COMP0127 Robotics System Engineering Coursework 3

Douglas Chiang (15055142)
hou.chiang.20@ucl.ac.uk

22^{nd} Jan, 2021

Question 1

a) Four. X,Y for moving into the correct location, Z and rotation θ for picking up and drop it into the basket.

b) Serial design like SCARA because it is fast so it can catch up the conveyor speed. Vacuum gripper is chosen candy is fragile.

c) Since this is a repetitive task, AC servo motor is a choice. Gears for transmission.

d) Encoder and tachometer for driving the manipulator to give the position and the velocity. Cameras for determining the correct object.

e) The joints, if the joints are not properly lubricated, it will wear because of friction. It need to be frequently lubricated because the heat generated can cause the lubricant to lost its function.

f) Add a weight sensor at the end effector to sense the weight that the end effector has picked up. If the weight is larger than the thershold, proceed to put it in the basket. If not, put it back.

Question 2

Forward kinematic at the centre of mass

The forward kinematics at the link center of mass, compared to the normal forward kinematics, instead of calculating ${}^0T_i$ for the i^{th} frame, we now calculate ${}^0T_{G_i} = {}^0\ T_{i-1}Rot(\theta_i)Trans(G_i)$, where G_i is the center of mass location of the i^{th} link relative to the $i-1^{th}joint$. The code that do this is as the following. Please note that the input frame number for this function is from 1 - 8:

```
def forward_kine_cm(self, joint, frame):
    ##TODO: Fill in this function to complete Q2.
    ## "joint" is a numpy array of double consisting of the joint value.
    ## "frame" is an integer indicating the frame you wish to calculate.
    ## The output is a numpy 4*4 matrix describing the transformation from the 'iiwa_link_0' frame to the centre of mass of the specified link.
    transform = TransformStamped()

    prev_frame       = frame - 1
    T_0_prev         = self.forward_kine(joint, prev_frame)
    Rot_theta_i      = self.T_rotationZ(joint[prev_frame] + self.DH_params[prev_frame, 3])
    Trans_G_i        = np.eye(4)
    Trans_G_i[0:3,3] = self.link_COM_pos[prev_frame,:].reshape(3,)
    T                = T_0_prev.dot(Rot_theta_i).dot(Trans_G_i)
    self.broadcast_joint(transform, T, frame)
    return T

def forward_kine(self, joint, frame):
    T = np.identity(4)
    ##Add offset from the iiwa platform.
    T[2, 3] = 0.1575
    ##TODO: Fill in this function to complete Q2.

    transform = TransformStamped()
    for i in range(0, frame):
        A = self.dh_matrix_standard(self.DH_params[i][0], self.DH_params[i][1], self.DH_params[i][2], joint[i] + self.DH_params[i][3])
        T = T.dot(A)
        self.broadcast_joint(transform, T, i+11)
    return T
```

Jacobian at the centre of mass

The jacobian at the center of mass of the link, in contrast to the normal one, we will make use of the forward kinematics at the links' center of mass to calculate the actual position of the links' center of mass $\mathbf{p}_{li}$. Then we will make use of $\mathbf{p}_{li}$ to calculate the jacobian $\mathbf{J}$ as follows:

$$\mathbf{J}_{\mathbf{P}_j}^{(l_i)} = \mathbf{z}_{j-1} \times (\mathbf{p}_{li} - \mathbf{p}_{j-1})$$
$$\mathbf{J}_{\mathbf{O}_j}^{(l_i)} = \mathbf{z}_{j-1}$$

```
def get_jacobian_cm(self, joint, frame):
    ##TODO: Fill in this function to complete Q2.
    ## "joint" is a numpy array of double consisting of the joint
       value.
    ## "frame" is an integer indicating the frame you wish to
       calculate.
    ## The output is a numpy 6*7 matrix describing the Jacobian matrix
        defining at the centre of mass of the specified link.

    J      = np.zeros((6,7)) # Initialize Jacobian
    T_All  = []              # Memory for transformation matrices
    T_0_Gi = self.forward_kine_cm(joint, frame)
    p_li   = T_0_Gi[0:3,3]

    # Forward KE to calculate Transformation matrices
    for frame_idx in range(0,frame):
        T_All.append(self.forward_kine(joint, frame_idx))

    for i in range(0,frame):
        # Select i th transformation matrix:
        Tr      = T_All[i]
        z_previ = Tr[0:3,2]
        p_previ = Tr[0:3,3]

        # J_Pi:
        J[0:3,i] = np.cross(z_previ, (p_li - p_previ))

        # J_Oi:
        J[3:6,i] = z_previ
    #print(J)
    #print(frame)
    return J
```

Iterative inverse kinematic

To compute the inverse kinematics iteratively, we do the followings:

1. Initialize looping parameter k = 0 and convert the 4×4 transformation matrix to the 6×1 vector $\mathbf{x}_e^*$
2. Then repeatly perform the followings until the solution converges:
   (a) Calculate the forward kinematics for joint positions $\mathbf{q}(k)$ to get the transformation matrices $(\mathbf{q}(k))$
   (b) Compute the Jacobian matrix $\mathbf{J}(k)$
   (c) Compute the pose $\mathbf{x}_e((\mathbf{q}(k)))$
   (d) Calculate the joint positions for the $k + 1^{th}$ loop:

$$\mathbf{q}(k+1) = \mathbf{q}(k) + \alpha \mathbf{J}(k)^T (\mathbf{x}_e^* - \mathbf{x}_e((\mathbf{q}(k))))$$

   where $\mathbf{x}_e^*$ is the desired pose
   (e) Update the looping parameter $k = k + 1$
3. Return $\mathbf{q}(k+1)$

The convergence criteria I used is to leverage the distance between the desired pose $\mathbf{p}_e^*$ and the current pose $\mathbf{p}_e$ such that the distance is smaller than a smaller value ε:

