Udacity: Robotic Arm Pick & Place Report

Shane Reynolds

August 12, 2018

Contents

1 Introduction & Background **2**

2 Methods & Implementation **3**
- 2.1 Determining the Denavit-Hartenburg Parameters . . . 3
- 2.2 Forward Kinematics . . . 6
  - 2.2.1 Derivation of the transformation matrices . . . 6
  - 2.2.2 Correction of Transformation Matrices . . . 6
  - 2.2.3 Verification of Transformation Matrices . . . 7
- 2.3 Inverse Kinematics . . . 9
  - 2.3.1 First Three Joint Angles (θ_1, θ_2, and θ_3) . . . 9
  - 2.3.2 Final Three Joint Angles (θ_4, θ_5, and θ_6) . . . 11
  - 2.3.3 Implementation of IK Server . . . 13

3 Results & Conclusion **15**

4 Further Enhancements **16**

5 References **16**

6 Appendix A **17**

7 Appendix B **18**

8 Appendix C **20**

1 Introduction & Background

The Amazon Pick and Place Robotics Challenge is a competition designed to help increase collaboration between the industrial and robotics research communities. Amazon has successfully implemented a number of robotic systems which largely eliminate the need for activities like searching and walking in their fulfilment centres, however, one of the main challenges that Amazon is yet to solve is picking and stowing objects reliably in an unstructured environment. To successfully achieve this objective, there are a number of tasks that need to be successfully completed. These include:

1. Identification of the target object in the unstructured environment;
2. Manipulator path planning to the object;
3. Successful execution of a reach and grasp manoeuvre; and
4. Physical relocation of the grasped object to the desired location.

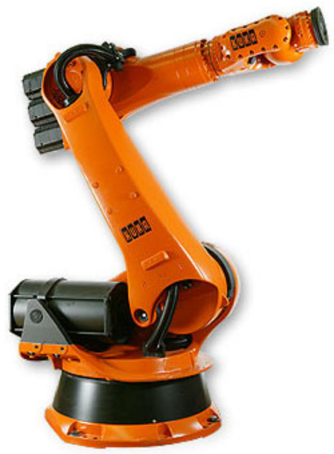


Figure 1: Kuka KR210 anthropomorphic industrial robot with 6 degrees of freedom

Path planning and execution of the desired manoeuvre is largely a solved problem. The move execution is dependent on inverse kinematics. The inverse kinematics (IK) of a robot is the mathematical conversion of position in Cartesian space to the joint angles which allows the robot end effector to reach the desired position. The end effector position in space can be thought of in 2 separate domains: Cartesian world coordinates, or Joint Angle space. Typically, analytical work is done in Cartesian space - three dimensional space is the native environment that humans live in and is easier to conceptualise. Robots, however, position themselves by making adjustments to electrical or hydraulic actuators - these actuators receive instructions based on Joint Angle. This project explores the IK derivation, and implementation, for the Kuka KR210. The KR210 is a 6 degree of freedom (dof) anthropomorphic robotic arm shown in Figure 1. The project culminates with the implementation of an IK server, which is a ROS service receiving a series of points in Cartesian space (world coordinate frame), and returning a vector of Joint Angles after applying the IK transform. The implementation will be undertaken in ROS, which utilises simulation engines Rviz and Gazebo. Figures 2 and 3 show the Kuka KR210 in simulation.

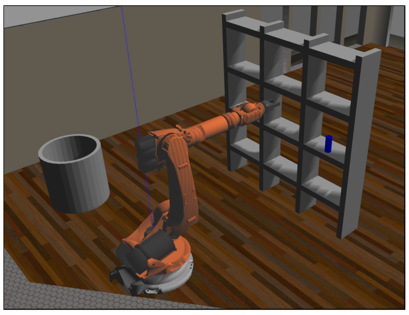


Figure 2: A picture of the KR210 in Gazebo

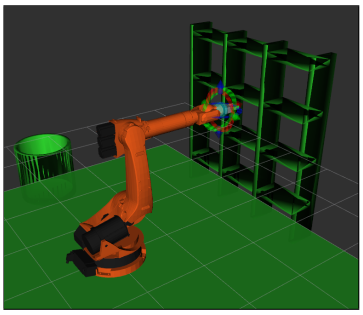


Figure 3: A picture of the KR210 in Rviz

2 Methods & Implementation

2.1 Determining the Denavit-Hartenburg Parameters

Forward and Inverse kinematic analysis relies heavily on successful specification of transformation matrices between the physical elements of the robot, which we call links. In order to determine these transformation matrices, we need to assign coordinate frames to the robot links. Doing this in an arbitrary fashion will often result in the determination of 6 parameters for each transformation matrix, which makes this process undesirably complex. Denavit and Hartenburg (1955) determined an algorithmic approach to the assignment of coordinate frames to the robot's links which reduces the number of parameters needed to describe each transformation matrix to 4. Assuming that $\hat{x}_i$, $\hat{y}_i$, and $\hat{z}_i$ are the x, y, and z axes respectively for coordinate frame i, then these parameters are defined in Table 1.

Table 1: Description of the Denavit-Hartenburg parameters

Parameter	**Description**
α_{i-1}	Twist angle, and is determined by the angle between the $\hat{z}_{i-1}$ and $\hat{z}_i$, measured about the $\hat{x}_{i-1}$ axis
a_{i-1}	Distance from $\hat{z}_{i-1}$ to $\hat{z}_i$ measured along $\hat{x}_{i-1}$, where $\hat{x}_{i-1}$ is orthogonal to $\hat{z}_{i-1}$, and $\hat{x}_{i-1}$ is orthogonal to $\hat{z}_i$
d_i	Signed distance between $\hat{x}_i$ and $\hat{x}_{i-1}$, measured along $\hat{z}_i$
θ_i	Angle between $x_{\hat{i-1}}$ and $\hat{x}_i$, measured about $\hat{z}_i$

The KR210 has a base link, 6 degrees of freedom, and an end effector. Each of the links require a coordinate frame assignment, making a total requirement of 8 coordinate frames. Each of the joints were systematically labelled from 1 to 6, starting with the joint closest to the `base_link`. Following this, each of the links were assigned a number from 0 to 7. It must be noted that link 0 is actually the `base_link`, and link 7 is the `end_effector`. For the sake of simplicity, the `base_link` and `end_effector` will retain their names throughout this report. Coordinate frames were assigned to the links according to the DH procedure. Each link can be thought of as being associated with a joint. The `base_link` is associated with the fixed ground, link 1 is associated with joint 1, and so on. To assign the coordinate frame to a link, DH requires the $\hat{z}_i$ coordinate axis for link i to pass through the joint i axis of rotation. To start the DH convention of coordinate frame assignment, the `base` frame is assigned arbitrarily. Each $\hat{x}_i$ axis, for coordinate frame i, is determined using $\hat{z}_i$, and $\hat{z}_{1+1}$. The $\hat{x}_i$ axes are assigned dependent on whether the $\hat{z}_i$, and $\hat{z}_{i+1}$ axes are:

1. **Skewed:** if the $\hat{z}_i$ and $\hat{z}_{i+1}$ axes are skewed, then the $\hat{x}_i$ axis is assigned along the normal from $\hat{z}_i$ to $\hat{z}_{i+1}$.
2. **Intersecting:** if the $\hat{z}_i$ and $\hat{z}_{i+1}$ axes intersect, then the $\hat{x}_i$ axis is assigned in an arbitrary position such that it is normal to the plane formed by $\hat{z}_i$ and $\hat{z}_{i+1}$
3. **Coincident:** if the $\hat{z}_i$ and $\hat{z}_{i+1}$ axes are parallel or coincident, then the $\hat{x}_i$ axis assignment is arbitrary along the $\hat{z}_i axis$

To reiterate, the $\hat{x}_i$ axis is assigned dependent on the geometric orientation of the $\hat{z}_i$, and $\hat{z}_{i+1}$ axes. The $\hat{y}_i$ axis is assigned to complete the right handed coordinate frame assignment. The full DH coordinate frame assignment for the KR210 can be seen in Figure 4.

