The Dish Mover

Jiajun Xu
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
jxu74@illinois.edu

Vinay Vemuri
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
vvemuri2@illinois.edu

***Abstract*—This report documents our original goal, our final implementation, and other relevant details about our ECE470 final project - the Dish Mover. We discuss the successes we've had, as well as the challenges we've faced. We also discuss the next steps for our current implementation**

***Index Terms*—Robotics Simulation, Vrep, UR3, Forward Kinematics, Inverse Kinematics, Collision Detection, Motion Planning**

I. Introduction

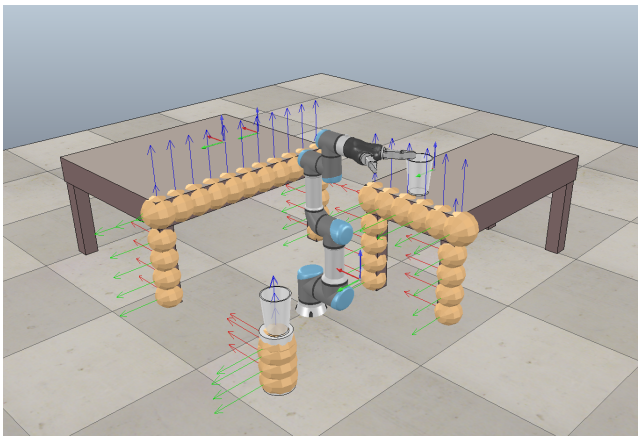

Fig. 1. Final scene of our Dish Mover demo

Our original goal of the project is to simulate a robot that can move dishes from a dishwasher to a dish rack. The goal was motivated by the pain in our experience with unloading the dishwasher. In our final simulation, we have a model of the kitchen, and a UR3 robot with a gripper attached to its end effector. We chose UR3 as our robot because we are familiar with the robot, and it has 6 degrees of freedom, which is capable to achieve any goal pose within its reach.

Our approach simplifies a kitchen setting by representing the dish rack as a table, the dish washer as a post, the dishes as 2 cups. In the final simulation, we demoed the ability for the robot to grab 2 cups at 2 different locations, and move them to 2 designated locations, all while avoiding obstacles along the way. We've implemented forward kinematics, inverse kinematics, collision avoidance, and motion planning in our own code. The code and the scene can be found in this github link[1]. We also have a video showing our final demo: demo[2]

II. Implementation Details

A. Modeling of the Environment

We cover the obstacles with a series of dummy objects, whose volumes cover the entire obstacles. Every dummy object has a position and a radius. For simplifications, as shown in figure 1, only one side of each desk is covered by the dummies. That's because we've decided that given the robot position, only those 2 sides have the possibility of resulting in collision with the robot. The dummies don't need to be drawn in the scene for our demo to be functional, since we have dedicated variables to keep track of them. They are there mainly for visualization purposes.

We also cover our robot arm with dummies extensively. Not only did we cover the joints of the robot, we also cover the links with several dummies, as well as the gripper with several smaller dummies. The dummies on the robot are different from the dummies on the obstacles, because they move with the parts of the robot they are attached to. The dummies on the robot arm were not shown in the demo or in figure 1 for the purpose of keeping the video look neat.

[1]https://github.com/jiajunxu/ECE470
[2]https://www.youtube.com/watch?v=NMZ6Y7S99F0

