Kinematics Project - Pick & Place

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it!

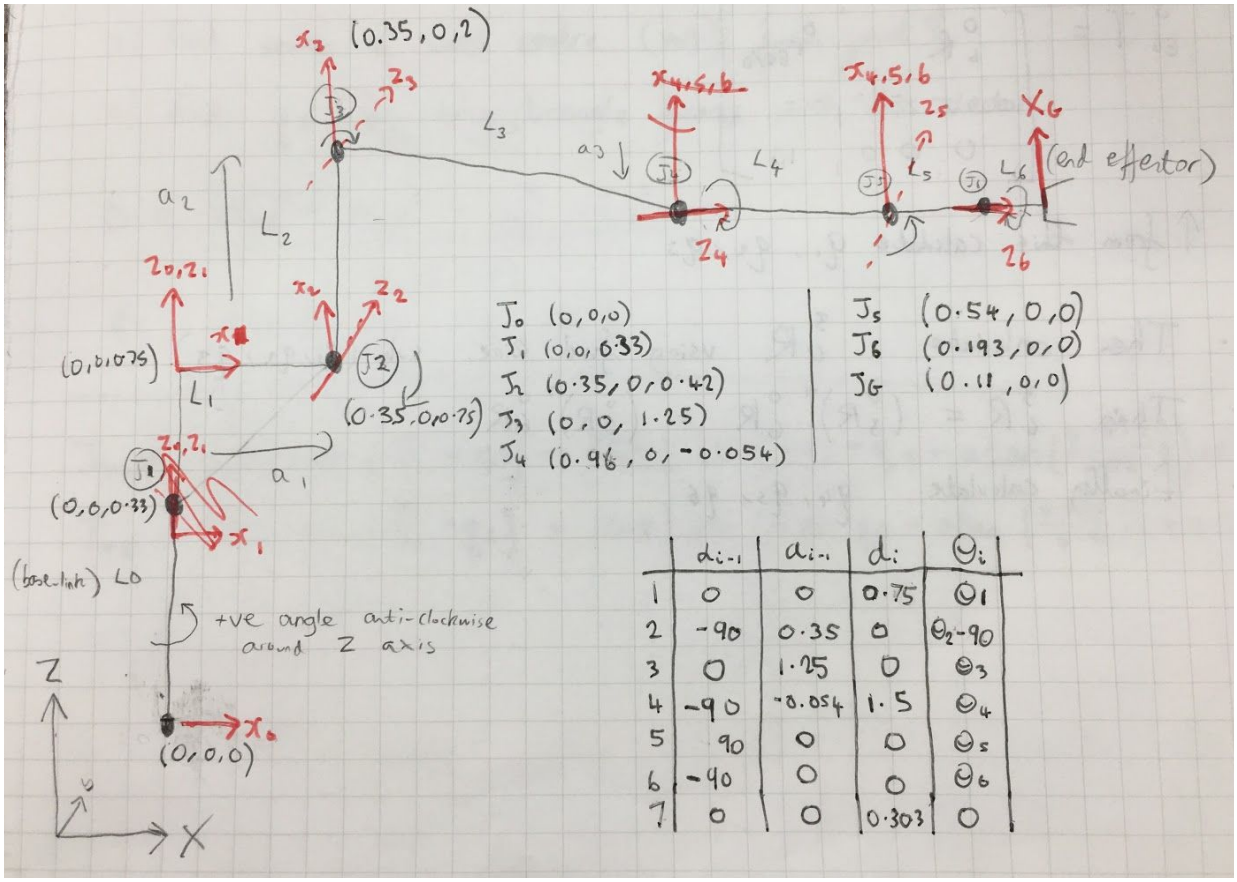
Kinematic Analysis

1. Run the forward_kinematics demo and evaluate the kr210.urdf.xacro file to perform kinematic analysis of Kuka KR210 robot and derive its DH parameters.

Based on the kr210.urdf.xacro file and the forward_kinematics demo I drew a model of the KUKA arm, as shown in Figure 1. All joints and links are clearly marked. In addition I added the axes at each joint in red. I then used this image to determine the DH parameters as shown in Table 1.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0.75	θ_1
2	$-\pi/2$	0.35	0	θ_2
3	0	1.25	0	θ_3
4	$-\pi/2$	-0.054	1.5	θ_4
5	$\pi/2$	0	0	θ_5
6	$-\pi/2$	0	0	θ_6
7 (gripper_link)	0	0	0.303	0

Table1 - Calculated DH parameters for KR210



	d_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0.75	θ_1
2	-90	0.35	0	$\theta_2 - 90$
3	0	1.25	0	θ_3
4	-90	-0.054	1.5	θ_4
5	90	0	0	θ_5
6	-90	0	0	θ_6
7	0	0	0.303	0

Figure 1 - Hand-drawn model of the KUKA arm used to determine DH parameters

2. Using the DH parameter table you derived earlier, create individual transformation matrices about each joint. In addition, also generate a generalized homogeneous transform between base_link and gripper_link using only end-effector(gripper) pose.

