Robotic Arm: Pick And Place

PROJECT 2

Christopher J. Rukas | Robotic Software Engineer | November 2, 2018

Project Description

In this lesson a Kuka KR210 robotic arm with six degrees of freedom is used to locate a small object from a shelf and place it in a bin. The primary focus of the project is to develop the inverse kinematics required to control a six degree of freedom arm. The SymPy library is also introduced as a form of simplifying trigonometry equations.

Denavit-Hartenberg Parameters

The DH (Denavit-Hartenberg) Parameters is a notation convention for reference frames in a chain of "spatial linkages." A six degree of freedom manipulator is represented of the form given below, where

α_i represents the twist angle between each coordinate system

a_i represents the length of the link (distance between perpendicular axes, to be specific)

d_i represents the offset of the link and

q_i represents the current angle of the joint.

Denavit-Hartenberg Parameters			
α_i	a_i	d_i	q_i
α_1	a_i	d_1	q_1
α_2	a_1	d_2	q_2
α_3	a_2	d_3	q_3
α_4	a_3	d_4	q_4
α_5	a_4	d_5	q_5
α_6	a_5	d_6	q_6
α_7	a_6	d_7	q_7

The DH Parameters are populated using information available from the Kuka KR210. First the KR210.URDF.XACRO file is used to identify all of the joint positions. The joint positions are defined in the joints section of the URDF file.

URDF File Data									
	Location			**Orientation**			**Axis**		
	x	**y**	**z**	**r**	**p**	**y**	**x**	**y**	**z**
Origin	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Joint 1	0.0	0.0	0.33	0.0	0.0	0.0	0.0	0.0	1.0
Joint 2	0.35	0.0	0.42	0.0	0.0	0.0	0.0	1.0	0.0
Joint 3	0.0	0.0	1.25	0.0	0.0	0.0	0.0	1.0	0.0
Joint 4	0.96	0.0	-0.054	0.0	0.0	0.0	1.0	0.0	0.0
Joint 5	0.54	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
Joint 6	0.193	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
EE	0.11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Location defines x,y,z position of component. Orientation defines a fixed axis roll in radians. Axis determines the direction of rotation for revolute joints, direction of translation for prismatic joints.									

The coordinate system utilized does not need to follow the URDF file, and is in fact simplified by strategically locating the coordinate systems to minimize the number of populated parameters. As such, when moving from the URDF file to the DH Parameter list link lengths are combined.

