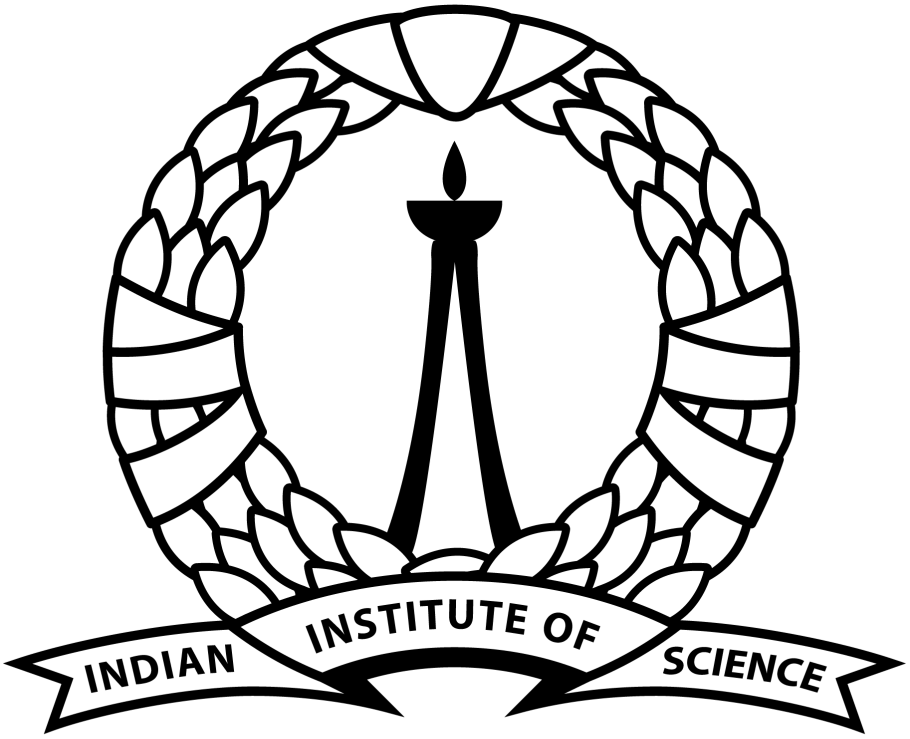

भारतीय विज्ञान संस्थान

CP 230

Motion Planning for Autonomous Systems - Lab Assignment

Designed by Mukunda Bharatheesha

Instructors:
Prof. Debasish Ghose
Dr. Mukunda Bharatheesha
Suryadeep Nath
Amit Kumar

1 PART I: Path Planning with A* for a two-arm manipulator

Run the `assignment5.m` script so that the two-arm manipulator parameters are initialized. The two joint angles are denoted by ϕ_1 and ϕ_2 and they are measured with respect to the vertical axis and positive in the counter clockwise direction.

ϕ_1 and ϕ_2 denote the two joint angles respectively, such that:

$$-0.8 \leq \phi_1 \leq 0.8$$
$$-1 \leq \phi_2 \leq 1$$

The complete configuration of the robot is denoted by $q = [\phi_1, \phi_2]$.

1.1 Setting up problem specific data structures

1. Define the initial and goal configurations for the manipulator as:

$$q_{init} = [-0.8, 1]$$
$$q_{goal} = [0.8, -1]$$

Use the `body_animate.m` file provided with the assignment and visualize the initial configuration. A typical configuration looks like Fig. 1.

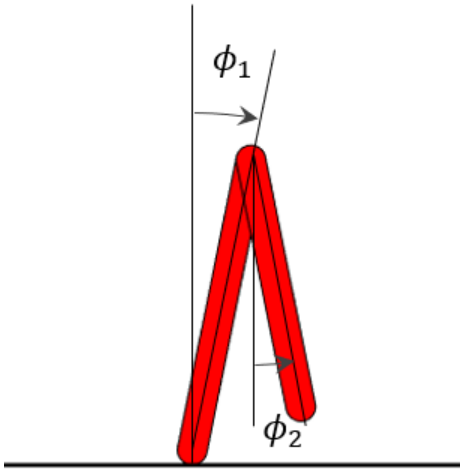

Figure 1: A typical configuration.

