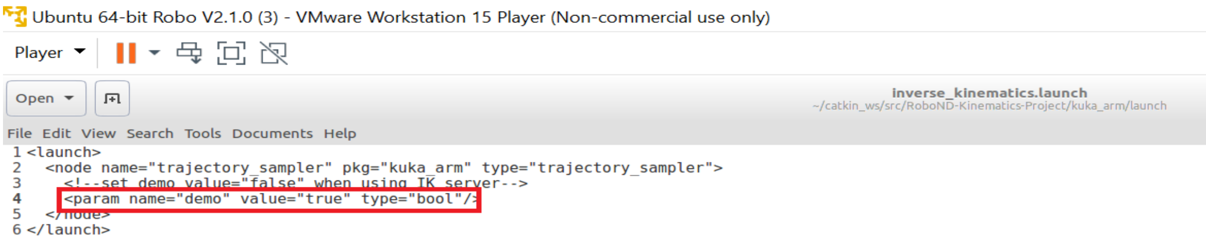
Project: Kinematics Pick & Place

Kinematic Analysis

Criteria 1:

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**.

- *Forward Kinematics Analysis*: Launching Forward Kinematics with Kuka arm in demo mode by setting demo flag to “true”




```
<launch>
  <node name="trajectory_sampler" pkg="kuka_arm" type="trajectory_sampler">
    <!--set demo value="false" when using IK server-->
    <param name="demo" value="true" type="bool"/>
  </node>
</launch>
```

Script 1: Inverse_inematics.launch

Launching Forward Kinematics in Gazebo and Rviz in Demo Mode:

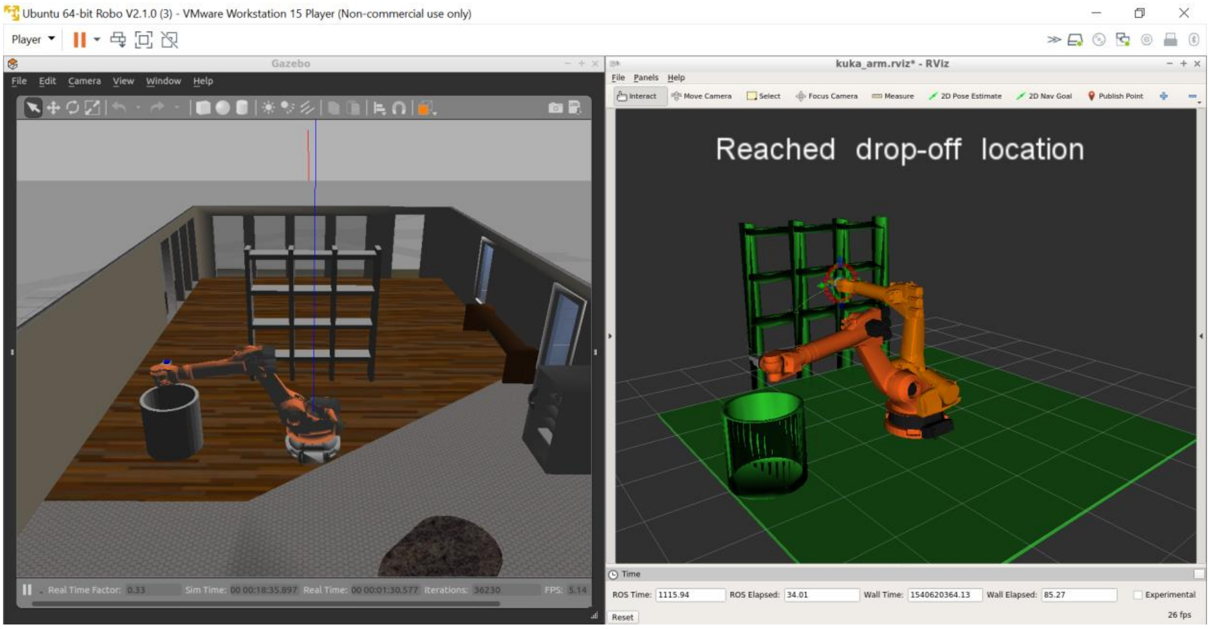


Fig 1: Kuka KR210 in Demo Mode

- *kr210.urdf.xacro file:* Analysis on joints section of kr210.urdf.xacro file is done by taking into consideration, individual joint <origin xyz> with respect to its <parent link>. In the following screenshots of urdf files, highlighted are the joint distances which are taken into account for DH parameter table construction.

```xml
<!-- 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>
```

Script 2: kr210.urdf.xacro – Base Joint, Joint 1, Joint 2, Joint 3

```xml
-<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>
```

Script 3: kr210.urdf.xacro – Joint 4, Joint 5, Joint 6

```xml
-<joint name="right_gripper_finger_joint" type="prismatic">
   <origin rpy="0 0 0" xyz="0.15 -0.0725 0"/>
   <parent link="gripper_link"/>
   <child link="right_gripper_finger_link"/>
   <axis xyz="0 1 0"/>
   <limit effort="100" lower="-0.01" upper="0.06" velocity="0.05"/>
   <dynamics damping="0.7"/>
 </joint>
-<joint name="left_gripper_finger_joint" type="prismatic">
   <origin rpy="0 0 0" xyz="0.15 0.0725 0"/>
   <parent link="gripper_link"/>
   <child link="left_gripper_finger_link"/>
   <axis xyz="0 -1 0"/>
   <limit effort="100" lower="-0.01" upper="0.06" velocity="0.05"/>
   <dynamics damping="0.7"/>
 </joint>
-<joint name="gripper_joint" type="fixed">
   <parent link="link_6"/>
   <child link="gripper_link"/>
   <origin xyz="0.11 0 0" rpy="0 0 0"/>
 </joint>
```

Script 4: kr210.urdf.xacro – Gripper Link

Based on the above information, the joint distances are plotted in the figure as follows:

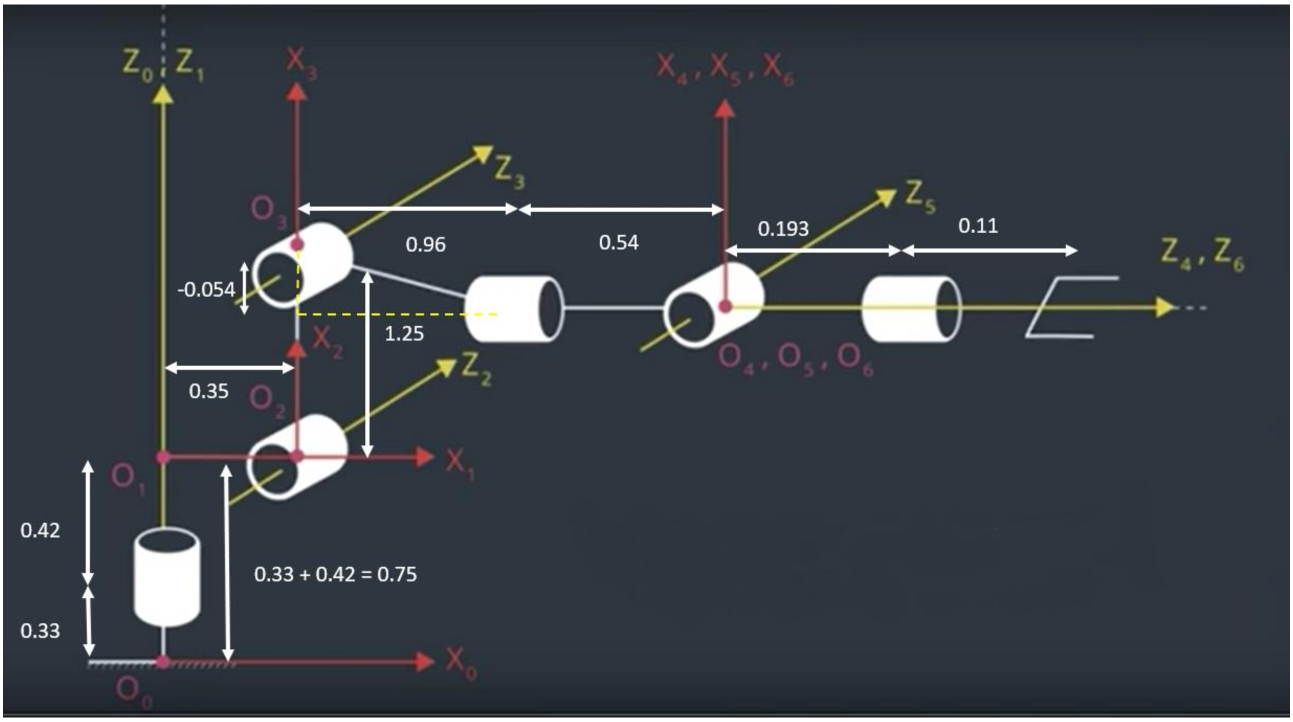


Fig 2: Schematic Diagram of Robotic arm with Joint Length representation

Based on Script 2, Script 3, Script 4 from kuka_arm urdf file and Fig 2, the values of twist angle, link length, link offset and joint angle are calculated. Fig 3 shows the hand written calculation of DH parameters in which individual joint section are drawn and respective DH parameters are calculated.

As represented in Fig 3, :

- α_{i-1} (twist angle) = angle between Z_{i-1} and Z_i measured about X_{i-1}
- a_{i-1} (link length) = distance from Z_{i-1} and Z_i measured along X_{i-1} where X_{i-1} is perpendicular to both Z_{i-1} and Z_i
- d_i (link offset) = signed distance from X_{i-1} and X_i measured along Z_i
- ϑ_i (joint angle) = angle between X_{i-1} and X_i measured about Z_i

O_0 is the origin of Fixed base Joint. O_1, O_2, O_3 are the origins of Revolute Joints J_1, J_2, J_3. Revolute Joints J_4, J_5, J_6 have their Origins O_4, O_5, O_6 placed at Joint J_5. In the end, there is a Prismatic Gripper Link connected to J_6 for holding the Target.

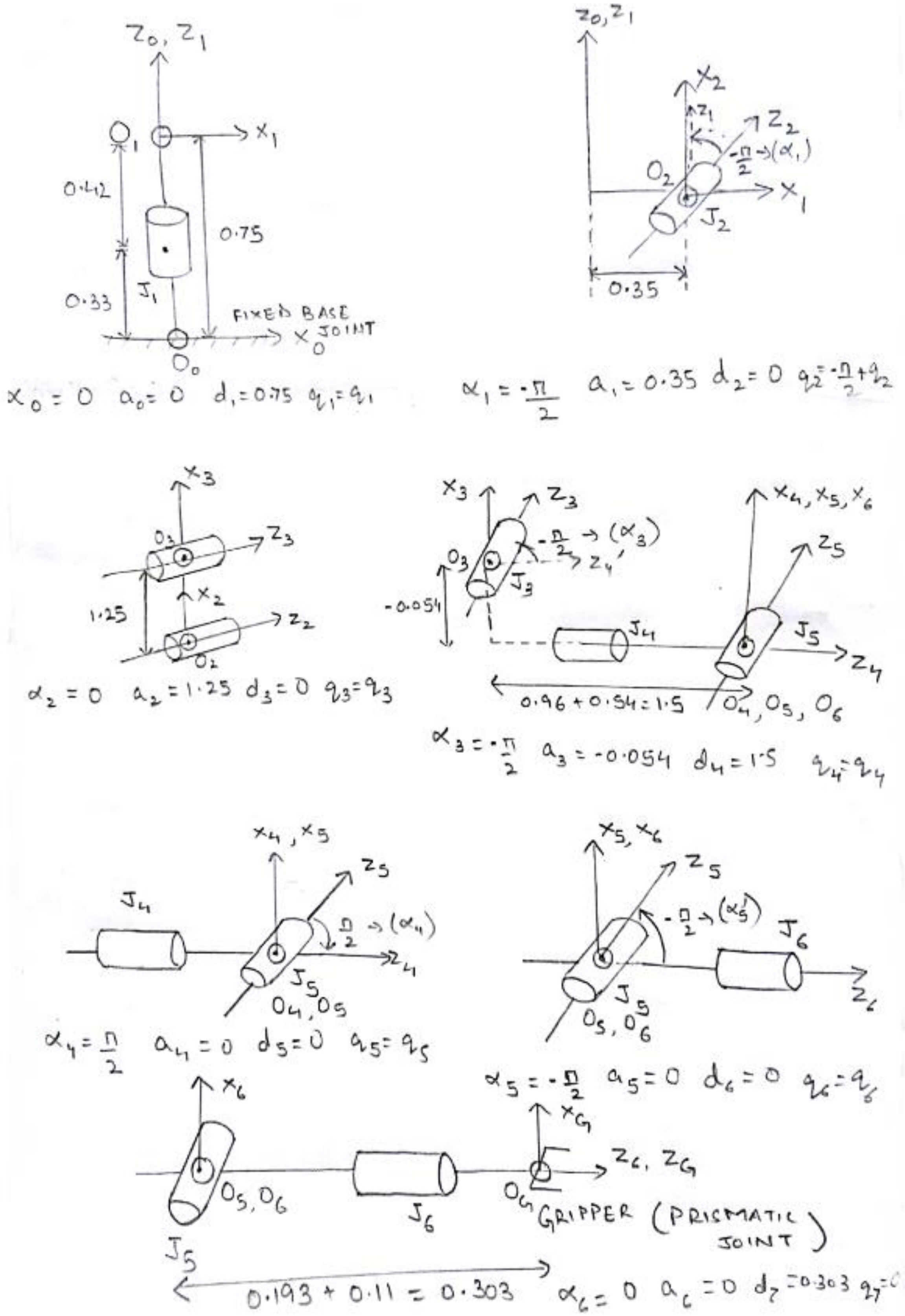


Fig 3: Hand written calculation of DH Parameters

The DH Parameter table following the above analysis, looks like this:

Links	alpha (i-1)	a (i-1)	d (i)	q (i)
0-1	0	0	0.75	q1
1-2	-pi/2	0.35	0	-pi/2 + q2
2-3	0	1.25	0	q3
3-4	-pi/2	-0.054	1.5	q4
4-5	pi/2	0	0	q5
5-6	-pi/2	0	0	q6
6-EE	0	0	0.303	0

Table 1: DH Parameter Table

In the IK_server.py code, this has been used like this:

```
#Joint Angle Symbols
q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8')

#Link Offset Symbols
d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8')

#Link Length symbols
a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7')

#Twist Angle symbols
alpha0, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6 = symbols('alpha0:7')

# Modified DH parameters
DH_Table = {    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: 1.5,        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}
```