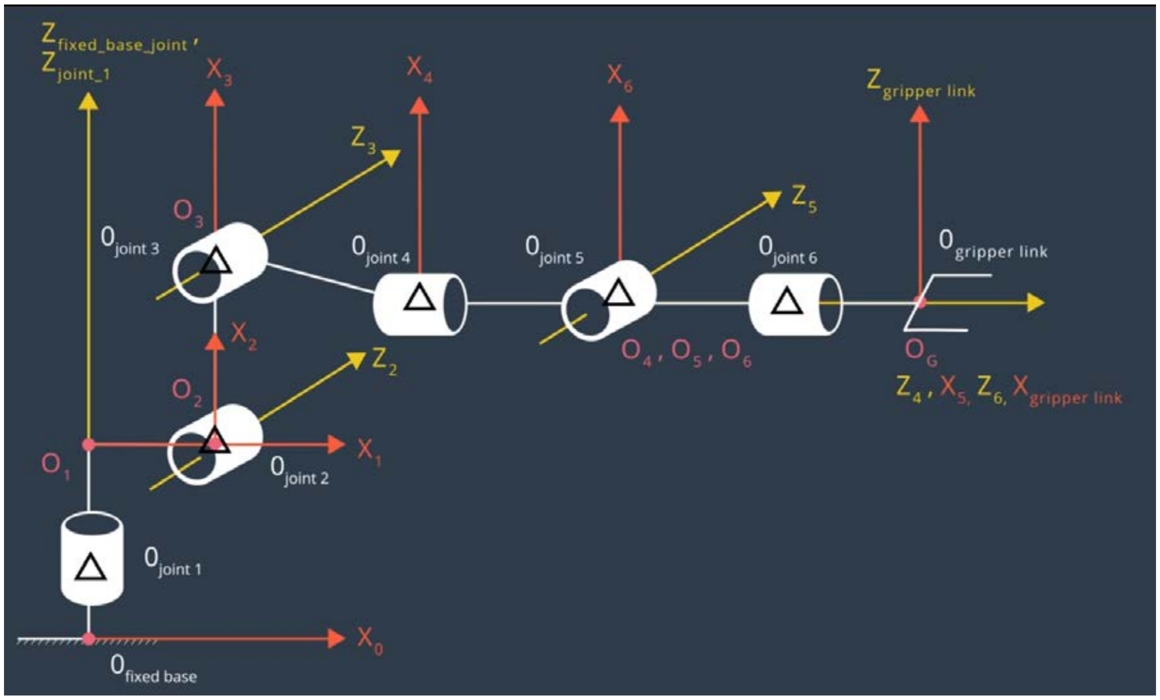
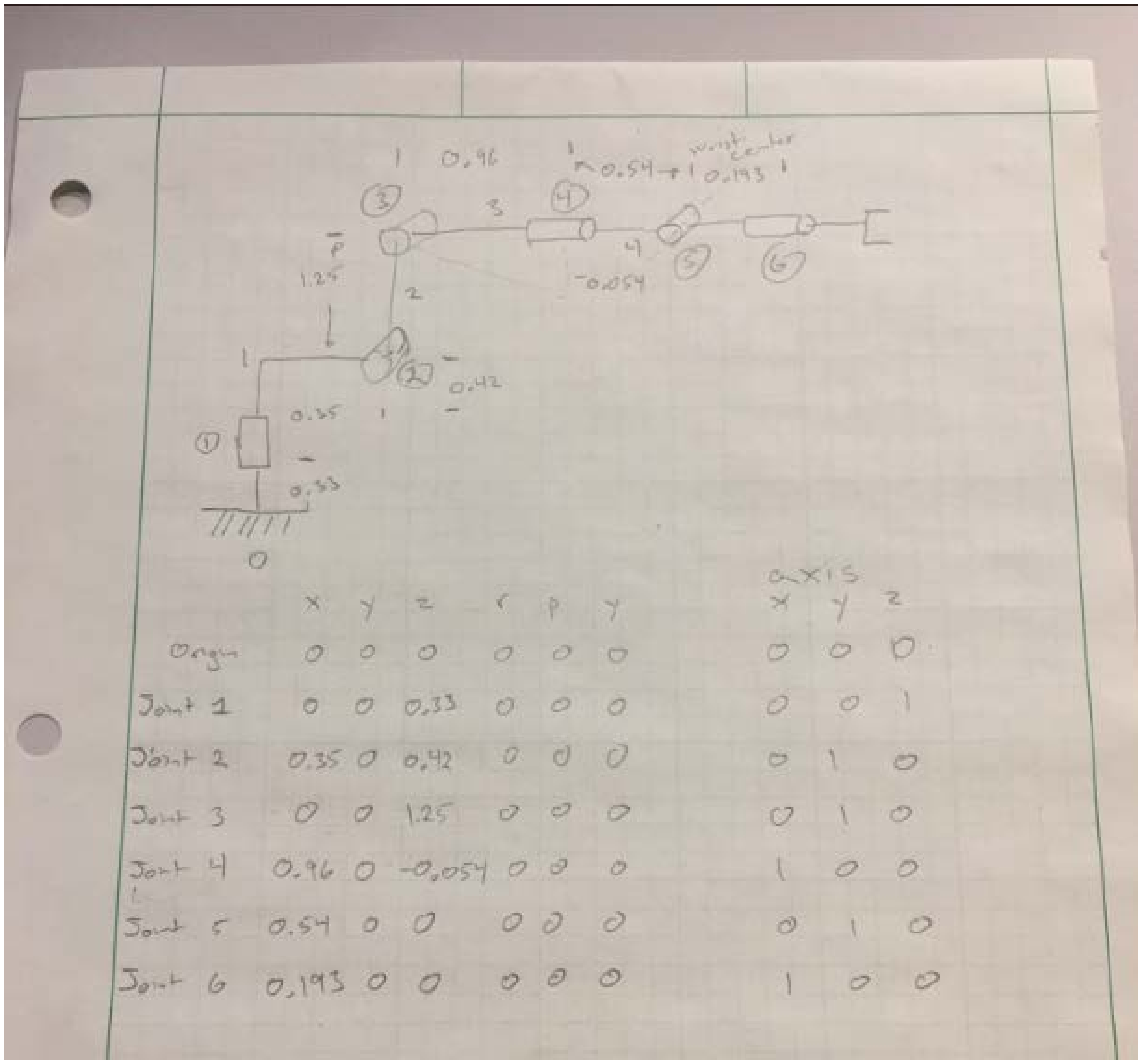


Figure 1: Manipulator Axes



	x	y	z	r	p	y	axis x	axis y	axis z
Origin	0	0	0	0	0	0	0	0	0
Joint 1	0	0	0.33	0	0	0	0	0	1
Joint 2	0.35	0	0.42	0	0	0	0	1	0
Joint 3	0	0	1.25	0	0	0	0	1	0
Joint 4	0.96	0	-0.054	0	0	0	1	0	0
Joint 5	0.54	0	0	0	0	0	0	1	0
Joint 6	0.193	0	0	0	0	0	1	0	0

Figure 2: Manipulator Diagram

Denavit-Hartenberg Parameters				
	$\boldsymbol{\alpha_i}$	$\boldsymbol{a_i}$	$\boldsymbol{d_i}$	$\boldsymbol{q_i}$
1	0.0	0	0.75	q_1
2	$-\frac{\pi}{2.0}$	0.35	0.0	$q_2 - \frac{\pi}{2.0}$
3	0.0	1.25	0.0	q_3
4	$-\frac{\pi}{2.0}$	-0.054	1.50	q_4
5	$\frac{\pi}{2.0}$	0.0	0.0	q_5
6	$-\frac{\pi}{2.0}$	0.0	0.0	q_6
7	0.0	0.0	0.303	0.0

