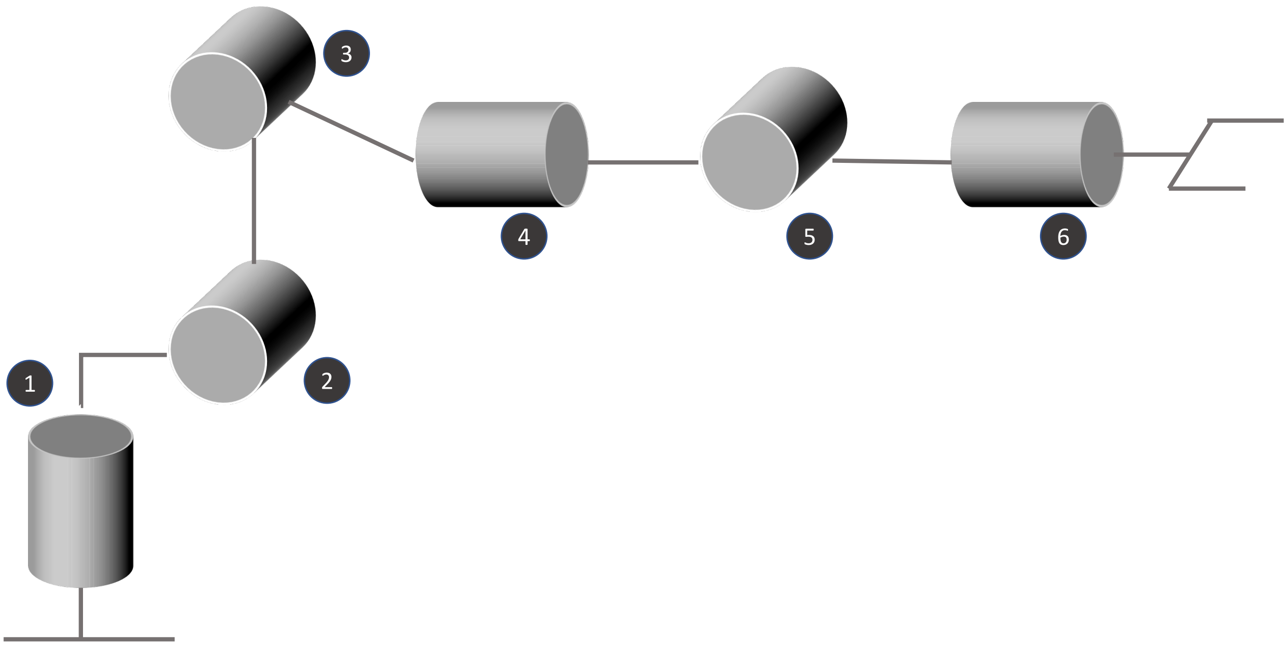
Write-up for the Project 2: Robotic Arm: Pick & Place

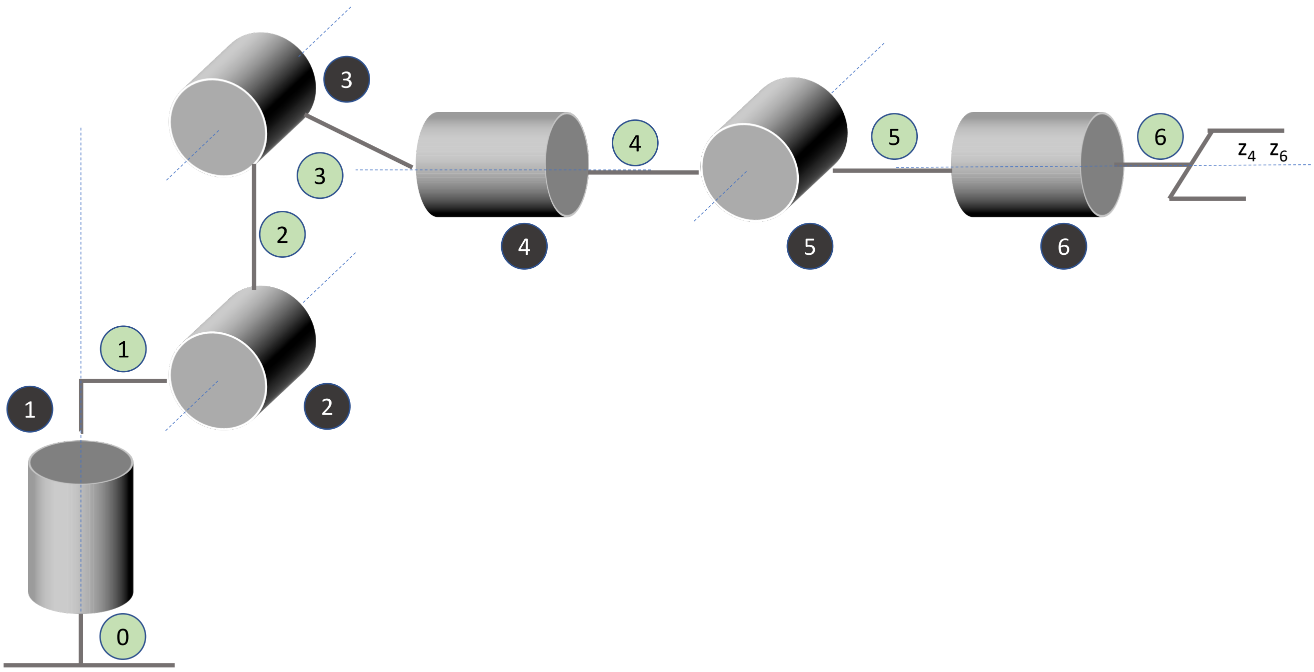
DH Parameters

First a representation of the Kuka KR210 robot as suggested in the lecture notes:



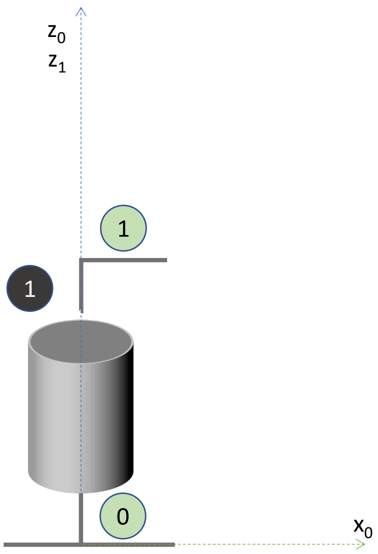
The 6 joints are numbered from 1 (next to base) to 6 (next to gripper). The orientation of the joints is determined by looking at the robot and understanding the degree of freedom that that particular joint implements.

Next, we draw the Z axis going through each of the joints (in blue in the next drawing). For the time being we're only interested in the direction of the Z-axes and how they relate to the axes of the adjacent joints. In the next drawing I also included the link numbers.

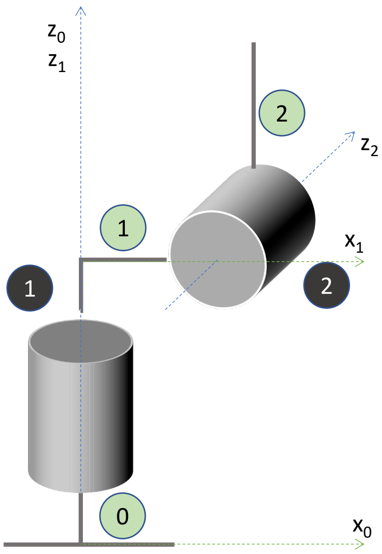


Now we want to define the orientation of the axes and add the X-axes corresponding to each joint.

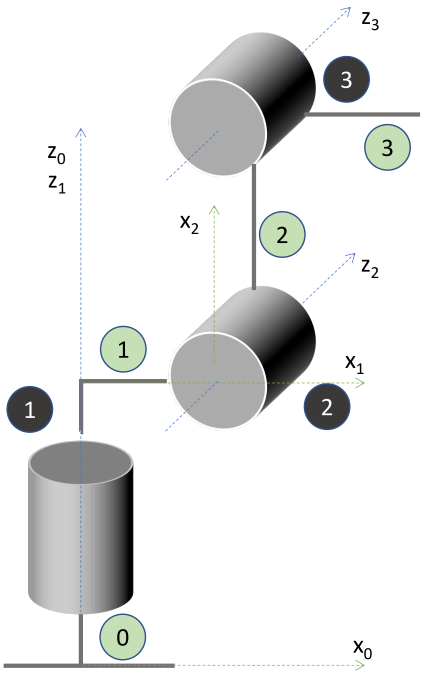
For Z_1 axis it will make sense to point upwards. Since we can choose relatively freely the axes for the base link (0) we will also consider Z_0 to point up, specifically to be coincident with the Z_1. We can choose X_0 almost in any way in the base plane, but since most of the robot is in the X-Z plane it will make more sense to choose the X_0 axis to be to the right (in the drawing above) parallel with the robot arm. This way, a 0 degrees orientation of joint 1 will align the arm along the base X axis.



