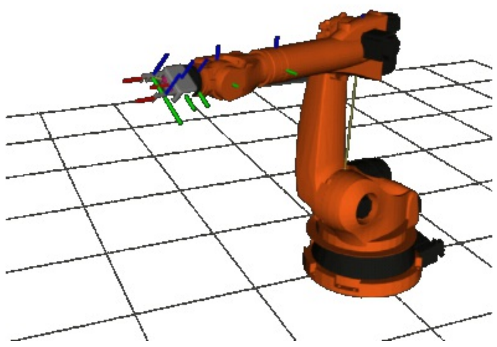
Project 2: Pick & Place Project

1. Project Summary

The goal of the project is to calculate the joint angles given the end-effector's pose for a 6 degree-of-freedom Kuka Arm 210 by applying principles of kinematics.



1.1 Objectives:

- Calculate joint angles - θ1, θ2, θ3, θ4, θ5, θ6
- Grasp the object from the shelf
- Drop the object in the bin

1.2 Outcomes:

- Calculated all joint angles (with optimized IK)
- Grasped and placed 6 objects in the bin
- Click on the image below to view the demo on YouTube. (https://www.youtube.com/watch?v=1BXRThDDH1Q)

2. Forward Kinematics Analysis

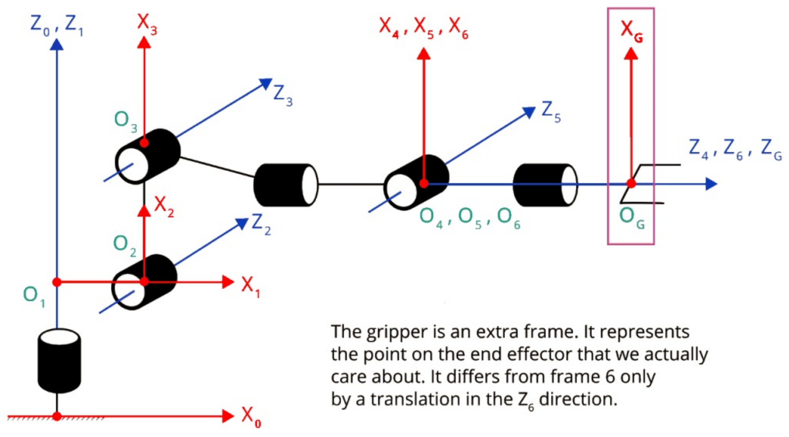
We use forward kinematics to find the end-effector's location in a frame of reference provided we know the joint angles.

2.1 Denavit-Hartenberg Derivation

To find the joint angles for a robotic arm manipulator, we use Denavit-Hartenberg convention to derive a set of necessary parameters.

The set of parameters in a DH table are:

Name	Symbol	Definition
Joint angle	θ(i)	Angle between x(i-1) and x(i) about z(i)
Link offset	d(i)	Distance from x(i-1) to x(i) along z(i)
Link length	a(i-1)	Distance from z(i-1) to z(i) along x(i-1)
Twist angle	α(i-1)	Angle between z(i-1) and z(i) about x(i-1)



The gripper is an extra frame. It represents the point on the end effector that we actually care about. It differs from frame 6 only by a translation in the Z_6 direction.

The set of derived DH parameters are shown below.

Links	θ(i)	d(i)	a(i-1)	α(i-1)
0->1	q1*	d1	0	0
1->2	q2* - 90	0	a1	-90
2->3	q3*	0	a2	0
3->4	q4*	d4	a3	-90
4->5	q5*	0	0	90
5->6	q7*	0	0	-90
6->EE	qG*	dG	0	0

Every joint in the Kuka arm are revolute joints and determine the angular rotation for the *i-th* *joint* - hence marked by qi*. Between `Joint 2` and `Joint 3`, there is an offset of 90 degrees which needs to be accounted for.

