Phil Hu and Rajat Bhageria
MEAM 520
16 October 2017

Project 3 Lab Report

1. Method:

1.1. Find the workspace and the free c-space:

When computing the workspace, we just remembered that the lynxServo() function in MATLAB tells us the minimum and maximum possible values that each theta value can take. Knowing this, we designed a series of for loops that loop over each of the theta values that the robot can take.

1.1.1. General loop for finding c-space and accounting for fact that robot is multi-link robot

The general loop:

- For range theta1
    - For range theta2
        - For range theta3
            - For range theta4
                - Using theta1, theta2, theta3, and theta4, calculate the x,y,z positions of each of the four joints using FK.
                - Interpolate an array of points between each of the respective joints (i.e., find all x,y,z points between joint i and i+1).
                - If none of the x,y,z s in both the positions of joints AND the lines between the joints *isWithinObstacle*(x,y,z), Add (theta1, theta2, theta3, theta4) to the C-space space matrix and add (x,y,z) to the workspace matrix. Note that we're doing this for every possible theta
                - Note that the *isWithinObstacle* function is discussed in 1.1.1.

Note that this loop takes into consideration the fact that this robot isn't just represented by an end-effector but rather is a multi-link robot with links connecting the joints. The interpolation between joints i and i+1 allows us to ensure that the robot doesn't bump into anything along the rigid links (as well as of course the joints themselves).

1.1.1. Checking for obstacles and accounting for the geometry of the robot:

We account for the geometry of the robot by still treating each of the links as lines and each of the joints as points; however, we add radius = 35mm to each of the obstacle. This essentially allows us to continue treating our robot as a series of points and lines but and just make our obstacles bigger by the radius of the robot.

When we're checking for obstacles, we're essentially passing in a set of x,y,z positions and testing whether they collide with either the spherical obstacles or the strings connecting them to the roof of the robot cage.

In particular, we decided that we would be okay with estimating our spheres as cubes of length = 2*radius of the spheres. Additionally, since there is a string connecting the spheres to the top of the cage, we essentially decided to represent the obstacles as rectangular prisms from from the lowest point of the sphere to the top of the robot cage. This prevents us from colliding with any string or cube.

Here is the collision detection function we wrote:

isWithinObstacle(x,y,z):

- The radius of the robot is 35mm at any arbitrary point. Thus we add w=35mm to our obstacle radius
- Return true if:
    - newRad = r+w
    - (xcenter-newRad)<x<(xcenter+newRad) &&
    - (ycenter-newRad)<y<(ycenter+newRad) &&
    - Z >= (zcenter-newRad) //includes the volume not possible bec string

1.2. Using the C-space points, find a graph of the free C-space with nodes and edges:

At this point we have two matrices: a 20kx6 matrix of the free configuration space and a 20kx3 matrix of the free workspace. It's important to note that in our implementation, these matrices are indexed to each other. In other words, given an arbitrary workspace position in cartesian coordinates i, we can find the free c-space matrix at position i as well.

Now our goal is to find a graph that we can use to find the shortest path between p_start and p_end. The basic idea behind our algorithm is to loop through all the possible values in our free configuration space / free workspace and then see which ones it makes sense to create an edge between.

Here's the general algorithm we used to create an adjacency matrix to represent our graph (note that it's an NxN graph where number of nodes we have in our free-configuration space)

- N = the number of points/nodes we have in our workspace
- For row = 1:n //we're looping over all the possible workspace values
    - For col = 1:n // loop again
        - Node i = index row
        - Node j = index col
        - Find a weight to assign to an edge between Node i and j

        - Find the Euclidean distance between node i and node j. We find this by referencing these indexes in our workspace and finding the distance between them.
        - Find the index distance between node i and j. The idea here is that we want to push the algorithm to choose edges that are close together in our matrix; since we used a series of for loops, the consecutive configuration values are very close but for example the configuration value of index 30 is going to be very different than index 3000. Thus we're going to use this metric in determining a weight to assign to the edge between Node i and j.
        - Now normalize between the euclidean distances and the the index distance. This is because the euclidean distance is a number between 0 and 1000mm in our case and the index distance is a number between 0 and 20,000 (20,000 nodes in our free c-space).
        - Then, assign a weight to the euclidean distance a weight to the index distance. We had to spend quite a lot of time doing parameter tuning to make sure that the robot had a good balance between finding consecutive values in the free c-space to go to and also between ensuring that the robot chose to go to configurations that are relatively close to each other in the c-space.
        - If the weight for the edge between Node i and j is less than a value, add the weight as an entry in the table A that is our adjacency matrix. The idea here is that since we're looping over all the possible two nodes that can be connected in the graph, we want to make sure that we assign a cost of 0 to the nodes that have a huge cost between them; i.e., if node i is really far from node j in terms of distance or in terms of indexes, we want to make sure that the robot doesn't go from node i to node j.
    - Now that our adjacency matrix is done, pass it into the graph function on MATLAB and we have our graph.