I made some scripts (https://github.com/adammalpass/RoboND_misc/tree/master/kinematics) with Python and Sympy to create and test the making of the transformation matrices before implementing them in the main IK_server.py code for the project. The forward_kuka.py script implements the forward kinematics based on the DH parameters from Table 1. I also used it to create individual transformation matrices about each joint, which you can see below:

T0_1
Matrix([
[cos(q1), -sin(q1), 0, 0],

T1_2
Matrix([
[sin(q2), cos(q2), 0, 0.35],

[sin(q1), cos(q1), 0, 0],
[0, 0, 1, 0.75],
[0, 0, 0, 1]])

T2_3
Matrix([
[cos(q3), -sin(q3), 0, 1.25],
[sin(q3), cos(q3), 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])

T4_5
Matrix([
[cos(q5), -sin(q5), 0, 0],
[0, 0, -1, 0],
[sin(q5), cos(q5), 0, 0],
[0, 0, 0, 1]])

T6_7
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0.303],
[0, 0, 0, 1]])

[0, 0, 1, 0],
[cos(q2), -sin(q2), 0, 0],
[0, 0, 0, 1]])

T3_4
Matrix([
[cos(q4), -sin(q4), 0, -0.054],
[0, 0, 1, 1.5],
[-sin(q4), -cos(q4), 0, 0],
[0, 0, 0, 1]])

T5_6
Matrix([
[cos(q6), -sin(q6), 0, 0],
[0, 0, 1, 0],
[-sin(q6), -cos(q6), 0, 0],
[0, 0, 0, 1]])

Another test script, wrist.py, was used as a test-bed for calculating the wrist centre (WC), which is part of the inverse kinematics step. In this script I also calculated the generalised homogeneous transform between base_link (T0) and gripper_link (T7) using only the end-effector(gripper) pose. This can be seen below, where px, py, pz is the target location of the gripper:

T0_7
Matrix([
[cos(pitch)*cos(yaw), -sin(yaw)*cos(pitch), sin(pitch), px],
[sin(pitch)*sin(roll)*cos(yaw) + sin(yaw)*cos(roll), -sin(pitch)*sin(roll)*sin(yaw) +
cos(roll)*cos(yaw), -sin(roll)*cos(pitch), py],
[-sin(pitch)*cos(roll)*cos(yaw) + sin(roll)*sin(yaw), sin(pitch)*sin(yaw)*cos(roll) +
sin(roll)*cos(yaw), cos(pitch)*cos(roll), pz],
[0, 0, 0, 1])

3. Decouple Inverse Kinematics problem into Inverse Position Kinematics and inverse Orientation Kinematics; doing so derive the equations to calculate all individual joint angles.

Please see the following answer. I describe in detail how I calculated the inverse position kinematics as part of the run-through of how my code operates.

Project Implementation

1. Fill in the `IK_server.py` file with properly commented python code for calculating Inverse Kinematics based on previously performed Kinematic Analysis. Your code must guide the robot to successfully complete 8/10 pick and place cycles. Briefly discuss the code you implemented and your results.

Here I'll talk about the code, what techniques I used, what worked and why, where the implementation might fail and how I might improve it if I were going to pursue this project further.

Code Description

The first part of the code (~lines 33-132) implements the DH table and creates the individual transform matrices around each joint, as discussed earlier in this report. Also as discussed in the lesson I applied intrinsic (body-fixed) rotations to gripper frame (7) to ensure orientations are aligned. First I rotated around z-axis by 180deg and then around y-axis by -90deg.Note that in the skeleton code provided this was done within the for loop that iterates over all requested poses, however I found that doing this caused a major slowdown in the code execution speed. Since the symbolic manipulation is the same each time I reasoned this didn't need to be inside the loop, so I moved it before and improved the performance.

Next comes the main loop where the inverse kinematics are calculated for each of the requested poses. The first step is to calculate the wrist centre (WC) location (~lines 153-189). The first sub-step is to construct R0_6 based on the target roll, pitch and yaw angles requested of the end effector. This is done by first rotating an amount *roll* around the x-axis, then an amount *pitch* around the y-axis and finally an amount *yaw* around the z-axis, before concatenating the transforms. Then it is just required to translate a distance equivalent to the length of the end effector (0.303m) along the z6 axis, as explained in the lesson notes.

The next step is to calculate theta1 (line 192). This is simple since we just have to project the Z-coordinate of the WC into the ground plane, as explained in lesson 2-19.

