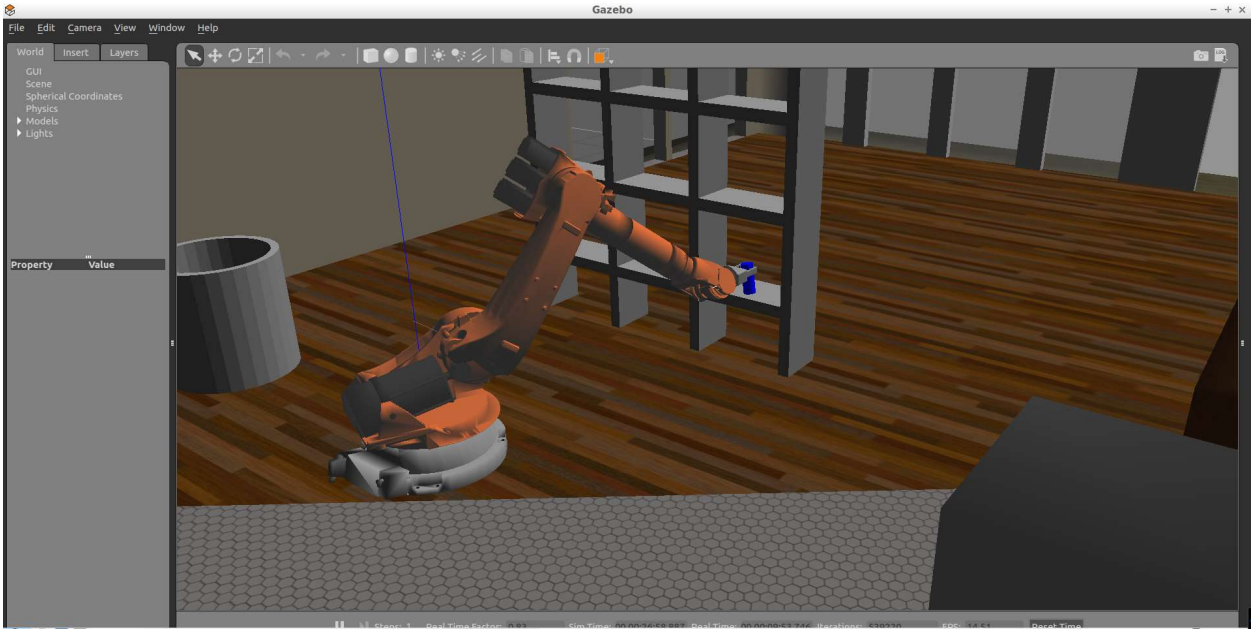
Pick and Place Project

1 Overview

We were tasked with solving the inverse kinematics problem for a simulated version of the KUKA KR210 robotic arm. This was then encapsulated in a ROS node (IK_server.py) which could be called by a planning algorithm to obtain the joint angles required for the arm to follow the provided plan.

Doing this required we first derive the Forward Kinematics for the robot (described in detail in the Forward Kinematics section of this write up). Given that, a geometric method was used to derive the inverse kinematics, with the assumption that the position and orientation of the end effector could be controlled independently. (An assumption justified by the Forward Kinematics analysis). Finally, these were combined in the IK_server.py and used in combination with provided simulation software to perform a "pick and place" operation in a simulated environment.

The KUKA in action:



2 Forward Kinematics

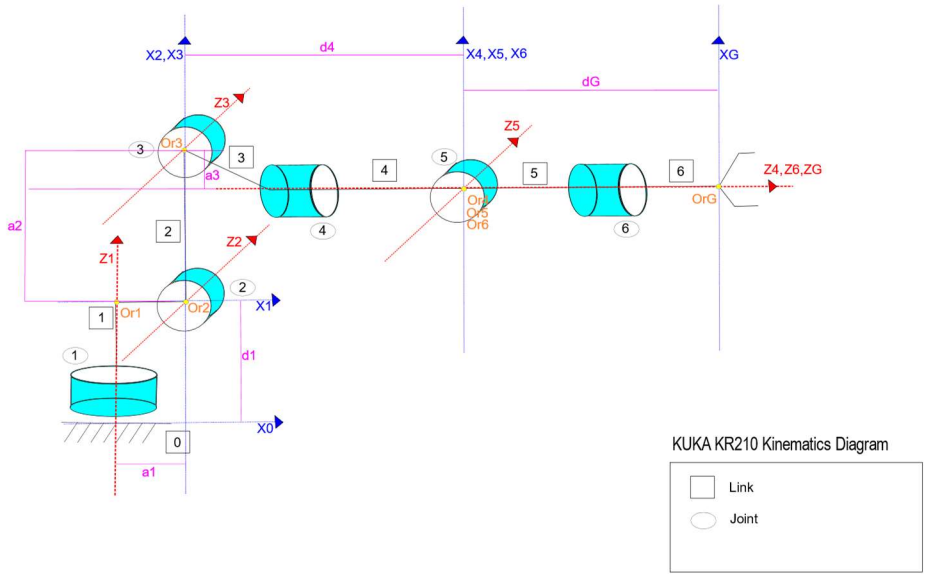
The KUKA KR210 (which shall be referred to as KUKA from here on in) is a six degree of freedom robotic arm with a two-fingered end effector attached to the end. All the joints are prismatic, although they do not all share the same axis of rotation. A schematic of the robot is shown in the diagram below,

