UR3 Robotic Simulation in V-REP

Robert Maksimowicz
ECE Department
University of Illinois
at Urbana-Champaign
rmaksi2@illinois.edu

Pratheek Muskula
ECE Department
University of Illinois
at Urbana-Champaign
muskula2@illinois.edu

Siyun He
Physics Department
University of Illinois
at Urbana-Champaign
siyunhe2@illinois.edu

***Abstract*—This document discusses the practices, algorithms and methods used for the final project in Introduction to Robotics class at University of Illinois at Urbana-Champaign. It covers the forward and inverse kinematics, as well as path planning techniques, in order to achieve the final project goal described in future work section.**

I. Introduction

For this project, we used a Robot Simulator from Coppelia Robotics V-REP (Virtual Experimentation Platform Coppelia Robotics). We also decided to use a Universal Robots UR3 robot, which is lightweight and flexible table-top cobot with 360-degree rotation on all wrist joints, and infinite rotation on the end joint. The robot initial configuration is shown in Figure 1, where it is placed in a scene inside the V-REP.

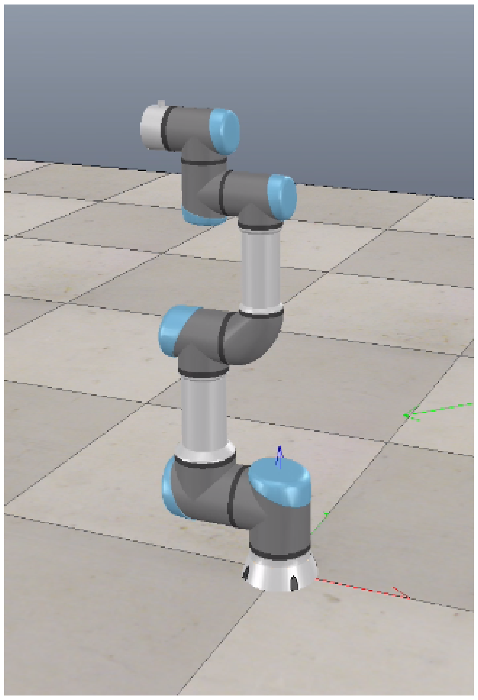
Figure 1. UR3 inside V-REP

II. Forward Kinematics

The schematics of the UR3 is shown in Figure 2. It consists of base frame, tool frame, as well as joints labeled accordingly. For all joints, axis Z+ points the same way as the link coming out from the joint.

The calculation for the end position of the robot, the tool frame T, is based on space screw axes S_n, joint variables a and q and the initial pose of the robot M. The user has to provide the set of θ values for each joint. The final tool frame is calculated by multiplying each space screw axis S_n by accompanying θ value. If $n = 6$, there will be six matrices as the end product ($e^{[S_n]\theta_n}$). Finally, the six matrices have to be multiplied by each other, followed by the base frame M, as in Equation 1.

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

Each coordinate of the twist, $e^{[S_n]\theta_n}$, is transformed to a twist, a 4x4 matrix based on Equation 2 and 3,

$$A = \begin{pmatrix} \hat{w} & v \\ 0 & 0 \end{pmatrix} \tag{2}$$

$$v \triangleq -w \times q \tag{3}$$

where w is the unit vector pointing along the axis of rotation (for a rotating joint), $\hat{w}$ is the 3×3 skew symmetric matrix derived from w, and q is any point along the axis of rotation. This is the product of exponentials formula describing the forward kinematics of an n-dof robot [1].

Below, we derived the joint variables. Matrix q_n represents the position of the joint n, with respect to the origin position that the robot was placed on (inside the V-REP scene). For the purpose of the paper, robot was placed at the world origin (coordinates $(0, 0, 0)$), as in Figure 1. We used V-REP Python API [2], functions to get the position of all six joints (q_n).