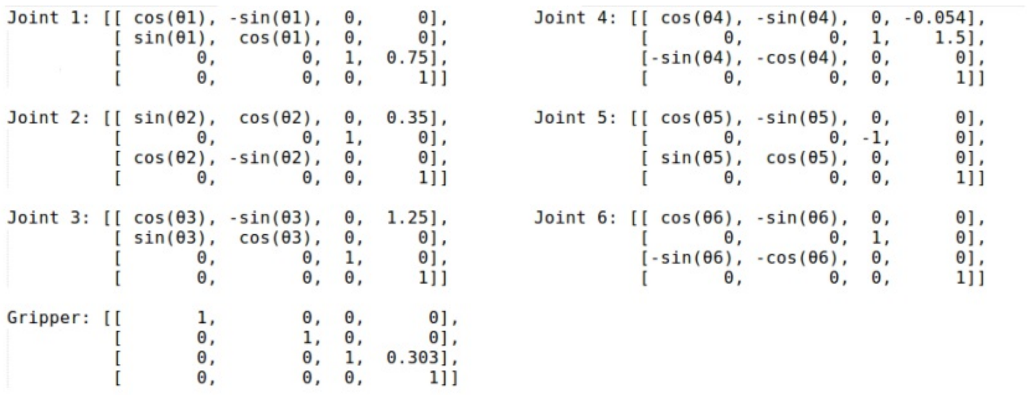
The values for the link offsets and link lengths are:
`d1 = 0.75, d4 = 1.50, dG = 0.303` and `a1 = 0.35, a2 = 1.25, a3 = -0.054`

2.2 Homogeneous Transformation Matrix

The homogeneous transform is a 4x4 matrix that contains information of the orientation and position of a frame in another reference frame.

$$T = \left[\begin{array}{ccc:c} & & & P_x \\ & R_T & & P_y \\ & & & P_z \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right]$$

The transform for each individual joint is shown below.



```
Joint 1: [[ cos(θ1), -sin(θ1),  0,      0],            Joint 4: [[ cos(θ4), -sin(θ4),  0, -0.054],
          [ sin(θ1),  cos(θ1),  0,      0],                      [        0,        0,  1,    1.5],
          [       0,        0,  1,   0.75],                      [-sin(θ4), -cos(θ4),  0,      0],
          [       0,        0,  0,      1]]                      [        0,        0,  0,      1]]

Joint 2: [[ sin(θ2),  cos(θ2),  0,   0.35],            Joint 5: [[ cos(θ5), -sin(θ5),  0,      0],
          [       0,        0,  1,      0],                      [        0,        0, -1,      0],
          [ cos(θ2), -sin(θ2),  0,      0],                      [ sin(θ5),  cos(θ5),  0,      0],
          [       0,        0,  0,      1]]                      [        0,        0,  0,      1]]

Joint 3: [[ cos(θ3), -sin(θ3),  0,   1.25],            Joint 6: [[ cos(θ6), -sin(θ6),  0,      0],
          [ sin(θ3),  cos(θ3),  0,      0],                      [        0,        0,  1,      0],
          [       0,        0,  1,      0],                      [-sin(θ6), -cos(θ6),  0,      0],
          [       0,        0,  0,      1]]                      [        0,        0,  0,      1]]

Gripper: [[       1,        0,  0,       0],
          [       0,        1,  0,       0],
          [       0,        0,  1,   0.303],
          [       0,        0,  0,       1]]
```

The equation for calculating a homogeneous transform between two reference frames and its resultant output is shown below

$$ {}^{i-1}_{\ \ i}T = R_X(\alpha_{i-1})\, D_X(a_{i-1})\, R_Z(\theta_i)\, D_Z(d_i) $$

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

The following code is used for generating the homogeneous transform for a set of DH parameters.

```
# Calculating homogenous transformation with a set of DH parameters
def H_transform(alpha, a, d, theta):
  cT, sT = cos(theta), sin(theta)
  cA, sA = cos(alpha), sin(alpha)

  A = Matrix([
    [ cT  ,   -sT,    0,    a   ],
    [sT*cA, cT*cA,  -sA,  -d*sA ],
    [sT*sA, cT*sA,   cA,   d*cA ],
    [  0  ,     0,    0,     1  ]
    ])
  return A
```

For calculating the homogeneous transform between the `gripper_link` and `base_ink`, an extrinsic body fixed rotations are performed. The sequence of operations involve performing a roll along x-axis, pitch along y-axis, and yaw along z-axis. For extrinsic rotations, we pre-multiply the sequence of operations like this:

`R0_EE = Rot(Z, yaw) * Rot(Y, pitch) * Rot(X, roll)`

To align the URDF model of the robot with the DH parameters, we apply a transformation to the gripper by rotating first along the z-axis by 180, and then rotating about the y-axis by -90 degrees.