$$\|\mathbf{x}_e^* - \mathbf{x}_e\| < \varepsilon$$

The code that do the above algorithm is as follows:

```
def inverse_kine_ite(self, desired_pose, current_joint):
    ##TODO: Fill in this function to complete Q2.
    ## "desired_pose" is a numpy 4*4 matrix describing the
       transformation of the manipulator.
    ## "current_joint" is an array of double consisting of the joint
       value (works as a starting point for the optimisation).
    ## The output is numpy vector containing an optimised joint
       configuration for the desired pose.

    # Initialize:
    k           = 0
    T_e         = np.identity(4)
    x_e         = np.zeros((6,1))
    x_e_star    = np.zeros((6,1))
    error       = 100000
    alpha       = 0.1               # Converge rate
    epsilon     = 0.001             # Error thershold

    # Convert desired_pose (4x4 np matrix) to a 6x1 vector (x_e^*):
    # Pose at step k
    p_e_desired = desired_pose[0:3,3]
    R_e_desired = desired_pose[0:3,0:3]
```

```
20
21      # Convert rotation matrix to rodrigues for x_e_star:
22      [rex_desired, rey_desired, rez_desired] = self.
           convert_quat2rodrigues(self.rotmat2q(R_e_desired))
23
24      # x_e_star:
25      x_e_star[0:3] = p_e_desired.reshape(3,1)
26      x_e_star[3]   = rex_desired
27      x_e_star[4]   = rey_desired
28      x_e_star[5]   = rez_desired
29
30      while error > epsilon:
31          # Forward KE for T(q(k))
32          T_e = self.forward_kine(current_joint, 7)
33
34          # Jacobian for q(k)
35          J = self.get_jacobian(current_joint)
36
37          # Pose at step k
38          p_e = T_e[0:3,3]
39          R_e = T_e[0:3,0:3]
40
41          # Convert rotation matrix to rodrigues for x_e:
42          [rex, rey, rez] = self.convert_quat2rodrigues(self.rotmat2q(
               R_e))
43
44          # Current x_e
45          x_e[0:3] = p_e.reshape(3,1)
46          x_e[3]   = rex
47          x_e[4]   = rey
48          x_e[5]   = rez
49
50          # x_e_star = desired_pose
51
52          # Pose information at q_k+1 (q_kp1)
53          current_joint = current_joint + np.squeeze(alpha*np.transpose(
               J).dot((x_e_star - x_e)))
54          # Check error
55          error = np.linalg.norm(x_e_star - x_e)
56
57          # Debug
58          print("Error: {}\r".format(error))
59
60          # Next step
61          k = k + 1
62      print("Done in iteration: {}".format(k))
63      return current_joint
```

Closed-form inverse kinematic

To derive the closed-form inverse kinematics solutions for the IIWA (イイヴァ) robot, since this time we have 7 joints, solving the close form inverse kinematic using the normal way will be super challenging. Therefore, the technique name parameterization of redundancy is used to solve the problem [1].

To do this we first need to introduce a redundancy parameter ψ which is defined as the angle between the arm plane, spanned by the shoulder, elbow, wrist and the reference plane, as shown in Figure 1:

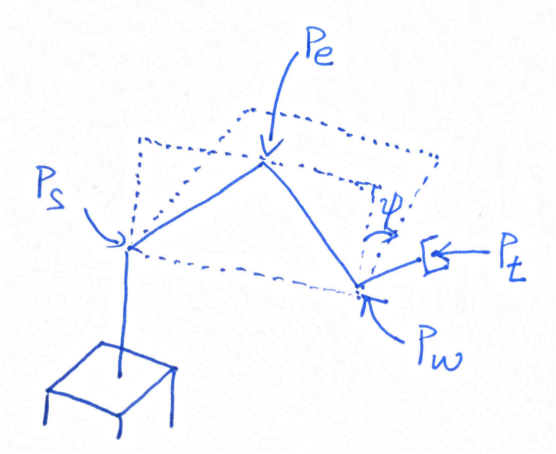


Figure 1: Definition of angle ψ

where the subscript s, e, w, t means shoulder, elbow, wrist and tip (end effector) respectively. In our case, our shoulder is at joint 2, the elbow is at joint 4 and the wrist is at joint 6 as shown in Figure 2:

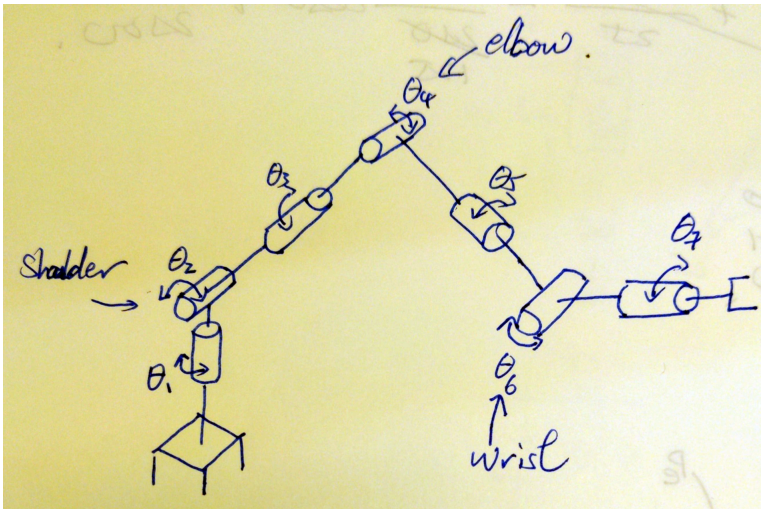


Figure 2: Joint orientations of IIWA

It is given that the robot have the DH parameters shown in Table 1

Table 1: DH Parameters of IIWA

i	a_i	α_i	d_i	θ_i
1	0	$-\pi/2$	$d_{bs} = 0.2025$	θ_1
2	0	$\pi/2$	0	θ_2
3	0	$\pi/2$	$d_{se} = 0.42$	θ_3
4	0	$-\pi/2$	0	θ_4
5	0	$-\pi/2$	$d_{ew} = 0.4$	θ_5
6	0	$\pi/2$	0	θ_6
7	0	0	$d_{wt} = 0.126$	θ_7

Then we further define the following four vectors:

$$^0\mathbf{l}_{bs} = \begin{bmatrix} 0 & 0 & d_{bs} \end{bmatrix}^T \tag{1}$$

$$^3\mathbf{l}_{se} = \begin{bmatrix} 0 & -d_{se} & 0 \end{bmatrix}^T \tag{2}$$

$$^4\mathbf{l}_{ew} = \begin{bmatrix} 0 & 0 & d_{ew} \end{bmatrix}^T \tag{3}$$

$$^7\mathbf{l}_{wt} = \begin{bmatrix} 0 & 0 & d_{wt} \end{bmatrix}^T \tag{4}$$

With these settings, we can first look into the forward kinematics analysis for the end effector position $^0\mathbf{x}_7$ and orientation $^0\mathbf{R}_7$ with the help of (1), (2), (3) and (4):

$$^0\mathbf{x}_7 = {}^0\mathbf{l}_{bs} + {}^0\mathbf{R}_3[{}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4({}^4\mathbf{l}_{ew} + {}^4\mathbf{R}_7\ {}^7\mathbf{l}_{wt})] \tag{5}$$

$$^0\mathbf{R}_7 = {}^0\mathbf{R}_3\ {}^3\mathbf{R}_4\ {}^4\mathbf{R}_7 \tag{6}$$

Since the elbow can freely rotate around the axis connecting the shoulder (joint 2) and wrist (joint 6), if the end effector does not move, then the axis mentioned will remain stationary. We can then derive such axis $^0\mathbf{x}_{sw}$:

$$^0\mathbf{x}_{sw} = {}^0\mathbf{x}_7 - {}^0\mathbf{l}_{bs} - {}^0\mathbf{R}_7\ {}^7\mathbf{l}_{wt} \tag{7}$$

$$= {}^0\mathbf{R}_3({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}) \tag{8}$$

such that all the variable in (7) are constant. This can also be verified by calculating the norm of (8):

$$\left\|{}^0\mathbf{x}_{sw}\right\| = \left\|{}^0\mathbf{R}_3({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew})\right\|$$

$$= \left\|{}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}\right\| \tag{9}$$