Script 5: Implementation of DH Parameter table in "IK_server.py" code

Criteria 2:

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.**

From the DH Parameter table constructed, the values are substituted in Transformation Matrix below:

$$
{}^{i-1}_{\ \ i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

For all the individual links, the transformation matrices are calculated, following which the product of all such transformation matrices are calculated to form a complete Transformation Matrix from Fixed Base Joint till gripper link.

$$ {}^{0}_{N}T = {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T \ldots {}^{N-1}_{N}T $$

The calculated individual Transformation Matrices are:

$$ {}^{0}_{1}T = \begin{bmatrix} \cos(q1) & -\sin(q1) & 0 & a_0 \\ \sin(q1)\cos(\alpha_0) & \cos(q1)\cos(\alpha_0) & -\sin(\alpha_0) & -\sin(\alpha_0)\,d1 \\ \sin(q1)\sin(\alpha_0) & \cos(q1)\sin(\alpha_0) & \cos(\alpha_0) & \cos(\alpha_0)\,d1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{1}_{2}T = \begin{bmatrix} \cos(q_2) & -\sin(q_2) & 0 & a_1 \\ \sin(q_2)\cos(\alpha_1) & \cos(q_2)\cos(\alpha_1) & -\sin(\alpha_1) & -\sin(\alpha_1)\,d2 \\ \sin(q_2)\sin(\alpha_1) & \cos(q_2)\sin(\alpha_1) & \cos(\alpha_1) & \cos(\alpha_1)\,d2 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Fig 4: Hand written calculation of Transformation Matrices for links 0-1 and 1-2

The rest of the transformation matrices are calculated likewise as shown in Fig 4, for all the links.

Product of all these individual matrices together gives the overall transformation matrix:

$$ {}^{0}_{EE}T = {}^{0}_{1}T \times {}^{1}_{2}T \times {}^{2}_{3}T \times {}^{3}_{4}T \times {}^{4}_{5}T \times {}^{5}_{6}T \times {}^{6}_{EE}T $$

Fig 5: Hand written calculation of Transformation Matrices

The same is implemented in IK_server.py code like this:

```
# Define Modified DH Transformation matrix
def TF_Matrix(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

# Create individual transformation matrices
T0_1 = TF_Matrix(alpha0, a0, d1, q1).subs(DH_Table)
T1_2 = TF_Matrix(alpha1, a1, d2, q2).subs(DH_Table)
T2_3 = TF_Matrix(alpha2, a2, d3, q3).subs(DH_Table)
T3_4 = TF_Matrix(alpha3, a3, d4, q4).subs(DH_Table)
T4_5 = TF_Matrix(alpha4, a4, d5, q5).subs(DH_Table)
T5_6 = TF_Matrix(alpha5, a5, d6, q6).subs(DH_Table)
T6_EE = TF_Matrix(alpha6, a6, d7, q7).subs(DH_Table)

T0_EE = T0_1 * T1_2 * T2_3 * T3_4 * T4_5 * T5_6 * T6_EE
```

Script 6: Implementation of Individual Transforms and overall Transformation matrix from base till Gripper link in "IK_server.py" code

A rotational matrix is created using the orientation and pose of Kuka arm. The Roll Pitch and Yaw, along x, y, z axes generates a Rotational Matrix:

```
# Define Roatation Matrix from End Effector Yaw, Roll, Pitch

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

ROT_x = Matrix([[ 1,     0,       0],                   #Roll
                [ 0,     cos(r), -sin(r)],
                [ 0,     sin(r), cos(r)]])

ROT_y = Matrix([[ cos(p),        0,   sin(p)],          #Pitch
                [       0,        1,        0],
                [-sin(p),        0,   cos(p)]])

ROT_z = Matrix([[ cos(y), -sin(y),        0],           #Yaw
                [ sin(y),  cos(y),        0],
                [ 0,              0,        1]])

ROT_EE = ROT_z * ROT_y * ROT_x
```

Script 7: Implementation of Rotational Matrix from request response of Kuka arm in "IK_server.py" code

To handle the discrepancy in angles: The correctional values of 180 and -90 degrees in z and y axes between Gazebo and xacro files are implemented to handle the discrepancy in angle difference.

Criteria 3:

3. **Decouple Inverse Kinematics problem into Inverse Position Kinematics and inverse Orientation Kinematics; doing so derive the equations to calculate all individual joint angles.**
    - *Inverse Kinematics Analysis*: Joint angles are calculated based on End Effector Position and Orientation from Request Response.

```
# Calculate End Effetor position
EE = Matrix([[px],
             [py],
             [pz]])

# Wrist Center Calculation
WC = EE - (0.303) * ROT_EE[:,2]
```

Script 8: Implementation of End Effector poses and calculation of Wrist Centre in "IK_server.py" code

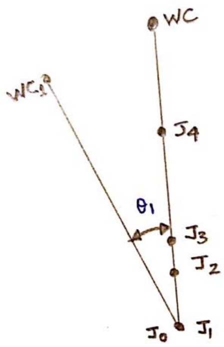


Fig 6: Hand written calculation of q1 (Theta1) Top View

The calculation of Joint angle Theta 1:

Theta 1 = $\tan^{-1}(WC_y/WC_x)$ [1]

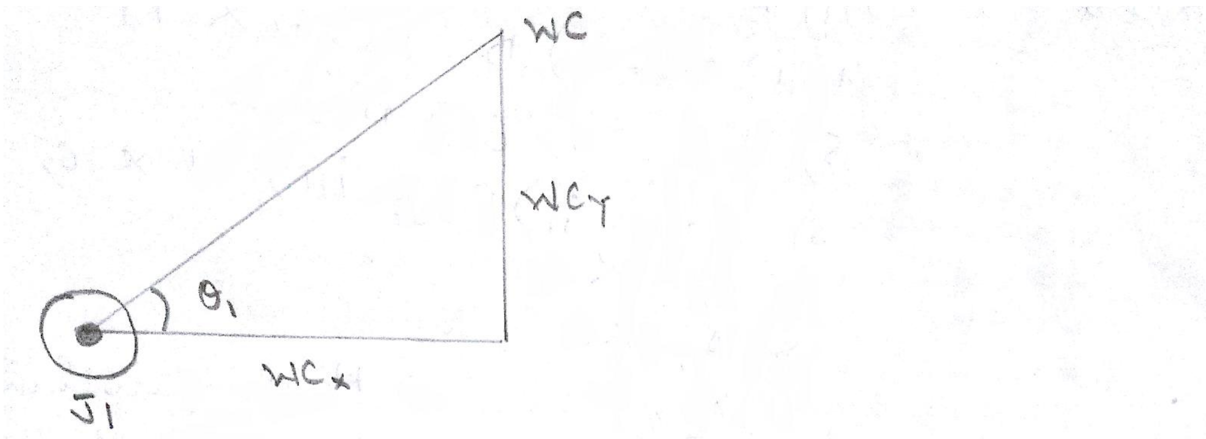


Fig 7: Hand written calculation of q1 (Theta1)

The calculation of Joint angle Theta 2 and Theta 3 is:

Theta 2 = $\frac{\pi}{2} - a - \tan^{-1}\left((WC_z - 0.75)/(\sqrt{(WC_x^2 + WC_y^2)} - 0.35)\right)$ [2]

Theta 3 = $\frac{\pi}{2} - b - \tan^{-1}(0.054/1.5))$ [3]

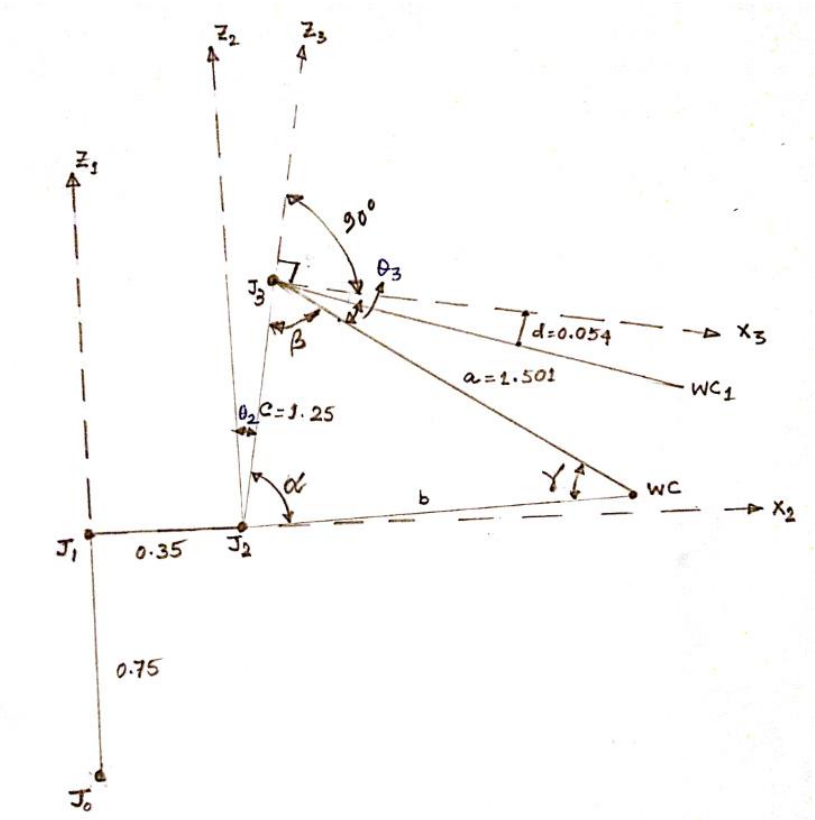


Fig 8: Hand written calculation of q2 and q3 (Theta2 & Theta3)

In the above Fig 8, J0, J1 and J2 are the Joint positions, whereas WC is the Wrist Center. In the diagram depiction 'α is angle a', 'β is angle b' and 'ϒ is angle c' from equations [2] and [3].

Calculation of Rotation Matrix from Transformation matrix as shown in Fig 5, from Link 0 till Link 3 is:

R0_3 = T0_1[0:3,0:3] * T1_2[0:3,0:3] * T2_3[0:3,0:3]

Rotational matrix from Link 3 to 6 is defined from above like this:

ROT_EE = R0_3 * R3_6

R3_6 = inv(R0_3) * ROT_EE

The calculation of Joint angle Theta 4, Theta 5 and Theta 6 are:

Theta 4 = $\tan^{-1}(\mathrm{R3\_6}[2,2] \,/(-\mathrm{R3\_6}[0,2]))$ [4]

Theta 5 = $\tan^{-1}\left((\sqrt{(\mathrm{R3\_6}[0,2]^2 + \mathrm{R3\_6}[2,2]^2)} \,/\mathrm{R3\_6}[1,2])\right)$ [5]

Theta 6 = $\tan^{-1}(-R3\_6[1,1]/R3\_6[1,0])$ [6]

Handling Multiple Solutions for Computation of Theta_4, Theta_5 and Theta_6

The calculation of Theta5 gives us:

Theta 5 = $\tan^{-1}\left((\sqrt{(\mathrm{R3\_6}[0,2]^2 + \mathrm{R3\_6}[2,2]^2)} \,/\mathrm{R3\_6}[1,2])\right)$

Here the value- $\sqrt{(\mathrm{R3\_6}[0,2]^2 + \mathrm{R3\_6}[2,2]^2)}$ may have the outputs in both positive and negative signs. By default in the code 'IK_server.py' I've used positive values. Although a negative value of square root output can also be an option. This value of Theta5 impacts angle Theta4 and Theta6 as well as all of them have similar origins O4, O5 and O6 shown in Fig 2.

For this to reflect, the Theta4 and Theta6 values can be changed with an opposite sign on R3_6 components in Theta4 and Theta6 depending on angle Theta5.

For E.g., if Theta5 is a positive value then,

Theta 4 = $\tan^{-1}(\mathrm{R3\_6}[2,2] \,/(-\mathrm{R3\_6}[0,2]))$ [7]

Theta 6 = $\tan^{-1}(-R3\_6[1,1]/R3\_6[1,0])$ [8]

Else, if Theta5 is a negative value then,

Theta 4 = $\tan^{-1}(-\mathrm{R3\_6}[2,2] \,/(\mathrm{R3\_6}[0,2]))$ [9]

Theta 6 = $\tan^{-1}(R3\_6[1,1]/-R3\_6[1,0])$ [10]

The joint angle calculations are shown in IK_server.py code like this:

```
# Calculate joint angles using Geometric IK method
    theta1 = atan2(WC[1],WC[0])

    side_b = sqrt(pow((sqrt(WC[0] * WC[0] + WC[1] * WC[1]) - 0.35), 2) + pow((WC[2] - 0.75), 2))

    if ((1.25 + 1.501)> side_b) & ((1.25 + side_b)> 1.501) & ((side_b + 1.501)> 1.25):

        angle_a = acos((side_b*side_b + 1.25*1.25 - 1.501*1.501) / (2*side_b*1.25))
        angle_b = acos((1.501*1.501 + 1.25*1.25 - side_b*side_b) / (2*1.501*1.25))
        angle_c = acos((1.501*1.501 + side_b*side_b - 1.25*1.25) / (2*1.501*side_b))

        theta2 = pi/2 - angle_a - atan2(WC[2]-0.75 , sqrt(WC[0]*WC[0] + WC[1]*WC[1])-0.35)
        theta3 = pi/2 - (angle_b  + atan2(0.054,1.5))

        R0_3 = R0_3_Temp.evalf(subs={q1:theta1, q2:theta2, q3:theta3})

        R3_6 = R0_3.inv("LU") * ROT_EE

        theta4 = atan2(R3_6[2,2], -R3_6[0,2])

        theta5 = atan2(sqrt(R3_6[0,2]*R3_6[0,2] + R3_6[2,2]*R3_6[2,2]), R3_6[1,2])

        theta6 = atan2(-R3_6[1,1], R3_6[1,0])
```

Script 9: Calculation of Joint angles

Project Implementation

Criteria 4:

4. **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. A screenshot of the completed pick and place process is included.**

In the IK_server.py code, all the joint angle implementation are done to properly guide the Kuka KR210 throughout the pick and place cycle. The various stages of robust implementation is described below:

I've added iteration counts for calculating number of Inverse Kinematics Loop.

```
# Calculating Iteration Count for Kuka arm
# To reach Target Location and from Target Location to Drop-off location
class count_state():
    def __init__(self):
        self.count = 0
Count = count_state()

```

```

        Count.count = Count.count + 1                           #Incrementing Iteration count for number of Inverse Kinematics loop

```

Script 10: Incremental Loops for every Inverse Kinematics Calculation

Forward Kinematics Code: In the forward Kinematics code, a transformation matrix is created from fixed base joint till gripper link. Based on, the values of twist angle, link length, link offset and joint angle, individual transformation matrices are obtained, which then produces an overall transformation matrix till End Effector.

```
        ### Forward Kinematics code

        ## Create symbols
        #Joint Angle Symbols
        q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8')

        #Link Offset Symbols
        d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8')

        #Link Length symbols
        a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7')

        #Twist Angle symbols
        alpha0, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6 = symbols('alpha0:7')

        # Modified DH parameters
        DH_Table = {    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: 1.5,        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}

        # Define Modified DH Transformation matrix
        def TF_Matrix(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

        # Create individual transformation matrices
        T0_1 = TF_Matrix(alpha0, a0, d1, q1).subs(DH_Table)
        T1_2 = TF_Matrix(alpha1, a1, d2, q2).subs(DH_Table)
        T2_3 = TF_Matrix(alpha2, a2, d3, q3).subs(DH_Table)
        T3_4 = TF_Matrix(alpha3, a3, d4, q4).subs(DH_Table)
        T4_5 = TF_Matrix(alpha4, a4, d5, q5).subs(DH_Table)
        T5_6 = TF_Matrix(alpha5, a5, d6, q6).subs(DH_Table)
        T6_EE = TF_Matrix(alpha6, a6, d7, q7).subs(DH_Table)

        T0_EE = T0_1 * T1_2 * T2_3 * T3_4 * T4_5 * T5_6 * T6_EE
```

Script 11: Calculation of Transformation Matrix

The End effector pose and orientation are extracted from Request response. The correctional values of 180 and -90 degrees in z and y axes between Gazebo and xacro files are implemented following which Wrist Centre of Kuka_arm is calculated.

```
# 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])

# Compensate for rotation discrepancy between DH parameters and Gazebo
Rot_Error = ROT_z.subs(y, radians(180)) * ROT_y.subs(p, radians(-90))
ROT_EE = ROT_EE * Rot_Error
ROT_EE=  ROT_EE.subs({'r':roll, 'p':pitch, 'y':yaw})

# Calculate End Effetor position
EE = Matrix([[px],
            [py],
            [pz]])

# Wrist Center Calculation
WC = EE - (0.303) * ROT_EE[:,2]
```

Script 11: Wrist Center calculation

Mathematical Rule on Triangles: For a given Triangle, as shown below, the sum of any 2 sides is greater than 3rd side. For proper calculation of angles opposite to sides A,B and C using inverse Cos, the value must range between 0-1. For this to satisfy,

Side_A + Side_B > Side_C ;

Side_B + Side_C > Side_A ;

Side_C + Side_A > Side_C

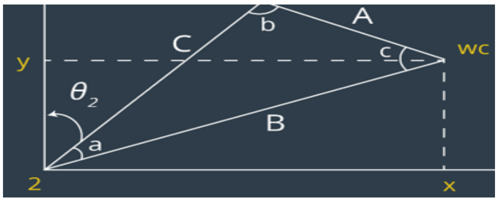


Fig 9: Mathematical Rule on Triangles

Here, Side_a = sqrt(1.5^2 + 0.054^2)

Side_c = 1.25

side_b = sqrt(((sqrt(WC[0] * WC[0] + WC[1] * WC[1]) - 0.35)^2) + ((WC[2] - 0.75)^2))

With this and law of cosines, the values for angles a, b and c are calculated, which are then used to derive theta1 , theta2 and theta3.

```
### Inverse Kinematics code

# Calculate joint angles using Geometric IK method

    theta1 = atan2(WC[1],WC[0])

    side_b = sqrt(pow((sqrt(WC[0] * WC[0] + WC[1] * WC[1]) - 0.35), 2) + pow((WC[2] - 0.75), 2))        #side_a = 1.501 #side_c = 1.25

    if ((1.25 + 1.501)> side_b) & ((1.25 + side_b)> 1.501) & ((side_b + 1.501)> 1.25):         # Mathematically, (sum of 2 sides of a triangle) > 3rd side

        angle_a = acos((side_b*side_b + 1.25*1.25 - 1.501*1.501) / (2*side_b*1.25))
        angle_b = acos((1.501*1.501 + 1.25*1.25 - side_b*side_b) / (2*1.501*1.25))
        angle_c = acos((1.501*1.501 + side_b*side_b - 1.25*1.25) / (2*1.501*side_b))

        theta2 = pi/2 - angle_a - atan2(WC[2]-0.75 , sqrt(WC[0]*WC[0] + WC[1]*WC[1])-0.35)
        theta3 = pi/2 - (angle_b  + atan2(0.054,1.5))

        R0_3 = R0_3_Temp.evalf(subs={q1:theta1, q2:theta2, q3:theta3})                 # Evaluating Rotation Matrix

        R3_6 = R0_3.inv("LU") * ROT_EE

        theta4 = atan2(R3_6[2,2], -R3_6[0,2])

        theta5 = atan2(sqrt(R3_6[0,2]*R3_6[0,2] + R3_6[2,2]*R3_6[2,2]), R3_6[1,2])

        theta6 = atan2(-R3_6[1,1], R3_6[1,0])

        # Forward Kinematics evaluation with derived joint angles
        FK = T0_EE.evalf(subs={q1: theta1, q2: theta2, q3: theta3, q4: theta4, q5: theta5, q6: theta6})

        # Error Calculation of End Effector position by comparing with end-effector poses, received as input
        ee_x_e = abs(FK[0,3]-px)                # End Effector error offset along x axis
        ee_y_e = abs(FK[1,3]-py)                # End Effector error offset along y axis
        ee_z_e = abs(FK[2,3]-pz)                # End Effector error offset along z axis

        # Overall end-effector offset
        ee_offset = sqrt(ee_x_e**2 + ee_y_e**2 + ee_z_e**2)
        print(ee_offset)
```

Script 11: Inverse Kinematics Code

After calculation of Joint angles, forward kinematics is calculated with derived theta values. The End effector position from this is then compared with End effector poses from Request Response. The end-effector offset is then calculated as:

ee_offset = sqrt(ee_x_e^2 + ee_y_e^2 + ee_z_e^2)

where, ee_x_e— Error in x direction, ee_y_e— Error in y direction, ee_z_e— Error in z direction

For the calculated ee_offset below a set threshold, the calculated joint angles are sent through service request for the Kuka arm to perform. Here in code , I've considered 0.01 as the threshold value. For the last joint angle calculation in the trajectory of End Effector positions, an analysed value is given to ensure that the target falls under the bin.