$$a_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad q_1 = \begin{pmatrix} 0 \\ 0 \\ 0.1045 \end{pmatrix} \tag{4}$$

$$a_2 = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \qquad q_2 = \begin{pmatrix} -0.116 \\ 0 \\ 0.109 \end{pmatrix} \tag{5}$$

$$a_3 = \begin{pmatrix}-1\\0\\0\end{pmatrix} \qquad q_3 = \begin{pmatrix}-0.117\\0\\0.353\end{pmatrix} \tag{6}$$

$$a_4 = \begin{pmatrix}-1\\0\\0\end{pmatrix} \qquad q_4 = \begin{pmatrix}-0.118\\0\\0.566\end{pmatrix} \tag{7}$$

$$a_5 = \begin{pmatrix}0\\0\\1\end{pmatrix} \qquad q_5 = \begin{pmatrix}-0.112\\0\\0.65\end{pmatrix} \tag{8}$$

$$a_6 = \begin{pmatrix}-1\\0\\0\end{pmatrix} \qquad q_6 = \begin{pmatrix}-0.112\\0\\0.651\end{pmatrix} \tag{9}$$

The rotational component of the screw, a derivation comes from Figure 2. They are based on the axes direction that the rotational joint moves about. The joints in Figure 2 are labeled accordingly, as well as the initial and final frame.

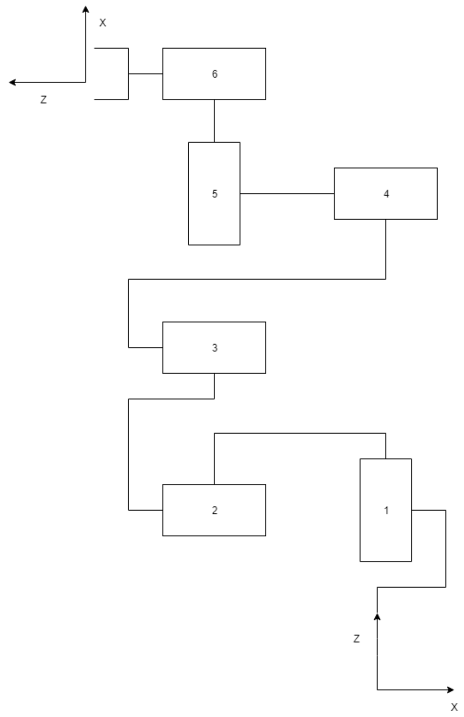

Figure 2. Schematics of UR3

The values of space screw axes are shown in Equation 10. Since all the joints are revolute joints, the screw calculation is based on the both, the rotational and positional component of the screw.

$$S = \begin{pmatrix}0 & -1 & -1 & -1 & 0 & -1\\ 0 & 0 & 0 & 0 & 0 & 0\\ 1 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & -0.109 & -0.352 & -0.566 & 0.112 & -0.651\\ 0 & 0 & 0 & 0 & 0 & 0\end{pmatrix} \tag{10}$$

The base frame M shown in Equation 11, is based on the position and orientation of the base of the robot to the world frame. We achieved these values using V-REP API functions [2]. The top-left, 3×3 of the matrix represents the orientation, and the top three values in the fourth column represent the position.

$$M = \begin{pmatrix}0 & 0 & -1 & -0.194\\ 0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0.651\\ 0 & 0 & 0 & 1\end{pmatrix} \tag{11}$$

III. Inverse Kinematics

Based on initial position, orientation, goal pose of the end effector of the robot as well as 6 screw axes, we calculate the joint variables accordingly.

We used the method of numerical inverse kinematics as an approximate solution, instead of using analytical solution for modeling such inverse kinematics problem. Because it is difficult to invert the forward kinematics equation, exclusively the Jacobian matrix. [5]

The general Jacobian inverse based algorithm is the following: Let there be n variables in the forward kinematics equation. Therefore we get the position function as

$$p_1 = p(x_0 + \delta x) \tag{12}$$

with initial position as $p_0 = p(x_0)$ and p_1 as the goal pose. The Jacobian inverse kinematics method iteratively calculate an estimate of δx which minimizes the error given by the

$$||p(x_0 + \delta x) - p_0|| \tag{13}$$

From there, we can use Taylor expansion on the position equation as

$$p(x_1) = p(x_0) + J_p(x_0)\delta x \tag{14}$$

where $J_p(x_0)$ is a $(3 \times n)$ Jacobian matrix of the position function at x_0. Using the Moore-Penrose pseudo-inverse of the Jacobian we get

$$\delta x = J_p^{\dagger}(x_0)\delta p \tag{15}$$

where

$$\delta p = p(x_0 + \delta x) - p(x_0) \tag{16}$$

[5]

Denote that when J is not invertible, then J^{-1} does not exist, however, we can still solve the equation using

$$\dot{\theta} = (J^T J + \mu I)^{-1} - J^T V \tag{17}$$

This method is equivalent of using the Moore-Penrose pseudo-inverse $J^{\dagger}$ because

$$J^{\dagger} = (J^T J)^{-1} J^T \tag{18}$$

if J is rank deficient. The $+\mu I$ term serves as an regularization term so that adding such term would prevent singular matrix. [3]

In general, there are tool configurations for which the solution exists when J is not invertible by either if it is not square matrix or it is not a singular matrix. However, **our Levenburg-Marquardt Method** as showed in line 7 in Algorithm III.1 resolves the issue as stated earlier.