Next (~lines 195-216) I calculate the location of joint 2 (since it depends on theta1). I also calculate the length of various segments which will be used in later calculations (NB: All coordinates with the suffix *__0 are joint coordinates at the time when q1=q2=q3=q4=q5=q6=0 which is a useful situation to calculate joint lengths).

The next step is to calculate theta 3 (~lines 217 - 229). For me this was the most difficult part of the entire project! I found this video (https://youtu.be/llUBbpWVPQE?t=4m45s) which helped me a lot to understand the approach I should take to solve, but even then it took me a while to ensure all the offsets were correct. I used my forward_kuka.py and inverse_kuka.py scripts extensively to test a variety of angles and see if I was generating the right results. Ultimately the correct approach was to calculate a vector between joint 2 and WC and then use the cosine rule to solve for theta3, as shown in Figure 2. It should be noted that there are in fact two possible solutions for theta3 (theta3 on line 224 and theta3_2 on line 225). I picked the first solution (positive square root) since the other one (negative square root) would lead to a theta3 exceeding the specified joint limits. I confirmed via the forward kinematics that this led to the correct final position of the end effector.

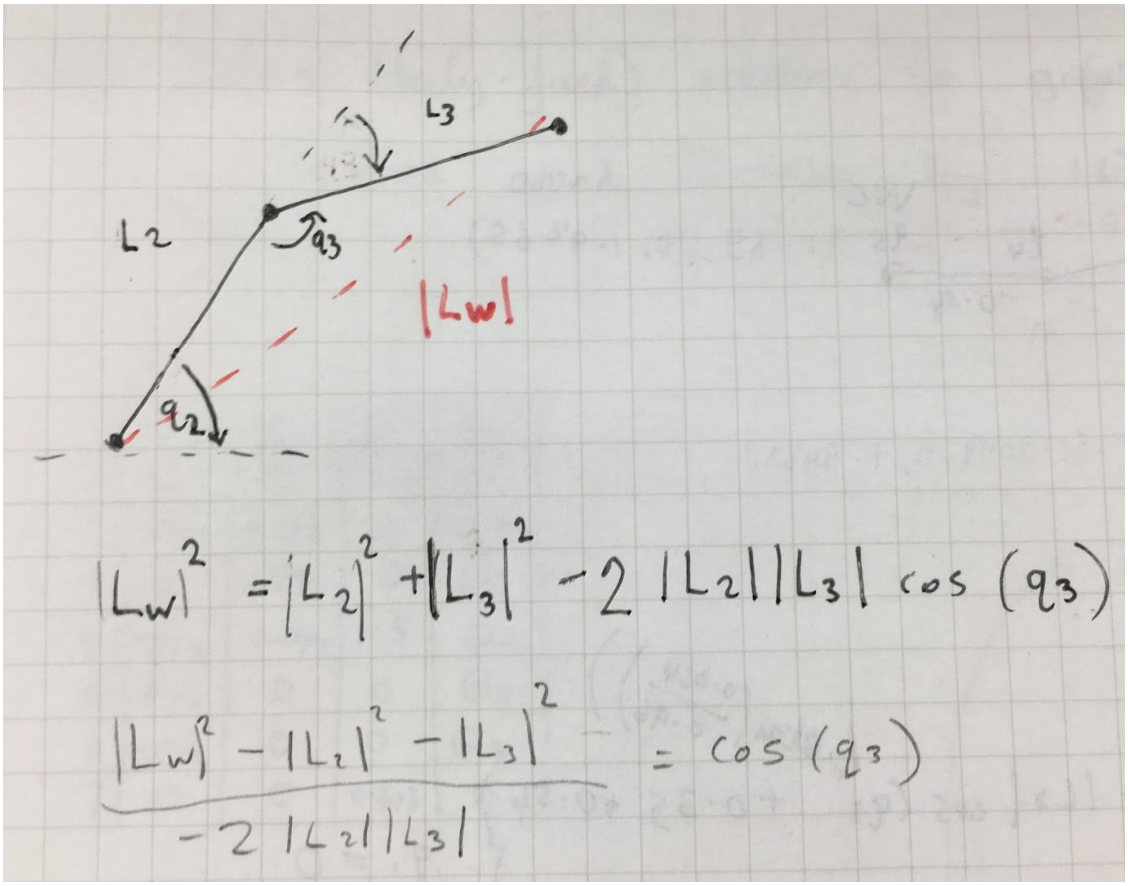


Figure 2 - Simplified model of joints 2 - 4 used to calculate theta3 via the cosine rule

The next step was to calculate theta2 (~lines 231 - 239), which was also extremely challenging. The trick here, as shown in Figure 3, was to identify two separate angles, beta1 and beta2 that were easier to calculate and then use them to help calculate theta2.