annotated by measurements corresponding the DH parameters. The values for the measurements were taken from the kr210.urdf.xacro file in the provided code.

A key takeaway from this diagram is the intersection of the Z-Axis for joints 4, 5, and 6 (notated in red in the diagram as Z4, Z5, and Z6), this allows the position of the end effector to be determined by Joints 1,2, and 3, while the orientation can be determined by the remaining joints. This split greatly simplifies the reverse kinematics analysis and allows for an analytical solution of inverse kinematics problem.

2.1 Forward Kinematics Diagram

Appended below represents the analysis for the DH parameters of the KUKA robot. This diagram assumes that all actuators are in set in the "zero" or default positions.



2.2 Modified DH Parameter Table

i	alpha_i	a_i	d_i	theta
T_01	0	0	0.75	0
T_12	-pi / 2	0.35	0	0
T_23	0	1.25	0	0
T_34	-pi/2	-0.054	1.5	0
T_45	pi/2	0	0	0
T_56	-pi/2	0	0	0
T_6G	0	0	0.303	0

The values above correspond to the notation in the Forward Kinematics diagram provided. Most of these are simply read directly from the appropriate section of the urdf file, however there are a few special cases which I will discuss below:

- d_1: This is the difference between the base link X axis and joint 1 x-axis. As the base link is offset from the origin by 0.33 units, and the second joint is offset from joint 1 by a distance of 0.42 units, this is added together to get a d_1 value of 0.75
- d_4: Here the distance is the combination of the distance between joint 3 and joint 4 (measured along the line between the Z4 axis) and joint 4 and joint 5 (also measured along this line). This is done since we have placed X4,X5, and X6 to match with the origin at joint 5 (see diagram). Numerically, from the urdf file we have 0.96 units between joint 3 and joint 4, and 0.54 units between joint 4 and joint 5 – this gives the 1.5 value seen in the DH table.
- d_6: Finally, d_6 was obtained by similar reasoning to that of d_4, in this case the sum of the offset from joint 5 to joint 6, and from joint 6 to the end effectors origin. Reading again from the urdf file these were 0.193 and 0.11 respectively – giving the value of 0.303 shown in the table.

2.3 TRANSFORMATIONS

Using the Modified DH parameters provided above, we can derive the Forward Kinematics for the robot. Note, as mentioned in class, that given the DH parameters for a transformation between link i-1 and link i, we can represent that transform as the following matrix multiplication:

T[i-1,i] = R_x(alpha_{i-1}) * D_x(a_{i-1}) * R_z(theta_i) * D_z(d_i)

Where:

T[i-1,i] is the transformation from i-1th joint to the ith joint.

R_x is a rotation matrix about the x axis.

D_x is a displacement along the x axis

R_z is a rotation about the z axis

D_z is a displacement along the z axis

Performing this calculation and filling in the modified DH parameters, gives the following matrix (from the course notes):

cos(theta)	-sin(theta)	0	a
sin(theta) * cos(alpha)	cos(theta)*cos(alpha)	-sin(alpha)	-sin(alpha) * d
sin(theta) * sin(alpha)	cos(theta)*cos(alpha)	cos(alpha)	cos(alpha) * d
0	0	0	1

The transformation from the base to the end-effector can then be obtained by post multiplying the various transformations given by the DH parameters:

$$T[0, EE] = T[0,1] * T[1,2] * T[2,3] * T[3,4] * T[4,5] * T[5,6] * T[6, EE]$$

Substituting the parameters and multiplying the matrices out with a computer, we obtain the following transformation matrix:

3 INVERSE KINEMATICS

Now that we understand the forward kinematics of the KUKA arm, we can perform the analysis for the inverse kinematics. Whereas in the previous section we were calculating the location of the end-effector given the various joint angles, we now will calculate the joint angles given the desired location of the end effector.

