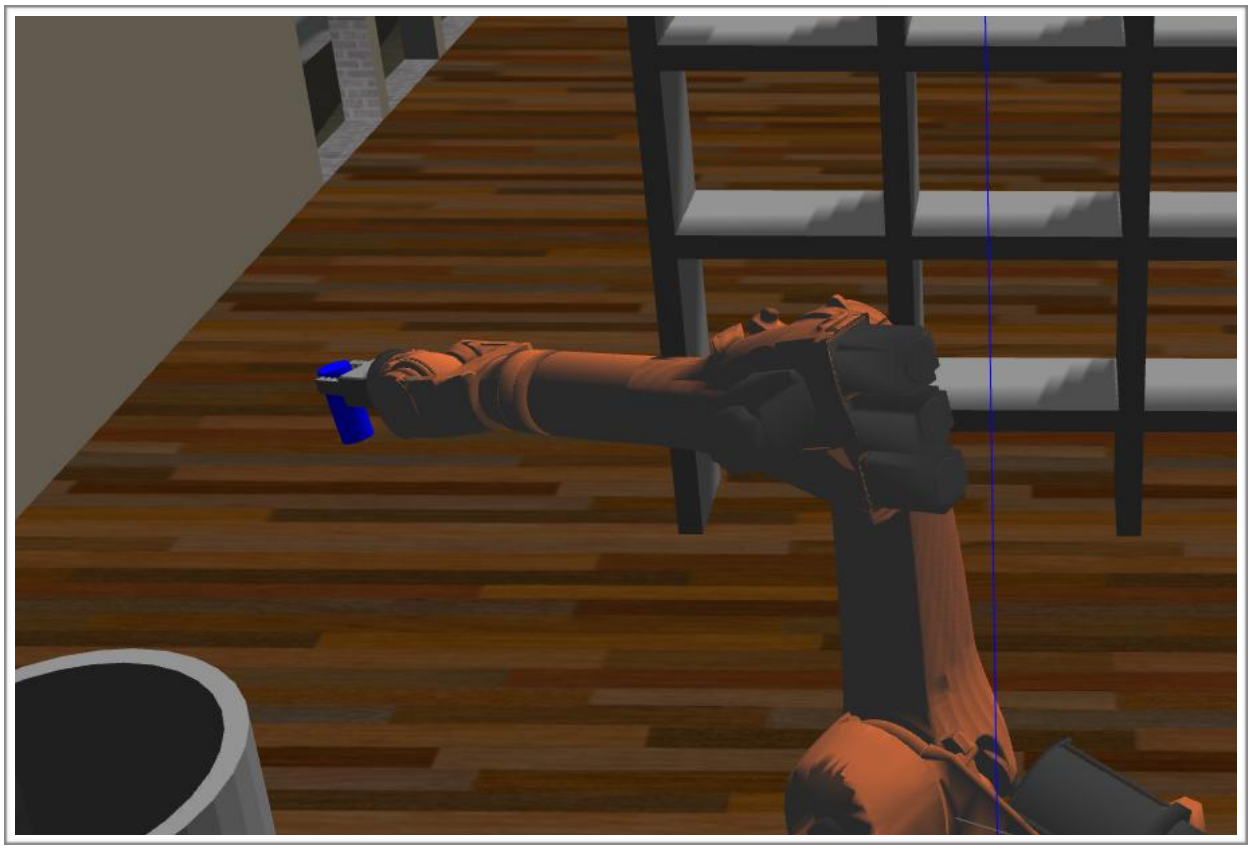
Robotics Arm: Pick & Place

Forward and Reverse Kinematics



Douglas Teeple

Udacity Robotics Nanodegree Program

July 2017

Robotics Arm: Pick & Place

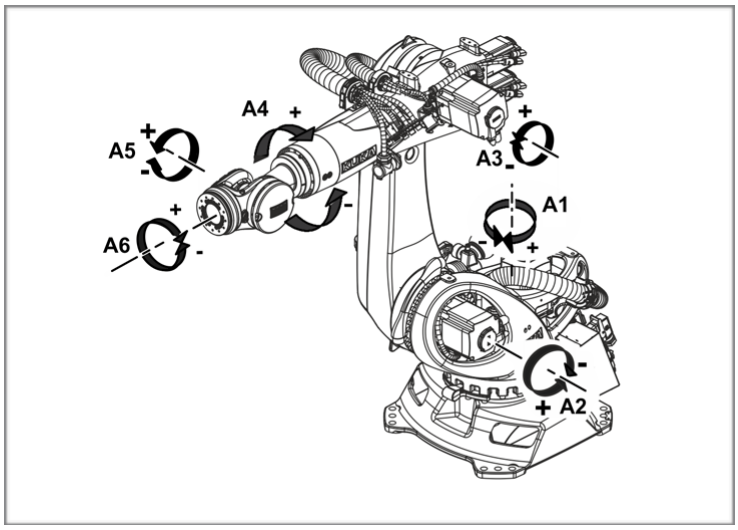
Forward and Inverse Kinematics

Introduction

The object of the Pick and Place project is to write inverse kinematics formulae that implement the Amazon Pick and Place challenge, to correctly pick an item of a shelf at a random location and place it in a bin. To succeed the implementation must do this 8 out of 10 times.