`R_corr = Rot(Z, 180) * Rot(Y, -90)`

Hence, `R0G = R0_EE * R_corr`.

The following code is used for performing extrinsic rotations linking the `gripper_link` and `base_link` as well as performing a correctional transform to the rotation matrix.

```

# extrinsic rotation matrix calculation
def r_zyx(roll, pitch, yaw):
  R_x = Matrix([
    [1,         0,           0],
    [0, cos(roll), -sin(roll)],
    [0, sin(roll),  cos(roll)]
    ])

  R_y = Matrix([
    [ cos(pitch), 0, sin(pitch)],
    [          0, 1,          0],
    [-sin(pitch), 0, cos(pitch)]
    ])

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

    return R_z*R_y*R_x
  # Create rotation transform from end-effector as seen from ground frame
  R0_EE   = r_zyx(roll, pitch, yaw)

  # Align EE from URDF to DH convention by multiplying with correction-matrix
  R_corr  = r_zyx(0, -pi/2, pi)
  R0_EE   = R0_EE*R_corr
```

3. Inverse Kinematics Analysis

We use inverse kinematics to find the joint angles of a robotic arm in a frame of reference given the pose of the end-effector.

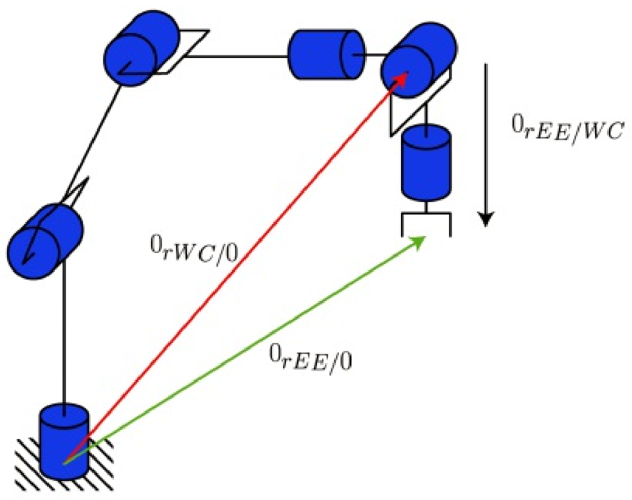
The Kuka Arm has a total of 6 joints where every joint is revolute. `Joints 4, 5, 6` form the spherical wrist of the arm with `Joint 5` being the wrist center. For performing inverse kinematics, we decouple the system into 2 parts.

The first part consists of `Joints 1, 2, 3` which are responsible for determining the position of the wrist center also called *Inverse Position*.

The second part consists of `Joints 4, 5, 6` which are responsible for determining the orientation of the wrist center also called *Inverse Orientation*.

3.1 Inverse Position

We first find the wrist center's position which is marked by the red vector in the diagram below. The green vector represents the end-effector's position from the ground frame relative to the ground frame. The black vector represents the end-effector's position in the wrist-center's frame relative to the ground frame.



$$ {}^0r_{WC/0_0} = {}^0 r_{0G/0_0} - r_{0_5/0_G} $$

$$ \begin{bmatrix} WC_x \\ WC_y \\ WC_z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} - {}^0_6R \begin{bmatrix} 0 \\ 0 \\ d_G \end{bmatrix} $$

By doing a simple vector subtraction, we can find the wrist-center's location in the ground frame relative to the ground frame. We use the following equation to find the wrist center's position. The corresponding vector's mathematical representation is color coded.

```
# Find wrist-center (WC) location
wx = px - dG*R0_EE[0,2]
wy = py - dG*R0_EE[1,2]
wz = pz - dG*R0_EE[2,2]
```