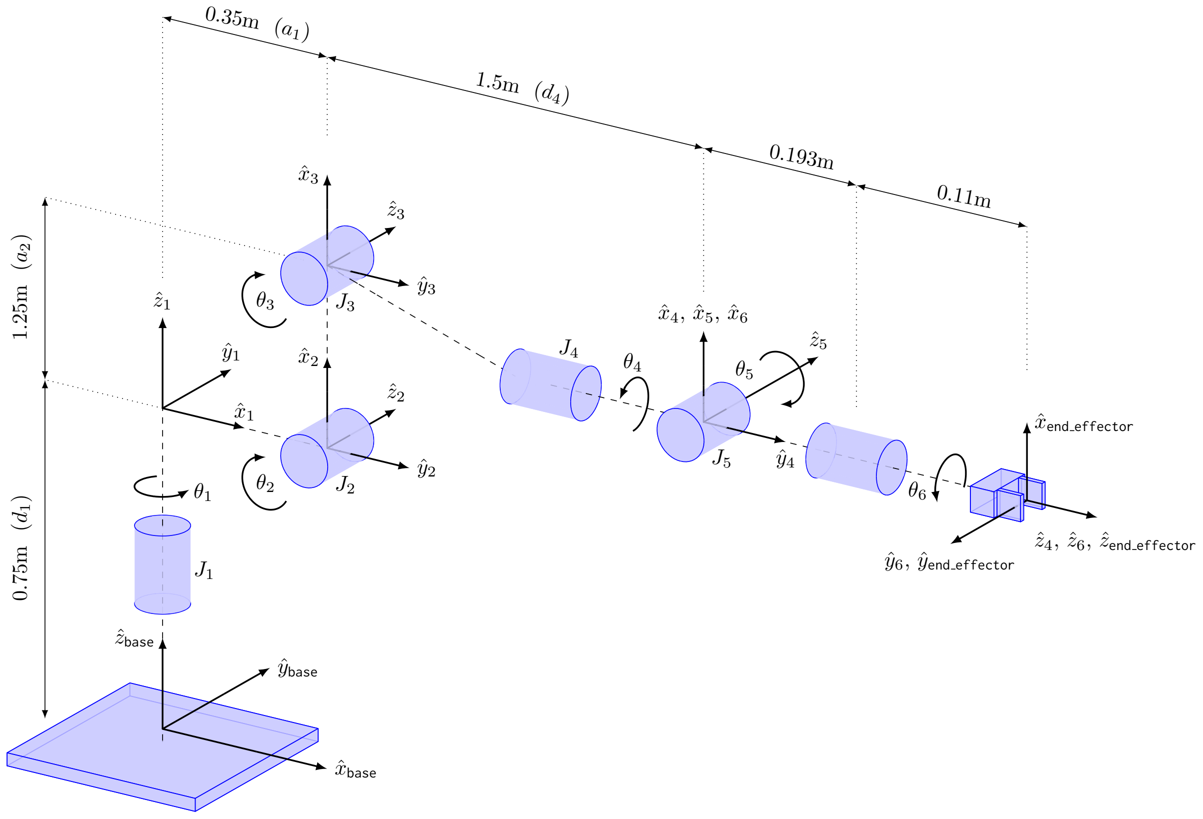


Figure 4: Sexy robot drawing

The DH parameters, which are used to specify the transformations from one coordinate frame to another, are determined once the coordinate frames have been assigned. The full DH parameter specification can be seen in Table 2. It must be noted that the values for d_i and a_{i-1} were found using the unified robot description format (urdf) file, which contains the model specifications for Gazebo. The partial urdf file can be seen in Appendix A, which shows the joint angle information. The dimensions taken from the urdf file are marked up on Figure 4, with one notable exception: a_3. The DH parameter for a_3 is the orthogonal distance between $\hat{y}_3$ and $\hat{y}_4$, and is -0.054m. This was omitted from the diagram to avoid cluttering the picture. The DH parameter specifications are used in conjunction with equation (1) to specify the transformation matrices from coordinate frame $i-1$ to coordinate frame i. This is further explored in Section 2.2.

$$
{}^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cdot \cos\alpha_{i-1} & \cos\theta_i \cdot \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i \cdot \sin\alpha_{i-1} \\ \sin\theta_i \cdot \sin\alpha_{i-1} & \cos\theta_i \cdot \sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i \cdot \cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}
$$

One final note of importance is the calculation of the angle between the axis $\hat{y}_3$ and the line formed by J_3 and J_5. Whilst it is not immediately clear why we need this value, it will be important for Section 2.3.1 when analysing the robot geometries. We note that the horizontal distance along $\hat{y}_3$, between J_3 and J_5, is 1.5m.

Table 2: DH parameter table

${}^{i-1}T_i$	d_i	θ_i	α_{i-1}	a_{i-1}
${}^{\text{base}}T_1$	0.750	θ_1	0	0.000
${}^1T_2$	0.000	$\theta_2 - \frac{\pi}{2}$	$-\pi/2$	0.350
${}^2T_3$	0.000	θ_3	0	1.250
${}^3T_4$	1.500	θ_4	$-\pi/2$	-0.054
${}^4T_5$	0.000	θ_5	$\pi/2$	0.000
${}^5T_6$	0.000	θ_6	$-\pi/2$	0.000
${}^6T_{\text{end\_eff}}$	0.303	0	0	0.000

The vertical distance, along $\hat{x}_3$, is 0.054m. Hence it follows that:

$$
\begin{aligned}
\tan\beta &= \frac{0.054}{1.5} \\
\beta &= \arctan\frac{0.054}{1.5} \\
\beta &= 2.06^o
\end{aligned}
$$

This angle of 2.06^o has been added to the 90^o formed by $\hat{x}_3$ and $\hat{y}_3$, and has been included in Figure 9.

2.2 Forward Kinematics

The forward kinematics problem is concerned with taking the agular positions of the individual joints and finding the position of the robot's end effector in three dimensional Cartesian space. This section of the report is broken down into three subsections:

1. Derivation of the transformation matrices, ${}^{i-1}T_i$
2. Derivation of transformation correction matrix
3. Verification of transformation matrices

2.2.1 Derivation of the transformation matrices

Using the DH parameters derived in Section 2.1, in addition to equation (1), we can derive a series of matrices, ${}^{i-1}T_i$, which describe the transformation of a vector in one coordinate frame i, to that of another coordinate frame $i-1$.