Kuka KR 210

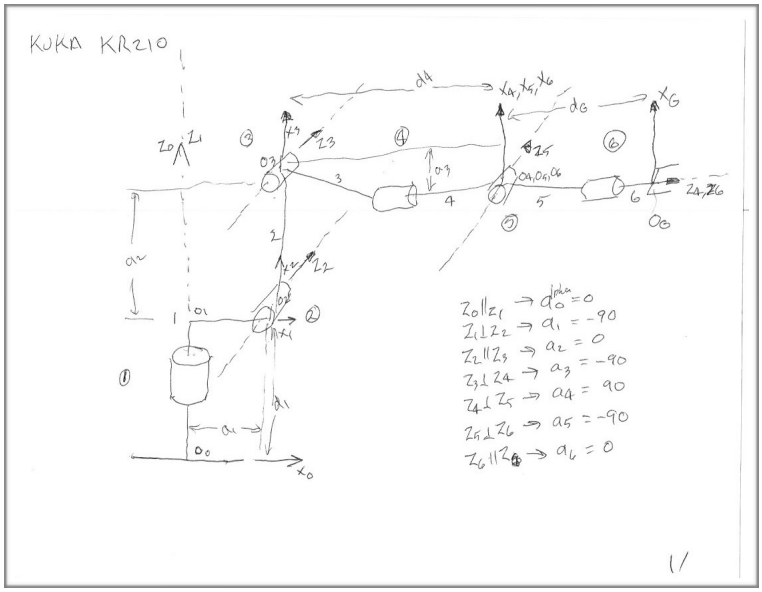
The Kuka KR 210 is a six degrees of freedom commercial robot. The documentation shows the joint angles as below:



Joint Angles

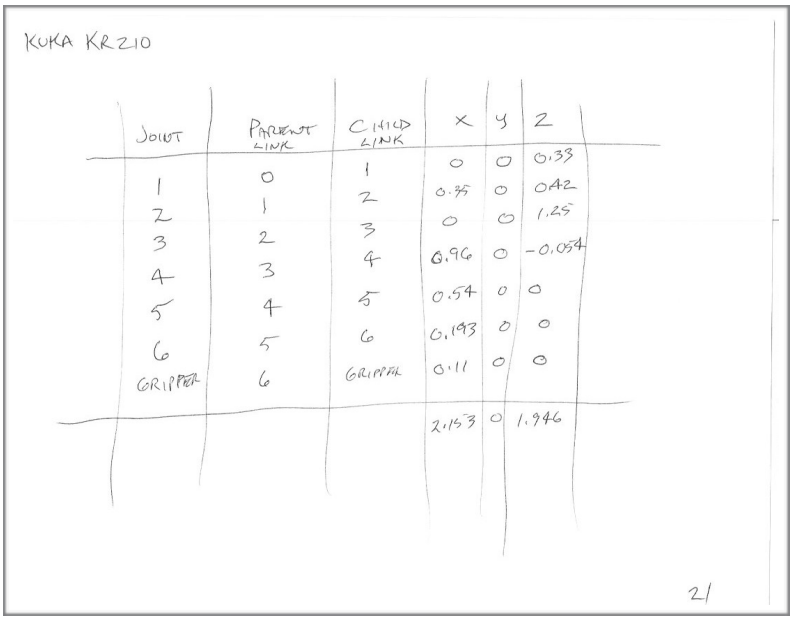
Analysis

I began by drawing a Kuka KR 210 schematic as shown below:



I labelled all the links and joints.

I then took the KR 210 measurements from the documentation[1]:

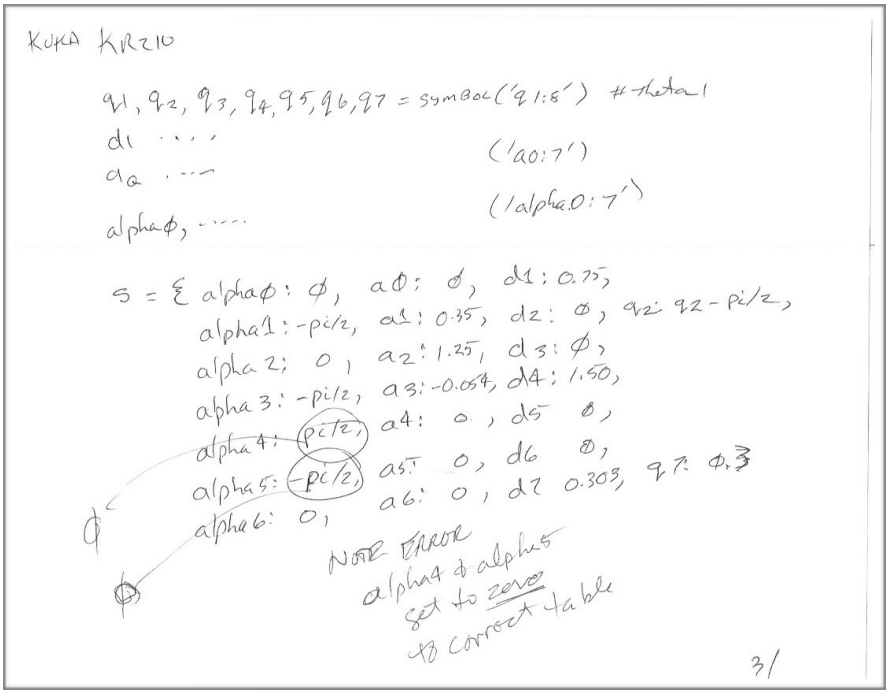


KUKA KR210

JOINT	PARENT LINK	CHILD LINK	x	y	z
1	0	1	0	0	0.33
2	1	2	0.35	0	0.42
3	2	3	0	0	1.25
4	3	4	0.96	0	-0.054
5	4	5	0.54	0	0
6	5	6	0.193	0	0
GRIPPER	6	GRIPPER	0.11	0	0
			2.153	0	1.946