3.1.1 - Finding cartesian distances between joints

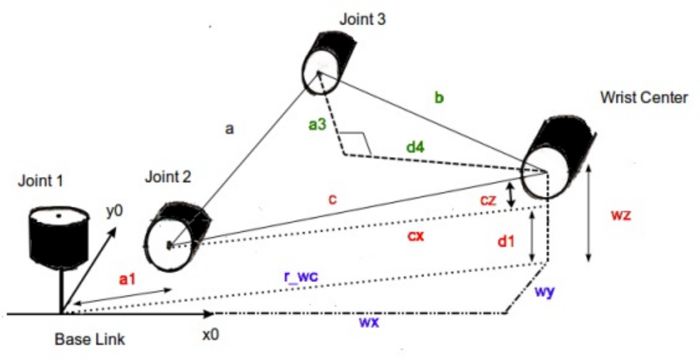
Before finding all the angles, first let us find all the cartesian distances between `Joint 2`, `Joint 3`, and `Wrist Center`. We are interested in `a, b, c`. Parameters used for deriving a particular side has been color-coded.
`a = a2` (just the link length between joints 2 and 3)

`b = √(a3 * a3 + d4 * d4)` *(color coded in green)*

`c = √(cx * cx + cz * cz)` *(color coded in red)*

where `cx = r_wc - a1` and `cz = wz - d1`

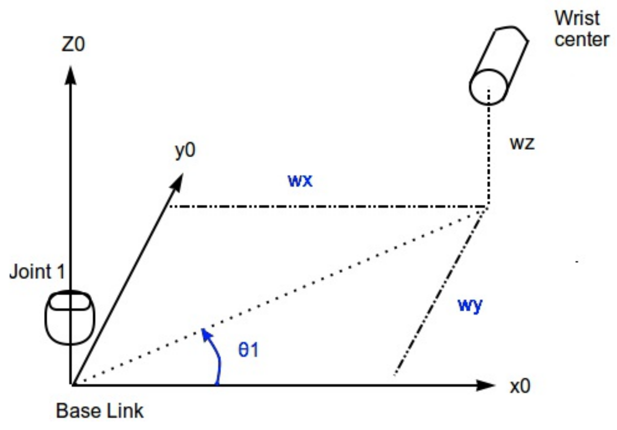
and `r_wc = √(wx * wx + wy * wy)` *(color coded in blue)*



```
# Finding cartesian distances b/w joints 2, 3, and WC
r_wc = hypotenuse(wx, wy)     # vector proj on x-y plane of WC
a    = a2
b    = hypotenuse(a3, d4)
c    = hypotenuse(r_wc-a1, wz-d1)
```

3.1.2 - Finding angles θ1, θ2, θ3

- θ1: For finding θ1, we project the vector going from the `base_link` to the end-effector (or `gripper_link`) onto the `XY-plane` and apply an inverse tangent operation. The following diagram shows how θ1 is derived where `θ1 = atan2(wy, wx)`.

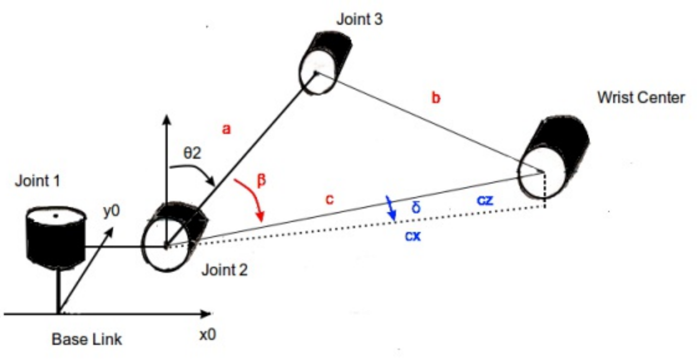


- θ2: For finding θ2, we use law of cosines for finding angle β (color coded in red) and inverse tangent function for finding angle δ (color coded in blue). The following diagram shows how θ2 is derived where `θ2 = 90 - β - δ`.