The individual transformation matrices are shown below:

$$
{}^{\mathsf{base}}T_1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0.75 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad
{}^{1}T_2 = \begin{bmatrix} \sin\theta_2 & \cos\theta_2 & 0 & 0.35 \\ 0 & 0 & 1 & 0 \\ \cos\theta_2 & -\sin\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{2}T_3 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 1.25 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad
{}^{3}T_4 = \begin{bmatrix} \cos\theta_4 & -\sin\theta_4 & 0 & -0.054 \\ 0 & 0 & 1 & 1.5 \\ -\sin\theta_4 & -\cos\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4}T_5 = \begin{bmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin\theta_5 & \cos\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad
{}^{5}T_6 = \begin{bmatrix} \cos\theta_6 & -\sin\theta_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin\theta_6 & -\cos\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{6}T_{\mathsf{end\_eff}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.303 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

The full transformation from the `base_link` to the `end_effector` is determined by matrix multiplication, shown in equation.

$$
{}^{\mathtt{base\_link}}T_{\mathtt{end\_effector}} = {}^{\mathtt{base\_link}}T_1 \cdot {}^1T_2 \cdot {}^2T_3 \cdot {}^3T_4 \cdot {}^4T_5 \cdot {}^5T_6 \cdot {}^6T_{\mathtt{end\_effector}} \tag{2}
$$

For the sake of brevity, the full specification for equation (2), which is ${}^{\mathtt{base\_link}}T_{\mathtt{end\_effector}}$, has not been shown - showing this transformation matrix without evaluating θ_i would take several paragraphs.

2.2.2 Correction of Transformation Matrices

The orientation of the DH base frame, shown in Figure 3, has been selected so that it aligns with the world simulation frame (defined in the urdf file). It must be noted, however, that the final DH frame for the gripper does not have the same orientation as the urdf file since the DH algorithm was employed to assign the frames. This has ramifications for any forward kinematic analysis we perform using transformation matrices defined with DH parameters. The difference in frame orientation can be better understood by comparing Figures 5 and 6.

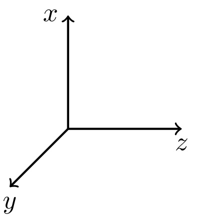


Figure 5: The orientation of the DH frame with respect to the world frame which is located at the base of the robot, shown in Figure 6

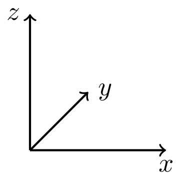


Figure 6: The orientation of thhe world coordinate frame.

To provide a correction to the transformation matrix, ${}^{\mathsf{base}}T_{\mathsf{end\_eff}}$, two steps need to be taken:

1. A rotation about the z-axis by π: $R_z(\pi)$; and
2. A rotation about the y-axis by $-\pi/2$: $R_y(-\pi/2)$

The correction matrix, $R_{correction}$, is defined mathematically as follows:

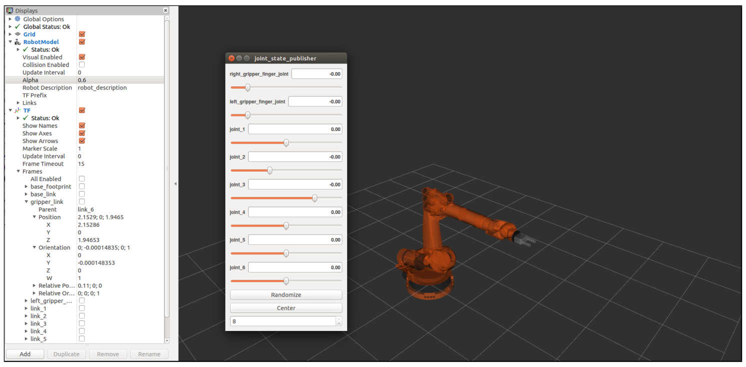
$$R_{correction} = R_z(\pi)R_y(-\pi/2) \tag{3}$$

Evaluating equation (3) allows the static implementation of this correction matrix, which saves on computational resources required to run the IK server. The implemented resultant correction can be seen in Listing 1, and the matrix is as follows:

$$R_{correction} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{4}$$

2.2.3 Verification of Transformation Matrices

To provide some assurance that the Python implementation is correct, analysis was undertaken using a script called `forwardKinematics.py`, which can be found in Appendix B. The ROS launch script, `forward_kinematics.launch`, provides a simulation of the robot in which the joint angles could be manually adjusted, and the position and orientation of the robot's frames observed. This is shown in Figure 5.



```
Global Options
Global Status: Ok
Grid                         ✓
RobotModel                   ✓
  Status: Ok
  Visual Enabled             ✓
  Collision Enabled          ☐
  Update Interval            0
  Alpha                      0.6
  Robot Description          robot_description
  TF Prefix
  Links
TF                           ✓
  Status: Ok
  Show Names                 ✓
  Show Axes                  ✓
  Show Arrows                ✓
  Marker Scale               1
  Update Interval            0
  Frame Timeout              15
  Frames
    All Enabled              ☐
    base_footprint           ☐
    base_link                ☐
    gripper_link             ☐
      Parent                 link_6
      Position               2.1529; 0; 1.9465
        X                    2.15286
        Y                    0
        Z                    1.94653
      Orientation            0; -0.00014835; 0; 1
        X                    0
        Y                    -0.000148353
        Z                    0
        W                    1
      Relative Po...         0.11; 0; 0
      Relative Or...         0; 0; 0; 1
    left_gripper_...         ☐
    link_1                   ☐
    link_2                   ☐
    link_3                   ☐
    link_4                   ☐
    link_5                   ☐
```

```
right_gripper_finger_joint   -0.00
left_gripper_finger_joint    -0.00
joint_1                       0.00
joint_2                      -0.00
joint_3                      -0.00
joint_4                       0.00
joint_5                       0.00
joint_6                       0.00
8
```

Figure 7: The Gazebo simulation of the robot using ROS, which allows the user to input joint angles, and get pose information as output. This was used to confirm that the the DH frames had been correctly assigned, and that the DH parameters were correctly determined resulting in accurate transformation matrices.

Rviz reports the postion of the gripper frame as a position vector, however, the gripper orientation is reported in Quarternions, which have to be converted to a rotation matrix in order to compare this to the calculated results from `forwardKinematics.py`. The Python script `forwardKinematics.py` contains four test cases with varying angles θ_i, such that $i \in \{1, 2, 3, 4, 5, 6\}$. These test cases can be seen in Table 3.

Table 3: Test cases for the joint angles, which were entered into the ROS similation to observe the end effector position and orientation.

Test Case	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6
Case 1	0	0	0	0	0	0
Case 2	-0.65	0.45	-0.37	0.96	0.78	0.46
Case 3	-0.79	-0.11	-2.34	1.96	1.14	-3.69
Case 4	-2.99	-0.12	0.94	4.06	1.29	-4.15

Each case was entered into the simulation, and the position vector was recorded, along with the orientation Quaternion. The recorded Qaternion was converted to a rotation matrix from which roll, pitch, and yaw can be read from the main diagonal of the matrix. The results of this process can be seen in Table 4.