2/

And created the DH table parameters:



KUKA KR210

q1, q2, q3, q4, q5, q6, q7 = symbol('q1:8') # theta1

d1 ....

a0 ....

alpha0, ...

('a0:7')

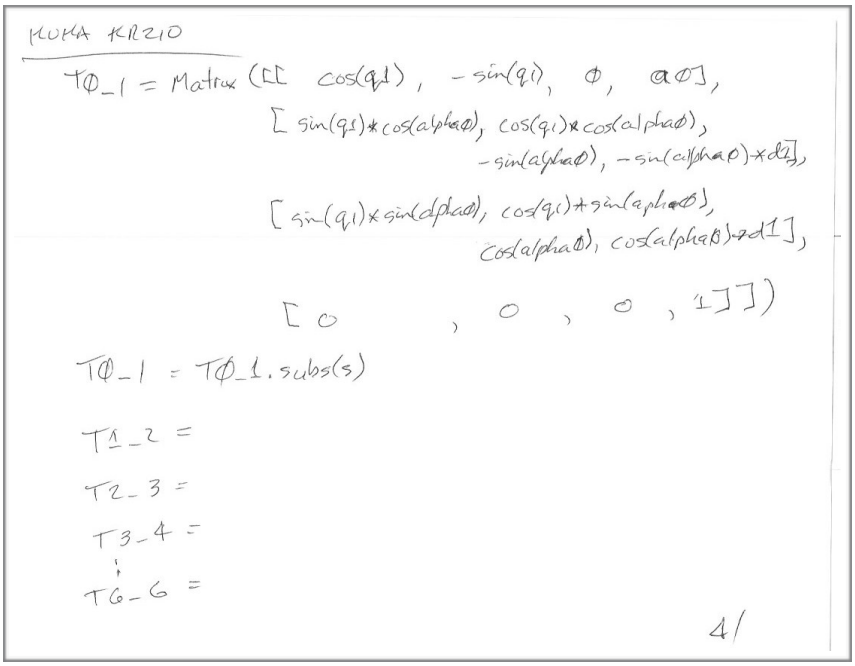
('alpha0:7')

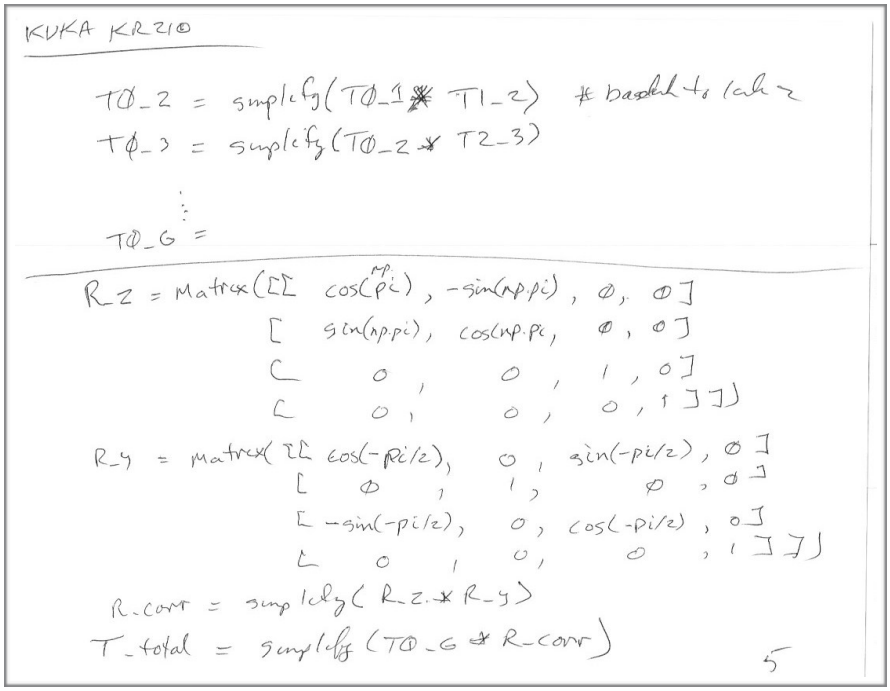
s = { alpha0: 0, a0: 0, d1: 0.75,
alpha1: -pi/2, a1: 0.35, d2: 0, q2: q2 - pi/2,
alpha2: 0, a2: 1.25, d3: 0,
alpha3: -pi/2, a3: -0.054, d4: 1.50,
alpha4: (pi/2), a4: 0, d5 0,
alpha5: (-pi/2), a5: 0, d6 0,
alpha6: 0, a6: 0, d7 0.303, q7: 0.3