  `cosβ = (a*a + c*c - b*b) / (2 * a * c)`

  `sinβ = √(1 - cosβ*cosβ)`

  `β = atan2(sinβ, cosβ)` *(color coded in red)* `δ = atan2(cz, cx)` *(color coded in blue)*



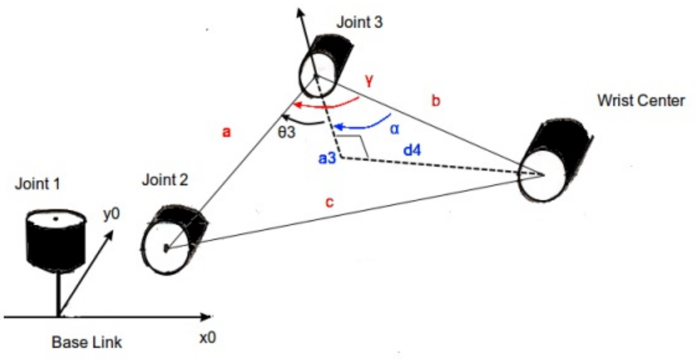
- θ3: For finding θ3, we use law of cosines for finding angle γ (color coded in red) and inverse tangent function for finding angle α (color coded in blue). The following diagram shows how θ3 is derived where `θ3 = - (γ - α)`.

`cosγ = (a*a + c*c - b*b) / (2 * a * c)`

`sinγ = √(1 - cosγ*cosγ)`

`γ = atan2(sinγ, cosγ)` (color coded in red)

`α = atan2(d4, a3)` (color coded in blue)



```
# Finding theta1
theta1 = atan2(wy, wx)    # project WC's vector on x-y plane
theta1 = theta1.evalf()

# Finding theta2 = 90 - Beta - Delta
beta   = cosine_angle(a, c, b)       # angle beta b/w a & c
delta  = atan2(wz - d1, r_wc - a1)
theta2 = pi/2 - beta - delta
theta2 = theta2.evalf()

# Finding theta3 = -(gamma - alpha)
```

```
            gamma  = cosine_angle(a, b, c)       # angle gamma b/w a & b
            alpha  = atan2(d4, a3)
            theta3 = -(gamma - alpha)
            theta3 = theta3.evalf()
```

3.2 Inverse Orientation - Finding angles θ4, θ5, θ6

For the inverse orientation problem, we will decompose the rotation transform from the `gripper_link` to the `base_link` as such:
`R0G = R03 * R36 * R6G`
`R03.inverse * R0G = R03.inverse * R03 * R36 * I` *(since the 6th frame and gripper frame have same orientation)*
`R36 = R03' * R0G` *(since rotation matrix' inverse is its transpose)*

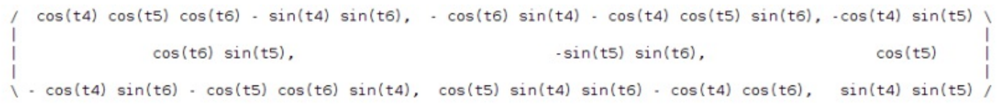
We know `R0G` from the extrinsic body fixed rotations calculated earlier.

`R0_EE = Rot(Z, yaw) * Rot(Y, pitch) * Rot(X, roll)`

`R_corr = Rot(Z, 180) * Rot(Y, -90)`

`R0G = R0_EE * R_corr`

Hence, `R36 = R03' * R0G` where the matrix R36 is shown below.


```
/  cos(t4) cos(t5) cos(t6) - sin(t4) sin(t6),  - cos(t6) sin(t4) - cos(t4) cos(t5) sin(t6), -cos(t4) sin(t5) \
|                                                                                                           |
|               cos(t6) sin(t5),                               -sin(t5) sin(t6),                cos(t5)     |
|                                                                                                           |
\ - cos(t4) sin(t6) - cos(t5) cos(t6) sin(t4),  cos(t5) sin(t4) sin(t6) - cos(t4) cos(t6),  sin(t4) sin(t5) /
```

```
    # Rotational transform from 6th to 3rd frame
    R3_0 = T0_3[:3, :3].transpose()   # take inverse (or transpose)
    R3_6 = R3_0 * R0_EE
```

- θ4: For finding θ4, we look at elements r13 and r33.

  `θ4 = atan2(r33, -r13)`

- θ5: For finding θ5, we look at elements r23.

  `θ5 = atan2(√(1 - r23*r23), r23)`

- θ6: For finding θ6, we look at elements r21 and r22.