1.3. Using the p_start, p_end, and the graph, find the shortest path:

The first step we took was to find the node qa in the graph that's closest to p_start in terms of Euclidean distances. We did this by looping through all the nodes in our graph and finding the one that minimized the Euclidean distance to p_start in the workspace. It's important to once again remember that for any node, we know both its cartesian coordinates and its configuration since our free c-space matrix and our free workspace matrix are indexed to the same nodes.

We did the same to find qb, the closest configuration to p_start (in terms of euclidean distances in the workspace).

Then, we use the MATLAB shortestPath function to find the shortest path from qa to qb. This function returns us a path with indexes. Now we can convert this mx1 matrix of indexes to a series of configurations that the robot must take during the trajectory planning.

Ultimately, we decided that in this particular case, it wouldn't be terribly useful to find the path from p_start to qa and from qb to p_end. This is simply because when we were constructing the workspaces and the free c-space initially, our for loops had quite a strong resolution and thus our point cloud when viewing the c-space is quite dense. What this means is that for any arbitrary point that may be q_start/q_end, there are two cases: 1) since our c-space is so dense, we have a node in our 20k height matrix that already contains a point that is extremely close to that or exactly that point or 2), if the q_start/q_end points are outside the workspace completely, our distance minimizing algorithm will find the configuration that will minimize the Euclidean distance between the q_start/q_end and the end effector thus once more giving us a point that's the closest possible point to q_start/q_end in the workspace.

Now we have a series of configurations that the robot can take to path plan without running into any obstacles.

1.4. Tradeoffs we Made:

We had to make many tradeoffs while we were working on this project. Here are some:

- Size/resolution of the graph vs the computational complexity. As described earlier we have a series of for loops that allow us to compute each possible configuration. We had to make a tradeoff between the resolution of thetas we choose (i.e., increment very small amounts each time) and the time it takes for this $O(n^4)$ algorithm to run. Ultimately, we decided to make a compromise and have high resolution for theta1 (the base) and not as high for the other thetas since we wanted to make sure that we could reach any arbitrary x and y in the xy plane (theta1 is the only joint that determines the angle on the xy plane that the robot lies).
- We had to make a compromise between using the c-space and the workspace. We knew we preferred using the c-space to the workspace since we would rather not have to use IK (especially since there could be multiple solutions). However, we also knew that in some cases, we would need to find the cartesian distance between two nodes in our graph (i.e., finding the closest qa to p_start or ensuring that nodes that are above a certain cartesian distance away are not connected with an edge so that the robot has a 0% probability of following that path. Ultimately, we ended up using a combination of both the free c-space and the free workspace and ensuring that we have indexing between them so that for any arbitrary node we can find the configuration and the x,y,z position of the end effector.
- Having an optimal solution vs. using a probabilistic finder. Initially we were spending a lot of time trying to write pseudocode that used a probabilistic PRM and then just used the first path it found (maybe after some path trimming). However, ultimately we decided that actually finding the free c-space and then finding a graph with the max number of points we can get (while ensuring that the program doesn't run forever) and then finding the optimal path using

the shortest path algorithms would be best since it would allow us to find a more optimal solution.
- Calculating the weights. We discuss calculating the weights and parameter turning further in the evaluation section.
- Whether to use the simulation or the the physical robot to test.

2. Evaluation:

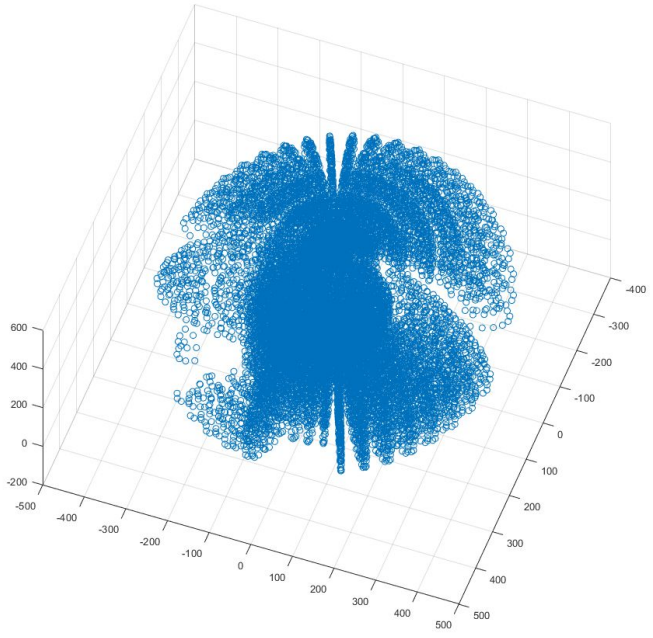
We decided to test our robot in the simulation rather than in the real setup. This is simply because we wanted to simply (at least for now) avoid some of the physical limitations of the real world:

- The robot will sag and we have to account for that
- There is a table that the could bump into the robot.

