COMP0127 Coursework 3

Jian Zhou

Q1

a. The manipulator should have three degrees of freedom.
b. A delta parallel manipulator will be applied. As candies are placed on the conveyor randomly within a width of 20 cm, an accurate manipulator is required. Furthermore, to adapt to the high production capacity of candies, multiple robots could be used, then the smaller workspace can be an advantage.
   End-effectors are grippers for picking and placing
c. Stepper motors are chosen, because, for delta parallel manipulator, there are multiple revolute joints in a parallel delta machine. Stepper motors can accurately make the joint rotate a required angle.
d. One fixed pinhole camera is required. The camera will be placed before the robot in case of obstruction. As the conveyor belt is running in a fixed velocity, the effect of distance between robot and camera can be eliminated.
   Pressure sensors are required to be placed on the surface of grippers, which can indicate whether the candy is carried correctly.
e. Firstly and obviously, the gripper could wear out as it constantly picking and placing, which using the friction to hold candies. Secondly, the differentials between stepper motor and manipulator could wear. In the end, the mechanical components of the robot such as joints could wear.
f. Placing multiple high accurate pressure sensors to the interaction point of frame (holding robot) and robot. Meanwhile, let the manipulator pause for a short time (say 0.5 second) when it holds a candy. This could make the system physically stable, so that can scale the weight accurately.

Q2

a. The Jacobian matrix is deduced in lecture 4 and the formula is shown below

$$J^{(li)} = \begin{bmatrix} J_{p_1}^{(li)} & \cdots & J_{p_j}^{(li)} & \cdots & J_{o_n}^{(li)} \\ J_{o_1}^{(li)} & \cdots & J_{o_j}^{(li)} & \cdots & J_{o_n}^{(li)} \end{bmatrix}$$

Where:

$$J_p^{(li)} = \begin{cases} z_{j-1} * (p_{li} - p_{j-1}) & \text{if joint i is revolute} \\ z_{j-1} & \text{if joint i is prismatic} \end{cases}$$

$$J_o^{(li)} = \begin{cases} z_{j-1} & \text{if joint i is revolute} \\ 0 & \text{if joint i is prismatic} \end{cases}$$

The parameter p_{j-1} can be obtained by applying *"orward_kinematics_center_of_mass()"* method. The first three elements in the last column are the position, representing the displacement of the center of gravity. Code are shown below.

```
# Your code starts here ----------------------------
T_0Gi = self.forward_kinematics_centre_of_mass(
    joint_readings, up_to_joint)
p_li = T_0Gi[0:3, -1]
jacobian = np.zeros((6, 7))
for i in range(up_to_joint):
    T_0i = self.forward_kinematics(joint_readings, i)
    p_i = T_0i[0:3, -1]
    z_i = T_0i[0:3, -2]
    jacobian[0:3, i] = np.cross(z_i, (p_li-p_i), axis=0)
    jacobian[3:6, i] = z_i
# Your code ends here ------------------------------
```

Figure 1: code for q1a

b. The inertia matrix **B(q)** can be expressed as

$$B(q) = \sum_{i=1}^{n} (m_{li} J_p^{(li)^T} J_p^{(li)} + J_o^{(li)^T} j_{li} J_o^{(li)})$$

Where j_{li} is the tensor of inertia:

$$j_{li} = {}^0R_{Gi} I_{l_i}^{O_i||} {}^0R_{Gi}^T$$

Where ${}^0R_{Gi}$ can be found in the transformation matrix. Codes are shown below.

```
# Your code starts here ------------------------------
for i in range(7):
    I_li = np.zeros((3, 3))
    I_li[0, 0] = self.Ixyz[i, 0]
    I_li[1, 1] = self.Ixyz[i, 1]
    I_li[2, 2] = self.Ixyz[i, 2]
    T_0Gi = self.forward_kinematics_centre_of_mass(
        joint_readings, up_to_joint=i+1)
    R_0Gi = T_0Gi[0:3, 0:3]
    J_li = R_0Gi@I_li@R_0Gi.T
    m_li = self.mass[i]
    jacobian_li = self.get_jacobian_centre_of_mass(
        joint_readings, up_to_joint=i+1)
    jp_li = jacobian_li[0:3, :]
    jo_li = jacobian_li[3:6, :]
    B = B+m_li*jp_li.T@jp_li+jo_li.T@jo_li

# Your code ends here ------------------------------
```

Figure 2: code for q2b

c. To compute the dynamic component $\boldsymbol{C(q,\dot{q})\dot{q}}$ we need to compute Christoffel symbol h_{ijk}:

$$h_{ijk} = \frac{\partial b_{ij}(q)}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_i}$$

There currently no closed form of this partial differential formula, so we implemented a micro difference and treated it with calculating a discrete slope to represent the differential. Details are shown in the figure below.

Then $c(i,j)$ can be calculated by:

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

Note that we are supposed to calculate the production of Christoffel symbol and velocities under the instruction of the description of the function in template.

```
# Your code starts here ------------------------------
h_ijk = np.zeros((7, 7))
B = self.get_B(joint_readings)
dq = 1e-5
C = np.zeros((7, 7))
for i in range(7):
    for j in range(7):
        for k in range(7):

            q_i = joint_readings.copy
            q_k = joint_readings.copy
            q_i[i] += dq
            q_k[k] += dq
            b_ij = self.get_B(q_k.tolist())
            b_jk = self.get_B(q_i.tolist())

            db_ij = (b_ij[i, j]-B[i, j])/dq
            db_jk = (b_jk[j, k]-B[j, k])/dq

            h_ijk = db_ij-db_jk/2

            C[i, j] = C[i, j]+h_ijk*joint_velocities[k]
C = C@np.array(joint_velocities)
# Your code ends here ------------------------------
```

Figure 3: code for q1c

d.