There are 9 shelf positions, where randomly at any location the target spawn could be present in a given cycle. To ensure the gripper arm grasps the target correctly, corresponding joint angles for individual target locations are determined and passed through the end trajectory point, before the Kuka arm grasps the target. This complete process of calculation of Joint angles based on error offset of End effector and pre analysis on last joint angle calculation ensures a clean pick and place operation for the robotic arm.

```
## Initializtion

# Define Joint angles for Target Spawn locations for the END trajectory point
#(theta1i,theta3i,theta4i,theta5i,theta6i, theta7i, theta8i, theta9i) are the joint angles for target sawn locations 1 to 9
# where i ranges from 1 to 6, while moving from initial location to target spawn location )

theta1 = 0; theta2=0; theta3=0; theta4=0; theta5=0; theta6=0;
theta61 = -0.460788639682829; theta62 = -1.4083946795092 + pi/2; theta63 = -1.49141431818708 + pi/2; theta64 = -1.0235104903567 + pi; theta65 = 0.495695716305110; theta66 = -pi + 0.972495930022601
theta41 = 0.466568380940290; theta42 = -1.40588840950334 + pi/2; theta43 = -1.49933810543832 + pi/2; theta44 = -pi + 1.12482883075263; theta45 = 0.521928836050086; theta46 = -1.06984849582538 + pi
theta91 = -0.460620971603582; theta92 = -1.41600678927374 + pi/2; theta93 = -2.01520122842406 + pi/2; theta94 =  0.912819391695163; theta95 = 0.538148746753259; theta96 = -0.855105912394959
theta71 = 0.460651236883725; theta72 = -1.41603690529533 + pi/2; theta73 = -2.015158587665 + pi/2; theta74 = -0.913449451374226; theta75 = 0.537900753574318; theta76 = 0.855694311433023
theta81 = -4.02985219558988e-6; theta82 = -1.59017523803714 + pi/2; theta83 = -1.81801235853875 + pi/2; theta84 = -8.71196937061172e-5; theta85 = 0.259284638790722 ; theta86 = -7.35675934196834e-6
theta11 = 0.461944532159549; theta12 = -1.11836645228303 + pi/2; theta13 = -1.31463177224073 + pi/2; theta14 = -pi + 0.725225037468592; theta15 =  0.676095580016169; theta16 = -0.62726276643435 + pi
theta51 = -0.00229054389158514; theta52 = -1.57765784191287 + pi/2; theta53 = -1.32851692115255 + pi/2; theta54 = -0.086229482068646 + pi; theta55 = 0.181470117894438; theta56 = -pi + 0.48845652679379
theta31 = -0.466518984306794; theta32 = -1.08486429666301 + pi/2; theta33 = -1.31281014892211 + pi/2; theta34 = -0.618700551351137 + pi; theta35 = 0.886705067213612; theta36 = -pi + 0.441222764426423

# Define Joint angle for the end trajectory point
# (thetaei) is the joint angle while moving from target spawn location to bin)

thetae1=-1.3683474710526 + pi; thetae2 = -0.965048732979534 + pi/2; thetae3 = -2.09403813236587 + pi/2; thetae4 = -1.55313297331450; thetae5 = -1.3697836855635 + pi; thetae6 = -1.48647561055299 + pi
```

Script 12: Pre-analysis on Joint angle for end point of trajectory

```
            # Error Calculation of End Effector position by comparing with end-effector poses, received as input
            ee_x_e = abs(FK[0,3]-px)                                    # End Effector error offset along x axis
            ee_y_e = abs(FK[1,3]-py)                                    # End Effector error offset along y axis
            ee_z_e = abs(FK[2,3]-pz)                                    # End Effector error offset along z axis

            # Overall end-effector offset
            ee_offset = sqrt(ee_x_e**2 + ee_y_e**2 + ee_z_e**2)
            print ("\nEnd effector error for x position is: %04.8f" % ee_x_e)
            print ("End effector error for y position is: %04.8f" % ee_y_e)
            print ("End effector error for z position is: %04.8f" % ee_z_e)
            print ("Overall end effector offset is: %04.8f units \n" % ee_offset)
```

Script 13: Calculation of error offset wrt Forward Kinematics calculation for every trajectory point

```
        # Populate response for the IK request

        if (x == len(req.poses)-1) & (Target_Spawn==6):                 #Last joint angle calculation for Target Spawn location: 6
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta61, theta62, theta63, theta64, theta65, theta66]
                joint_trajectory_list.append(joint_trajectory_point)

        elif (x == len(req.poses)-1) & (Target_Spawn==4):               #Last joint angle calculation for Target Spawn location: 4
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta41, theta42, theta43, theta44, theta45, theta46]
                joint_trajectory_list.append(joint_trajectory_point)

        elif (x == len(req.poses)-1) & (Target_Spawn==9):               #Last joint angle calculation for Target Spawn location: 9
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta91, theta92, theta93, theta94, theta95, theta96]
                joint_trajectory_list.append(joint_trajectory_point)

        elif (x == len(req.poses)-1) & (Target_Spawn==7):               #Last joint angle calculation for Target Spawn location: 7
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta71, theta72, theta73, theta74, theta75, theta76]
                joint_trajectory_list.append(joint_trajectory_point)

        elif (x == len(req.poses)-1) & (Target_Spawn==8):               #Last joint angle calculation for Target Spawn location: 8
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta81, theta82, theta83, theta84, theta85, theta86]
                joint_trajectory_list.append(joint_trajectory_point)

        elif (x == len(req.poses)-1) & (Target_Spawn==1):               #Last joint angle calculation for Target Spawn location: 1
            if ((Count.count%2) == 1):

                joint_trajectory_point.positions = [theta11, theta12, theta13, theta14, theta15, theta16]
                joint_trajectory_list.append(joint_trajectory_point)
```

```
        elif (x == len(req.poses)-1) & ((Count.count%2) == 0):                          #Last joint angle calculation towards Bin
            if (ReachCount == 0):
                if (ee_offset<0.01):
                    joint_trajectory_point.positions = [theta1, theta2, theta3, thetae4, thetae5, thetae6]
                    joint_trajectory_list.append(joint_trajectory_point)
                else:
                    joint_trajectory_point.positions = [thetae1, thetae2, thetae3, thetae4, thetae5, thetae6]
                    joint_trajectory_list.append(joint_trajectory_point)

                zt = (joint_trajectory_list[len(joint_trajectory_list)-2])
                Theta_6 = zt.positions[5]
                Theta6Diff = (np.absolute(Theta_6-thetae6))

                if (Theta6Diff>= 1.7):
                    joint_trajectory_point.positions = [thetae1, thetae2, thetae3, thetae4, thetae5, Theta_6]
                    joint_trajectory_list.append(joint_trajectory_point)

        elif (ee_offset<0.01):                                                            #Allow joint angles for which error is below threshold
            joint_trajectory_point.positions = [theta1, theta2, theta3, theta4, theta5, theta6]
            joint_trajectory_list.append(joint_trajectory_point)
            ee_offset_list.append(ee_offset)
            t = time()
            time_list.append(t)
```

Script 12: Populate Response for Inverse Kinematics Request with Joint Angle

A setup of running code, with calculated inverse kinematics in Gazebo and Rviz:

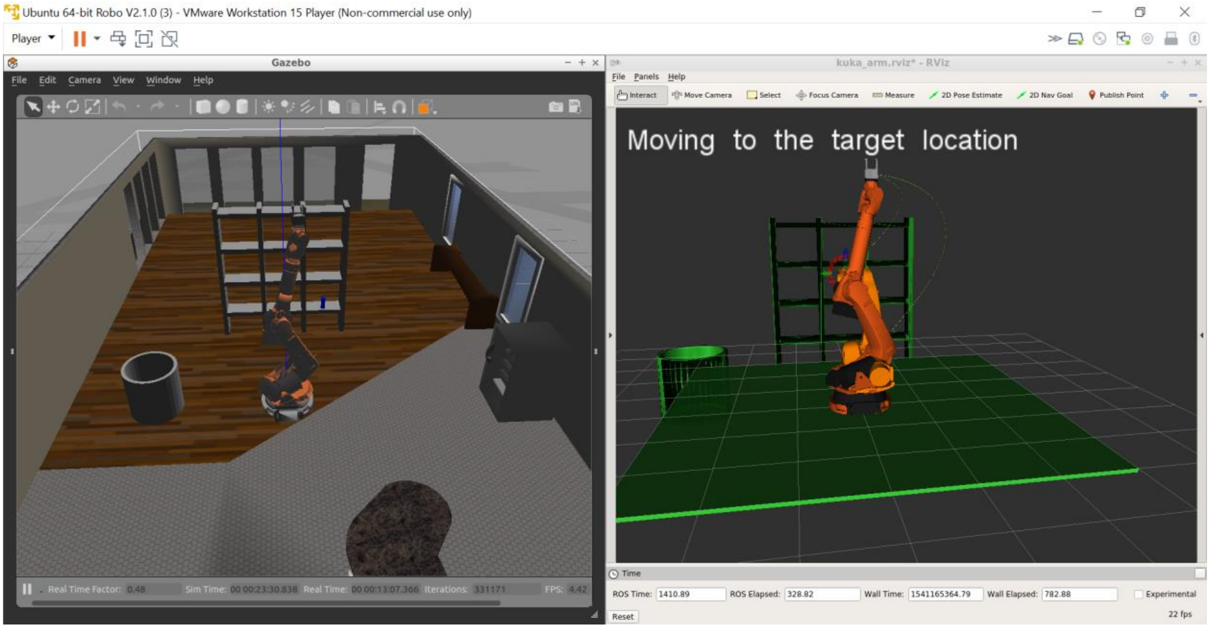


Fig 10: Setup of running code in Gazebo and Rviz

Results and Analysis: Below, I’ve shown the results with example figures. In this project, I’ve accepted the challenge for error plots of trajectory points, discussed later.

Successful Pick and place operation in 1st cycle:

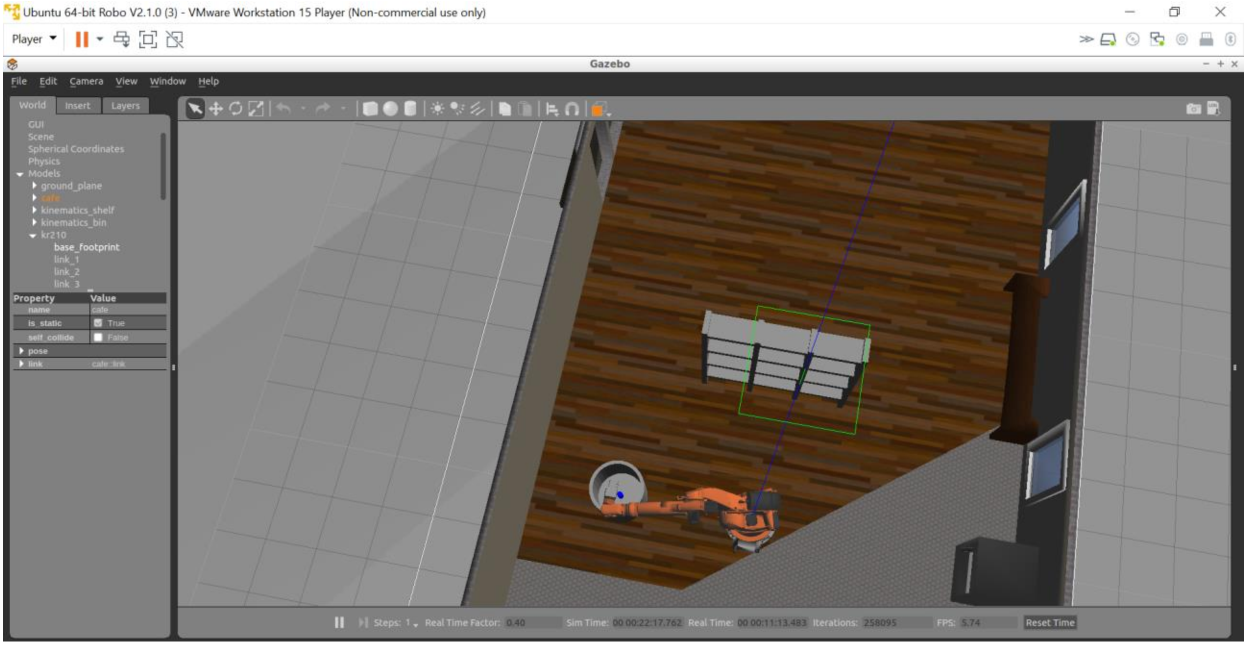


Fig 11: Pick and place operation in 1st cycle

For fulfilling the project submission criteria, successful pick and place operation of 8/10 cycles, has to be met. As shown below, is the example figure of 8 target spawn in the bin after 8th cycle:

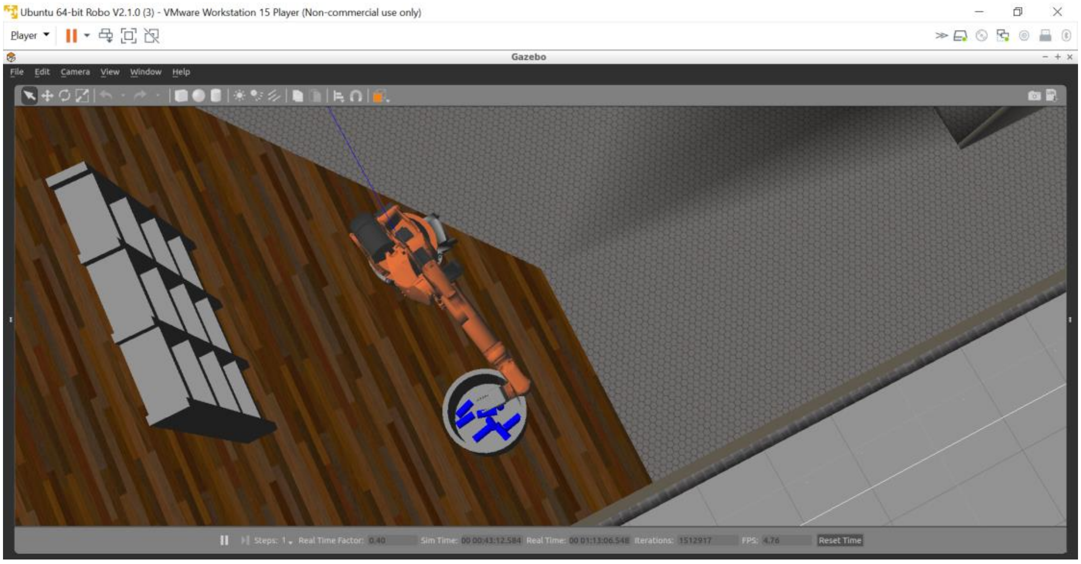


Fig 11: Successful Pick and place operation in 8/10 cycles

The code is able to move the robotic arm for 10/10 cycles cleanly as well, below is the example figure, where 10 target spawns are grasped from their random location and placed inside the bin:

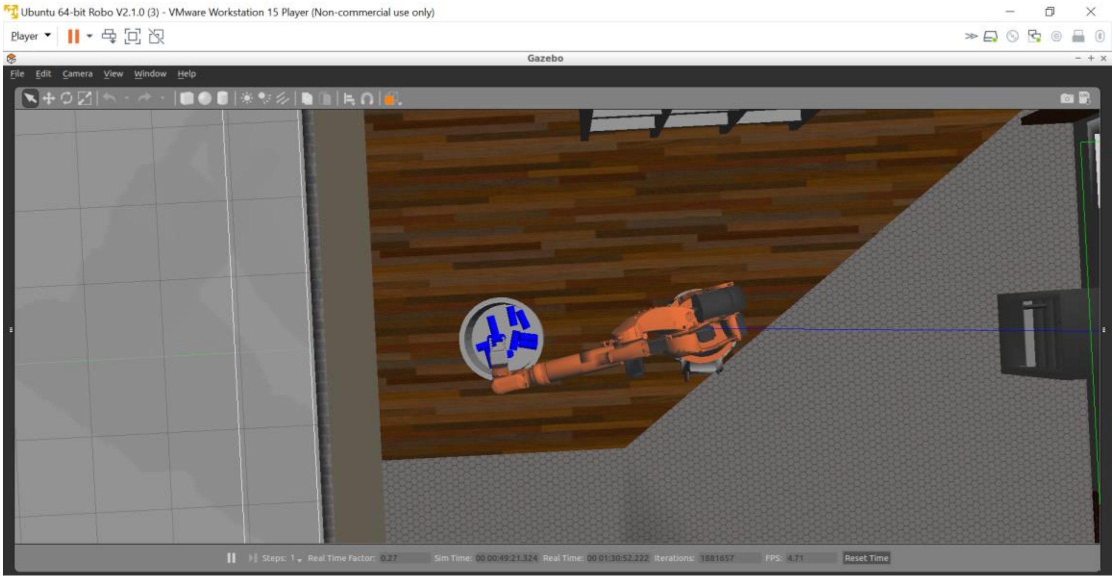


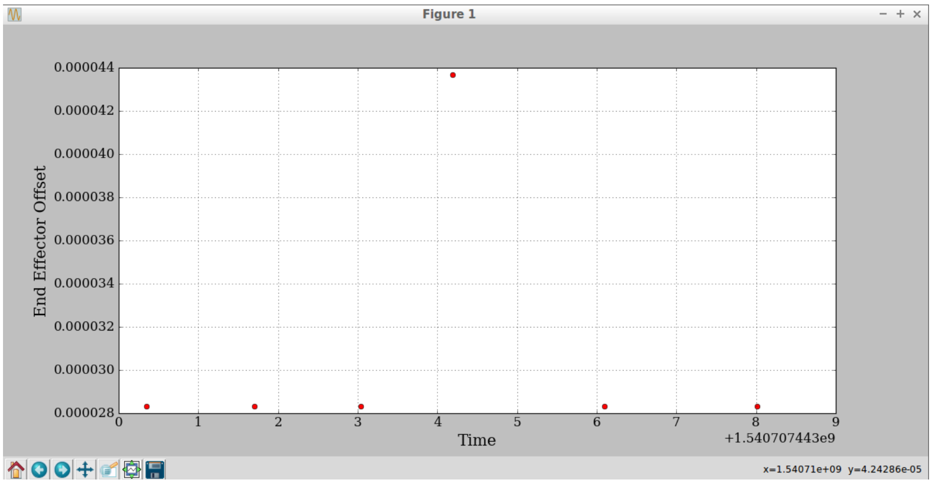
Fig 12: Successful Pick and place operation in 10/10 cycles

Challenge Accepted: I've accepted the Udacity challenge of plotting error values. The code for which is as below:

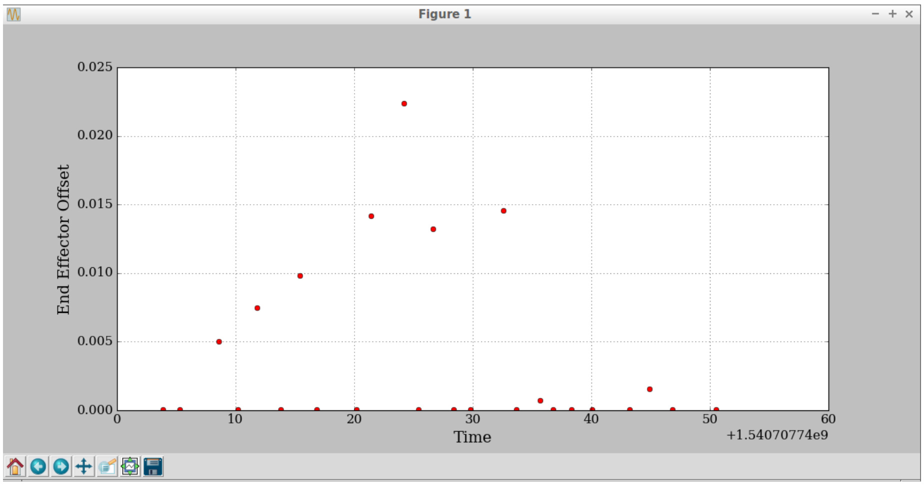
```
    ## Plotting error outputs of end effector w.r.t to received responses (Close the plot window for code to run further)
    ## UNCOMMENT BELOW LINES TO CHECK THE ERROR PLOTS FOR CHALLENGE COMPLETION

    #ee_offset_ar = np.array([ee_offset_list])
    #time_list_ar = np.array([time_list])
    #plt.figure(figsize=(15,7))
    #plt.rc('font', family='serif', size=15)
    #plt.plot(time_list_ar,ee_offset_ar, marker='o', color='r')
    #plt.xlabel('Time',fontsize=18)
    #plt.ylabel('End Effector Offset',fontsize=18)
    #plt.grid(True)
    #plt.show()

    return CalculateIKResponse(joint_trajectory_list)

```

Note: This code is commented for now to allow smooth operation. To check the error plots, please uncomment these lines. After every inverse kinematics joint angle calculation cycle, the plot for the error will be visible on screen, please close it for the code and the Kuka arm to execute further steps.

The error values are also printed on terminal. The error values usually remains at 2.831 * 10^-5 units. The maximum error at times goes till 0.3 units. On the basis of a set threshold of 0.01, only the joint angles calculated below this threshold are allow to pass. Hence the error plots are also limited till the set threshold as maximum value. Error plot of Pick and place Operation in 1st cycle, from starting location of Kuka_arm till the target location:



Error Plot of Pick and place Operation in 1st cycle, from target location till drop-off location:



As it could be seen from the error plots , the maximum error goes till 0.0025 in the executed run cycle.

Conclusion: In the given project, the robotic arm is able to pick and place and follow the trajectory given, for 8/10 cycles at any time. Along with this , the error calculation for the end effector off the trajectory is very low in the order of 10^-5 mostly. This owes to the accurate Inverse Kinematics calculation using DH parameters and other logics for grasping target and placing in bin smoothly.