2. Create a 3 × 3 C-space grid for the two joints within the range of values they can take:

- Complete the function in the `discretizeJointPositions.m` file which will be called from the `assignment5.m` script as follows:

$[\phi_1, \phi_2] =$
`discretizeJointPositions`([joint 1 range], [joint 2 range], gridSize);

- Use the `createGrid.m` function provided with the assignment files to generate the grid coordinates. It can be called from `assignment5.m` script as follows:

[gridCoords] = `createGrid`([joint 1 values, joint 2 values]);

3. Complete the function `getDistances.m` which will be called from the `assignment5.m` script as follows. This function will create a distance matrix for the grid coordinates.

$$[\text{distanceMatrix}] = \texttt{getDistances}(\text{gridCoords});$$

4. Check the correctness of the generated distance matrix using `checkGrid.m`. It is important that this check is successful for the rest of the assignment.

1.2 Implementing the A* algorithm

1. Complete the function `createAstarNodes.m` that will generate a *node* data structure for each grid point which contains the following elements:

   - ID: A positive integer identifier for each node (this will be useful later for the A* search.)
   - Coordinates: Vector containing the C-Space coordinates.
   - nextDoorNeighbors: An array containing the ID list of $N4$ neighbors of this node.
   - parent: Node parent.

   This function should be called from the `assignment5.m` script as follows:

$$[\text{nodesList}] = \texttt{createAstarNodes}(\text{numGridSteps, gridCoords});$$

2. You will implement the A* algorithm in the `Astar.m` file that is provided with the assignment. You can use the following guidelines for the implementation.

   - First, implement a nearest neighbor algorithm to find the $N4$ nearest neighbors (or the next door neighbors) in the file `findNextDoorNeighbors.m`. Before you proceed with the implementation, answer the questions below.You can see that there is a certain pattern given in the file. Observe this pattern carefully and answer the following questions:
     - Is this the only pattern possible? Explain your answer in a few sentences.
     - Is there a relation between the ordering of the node IDs in the *nodeMatrix* and the Distance Matrix you computed before?

3. Use the hints provided in the `Astar.m` file to complete the A* algorithm implementation. There is also a Pseudo code of the A* algorithm provided at the end of `Astar.m` file.

4. Use your implementation to find the path between the start and the goal nodes that you defined earlier. Create a visualization of the path using the `body_animate.m` file by generating a trajectory via linear interpolation between the generated trajectory points. Remember that the trajectory points are "C-Space" coordinates and not just node numbers!

1.3 A* search with obstacles

In this part, you will be modifying your A* algorithm to plan a path for the robot arm such that it can reach a ball which is placed inside a bin in front of the robot as shown in Fig. 2.

For this purpose, you will first enable the obstacle visualization by setting `enable_obstacle` parameter to `true` in `assignment6.m`. The bin (represented by two vertical lines) is 0.43 m wide and 0.39 m tall. The location of the bin with reference to the robot's first arm (that is fixed to ground) is as shown in Fig. 2. It is required that the "end-effector" (tip of the arm) maintains a minimum clearance of $0.1\,m$ from any part of the bin. (The ball is only a visualization and NOT an obstacle). For this part of the assignment, you will be working with a grid of size 15×15.

1. The goal location at which the robot arm can reach the ball is given by

$$q_{goal} = [-0.8,\ 0.0]$$

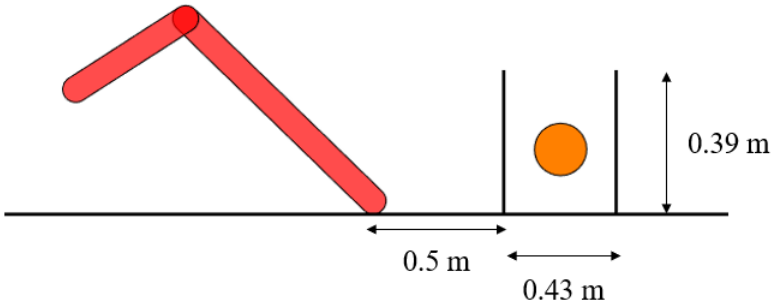

Figure 2: Bin obstacle with the ball.