Here, $^3\mathbf{R}_4$ is the rotation matrix of the elbow joint angle. Since we have the end effector fixed, this matrix is constant.
Now, consider the elbow move ψ degree around the axis $^0\mathbf{x}_{sw}$, the orientational change resulted is:

$$^0\mathbf{R}_\psi = \mathbf{I} + sin\psi \left[{}^0\mathbf{u}_{sw}\times\right] + (1 - cos\psi) \left[{}^0\mathbf{u}_{sw}\times\right]^2 \tag{10}$$

where $\mathbf{I}$ is a 3×3 identity matrix, $^0\mathbf{u}_{sw}$ is the unit vector of $^0\mathbf{x}_{sw}$, and $[^0\mathbf{u}_{sw}\times]$ is the skew-symmetric matrix of $^0\mathbf{u}_{sw}$. With this, the wrist orientation from the perspective of the base coordinate system is:

$$^0\mathbf{R}_4 = {}^0\ \mathbf{R}_\psi\ {}^0\mathbf{R}_4^o \tag{11}$$

where ${}^4\mathbf{R}_4^o$ is the wrist orientation when the arm plane touches the reference plane. If we combine the result we derived from (9) with the above, we also have ${}^3\mathbf{R}_4 = {}^3\mathbf{R}_4^o$. With this, the relation in (11) becomes:

$$\begin{aligned} {}^0\mathbf{R}_4\ {}^3\mathbf{R}_4 &= {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o\ {}^3\mathbf{R}_4^o \\ {}^0\mathbf{R}_3 &= {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o \end{aligned} \tag{12}$$

Applying the result from (12) to the forward kinematics, we now have (5) and (6) as follows:

$${}^0\mathbf{x}_7 = {}^0\mathbf{l}_{bs} + {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o[{}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4({}^4\mathbf{l}_{ew} + {}^4\mathbf{R}_7\ {}^7\mathbf{l}_{wt})] \tag{13}$$

$${}^0\mathbf{R}_7 = {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o\ {}^3\mathbf{R}_4\ {}^4\mathbf{R}_7 \tag{14}$$

With all the above, we are now ready to work out the joint angles.

Suppose we have the desired end effector position ${}^0\mathbf{x}_7^d$ and orientation ${}^0\mathbf{R}_7^d$, since the wrist position is fixed the elbow joint angle θ_4 can be calculated first from (9):

$$\begin{aligned} \left\|{}^0\mathbf{x}_{sw}\right\| &= \left\|{}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}\right\| \\ \left\|{}^0\mathbf{x}_{sw}\right\| &= \left\| \begin{bmatrix} -d_{ew}sin\theta_4 \\ d_{ew}cos\theta_4 - d_{se} \\ 0 \end{bmatrix} \right\| \\ \left\|{}^0\mathbf{x}_{sw}\right\|^2 &= d_{ew}sin^2\theta_4 + [d_{se} - d_{ew}cos\theta_4]^2 \\ \theta_4 &= cos^{-1}\left[\frac{\left\|{}^0\mathbf{x}_{sw}\right\|^2 - d_{se}^2 - d_{ew}^2}{2d_{se}d_{ew}}\right] \\ \theta_4 &= cos^{-1}\left[\frac{\left\|{}^0\mathbf{x}_{sw}\right\|^2 - 0.42^2 - 0.4^2}{2(0.42)(0.4)}\right] \\ \theta_4 &= cos^{-1}\left[\frac{\left\|{}^0\mathbf{x}_{sw}\right\|^2 - 0.3364}{0.336}\right] \end{aligned} \tag{15}$$

where ${}^0\mathbf{x}_{sw} = {}^0\mathbf{x}_7^d - {}^0\mathbf{l}_{bs} - {}^0\mathbf{R}_7^d\ {}^7\mathbf{l}_{wt}$ is a constant vector since all the elements are known.

With θ_4 calculated, we can now proceed to calculate the shoulder joints. As we can see in Figure 1, the shoulder joint angles depends on the arm angle ψ, we need to know the reference joint angles θ_1^o, θ_2^o when the arm angle is zero. Here we need to first fixed the third joint angle $\theta_3 = \theta_3^o = 0$, which means the condition in (8) is satisfied:

$$\begin{aligned} {}^0\mathbf{x}_{sw} &= {}^0\mathbf{R}_3({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}) \\ &= {}^0\mathbf{R}_1^o\ {}^1\mathbf{R}_2^o\ {}^2\mathbf{R}_3|_{\theta_3=0}({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}) \\ &= {}^0\mathbf{R}_1^o\ {}^1\mathbf{R}_2^o\ {}^2\mathbf{R}_3^o|_{\theta_3^o=0}({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}) \end{aligned} \tag{16}$$

In equation (16), the only unknown that we have are θ_1^o, θ_2^o in ${}^0\mathbf{R}_1^o, {}^1\mathbf{R}_2^o$ respectively. Continuing,

we have:

$$
\begin{aligned}
{}^0\mathbf{x}_{sw} &= {}^0\mathbf{R}_1^o\ {}^1\mathbf{R}_2^o\ {}^2\mathbf{R}_3^o|_{\theta_3^o=0}({}^3\mathbf{l}_{se} + {}^3\mathbf{R}_4\ {}^4\mathbf{l}_{ew}) \\
&= \begin{bmatrix} cos\theta_1^o & 0 & -sin\theta_1^o \\ sin\theta_1^o & 0 & cos\theta_1^o \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} cos\theta_2^o & 0 & sin\theta_2^o \\ sin\theta_2^o & 0 & -cos\theta_2^o \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -d_{ew}sin\theta_4 \\ d_{ew}cos\theta_4 - d_{se} \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} cos\theta_1^o cos\theta_2^o & cos\theta_1^o sin\theta_2^o & sin\theta_1^o \\ cos\theta_2^o sin\theta_1^o & sin\theta_1^o sin\theta_2^o & -cos\theta_1^o \\ -sin\theta_2^o & cos\theta_2^o & 0 \end{bmatrix} \begin{bmatrix} -d_{ew}sin\theta_4 \\ d_{ew}cos\theta_4 - d_{se} \\ 0 \end{bmatrix} \\
{}^0\mathbf{x}_{sw} &= \begin{bmatrix} -cos\theta_1^o sin\theta_2^o(d_{se} - d_{ew}cos\theta_4) - d_{ew}cos\theta_1^o cos\theta_2^o sin\theta_4 \\ -sin\theta_1^o sin\theta_2^o(d_{se} - d_{ew}cos\theta_4) - d_{ew}cos\theta_2^o sin\theta_1^o sin\theta_4 \\ d_{ew}sin\theta_2^o sin\theta_4 - cos\theta_2^o(d_{se} - d_{ew}cos\theta_4) \end{bmatrix} \qquad (17)
\end{aligned}
$$

In (17) we can then find θ_1^o by working out $\theta_1^o = atan2({}^0\mathbf{x}_{sw}^x, {}^0\mathbf{x}_{sw}^y)$. For θ_2^o, since ${}^0\mathbf{x}_{sw} = {}^0\mathbf{x}_7^d - {}^0\mathbf{l}_{bs} - {}^0\mathbf{R}_7^d\ {}^7\mathbf{l}_{wt}$, we can match the z component ${}^0\mathbf{x}_{sw}^z$ with (17). So we know all the reference joint angles $\theta_1^o, \theta_2^o, \theta_3^o$.