Transformation Matrices

Utilizing the DH parameters the transformation matrices are easily generated between each joint. The general form of the Rotation and Translational matrix is as follows:

$$T_n = \begin{bmatrix} \cos(q) & -\sin(q) & 0 & a \\ \sin(q)*\cos(\alpha) & \cos(q)*\cos(\alpha) & -\sin(\alpha) & -\sin(\alpha)*d \\ \sin(q)*\sin(\alpha) & \cos(q)*\sin(\alpha) & \cos(\alpha) & \cos(\alpha)*d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And by populating parameters, the rotation and transformation matrix between each joint:

$$T_{0\,1} = \begin{bmatrix} \cos(q_1) & -\sin(q) & 0 & 0 \\ \sin(q_1)*\cos(0) & \cos(q_1)*\cos(0) & -\sin(0) & -\sin(0)*0.75 \\ \sin(q_1)*\sin(0) & \cos(q_1)*\sin(0) & \cos(0) & \cos(0)*0.75 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{1\,2} = \begin{bmatrix} \cos(q_2-\frac{\pi}{2}) & -\sin(q_2-\frac{\pi}{2}) & 0 & 0.35 \\ \sin\left(q_2-\frac{\pi}{2}\right)*\cos(-\frac{\pi}{2}) & \cos\left(q_2-\frac{\pi}{2}\right)*\cos(-\frac{\pi}{2}) & -\sin\left(-\frac{\pi}{2}\right) & 0 \\ \sin\left(q_2-\frac{\pi}{2}\right)*\sin(-\frac{\pi}{2}) & \cos\left(q_2-\frac{\pi}{2}\right)*\sin\left(-\frac{\pi}{2}\right) & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{2\,3} = \begin{bmatrix} \cos(q_3) & -\sin(q_3) & 0 & 1.25 \\ \sin(q_3)*\cos(0) & \cos(q_3)*\cos(0) & -\sin(0) & 0 \\ \sin(q_3)*\sin(0) & \cos(q_3)*\sin(0) & \cos(0) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{3\,4} = \begin{bmatrix} \cos(q_4) & -\sin(q_4) & 0 & -0.054 \\ \sin(q_4)*\cos(-\frac{\pi}{2}) & \cos(q_4)*\cos(-\frac{\pi}{2}) & -\sin\left(-\frac{\pi}{2}\right) & -\sin\left(-\frac{\pi}{2}\right)*1.50 \\ \sin(q_4)*\sin(-\frac{\pi}{2}) & \cos(q_4)*\sin\left(-\frac{\pi}{2}\right) & \cos(-\frac{\pi}{2}) & \cos\left(-\frac{\pi}{2}\right)*1.50 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{4\,5} = \begin{bmatrix} \cos(q_5) & -\sin(q_5) & 0 & 0 \\ \sin(q_5)*\cos(\frac{\pi}{2}) & \cos(q_5)*\cos(\frac{\pi}{2}) & -\sin\left(\frac{\pi}{2}\right) & 0 \\ \sin(q_5)*\sin(\frac{\pi}{2}) & \cos(q_5)*\sin\left(\frac{\pi}{2}\right) & \cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{5\,6} = \begin{bmatrix} \cos(q_6) & -\sin(q_6) & 0 & 0 \\ \sin(q_6) * \cos(-\frac{\pi}{2}) & \cos(q_6) * \cos(-\frac{\pi}{2}) & -\sin\left(-\frac{\pi}{2}\right) & 0 \\ \sin(q_6) * \sin(-\frac{\pi}{2}) & \cos(q_6) * \sin\left(-\frac{\pi}{2}\right) & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{6\,E} = \begin{bmatrix} \cos(q_7) & -\sin(q_7) & 0 & 0 \\ \sin(q_7) * \cos(0) & \cos(q_7) * \cos(0) & -\sin(0) & -\sin(0) * 0.303 \\ \sin(q_7) * \sin(0) & \cos(q_7) * \sin(0) & \cos(0) & \cos(0) * 0.303 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note: The matrices could be simplified by populating values for sine and cosine.

All of these transformations may be multiplied in order from the base to the end effector to create a final transformation.