B. Schematic

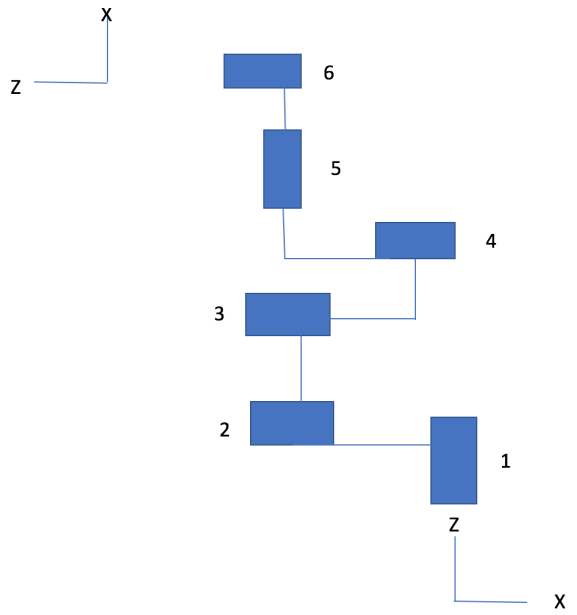

Fig. 2. schematics of the UR3 robot

C. Forward Kinematics

Forward kinematics provides the foundation for moving the joints of the robot and is crucial to understand in order to implement inverse kinematics, which is heavily used in our design. Mathematically, the aim of forward kinematics is to calculate the pose of the tool frame given a set of joint angles. As our robot has 6 joints, the size of the set would be 6. There are three terms required to calculate the pose of the tool frame. The first terms needed are the 6 thetas which should be provided for the computation. The second term needed for the computation is the twist variable, S. S is computed based on two variables, a and q. a is the matrix that corresponds to the rotational component vector in the 3 dimensional space of a joint and q corresponds to the position. The two terms are unique to every joint. Based on the type of joint, there are two ways to compute the twist. Since all the joints in our robot are revolute joints, the matrix S is computed in the following way:

$$S = \begin{bmatrix} a \\ -1 * skew(a) * p \end{bmatrix}$$

The function $skew(a)$ takes an input of a 3 x 1 matrix and converts it into a 3 x 3 skew-symmetric matrix. The size of S that has been calculated is 6 x 1. S is found for every joint of the robot. The next step is for each respective S to be converted into a 4 x 4 skew-symmetric matrix. This result is then multiplied with the respective θ, and then is taken into the exponential. This is done for every joint given the respective a, p, and θ. Finally all the 6 exponentials are multiplied with M, the initial pose of the robot. Given below are example values that are used to calculate the pose of the tool frame:

$$a1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} p1 = \begin{bmatrix} 0 \\ 0 \\ 0.1045 \end{bmatrix} a2 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} p2 = \begin{bmatrix} -0.1115 \\ 0 \\ 0.1089 \end{bmatrix}$$

$$a3 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} p3 = \begin{bmatrix} -0.1115 \\ 0 \\ 0.3525 \end{bmatrix} a4 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} p4 = \begin{bmatrix} -0.1115 \\ 0 \\ 0.5658 \end{bmatrix}$$

$$a5 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} p5 = \begin{bmatrix} -0.1122 \\ 0 \\ 0.65 \end{bmatrix} a6 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} p6 = \begin{bmatrix} 0.1115 \\ 0 \\ 0.6511 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & -1 & -1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.10 & -0.35 & -0.56 & 0.11 & -0.65 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M = \begin{bmatrix} 0 & 0 & -1 & -0.1940 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0.6511 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combining all the above steps together would produce the below equation:

$$T(\theta) = e^{[S_1]\theta_1} \times e^{[S_2]\theta_2} \times e^{[S_3]\theta_3} \times e^{[S_4]\theta_4} \times e^{[S_5]\theta_5} \times e^{[S_6]\theta_6} \times M \quad (1)$$

For simplicity, the position of the robot is placed at the Origin of the world frame. The figure 2 below shows the schematics of the UR3 robot.

D. Inverse Kinematics

The role of Inverse Kinematics is to output the robot joints given the destination for the robot's end effect-or. We choose an iterative method to calculate the inverse kinematics for our robot. In this method, the first step is to find the current pose of the robot similar to the previous section above with an arbitrary set of theta values. The next step is finding V, where V contains the value of screw axis provided by computing from a computed twist. This is found by the below equation where T_2^0 is the goal pose and T_1^0 is the current pose.

The next step is finding the Jacobian, J. The approach uses the adjoint of the product of the exponentials. This product of exponentials is calculated in the same manner as mentioned in the Forward Kinematics Section. The term Si is equivalent to the twist of the respective joint.The Jacobian is found based on the below equation.

Next, $\dot{\theta}$ is calculated using the formula below, where the value of mu is arbitrary, preferably 0.01.

Finally, theta, the solution we are trying to find, is updated by adding it with the $\dot{\theta}$ we have just calculated. This is all wrapped in an infinite while loop and is broken when the norm of the spatial twist is above the threshold epsilon.

The algorithm below contains all the equations mentioned in the paragraphs above in order to find the set of theta values with respect to the steps mentioned above.