As the joint axis for joint 4,5, and 6 all intersect, we can treat this robot as having a spherical wrist (as suggested by the Inverse Kinematics lesson in the course materials). This allows us to decouple the position and orientation of the end-effector into two separate problems. We'll control the position of the end-effector using joints 1, 2, and 3. The remaining joints will control the orientation of the end effector.

Before any calculations can be properly done however, we need to derive the wrist center (WC) for the desired position of the end effector (EE). This WC will be the point where the three axis' intersect in the kinematics diagram.

3.1 WRIST CENTER

We are provided with six values – three position values, and three orientation values (roll, pitch, and yaw). To obtain the WC, we start by modelling the position and orientation of the end effector, for which we will need a rotation matrix representing the desired roll, pitch, and yaw of said end effector.

$$R\_EE = Rot\_z(yaw) * Rot\_y(pitch) * Rot\_x(roll)$$

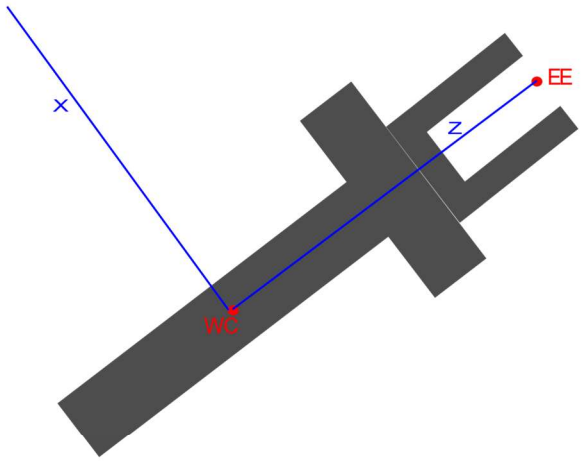
This does not consider the difference between the reference frames used for the DH parameters, and those used by the urdf file. As we will be performing all our transformations based on the DH parameters, we will need to make this correction. This correction was provided in the course materials, and is as follows:

$$R\_corr = Rot\_z(pi) * Rot\_x(-pi/2)$$

Updating our end effector orientation to be:

$$R\_EE = Rot\_z(yaw) * Rot\_y(pitch) * Rot\_x(roll) * R\_corr$$

We can then use R_EE to obtain the wrist center (WC). We do this by noting that the WC lies on the z-axis of the gripper link – as the following diagram illustrates:



The WC is then the result of translating the desired EE position along the z-axis. Since the unit vector for the z-axis can be obtained from the third column of the Rot_EE matrix, this gives the formula:

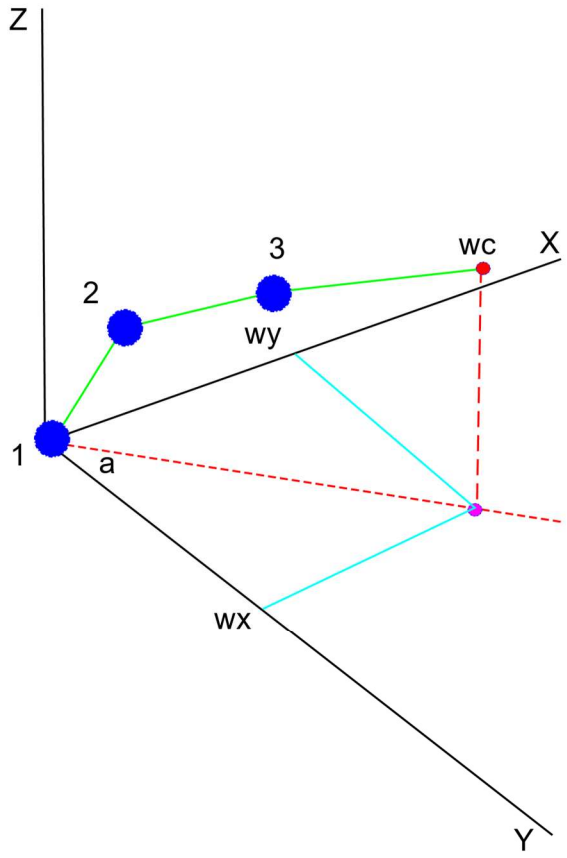
$$WC = EE - 0.303 * nz$$

Where the constant "0.303" is the distance between the WC origin and the gripper origin as seen in the forward kinematics diagram and the modified DH parameters, and nz is the third column of the Rot_EE matrix.