Authors note: The rubric states a transformation matrix shall be provided from the Base to the End effector. For the author to keep his sanity, the SymPy output is provided as raw text after simplifying the various transformations.

```
Matrix([

[((sin(q1)*sin(q4) + sin(q2 + q3)*cos(q1)*cos(q4))*cos(q5) + sin(q5)*cos(q1)*cos(q2 +
q3))*cos(q6) + (sin(q1)*cos(q4) - sin(q4)*sin(q2 + q3)*cos(q1))*sin(q6), -
((sin(q1)*sin(q4) + sin(q2 + q3)*cos(q1)*cos(q4))*cos(q5) + sin(q5)*cos(q1)*cos(q2 +
q3))*sin(q6) + (sin(q1)*cos(q4) - sin(q4)*sin(q2 + q3)*cos(q1))*cos(q6), -
(sin(q1)*sin(q4) + sin(q2 + q3)*cos(q1)*cos(q4))*sin(q5) + cos(q1)*cos(q5)*cos(q2 + q3),
-0.303*sin(q1)*sin(q4)*sin(q5) + 1.25*sin(q2)*cos(q1) - 0.303*sin(q5)*sin(q2 +
q3)*cos(q1)*cos(q4) - 0.054*sin(q2 + q3)*cos(q1) + 0.303*cos(q1)*cos(q5)*cos(q2 + q3) +
1.5*cos(q1)*cos(q2 + q3) + 0.35*cos(q1)],

[((sin(q1)*sin(q2 + q3)*cos(q4) - sin(q4)*cos(q1))*cos(q5) + sin(q1)*sin(q5)*cos(q2 +
q3))*cos(q6) - (sin(q1)*sin(q4)*sin(q2 + q3) + cos(q1)*cos(q4))*sin(q6), ((-
sin(q1)*sin(q2 + q3)*cos(q4) + sin(q4)*cos(q1))*cos(q5) - sin(q1)*sin(q5)*cos(q2 +
q3))*sin(q6) - (sin(q1)*sin(q4)*sin(q2 + q3) + cos(q1)*cos(q4))*cos(q6), (-sin(q1)*sin(q2
+ q3)*cos(q4) + sin(q4)*cos(q1))*sin(q5) + sin(q1)*cos(q5)*cos(q2 + q3),
1.25*sin(q1)*sin(q2) - 0.303*sin(q1)*sin(q5)*sin(q2 + q3)*cos(q4) - 0.054*sin(q1)*sin(q2
+ q3) + 0.303*sin(q1)*cos(q5)*cos(q2 + q3) + 1.5*sin(q1)*cos(q2 + q3) + 0.35*sin(q1) +
0.303*sin(q4)*sin(q5)*cos(q1)],

[                                        (-sin(q5)*sin(q2 + q3) + cos(q4)*cos(q5)*cos(q2 +
q3))*cos(q6) - sin(q4)*sin(q6)*cos(q2 + q3),
(sin(q5)*sin(q2 + q3) - cos(q4)*cos(q5)*cos(q2 + q3))*sin(q6) - sin(q4)*cos(q6)*cos(q2
+ q3),                          -sin(q5)*cos(q4)*cos(q2 + q3) - sin(q2 + q3)*cos(q5),
-0.303*sin(q5)*cos(q4)*cos(q2 + q3) - 0.303*sin(q2 + q3)*cos(q5) - 1.5*sin(q2 + q3) +
1.25*cos(q2) - 0.054*cos(q2 + q3) + 0.75],

[
0,
0,                                                        0,
1]])
```

The matrix above is generated via substituting the values of the DH parameters into the generic Rotation and Transform matrix for each joint, and multiplying the transformations together in sequence from the base to the end effector. Rotations and translations are non-communicative and must be applied in order.

SymPy is used to simplify the equations.

REFERENCE FRAME CORRECTION

Gazebo utilizes its own coordinate system in the URDF file. This coordinate system does not match what was drawn for the DH parameters.

The following rotation matrices are known.

$$ROT_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(r) & -\sin(r) \\ 0 & \sin(r) & \cos(r) \end{bmatrix}$$

$$ROT_Y = \begin{bmatrix} \cos(\mathrm{p}) & 0 & \sin(p) \\ 0 & 1 & 0 \\ -\sin(p) & 0 & \cos(p) \end{bmatrix}$$

$$ROT_Z = \begin{bmatrix} \cos(\mathrm{y}) & -\sin(y) & 0 \\ \sin(y) & \cos(\mathrm{y}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where r, p, and y represent roll, pitch, and yaw respectively.

The axes are corrected by 180 degrees in yaw (Z axis) and -90 degrees in pitch (Y axis).

Inverse Kinematics

In this project we are given a position and an orientation of the end effect. Since the size of the end effector (wrist to gripper) is known, the wrist location is determinable. When the wrist center is known, a rotation angle of the Kuka arm is discoverable. The rotation angle is determined by projecting the wrist center onto the X and Y axes.