NOTE ERROR
alpha4 & alpha5
set to zero
to correct table

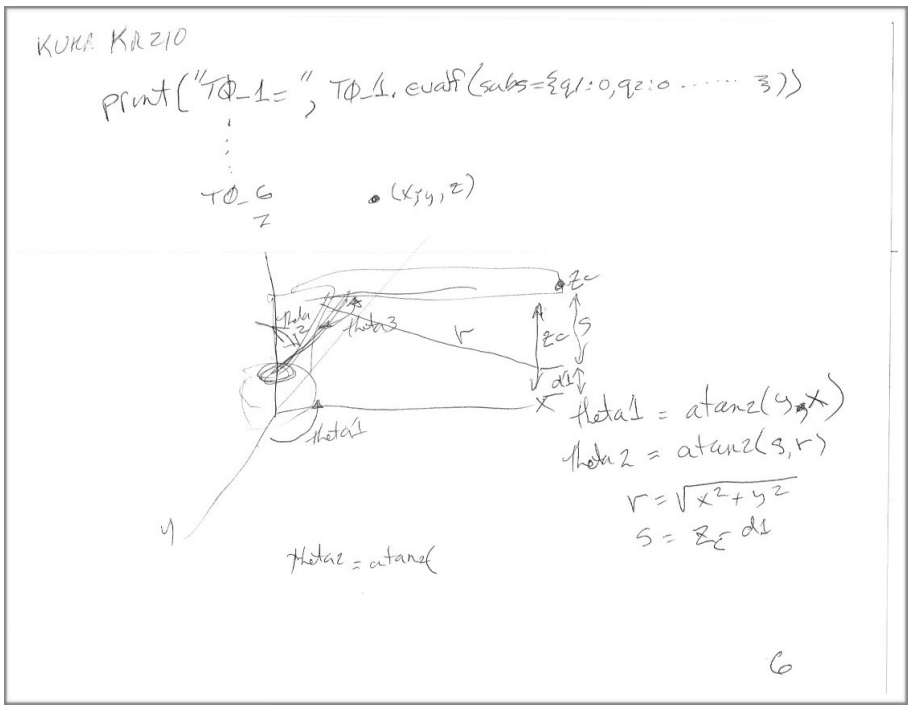
3/

Then I created to forward kinematics matrices. The forward kinematics is later used to check the path trajectory calculate by reverse kinematics.





I then began the inverse kinematics equations based on the class materials.



The project is similar to that taught in class except that it is not RRP, so the equations are different. Instead the first three joints of the Kuka KR 210 are RRR. The equations I developed are:

θ_1 = atan2(y, x)

θ_2 = atan2(s, r) where:

$r^2 = x^2 * y^2$

s = z - (d1+a2)

θ_3 = atan2(s_2, r) where:

s_2 = x - (a1+d4+d7)

In general, these equations are based on the observation that θ_2 contributes mainly to the x axis and θ_3 mainly to the z axis. Reverse kinematic equations have multiple

solutions. Here θ_2 and θ_3 are co-dependent. Iteration can refine these co-dependent angles though I did not need to iterate.

Software Design

I split the IK_Server.py module into two modules for readability. The IK_Server.py module simply gathers position and orientation information and passes it to the IK_calcs.py module.

```
            px = req.poses[x].position.x
            py = req.poses[x].position.y
            pz = req.poses[x].position.z

            # roll, pitch, yaw = end-effector orientation
            lx = req.poses[x].orientation.x
            ly = req.poses[x].orientation.y
            lz = req.poses[x].orientation.z
            lw = req.poses[x].orientation.w

            (roll, pitch, yaw) = tf.transformations.euler_from_quaternion([lx, ly,lz, lw])

            # Calculate joint angles using Geometric IK method

            theta1, theta2, theta3, theta4, theta5, theta6 = IK_calcs(px, py, pz, roll, pitch,
yaw)
```

The main entry point to IK_calcs.py first creates the forward kinematic compound matrix and then if called from the command line accepts test data arguments and calls the `IK_calc()` function.

IK_calcs.py is split into two sections: forward kinematics and inverse kinematics. The main function `IK_calcs()` accepts 3 dimensional point data, position data and pitch, roll, yaw parameters. It first calls the `evalf()` function on the forward kinematic matrix at the rest position as a sanity check.

```
def IK_calcs(px, py, pz, lx=0., ly=0., lz=0., roll=0., pitch=0., yaw=0.):

    # set up initial values at rest

    theta1 = 0.
    theta2 = 0.
    theta3 = 0.
    theta4 = 0.
    theta5 = 0.
    theta6 = 0.
    theta7 = 0.

    T_rest = T_total.evalf(subs={q1:theta1, q2:theta2, q3:theta3, q4:theta4, q5:theta5,
q6:theta6, q7:theta7})
    T_np_rest = np.array(T_rest).astype(np.float64)

    rx = T_np_rest[0,3]
    ry = T_np_rest[1,3]
    rz = T_np_rest[2,3]

    if verbosity >= 3:
        print "T_rest=", T_rest

    if verbosity >= 1:
        print "Rest position:x,y,z= %.3f %.3f %.3f" % (rx, ry, rz)
```