Table 4: Position vector, Quaternion, and rotation matrix observered from the manual simulation of the KR210.

Case	**Position Vector**	**Quaternion**	**Rotation Matrix**
1	$\begin{bmatrix} 2.153 & 0.000 & 1.946 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
2	$\begin{bmatrix} 2.167 & -1.429 & 1.560 \end{bmatrix}$	$\begin{bmatrix} 0.698 & 0.183 & -0.153 & 0.674 \end{bmatrix}$	$\begin{bmatrix} 0.886 & 0.462 & 0.033 \\ 0.049 & -0.022 & -0.998 \\ -0.461 & 0.886 & -0.043 \end{bmatrix}$
3	$\begin{bmatrix} -0.566 & 0.940 & 2.993 \end{bmatrix}$	$\begin{bmatrix} 0.612 & 0.488 & 0.388 & 0.485 \end{bmatrix}$	$\begin{bmatrix} 0.221 & 0.221 & 0.949 \\ 0.975 & -0.051 & -0.215 \\ 0.001 & 0.973 & -0.227 \end{bmatrix}$
4	$\begin{bmatrix} -1.393 & 0.017 & 0.915 \end{bmatrix}$	$\begin{bmatrix} 0.013 & -0.229 & 0.901 & 0.368 \end{bmatrix}$	$\begin{bmatrix} -0.728 & -0.669 & -0.145 \\ 0.657 & -0.623 & -0.423 \\ 0.192 & -0.403 & 0.895 \end{bmatrix}$

Running the `forwardKinematics.py` script outputs the numerical, corrected transformation matrix, ${}^{\mathrm{i-1}}T_{\mathrm{i}}$, for each of the four test cases. The results can be seen in Table 5. Comparing the simulation results shown in Table 4 with the calcuated results from the transformation matrix, shown in Table 5, we see that position vecotrs and rotation matrices are almost identical for all four test cases. There are some small variations, however, this can be, in part, attributed to numerical truncation when performing calculations.

Table 5: Calculated position vector and orientation matrix using the matrix transfrom from the end effector to the base link.

Case	**Position Vector**	**Rotation Matrix**
1	$\begin{bmatrix} 2.153 & 0 & 1.946 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
2	$\begin{bmatrix} 2.167 & -1.428 & 1.562 \end{bmatrix}$	$\begin{bmatrix} 0.887 & 0.460 & 0.030 \\ 0.049 & -0.029 & -0.998 \\ -0.458 & 0.887 & -0.049 \end{bmatrix}$
3	$\begin{bmatrix} -0.573 & 0.941 & 2.99 \end{bmatrix}$	$\begin{bmatrix} 0.216 & 0.221 & 0.951 \\ 0.976 & -0.049 & -0.210 \\ 0.001 & 0.974 & -0.227 \end{bmatrix}$
4	$\begin{bmatrix} -1.389 & 0.022 & 0.916 \end{bmatrix}$	$\begin{bmatrix} -0.723 & -0.674 & -0.145 \\ 0.662 & -0.619 & -0.423 \\ 0.195 & 0.402 & 0.894 \end{bmatrix}$

2.3 Inverse Kinematics

Inverse kinematics (IK) is the process of determining the joint angles for each degrees of freedom in a robotic system, given the position of the robot's end effector in Cartesian space. The Kuka KR210 has 6 degrees of freedom, and hence, there are 6 joint angles which need to be determined. The anthropomorphic arm design allows us to exploit the geometry in the final 3 joints of the system. It is the intersection of the axes of rotation that is the important geometrical characteristic. This configuration of the final three joints is refered to as a spherecal wrist. This allow kinematic decoupling of the first three joints from the last three joints, providing the facility to find a closed form solution to the problem. The closed form solution is presented below, in two parts: first three joint angles, and final three joint angles

2.3.1 First Three Joint Angles (θ_1, θ_2, and θ_3)

Consider the first three joints in a pose shown in Figure 8. The point $(\omega_x, \omega_y, \omega_z)$ represents the location of the spherical wrist. Using this information, the first joint angle, θ_1, can be found using basic trigonometry:

$$\theta_1 = \text{atan2}\left(\frac{\omega_y}{\omega_x}\right) \tag{5}$$

To determine expressions for θ_2 and θ_3, we focus our attention on joints 2 and 3, as shown in Figure 9. We note that this Figure uses the existing z-axis, however, to help with the analysis a new axis is established: $(xy)'$. This is done since joints J_2 and J_3 operate in the same 2D plane, which simplifies the analysis. This allows the re-expression of the spherical wrist coordinates as (r, s), where:

$$r = \sqrt{\omega_x^2 + \omega_y^2} - 0.35 \tag{6}$$

$$s = \omega_z - 0.75 \tag{7}$$

Considering the triangle formed by the vertices at J_2, J_3, and W, we can find an experssion for η, as follows:

$$\begin{aligned}180^o &= \eta + 92.06^o + \theta_3 \\ \eta &= 87.94^o - \theta_3 \\ \eta &= 1.5348\text{rad} - \theta_3 \qquad (8)\end{aligned}$$

Employing the cosine rule on the triangle formed by vertices J_2, J_3, and W, we note that:

$$L^2 = k_1^2 + k_2^2 - 2 \cdot k_1 \cdot k_2 \cdot \cos\eta$$

Rearranging this expression we get the following expression for $\cos\eta$:

$$\cos\eta = \frac{k_1^2 + k_2^2 - L^2}{2 \cdot k_1 \cdot k_2} := D \qquad (9)$$

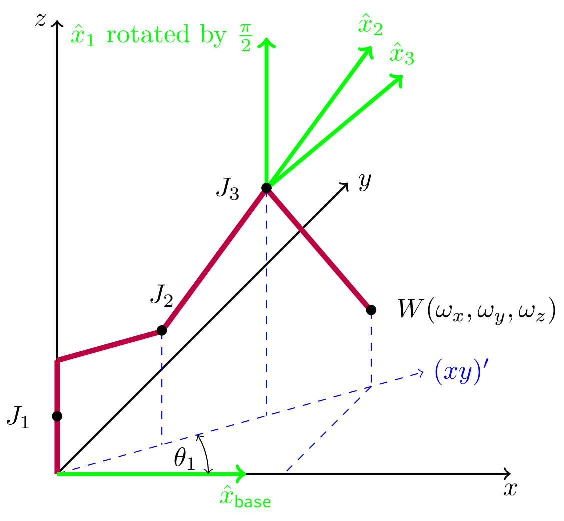


Figure 8: Robot geometry showing the first three joints, and the spherical wrist in 3D space.