Data: Goal pose for the end effector, T_2^0
Data: robot's initial pose, M
Data: Robot's screws for all the joints, S
Initialize θ randomly
v = 10000000
while v ¿ *0.1* **do**
  T_1^0 =Forward Kinematics(S, M, θ)
  $[V] = log(T_2^0(T_1^0)^{-1})$
  $J = spacejacobianof\theta$
  $\dot{\theta} = (J^T J + \mu I)^{-1} J^T V$
  $\theta = \theta + \dot{\theta}$
end
return θ

Algorithm 1: Algorithm for motion planning

There are tool poses for which no solution exists. For example, when the goal pose are outside the bounding box of the robot, theres no way for the end effector to reach outside the bounding box. Our algorithm finds the goal configuration iteratively. It terminates the algorithm once it iterates over 1000 times, which determines that a solution could not be found.

There are tool poses for which more than one solution exists. Since our algorithm is initializes θ randomly, different initialization of θ will sometimes converge to different values. Our algorithm will simply return the first solution it finds.

It is possible to have only solutions that are singular configurations. To prevent the singular matrix we add the Identity matrix.

Inverse Kinematics is a primary step in this robot's approach to generate 6 joint values, given a final pose. This final pose is usually the position of the dish or the position where the dish must be placed. However, not all joint values are legal as the would cause collisions with other objects in the environment or the robot would collide with itself. The sections below aim to resolve this issue.

E. Collision Detection

Collision detection is done by modeling the robot and the obstacles in the environments as 2 sets of spheres with certain radius.

In our implementation, the first set of spheres have the positions stored in the array, p_robot. The radius of each sphere is stored in the array, r_robot. Each element of the r_robot corresponds to each element in p_robot. Combined together, p_robot and r_robot describe a set of spheres that covers the robot entirely but not much space other than the robot. The values in the p_robot change as the robot moves.

The second set of spheres are described by p_obstacles and r_obstacles. They cover the obstacles in the environment that can possibly collide with the spheres describing the robot.

To actually perform the collision detection, we check if every single one of the robot sphere is in collision with any of the spheres in the environment other than itself. We decide 2 spheres are in collision if the sum of the radius of the 2 spheres are greater than the distance between their centers. It can be described in algorithm form show in Algorithm 2

if $r_1 + r_2 > p_1 - p_2$ **then**
  Collision detected
else
  Not in collision
end

Algorithm 2: algorithm for checking if 2 spheres are in collision

To check if the robot as a whole is in collision, we loop through every sphere in the robot spheres, and check to see if any of the sphere is in collision with any of the obstacle spheres in the environment. That is not enough, since in some configurations, the robot will be in self collision. So on top of checking if the robot sphere is in collision with the environment, we also need to check if any of these spheres is in collision with any other robot spheres to avoid self collision

F. Motion Planning

The motion planning is done using the inverse kinematics and the collision detection mechanism we've implemented.

We are using an array, p_robot, that stores different positions on the robot arm, and another array, r_robot, to store the radius of those positions. For every set of given joint variables, we calculate where those joints will be. Using the positions and their corresponding radius, we can check if any of the positions is in collision with any of the obstacles in the environment.

The main algorithm for motion planning is described in algorithm 3.

There aren't configurations that would result in a false positive, because our algorithm only detects collisions when the distances between centers of the robots and obstacles are shorter than the sum of the corresponding radius. However, there are configurations that would result in a false negative, because in order to model the obstacles in the world, we cover obstacles with dummy objects that have a center and a radius. Because of this approach, its almost impossible to cover the entire world with dummies. The parts of the world that are not covered by the dummies will result in a false negative for the robot.

Even though our algorithm can return functional paths in most cases, there are choices of start and goal between which there are no collision-free paths. Our planner finds the path by iteratively looking for possible configuration that connects both the starting and the goal tree. When the planner has iterated over more than a certain number of times, it will decide that its not probable that a solution can be found, so it will terminate the search.

In general, planning a collision-free path usually takes about 10 seconds. For the same start and goal configuration, the time it takes the planner to find a path varies, but usually within a few seconds of each other. For different start and

Data: Starting configuration of the robot
Data: Goal configuration of the robot
Initialize the start tree that roots in the starting configuration;
Initialize the goal tree that roots in the goal configuration;
while *path not found* **do**
  Sample a set of joint values, that is collision free;
  if *theres a collision free path between and the closest configuration in the starting tree* **then**
    add θ to the starting tree, with its parent being the closest node
  else
  end
  if *theres a collision free path between and the closest configuration in the goal tree* **then**
    add θ to the goal tree, with its parent being the closest node
  else
  end
  if *both of the above if statements are true* **then**
    output the path by tracing the parents the θ in both trees
  else
    continue
  end
end

Algorithm 3: Algorithm for motion planning

goal configurations, depending on the specific configurations, it can differ from as short as a couple of seconds to as long as more than 10 seconds.

