

Pick & Place Project Review

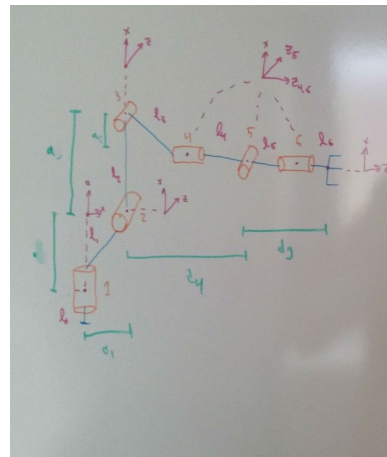
By: Raymond Andrade

Project Overview:

The goal of this project was to program the Kuka KR210 robotic arm in simulation to pick up an object and place it in a bin. This was done using Forward and Inverse Kinematics in Python which was then interpreted in ROS, Gazebo, and Rviz.

DH Parameter Table:

Denavit-Hartenburg parameters are basically tables with all the relationships between each revolute joint axes of a robot. The DH parameter table for this project described the 6 revolute joints of the robot, as well as the orientation of the gripper arm, using the variables n , α , a , d , and θ , to describe the revolute joint number, axis orientation, x-axis distance, z-axis distance, and angle respectively in terms of the revolute joint before it. The actual values of these variables can be derived by launching the project and running the `roslaunch tf tf_echo [reference frame] [target frame]` script to find the transform between the joint frames.



n	α_{n-1}	a_{n-1}	d	θ
1	0	0	0.75	
2	$-\frac{\pi}{2}$	0.35	0	$\theta_2 - \frac{\pi}{2}$
3	0	1.25	0	
4	$-\frac{\pi}{2}$	-0.054	1.5	
5	$\frac{\pi}{2}$	0	0	
6	$-\frac{\pi}{2}$	0	0	
6	0	0	0.343	0

Kinematics Approach:

The kinematics of this project was the backbone of this project as it is the basis of learning how to control serial manipulators of robots to go to a set position. This was done using transformation and rotation matrices between the revolute joints of the robot arm. Since there are 6 revolute joint on Kuka robot used in the simulation, there are 6 different angles (thetas) that

```
def transform_matrix(q, alpha, a, d):  
    T_Matrix = Matrix([ [cos(q),          -sin(q),          0,          a],  
                        [sin(q)*cos(alpha), cos(q)*cos(alpha), -sin(alpha), -sin(alpha)*d],  
                        [sin(q)*sin(alpha), cos(q)*sin(alpha), cos(alpha),  cos(alpha)*d],  
                        [0,                0,                0,                1]])  
    return T_Matrix
```

need to be found. First a DH parameter table was created with known values of the robot. Each angle was then calculated by combining the transformation matrices with substituted values from the DH parameter table, and then applying the rotation matrices, for each revolute joint to find the relative positions of the end effector and wrist center. One optimization here from what was

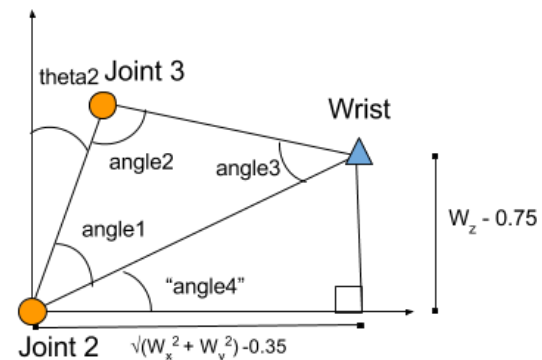
taught in class was creating a python class for the Kuka robot which already had the multiplied out matrix values for each transformation so it did not need to be calculated each time the program was run. I did this by multiplying all the transform matrices together (in order) to obtain the homogeneous transform from base to each link all the way to gripper, printing out the result of each matrix multiplied and saving it into the python class. This took my calculation time from 59 seconds to 0.8 seconds. After these transforms and rotations were completed, the angles could then be calculated using inverse kinematics.

Inverse Kinematics:

The inverse kinematics portion of this project is to determine what the angles of each joint on the robot arm should be to get it's end effector to the desired position. This is done with trigonometric identities by incorporating the position of the wrist and effector effector, as well as the physical measurements between the joints of the robot (which is given in the DH-Parameter table).

Theta 1 was calculated using the position of the of the wrist center, taking the arctan of the y and x position of the wrist gives the angle needed since it is the only revolute joint who's z-axis is lined up with the world's z-axis. Theta 2 and 3 needed a

virtual triangle to help determine their angles since they are orientated similarly and next to each other. This virtual triangle is created from the position of Joint 2, Joint 3, and Wrist. The triangle's sides will then consist of: the x distance between the wrist and joint 2 when frame is at origin (1.5), the distance between joint 2 and 3 (1.25), and the xyz distance between the wrist to joint 2 ($\sqrt{\{[\sqrt{(W_x^2 + W_y^2)} - 0.35]^2 + [W_z^2 - 0.75]^2\}}$). We can then find the angles of the triangle using the law of cosine [$c^2 = a^2 + b^2 - 2ab \cdot \cos(\theta)$] rearranged to ($\theta = \cos^{-1}([a^2 + b^2 - c^2] / [2ab])$).