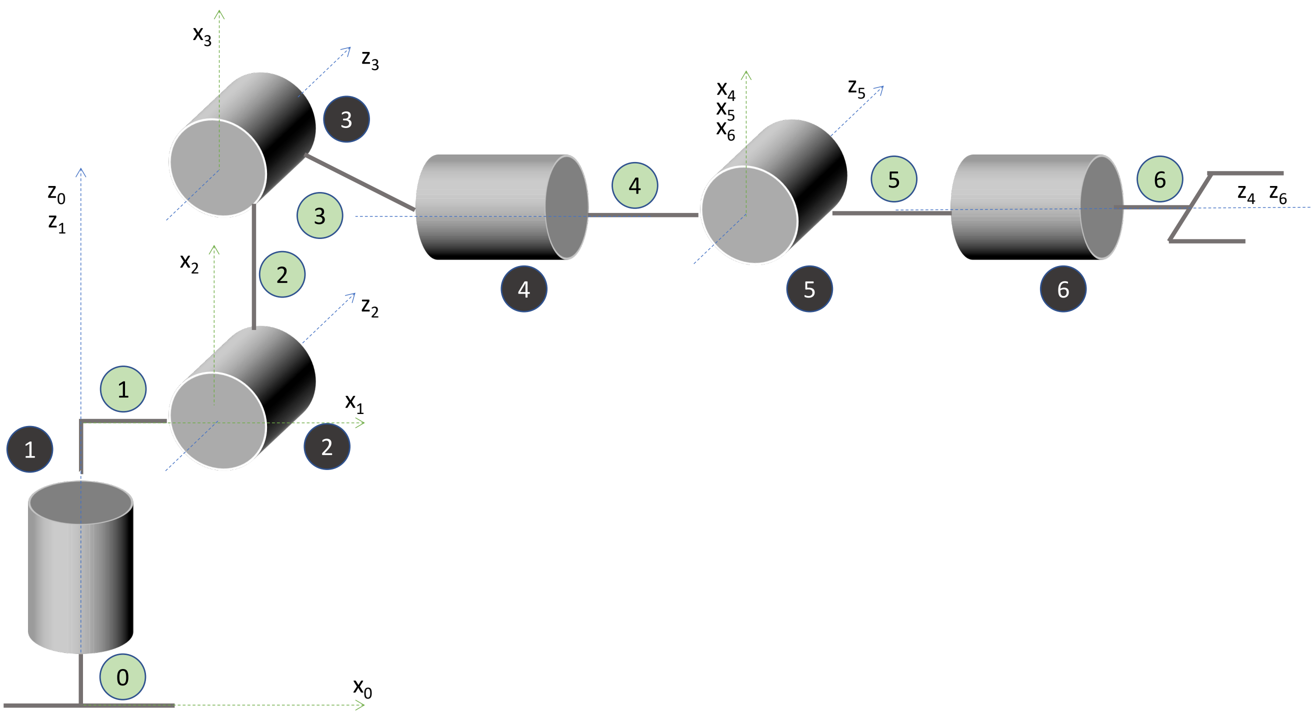
When choosing the X_1 axis we are interested in the normal between Z_1 and Z_2 and we're looking to at the intersection between this normal and Z_1. Therefore, the choice for X_1 is unique as seen in the next picture. When it comes to Z_2 we have the option to point it towards the back of the drawing or to have it "pop" out the picture towards us. We have used the suggestion from the lectures and defined Z_2 "going in" the image. The defined axes up to this point can be seen in the following picture.



Next, for joint 3, the Z axis is parallel with Z axis of joint 2 hence it will make sense to preserve the same orientation of the Z_3 axis, specifically it will "go into" the drawing, similar to the Z_2. The X_2 is on the normal between Z_2 and Z_3, but since these two are parallel there is an infinity of options. To keep things simple, we will consider X_2 simply pointing up from the same centre where X_1 and Z_2 intersect. This will make the parameter d_2 later 0. Here is the drawing with the axes up to this moment:

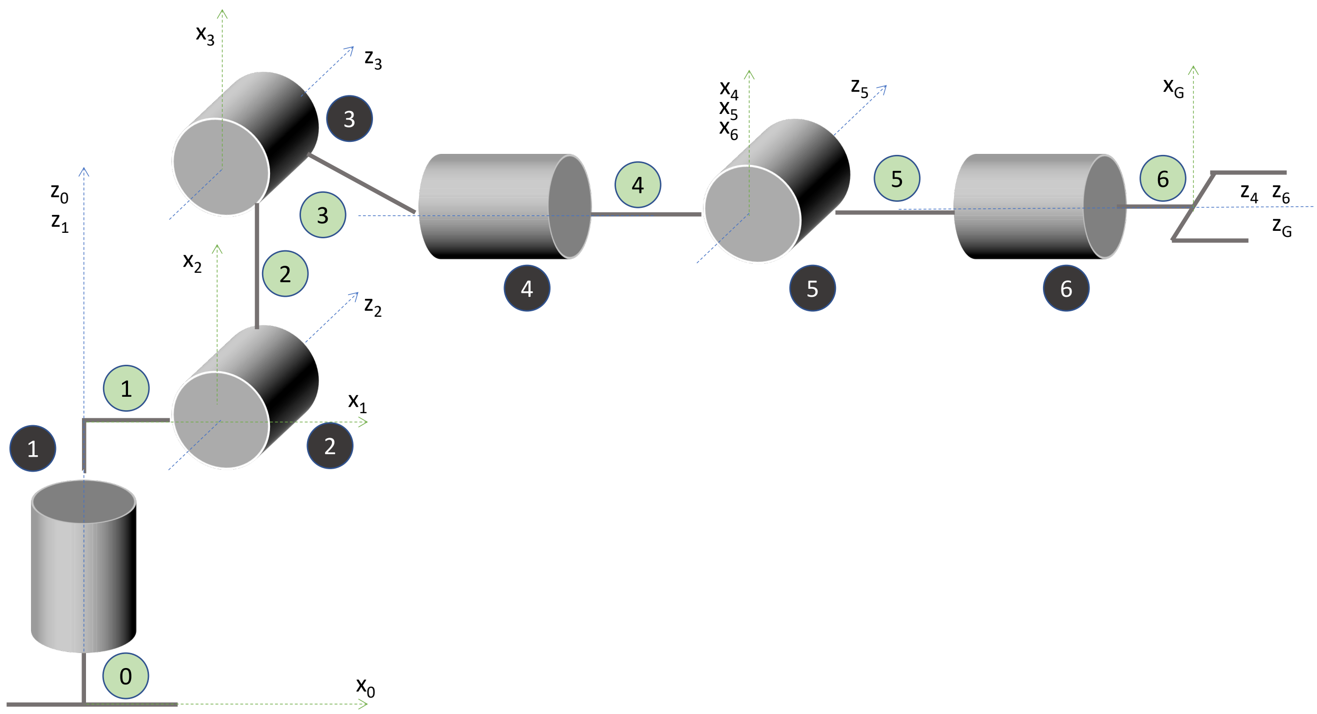


When considering the next 3 joints we will take the advice from the lecture to structure them as if they are a 3DOF orientation joint. Since joint 4 and 6 are collinear in Z axis and joint 5 is perpendicular on them we can define X_4, X_5 and X_6 anywhere perpendicular on the plane produced by Z_5 and Z_4,Z_6. One choice is to define all three of them at the intersection of Z_5 and $Z_4(Z_6)$ as shown in the following picture and also as suggested in the lecture.



In addition, X_3 is also defined perpendicular on the plane defined by Z_3 and $Z_4(Z_6)$, and we have chosen it to pint up from that intersection point.

As suggested in the lectures, we add one additional detail: the end-effector (gripper) and we describe the frame for this at the centre of the gripper. To keep things simple the Z axis will be also collinear with the Z_4 and Z_6 and X_G will also point up (like the other X axes) and will be located in the centre of the gripper. The full diagram of the axes will therefore be:



We will now determine the DH parameters for this kinematic chain. We will start with the d parameters:

- d_1 is the distance between the X_0 and X_1 axes; we will take this measure from the URDF file
- d_2 is the distance between the X_1 and X_2 axes; since they are intersecting this is 0
- d_3 is the distance between the X_2 and X_3 axes; since they are intersecting this is 0
- d_4 is the distance between the X_3 and X_4 axes; we will take this measure from the URDF file
- d_5 is the distance between the X_4 and X_5 axes; since they are collinear this is 0
- d_6 is the distance between the X_5 and X_6 axes; since they are collinear this is 0
- d_G is the distance between the X_G and X_6 axes; we will take this measure from the URDF file

The θ parameters are as follows:

- θ_1 is the angle between X_0 and X_1 around Z_1; this will be a parameter and <u>positive values for this parameter will indicate counter-clockwise rotation around Z_1 axis as viewed from the top</u>
- θ_2 is the angle between X_1 and X_2 around Z_2; this is -90 degrees plus a parameter. When this parameter is 0 (the "rest position" of the arm) the full angle is -90 degrees – consistent with the upward position of the robot arm. <u>Positive values for this parameter represent a counter-clockwise rotation around Z_2 axis.</u> Specifically, in the drawing above, it will be reflect a movement towards right (clock-wise) of the link 2. Although a little confusing we will consider the parameter that reflects the control of this joint to be θ_2 and the DH parameter that reflects the angle between the X_1 and X_2 axes as being θ_2 - 90°
- θ_3 is the angle between X_2 and X_3 around Z_3; this will be a parameter and <u>positive values for this parameter will indicate counter-clockwise rotation around Z_3 axis reflected by a lowering of the arm (or link 2)</u>

- θ_4 is the angle between X_4 and X_5 around Z_4; this will be a parameter and <u>positive values for this parameter will indicate counter-clockwise rotation around Z_4 axis as viewed from the end-effector.</u>
- θ_5 is the angle between X_5 and X_6 around Z_5; this will be a parameter and <u>positive values for this parameter will indicate counter-clockwise rotation around Z_5 axis reflected by a lowering of the gripper</u>
- θ_G is the angle between X_6 and X_G around Z_G; since the two are parallel this will be 0

The parameters α are as follows:

- α_0 is the angle between Z_0 and Z_1 around X_0; since Z_0 and Z_1 are collinear this parameter will be 0
- α_1 is the angle between Z_1 and Z_2 around X_1; this will be -90° (negative as it is clock-wise as seen from above X_1 axis)
- α_2 is the angle between Z_2 and Z_3 around X_2; since Z_3 and Z_2 are parallel this parameter will be 0
- α_3 is the angle between Z_3 and Z_4 around X_3; this will be -90° (negative as it is clock-wise as seen from above X_3 axis)
- α_4 is the angle between Z_4 and Z_5 around X_4; this will be 90° (positive as it is counter-clock-wise as seen from above X_4 axis)
- α_5 is the angle between Z_5 and Z_6 around X_5; this will be -90° (negative as it is clock-wise as seen from above X_5 axis)
- α_6 is the angle between Z_6 and Z_G around X_6; this will be 0