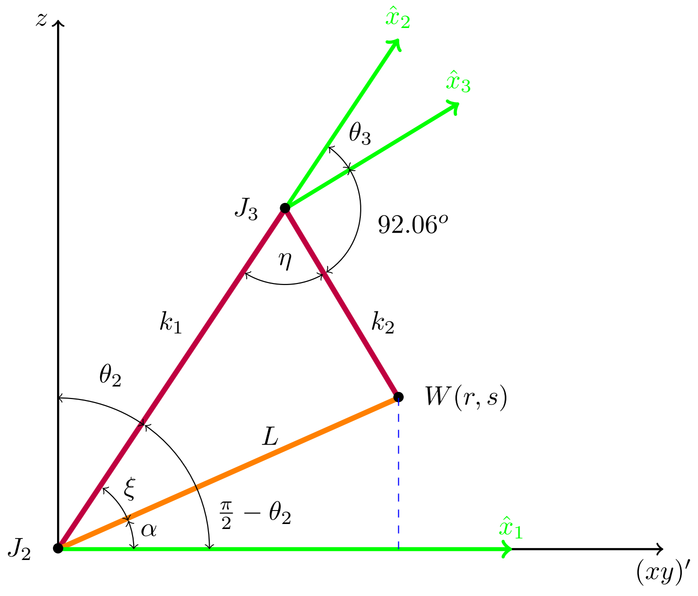


Figure 9: Roboto goemetry focusing on joints 2 and 3, which operate in the same plane allowing for analysis in 2D.

Equation (9), along with Pythagoras' theorem, allows us to determine an expression for $\sin\eta$:

$$\sin\eta = \sqrt{1 - D^2} \qquad (10)$$

Using equations (9) and (10), we get:

$$\tan\eta = \frac{\sin\eta}{\cos\eta} = \frac{\sqrt{1-D^2}}{D}$$

$$\eta = \text{atan2}\left(\frac{\sqrt{1-D^2}}{D}\right) \tag{11}$$

Substituting equation (11) into equation (8), we arrive at an expression for θ_3:

$$\theta_3 = 1.5348\text{rad} - \text{atan2}\left(\frac{\sqrt{1-D^2}}{D}\right) \tag{12}$$

Again, considering the triangle formed by the vertices J_2, J_3, and W, we can use the cosine rule to get the following expression:

$$k_2^2 = k_1^2 + L^2 - 2 \cdot k_1 \cdot L \cdot \cos\xi$$

Rearranging the above equation we find an expression for $\cos\xi$:

$$\cos\xi = \frac{k_1^2 + L^2 - k_2^2}{2 \cdot k_1 \cdot L} := K \tag{13}$$

Equation (13), along with Pythagoras' theorem, allows us to determine an expression for $\sin\xi$:

$$\sin\xi = \sqrt{1-K^2} \tag{14}$$

Using equations (13) and (14), we get:

$$\tan\xi = \frac{\sin\xi}{\cos\xi} = \frac{\sqrt{1-K^2}}{K}$$

$$\xi = \text{atan2}\left(\frac{\sqrt{1-K^2}}{K}\right) \tag{15}$$

Now, considering the right angled triangle formed between the line connecting points J_2 and W, we can find and expression for α:

$$\alpha = \text{atan2}\left(\frac{s}{r}\right) \tag{16}$$

Finally, we note that the angle between assigned DH coordinate axes $\hat{x}_1$ and $\hat{x}_2$ is $\pi/2 - \theta_2$, and hence we get the following expression:

$$\frac{\pi}{2} - \theta_2 = \alpha + \xi$$

$$\theta_2 = \frac{\pi}{2} - \alpha - \xi \tag{17}$$

Substituing equations (15) and (16) into equation (17), we arrive at an expression for θ_2:

$$\theta_2 = \frac{\pi}{2} - \text{atan2}\left(\frac{s}{r}\right) - \text{atan2}\left(\frac{\sqrt{1-K^2}}{K}\right) \tag{18}$$

2.3.2 Final Three Joint Angles (θ_4, θ_5, and θ_6)

To find the final three joint angles, θ_4, θ_5, and θ_6 we expolit the following relationship between the FK rotation matrix ${}^{\text{base}}R_6$, and the desired orientation of the end gripper, R_{rpy}, which is given in an IK problem. We note that:

$${}^{\text{base}}R_6 = R_{rpy} \tag{19}$$

It is important to note that the orientation of the robotic frame at joint 6 will be the same as the `end_effector` orientation, since the `end_effector` is simply a translation from joint 6 , with no rotation. From our FK analysis, we have the ability to break up the rotation matrix, in equation (15), as follows:

$$^{\mathsf{base}}R_6 = {}^{\mathsf{base}}R_3 \cdot {}^3R_6 = R_{rpy} \tag{20}$$

Hence, rearranging equation (16), we get:

$$^3R_6 = \left({}^{\mathsf{base}}R_3\right)^{-1} \cdot R_{rpy}$$

Since any rotation matrix is orthogonal, we can write:

$$^3R_6 = \left({}^{\mathsf{base}}R_3\right)^{T} \cdot R_{rpy} \tag{21}$$

Now, using `SymPy` it is easy to obtain an expression for $^3R_6$. For the sake of brevity, and ease of reading, $\cos\theta_i$ will be expressed as c_i, and $\sin\theta_i$ will similarly be expressed as s_i. The expression is as follows:

$$^3R_6 = \begin{bmatrix} c_4 \cdot c_5 \cdot c_6 - s_4 \cdot s_6 & -s_4 \cdot c_6 - s_6 \cdot c_4 \cdot c_5 & -s_5 \cdot c_4 \\ s_5 \cdot c_6 & -s_5 \cdot s_6 & c_5 \\ -s_4 \cdot c_5 \cdot c_6 - s_6 \cdot c_4 & s_4 \cdot s_6 \cdot c_5 - c_4 \cdot c_6 & s_4 \cdot s_5 \end{bmatrix} \tag{22}$$

The rotation matrix $\left({}^{\mathsf{base}}R_3\right)^{T}$ could be expressed in a similar way, but at this stage of solving the problem we have explicit expressions for θ_1, θ_2, and θ_3. Evaluating these expressions allows the calculation of a numerical matrix expression for $\left({}^{\mathsf{base}}R_3\right)^{T}$. Further to this, since the desired orientation of the end effector is specificed at the beginning of an IK problem, we can use the given θ_{roll}, θ_{pitch}, and θ_{yaw} to evaluate the matrix R_{rpy}. Hence, the resultant matrix from the multiplicaiton of $\left({}^{\mathsf{base}}R_3\right)^{T}$ and R_{rpy} is numerical. We express the resultant matrix elements with r_{ij}, and assume the following:

$$\left({}^{\mathsf{base}}R_3\right)^{T} \cdot R_{rpy} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{23}$$

Using matrix equality with equation (22) and (23), we note that:

$$c_5 = r_{23} \tag{24}$$

Which we can also write as:

$$s_5 = \sqrt{1 - r_{23}^2} \tag{25}$$

Hence, using equations (24) and (25) we solve for θ_5 as follows:

$$\theta_5 = \operatorname{atan2}\left(\frac{\sqrt{1 - r_{23}^2}}{r_{23}}\right) \tag{26}$$