3.2 End Effector Position

Now that we have the WC, we can use that to deduce the joint angles that control the end effectors position. As mentioned earlier, these will be joint angles 1, 2, and 3. To avoid confusion I will refer to the joint angles as theta 1, theta 2, …, theta 6 from this point on.

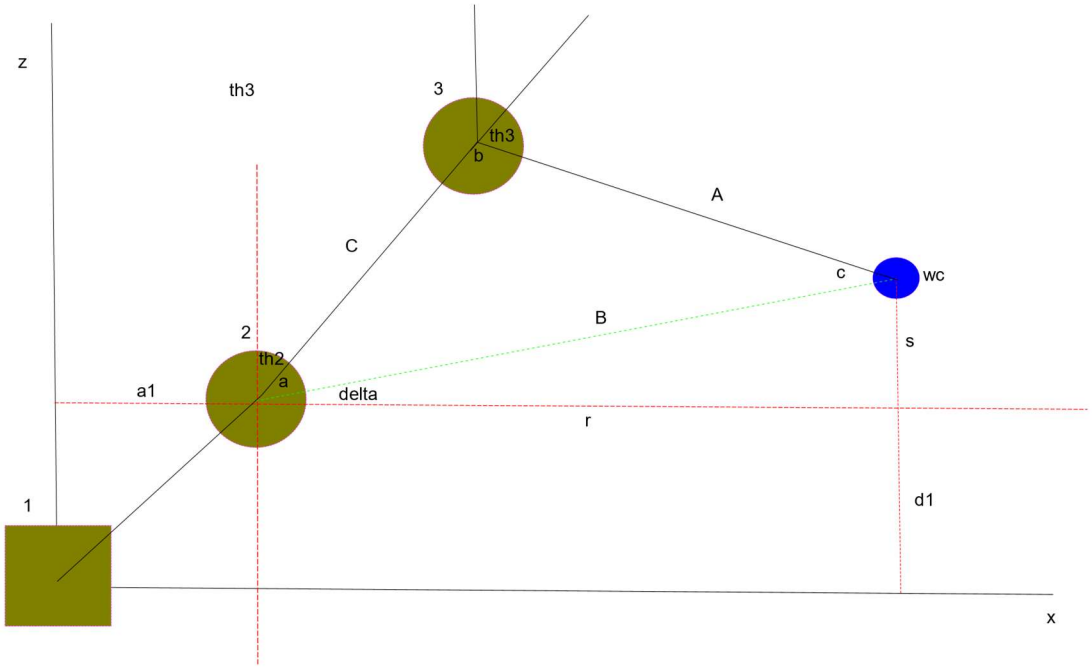
To calculate theta 1, we first project the WC onto the x,y plane, as visualized in the diagram below:



Theta 1 is denoted by a in the diagram (a limitation of my ability to use the svg software the diagram was built with). From this diagram, we see that theta 1 can be calculated as the arctan2 of wy and wx:

Theta 1 = arctan2(wy, wx)

We will use the following diagram to help derive the remaining two joint angles:



In this diagram, th2 and th3 are short for theta 2 and theta 3 – the joint angles we are attempting to derive.

We will start by deriving the length of the line segment labeled B in the diagram. Once we have this we can use the law of cosines and bit of basic geometry to calculating the other angles.

Since B is the hypotenuse of the triangle r,s,B, we must have that:

$$B^2 = s^2 + r^2$$

From the diagram we can see that r is just the length of the line from (0,0,z) to (wx, wy, z) minus the distance between the first two joint links (a1). (wx and wy are the x and y coordinates of the wrist center respectively). Thus we have:

$$r^2 = (\sqrt{wx^2 + wy^2} - a1)^2$$

By a similar line of reasoning, we can get s:

$$s^2 = (wz - d1)$$

Which allows us to get B:

$$B = \text{sqrt}((\ \text{sqrt}(wx^2 + wy^2\ ) - a1\ )^2 + (wz - d1))$$

Now we can calculate angle a using the law of cosines:

$$a = \arccos(\ (B^2 + C^2 - A^2) / 2*B*C)$$

We will also need the angle delta, which is obtained by calculating the arctan of s and r (as previously defined earlier):

$$\text{delta} = \text{arctan2}(s,r)$$

We can see from the diagram that we must have:

$$th2 + a + \text{delta} = pi$$

Therefore:

$$\mathbf{th2 = pi - a - delta}$$

Now it only remains to calculate th3. To do this we first note that b + th3 = pi (since the angles are formed by the bisection of the line alongside C of triangle ABC). As b can be obtained from the law of cosines, this means we can find th3:

$$\mathbf{th3 = pi - b}$$

Which gives us our final joint angle to finding the position of the end-effector.