After this, we can then substitute (10) into (12), which lead us to:

$$
\begin{aligned}
{}^0\mathbf{R}_3 &= {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o \\
&= \left[\mathbf{I} + sin\psi\left[{}^0\mathbf{u}_{sw}\times\right] + (1 - cos\psi)\left[{}^0\mathbf{u}_{sw}\times\right]^2\right]\ {}^0\mathbf{R}_3^o \\
{}^0\mathbf{R}_3 &= sin\psi\left[{}^0\mathbf{u}_{sw}\times\right]\ {}^0\mathbf{R}_3^o - cos\psi\left[{}^0\mathbf{u}_{sw}\times\right]^2\ {}^0\mathbf{R}_3^o + \left[{}^0\mathbf{u}_{sw}\ {}^0\mathbf{u}_{sw}^T\right]\ {}^0\mathbf{R}_3^o \qquad (18)
\end{aligned}
$$

In (18), we have:

$$
{}^0\mathbf{R}_3 = \begin{bmatrix} cos\theta_1 cos\theta_2 cos\theta_3 - sin\theta_1 sin\theta_3 & cos\theta_1 sin\theta_2 & cos\theta_3 sin\theta_1 + cos\theta_1 cos\theta_2 sin\theta_3 \\ cos\theta_1 sin\theta_3 + cos\theta_2 cos\theta_3 sin\theta_1 & sin\theta_1 sin\theta_2 & cos\theta_2 sin\theta_1 sin\theta_3 - cos\theta_1 cos\theta_3 \\ -cos\theta_3 sin\theta_2 & cos\theta_2 & -sin\theta_2 sin\theta_3 \end{bmatrix} \qquad (19)
$$

$$
{}^0\mathbf{u}_{sw} = \frac{{}^0\mathbf{x}_{sw}}{\|{}^0\mathbf{x}_{sw}\|} \qquad (20)
$$

$$
\left[{}^0\mathbf{u}_{sw}\times\right] = \begin{bmatrix} 0 & -{}^0\mathbf{u}_{sw}^z & {}^0\mathbf{u}_{sw}^y \\ {}^0\mathbf{u}_{sw}^z & 0 & -{}^0\mathbf{u}_{sw}^x \\ -{}^0\mathbf{u}_{sw}^y & {}^0\mathbf{u}_{sw}^x & 0 \end{bmatrix} \qquad (21)
$$

From the above, we can see that (20) and (21) are known, so we can calculate the right and side of (18) and compare with the entries in (19).

Now in (18) we define $\mathbf{A}_s, \mathbf{B}_s, \mathbf{C}_s$ for simplicity, we get:

$$
\begin{aligned}
{}^0\mathbf{R}_3 &= sin\psi\left[{}^0\mathbf{u}_{sw}\times\right]\ {}^0\mathbf{R}_3^o - cos\psi\left[{}^0\mathbf{u}_{sw}\times\right]^2\ {}^0\mathbf{R}_3^o + \left[{}^0\mathbf{u}_{sw}\ {}^0\mathbf{u}_{sw}^T\right]\ {}^0\mathbf{R}_3^o \\
{}^0\mathbf{R}_3 &= \mathbf{A}_s sin\psi + \mathbf{B}_s cos\psi + \mathbf{C}_s \\
\begin{bmatrix} * & cos\theta_1 sin\theta_2 & * \\ * & sin\theta_1 sin\theta_2 & * \\ -cos\theta_3 sin\theta_2 & cos\theta_2 & -sin\theta_2 sin\theta_3 \end{bmatrix} &= sin\psi \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\
&\quad + cos\psi \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \qquad (22)
\end{aligned}
$$

Then by matching the entries in 22, we can derive θ_1 from entries $(1,2)$ and $(2,2)$, θ_2 from $(3,1)$ and θ_3 from $(3,1)$ and $(3,3)$, we get:

$$\theta_1 = atan2(-a_{22}sin\psi - b_{22}cos\psi - c_{22}, -a_{12}sin\psi - b_{12}cos\psi - c_{12}) \tag{23}$$
$$\theta_2 = \pm cos^{-1}(-a_{32}sin\psi - b_{32}cos\psi - c_{32}) \tag{24}$$
$$\theta_3 = atan2(a_{33}sin\psi + b_{33}cos\psi + c_{33}, -a_{31}sin\psi - b_{31}cos\psi - c_{31}) \tag{25}$$

Then for the wrist part, we substitite (10) into (14) we have:

$$\begin{aligned}
{}^0\mathbf{R}_7 &= {}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o\ {}^3\mathbf{R}_4\ {}^4\mathbf{R}_7 \\
({}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o\ {}^3\mathbf{R}_4)^{-1}\ {}^0\mathbf{R}_7^d &= {}^4\mathbf{R}_7 \\
{}^4\mathbf{R}_7 &= ({}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o\ {}^3\mathbf{R}_4)^{-1}\ {}^0\mathbf{R}_7^d \\
&= {}^3\mathbf{R}_4^{-1}({}^0\mathbf{R}_\psi\ {}^0\mathbf{R}_3^o)^{-1}\ {}^0\mathbf{R}_7^d \\
&= {}^3\mathbf{R}_4^T\ {}^0\mathbf{R}_3^T\ {}^0\mathbf{R}_7^d \\
&= {}^3\mathbf{R}_4^T\ (\mathbf{A}_s sin\psi + \mathbf{B}_s cos\psi + \mathbf{C}_s)^T\ {}^0\mathbf{R}_7^d \\
&= {}^3\mathbf{R}_4^T\mathbf{A}_s^T\ {}^0\mathbf{R}_7^d sin\psi + {}^3\mathbf{R}_4^T\mathbf{B}_s^T\ {}^0\mathbf{R}_7^d cos\psi + {}^3\mathbf{R}_4^T\mathbf{C}_s^T\ {}^0\mathbf{R}_7^d \qquad (26) \\
{}^4\mathbf{R}_7 &= \mathbf{A}_w sin\psi + \mathbf{B}_w cos\psi + \mathbf{C}_w \qquad (27)
\end{aligned}$$

In (26) we can see that all the elements on the right hand side are known, so if we compute ${}^4\mathbf{R}_7$ we can compare the entries and find $\theta_5, \theta_6, \theta_7$ using (27), as follows:

$$\begin{aligned}
{}^4\mathbf{R}_7 &= \mathbf{A}_w sin\psi + \mathbf{B}_w cos\psi + \mathbf{C}_w \\
\begin{bmatrix} * & * & cos\theta_5 sin\theta_6 \\ * & * & sin\theta_5 sin\theta_6 \\ -cos\theta_7 sin\theta_6 & sin\theta_6 sin\theta_7 & cos\theta_6 \end{bmatrix} &= \begin{bmatrix} a_{w11} & a_{w12} & a_{w13} \\ a_{w21} & a_{w22} & a_{w23} \\ a_{w31} & a_{w32} & a_{w33} \end{bmatrix} sin\psi \\
&\quad + \begin{bmatrix} b_{w11} & b_{w12} & b_{w13} \\ b_{w21} & b_{w22} & b_{w23} \\ b_{w31} & b_{w32} & b_{w33} \end{bmatrix} cos\psi + \begin{bmatrix} c_{w11} & c_{w12} & c_{w13} \\ c_{w21} & c_{w22} & c_{w23} \\ c_{w31} & c_{w32} & c_{w33} \end{bmatrix}
\end{aligned} \tag{28}$$