We can use a similar approach to solve for both θ_4 and θ_6. Using matrix equality with equation (22) and (23), we get that:

$$s_5 \cdot c_6 = r_{21} \tag{27}$$
$$-s_5 \cdot s_6 = r_{22} \tag{28}$$

Hence, dividing (28) by (27), we get that:

$$\theta_6 = \operatorname{atan2}\left(-\frac{r_{22}}{r_{21}}\right) \tag{29}$$

Finally, we get that:

$$-s_5 \cdot c_4 = r_{13} \tag{30}$$

$$s_4 \cdot s_5 = r_{33} \tag{31}$$

Hence, dividing equation (31) by (30) we solve for the final joint angles:

$$\theta_4 = \operatorname{atan2}\left(-\frac{r_{33}}{r_{13}}\right) \tag{32}$$

2.3.3 Implementation of IK Server

The main objective of this paper is to implement the `IK_server.py`. This server waits to recieve a list of required poses, with each pose consisting of a Cartesian position in 3D space, along with roll, pitch and yaw orientation angles for the end effector. The full `IK_server.py` implementation can be found in Appendix C. This section of the report will explain each step of the implementation, providing code snippets to aid understanding. The main body of the `IK_server.py` is contained in the callback function `handle_calculate_IK`. Once a list of poses is received, `handle_calculate_IK` is called, and transformation matrix ${}^{\text{base}}T_3$, is evaluated and stored in the variable `T0_3`. This process includes defining symbols for the DH parameters, and assigning them values - determined in Section 2.1. Further, the rotation correction matrix, found in Section 2.2.2, is specified, and the rotation matrix ${}^{\text{base}}R_3$ is extracted from the ${}^{\text{base}}T_3$ transformation. The code snippet showing these steps can be found in Listing 1.

Listing 1: Initialsation of variables and specification of transformation matrices

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

# Create Modified DH parameters
s = {alpha0:     0,    a0:      0,    d1: 0.75,
     alpha1: -pi/2,    a1:   0.35,    d2:    0,    q2: (q2 - pi/2),
     alpha2:     0,    a2:   1.25,    d3:    0}

# Define Modified DH Transformation matrix
T0_1 = transMat(q1, alpha0, d1, a0)
T1_2 = transMat(q2, alpha1, d2, a1)
T2_3 = transMat(q3, alpha2, d3, a2)

T0_1 = T0_1.subs(s)
T1_2 = T1_2.subs(s)
T2_3 = T2_3.subs(s)

# Create individual transformation matrices
T0_2 = T0_1*T1_2
T0_3 = T0_2*T2_3

# Specify the intrinsic rotation matrix for correcting from DH to urdf
R_corr = Matrix([[0,  0, 1],
[0, -1, 0],
[1,  0, 0]])

# Extract rotation matrices from the transformation matrices
R0_3 = T0_3[0:3,0:3]
```

The transformation matrices are built using the function `transMat(q, alpha, d, a)`, which takes the DH parameters specified earlier as arguements. The `transMat` function can be seen in Listing 2.

Listing 2: The function used to build transformation matrices

```
def transMat(q, alpha, d, a):
    T = 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
```

The next part of the code extracts the position in three dimensional Cartesian space and stores the values in the variables `px`, `py`, and `pz`. Similarly, the orientation is stored in the variables `roll`, `pitch`, and `yaw`. Using these 6 variables, position of the spherical wrist is determined and stored in `wx`, `wy`, and `wz`. This is shown in Listing 3

Listing 3: Position/orientation is extracted and used to calculate spherical wrist position

```
# Extract end-effector position and orientation from request
# px,py,pz = end-effector position
# roll, pitch, yaw = end-effector orientation
px = req.poses[x].position.x
py = req.poses[x].position.y
pz = req.poses[x].position.z

(roll, pitch, yaw) = tf.transformations.euler_from_quaternion(
[req.poses[x].orientation.x, req.poses[x].orientation.y,
req.poses[x].orientation.z, req.poses[x].orientation.w])
# Define Modified DH Transformation matrix

### Your IK code here
# Compensate for rotation discrepancy between DH parameters and Gazebo
Rrpy = R_z(yaw) * R_y(pitch) * R_x(roll) * R_corr
#
# Calculate spherical wrist position
wx = px - (0.303) * Rrpy[0,2] # x-coord of wrist position
wy = py - (0.303) * Rrpy[1,2] # y-coord of wrist position
wz = pz - (0.303) * Rrpy[2,2] # z-coord of wrist position
```

Finally, the callback function calculates the joint angles. To do this, the code first establishes variables `r`, `ss`, `k1`, `k2`, `D`, and `K`, which were calculated based on the robot's geometry - outlined in Section 2.3.1. The first three joint angles θ_1, θ_2, and θ_3 are calculated using implementations of equations (5), (18), and (12) respectively. The numerical evaluation of $\left({}^{\text{base}}R_3\right)^T \cdot R_{rpy}$ is based on recently determined values of θ_1, θ_2, and θ_3, and the desired roll, pitch, and yaw orientation angles. Finally, θ_4, θ_5, and θ_6 are determined using an implementation of equations (32), (26), and (29) respectively. Listing 4 shows the implementation of this code. The call back function calculates and returns the required joint angles for each of the desired poses in the input list.

Listing 4: Code snippet showing the evaluation of the joint angles for a given position and orientation

```
r = sqrt(wx**2 + wy**2) - 0.35 #implemented okay
ss = wz - 0.75 #implemented okay

k1 = 1.25 #implemented okay
k2 = 1.5 #implemented okay

D = (k1**2 + k2**2 - (r**2 + ss**2))/(2 * k1 * k2) #implemented okay
K = (k1**2 + (r**2 + ss**2) - k2**2)/(2*sqrt(r**2 + ss**2)*k1) #implemented okay

# First three joint variables
theta1 = atan2(wy,wx).evalf()
theta2 = (pi/2 - atan2(ss,r) - atan2(sqrt(1 - K**2), K)).evalf()
theta3 = 1.53484 - atan2(sqrt(1 - D**2), D).evalf()

R36rpy = (R0_3.transpose() * Rrpy).evalf(subs={q1: theta1, q2: theta2, q3: theta3})

# Second three joint variables
theta4 = atan2(R36rpy[2,2], -R36rpy[0,2]).evalf()
theta5 = atan2(sqrt(1 - R36rpy[1,2]**2), R36rpy[1,2]).evalf()
theta6 = atan2(-R36rpy[1,1], R36rpy[1,0]).evalf()
```

3 Results & Conclusion

The FK verification of the transformation matrices revealed some small numerical errors, however, implementation of the `IK_server.py` worked as intended. The pick and place operation can be seen in Figures 10 and 11. A full video of the Kuka KR210 pick and place operation can be seen on the YouTube link provided below. Note the simulation execution is slow and you may like to watch this at 2 $\times$ speed.

`https://youtu.be/F2vZQHOaYb8`

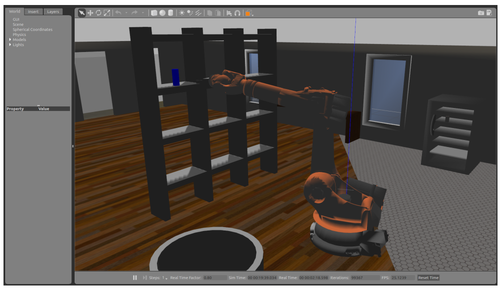


Figure 10: Kuka KR210 completing a pick trajectory.

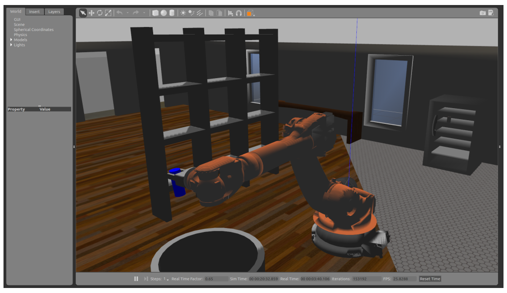


Figure 11: Kuka KR210 completing a place trajectory.

4 Further Enhancements

Since the joint angles are bound to $-\pi$ and π, this can lead to a complication in the arm movements when traversing from one reference point to another. The complication arises when the robot needs to make a small adjustment beyond $-\pi$ or π when it is at either of these boundries. The resulting movement sees the robot execute a full joint revolution, which can appear awkward, especially for the wrist joints. A simple example of this phenomena would be if the robot needed to move a joint to -182^o. Instead of performing this movement, the robot would instead send the joint to +178^o causing a full joint revolution.

Another bug that can be observed in the continuous mode of robot motion is that execution doesn't allow the robot enough time to actually grip the object, resulting in the object being left on the shelf in some instances. The final two improvements that could be undertaken are adding an error tracking visualisation, and speeding up the simulation, though these are not essential.

5 References

Denavit, J., Hartenberg, R. (1955). "A kinematic notation for lower-pair mechanisms based on matrices". Trans ASME J. Appl. Mech. 23: 215221.

6 Appendix A

A section of the KR210 urdf which lists the incremental information of the joint locations with respect to the base. This information was used to determine the DH parameter values of d_i and a_{i-1}.