Using trigonometry, the base angle is determined. $\tan(\theta) = \frac{y}{x}$

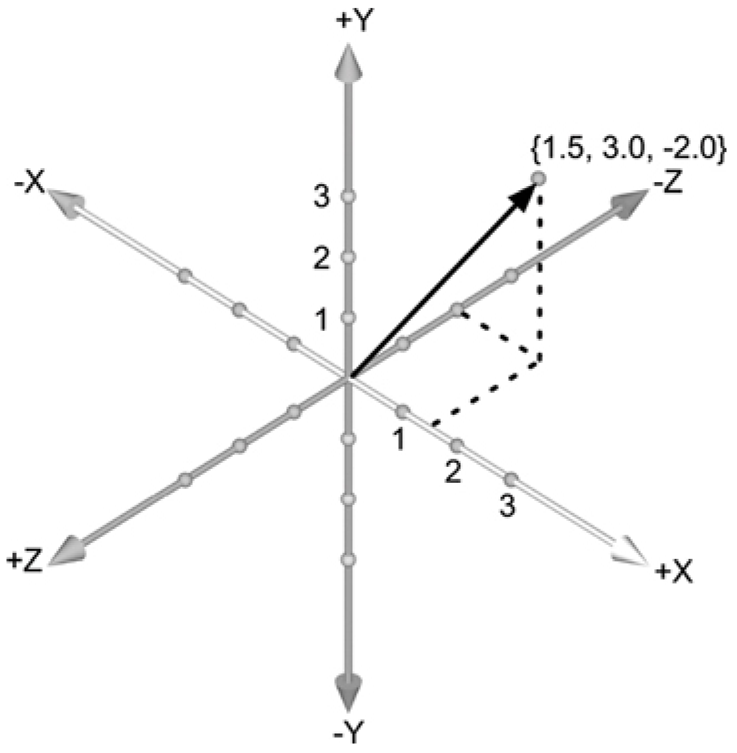


Figure 3: Example Wrist Center Location

With additional information about the position of each joint (2, 3, and the WC) a triangle is formed. The law of cosines is utilized to determine the remaining angles. The triangle consists of Joint 2, Joint 3, and the Wrist Center. The first leg is a length of 1.25 units, the second leg a length of 1.5 units, and the hypotenuse is calculated as the distance between the wrist center and joint 2. Using the law of cosines, the angles of the triangle are determined.

Joint 2 and Joint 3 can provide multiple solutions. For this project it was assumed that the manipulators would face in the positive direction. Values were assumed positive when offered the choice between positive and negative.

Note: Joints may have physical limitations that prevent a solution. Joint limitations were not observed.

Joint 4, 5, and 6 are treated as if they exist at one position. Kinematically, they create a “spherical” joint. The orientation of the wrist is provided in the form of a rotation matrix. Each angle in the spherical joint can be calculated by manipulating the rotation matrix values into a form of $\frac{\sin(\theta)}{\cos(\theta)}$, which is equivalent to $\tan(\theta)$.

During this process, Joint 5 can be determined with either a $\sin(\theta)$ term or more rigorously through a $\tan(\theta)$. In this project $\tan(\theta)$ is used because it can be difficult to keep track of quadrants with $\sin(\theta)$. The function atan2(y,x) addresses quadrant position.

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0, \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0, \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

Figure 4: atan2 function courtesy of Wikipedia

When calculating a form of the equation that utilizes atan2, a square root is taken which provides two solutions. The choice of solution impacts whether or not $\operatorname{atan2}\left(\frac{g}{\sqrt{h^2+i^2}}\right)$ adds or subtracts pi. g represents a $-\sin(z)$ while the denominator represents $\pm\sqrt{\sin(x)^2 * \cos(y)^2 + \cos(x)^2 * \cos(y)^2}$ which simplifies to $\pm\sqrt{\cos(y)^2}$, or $\pm\cos(y)$.

Selecting one solution over another does impact the results, as a shift by π occurs if x is negative. For small angles $\cos(y)$ will be positive; if $y > \frac{\pi}{2}$, $\cos(y)$ will be negative. In general, solutions (solved angles) are smaller when the denominator in atan2 is kept positive.

The solution for the other angles is independent of this joint.

See supplemental PDF for hand calculations.

In multiple sections square roots are taken to find a solution. Positive and negative solutions are possible. In all cases positive solutions were assumed.

Solution

IK_Server.py is a ROS service providing the target angles to a KUKA KR210 6 degree of freedom robot manipulator. When requested, the service “calculate_ik” will provide a list of six angular positions.

First, IK_Server.py loads the necessary tools. This includes ros, tf, the kuka_arm service, and messages from other ros topics. SymPy is also included to process trigonometry math.

The function handle_calculate_IK creates:

- Joint Variable Symbols
- DH Parameter Table
- Generic Transformation Matrix (function)
- Transformation Matrices between each joint
- End Effector rotation matrix
- Inverse Kinematics Math for each desired point

Within the loop, the end effector position and orientation is provided. With the known end effector position, the physical size of the end effector, and its orientation the wrist center is determined.

At this point the angular position of Joint 1 is determined.

Kinematically, the position of Joint 2 is known due to its rigid connection to Joint 1.

A triangle is formed between Joint 2, Joint 3, and the Wrist center. The law of cosines is utilized to determine the remaining angles of the joints.

The rotation of joints 0 to 3 are calculated now that the variables theta 1, 2, and 3 are known.

With the rotations of 0 to 3 known, and the rotations at the end effector known, the remaining rotations of 3 to 6 can be determined with some matrix math. Specifically LU decomposition.

With a known rotation matrix (which consists of sin(theta) and cos(theta)), theta is calculated with an arctangent.

The joint positions are then appended to a list.