To find Theta 2, subtract angle1 and "angle4" from pi/2 as shown in the figure on the right. As a note, "angle4" is not specifically defined in my code, and instead is written directly into the Theta 2 calculation using the arctan of its opposite and adjacent side. Theta 3 is calculated by subtracting angle2 (in the same figure) from pi/2 and adding the 0.036 radian offset that the robot arm has in its starting position.

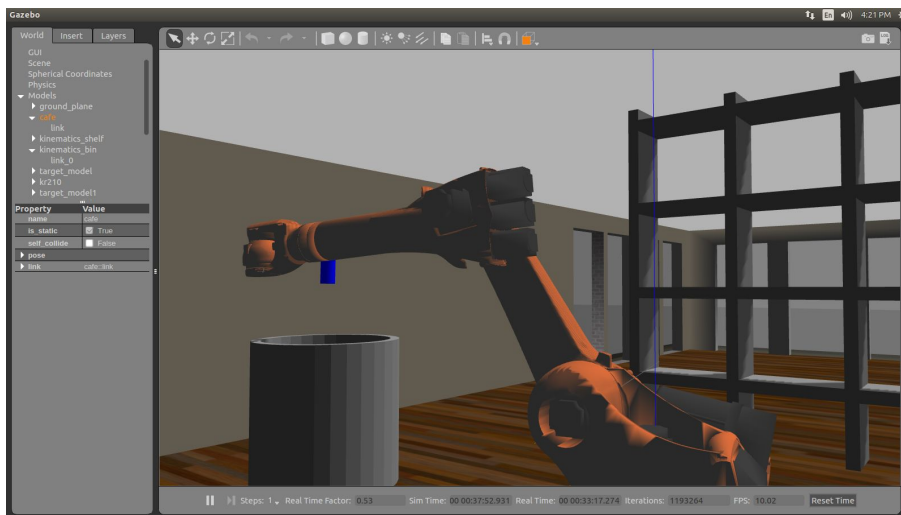
Theta 4, 5, and 6 require rotation matrices from joint 0 to joint 3 (R0_3) and joint 3 to joint6 (R3_6). R0_3 is calculated using the defined transforms between 0 and 3 evaluated with the theta 1-3 values plugged in. R3_6 is just the R0_3 rotation matrix transpose (inverse rotation) multiplied by the rotation of the gripper. The R3_6 rotation matrix can then be applied to find the Euler angles. Theta 4 then can be solved by taking $\tan^{-1}(R3_6[2,2] / -R3_6[0,2])$. Theta 5 takes into account the z height difference from the wrist to the gripper which translates to $\tan^{-1}(((R3_6[0,2])^2 + (R3_6[2,2])^2) / (R3_6[1,2]))$. Lastly Theta 6 is $\tan^{-1}(-R3_6[1,1], R3_6[1,0])$.

Since theta 4 and theta 6 share the same z-axis, it is possible for there to be multiple rotations that will achieve the same end positional result. Using theta 5 as a determinant, if the angle comes out greater than 0. If it does, then theta 4 and 6 are calculated as documented above, if not, the negative signs for the rotation matrices are flipped. This leads to a more direct indicator of rotation and leads to less needless rotations to get to the same end position.

Project Runtime:

The project runtime was definitely a challenge itself. Many times I would run the `safe_spawner.sh` script to result in errors and Rviz or Gazebo crashing. This aside, once the `IK_server.py` which had my Kinematics code was started, everything went through relatively smoothly. One thing I did notice was that the robot arm would needlessly rotate its end effector in ways it didn't need to to complete the desired motion, which would prolong the process of

putting the cylinder in the bin by several minutes. I later found out, thanks to the slack community in `#udacity_pick_place`, that a lot of were calling the `.inv('LU')` function for our matrix, which although seemed to get the end effector where it needed to go (most of the time), caused a lot of needless rotations certain revolute joints. Once I changed it to



`.transpose()` which was recommended, a big majority of these needless rotations disappeared, and as a bonus, dropped my calculation time from 0.8 to 0.4 seconds. After resetting the environment and my server, my test trials came out with a 93% (14/15) success rating of dropping the cylinder into the bin (the one failure seemed to be the gripper not having enough time to close before it left for the bin).

Conclusion:

This project definitely was a big step up from the Rover Project in terms of difficulty. Between the use of Matrices in python for calculations, learning kinematics and all it's notations, and being introduced to different software frameworks (ROS, Gazebo, and Rviz), it was a challenge trying to meet with the time constraint of the project. A big portion of this project was spent trying to visualize and correctly classify relationships between the components of the Kuka robot and then learning how to actually apply Kinematic relationships to literally get from "point

A” to “point B”. If I were to spend more time on the project, I would definitely try to further optimize the python code to decrease the project runtime by preventing scenarios of rotating certain revolute joints, like the end effector, when it is not needed.