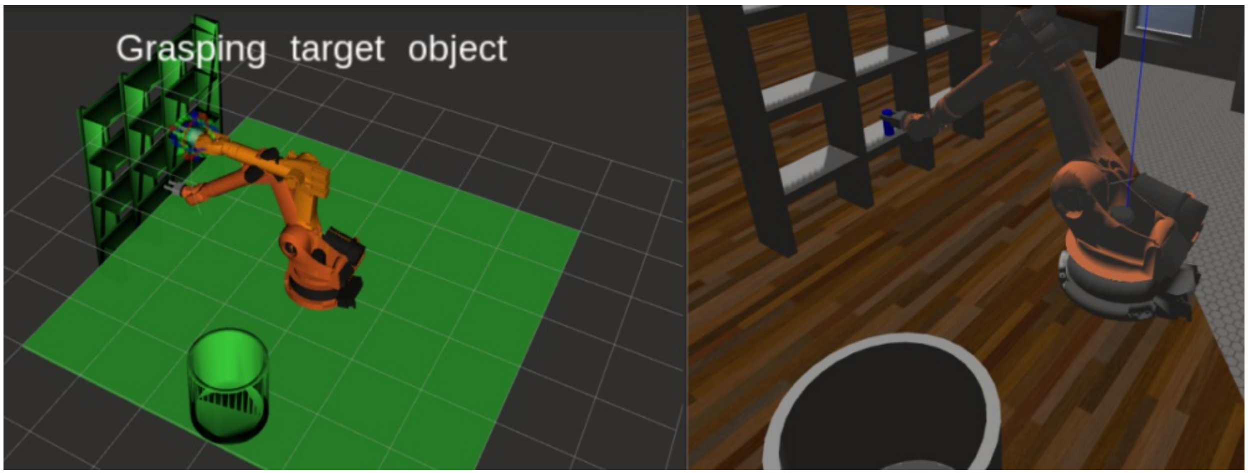
`θ6 = atan2(-r22, r21)`

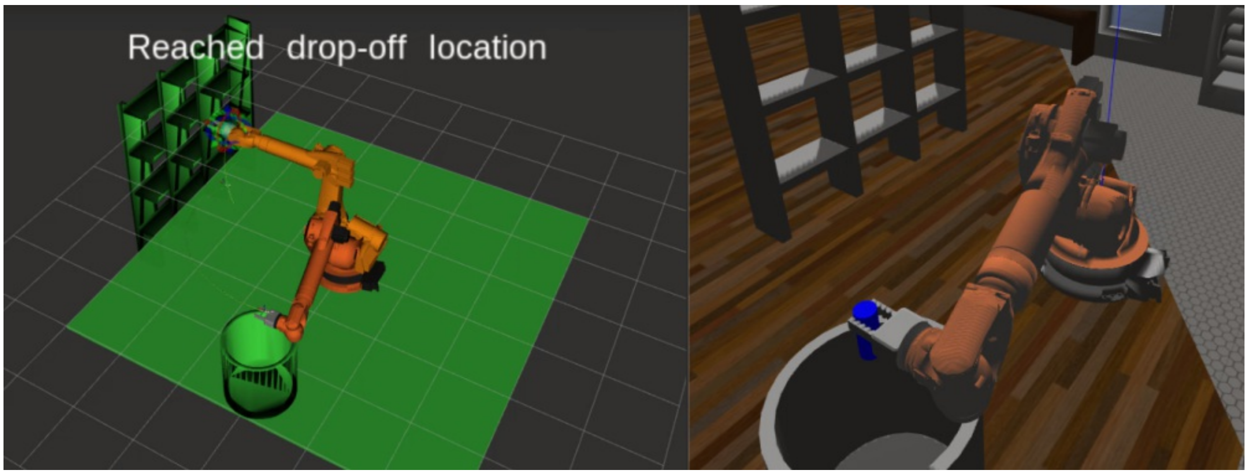
```
# Finding rotational components from 6th to 3rd frame
r33, r13 = R3_6[2,2], R3_6[0,2]
r23      = R3_6[1,2]
r22, r21 = R3_6[1,1], R3_6[1,0]

# Finding values of theta4, 5, and 6
theta4 = atan2(r33, -r13)
theta4 = theta4.evalf()

theta5 = atan2(√(1-r23*r23), r23)
theta5 = theta5.evalf()

theta6 = atan2(-r22, r21)
theta6 = theta6.evalf()
```

4. Results

The following section shows the Kuka Arm 210 grasping the object from the shelf and dropping it into the bin. The trajectory generated is followed by the Kuka Arm for the most part. There were times when the Kuka Arm does not follow the trajectory fully. This is due to the multiple solutions generated while calculating for joint angles.