IK_Server.py

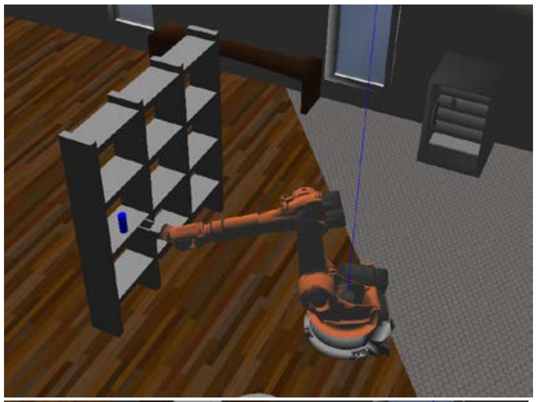
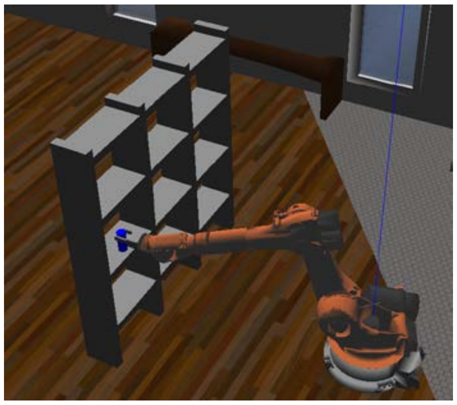
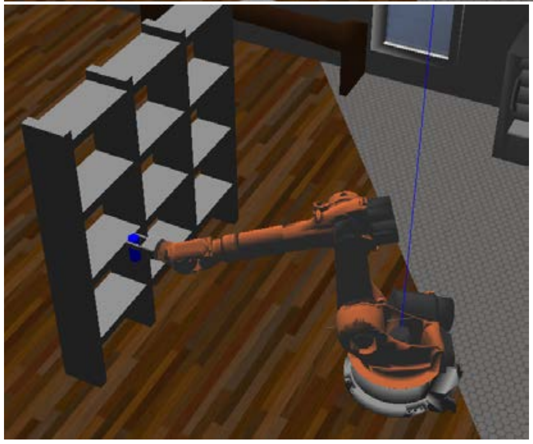
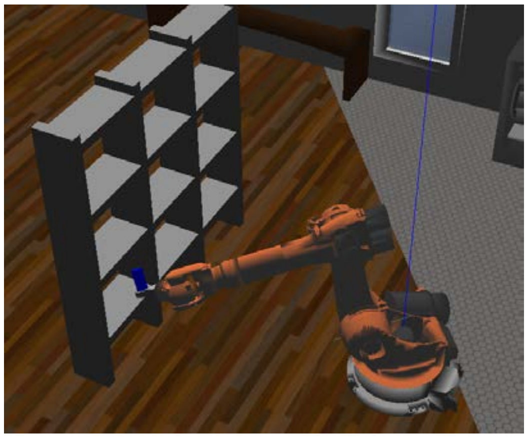
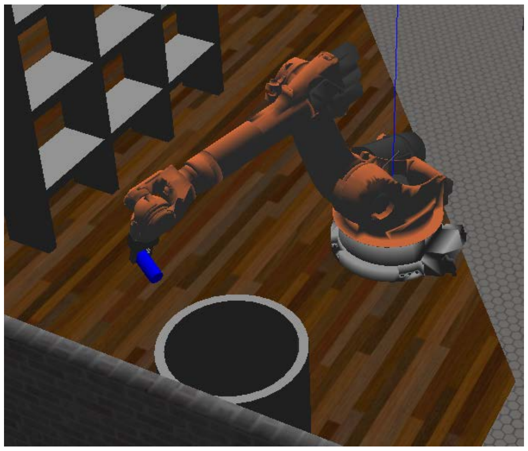
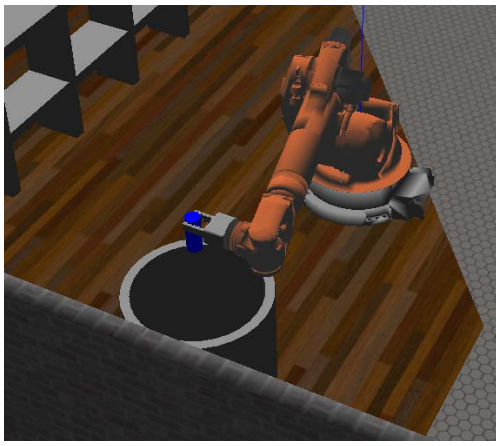
```
#!/usr/bin/env python

# Copyright (C) 2017 Udacity 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
        ### Create symbols for joint variables
        q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8') # theta_i
        d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8') # Link offset
        a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7') # Link length
        alpha0, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6 = symbols('alpha0:7') # Twist angles

          #
          #
          # Create Modified DH parameters
        # DH Parameters
        s = {       alpha0:   0.0,                a0:       0.0,      d1: 0.75, q1:       q1,
                              alpha1:   -pi/2.0,  a1:       0.35,     d2:       0.0,      q2:       q2 - pi/2.0,
                              alpha2:   0.0,                a2:       1.25,     d3:       0.0,      q3:       q3,
                              alpha3:   -pi/2.0,  a3:       -0.054,   d4:       1.50,     q4:       q4,
                              alpha4:    pi/2.0,  a4:       0.0,      d5:       0.0,      q5:       q5,
                              alpha5:   -pi/2.0,  a5:       0.0,      d6:       0.0,      q6:       q6,
                              alpha6:   0.0,                a6:       0.0,      d7:       0.303,    q7:       0.0}
          #
          #
          # 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(s)
                    # Link 1 to Link 2
        T1_2 = TF_Matrix(alpha1, a1, d2, q2).subs(s)
                    # Link 2 to Link 3
        T2_3 = TF_Matrix(alpha2, a2, d3, q3).subs(s)
                    # Link 3 to Link 4
        T3_4 = TF_Matrix(alpha3, a3, d4, q4).subs(s)
                    # Link 4 to Link 5
        T4_5 = TF_Matrix(alpha4, a4, d5, q5).subs(s)
                    # Link 5 to Link 6
        T5_6 = TF_Matrix(alpha5, a5, d6, q6).subs(s)
                    # Link 6 to Gripper
        T6_G = TF_Matrix(alpha6, a6, d7, q7).subs(s)
          #
          #
          # Extract rotation matrices from the transformation matrices
        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

        Rot_Error = ROT_z.subs(y, radians(180)) * ROT_y.subs(p, radians(-90))

        ROT_EE = ROT_EE * Rot_Error

        ###

        # 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])
            ROT_EE = ROT_EE.subs({'r': roll, 'p': pitch, 'y': yaw})
            ### Your IK code here
            EE = Matrix([     [px],
                                                                      [py],
                                                                      [pz]      ])
              # Compensate for rotation discrepancy between DH parameters and Gazebo
              #
              #
              # Calculate joint angles using Geometric IK method
              #
              #
            # Determine wrist center position
            # Equal to end effector minus length of end effector and given length
            WC = EE - (0.303) * ROT_EE[:,2]

            # theta1 = y/x of wrist location
            theta1 = atan2(WC[1],WC[0])

            # Calculate lengths of
            side_a = 1.501
            # side b is line between joint 2 and WC
            side_b = sqrt(pow((sqrt(WC[0] * WC[0] + WC[1] * WC[1]) - 0.35),2) + pow((WC[2] - 0.75), 2))
            side_c = 1.25
            # Law of cosine for theta2 and theta3
            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))
            angle_c = acos((side_a * side_a + side_b * side_b - side_c * side_c) / (2 * side_a * 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 + 0.036)

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

            # 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 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()
```