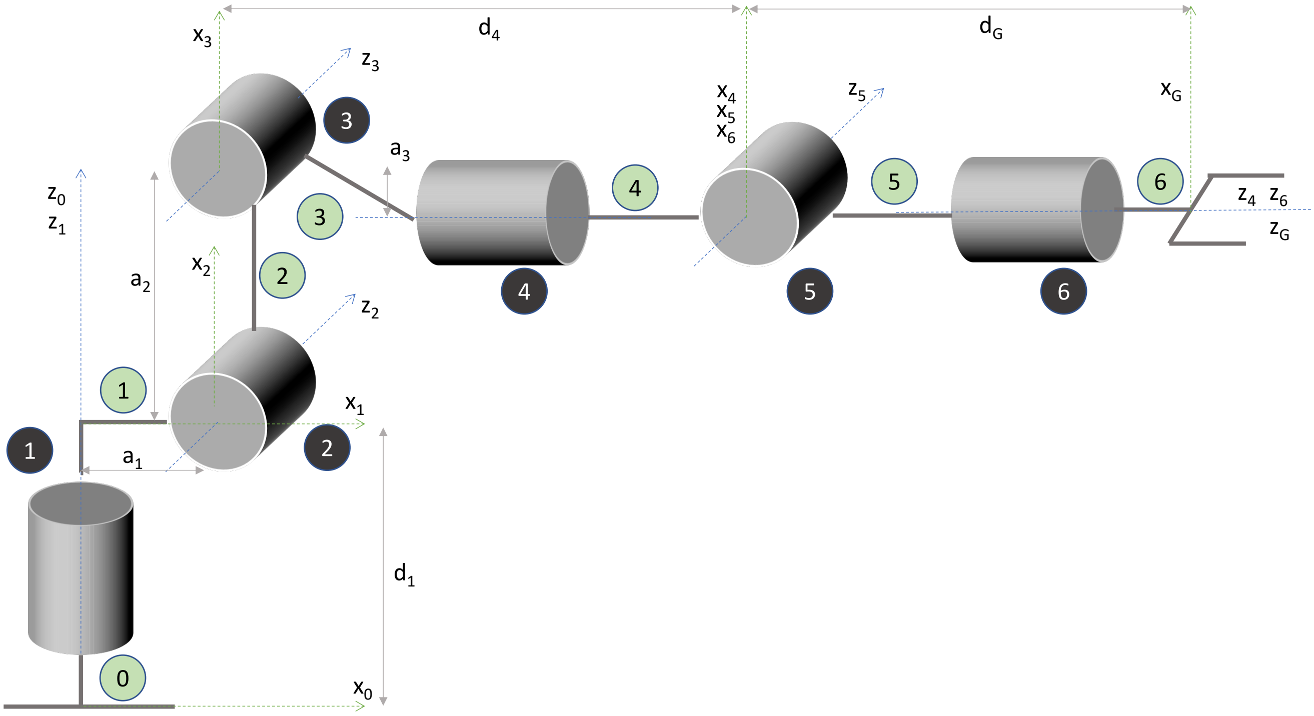
The parameters a are as follows:

- a_0 is the distance between Z_0 and Z_1 along X_0 axis; since Z_0 and Z_1 are collinear this parameter is 0
- a_1 is the distance between Z_1 and Z_2 along X_1 axis; this is a parameter we will determine from the URDF file
- a_2 is the distance between Z_2 and Z_3 along X_2 axis; this is a parameter we will determine from the URDF file
- a_3 is the distance between Z_3 and Z_4 along X_3 axis; this is a parameter we will determine from the URDF file
- a_4 is the distance between Z_4 and Z_5 along X_4 axis; since Z_4 and Z_5 are intersecting with X_4 in the same point this will be 0
- a_5 is the distance between Z_5 and Z_6 along X_5 axis; since Z_5 and Z_6 are intersecting with X_5 in the same point this will be 0
- a_6 is the distance between Z_6 and Z_G along X_6 axis; since Z_6 and Z_G are collinear this parameter is 0

The following table summarises the DH paramters.

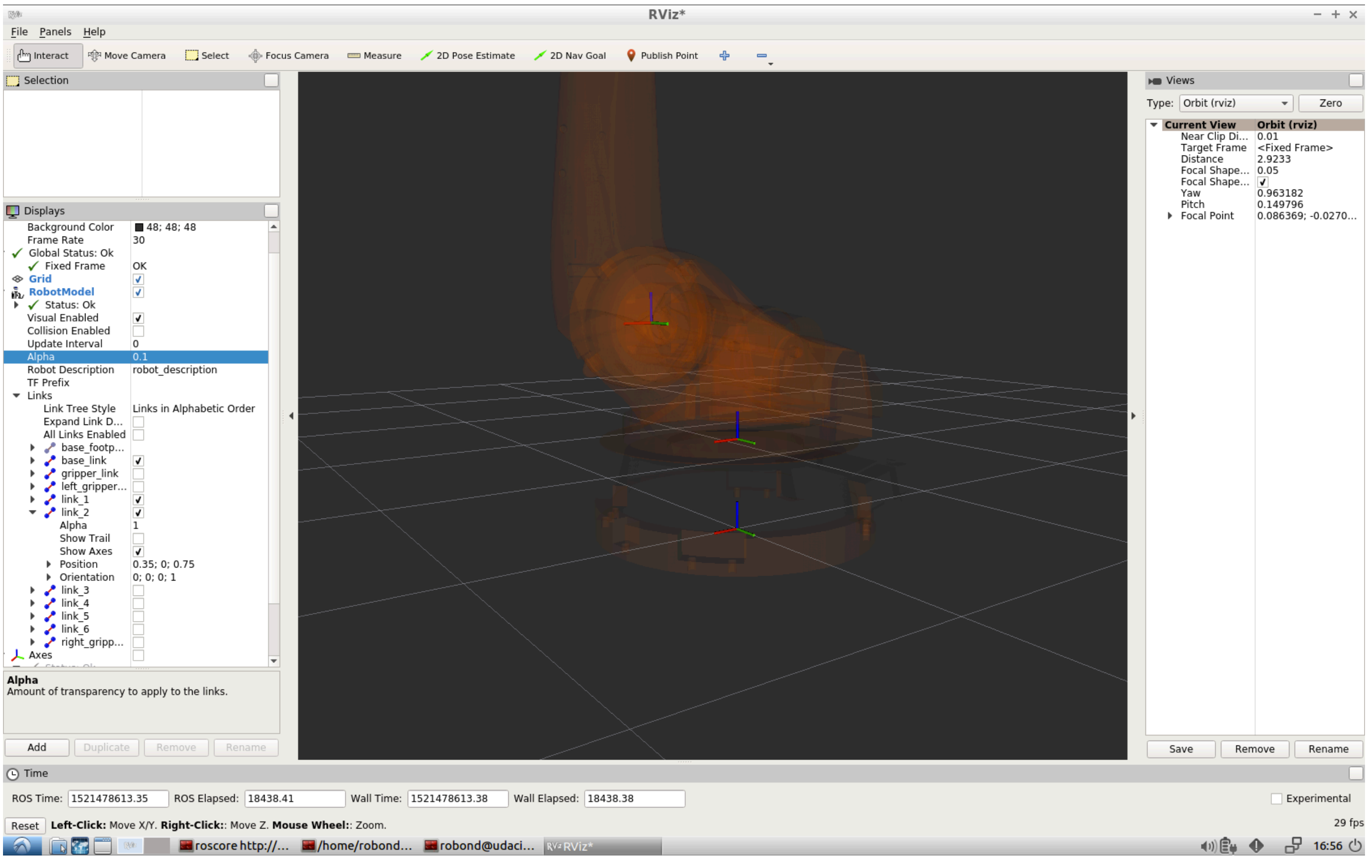
i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	d_1	θ_1
2	-90°	a_1	0	θ_2 - 90°
3	0	a_2	0	θ_3
4	-90°	a_3	d_4	θ_4
5	90°	0	0	θ_5
6	-90°	0	0	θ_6
G	0	0	d_G	0

In the table above d_i and α_i are values we will determined from the URDF file (constants) while θ_i will be variable that determine the positioning of the arm of the robot:



To determine the d_i and a_i parameters we are looking at the URDF file.

d_1 is the position of the joint 2 on the vertical axis relative to the ground as can be seen in the following image from RViz. The three axes shown are the ones for base_link, link_1 and link_2 (in order from bottom)



As seem from the picture the link_2 is located at 0.75 on the Z axis. This is also visible in the URDF file in the definition of the two joints (1 and 2):

```
  <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>
```

The position of the joint_2 is 0.33 + 0.42 = 0.75 relative to the base, consistent with the number from RViz.

From the same numbers above we can also determine the parameter a_1 which is 0.35 as reflected in the origin of the joint_2 (while the origin of joint_1 has a 0 on X axis).

a_2 is determined from the joint_3 paramters:

```
  <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>
```

It is the Z displacement between joint_2 and joint_3, specfically a_2 = 1.25.