A demo video (https://www.youtube.com/watch?v=1BXRThDDH1Q) on YouTube shows the Kuka Arm in action where it is grasping an object from 6 different locations on the shelf and dropping into the bin. The video runs at 2x speed.





5. Optimizing Inverse Kinematics

The initialization for all the homogeneous transforms are done outside of the main loop. To prevent numerical drift, only the transform from the *3rd-frame* to the ground frame is performed in every loop. The initialization function is shown below.

```
def initialize_transforms():
    start = time.time()

    # Creating global variables
    global T0_3, T0_1, T1_2, T2_3, T0_EE
    global q1, q2, q3, q4, q5, q6

    # Define DH param symbols
    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')

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

    # Modified DH params
    theta1, theta2, theta3 = q1, q2 - pi/2, q3
    theta4, theta5, theta6 = q4, q5, q6
    theta7 = q7

    s = {alpha0:     0, a0:   0   , d1: 0.75,
         alpha1: -pi/2, a1: 0.35  , d2: 0,
         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}
    # Define Modified DH Transformation matrix
    T0_1  = H_transform(alpha0, a0, d1, theta1)
    T1_2  = H_transform(alpha1, a1, d2, theta2)
    T2_3  = H_transform(alpha2, a2, d3, theta3)
    T3_4  = H_transform(alpha3, a3, d4, theta4)
    T4_5  = H_transform(alpha4, a4, d5, theta5)
    T5_6  = H_transform(alpha5, a5, d6, theta6)
    T6_EE = H_transform(alpha6, a6, d7, theta7)

    # Create individual transformation matrices
    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_EE = T6_EE.subs(s)
    T0_3  = T0_1*T1_2*T2_3

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

    print "[INITIALIZATION TIME] Time %04.4f seconds." % (time.time() - start)
```

The intialization step happens outside of the callback function for the ROS service requesting for poses.

```
def IK_server():
    # initialize node and declare calculate_ik service
    rospy.init_node('IK_server')

    # Initialize all DH parameters and relevant homogeneous transforms
    initialize_transforms()

    # Request service of type CalculateIK
    s = rospy.Service('calculate_ik', CalculateIK, handle_calculate_IK)
    print "Ready to receive an IK request."
    rospy.spin()

if __name__ == "__main__":
    global error

    ap = argparse.ArgumentParser()
    ap.add_argument("-e", "--error", type=int, default=-1)
    args    = vars(ap.parse_args())
    error   = args["error"]

    IK_server()
```

6. Calculating Forward Kinematics error

A flag can be accepted from the user for performing error analysis on the end-effector's position calculated by using forward kinematics and the end-effector's location provided as input from the `pose`.

The segment of code responsible for performing this error analysis is shown below:

```
# Calc FK EE error
if error is 1:
      FK = T0_EE.evalf(subs={q1: theta1, q2: theta2, q3: theta3, \
      q4: theta4, q5: theta5, q6: theta6 })
      ex = FK[0,3]
      ey = FK[1,3]
      ez = FK[2,3]
      ee_x_e = abs(ex-px)
      ee_y_e = abs(ey-py)
      ee_z_e = abs(ez-pz)
      ee_offset = math.sqrt(ee_x_e**2 + ee_y_e**2 + ee_z_e**2)
      print ("\nEnd effector x-error: %04.5f" % ee_x_e)
      print ("End effector y-error: %04.5f" % ee_y_e)
      print ("End effector z-error: %04.5f" % ee_z_e)
      print ("Overall end effector error: %04.5f units \n" % ee_offset)

      print ("[TOTAL RUN TIME: IK & FK] %04.4f seconds" % (time.time()-kinematics_s
```