3.3 End Effector Orientation

Now that the position of the end effector has been determined, we can use that to determine the orientation. To do this, we start with the observation that:

$$R\_EE = R\_01 * R\_12 * R\_23 * R\_34 * R\_45 * R\_56 * R\_6G$$

Where R_n-1n is the rotation submatrix of the homogeneous transform T[n-1,n] described in the forward kinematics section of this paper. The above result must hold true, since the left side the final orientation of the end effector, and the right side represents the portion of the transformation from the base link to the gripper link responsible for the end effectors orientation.

In the previous section we derived the joint angles for the R_01, R_12, and R_23 transforms, we can now sub these joint angles into their respective matrices to get R_03:

$$R\_03 = R\_01 * R\_12 * R\_23$$

We can now use the information we know from R_EE and R_03 to derive a matrix equation for R_34 * R_45 * R_56 * R_6G, which will be denoted R_0G for brevity:

$$R\_0G = inv(R\_03) * R\_EE$$

Here we are using inv(M) to mean the inverse of the matrix M. Once R_0G is obtained, we can extract the euler angles to get the remaining joint angles.

To this we need the general form of the R_0G matrix. We make use of the fact that that R_0G is:

$$R\_0G = R\_34 * R\_45 * R\_56 * R\_6G$$

(which was mentioned earlier), and let the computer derive the matrix from the DH parameters. When we do this, we get:

-sin(q4)*sin(q6) + cos(q4)*cos(q5)*cos(q6)	-sin(q4)*cos(q6) - sin(q6)*cos(q4)*cos(q5)	-sin(q5)*cos(q4)
sin(q5)*cos(q6)	-sin(q5)*sin(q6)	cos(q5)
-sin(q4)*cos(q5)*cos(q6) - sin(q6)*cos(q4)	sin(q4)*sin(q6)*cos(q5) - cos(q4)*cos(q6)	sin(q4)*sin(q5)

(Note: q4 is theta 4, q5 is theta 5, and q6 is theta 6 – I've left the equations in the form they are to save space as the matrix already looks a bit messy in this format).

I will derive theta 4 and theta 6 first, as they are the simplest. I will use the following notation to refer to the matrix cells: r_ij will designate the cell at the ith row in the jth column. For theta 4, consider cells r_33 and r_13. These are:

$$\sin(q4) * \sin(q5)\ (r\_33)$$

$$-\sin(q5) * \cos(q4)\ (r\_13)$$

If we divide these out we get sin(q4) / (-cos(q4)), from which we can infer that:

theta 4 = atan2(r_33, -r_13)

A similar line of reasoning allows us to work out theta 6, only here we use r_22 and r_21 respectively to get:

theta 6 = atan2(r_22, r_21)

Finally, it remains to derive theta 5. First note that:

$$r\_21^2 + r\_22^2 = \sin^2(q5)*(\cos^2(q4) + \sin^2(q4))$$

Use Pythagorean trig identity to get:

$$r\_21^2 + r\_22^2 = \sin^2(q5)$$

Now if we combine this observation with r_23 we see that tanq5 = sqrt(r_21^2 + r_22^2) / r_23, from which we can infer that:

theta 5 = atan2(sqrt(r_21^2 + r_22^2), r_23)

3.4 JOINT EQUATIONS

The joint equations are summarized in the following table:

Theta 1	**arctan2(wy, wx)**
Theta 2	**pi – a – delta**
Theta 3	**pi – b**
Theta 4	**atan2(r_33, -r_13)**
Theta 5	**atan2(sqrt(r_21^2 + r_22^2), r_23)**
Theta 6	**atan2(r_22, r_21)**

Please see the relevant sections of this paper for full explanations of these equations.

4 IMPLEMENTATION

The primary code for this project was place in the IK_server.py script file. This is the ROS node that is responsible for providing the joint angles to the simulation based on the results of the planning algorithm provided. It takes in a set of poses (which are given as cartesian coordinates for the position and quaternion for the orientation) and outputs a set of joint angles for each pose.

First a generic function for creating transformations based on the modified DH parameters is declared. This is done to simplify the transformation code:

```
def create_transform(alpha, a, d, q):
    TF = 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 TF
```

This is followed by the handle_calculate_IK function, which is the callback function called by the service whenever a request is received by this node. For brevity, the code that was provided by the course materials will be omitted and we will concentrate on the code that was written for this specific problem.

