle
% Output Qfinal = set of Qs to get from Qstart to Qend
function Qfinal = pathPlanning(Qstart,Qend,R,P)

% Array of Qs from Qstart and corresponding tree
Q1array = Qstart;
T1 = [1];
aTracker = 2;

% Array of Qs from Qend and corresponding tree
Q2array = Qend;
T2 = [1];
bTracker = 2;

% Final array of Qs
Qfinal = Qstart;

% Flag to check if path is found
flag = 1;
% Go until a full path is found
while flag == 1

    % Random configuration in freespace
    q = freespaceQ(R,P);

    % Closest node row in Q1 array
    rowQ1array = closestNode(q,Q1array);
    % Associated row of Q
    qa = Q1array(rowQ1array,:);

    % If q and qa don't collide, add this to tree 1
    if (isCollide(q,qa,R,P) == 0)

        % Add new Q to Q1array
        Q1array = [Q1array ; q];

        % See where in tree previous point exists
        memberTest = ismember(T1,[rowQ1array]);

        % Find which row in the tree ends in corresopnding node
        % If none do, then "selected" remains 0, need to make new row
        selected = 0;
        for row = 1:size(memberTest,1)
            if (memberTest(row,length(memberTest)) == 1)
                selected = row;
            end
        end

        % Add extra row for new branch
        T1 = [T1 ; T1(selected,:)];

        % Add a column of zeros at left
        T1 = [zeros(size(T1,1),1) T1];

        % Create new branch in final column
        T1(size(T1,1),:) = [T1(size(T1,1),2:length(T1)) , aTracker];

        % Increment aTracker
        aTracker = aTracker + 1;

    end
```

```
% Closest node row in Q2 array
rowQ2array = closestNode(q,Q2array);
% Associated row of Q
qb = Q2array(rowQ2array,:);

% If q and qb don't collide, add this to tree 2
if (isCollide(q,qb,R,P) == 0)

    % Add new Q to Q2array
    Q2array = [Q2array ; q];

    % See where in tree previous point exists
    memberTest2 = ismember(T2,[rowQ2array]);

    % Find which row in the tree ends in corresopnding node
    % If none do, then "selected" remains 0, need to make new row
    selected = 0;
    for row = 1:size(memberTest2,1)
        if (memberTest2(row,length(memberTest2)) == 1)
            selected = row;
        end
    end

    % Add extra row for new branch
    T2 = [T2 ; T2(selected,:)];

    % Add a column of zeros at left
    T2 = [zeros(size(T2,1),1) T2];

    % Create new branch in final column
    T2(size(T2,1),:) = [T2(size(T2,1),2:length(T2)) , bTracker];

    % Increment aTracker
    bTracker = bTracker + 1;

end

% If both sides connect
if ((isCollide(q,qa,R,P) == 0) && (isCollide(q,qb,R,P) == 0))

    % See steps for start to connector
    steps1 = T1(aTracker-1,:);
    spot1 = 1;
    flag11 = 0;
    while (flag11 == 0)
        for stepper1 = 1:length(steps1)
            if (steps1(stepper1) == 0)
                spot1 = spot1 + 1;
            else
                flag11 = 1;
            end
        end
    end
    steps1 = steps1(spot1:length(steps1));

    % See steps from end to connector
    steps2 = T2(bTracker-1,:);
    spot2 = 1;
    flag12 = 0;
    while (flag12 == 0)
        for stepper2 = 1:length(steps2)
```

```
                if (steps2(stepper2) == 0)
                    spot2 = spot2 + 1;
                else
                    flag12 = 1;
                end
            end
        end
        steps2 = steps2(spot2:length(steps2));

        % Add necessary Qs from beginning to end
        for i = 2:(length(steps1)-1)
            Qfinal = [Qfinal ; Q1array(steps1(i),:)];
        end
        for j = length(steps2):-1:1
            Qfinal = [Qfinal ; Q2array(steps2(j),:)];
        end

        % Stop the while loop
        flag = 0;

    end

end

%{
pause on

% Peform movement
for k = 1:size(Qfinal,1)
    lynxServo(Qfinal(k,1),Qfinal(k,2),Qfinal(k,3),Qfinal(k,4), ...
        Qfinal(k,5),Qfinal(k,6));
    pause(2);
    Qfinal(k,:)
end
%}
```