Next the inverse kinematic function `ReverseGeometryCalcs()` is called the evaluate the inverse kinematic formulae to create the θ_i values.

```
    ###########################################################################
    # Part 2: Reverse Kinematics
    ###########################################################################

    # Calculate first three joint angles using Geometric IK method

    theta1, theta2, theta3, theta4, theta5, theta6 = ReverseGeometryCalcs(px, py, pz)

    # Calculate how far we are from the requested position

    T_new = T_total.evalf(subs={q1:theta1, q2:theta2, q3:theta3, q4:theta4, q5:theta5,
q6:theta6, q7:theta7})
```

Then statistics are calculated and the angles returned. The T_total forward kinematics matrix is evaluated using `evalf()` with the floating angles calculated by inverse kinematics to get the evaluated x,y,x coordinates.

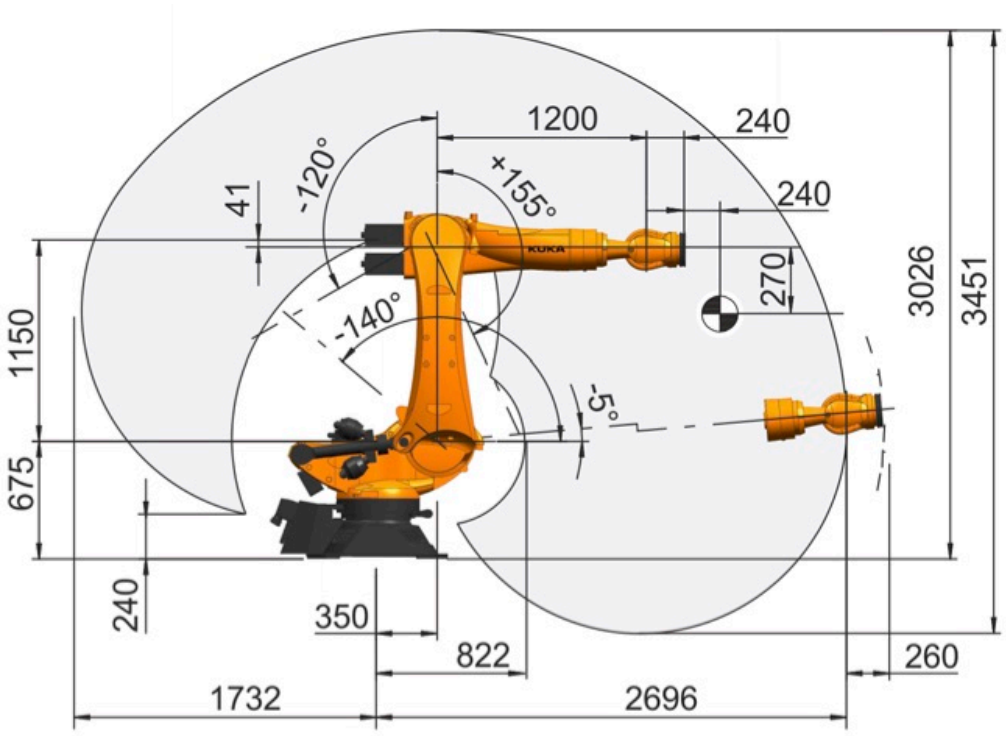
```
    T_np_new = np.array(T_new).astype(np.float64)

    nx = T_np_new[0,3]
    ny = T_np_new[1,3]
    nz = T_np_new[2,3]

    dx = px-nx
    dy = py-ny
    dz = pz-nz

    dist = sqrt(dx**2+dy**2+dz**2)

    if verbosity >= 1:
        print "End Effector position:x,y,z   = %.3f %.3f %.3f" % (lx, ly, lz)
        print "Request position:x,y,z        = %.3f %.3f %.3f" % (px, py, pz)
        print "Calculated position:x,y,z     = %.3f %.3f %.3f" % (nx, ny, nz)
        print "Delta dx, dy, dz              = %.3f %.3f %.3f" % (dx, dy, dz), "
dist=%.3f"%(dist)
        print "Roll, pitch, yaw:             = %.3f %.3f %.3f" % (roll, pitch, yaw)
        print "Joint angles (degrees)        = %.3f %.3f %.3f %.3f %.3f %.3f %.3f" %
(theta1*rtd, theta2*rtd, theta3*rtd, theta4*rtd, theta5*rtd, theta6*rtd, theta7*rtd)
```

The function `astype(np.float64)` converts from a pysym float to a numpy floating point number.