DH parameter table and supporting symbols are declared:

```
q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8')
a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7')
d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8')
alpha0, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6 = symbols('alpha0:7')

# Create Modified DH parameters
dh_params = {
    alpha0: 0, a0: 0, d1: 0.75, q1: q1,
    alpha1: -pi / 2., a1: 0.35, d2: 0, q2: -pi / 2. + q2,
    alpha2: 0, a2: 1.25, d3: 0, q3: q3,
    alpha3: -pi / 2., a3: -0.054, d4: 1.50, q4: q4,
    alpha4: pi / 2., a4: 0, d5: 0, q5: q5,
    alpha5: -pi / 2., a5: 0, d6: 0, q6: q6,
    alpha6: 0, a6: 0, d7: 0.303, q7: 0
}
```

With those in place, the forward kinematics transformation is calculated. Although the full transformation is not strictly needed to solve the problem, it helps in printing out the error values later.

```
T_01 = create_transform(alpha0, a0, d1, q1).subs(dh_params)
T_12 = create_transform(alpha1, a1, d2, q2).subs(dh_params)
T_23 = create_transform(alpha2, a2, d3, q3).subs(dh_params)
T_34 = create_transform(alpha3, a3, d4, q4).subs(dh_params)
T_45 = create_transform(alpha4, a4, d5, q5).subs(dh_params)
T_56 = create_transform(alpha5, a5, d6, q6).subs(dh_params)
T_6G = create_transform(alpha6, a6, d7, q7).subs(dh_params)

# transform from base to end-effector
T_0G = T_01 * T_12 * T_23 * T_34 * T_45 * T_56 * T_6G
```

With that done, we move on to working out the orientation matrix of the end effector. A few helper matrices are declared for this, along with hard coded matrices for the end effector correction described earlier in this paper.

```
r, p, y = symbols('r p y')

Rot_X = Matrix([
    [1, 0, 0],
    [0, cos(r), -sin(r)],
    [0, sin(r), cos(r)]
])

Rot_Y = Matrix([
    [cos(p), 0, sin(p)],
    [0, 1, 0],
    [-sin(p), 0, cos(p)]
])

Rot_Z = Matrix([
    [cos(y), -sin(y), 0],
    [sin(y), cos(y), 0],
    [0, 0, 1]
])
```

```
Rot_EE = Rot_Z * Rot_Y * Rot_X
# 180 degree rotation around z
M1 = Matrix([
    [-1, 0, 0],
    [0, -1, 0],
    [0, 0, 1]
])

# -90 degree rotation around y
M2 = Matrix([
    [0, 0, -1],
    [0, 1, 0],
    [1, 0, 0]
])

ROT_Correction = M1 * M2
```

Everything thus far is done only once per call. After these initial calculations have been made, the system enters the main loop where it works out the joint angles for each of the individual poses that have been passed to the service.

First, we get the wrist center:

```
#calculate wrist center
Rot_EE = Rot_EE.subs({'r': roll, 'p': pitch, 'y': yaw})

EE = Matrix([
    [px],
    [py],
    [pz]
])

WC = EE - 0.303 * Rot_EE[:, 2]

wx = WC[0]
wy = WC[1]
wz = WC[2]
```

With that in hand, we obtain the joint angles required to determine the position of the wrist center:

```
#calculate joint angles to determine end effecot position
theta1 = atan2(WC[1], WC[0])

# law of cosines from diagram in lesson
side_a = 1.5
side_b = sqrt(pow(sqrt(wx* wx + wy * wy) - 0.35, 2) + pow(wz - 0.75, 2))
side_c = 1.25

angle_a = acos((side_b * side_b + side_c * side_c - side_a * side_a) / (2 * side_b *
side_c))
angle_b = acos((side_a * side_a + side_c * side_c - side_b * side_b) / (2 * side_a *
side_c))

#not sure why, but occasionally getting imaginary values for angle_a, and angle_b -
taking the real parts
angle_a = re(angle_a)
angle_b = re(angle_b)
```

```
theta2 = pi / 2 - angle_a - atan2(wz - 0.75, sqrt(wx * wx + wy * wy) - 0.35)
theta3 = pi / 2 - (angle_b)
```