In (28), we can then make use of entries $(1,3)$ and $(2,3)$ to calculate θ_5, $(3,3)$ to calculate θ_6, and $(3,1)$ and $(3,2)$ to calculate θ_7, which gives the followings:

$$\theta_5 = atan2(a_{w23}sin\psi + b_{w23}cos\psi + c_{w23}, a_{w13}sin\psi + b_{w13}cos\psi + c_{w13}) \tag{29}$$
$$\theta_6 = \pm cos^{-1}(a_{w33}sin\psi + b_{w33}cos\psi + c_{w33}) \tag{30}$$
$$\theta_7 = atan2(a_{w32}sin\psi + b_{w32}cos\psi + c_{w32}, -a_{w31}sin\psi - b_{w31}cos\psi - c_{w31}) \tag{31}$$

Dynamic component $\mathbf{B}(\mathbf{q})$

The following steps are used to calculate the dynamic component matrix $\mathbf{B}(\mathbf{q})$:

1. Initialize a 6×7 matrix with zeros because there are 7 frames.
2. For every frame:
   (a) Calculate the transformation matrix of the i^{th} center of mass frame and take the rotation matrix ${}^0\mathbf{R}_{G_i}$

(b) Form the inertia matrix $\mathbf{I}_{l_i}^{O_i||}$ for the i^{th} frame

(c) Calculate the jacobian $\mathbf{J}$ using the center of mass version and then split the jacobian into $\mathbf{J}_P^{li}$ and $\mathbf{J}_O^{li}$

(d) Get the mass of the i^{th} joint m_{li}

(e) Perform the update:

$$\mathbf{B}(\mathbf{q}) \longleftarrow \mathbf{B}(\mathbf{q}) + m_{li}\mathbf{J}_P^{li\,T}\mathbf{J}_P^{li} + \mathbf{J}_O^{li\,T0}\mathbf{R}_{G_i}\mathbf{I}_{l_i}^{O_i||0}\mathbf{R}_{G_i}^T\mathbf{J}_O^{li}$$

The code that do this is as follows:

```
def getB(self, joint):
    ##TODO: Fill in this function to complete Q2.
    ## "joint" is a numpy array of double consisting of the joint value.
    ## The output is a numpy 7*7 matrix.

    # Initialize:
    B = np.zeros((7,7))

    for frame_idx in range(1,8):
        # 1. Forward KE at COM --> Get T --> R
        R_0_Gi          = self.forward_kine_cm(joint, frame_idx)[0:3, 0:3]

        # 2. Inertia matrix
        I_li_Oi_p       = np.eye(3)
        I_li_Oi_p[0,0] = self.Ixyz[frame_idx - 1, 0]
        I_li_Oi_p[1,1] = self.Ixyz[frame_idx - 1, 1]
        I_li_Oi_p[2,2] = self.Ixyz[frame_idx - 1, 2]

        # 3. I_CAL_li
        I_CAL_li        = np.matmul(np.matmul(R_0_Gi, I_li_Oi_p), R_0_Gi.T)

        # 4. Jacobian at COM
        J_li            = self.get_jacobian_cm(joint, frame_idx)
        J_P_li          = J_li[0:3,:]
        J_O_li          = J_li[3:6,:]

        # 5. mass
        m_li            = self.mass[frame_idx - 1]

        # 6. Combine:
        B             += m_li*np.matmul(J_P_li.T, J_P_li) + np.matmul(np.matmul(J_O_li.T, I_CAL_li), J_O_li)
        #print(B)
        #print(frame_idx)
    assert B.shape == (7,7)
    return B
```

Dynamic component $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$

For the dynamic component $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$, a finite difference approach is used, the procedure of calculating the 7×7 matrix $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ is as follows:

1. Initialize a 7×7 matrix with zeros and a very small number $\varepsilon = 10^{-6}$
2. For each entry (i, j) in the initialized matrix:
   (a) Calculate the dynamic component $\mathbf{B}$ with $\mathbf{q}$
   (b) Calculate the dynamic component $\mathbf{B}_{ij}$ with $\mathbf{q}$ which had ε added to the k^{th} entry of $\mathbf{q}$
   (c) Calculate the dynamic component $\mathbf{B}_{jk}$ with $\mathbf{q}$ which had ε added to the i^{th} entry of $\mathbf{q}$
   (d) Calculate the Christoffel symbols h_{ijk} using finite difference:

$$h_{ijk} = \frac{(i,j)^{th} \text{ component of } \mathbf{B}_{ij}}{\varepsilon} - \frac{(i,j)^{th} \text{ component of } \mathbf{B}}{\varepsilon} - \frac{1}{2}\left[\frac{(j,k)^{th} \text{ component of } \mathbf{B}_{jk}}{\varepsilon} - \frac{(j,k)^{th} \text{ component of } \mathbf{B}}{\varepsilon}\right]$$

   (e) Get the joint velocities $\dot{\mathbf{q}}$
   (f) Calculate the values of the entries c_{ij}:

$$c_{ij} = \sum_{k=1}^{n} h_{ijk}\dot{q}_k$$

The code that do the above is as follows:

```
def getC(self, joint, vel):
    ##TODO: Fill in this function to complete Q2.
    ## "joint" is a numpy array of double consisting of the joint value.  q
    ## "vel" is a numpy array of double consisting of the joint velocity. q_dot
    ## The output is a numpy 7*7 matrix.

    # Initialize:
    C      = np.zeros((7,7))
    eps    = 0.000001

    for i in range(0,7):
        for j in range(0,7):
            for k in range(0,7):
                B           = self.getB(joint)
                joints_k    = np.copy(joint)
                joints_k[k] = joints_k[k] + eps
                B_ij        = self.getB(joints_k)
                joint_i     = np.copy(joint)
                joint_i[i]  = joint_i[i] + eps
```

```
                B_jk        = self.getB(joint_i)
                x           = (B_ij[i,j]/eps) - (B[i,j]/eps)
                y           = 0.5*((B_jk[j,k]/eps) - (B[j,k]/eps))
                C[i,j]     += (x - y)*vel[k]

    assert C.shape == (7,7)
    return C
```

Dynamic component $\mathbf{g}(\mathbf{q})$

For the dynamic component $\mathbf{g}(\mathbf{q})$, it is calculated as follows:

1. Initialize a 7×1 vector with zeros, the gravitation vector with $\mathbf{g}_0 = [0, 0, -9.8]^T$ and $\varepsilon = 10^{-6}$
2. For every frame i:
   (a) Calculate the i^{th} component of $\mathbf{g}$ by finite difference as follows:

$$g_i(\mathbf{q}) = \frac{\mathbf{P}(\mathbf{q} + \varepsilon_i) - \mathbf{P}(\mathbf{q})}{\varepsilon}$$

   where $\mathbf{P}(\mathbf{q}+\varepsilon_i)$ is the potential energy with ε added to $\mathbf{q}$'s i^{th} entry and $\mathbf{P}(\mathbf{q})$ is the potential energy no ε added. The potential energy is calculated as follows:

$$\mathbf{P}(\mathbf{q}) = -\sum_{j=1}^{n} m_{li} \mathbf{g}_0^T \mathbf{p}_{li}$$

   where $\mathbf{p}_{li}$ is the position of the center of mass of the link and can be calculated using forward kinematics at the center of mass.