The a_3 parameter is determined from the joint_4 paramters:

```
  <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>
```

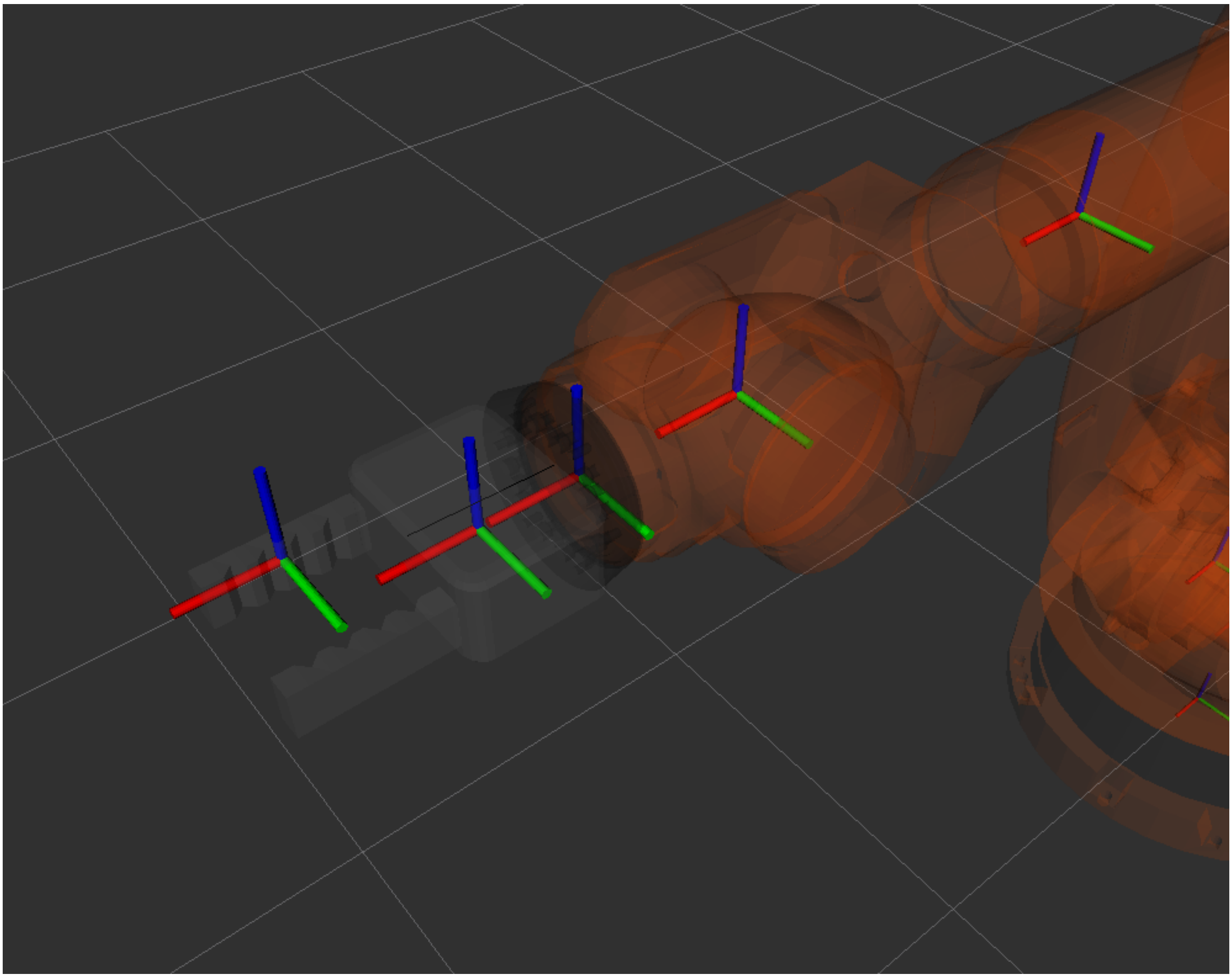
In this case the displacement along the RViz Z axis between the link_3 and link_4 is what we are after. This is -0.054 consistent with the fact that the difference is downwards (as both the RViz Z axis and the DH model Z axis is pointing upwards). So a_3 = -0.054.

The d_4 parameter can be determined from the information from joint_4 and joint_5. According to the URDF model the joints are all aligned in the same way (rpy = “0 0 0”) which means we can simply add the X displacement to produce the d_4 parameter:

```
  <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>
```

So d_4 = 0.96 + 0.54 = 1.5.

The last parameter we need to determine is d_G which can be produced from the information of joint_6 and the gripper. If we look carefully on how the joints are presented in the URDF (RViz) we can see the alignment of the joint_5, joint_6 and the gripper details. What we want for the d_G to be long enough so that the end-effector sits in a comfortable range within the grasp of the gripper and not too close to the joint_6 to that we will hit the target object with the gripper joint.



Looking carefully at the picture above we can determine d_G as the summation of the displacements along X axis from joint_5 to joint_6 then from joint_6 to gripper_joint. With this value of d_G simply guiding the resulting G point in the kinematics will give us a sufficient enough precision.

To determine the displacements, we look at the definitions of the joints and gripper in the URDF file:

```
  <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>
```

And

```
  <joint name="gripper_joint" type="fixed">
    <parent link="link_6"/>
    <child link="gripper_link"/>
    <origin xyz="0.11 0 0" rpy="0 0 0"/><!--0.087-->
    <axis xyz="0 1 0" />
```

```
  </joint>
```

So, $d_G = 0.193 + 0.11 = 0.303$.

We can now re-write the DH parameters table as follows:

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0.75	θ_1
2	-90°	0.35	0	θ_2 - 90°
3	0	1.25	0	θ_3
4	-90°	- 0.054	1.5	θ_4
5	90°	0	0	θ_5
6	-90°	0	0	θ_6
G	0	0	0.303	0

Now, the end-effector (G) position and orientation can be entirely controlled using the 6 parameters θ_1 - θ_6

As mentioned in the course notes in the kinematic chain we designed the orientation of the Gripper is with X axis pointing up and the Z axis pointing towards the front. The URDF definition of the robot and the resulting representation in RViz has the gripper oriented with the X axis towards the front and the Z axis upwards.

To match the reported position of the gripper with the calculated one from our kinematic chain we need to perform a transformation of the final gripper by: rotating around Z axis by 180° (pi) and then rotating around Y axis by -90° (-pi/2). These final calculation will be included in the code bellow.

Forward Kinematic Transformations

The code for calculating the forward kinematics was written in Python using numpy and is provided in the submission as a jupyter notebook (forward_kinematics.ipynb). I have preferred using numpy as opposed to the sympy used in the lectures as I found sympy very slow in execution and not necessarily more accurate nor easier to use.

We will detail here the main parts of the code.

We first define the DH parameters as a two dimensional list. Each row represents the parameters for one transformation from one frame to the other, in order alpha (α), a, theta(θ) and d as defined in the table in the previous section of the document:

```
DH = [[0        , 0     , 0.75 , 0          ],
      [-np.pi/2, 0.35  , 0    , - np.pi/2],
      [0        , 1.25  , 0    , 0          ],
      [-np.pi/2, -0.054, 1.5  , 0          ],
      [np.pi/2 , 0     , 0    , 0          ],
      [-np.pi/2, 0     , 0    , 0          ],
      [0        , 0     , 0.303, 0          ]]
```

The last column in the table (theta) contains only the offsets applied to the control angles that we provide to the robot arm to position the end-effector. Later we will add the actual control angles to that column before calculating the transformation matrices.

We then define a help function that calculates a transformation matrix 4 x 4 for a given set of DH parameters according to the formula:

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

```
def LinkTransform(params):
    # params is a list of length 4: alpha, a, d, theta in this order
    alpha, a, d, theta = params

    # returns a Matrix of transformation for the given DH paramters
    return np.array([[np.cos(theta), -np.sin(theta), 0, a],
                     [np.sin(theta)*np.cos(alpha), np.cos(theta)*np.cos(alpha),
                                 np.sin(alpha), -np.sin(alpha)*d],
                     [np.sin(theta)*np.sin(alpha), np.cos(theta)*np.sin(alpha),
                                 np.cos(alpha) , np.cos(alpha)*d],
                     [0, 0, 0, 1]])
```

We can now define a function that will iterate over the DH parameter table and receive an actual list of theta angles and constructs a full transformation matrix from the beginning of the chain to the end. In this function we add the particular theta for that step to the previous value of theta from the DH parameters table then pass the set of parameters to the LinkTransform above to calculate each transformation matrix. Finally, we multiply the matrices to create the complete transform across the whole chain and we return this full transform matrix:

```
def ChainLinkTransform(DH, thetas, printTransformations = False):
    for i in range(len(DH)):
        params = list(DH[i])    # we need to make a copy
        params[3] += thetas[i]
        T = LinkTransform(params)
        if i == 0:
            result = T
        else:
            result = np.matmul(result, T)

        if printTransformations:
            print("T%d_%d = " % (i, i+1))
            print(T)
            print("T0_%d = " % (i+1))
            print(result)

    return result
```

As mentioned above we also need to define a function that will provide the orientation adjustment of the gripper so that the results are aligned with the ones reported by ROS. This function simply builds a Z rotation and a Y rotation transformation and multiplies them using the angles specified in the function parameters. By default, the angles are pi for Z transform and -pi/2 for Y transform. The function returns the resulting matrix.

```
def GripperAdust(r_z = np.pi, r_y = -np.pi/2):
    R_z = np.array([[np.cos(r_z), -np.sin(r_z), 0, 0],
                    [np.sin(r_z), np.cos(r_z) , 0, 0],
                    [0          , 0           , 1, 0],
                    [0          , 0           , 0, 1]])
    R_y = np.array([[np.cos(r_y) , 0          , np.sin(r_y), 0],
                    [0           , 1          , 0          , 0],
                    [-np.sin(r_y), 0          , np.cos(r_y), 0],
                    [0           , 0          , 0          , 1]])
    return np.matmul(R_z, R_y)
```

We also define a help function that extracts the position and the orientation of a point from a 4x4 quaternion according to the formulas presented in the course.

```
def OrientationFromQuaternion(Q):
    pos = Q[:,3].T
    orient = np.array([np.arctan2(Q[2,1], Q[2,2]),
                       np.arctan2(-Q[2,0], np.sqrt(Q[0,0]**2 + Q[1,0]**2)),
                       np.arctan2(Q[1,0], Q[0,0])])

    return pos[0:3], orient
```

We can now combine all the functions in one function that receives the DH table, the list of angles theta and prints the resulting position and orientation of the gripper. This function first calculates the chain transformation matrix, then the adjustment matrix for the gripper, multiples them and then passes the resulting matrix to the function that extract the position and the orientation.

```
def CalculateEffector(DH, thetas, printTransformations = False):
    res = ChainLinkTransform(DH, thetas, printTransformations)
    # adjust orientation
    adj = GripperAdust()
    res = np.matmul(res, adj)
    if printTransformations:
        print("Adjustment= ")
        print(adj)
        print("Adjusted  = ")
        print(res)

    pos, orient = OrientationFromQuaternion(res)
    print("pos    = "+str(pos))
    print("orient = "+str(orient))
```