```
<!-- joints -->
<joint name="fixed_base_joint" type="fixed">
  <parent link="base_footprint"/>
  <child link="base_link"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
<joint name="joint_1" type="revolute">
  <origin xyz="0 0 0.33" rpy="0 0 0"/>
  <parent link="base_link"/>
  <child link="link_1"/>
  <axis xyz="0 0 1"/>
  <limit lower="${-185*deg}" upper="${185*deg}" effort="300" velocity="${123*deg}"/>
</joint>
<joint name="joint_2" type="revolute">
  <origin xyz="0.35 0 0.42" rpy="0 0 0"/>
  <parent link="link_1"/>
  <child link="link_2"/>
  <axis xyz="0 1 0"/>
  <limit lower="${-45*deg}" upper="${85*deg}" effort="300" velocity="${115*deg}"/>
</joint>
<joint name="joint_3" type="revolute">
  <origin xyz="0 0 1.25" rpy="0 0 0"/>
  <parent link="link_2"/>
  <child link="link_3"/>
  <axis xyz="0 1 0"/>
  <limit lower="${-210*deg}" upper="${(155-90)*deg}" effort="300" velocity="${112*deg}"/>
</joint>
<joint name="joint_4" type="revolute">
  <origin xyz="0.96 0 -0.054" rpy="0 0 0"/>
  <parent link="link_3"/>
  <child link="link_4"/>
  <axis xyz="1 0 0"/>
  <limit lower="${-350*deg}" upper="${350*deg}" effort="300" velocity="${179*deg}"/>
</joint>
<joint name="joint_5" type="revolute">
  <origin xyz="0.54 0 0" rpy="0 0 0"/>
  <parent link="link_4"/>
  <child link="link_5"/>
  <axis xyz="0 1 0"/>
  <limit lower="${-125*deg}" upper="${125*deg}" effort="300" velocity="${172*deg}"/>
</joint>
<joint name="joint_6" type="revolute">
  <origin xyz="0.193 0 0" rpy="0 0 0"/>
  <parent link="link_5"/>
  <child link="link_6"/>
  <axis xyz="1 0 0"/>
  <limit lower="${-350*deg}" upper="${350*deg}" effort="300" velocity="${219*deg}"/>
</joint>
```

7 Appendix B

The code used to help verify that the DH coordinate frames had been applied correctly, and the DH parameters correctly determined. The script calculates the transformation matrices, and the final end effector pose, which can be compared with actual results from a simulator to ensure that the FK calculations match.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 23 22:35:58 2017

@author: shane
"""

from sympy import *
import tf

def R_x(q):
    M_x = Matrix([[ 1,      0,       0],
                  [ 0, cos(q), -sin(q)],
                  [ 0, sin(q),  cos(q)]])
    return M_x

def R_y(q):
    M_y = Matrix([[ cos(q), 0, sin(q)],
                  [      0, 1,      0],
                  [-sin(q), 0, cos(q)]])
    return M_y

def R_z(q):
    M_z = Matrix([[ cos(q), -sin(q), 0],
                  [ sin(q),  cos(q), 0],
                  [ 0,            0, 1]])
    return M_z

def transMat(q, alpha, d, a):
    T = 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

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

### Populate fixed DH parameter table
s = {alpha0:     0,    a0:      0,    d1:  0.75,
     alpha1: -pi/2,    a1:   0.35,    d2:     0,    q2: (q2 - pi/2),
     alpha2:     0,    a2:   1.25,    d3:     0,
     alpha3: -pi/2,    a3: -0.054,    d4:   1.5,
     alpha4:  pi/2,    a4:      0,    d5:     0,
     alpha5: -pi/2,    a5:      0,    d6:     0,
     alpha6:     0,    a6:      0,    d7: 0.303,    q7: 0}

### Create homogeneous transfroms between neighbouring links
T0_1 = transMat(q1, alpha0, d1, a0)
T1_2 = transMat(q2, alpha1, d2, a1)
T2_3 = transMat(q3, alpha2, d3, a2)
T3_4 = transMat(q4, alpha3, d4, a3)
T4_5 = transMat(q5, alpha4, d5, a4)
T5_6 = transMat(q6, alpha5, d6, a5)
T6_7 = transMat(q7, alpha6, d7, a6)

### Substitute DH parameters into the homogeneous transforms
T0_1 = T0_1.subs(s)
T1_2 = T1_2.subs(s)
T2_3 = T2_3.subs(s)
T3_4 = T3_4.subs(s)
T4_5 = T4_5.subs(s)
T5_6 = T5_6.subs(s)
T6_7 = T6_7.subs(s)

print('\nT0_1 =\n')
pprint(T0_1)
print('\nT1_2 =\n')
pprint(T1_2)
print('\nT2_3 =\n')
pprint(T2_3)
print('\nT3_4 =\n')
pprint(T3_4)
print('\nT4_5 = \n')
pprint(T4_5)
print('\nT5_6 =\n')
pprint(T5_6)
print('\nT6_7 =\n')
pprint(T6_7)

### Incrementally build the transform from the base link to the
### gripper link (intrinsic rotation = post-mult)
T0_2 = T0_1*T1_2
T0_3 = T0_2*T2_3
T0_4 = T0_3*T3_4
T0_5 = T0_4*T4_5
T0_6 = T0_5*T5_6
T0_7 = T0_6*T6_7

### Create the correction matrrix used to transform from urdf to DH
R_Z = Matrix([[-1,  0, 0, 0],
              [ 0, -1, 0, 0],
              [ 0,  0, 1, 0],
              [ 0,  0, 0, 1]])
```

```

R_Y = Matrix([[0, 0, -1, 0],
              [0, 1,  0, 0],
              [1, 0,  0, 0],
              [0, 0,  0, 1]])

T_corr = R_Z*R_Y # intrinsic rotation = post

### Numerically evaluate transforms
s_dict = {q1: 0, q2: 0, q3: 0, q4: 0, q5: 0, q6: 0} #test_initial
#s_dict = {q1: -0.65, q2: 0.45, q3: -0.35, q4: 0.91, q5: 0.78, q6: 0.5} #test_1
#s_dict = {q1: -0.79, q2: -0.11, q3: -2.33, q4: 1.94, q5: 1.14, q6: -3.68} #test_2
#s_dict = {q1: -2.99, q2: -0.12, q3: 0.94, q4: 4.06, q5: 1.29, q6: -4.12} #test_3
### (compared with the output of tf_echo)
print('\nT0_1 =\n')
pprint(T0_1.evalf(subs=s_dict))
print('\nT0_2 =\n')
pprint(T0_2.evalf(subs=s_dict))
print('\nT0_3 =\n')
pprint(T0_3.evalf(subs=s_dict))
print('\nT0_4 =\n')
pprint(T0_4.evalf(subs=s_dict))
print('\nT0_5 = \n')
pprint(T0_5.evalf(subs=s_dict))
print('\nT0_6 =\n')
pprint(T0_6.evalf(subs=s_dict))
print('\nT0_7 =\n')
pprint(T0_7.evalf(subs=s_dict))

### The corrected homogeneous tform from the base link to the end
### effector
T_total = T0_7*T_corr
print('\nT0_7_corrected =\n')
pprint(T_total.evalf(subs=s_dict))

print('\nRotation Matrix from Simulation = \n')
R = tf.transformations.quaternion_matrix([0.013, -0.229, 0.900, 0.368])
print(R)

print('\nThe transformation from 0 to 3:\n')
pprint(T3_4*T4_5*T5_6)
```