The code that do this is as follows:

```
def getG(self, joint):
    ##TODO: Fill in this function to complete Q2.
    ## "joint" is a numpy array of double consisting of the joint
       value.
    ## The output is a numpy array 7*1.
    G   = np.zeros((7,1))
    g_0 = np.array([0, 0, -self.g]).reshape(3,1)
    eps = 0.000001

    for i in range(0,7):
        P_q = 0.0
        P_q_eps = 0.0
        for j in range(0,7):
            m_lj        = self.mass[j]
            joint_i     = np.copy(joint)
            joint_i[i]  = joint[i] + eps
            p_lj        = self.forward_kine_cm(joint, j + 1)[0:3,3]
```

```
            p_li_eps    = self.forward_kine_cm(joint_i, j + 1)[0:3,3]

            # Sum:
            P_q       -= m_lj*g_0.T.dot(p_lj)
            P_q_eps   -= m_lj*g_0.T.dot(p_li_eps)
        G[i] = (P_q_eps - P_q)/eps
    assert G.shape == (7,1)
    return G
```

Question 3

Forward dynamics is the computation of the accelerations' vector and the possible end effector forces given the torques $\tau(t)$ applied at the joints. Since forward dynamics use torques to compute the accelerations for resultant motions, this is useful in predicting human motions because this process is mimicking the process when the neurons in human body control the muscles to give the accelerations in our joints which lead to motions. Another application is to simulate manipulators. A difficulty in computing the forward dynamics is that we need to compute the inverse of the dynamic component **B** which may need significant computation time. Another difficulty is when there are existance of external forces like wind or earthquake as they may not be observed easily as our calculations do not concern random forces.

Question 4

Inverse dynamics is to compute the torques applied at every joints with the given joint accelerations velocities and positions. The difficulty in inverse dynamics is that external forces do affect motion but can not be observed from the kinematic motion straight away. Apart from robotics, inverse dynamics computes the torque of anatomical structures across a joint needed to produce the observed motions of the joint in the field of biomechanics.

Question 5

a)

In this part, the three poses from cw3bag1.bag are published to the robot and the dynamic components are used to calculate the acceleration using the steps as follows:

1. Get joint position, velocity and torque from the subscriber subscribing to JointState
2. Use KDL library to calculate the dynamic components $\mathbf{B}, \mathbf{C}$ and $\mathbf{g}$ (I modified the KDL library, so please use my KDL library named kdl_kine_solver_cw3q5.py)

3. The acceleration $\ddot{\mathbf{q}}$ is then calculated as:

$$\ddot{\mathbf{q}} = \mathbf{B}^{-1}(\tau - \mathbf{C} - \mathbf{g})$$

PS: We need to subscribe and publish to the /iiwa/'s JointState and JointTrajectory (I use iiwa14Kine.py to subscribe so the subscriber has to be changed accordingly).

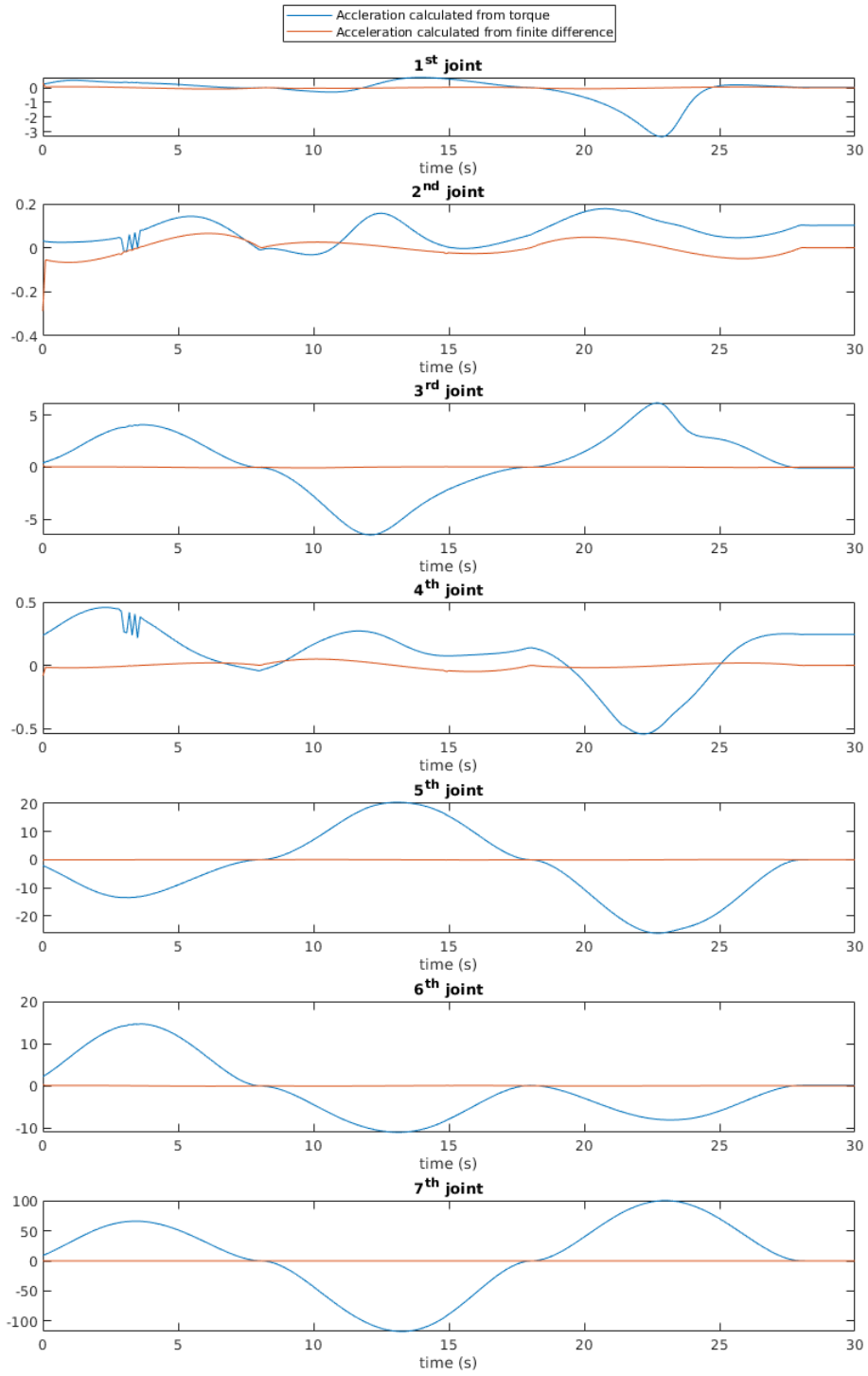
This is a forward dynamics problem since we use the torques to calculate the accelerations. These three steps are coded as the following:

```
def acceleration(my_iiwa, KDL):
    # Get torque velocity and position:
    torque = np.array(my_iiwa.current_joint_torque).reshape(7,1)
    q_dot  = np.array(my_iiwa.current_joint_velocity)
    q      = np.array(my_iiwa.current_joint_position)

    #print("q: {}".format(q))
    # 1. Calculate B
    B_KDL = KDL.getB(q)
    B = np.zeros((7,7))
    for i in range(7):
        for j in range(7):
            B[i,j] = B_KDL[i,j]

    # 2. Calculate C
    C_KDL = KDL.getC(q, q_dot)
    C_and_q = np.zeros((7,1))
    for i in range(7):
        C_and_q[i] = C_KDL[i]

    # 3. Calculate G
    G_KDL = KDL.getG(q)
    G = np.zeros((7,1))
    for i in range(7):
        G[i] = G_KDL[i]

    # 5. Calculate q_ddot
    q_ddot = np.matmul(np.linalg.inv(B), (torque - C_and_q - G))

    return q_ddot, q_dot
```