There are also tool configurations for which more than one solution exists. Because the Jacobian matrix could possibly not be a unique solution to the position function. There are other possible joint variables to reach the same goal pose. Our algorithm uses iterative optimization to find one of the closest solutions of the position function.

There are tool configurations for which the only solutions are singular configurations. However, our **Levenburg-Marquardt Method** as showed in line 7 in Algorithm III.1 solves the issue by adding μI. This serves as an regularization term so that adding it would prevent singular matrix. [3]

Algorithms can be included using the commands as shown in Algorithm III.1 [4].

Algorithm III.1 Numerical Inverse Kinematics algorithm

```
1: procedure GIVEN S,M AND $T_2^0(\theta)$        ▷ The initial description of robot
2: choose $\theta$ with initial guess
3: while$||V|| < \epsilon$, do:
4: find $T_1^0(\theta)$
5: find
       $[V] = log((T_2^0(\theta) \times (T_1^0(\theta))^{-1})$
6: find
       $J(\theta)$
7: find
       $\dot{\theta} = (J^T J + \mu I)^{-1} - J^T V$
8: find $\theta = \theta + \dot{\theta} \times t$
9:     return $\theta$
```

IV. Motion Planning

The method used for motion planning for the robot can be divided into to two portions: collision detection, and path generation. For collision detection we used the colliding spheres method for detecting both self-collisions and collisions with the environment. In regards to path generation, we used a sampling-based planning method.

A. Collision Detection

The colliding spheres methods work by checking for intersections between the spheres that have been defined. It does this by taking the norm of the difference of the centers of the spheres, which is distance between the centers of the spheres, and then compares that value to the sum of the radii of those respective spheres. If the sum of the radii is greater than or equal to the distance between the centers of the spheres, then a collision is detected.

The way self collision works in this design is that spheres are placed around the joints of the UR3, and the colliding spheres method is applied between these spheres anytime the positions of the spheres have moved within the path generation algorithm. Since we decided to not place spheres at non-joint areas of the UR3, there is a chance for false negatives since every part of the robot is not encapsulated by a sphere. However, it is also important to note that the colliding spheres method is inherently susceptible to false negatives, since we are covering a robot with non-spherical components in a sphere, so the robot is unlikely to be completely inside spheres at all times.

Collisions with the environment are handled similarly to how self collision works. The difference being that the spheres designated to the environment, which are static for this project are checked against the robot's spheres using the colliding spheres method whenever self collision is also checked. The same inherently possibilities for false negatives when using the colliding spheres method also applies to checking for collision with the environment. This can clearly be seen when viewing the positions of the spheres in the concrete blocks, during the simulation.

B. Path Generation

Sampling-based planning is when we generate a path from the θ_{start} set to the θ_{goal} set. In order to use this method, some overhead needs to first be initialized. That overhead including two linked list structures for holding the points in the path. The first holds the points in the path from the start configuration and the second holds the path from the end configuration. After these structures are initialized the algorithm can start.

The way the sampling-based planning method begins is by generating a random point within the bounds of the robot range of the six joints. Then the most recently added point to the linked lists is checked for if the robot can move from that configuration to the randomly generated configuration. This is done by thinking of a line that connects these positions in six dimensional space, and sampling by a step size along that line and checking for self, and environment collisions at each sample. If a collision occurs at any point on this line, then the randomly generated point is not added to the list.

The algorithm will continue until a configuration of the robot is shared by both lists. Once a shared point is found the goal theta's list is appended to the start theta's list and put into a list denoting the final path. At this point the the path without collisions is found and the robot will move from the start configuration to all the configurations in the final path list. To make sure the algorithm doesn't continue infinitely in cases moving from the start pose to the goal pose is impossible, we have a counter and place a bound on the amount of iterations that the algorithm goes through. It usually takes the algorithm two to six seconds to find a collision free path, with the only noticeable factor that effects this being the proximity of an object in the environment to the goal pose.