8 Appendix C

The final inverse kinematics server implementation.

```python
#!/usr/bin/env python2

# Copyright (C) 2017 Electric Movement Inc.
#
# This file is part of Robotic Arm: Pick and Place project for Udacity
# Robotics nano-degree program
#
# All Rights Reserved.

# Author: Harsh Pandya

# import modules
import rospy
import tf
from kuka_arm.srv import *
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from geometry_msgs.msg import Pose
from mpmath import *
from sympy import *


def handle_calculate_IK(req):
    rospy.loginfo("Received %s eef-poses from the plan" %  len(req.poses))
    if  len(req.poses) < 1:
        print "No valid poses received"
        return -1
    else:
        ### Your FK code here
        # Create symbols
        q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8')
        d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8')
        a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7')
        alpha0, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6 = symbols('alpha0:7')

        # Create Modified DH parameters
        s = {alpha0:      0,     a0:       0,     d1: 0.75,
             alpha1: -pi/2,     a1:    0.35,     d2:    0,     q2: (q2 - pi/2),
             alpha2:      0,     a2:    1.25,     d3:    0}

        # Define Modified DH Transformation matrix
        T0_1 = transMat(q1, alpha0, d1, a0)
        T1_2 = transMat(q2, alpha1, d2, a1)
        T2_3 = transMat(q3, alpha2, d3, a2)

        T0_1 = T0_1.subs(s)
        T1_2 = T1_2.subs(s)
        T2_3 = T2_3.subs(s)

        # Create individual transformation matrices
        T0_2 = T0_1*T1_2
        T0_3 = T0_2*T2_3

        # Specify the intrinsic rotation matrix for correcting from DH to urdf
        R_corr = Matrix([[0,  0, 1],
                         [0, -1, 0],
                         [1,  0, 0]])

        # Extract rotation matrices from the transformation matrices
        R0_3 = T0_3[0:3,0:3]
        ###

        # Initialize service response
        joint_trajectory_list = []
        for x in  xrange(0,  len(req.poses)):
            # IK code starts here
            joint_trajectory_point = JointTrajectoryPoint()

                # Extract end-effector position and orientation from request
                # px,py,pz = end-effector position
                # roll, pitch, yaw = end-effector orientation
            px = req.poses[x].position.x
            py = req.poses[x].position.y
            pz = req.poses[x].position.z

            (roll, pitch, yaw) = tf.transformations.euler_from_quaternion(
                [req.poses[x].orientation.x, req.poses[x].orientation.y,
                    req.poses[x].orientation.z, req.poses[x].orientation.w])
            # Define Modified DH Transformation matrix

            ### Your IK code here
                # Compensate for rotation discrepancy between DH parameters and Gazebo
            Rrpy = R_z(yaw) * R_y(pitch) * R_x(roll) * R_corr
                #
                # Calculate joint angles using Geometric IK method
            wx = px - (0.303) * Rrpy[0,2] # x-coord of wrist position
            wy = py - (0.303) * Rrpy[1,2] # y-coord of wrist position
            wz = pz - (0.303) * Rrpy[2,2] # z-coord of wrist position

            r = sqrt(wx**2 + wy**2) - 0.35 #implemented okay
            ss = wz - 0.75 #implemented okay

            k1 = 1.25 #implemented okay
            k2 = 1.5 #implemented okay

            D = (k1**2 + k2**2 - (r**2 + ss**2))/(2 * k1 * k2) #implemented okay
            K = (k1**2 + (r**2 + ss**2) - k2**2)/(2*sqrt(r**2 + ss**2)*k1) #implemented okay

            # First three joint variables
            theta1 = atan2(wy,wx).evalf()
            theta2 = (pi/2 - atan2(ss,r) - atan2(sqrt(1 - K**2), K)).evalf()
            theta3 = 1.53484 - atan2(sqrt(1 - D**2), D).evalf()

            R36rpy = (R0_3.transpose() * Rrpy).evalf(subs={q1: theta1, q2: theta2, q3: theta3})

            # Second three joint variables
            theta4 = atan2(R36rpy[2,2], -R36rpy[0,2]).evalf()
            theta5 = atan2(sqrt(1 - R36rpy[1,2]**2), R36rpy[1,2]).evalf()
            theta6 = atan2(-R36rpy[1,1], R36rpy[1,0]).evalf()
                #
            ###

            # Populate response for the IK request
            # In the next line replace theta1,theta2...,theta6 by your joint angle variables
            joint_trajectory_point.positions = [theta1, theta2, theta3, theta4, theta5, theta6]
            joint_trajectory_list.append(joint_trajectory_point)

        rospy.loginfo("length of Joint Trajectory List: %s" %  len(joint_trajectory_list))
        return CalculateIKResponse(joint_trajectory_list)


def transMat(q, alpha, d, a):
    T = 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

def R_x(q):
    M_x = Matrix([[ 1,      0,       0],
                  [ 0, cos(q), -sin(q)],
                  [ 0, sin(q),  cos(q)]])
    return M_x

def R_y(q):
    M_y = Matrix([[ cos(q), 0, sin(q)],
                  [      0, 1,       0],
                  [-sin(q), 0, cos(q)]])
    return M_y

def R_z(q):
    M_z = Matrix([[ cos(q), -sin(q), 0],
                  [ sin(q),  cos(q), 0],
                  [ 0,            0, 1]])
    return M_z


def IK_server():
    # initialize node and declare calculate_ik service
    rospy.init_node('IK_server')
    s = rospy.Service('calculate_ik', CalculateIK, handle_calculate_IK)
    print "Ready to receive an IK request"
    rospy.spin()

if __name__ == "__main__":
    IK_server()
```