Then the ground truth acceleration is calculated using finite difference as follows:

$$\ddot{\mathbf{q}}_{truth} = \frac{\dot{\mathbf{q}}(t) - \dot{\mathbf{q}}(t-1)}{dt}$$

where $dt = 0.1$ in my case. The values are stored in q_ddot.txt and ground_accel.txt and the results are plotted using MATLAB. The result is then plotted as shown:



We can see that the acceleration calculated from torque in joint 2 and 4 somehow follows that calculated from finite difference. The reason why the other joints have the finite different almost a straight line is because the controllers in the robot simulation are not properly tuned.

b)

In this part we have to calculate the mass of a block m, and the center of mass position $\mathbf{r} = [x, y, z]^T$ of the object attached to the robot using the three joint poses as shown:

$$\mathbf{q}_a = [45^\circ, -35^\circ, 55^\circ, -30^\circ, -25^\circ, 65^\circ, -10^\circ]^T$$
$$\mathbf{q}_b = [5^\circ, 40^\circ, 80^\circ, 10^\circ, 8^\circ, -50^\circ, -10^\circ]^T$$
$$\mathbf{q}_c = [-50^\circ, 60^\circ, 0^\circ, -15^\circ, 60^\circ, -25^\circ, -50^\circ]^T$$

In this problem, we will first publish the three pose above to the robot and take the real time pose $\mathbf{q}$ from the subscriber. Then after the robot moved to the pose we can use the equation of motion:

$$\mathbf{B}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) + \tau_{ext} = \tau$$

After the robot stop moving we have:

$$\mathbf{g}(\mathbf{q}) + \tau_{ext} = \tau$$
$$\tau_{ext} = \tau - \mathbf{g}(\mathbf{q})$$

Now we have the values for τ_{ext}, and the torque at every joint τ^i_{ext} can be related to the external force F_{ext} and it's center of mass coordinates as follows:

$$\tau^i_{ext} = m\mathbf{g}^T[\mathbf{z}_{i-1} \times (\mathbf{r} - \mathbf{p}_i)]$$

so all in all we have 7 equations. If we further expand the equation above, we have:

$$\begin{aligned}
\tau^i_{ext} &= m \begin{bmatrix} 0 & 0 & g \end{bmatrix} \left[\begin{bmatrix} \mathbf{z}^x_{i-1} \\ \mathbf{z}^y_{i-1} \\ \mathbf{z}^z_{i-1} \end{bmatrix} \times \begin{bmatrix} x - \mathbf{p}^x_i \\ y - \mathbf{p}^y_i \\ z - \mathbf{p}^z_i \end{bmatrix} \right] \\
&= m \begin{bmatrix} 0 & 0 & g \end{bmatrix} \begin{bmatrix} \mathbf{z}^y_{i-1}(z - \mathbf{p}^z_i) - \mathbf{z}^z_{i-1}(y - \mathbf{p}^y_i) \\ \mathbf{z}^z_{i-1}(x - \mathbf{p}^x_i) - \mathbf{z}^x_{i-1}(z - \mathbf{p}^z_i) \\ \mathbf{z}^x_{i-1}(y - \mathbf{p}^y_i) - \mathbf{z}^y_{i-1}(x - \mathbf{p}^x_i) \end{bmatrix} \\
&= mg[\mathbf{z}^x_{i-1}(y - \mathbf{p}^y_i) - \mathbf{z}^y_{i-1}(x - \mathbf{p}^x_i)] \\
\frac{\tau^i_{ext}}{mg} &= \mathbf{z}^x_{i-1}y - \mathbf{z}^x_{i-1}\mathbf{p}^y_i - \mathbf{z}^y_{i-1}x + \mathbf{z}^y_{i-1}\mathbf{p}^x_i \\
\frac{\tau^i_{ext}}{mg} - \mathbf{z}^x_{i-1}y + \mathbf{z}^y_{i-1}x &= \mathbf{z}^y_{i-1}\mathbf{p}^x_i - \mathbf{z}^x_{i-1}\mathbf{p}^y_i
\end{aligned}$$

We can see on the left side we have three unknowns m, x, y and we can calculate the value on the right side since $\mathbf{z}_{i-1}, \mathbf{p}_i$ can be obtained from forward kinematics. If we stack three equations (three joints) together we can solve m, x, y with linear algebra, here we choose joint three, five and seven:

$$\frac{\tau^3_{ext}}{mg} - \mathbf{z}^x_2 y + \mathbf{z}^y_2 x = \mathbf{z}^y_2\mathbf{p}^x_3 - \mathbf{z}^x_2\mathbf{p}^y_3$$
$$\frac{\tau^5_{ext}}{mg} - \mathbf{z}^x_4 y + \mathbf{z}^y_4 x = \mathbf{z}^y_4\mathbf{p}^x_5 - \mathbf{z}^x_4\mathbf{p}^y_5$$
$$\frac{\tau^7_{ext}}{mg} - \mathbf{z}^x_6 y + \mathbf{z}^y_6 x = \mathbf{z}^y_6\mathbf{p}^x_7 - \mathbf{z}^x_6\mathbf{p}^y_7$$

$$\begin{bmatrix} \frac{\tau^3_{ext}}{g} & -\mathbf{z}^x_2 & \mathbf{z}^y_2 \\ \frac{\tau^5_{ext}}{g} & -\mathbf{z}^x_4 & \mathbf{z}^y_4 \\ \frac{\tau^7_{ext}}{g} & -\mathbf{z}^x_6 & \mathbf{z}^y_6 \end{bmatrix} \begin{bmatrix} \frac{1}{m} \\ y \\ x \end{bmatrix} = \begin{bmatrix} \mathbf{z}^y_2\mathbf{p}^x_3 - \mathbf{z}^x_2\mathbf{p}^y_3 \\ \mathbf{z}^y_4\mathbf{p}^x_5 - \mathbf{z}^x_4\mathbf{p}^y_5 \\ \mathbf{z}^y_6\mathbf{p}^x_7 - \mathbf{z}^x_6\mathbf{p}^y_7 \end{bmatrix}$$