We can now simply call this function as follows:

```
th = [0, 0, 0, 0, 0, 0, 0]
CalculateEffector(DH, th)
```

And the function will report :

```
pos    = [ 2.15300  0.00000  1.94600]
orient = [-0.00000  0.00000  0.00000]
```

We have performed the calculations for a few angles and compared them with the values reported by ROS:

In [22]:
```
th = [0, 0, 0, 0, 0, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 2.15300  0.00000  1.94600]
orient = [-0.00000  0.00000  0.00000]
```

ROS Reported:
```
Trans = [2.153, 0.000, 1.947]
Rot = [0.000, 0.000, 0.000]
```

In [23]:
```
th = [0.99, 0, 0, 0, 0, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.18133  1.79996  1.94600]
orient = [-0.00000  0.00000  0.99000]
```

ROS Reported:
```
Trans = [1.173, 1.805, 1.947]
Rot = [0.000, 0.000, 0.994]
```

In [27]:
```
th = [0.99, 0.32, 0, 0, 0, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.33754  2.03797  1.31812]
orient = [-0.00000  0.32000  0.99000]
```

ROS Reported:
```
Trans = [1.328, 2.044, 1.321]
Rot = [0.000, 0.319, 0.994]
```

In [28]:
```
th = [0.99, 0.32, -0.49, 0, 0, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.38783  2.11461  2.18836]
orient = [-0.00000 -0.17000  0.99000]
```

ROS Reported:
```
Trans = [1.377, 2.120, 2.190]
Rot = [0.000, -0.171, 0.994]
```

In [29]:
```
th = [0.99, 0.32, -0.49, 1.05, 0, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.38783  2.11461  2.18836]
orient = [ 1.05000 -0.17000  0.99000]
```

ROS Reported:
```
Trans = [1.377, 2.120, 2.190]
Rot   = [1.046, -0.171, 0.994]
```

In [30]:
```
th = [0.99, 0.32, -0.49, 1.05, 0.99, 0, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.14188  2.14032  2.04100]
orient = [ 1.12313  0.32273  1.86052]
```

ROS Reported:
```
Trans = [1.133, 2.144, 2.042]
Rot   = [1.119, 0.324, 1.860]
```

In [31]:
```
th = [0.99, 0.32, -0.49, 1.05, 0.99, -0.44, 0]
CalculateEffector(DH, th)
```
```
pos    = [ 1.14188  2.14032  2.04100]
orient = [ 0.68313  0.32273  1.86052]
```

ROS Reported:
```
Trans = [1.133, 2.144, 2.042]
Rot   = [0.678, 0.324, 1.860]
```

You can see that there are some differences in the order of 7-8mm for the position and less than 0.005 rad for angles.

(Update for Forward Kinematics)

In the review of my submission it was pointed out that there is a requirement to present the results for the forward kinematics as homogenous parametric equation of theta angles. In this updated section I will indicate the results of using a Jupyter notebook and scypy to calculate these homogenous transformations.

In this notebook we first define the symbols for the 6 theta angles and the DH parameters for the robot:

```
th1, th2, th3, th4, th5, th6 = sp.symbols('th1:7')

# the DH paramters for KUKA robot as per written document
# alpha, a, d, theta (adjustmet)
# the angles we will use to control the robot will be added to the theta paramters
DH = [[0        , 0     , 0.75 , th1          ],
      [-sp.pi/2, 0.35  , 0    , th2 - sp.pi/2],
      [0        , 1.25  , 0    , th3          ],
      [-sp.pi/2, -0.054, 1.5  , th4          ],
      [sp.pi/2 , 0     , 0    , th5          ],
      [-sp.pi/2, 0     , 0    , th6          ],
      [0        , 0     , 0.303, 0            ]]
```

We now define a method that produces a symbolic homogenous transformation from the DH parameters:

```
# a help function that builds a transformation matrix given the 4 DH paramters for
that joint
def LinkTransform(params):
    # params is a list of length 4: alpha, a, d, theta in this order
    alpha = params[0]
    a = params[1]
    d = params[2]
    theta = params[3]
    # returns a Matrix of transformation for the given DH paramters
    return Matrix([[sp.cos(theta), -sp.sin(theta)             , 0              , a],
                   [sp.sin(theta)*sp.cos(alpha), sp.cos(theta)*sp.cos(alpha), -
sp.sin(alpha), -sp.sin(alpha)*d],
                   [sp.sin(theta)*sp.sin(alpha), sp.cos(theta)*sp.sin(alpha),
sp.cos(alpha) , sp.cos(alpha)*d],
                   [0            , 0                          , 0              , 1]])
```

We also have a method that produces the homogenous transformation for the griper adjustment (similar to what we did earlier but this time using sympy):

```
# builds a transformation matrix for the orientation adjustment of the gripper
# so that we are consistent with the URDF representaiton of the gripper
def GripperAdjust(r_z = sp.pi, r_y = -sp.pi/2):
    R_z = Matrix([[sp.cos(r_z), -sp.sin(r_z), 0, 0],
```

```
                 [sp.sin(r_z), sp.cos(r_z) , 0, 0],
                 [0          , 0           , 1, 0],
                 [0          , 0           , 0, 1]])
    R_y = Matrix([[sp.cos(r_y) , 0         , sp.sin(r_y), 0],
                 [0           , 1          , 0          , 0],
                 [-sp.sin(r_y), 0          , sp.cos(r_y), 0],
                 [0           , 0          , 0          , 1]])
    return sp.simplify(R_z * R_y)
```

We now can compute each individual transformation and print it:

```
T01 = LinkTransform(DH[0])
print('T01 = '+str(sp.simplify(T01))+'\n')
T12 = LinkTransform(DH[1])
print('T12 = '+str(sp.simplify(T12))+'\n')
T23 = LinkTransform(DH[2])
print('T23 = '+str(sp.simplify(T23))+'\n')
T34 = LinkTransform(DH[3])
print('T34 = '+str(sp.simplify(T34))+'\n')
T45 = LinkTransform(DH[4])
print('T45 = '+str(sp.simplify(T45))+'\n')
T56 = LinkTransform(DH[5])
print('T56 = '+str(sp.simplify(T56))+'\n')
T6G = LinkTransform(DH[6])
print('T6G = '+str(sp.simplify(T6G))+'\n')
print('TGA = '+str(GripperAdjust()))
```

The last two are the transformation from the joint 6 to the gripper according to the coordinate system that we defined in the pictures in the first section of the document and the transformation from that to the coordinate system of the gripper according to the Gazebo coordinates (TGA – A stands for "adjusted")

The results are as follows:

```
T01 = Matrix([
[cos(th1), -sin(th1), 0,    0],
[sin(th1),  cos(th1), 0,    0],
[       0,         0, 1, 0.75],
[       0,         0, 0,    1]])

T12 = Matrix([
[sin(th2),  cos(th2), 0, 0.35],
[       0,         0, 1,    0],
[cos(th2), -sin(th2), 0,    0],
[       0,         0, 0,    1]])

T23 = Matrix([
[cos(th3), -sin(th3), 0, 1.25],
[sin(th3),  cos(th3), 0,    0],
[       0,         0, 1,    0],
[       0,         0, 0,    1]])

T34 = Matrix([
[ cos(th4), -sin(th4), 0, -0.054],
[        0,         0, 1,    1.5],
[-sin(th4), -cos(th4), 0,      0],
[        0,         0, 0,      1]])
```

```
T45 = Matrix([
[cos(th5), -sin(th5),  0, 0],
[       0,         0, -1, 0],
[sin(th5),  cos(th5),  0, 0],
[       0,         0,  0, 1]])

T56 = Matrix([
[ cos(th6), -sin(th6), 0, 0],
[        0,         0, 1, 0],
[-sin(th6), -cos(th6), 0, 0],
[        0,         0, 0, 1]])

T6G = Matrix([
[1, 0, 0,     0],
[0, 1, 0,     0],
[0, 0, 1, 0.303],
[0, 0, 0,     1]])

TGA = Matrix([
[0,  0, 1, 0],
[0, -1, 0, 0],
[1,  0, 0, 0],
[0,  0, 0, 1]])
```

We can calculate the combined transformation in symbolic format for the whole chain T0_G (without adjustment):

```
T0G = sp.simplify(T01*T12*T23*T34*T45*T56*T6G)
print('T0G unajusted = '+str(T0G))
```

The result is quite long and to make it easier to read, if we represent the resulting matrix as:

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ r_{10} & r_{11} & r_{12} & r_{13} \\ r_{20} & r_{21} & r_{22} & r_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then we can express T0_G (without adjustment) parameters as follows:

```
r00 = ((sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*cos(th5) + sin(th5)
*cos(th1)*cos(th2 + th3))*cos(th6) - (-sin(th1)*cos(th4) + sin(th4)*sin(th2 + th
3)*cos(th1))*sin(th6)

r01 = -((sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*cos(th5) + sin(th5
)*cos(th1)*cos(th2 + th3))*sin(th6) + (sin(th1)*cos(th4) - sin(th4)*sin(th2 + th
3)*cos(th1))*cos(th6)

r02 = -(sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*sin(th5) + cos(th1)
*cos(th5)*cos(th2 + th3)

r03 = -0.303*sin(th1)*sin(th4)*sin(th5) + 1.25*sin(th2)*cos(th1) - 0.303*sin(th5)
*sin(th2 + th3)*cos(th1)*cos(th4) - 0.054*sin(th2 + th3)*cos(th1) + 0.303*cos(th
1)*cos(th5)*cos(th2 + th3) + 1.5*cos(th1)*cos(th2 + th3) + 0.35*cos(th1)
```

```
r10 = ((sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*cos(th5) + sin(th1)*sin(th5)*cos(th2 + th3))*cos(th6) - (sin(th1)*sin(th4)*sin(th2 + th3) + cos(th1)*cos(th4))*sin(th6)
r11 = -((sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*cos(th5) + sin(th1)*sin(th5)*cos(th2 + th3))*sin(th6) - (sin(th1)*sin(th4)*sin(th2 + th3) + cos(th1)*cos(th4))*cos(th6)

r12 = -(sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*sin(th5) + sin(th1)*cos(th5)*cos(th2 + th3)

r13 = 1.25*sin(th1)*sin(th2) - 0.303*sin(th1)*sin(th5)*sin(th2 + th3)*cos(th4) - 0.054*sin(th1)*sin(th2 + th3) + 0.303*sin(th1)*cos(th5)*cos(th2 + th3) + 1.5*sin(th1)*cos(th2 + th3) + 0.35*sin(th1) + 0.303*sin(th4)*sin(th5)*cos(th1)

r20 = -(sin(th5)*sin(th2 + th3) - cos(th4)*cos(th5)*cos(th2 + th3))*cos(th6) - sin(th4)*sin(th6)*cos(th2 + th3)

r21 = (sin(th5)*sin(th2 + th3) - cos(th4)*cos(th5)*cos(th2 + th3))*sin(th6) - sin(th4)*cos(th6)*cos(th2 + th3)

r22 = -sin(th5)*cos(th4)*cos(th2 + th3) - sin(th2 + th3)*cos(th5)

r23 = -0.303*sin(th5)*cos(th4)*cos(th2 + th3) - 0.303*sin(th2 + th3)*cos(th5) - 1.5*sin(th2 + th3) + 1.25*cos(th2) - 0.054*cos(th2 + th3) + 0.75
```