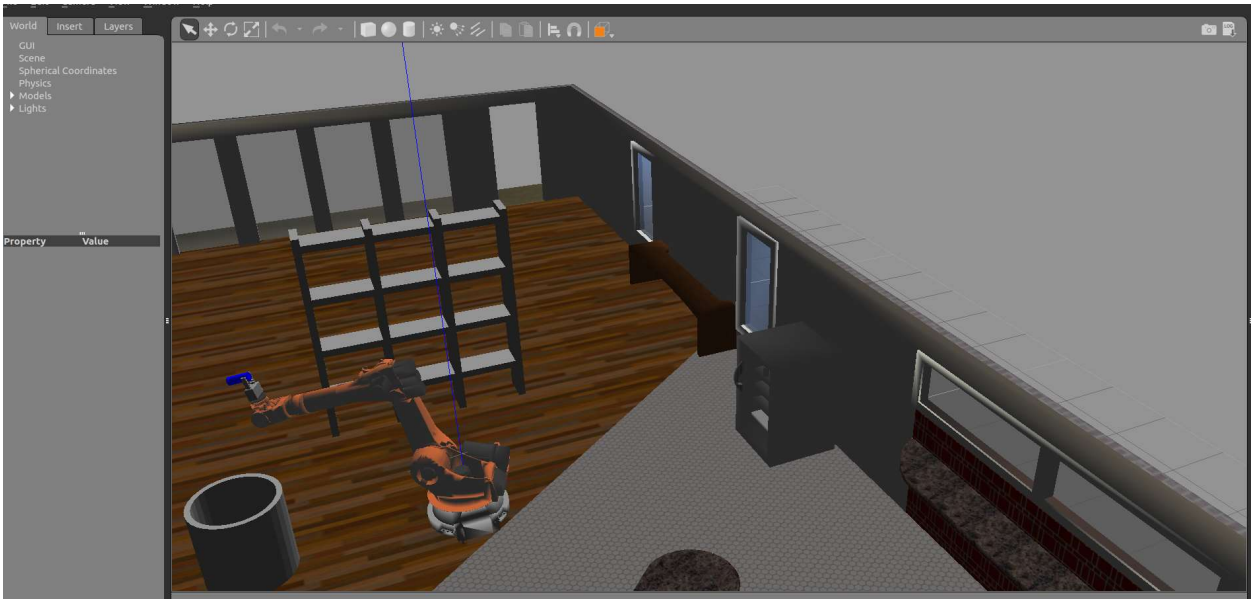
Note the handling of angle_a and angle_b. Occasionally, as a result of what seems to be a floating point error of some kind (based on https://stackoverflow.com/questions/36093673/why-do-i-get-a-complex-number-using-acos) these calculations are resulting in complex numbers. A solution that seemed to work 100% of the time was not found, instead these cases are handled safely by considering only the real part of the result. This may be contributing to some of the occasional errors seen later on.

Finally, we calculate the orientation of the wrist center, however here I have deviated slightly from the calculations provided, the reasons for which will be explained in the discussion section.

```
if x == len(req.poses) - 1:
    R_03 = T_01[0:3, 0:3] * T_12[0:3, 0:3] * T_23[0:3, 0:3]
    R_03 = R_03.evalf(subs={q1: theta1, q2: theta2, q3: theta3})
    R_36 = R_03.inv("LU") * Rot_EE

    print ("cos beta = ", R_36[1, 2])

    # extract euler angles
    theta4 = atan2(R_36[2, 2], -R_36[0, 2])

    theta5 = pi / 2

    if np.abs(R_36[1, 2]) > 0.005:
        theta5 = atan2(sqrt(R_36[0, 2] * R_36[0, 2] + R_36[2, 2] * R_36[2, 2]),
R_36[1, 2])

    theta6 = atan2(-R_36[1, 1], R_36[1, 0])
else:
    theta4 = 0
    theta5 = -pi / 2  # keep the hand down so it doesn't collide with anything
    theta6 = 0
```

The last few lines code simply populate the joint angles for this pose, and print out the error by comparing the expected end effector position to the one calculated by the forward kinematics.