One issue our current design has is false positives in regards to some of the goal poses that are given. When viewing where the goal pose is and the simulator with regards to the environment, it seems like the robot should be able to reach the position, but is unable to. A possible reason for this error is an algorithm issue, where instead of saving the points in a list, the points could be saved in a tree. This may make the path finding process run faster, since the reason a false positive is generated is due to too many iterations of the algorithm.

V. Future Work

Our goal for the end of the semester is to be able to move a set of two blocks from separate initial locations to final location, while avoiding objects in its path. In the final position, the blocks will be stacked. The blocks will be initially placed at a predetermined location, but the final location will be entered by the user. If the position given is invalid, then the program will prompt the user for a different goal position. The conditions that will make the goal position invalid is if it is impossible to reach the position without collision to the robot, or if the blocks collide with the environment. Time permitting, implementing the Tower of Hanoi, solving three blocks is something we are considering.

To complete this goal we need to find a proper end effector to use in V-REP to accomplish this task, and learn how to use it's child scripts. Possible options for end effectors that fit this task are the Baxter gripper, Baxter suctions cup, and P-grippers. In addition, collision detection for the block and checking for valid block locations would need to be implemented. In regards to collision detection we have a set of spheres for the UR3 and a set of spheres for the environment. We'll need to add a set for blocks, which will have slightly different collision detection handling, since we still need to be able to stack blocks on top of each other.

VI. Possible Improvements

In current project, we used a linked list data structure for sampling based motion. However, sampling the path with a binary tree structure would be more robust and efficient. In a linked list structure, the items are linked together through a single next pointer. However, the children nodes of a parent node in binary tree structure could be 0, 1 or 2 nodes. Therefore, while we try to check whether the θ we had sampling could be added a valid node in the tree, the search path to each item is a lot shorter than that in a linked list.

The alternative algorithm for sampling based motion planing with tree structure is as following:

a) Create a tree structure for θ_{start}, sampling from forward path and create a tree structure for θ_{goal}, sampling from backward path. The rest of the steps are described in Algorithm 2.

Algorithm VI.1 Sampling based Motion Planing algorithm

```
 1: procedure ADD $\theta_{start}$,$\theta_{goal}$ TO $T_{forward}$, $T_{backward}$
 2: Sampling $\theta$ with in free space;
 3: find the closest node $\theta_{forward}$ in $T_{forward}$ to $\theta$
 4:     if (1-s)$\theta_{forward}$ + s$\theta$ in desired free space for all$s \in [0,1]$
 5:  then
 6: add $\theta$ to $T_{forward}$ with parent node $\theta_{forward}$
 7: find the closest node $\theta_{backward}$ in $T_{backward}$ to $\theta$
 8:     if (1-s)$\theta_{backward}$ + s$\theta$ in desired free space for all$s \in [0,1]$
 9:  then
10: add $\theta$ to $T_{backward}$ with parent node $\theta_{backward}$
11:     if $\theta \in T_{forward}$ and $\theta \in T_{backward}$ then
12:     return $\theta$ = $[\theta_{forward}, \theta_{backward}]$
```

Acknowledgment

We would like to acknowledge the Stack Overflow website, as well as V-REP discussion forum, which were great sources of information to approach this project.

We would also like to acknowledge Professor Tim Bretl, since most of the implementations that we came up with were based on the lectures and homework for the course.

References

[1] K. Lynch and F.C. Park, "Modern Robotics: Mechanics, Planning, and Control" Cambridge University Press, 2017, pp. 141–142.
[2] Coppelia Robotics, V-REP Python API. [Online]. Available: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm [Accessed: April 2018]
[3] K. Lynch and F.C. Park, "Modern Robotics: Mechanics, Planning, and Control" Cambridge University Press, 2017, pp. 229
[4] K. Lynch and F.C. Park, "Modern Robotics: Mechanics, Planning, and Control" Cambridge University Press, 2017, pp. 230-234
[5] Wikipedia - Inverse Kinematics. [Online]. Available: https://en.wikipedia.org/wiki/Inverse_kinematics.htm [Accessed: April 2018]