And T0_GA (with adjustment) as follows:

```
r00 = -(sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*sin(th5) + cos(th1)*cos(th5)*cos(th2 + th3)

r01 = ((sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*cos(th5) + sin(th5)*cos(th1)*cos(th2 + th3))*sin(th6) - (sin(th1)*cos(th4) - sin(th4)*sin(th2 + th3)*cos(th1))*cos(th6)

r02 = ((sin(th1)*sin(th4) + sin(th2 + th3)*cos(th1)*cos(th4))*cos(th5) + sin(th5)*cos(th1)*cos(th2 + th3))*cos(th6) + (sin(th1)*cos(th4) - sin(th4)*sin(th2 + th3)*cos(th1))*sin(th6)

r03 = -0.303*sin(th1)*sin(th4)*sin(th5) + 1.25*sin(th2)*cos(th1) - 0.303*sin(th5)*sin(th2 + th3)*cos(th1)*cos(th4) - 0.054*sin(th2 + th3)*cos(th1) + 0.303*cos(th1)*cos(th5)*cos(th2 + th3) + 1.5*cos(th1)*cos(th2 + th3) + 0.35*cos(th1)

r10 = -(sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*sin(th5) + sin(th1)*cos(th5)*cos(th2 + th3)

r11 = ((sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*cos(th5) + sin(th1)*sin(th5)*cos(th2 + th3))*sin(th6) + (sin(th1)*sin(th4)*sin(th2 + th3) + cos(th1)*cos(th4))*cos(th6)

r12 = ((sin(th1)*sin(th2 + th3)*cos(th4) - sin(th4)*cos(th1))*cos(th5) + sin(th1)*sin(th5)*cos(th2 + th3))*cos(th6) - (sin(th1)*sin(th4)*sin(th2 + th3) + cos(th1)*cos(th4))*sin(th6)
```

r_{13} = 1.25*sin(th1)*sin(th2) - 0.303*sin(th1)*sin(th5)*sin(th2 + th3)*cos(th4) - 0.054*sin(th1)*sin(th2 + th3) + 0.303*sin(th1)*cos(th5)*cos(th2 + th3) + 1.5*sin(th1)*cos(th2 + th3) + 0.35*sin(th1) + 0.303*sin(th4)*sin(th5)*cos(th1)

r_{20} = -sin(th5)*cos(th4)*cos(th2 + th3) - sin(th2 + th3)*cos(th5)

r_{21} = -(sin(th5)*sin(th2 + th3) - cos(th4)*cos(th5)*cos(th2 + th3))*sin(th6) + sin(th4)*cos(th6)*cos(th2 + th3)

r_{22} = -(sin(th5)*sin(th2 + th3) - cos(th4)*cos(th5)*cos(th2 + th3))*cos(th6) - sin(th4)*sin(th6)*cos(th2 + th3)

r_{23} = -0.303*sin(th5)*cos(th4)*cos(th2 + th3) - 0.303*sin(th2 + th3)*cos(th5) - 1.5*sin(th2 + th3) + 1.25*cos(th2) - 0.054*cos(th2 + th3) + 0.75

Inverse Kinematics

For inverse kinematics, since we need to use functionality in both the `IK_debug.py` and the actual server `IK_server.py`, I have decided to create a separate Python file that will define the functions needed in the IK calculations and include it in both the `IK_debug.py` and `IK_server.py`. To keep things simple, I have moved the `IK_debug.py` file in the scripts directory so that all these three files are in the same place. The code for the `IK_support.py` is included with the other two scripts in the submission.

Here we describe the content of the scripts and detail some of the calculations performed.

We start the IK_support.py by defining the DH parameters for the KUKA arm, similar to the definition we used in the forward kinematics earlier (the forward kinematics was included in a Jupyter notebook and unfortunately we cannot re-use it in a plain Phyton script so there will be some repetition here):

```
# DH robot paramters
DH = [[0        , 0     , 0.75 , 0         ],
      [-np.pi/2, 0.35   , 0    , - np.pi/2],
      [0        , 1.25   , 0    , 0         ],
      [-np.pi/2, -0.054, 1.5   , 0         ],
      [np.pi/2 , 0      , 0    , 0         ],
      [-np.pi/2, 0      , 0    , 0         ],
      [0        , 0      , 0.303, 0         ]]
```

There is nothing special about the DH parameters – we are using the exact same rules we have used in the forward kinematics.

We then define 3 help functions that will return a 3x3 rotation matrix around axes X. Y and Z given a certain angle for each:

```
def Rot_X(roll):
    # computes a 4x4 transformation matrix for rotation around X axis
    return np.array([[ 1, 0           , 0            ],
                     [ 0, np.cos(roll), -np.sin(roll)],
                     [ 0, np.sin(roll),  np.cos(roll)]])

def Rot_Y(pitch):
    # computes a 4x4 transformation matrix for rotation around Y axis
```

```
    return np.array([[ np.cos(pitch), 0, np.sin(pitch)],
                     [ 0            , 1, 0            ],
                     [-np.sin(pitch), 0, np.cos(pitch)]])

def Rot_Z(yaw):
    # computes a 4x4 transformation matrix for rotation around Z axis
    return np.array([[ np.cos(yaw), -np.sin(yaw), 0],
                     [ np.sin(yaw),  np.cos(yaw), 0],
                     [ 0          ,  0          , 1]])
```

In principle we could have used stock methods from tf or numpy to perform these, but for learning purposes we're defining our own functions.

We now can also define a combined method that does a ZYX rotation (in that order) given the yaw, pitch and roll angles for each rotation:

```
def Rot_ZYX(yaw, pitch, roll):
    # combines rotations in Z, Y, X axis in this order using the angles
    # provided
    rot_zy = np.matmul(Rot_Z(yaw), Rot_Y(pitch))
    rot_zyx = np.matmul(rot_zy, Rot_X(roll))
    return rot_zyx
```

Because of the difference in the orientation of the gripper in ROS and in DH model (discussed before in the forward kinematics chapter) we also define a help function that is performing the pi Z axis and -pi/2 Y axis rotation to convert from one orientation to the other. Since we are only interested in the rotation matrices we are using the help functions defined above and we only produce the rotation matrix as opposed to the full homogenous 4x4 matrix that we used in the chapter with the forward kinematics. For what we need in IK this is sufficient. So here is the function:

```
def GripperCorrection():
    # returns a correction matrix 4x4 with the pose correction
    # for the gripper: rotation in Z by pi and Y by -pi/2
    return np.matmul(Rot_Z(np.pi), Rot_Y(-np.pi/2.0))
```

Because in the IK_debug we will also apply the forward kinematics to compare the results of the calculations from our IK model, we also use the following method that produces a transformation matrix for a given set of DH parameters, similar to the code from the forward kinematics chapter:

```
def LinkTransform(params):
    # produces a homogenous transformation matrix 4x4 base on DH params
    # params is a list of length 4: alpha, a, d, theta in this order
    alpha, a, d, theta = params

    # returns a Matrix of transformation for the given DH paramters
    return np.array([[np.cos(theta)              , -np.sin(theta)             , 0
, a],
                     [np.sin(theta)*np.cos(alpha), np.cos(theta)*np.cos(alpha), -
np.sin(alpha), -np.sin(alpha)*d],
                     [np.sin(theta)*np.sin(alpha), np.cos(theta)*np.sin(alpha),
np.cos(alpha) , np.cos(alpha)*d],
                     [0                          , 0                          , 0
, 1]])
```

And the method that produces a matrix for a full chain of DH parameters:

```
def ChainLinkTransform(DH, thetas, show = False):
    # builds the transformation matrices based on the DH and the theta paramters
    # it resurns the final transformation matrix
    for i in range(len(DH)):
        params = list(DH[i])    # we need to make a copy
        params[3] += thetas[i]
        T = LinkTransform(params)
        if i == 0:
            result = T
        else:
            result = np.matmul(result, T)

        if show:
            print("T%d_%d = " % (i, i+1))
            print(T)
            print("T0_%d = " % (i+1))
            print(result)

    return result
```

In this function we feed the actual θ angles that are added to the values provided for θ from the DH paramters.