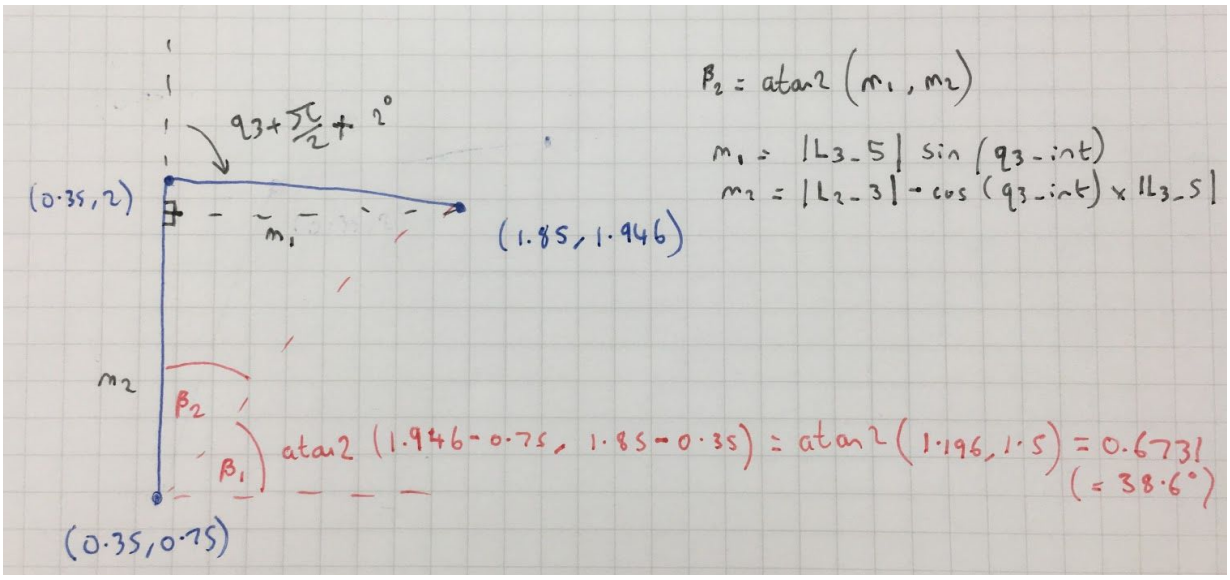


Figure 3 - Simplified model of joints 2 - 4 used to calculate theta2

Once theta 1, 3 and 2 and were correctly calculated and hence the wrist centre correctly positioned, the next main step is to calculate angles 4, 5 and 6 which set the end effector orientation (~lines 240-256). The first sub-step was to numerically evaluate the transform from joint 0 to joint 3 (T0_3) by substituting the calculated values of theta 1, 2 and 3. From this it is simple task to extract the rotational component, R0_3 to see the orientation of the WC. Next we can simply calculated the required rotation from WC to end effector, R3_6, by multiplying the inverse of R0_3 by R0_6. Once R3_6 is known we can calculate theta 6, 5 and 4 by extracting the Euler angles, as shown in Lesson 2-8. This code was taken straight from that lesson's exercise without changes.

With this all the angles are calculated and I confirmed their accuracy via my forward_kuka.py script. Now it is time to test the arm!

Results Discussion

I posted a video of the arm successfully completing the pick and place operation on Slack (https://udacity-robotics.slack.com/files/tokyo_adam/F5VLDDPL1/success_.mp4). It was an amazing feeling after lots of struggle when it correctly picked the object off the shelf and successfully dropped it into the dropbox.

It doesn't pick up successfully every single time (see 'Areas for Improvement' section below), but it does work well and I believe well enough to be able to pass the project.

Areas of Improvement

Sometimes the arm arrives at the correct point, in the correct orientation, however it comes in from the side rather than from the top, and hence knocks the object of the shelf before it has the chance to pick it up. Since it is clear the final orientation of the arm is correct, I don't believe this is an issue with my inverse kinematics calculation and code, but rather an issue with the motion planning algorithm. This assertion is stronger since I also have noticed similar behaviour in the demo mode, not just when my code is running. Furthermore the motion planning algorithm sometimes returns an error such as 'no route can be found – planning failed' which again suggests some issues with this section of the code. So if I had more time I would like to be able to dive into the motion planning code and fix this, and not just work on the inverse kinematics side.

Furthermore another issue I had was with the performance of Gazebo inside the VM. Even upping the VM resources (6 processor cores, 1024MB graphics memory, 12 GB RAM) the code performs extremely slowly. I think some of this issue is due to the heavy graphics loading, but also some of it comes from the many advanced maths and symbolic operations that have to be run more every inverse kinematics point. I moved many items outside the main loop in the code so they are only run once, rather than for every point, but it still seems not enough. If I have more time I would like to further optimise the code so it runs more smoothly. This in turn would allow me to further test the arm in more different situations, and hence further improve the accuracy of the calculated angles.