For z we need to make use of the position of joint seven. Using trigonometry and similar triangles as shown in Figure 3

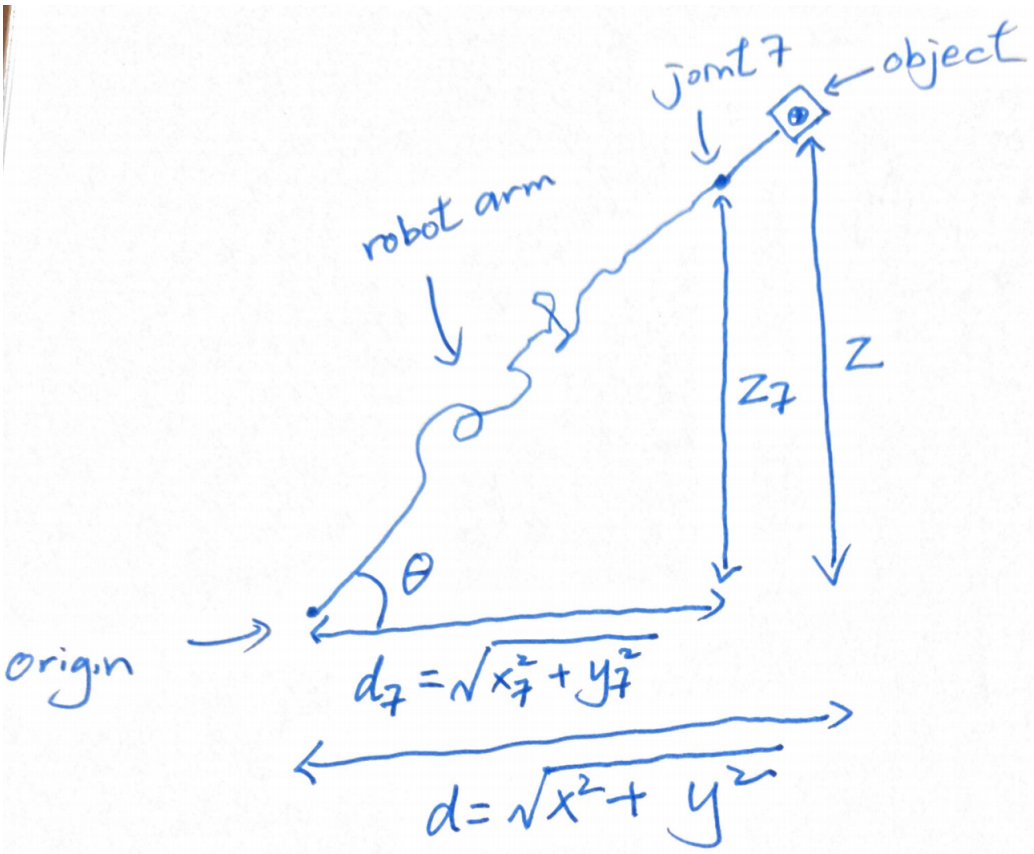


Figure 3: Trigonometry to calculate the z component of the center of mass

We have the following relation:

$$\frac{z_7}{d_7} = \frac{z}{d}$$
$$\frac{z_7}{\sqrt{x_7^2 + y_7^2}} = \frac{z}{\sqrt{x^2 + y^2}}$$
$$z = \frac{z_7\sqrt{x^2 + y^2}}{\sqrt{x_7^2 + y_7^2}}$$

So z is recovered. The code that do the above is as follows, we need to subscribe and publish to the /object_iiwa/'s JointState and JointTrajectory:

```
def torque_external(KDL, torque, q_dot, q):

    G_KDL = KDL.getG(q)
    G = np.zeros((7,1))
    for i in range(7):
        G[i] = G_KDL[i]

    torque_ext = torque - G

    return torque_ext

```

```
def joint_positions(my_iiwa, q):
    # Get positions of joints

    joints_pos = []
    joints_z   = []
    for frame_idx in range(7):
        T = my_iiwa.forward_kine(q, frame_idx)
        joints_pos.append(T[0:3,3])
        joints_z.append(T[0:3,2])

    return joints_pos, joints_z

def mass_COM(torque_ext, selected_joints_pos, selected_joints_z):
    g = 9.8
    A = np.array([
        [torque_ext[0][0]/g, - selected_joints_z[0][0],
           selected_joints_z[0][1]],
        [torque_ext[1][0]/g, - selected_joints_z[1][0],
           selected_joints_z[1][1]],
        [torque_ext[2][0]/g, - selected_joints_z[2][0],
           selected_joints_z[2][1]]
    ])
    B = np.array([
        [selected_joints_z[0][1]*selected_joints_pos[0][0] -
           selected_joints_z[0][0]*selected_joints_pos[0][1]],
        [selected_joints_z[1][1]*selected_joints_pos[1][0] -
           selected_joints_z[1][0]*selected_joints_pos[1][1]],
        [selected_joints_z[2][1]*selected_joints_pos[2][0] -
           selected_joints_z[2][0]*selected_joints_pos[2][1]]
    ])
    soln = np.linalg.solve(A, B)
    m = 1/soln[0][0]
    y = soln[1][0]
    x = soln[2][0]
    z = (selected_joints_pos[2][2]*np.sqrt(x**2 + y**2))/np.sqrt(
       selected_joints_pos[2][0]**2 + selected_joints_pos[2][1]**2)
    COM = [x, y, z]
    return m, COM
```

As the program runs sufficiently long the we will see the mass m converge to 1.083, which is the mass of the block. By switching q_choice to a, b and c in the program, we can obtain the mass m and the center of mass (COM) in $[x, y, z]$ as follows:

$$\mathbf{q}_a : m = 1.08301212892 \qquad COM = \begin{bmatrix} -0.32703449479174301 \\ 0.11685392890830354 \\ 0.95532032240813314 \end{bmatrix}$$

$$\mathbf{q}_b : m = 1.08299536392 \qquad COM = \begin{bmatrix} 0.61967323297099364 \\ -0.25448320523720852 \\ 1.2801873683402074 \end{bmatrix}$$

$$\mathbf{q}_c : m = 1.08478463918 \qquad COM = \begin{bmatrix} 0.55118926484499198 \\ -0.81312896966517367 \\ 0.88204883992119987 \end{bmatrix}$$

which gives a 0.0552% error for the mass.

For $\mathbf{q}_d$ the external torque acting on every joint is:

$$\tau_{ext} = \begin{bmatrix} 7.76333815 \times 10^{-5} \\ 4.36781632 \\ -3.79413785 \\ 9.11462430 \times 10^{-2} \\ -5.77635795 \times 10^{-2} \\ 4.07064037 \times 10^{-1} \\ -1.64104577 \times 10^{-5} \end{bmatrix}$$

If the program run too long there may be errors accumulated because there time delay for the program to receive the values. So as soon as the robot stabilized at the pose, the values are noted.

References

[1] M. Shimizu, H. Kakuya, W. Yoon, K. Kitagaki, and K. Kosuge, "Analytical inverse kinematic computation for 7-dof redundant manipulators with joint limits and its application to redundancy resolution," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1131–1142, 2008.