EOF

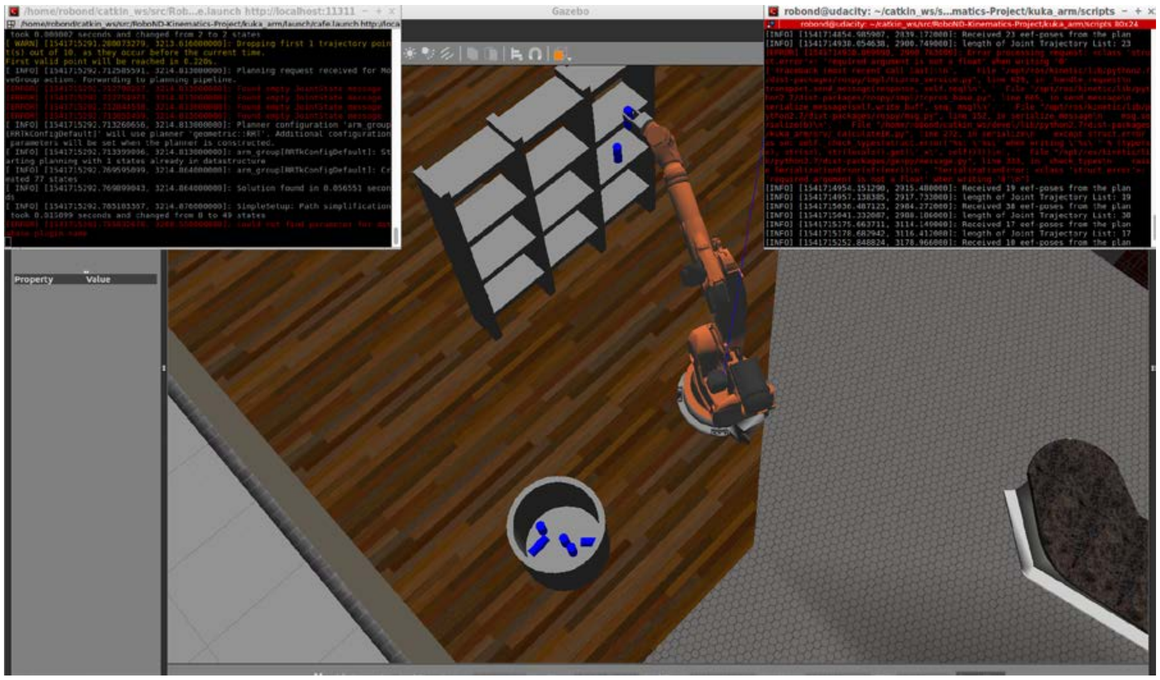
Results

The arm is capable of reaching a cup, picking it up, and placing it in the trash. However, it does seem to go through superfluous maneuvers.