2. Introduce "C-Space Obstacles" on the grid to represent the bin containing the ball:

   - Implement a function `getEndEffectorPositions.m` that computes the workspace coordinate(s) (x, y) corresponding to a specified joint angle pair(s). This function is called from the `assignment6.m` script as follows:

     [eef_coords] = `getEndEffectorPositions`(par, `gridCoords`);

   - There are multiple ways in which the workspace obstacle (that is, the bin) can be represented in C-space. In this assignment, you will use the concept of discretizing the workspace obstacle. You will first discretize the obstacle in 2 steps by implementing a function `discretizeObstacle.m` which is called from the `assignment6.m` script as follows:

     [obs_coords] =
     `discretizeObstacle`(obstacle range, obs_discretization_steps);

     This function will return the coordinates of the discretized obstacle in the workspace.

   - Implement a function `findObstacles.m` that computes all C-space grid indices that would not only cause a collision of the end-effector with the bin but also have the end-effector violate the minimum clearance constraint. This function is called from `assignment6.m` as follows:

     [obstacle_ids] = `findObstacles`(eef_coords, obs_coords);

   - What is the limitation of using C-Space obstacles?.

3. Modify the main algorithm in `Astar.m` to now include the check for avoiding obstacles.

4. Plan a path using the newly modified `Astar.m` to the goal location.

5. Create a visualization of the path using the `body_animate.m` file. What do you observe and why?

6. Now discretize the obstacle in 3 steps and repeat steps 2 to 5. What do you observe and Why?

2 PART II: Path Planning with RRT

This part of the assignment involves implementing the RRT algorithm to find a path for the robot arm between the same q_{init} and q_{goal} configurations that avoids collisions with the bin. The joint angles from the previous part will be used for this assignment.

2.1 Setting up necessary data structures for RRT

1. Define the initial and goal configurations for the robot as:

$$q_{init} = [0.8, -1.0]$$
$$q_{goal} = [-0.8, 0.0]$$

   Initialize all the robot parameters by running the `assignment7.m` script.

2. Set MAX_NODES parameter to 1000 and num_path_steps to 10.

3. You will incrementally generate and maintain a *node* data structure for each tree node which contains the following elements. The list containing all tree nodes will be called treeNodes:

   - parent: A positive integer identifier of the parent node.
   - coord: Vector containing the C-Space coordinates of the tree node.

2.2 Implementing the RRT algorithm

1. Complete the RRT algorithm in `rrt.m` using the hints provided. You will be using some of the functions you implemented for Part1 of the assignment and writing some new functions. The functions that will be re-used are:

   - `getEndEffectorPositions`
   - `discretizeObstacle`
   - `findObstacles`

   .

   The new functions you will be implementing are:

   - `sampleRandomNode.m` which will sample a uniformly random value for ϕ_1 and ϕ_2 in their respective range. This function will be called from `rrt.m` as:

     [randNode] =
     `sampleRandomNode`([{$\phi_1$range}], [{$\phi_2$range}]);

   - `findNearestNeighbor.m` which will be called from `rrt.m` as:

     [nnId] =
     `findNearestNeighbor`(treeNodes, randNode);

   - `computePath.m` which will compute a linear path between the nearest neighbor and the random node with a specified number of steps. This function will be called from `rrt.m` as:

     [pathToRandNode] =
     `computePath`(neighbor, randNode, num_path_steps);

   - `goalReached.m` which will check if a newly added tree node is within a certain tolerance of the goal node. This function will be called from `rrt.m` as:

     [success] =
     `goalReached`(newNode, goalNode, goal_tolerance);

     For this assignment, goal_tolerance = 0.01.

2. Plan a path between q_{init} and q_{goal} using the completed RRT algorithm. Run the algorithm ten times and report the success or failure to find a plan. Reason in a few sentences, why RRT fails to find a path to the goal in one or multiple cases.

2.3 RRT algorithm with goal bias

This is the final part of the assignment where you will modify the RRT algorithm from the previous part to include a goal_bias factor. A goal_bias factor decides how often the goal will also be considered as a random sample. You are free to use any method of your choice to incorporate the goal_bias factor as long as the choice is clearly explained (Hint: You can use the `rand` function of MATLAB for instance).

1. Set goal_bias = 1.0. This means, the probability that the random sample is the goal is 1. Plan a path to the goal using the modified RRT with the goal_bias factor. What do you observe and why?.
2. Set goal_bias = 0.05. This means, the probability that the random sample is the goal is 0.05. Plan a path to the goal using the modified RRT. What do you observe and why?
3. Visualize the solution you find using `body_animate.m` and plot the path of the end effector from the start to the goal location similar to what was done in the A* part.