There were many problems we had when we initially started testing:

- Our end effector was indeed following a smooth path from p_start to p_end but the joint angles were shifting dramatically (i.e., theta3 going from -1 to 1 from one frame to the next). Thus we had to go back and change the weights we were assigning to each of the values in our A adjacency matrix.
- But when we added this cost in, we noticed that this cost had a significantly higher values than the Euclidean distance so we had to normalize the weights
- Then for 2-3 hours we had to do parameter tuning to make sure that we were weighting each of the costs in the right way (i.e.., put 40% cost to the Euclidean distance and 60% cost to index cost or weigh them equally?).
- Then we had to take care of a situation where we saw that the robot joints were moving from configuration to the exact opposite configuration. We realized that this was because the robot was indifferent between the path it took as long as it got to same end frame correctly. Thus, we decided that if a particular node i and i+1 have a cost greater than a parameter we had to tune, we're not even going to add it as an edge.
- Whether or not to implement trajectory smoothing. Described below in depth.

Here's a picture of our robot free c-space:



Here's the code we used to test our robot in the simulation:

```
function [path] =
testTrajectoryPlanning(posOfObstacles,ri,p_start,p_end,G)
    %find the free c-s
    [c_space,workspace] = freeCSpaceCalc(posOfObstacles, ri);

    %create the graph (maintain indexes of c-space)
    G = createGraph(workspace);

    %find the shortest path between pstart and pend
    path = findShortestPath(G,c_space,workspace,p_start,p_end);

    lynxStart();

    %simulate the movement of the robot by looping through path
    for time = 1:size(path,1)
    config = path(time,:);
```

```
      lynxServoSim(config(1,1),config(1,2),config(1,3),config(1,4),co
      nfig(1,5),config(1,6));
     pause(1);

    end

end
```

Note that every time we are pausing 1 second on our simulation to make sure that we can see the robot end configuration moving.

Trajectory smoothing: Deciding whether or not to do path trajectory smoothing was definitely a challenge. We knew that in the real Lynx already had in built trajectory smoothing algorithms. Thus we decided not to implement path trajectory smoothing. This meant however that between frames in our simulation, the robot was jumping from point to point in between the different configurations. We decided this was a non-issue since in the real Lynx this jerkiness wouldn't be present (and we'd rather avoid problems like sagging in the real robots than deal with a non-perfect trajectory smoothing in the simulation.

Time complexity: In terms of timing, the main time consumption was in finding the graph. This is because we were essentially constructing and populating a 20,000x20,000 sparse adjacency matrix. This took a long time (45+ seconds) but since it was a fixed cost we decided that we were okay giving up time in order to get more resolution. The optimal path finding was very fast and was essentially very small relative to finding the graph itself.

Success in each trial : Our path planner did indeed succeed each time and didn't run into any obstacles. This is simply because were were always only testing in the free configuration and free workspace and also because we spent a large amount of time earlier while writing pseudocode ensuring that the robot's geometry was taken care of and ensuring that the obstacles and the strings were taken care of.

Replication of results: Since we decided not to implement a PRM and instead chose to use an approach that found the optimal path upfront, we had a matrix of configurations that the robot should go through to get to the end point and thus our results were 100% replicable given a particular q_start and q_end.

3. Analysis:

1. Our planner worked particularly well for widely located points but was poor in practice, as we had not previously accounted for the optimization in the configuration space and rather just focused on the workspace. As a result in our final path simulations, the end effector moved incredibly smoothly from point A to point B but the configurations of the robot were incredibly jerky and lacking smoothness in joint movement.

2. A fairly large difference between the simulations and the testing on robot 1 was the level of sagging apparent in the physical model. Where we had expected the robot to be very close to the obstacle, we saw very large margins. Thankfully our modeling of the obstacles as a continuation of the obstacle itself above the (center points - margin) allowed us to avoid the issue of colliding or getting tangled with the string.
3. If we had more time with the lab we would would experiment with more optimal cost functions to enable smoother movements for the robot from node to node along the path from pointA to pointB. We also observed a number of other teams having a lot of success with probabilistic models and would have liked to incorporate that into our method. The biggest thing was smoothing out the jerky joint movement by optimizing for a path within the configuration space rather than within the workspace, which would be made easier if we had also vectorized and optimized for the performance of our code.