Now we can start with the actual IK calculation. We have split this into two parts:

- the calculation of the writs centre (WC) given a position for the EE in a request
- the calculation of the 6 θ angles from the WC information

Let's start first with the WC calculation:

```
def WCfromEE(req, x):
    # determines the positon of the WC from the end-effector's
    # orientation as provided in the req
    (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])

    # calculate Rrpy matrix as in course notes
    Rrpy = np.matmul(Rot_ZYX(yaw, pitch, roll), GripperCorrection())

    # extract vector n
    n = Rrpy[0:3,2].T

    # vector p from request
    p = np.array([req.poses[x].position.x,
                  req.poses[x].position.y,
                  req.poses[x].position.z])

    # arm constants from DH paramters
    dG = DH[6][2]
    l = 0.0             # small letter L; end-effector length

    # and calculate WC position
    wc = p - (dG + l)*n

    return wc, Rrpy
```

We use the method suggested in the course material:

- we first determine the roll, pitch and yaw angles of the end effctor by using the `tf.transformations.euler_from_quaternion` method where we pass the quaternion provided in the request
- we then calculate a Rrpy matrix (3x3) by obtaining the equivalent rotation matrix around Z, Y and X given the yaw, pitch and roll angles and by multiplying it with the correction matrix so that the Rrpy is reflecting the orientation of the end-effector according to our DH model
- as defined in the course we extract from the Rrpy matrix the last column to define the n vector that represents the orientation around the Z axis
- we then define the p vector from the x, y, z positions of the end effector as provided In the request data
- we use then the d_G parameters from the DH table; according to the formals in the course notes there is an additional l (small letter L) that represents the length of the end-effector, but because of the way the d_G parameter was built (to the end-effector centre) and because the information provided in the request is related to the end-effector centre we will simply consider this l = 0
- we then use numpy matrix elementwise multiplication to calculate the WC coordinates using the formula `wc = p - (dG + l)*n`
- we return the coordinates calculated for WC as well as Rrpy that we will need later for the calculation of the θ angles

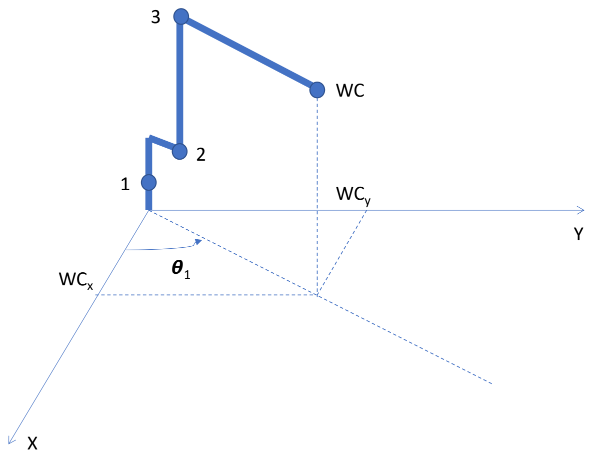
The second function that calculates the angles is:

```
def AnglesFromWC(wc, Rrpy):
```

I will present the content of this method step by step. First, to make things easier to read in code we read the parameters from the DH table into variables that are similar to the ones we used on our drawings in the first chapter related to the DH parameters:

```
    # extract elements from DH table to make things easier to understand
    a1 = DH[1][1]
    a2 = DH[2][1]
    a3 = DH[3][1]
    d1 = DH[0][2]
    d4 = DH[3][2]
    wcx = wc[0]
    wcy = wc[1]
    wcz = wc[2]
```

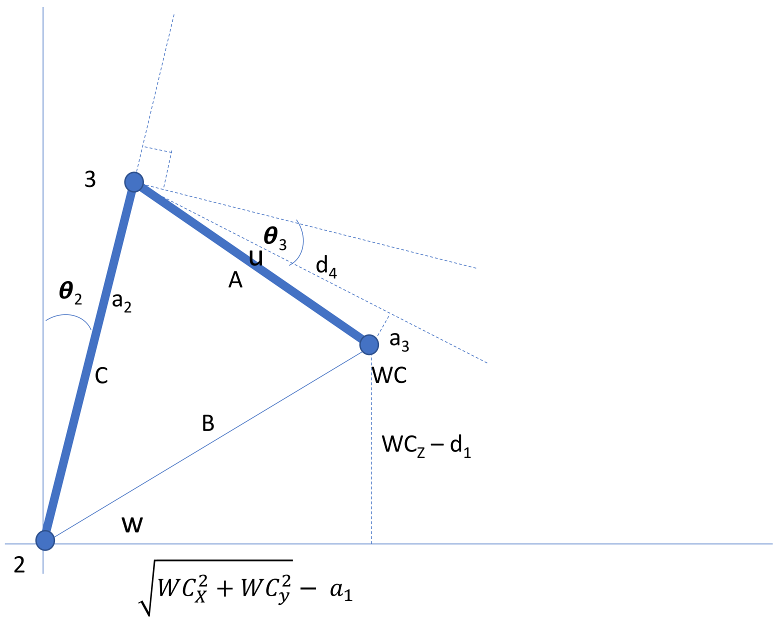
The calculation of θ_1 is easy and follows from the fact that the robot arm (up to the WC) is located in one single plan and the angle θ_1 results in the relation between WC_x and WC_y as you can see in the following image:



From this diagram it is easy to see that the θ_1 angle can be determined by applying the acrtan2 function using WC_y and WC_x:

```
# calculates th1, th2 and th3 from the position of WC
th1 = np.arctan2(wcy, wcx)
```

For the determination of the angles θ_2 and θ_3 we will use the following diagram that represents the robot arm in the plane that naturally produces between the joints 2, 3 and WC:



In the image above, we know certain elements from the robot structure and the coordinates of the WC point and we need to determine θ_2 and θ_3 angles. The main focus in the image above is the triangle in the middle described by the three sides A, B and C.

First, when the θ_3 is 0 the arm is pointing to the right, but the WC point is not on the same level with the joint 3 because of the small displacement a_3 downwards. In the image above, you can see that we can determine the length of side A by using the Pythagoras theorem:

```
# we use the notation from the course diagram
A = np.sqrt(a3**2 + d4**2)
```

Similarly, we can use Pythagoras for the lower triangle to determine the B side. For this triangle the horizontal is equal to the projection the WC into the XY plane (which is the square root of the sum of WC_x squared and WC_y squared) and a_1 as the picture above has the origin in joint 2 which is displaced by a1 along that projection of WC. The vertical side is simply $WC_Y - d_1$ as again the origin in joint 2 which is d_1 above the world origin. So:

```
wcxy = np.sqrt(wcx**2 + wcy**2)
B = np.sqrt( (wcxy - a1)**2 + ( wcz - d1)**2)
```

Finally, side C is simply a_2 as this is a rigid link.

```
C = a2
```

Knowing the 3 sides of the triangle we can apply the Cosine Law an determine the angles of the triangle. We start with the angle a (opposite side A):

```
cosa = (B**2 + C**2 - A**2) / (2*B*C)
a = np.arccos(cosa)
```

We can also calculate the angle w that sits under the triangle from the two sides:

```
w = np.arctan2(wcz - d1, wcxy - a1)
```

Which means we can simply now calculate θ_2:

```
th2 = np.pi/2 - w - a
```

For θ_3 we first calculate the angle b (opposite side B):

```
cosb = (A**2 + C**2 - B**2) / (2*A*C)
b = np.arccos(cosb)
```

Then the small angle u between the actual arm and "unbent" arm (if it would not be distorted by a_3):

```
u = np.arctan2(a3, d4)
```

And finally calculate the θ_3 by subtracting the previous 2 angles from 90°:

```
th3 = np.pi/2 - b + u  # u is actually negative because s3 < 0
```

Now that we know θ_1, θ_2 and θ_3 we apply the algorithm suggested in the course notes: first we calculate the combined transformation for the chain 1-2-3 only. We will use the help function from the forward kinematics and we provide only the first 3 records in the DH table and the three angles we already know:

```
T0_3 = ChainLinkTransform(DH[0:3], [th1, th2, th3])
```

We then extract the rotation matrix R0_3 from this and invert it. As explained in the course by multiplying this inverse with the Rrpy (the rotation matrix for the whole chain) we obtain the R3_6 representing the rotation matrix corresponding to the rotations produced by the θ_4, θ_5 and θ_6:

```
R0_3 = T0_3[0:3,0:3]
R0_3inv = np.linalg.inv(R0_3)
R3_6 = np.matmul(R0_3inv, Rrpy)
```

Knowing this matrix we can determine the angles θ_4, θ_5 and θ_6 in the following way: if we are writing the DH transformations in symbolic manner (using sympy in a Jupyter notebook)

```
In [1]: import sympy as sp
        from sympy.matrices import Matrix
        #import numpy as np

        # pretty print numpy matrices
        #np.set_printoptions(formatter={'float': '{: 0.5f}'.format})
```

```
In [2]: th1, th2, th3, th4, th5, th6 = sp.symbols('th1:7')

        # the DH paramters for KUKA robot as per written document
        # alpha, a, d, theta (adjustmet)
        # the angles we will use to control the robot will be added to the theta paramters
        DH = [[0       , 0     , 0.75 , th1           ],
              [-sp.pi/2, 0.35  , 0    , th2 - sp.pi/2],
              [0       , 1.25  , 0    , th3           ],
              [-sp.pi/2, -0.054, 1.5  , th4           ],
              [sp.pi/2 , 0     , 0    , th5           ],
              [-sp.pi/2, 0     , 0    , th6           ],
              [0       , 0     , 0.303, 0             ]]
```

```
In [3]: # a help function that builds a transformation matrix given the 4 DH paramters for that joint
        def LinkTransform(params):
            # params is a list of length 4: alpha, a, d, theta in this order
            alpha = params[0]
            a = params[1]
            d = params[2]
            theta = params[3]
            # returns a Matrix of transformation for the given DH paramters
            return Matrix([[sp.cos(theta)              , -sp.sin(theta)              , 0              , a],
                           [sp.sin(theta)*sp.cos(alpha), sp.cos(theta)*sp.cos(alpha), -sp.sin(alpha), -sp.sin(alpha)*d],
                           [sp.sin(theta)*sp.sin(alpha), sp.cos(theta)*sp.sin(alpha), sp.cos(alpha) , sp.cos(alpha)*d],
                           [0                          , 0                          , 0              , 1]])
```

```
In [11]: T34 = LinkTransform(DH[3])
         T45 = LinkTransform(DH[4])
         T56 = LinkTransform(DH[5])
         T36 = sp.simplify(T34 * T45 * T56)
         print(T36[0:3,0:3])
```

and apply the calculation for the (variables) θ_4, θ_5 and θ_6 and α (constants) coming from the DH table for each transformation we end up with the following equivalent symbolic version of the R3_6:

```
Matrix([
[-sin(th4)*sin(th6) + cos(th4)*cos(th5)*cos(th6), -sin(th4)*cos(th6) - sin(th6)*
cos(th4)*cos(th5), -sin(th5)*cos(th4)],
[                           sin(th5)*cos(th6),                                 -
sin(th5)*sin(th6),              cos(th5)],
[-sin(th4)*cos(th5)*cos(th6) - sin(th6)*cos(th4),  sin(th4)*sin(th6)*cos(th5) -
cos(th4)*cos(th6),  sin(th4)*sin(th5)]])
```