The inverse geometry calculations are done in a separate function:

```
#############################################################################
# Reverse Geometry Calculations
#############################################################################

# Note that theta2 and theta3 are co-dependent, but to get a good guess, I use
# theta2 as the prinicpal contributor to movement in the x axis and theta3 as
# the principal contributor in the z axis.

def ReverseGeometryCalcs(px, py, pz):

    theta1 = inrange(np.arctan2(py, px), m[jmin0], m[jmax0])
    # The total x and z distances from geometry
    xdist  = s[a1]+s[d4]+s[d7]      # from joint 3 to gripper
    zdist  = s[d1]+s[a2]+0.058      # from base to joint 3 when theta2 is 90 degrees
    # now calculate theta2 and theta3
    # Note that limits of movement for each joint are applied
    theta2 = inrange(-np.arctan2(px-xdist, np.sqrt(px**2+py**2))/3., m[jmin2], m[jmax2])
    theta3 = inrange(-np.arctan2(pz-zdist, np.sqrt(px**2+py**2))-theta2, m[jmin3], m[jmax3])
    # all these are rotations and do not contribute to x,y,z displacements
    theta4 = 0.
    theta5 = 0.
    theta6 = 0.

    return [theta1, theta2, theta3, theta4, theta5, theta6]
```

The equations follow those developed in the analysis section. Note the addition of a call to the function `inrange()` which trims any angle calculations to be within the minima and maxima for the Kuka KR 210, taken from the Kuka documentation[1].



Kuka KR 210 Angle Maxima and Minima

```
# make sure the angle (given in radians) is between min and max (given in degrees)
def inrange(prad, mindegrees, maxdegrees):
    plim = prad
    pdegrees = prad * rtd
    if pdegrees < mindegrees:
        plim = mindegrees * dtr
    if pdegrees > maxdegrees:
        plim = maxdegrees * dtr
    return plim
```

The forward kinematics matrices are calculated in `IK_mathinit()`. The equations follow exactly those developed in the class.

```
def IK_mathinit():

    ###########################################################################
    # Part 1: Forward Kinematics
    ###########################################################################

    print("Initializing Transformation matrix, please wait...")

    # Define Modified DH Transformation matrix

    T0_1 = Matrix([[cos(q1), -sin(q1), 0, a0],
                   [sin(q1)*cos(alpha0), cos(q1)*cos(alpha0), -sin(alpha0), -sin(alpha0)*d1],
                   [sin(q1)*sin(alpha0), cos(q1)*sin(alpha0),  cos(alpha0),  cos(alpha0)*d1],
                   [ 0, 0, 0, 1]])
    T0_1 = T0_1.subs(s)

    T1_2 = Matrix([[cos(q2), -sin(q2), 0, a1],
                   [sin(q2)*cos(alpha1), cos(q2)*cos(alpha1), -sin(alpha1), -sin(alpha1)*d2],
                   [sin(q2)*sin(alpha1), cos(q2)*sin(alpha1),  cos(alpha1),  cos(alpha1)*d2],
                   [ 0, 0, 0, 1]])
    T1_2 = T1_2.subs(s)

    T2_3 = Matrix([[cos(q3), -sin(q3), 0, a2],
                   [sin(q3)*cos(alpha2), cos(q3)*cos(alpha2), -sin(alpha2), -sin(alpha2)*d3],
                   [sin(q3)*sin(alpha2), cos(q3)*sin(alpha2),  cos(alpha2),  cos(alpha2)*d3],
                   [ 0, 0, 0, 1]])
    T2_3 = T2_3.subs(s)

    T3_4 = Matrix([[cos(q4), -sin(q4), 0, a3],
                   [sin(q4)*cos(alpha3), cos(q4)*cos(alpha3), -sin(alpha3), -sin(alpha3)*d4],
                   [sin(q4)*sin(alpha3), cos(q4)*sin(alpha3),  cos(alpha3),  cos(alpha3)*d4],
                   [ 0, 0, 0, 1]])
    T3_4 = T3_4.subs(s)

    T4_5 = Matrix([[cos(q5), -sin(q5), 0, a4],
                   [sin(q5)*cos(alpha4), cos(q5)*cos(alpha4), -sin(alpha4), -sin(alpha4)*d5],
                   [sin(q5)*sin(alpha4), cos(q5)*sin(alpha4),  cos(alpha4),  cos(alpha4)*d5],
                   [ 0, 0, 0, 1]])
    T4_5 = T4_5.subs(s)

    T5_6 = Matrix([[cos(q6), -sin(q6), 0, a5],
                   [sin(q6)*cos(alpha5), cos(q6)*cos(alpha5), -sin(alpha5), -sin(alpha5)*d6],
                   [sin(q6)*sin(alpha5), cos(q6)*sin(alpha5),  cos(alpha5),  cos(alpha5)*d6],
                   [ 0, 0, 0, 1]])
    T5_6 = T5_6.subs(s)

    T6_G = Matrix([[cos(q7), -sin(q7), 0, a6],
                   [sin(q7)*cos(alpha6), cos(q7)*cos(alpha6), -sin(alpha6), -sin(alpha6)*d7],
                   [sin(q7)*sin(alpha6), cos(q7)*sin(alpha6),  cos(alpha6),  cos(alpha6)*d7],
                   [ 0, 0, 0, 1]])
    T6_G = T6_G.subs(s)

    # Create individual transformation matrices

    T0_2 = simplify(T0_1*T1_2)
    T0_3 = simplify(T0_2*T2_3)
    T0_4 = simplify(T0_3*T3_4)
    T0_5 = simplify(T0_4*T4_5)
    T0_6 = simplify(T0_5*T5_6)
    T0_G = simplify(T0_6*T6_G)
```