The calculation of gravity matrix is similar to the Christoffel symbol, as the gravity is calculated by the partial differential as well.

Firstly, we calculate the potential energy $P(q)$:

$$P(q) = -\sum_{i=1}^{n} m_{li} g_0^T P_{li}$$

And then $g(q)$:

$$g_n = \frac{\partial P(q)}{\partial q_n}$$

This formula is implemented by apply finite difference method.

```
# Your code starts here ----------------------------
g = np.zeros((7,))
g0 = np.array([0, 0, self.g]).reshape((3, 1))
dq = 1e-5
for i in range(7):
    Potential_i = 0
    Potential_i_h = 0
    for j in range(7):
        q_i = joint_readings.copy
        q_i[i] = joint_readings[i]+dq

        p_li = self.get_jacobian_centre_of_mass(
            joint_readings, j+1)[0:3, i]
        p_li_h = self.get_jacobian_centre_of_mass(q_i, j+1)[0:3, i]

        m_lj = self.mass[j]

        Potential_i = Potential_i-m_lj*g0@p_li
        Potential_i_h = Potential_i_h-m_lj*g0@p_li_h
    g[i] = (Potential_i_h-Potential_i)/dq
# Your code ends here ----------------------------
```

Q3

Huyges-Steiner theorem refers to parallel-axis theorem, which can calculate the moment of inertia about any parallel axis when we have already known the tensor of inertia of a rigid object.

Firstly, it assumes that the object is rigid, which will not bend in motion. Secondly, the tensor of inertia about an axis through the object's center of gravity and the perpendicular distance between the axes are supposed to be known. Lastly, the target axis should be parallel to the known reference inertial axis.

Derivation: we define the object has its moment of inertia with respect to z as I.

$$I_{cm} = \int (x^2 + y^2) dm$$

Where:

- I_{cm} Moment of inertia at the center of mass
- dm Elementary mass

Then we move it to another position with current z' axis being parallel to z axis. Then we have:

$$I'_{cm} = \int ((x - D)^2 + y^2) dm$$

$$= \int (x^2 + y^2) dm + D^2 \int dm - 2D \int x dm$$

$$= \int (x^2 + y^2) dm + D^2 \int dm$$

$$= I_{cm} + md^2$$

Importance: It provides an approach to calculate the inertial matrix when moving axis, which frequently happens in dynamics of robot.

Q4

Forward dynamics uses the kinematic formula to predict the motion of robot when we have already known the torques or forces applied to robots. The main application of forward dynamics is simulation. For instance, unreal engine has the ability to build virtual worlds that have realistic physical performance basing on its outstanding forward dynamics. One of the greatest difficulties of forward dynamics is the cost on computation. Simulation of fluid mechanics has been consuming huge computational resources but still no good results.

Inverse dynamics refers to the deduction of forces or torques that we need to move the robot to a target pose or position. The main application of inverse dynamics is controlling the robot to get to the positions or poses that human want. One of the difficulties is that the performance of inverse dynamics really counts on the measurement feedback. Therefore, the performance of inverse dynamics can be greatly influenced by measurement errors. Another difficulty is the selection of multiple configurations.

Q5

a. The bag file contains the message type of **'trajectory_msgs/JointTrajectory'**. Only one message is included in this cw3q5.bag. The message contains the positions, velocities, and accelerations of seven joints at 10, 20 and 30 seconds.
   Note that time information is stored in the '$header.stamp.time\_from\_start$'.
b. The problem is an inverse dynamics problem because we only have desired position information without velocities and accelerations. So we need to calculate the kinematic parameters basing on desired positions.
c. The messages with type '$trajectory\_msgs/JointTrajectory$' are supposed to be published to the topic "**$/iiwa/EffortJointInterface\ trajectory\ controller/command$**". Using the launch file in template we can find the simulated iiwa robot.
d. A subscriber was created to subscribe to the topic of '**$/iiwa/joint\_states$**', which contains position, velocities, and efforts. Acceleration can be calculated by a closed form of formula shown below.

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

e. Plotting the accelerations as a function of time is quite easy. All we need to do is storing the acceleration in each frame and its corresponding time stamp. Then with the help of python module 'matplotlib.pyplot', the figure can be easily drawn below.

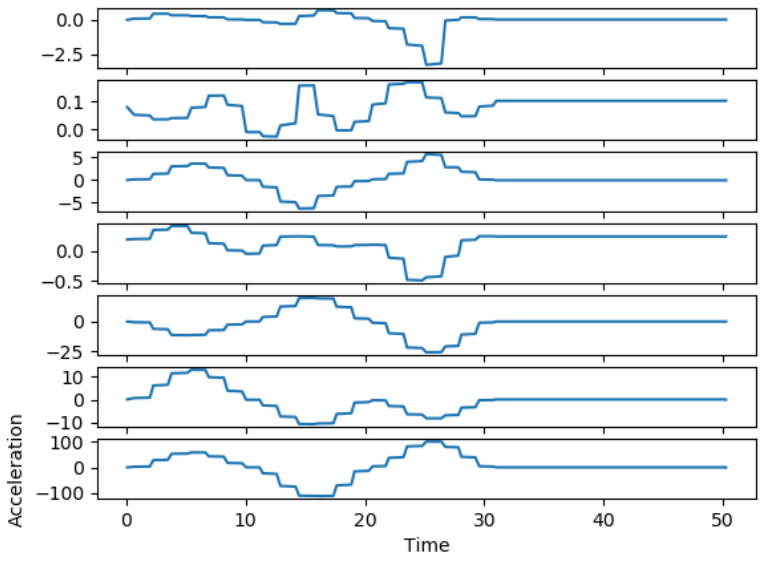

Figure: Acceleration plot of the motion