DISCUSSION

The arm moves consistently, but the path is far more complex than necessary. Things could be sped up for the inverse calculation, but bigger improvements could be made with the manual delays and the overly complex paths.



```
/home/robond/catkin_ws/src/Rob...e.launch http://localhost:11311
/home/robond/catkin_ws/src/RoboND-Kinematics-Project/kuka_arm/launch/cafe.launch http://loca
 took 0.009002 seconds and changed from 2 to 2 states
[ WARN] [1541715291.280073279, 3213.616000000]: Dropping first 1 trajectory poin
t(s) out of 10, as they occur before the current time.
First valid point will be reached in 0.220s.
[ INFO] [1541715292.712585591, 3214.813000000]: Planning request received for Mo
veGroup action. Forwarding to planning pipeline.
[ERROR] [1541715292.712790297, 3214.813000000]: Found empty JointState message
[ERROR] [1541715292.712750973, 3214.813000000]: Found empty JointState message
[ERROR] [1541715292.712844538, 3214.813000000]: Found empty JointState message
[ERROR] [1541715292.713052439, 3214.813000000]: Found empty JointState message
[ INFO] [1541715292.713260656, 3214.813000000]: Planner configuration 'arm_group
[RRTkConfigDefault]' will use planner 'geometric::RRT'. Additional configuration
 parameters will be set when the planner is constructed.
[ INFO] [1541715292.713399006, 3214.813000000]: arm_group[RRTkConfigDefault]: St
arting planning with 1 states already in datastructure
[ INFO] [1541715292.769595099, 3214.864000000]: arm_group[RRTkConfigDefault]: Cr
eated 77 states
[ INFO] [1541715292.769899043, 3214.864000000]: Solution found in 0.056551 secon
ds
[ INFO] [1541715292.785103367, 3214.876000000]: SimpleSetup: Path simplification
 took 0.015099 seconds and changed from 8 to 49 states
[ERROR] [illegible]: could not find parameter for [illegible]
[illegible] plugin name
```

```
robond@udacity: ~/catkin_ws/s...matics-Project/kuka_arm/scripts
robond@udacity: ~/catkin_ws/src/RoboND-Kinematics-Project/kuka_arm/scripts 80x24
[INFO] [1541714854.985907, 2839.172000]: Received 23 eef-poses from the plan
[INFO] [1541714930.054638, 2900.749000]: length of Joint Trajectory List: 23
[ERROR] [1541714930.099990, 2900.763000]: Error processing request: <class 'stru
ct.error'>: 'required argument is not a float' when writing '0'
['Traceback (most recent call last):\n', '  File "/opt/ros/kinetic/lib/python2.7
/dist-packages/rospy/impl/tcpros_service.py", line 629, in _handle_request\n
transport.send_message(response, self.seq)\n', '  File "/opt/ros/kinetic/lib/pyt
hon2.7/dist-packages/rospy/impl/tcpros_base.py", line 665, in send_message\n
serialize_message(self.write_buff, seq, msg)\n', '  File "/opt/ros/kinetic/lib/p
ython2.7/dist-packages/rospy/msg.py", line 152, in serialize_message\n    msg.se
rialize(b)\n', '  File "/home/robond/catkin_ws/devel/lib/python2.7/dist-packages
/kuka_arm/srv/_CalculateIK.py", line 272, in serialize\n    except struct.error
as se: self._check_types(struct.error("%s: '%s' when writing '%s'" % (type(s
e), str(se), str(locals().get('_x', self)))))\n', '  File "/opt/ros/kinetic/li
b/python2.7/dist-packages/genpy/message.py", line 333, in _check_types\n    rais
e SerializationError(str(exc))\n', "SerializationError: <class 'struct.error'>:
'required argument is not a float' when writing '0'\n"]
[INFO] [1541714954.151290, 2915.480000]: Received 19 eef-poses from the plan
[INFO] [1541714957.138385, 2917.732000]: length of Joint Trajectory List: 19
[INFO] [1541715036.487123, 2984.272000]: Received 38 eef-poses from the plan
[INFO] [1541715041.332087, 2988.106000]: length of Joint Trajectory List: 38
[INFO] [1541715175.663711, 3114.149000]: Received 17 eef-poses from the plan
[INFO] [1541715178.682942, 3116.412000]: length of Joint Trajectory List: 17
[INFO] [1541715252.848824, 3178.966000]: Received 10 eef-poses from the plan
```

See picture below for proof of 8 items. Two were not grabbed, and the 11th is ready to go.