Next the total transformation is calculated and the Matrix returned:

```
    ###########################################################################
    # Correction, in y, z for gripper
    ###########################################################################

    R_y = Matrix([[ cos(-pi/2.),  sin(-pi/2.), 0, 0],
                  [ 0,            1,           0, 0],
                  [ -sin(-pi/2.), cos(-pi/2.), 0, 0],
                  [ 0,            0,           0, 1]])

    R_z = Matrix([[ cos(pi), -sin(pi), 0, 0],
                  [ sin(pi),  cos(pi), 0, 0],
                  [ 0,              0, 1, 0],
                  [ 0,              0, 0, 1]])

    R_corr = simplify(R_z*R_y)
    T_total = simplify(T0_G * R_corr)
    if verbosity >= 3:
        print("R_corr=", R_corr.evalf(subs={q1:0., q2:0., q3:0., q4:0., q5:0., q6:0., q7:0.}))
        print("T_total=", T_total)

    print("Transformation matrix ready...")

    return T_total
```

Note that prior to substituting values into the combined matrices, the DH parameters are created:

```
#############################################################################
#
# Do the IK Calculations here. Module separated so it can be run
# independently of the whole simulation.
#
# px, py, pz - the requested location
# lx, ly, lz - the current location
# roll, pitch, yaw - the current orientation
#
#############################################################################

# Define DH param symbols

q1, q2, q3, q4, q5, q6, q7 = symbols('q1:8')    # theta_i
d1, d2, d3, d4, d5, d6, d7 = symbols('d1:8')    # link offsets
a0, a1, a2, a3, a4, a5, a6 = symbols('a0:7')    # link length

# Joint angle symbols

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

# Joint min max angle symbols

jmin0, jmin1, jmin2, jmin3, jmin4, jmin5, jmin6 = symbols('jmin0:7')
jmax0, jmax1, jmax2, jmax3, jmax4, jmax5, jmax6 = symbols('jmax0:7')

# Modified DH params

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: 1.50,
      alpha4: 0.,     a4: 0.,     d5: 0.,
      alpha5: 0.,     a5: 0.,     d6: 0.,
      alpha6: 0.,     a6: 0.,     d7: 0.303, q7: 0. }

# Joint min max values
m = { jmin0: -185.,  jmax0: 185.,
      jmin1: -140.,  jmax1: 5.,
      jmin2: -120.,  jmax2: 155.,
      jmin3: -350.,  jmax3: 350.,
      jmin4: -122.5, jmax4: 122.5,
      jmin5: -350.,  jmax5: 350. }
```

These definitions follow the class definitions. Note the addition of the joint minimum and maximum values. These are used in the function `inrange()` which trims any angle calculations to be within the minima and maxima for the Kuka KR 210[1].

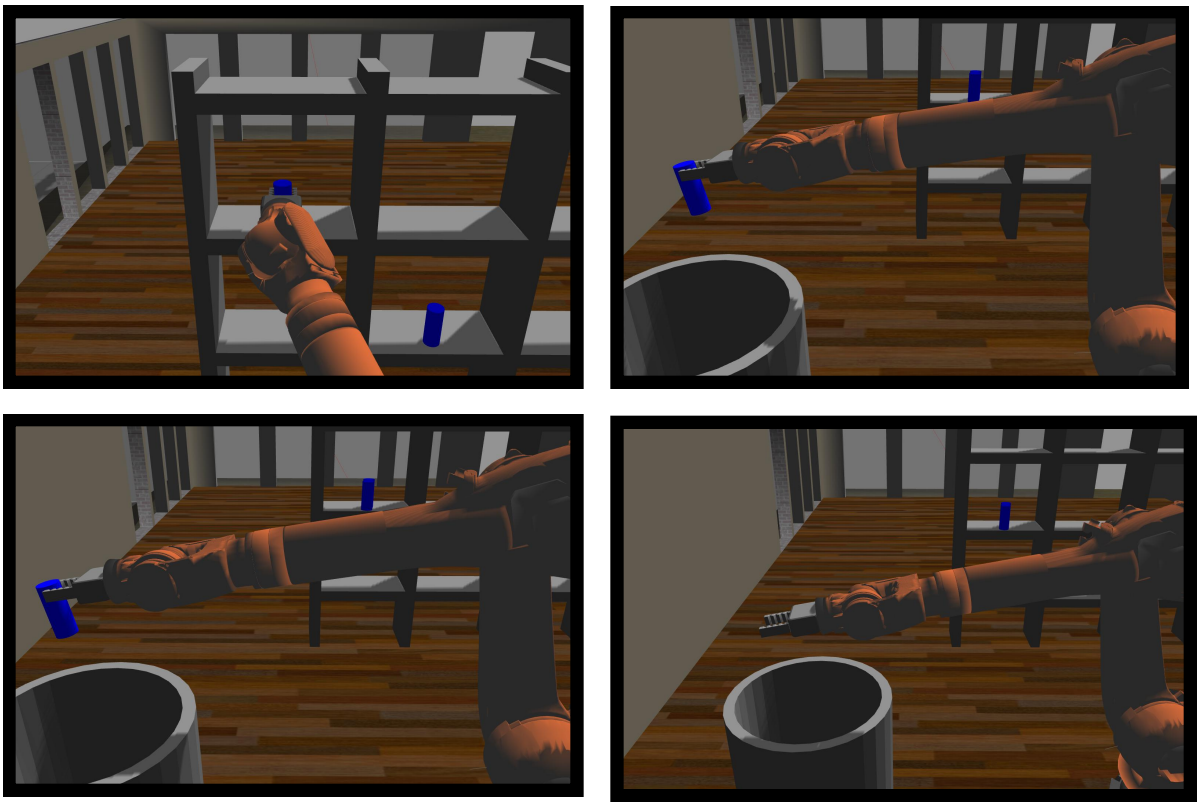
Software Environment

I installed Ubuntu Linux 16.04 and all the required software on an external USB drive bootable from a MacBook pro. I installed the Kinetic version of ROS native on Ubuntu. I also installed the RoboND-Kinematics-Project as per the class instructions.