G. Demo Specifics

1) Challenge 1: Originally, the cups were not considered obstacles. The reason being that in order for the gripper to grip the cup, the cup cannot be considered in collision with the gripper. However, in our trials, we found that the gripper, or other parts of the robot, can easily collide with the cups and knock them off their positions.

So we've decided to model the cups as obstacles, too, except that when a cup is about to be gripped, we turn off collision detection so the robot can grip the cup without constantly detect collision.

2) Challenge 2: Another challenge is to be able to move the gripper to a good position to successfully grip the cups. In our experiments, we've found that if we move the gripper to the position where it should close the grip directly, it will often knock off the cup. So we've decided to make the gripper move to a position close to the ending position, and have it move in a straight line, in segments, to the ending pose.

To have the gripper move in a straight line, we have the robot move to several intermediate configurations, then to the goal configuration. The intermediate configurations are calculated by

$$\theta = \theta_{start} + i\frac{(\theta_{goal} - \theta_{start})}{k} \tag{2}$$

where θ is a particular intermediate configuration, and k is the number of intermediate steps we are taking, and i is the number of step the robot is currently in.

3) Challenge 3: Once the cup is in the gripper, it comes our next challenge – making the cup stay in the gripper. Initially, without doing any extra work, we observed that the cup can occasionally stay in the gripper when the robot moves. However, the cup will get thrown away unintentionally majority of the times we run the simulation. Our first hypothesis was that the gripper was not gripping the robot with enough force, but we couldn't figure out how to change the force the gripper is gripping with. Our second hypothesis was that the robot was moving too fast, which effectively swings the cup out of the gripper. So what we did to test that hypothesis was "slowing down" the movement of the robot by moving the robot from the initial configuration to the destination configuration in segments (much like what we did to make the robot grip the cup). The result of that is we successfully made the cup stay in the gripper for almost all cases, with the drawback that the robot moves less smoothly. We decided to go with this solution because functionality is more important.

III. Results

A.

By the time of the demo, we had successfully created a reliable system that can pick up the cup from predefined locations, and move them to the predefined destination locations, without any part of the robot colliding with obstacles along the way. Our system works most of the time.

In terms of quantifying our result, we've achieved our original goals of being able to grab a dish, and moving it to a destination, and avoiding obstacles along the way. Even though our functionality is still primitive, we have demonstrated the capability of writing our solution from scratch and having a functional prototype.

IV. Future Work

A. Computer vision

Currently, the location of the cups and the location of the obstacles are acquired by calling the VREP API. That works well in a simulation setting. Because in the simulation setting, we are aware of all the objects, obstacles, and goals. However, in real world, the environment is usually not static, which means we can't hard code all the obstacles and goals. So if we want our model to be any useful at all in the real-world setting, we need to be able to detect the locations of the cups and the obstacles.

One of the methods we'd like to try is to use computer vision techniques to figure out the environment. We'd like to approach this as a SLAM (Simultaneous Localization and Mapping) problem, where we'd need to use different sensors to detect and keep track of the environment. That way, we'd be able to deploy the system in the real world. As a way to simplify the problem, we can employ fiducial markers to be on targets, and have a few camera systems set up in the environment to figure out the pose of the fiducial markers.

B. More versatile

Right now, our robot is only capable of grabbing the cup. Since our original goal is to be able to move all the dishes from the dish washer to the cabinet, grabbing only cups is obviously not enough. We'd like to extend our robot to be able to grab other dishes like plates, bowls, as well as utensils such as forks, knifes, and spoons.

C. More robust

In order for the cup to stay in the gripper when the robot is moving, we slow down the robot movement by commanding it to move in segments. However, if we want it to be useful in real world, we need to make it move as fast as a human would. So that means we need to figure out how to apply more force to the gripper so the cup or other dishes will be held firmly in the gripper.

ACKNOWLEDGMENT

We would like to thank Professor Timothy Bretl and the ECE470 course staff for providing us with resources and teaching us the methods and procedures needed for the project.

REFERENCES

[1] K. Lynch and F.C. Park, "Modern Robotics: Mechanics, Planning, and Control" Cambridge University Press, 2017