5 DISCUSSION



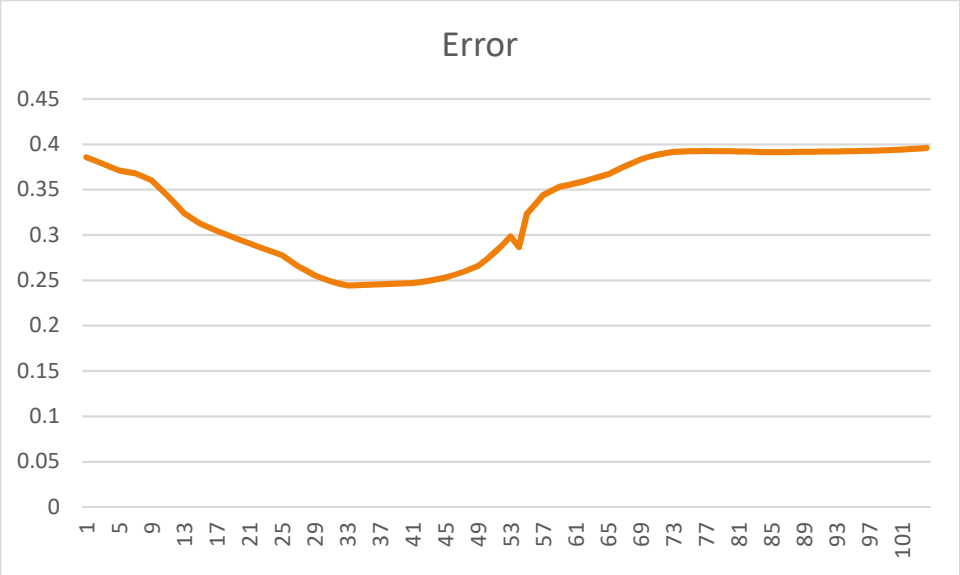
"KUKA in action"

Using the code provided (based on the calculations in the previous) section, the robot does a reasonable job of completing its task, however it seems a bit unsatisfying. It seems to twist and turn a lot more than should be required to hit each of the orientation / end effector positions specified by the planning algorithm.

To get around this annoyance, a slight change was introduced into the system, rather than trying to perfectly match the path at every step, the end effector is moved into a safe position until the final pose. This allows the robot to move much more smoothly though the simulation space and reduces the amount of wait time between attempts. Of course, the tradeoff here is that the absolute error between the end effector and desired path is now always off by a constant.

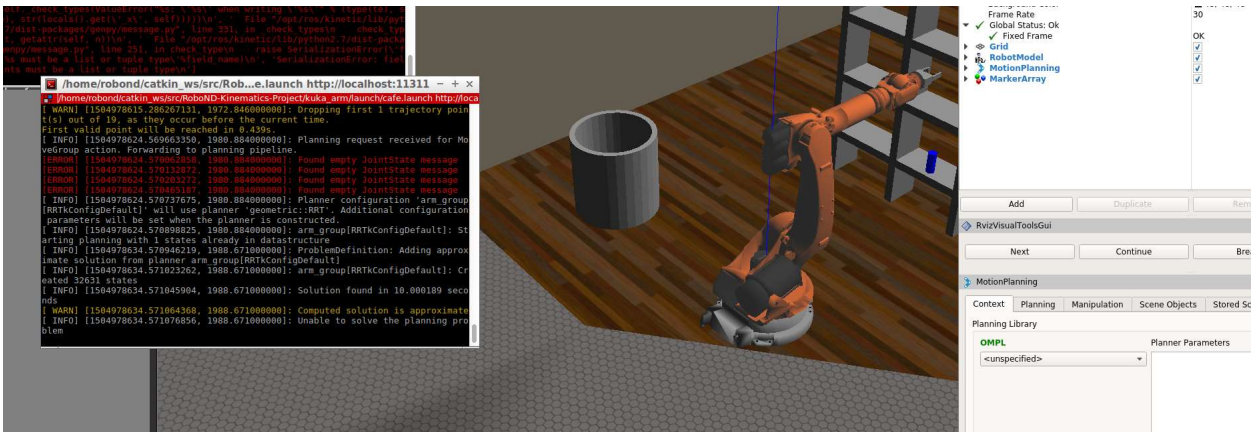
As the error between the end effector and the path is purposefully inflated by a constant during these motions to keep the system simple, the errors recorded are split into two parts – path tracking and final pose.

The graph below charts the difference between the end effector location and the desired end effector location during the path tracking phase of the robot's behavior, these tends to however around 0.4, except in cases where the safe position the arm has been placed in happens to coincide with the requested pose during movement.



Error in the final pose is generally constants – around 0.054. However, there are a few edge cases that have not yet been solved where the error will increase, the maximum observed of which is 0.34.

In most cases the ARM can complete the task that it has been given. Generally, if it fails it will do so during the pickup phase of the motion, where it occasionally knocks the target off the shelf before it can close its end effector. There is also a curious error which appears to be outside the control of the Kinematics Code where this will occur:



"Unable to solve the planning problem"

When this occurs, the only solution that was seen to work was to shut down the simulation and start it again.