Results

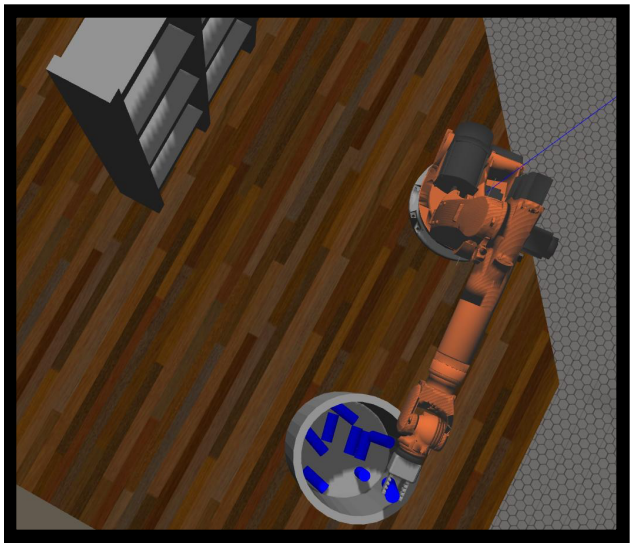
Well the first trials resulted in many cylinders on the floor…

After revising the Inverse Kinematics routines, the results were more positive:



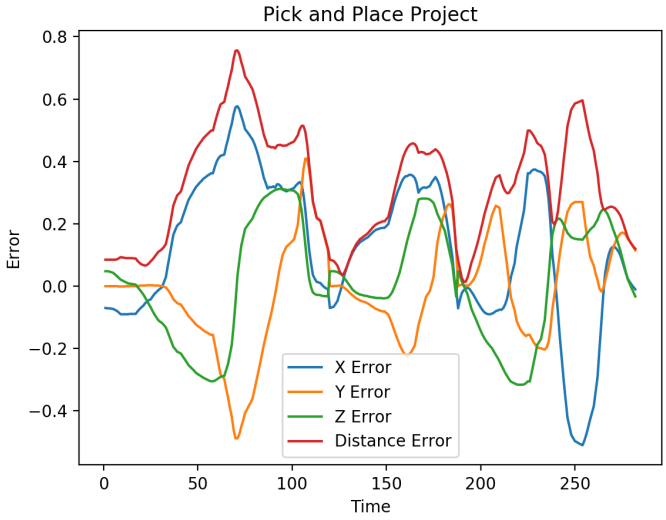
Here is a complete pickup and drop sequence.

And finally the 9th cylinder goes in!



Here you can see the 10 cylinders. One is in the corner and the other 9 in the bin. Mission accomplished!

This plot shows errors in x,y,z and total distance over time:



Conclusion

In conclusion, the inverse geometric calculations were definitely a challenge. I ran in to one problem that I had to solve by changing configuration parameters. I found that although the gripper aligned perfectly, it would not pick up the cylinders. I changed kr210_controllers.yaml as follows:

```
    gripper_controller:
      type: "effort_controllers/JointTrajectoryController"
      joints:
        - right_gripper_finger_joint
        - left_gripper_finger_joint
      gains:
        right_gripper_finger_joint: {p: 100, i: 1, d: 10, i_clamp: 1.0}
        left_gripper_finger_joint: {p: 100, i: 1, d: 10, i_clamp: 1.0}
      constraints:
        goal_time: 3.0
        right_gripper_finger_joint:
          goal: 0.01
        left_gripper_finger_joint:
          goal: 0.01
```

The right and left gripper finger joint goal was changed from 0.02 to 0.01. This fixed the gripping problem, and I was able to complete the challenge.

References

[1] KR QUANTEC prime Mit F-, C und CR-Varianten Spezifikation, KUKA Roboter GmbH Zugspitzstraße 140 D-86165 Augsburg Deutschland 2016

[2] Inverse Kinematics Analysis KUKA KR Agilus Robot, Denis A. Tatarnikov1,a, Gennady P. Tsapko1,bl National Research Tomsk Polytechnic University (TPU), 30 Lenin Ave., Tomsk, 634050, Russia

[3] Graphical Representation of the Significant 6R KUKA Robots Spaces Ana M. Djuric*, Mirjana Filipovic**, Ljubinko Kevac, The University of Belgrade, Bulevar Kralja Aleksandra 73, 11000 Belgrade, Serbia

[4] Solving Kinematics Problems of a 6-DOF Robot Manipulator, Alireza Khatamian, Computer Science Department, The University of Georgia, Athens, GA, U.S.A