The format is a little ugly so, I have replaced the items in the matrix that are complicated and not very useful for our calculations with XXXXXX and left only the terms that are useful for our calculation:

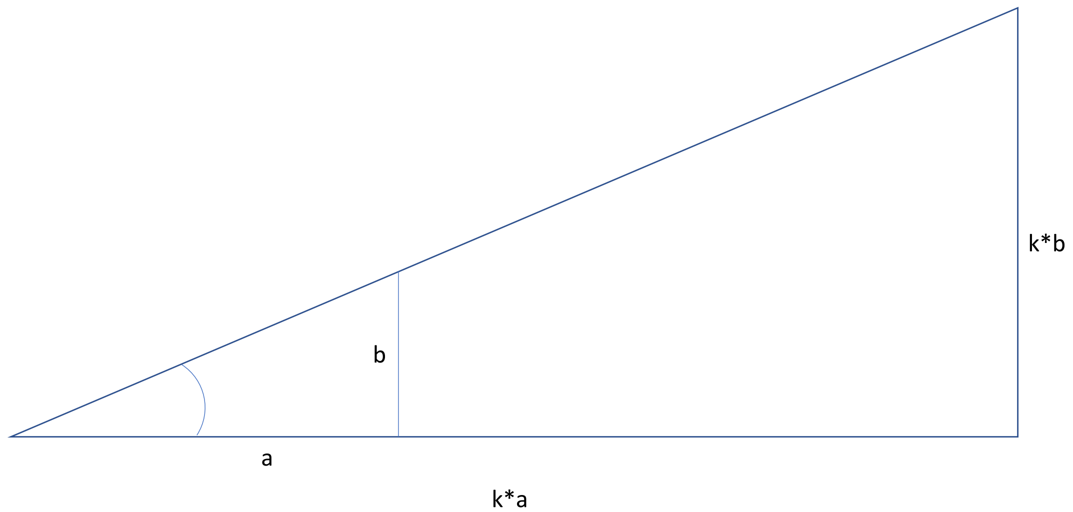
```
 Matrix([
[XXXXXX              , XXXXXX              , -sin(th5)*cos(th4)],
[ sin(th5)*cos(th6), -sin(th5)*sin(th6),            cos(th5)],
[XXXXXX              , XXXXXX              ,  sin(th4)*sin(th5)]])
```

We see there is a cos term at R3_6[1][2] and theoretically we could use an arccos function to determine the angle – but this is not very accurate as arccos may not return the correct angle. We'd rather use the arctan2 but for this we need a sin and a cos value. We can produce a sin by using the R3_6[0][2] and R3_6[2][2] if we square the two terms, add them (the th4 term will become 1 as $\sin^2 + \cos^2 = 1$) and then take the square root of that. We can then use those two terms to calculate the θ_5 through the arctan2 fucntion:

```
th5 = np.arctan2(np.sqrt(R3_6[0][2]**2+R3_6[2][2]**2), R3_6[1][2])
```

We can determine the other two angles by noticing 2 important things:

1. arctan2(k*sin(a), k*cos(a)) provides a correct answer for angle a no matter what the k is (as long as it not 0); this is as if k would simplify from the calculation of arctan2 and would leave only the other terms. This is easy to see in a drawing where we have a triangle that is scaled by k:



2. arctan2(1,1) <> arctan2(-1, -1); in other words we need to be careful when we "simplify" with k in the case above if it is a positive or a negative number

In our case for the calculation of th4 we would need to use the R3_6[2][2] and R3_6[0][2] but one has positive sign and the other a negative sign. So, when calling the arctan2 function one of the parameters will need to have the sign changed. But which one? Because if we choose for instance to change the sign for R3_6[0][2] - which would seem natural as that has a negative sign in front in the matrix, but the sin(th5) is negative then both items are passed with the changed signs to the function and we will get a wrong value for the th4. It is therefore important to check for the value of sin(th5) and to change the signs for those parameters that would otherwise be negative when calling arctan2:

```
if np.sin(th5) < 0:
    th4 = np.arctan2(-R3_6[2][2], R3_6[0][2])
    th6 = np.arctan2(R3_6[1][1], -R3_6[1][0])
else:
    th4 = np.arctan2(R3_6[2][2], -R3_6[0][2])
    th6 = np.arctan2(-R3_6[1][1], R3_6[1][0])
```

Finally, we return the angles:

```
return th1, th2, th3, th4, th5, th6
```

And this concludes the IK calculations.

Debugging the IK

We have used the IK_debug program to check the calculations. The following updates were made to the program to invoke the help functions from the IK_support:

```
########################################################################################

# determine WC
wc, Rrpy = WCfromEE(req, x)

# calculate first 3 thetas
theta1, theta2, theta3, theta4, theta5, theta6 = AnglesFromWC(wc, Rrpy)

# forward kinematics
Tforw = ChainLinkTransform(DH, [theta1, theta2, theta3, theta4, theta5, theta6, 0])
# we are only interested in the position of the EE so we'll not do the GripperCorrection

your_wc = wc            # <--- Load your calculated WC values in this array
your_ee = Tforw[0:3,3]  # <--- Load your calculated end effector value from your forward
kinematics
########################################################################################
```

Since the heavy lifting is done in the functions we simply:

- determine the position of WC
- determine the theta angles
- calculate the forward kinematics for checking
- pass the WC to the rest of the program
- pass the EE position for the rest of the program

Running it for the 3 examples we have in the debug we get.

<u>For test 1:</u>

```
Total run time to calculate joint angles from pose is 0.0007 seconds
```

```
Wrist error for x position is: 0.00000046
Wrist error for y position is: 0.00000032
Wrist error for z position is: 0.00000545
Overall wrist offset is: 0.00000548 units

Theta 1 error is: 0.00093770
Theta 2 error is: 0.00178633
Theta 3 error is: 0.00206506
Theta 4 error is: 0.00172809
Theta 5 error is: 0.00198404
Theta 6 error is: 0.00252923

**These theta errors may not be a correct representation of your code, due to the
fact
that the arm can have muliple positions. It is best to add your forward kinmeatics to
confirm whether your code is working or not**


End effector error for x position is: 0.00000000
End effector error for y position is: 0.00000000
End effector error for z position is: 0.00000000
Overall end effector offset is: 0.00000000 units
```

Because we're using numpy instead of sympy the execution is extremely fast (0.7ms) and you can see the errors are only due to the representation of the data in the test. The forward kinematics is matching to all 8 decimals the result.

<u>For Test 2:</u>

```
Total run time to calculate joint angles from pose is 0.0006 seconds

Wrist error for x position is: 0.00002426
Wrist error for y position is: 0.00000562
Wrist error for z position is: 0.00006521
Overall wrist offset is: 0.00006980 units

Theta 1 error is: 3.14309971
Theta 2 error is: 0.27927962
Theta 3 error is: 1.86833314
Theta 4 error is: 3.08639539
Theta 5 error is: 0.06340277
Theta 6 error is: 6.13524929

**These theta errors may not be a correct representation of your code, due to the
fact
that the arm can have muliple positions. It is best to add your forward kinmeatics to
confirm whether your code is working or not**


End effector error for x position is: 0.00000000
End effector error for y position is: 0.00000000
End effector error for z position is: 0.00000000
Overall end effector offset is: 0.00000000 units
```

For this test it seems like the IK has suggested a position that starts with theta 1 being completely opposite from the position that was suggested in the test. As a result the other angles have different solutions. The forward kinematics though confirms that the solution produced is valid and there are no differences in the position of the end-effector.

For Test 3:

```
Total run time to calculate joint angles from pose is 0.0006 seconds

Wrist error for x position is: 0.00000503
Wrist error for y position is: 0.00000512
Wrist error for z position is: 0.00000585
Overall wrist offset is: 0.00000926 units

Theta 1 error is: 0.00136747
Theta 2 error is: 0.00329800
Theta 3 error is: 0.00339863
Theta 4 error is: 6.28213720
Theta 5 error is: 0.00287049
Theta 6 error is: 6.28227458

**These theta errors may not be a correct representation of your code, due to the
fact
that the arm can have muliple positions. It is best to add your forward kinmeatics to
confirm whether your code is working or not**


End effector error for x position is: 0.00000000
End effector error for y position is: 0.00000000
End effector error for z position is: 0.00000000
Overall end effector offset is: 0.00000000 units
```

For this it seems that we have the same numbers, just that the code that calculates the errors for theta is only determining as positive errors and in our case they are very, very close to 2* pi. For instance the error for theta 4 can also be written as -0.0010481. As in the previous 2 cases the forward kinematics confirms the correctness of the solution.

IK_server

Implementing the IK in the server code is very simple. After importing IK_support at the beginning of the file the method that does the calculation only includes 2 lines of code that call the calculation routines. I have also commented the extraction of data from the request (as this is now performed in the WCfromEE method):

```
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:

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

            #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])

            wc, Rrpy = WCfromEE(req, x)
            theta1, theta2, theta3, theta4, theta5, theta6 = AnglesFromWC(wc, Rrpy)

              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)
```

The recording with the processing in Gazebo is avaialable on Youtube:

https://youtu.be/0pJCZpW8gsk