MPHY0054 Robotic Systems Engineering Coursework 3: Actuators, Mechanisms and Robot Dynamics

Harry Yao
zcabry0@ucl.ac.uk

Jan 01, 2025

Section 1

1. a. This robotic system requires 3 DOF:

   - X-axis: To locate the candy horizontally along the width of the conveyor
   - Y-axis: To select candies along the conveyor's movement
   - Z-axis: To lift the candy from the conveyor and place it into the basket

   b. A serial SCARA(Selective Compliance Articulated Robot Arm) is the most ideal design due to its speed and simplicity for candy grab robot. A small two finger soft gripper is suit for the end effector

   c. Brushless DC motors are the best choice for candy grabbing due to their high efficiency, speed, and durability. Combine with a belt transmission, it will ensure smooth and reliable operation for SCARA robots

   d. Choice of Sensors:

   - RGB Camera: Set above the conveyor to cover the entire workspace. Detects candy color and position in real-time.
   - Rotary Encoders: Attached to each joint of the manipulator to provide real-time positional feedback. Ensures precise control of manipulator movement.
   - Proximity Sensor: Connected with the end-effector for direct contact detection. Confirming successful candy pickup before moving to the basket.

e. Joints in the SCARA robot are prone to wear due to frequent rotations and load stress. To minimize wear, use high-quality bearings, apply regular lubrication, and add dust seals. Optimizing load distribution and scheduling maintenance ensures long-term reliability and precision in repetitive pick-and-place tasks

f. Install a small load cell into the two-finger gripper to measure candy weight during pickup. The system can compare the measured weight to the threshold 5 grams. Only candies meeting both color by RGB camera and weight criteria are placed in the basket, ensuring accuracy without altering the conveyor

Section 2

2. a. Initialization of the Jacobian matrix:
The Jacobian matrix is initialized as a 6×7 zero matrix, where each column corresponds to a joint of the manipulator. The top three rows represent linear velocity contributions, and the bottom three rows represent angular velocity contributions.

```
    jacobian = np.zeros((6, 7))
```

Computation of the center of mass position:
The function `forward_kinematics_centre_of_mass` calculates the transformation matrix T_{com} for the center of mass up to the specified joint. The position of the CoM, $\mathbf{p}_{\text{com}}$, is extracted from the fourth column of T_{com}:

$$\mathbf{p}_{\text{com}} = T_{\text{com}}[0:3, 3]$$

```
    T_com = self.forward_kinematics_centre_of_mass(
        ↪ joint_readings, up_to_joint)
    p_com = T_com[0:3, 3]
```

Iteration over joints:
The code iterates over all joints up to the specified joint index. For each joint, the following computations are performed:

1. Compute the forward kinematics T_i to get the position and orientation of the joint. 2. Extract $\mathbf{z}_i$, the rotation axis (z-axis), and $\mathbf{p}_i$, the origin of the joint in the base frame.

```
    for i in range(up_to_joint):
        T_i = self.forward_kinematics(joint_readings, i)
        z_i = T_i[0:3, 2]  # z-axis in base frame
        p_i = T_i[0:3, 3]  # origin of the joint in base
            ↪ frame
```

Computation of the Jacobian components:
The linear velocity contribution $\mathbf{Jv}_i$ is calculated as the cross product of $\mathbf{z}_i$ and $(\mathbf{p}_{\text{com}} - \mathbf{p}_i)$. The angular velocity contribution $\mathbf{Jw}_i$ is simply $\mathbf{z}_i$.

$$\mathbf{Jv}_i = \mathbf{z}_i \times (\mathbf{p}_{\text{com}} - \mathbf{p}_i), \quad \mathbf{Jw}_i = \mathbf{z}_i$$

These values are then assigned to the appropriate rows of the Jacobian matrix.

```
        Jv_i = np.cross(z_i, (p_com - p_i))
        Jw_i = z_i
        jacobian[0:3, i] = Jv_i
        jacobian[3:, i] = Jw_i
```

b. Iteration over joints:
The loop iterates over all the joints from 1 to the total number of joints. For each joint i, the Jacobian at the center of mass is computed using the function `get_jacobian_centre_of_mass`.

```
    for i in range(1, len(joint_readings) + 1):
        jacobian = self.get_jacobian_centre_of_mass(
            ↪ joint_readings, i)
```

Splitting the Jacobian:
The Jacobian is split into two parts:

- J_p: The top three rows, representing the linear velocity contribution.
- J_o: The bottom three rows, representing the angular velocity contribution.

```
        J_p = jacobian[0:3, :]
        J_o = jacobian[3:, :]
```

Mass and inertia extraction:
For each link, the mass m_i and the diagonal inertia tensor I_{link} in the local frame are retrieved.

```
        m_i = self.mass[i - 1]
        I_link = np.diag(self.Ixyz[i - 1])
```

Transformation to base frame:
The rotation matrix R_i from the base to the link's frame is extracted, and the inertia tensor is transformed to the base frame using the formula:

$$I_{\text{base}} = R_i \cdot I_{\text{link}} \cdot R_i^T$$

```
        T_com_i = self.forward_kinematics_centre_of_mass(joint_readings, up_to_joint=i)
        R_i = T_com_i[0:3, 0:3]
        I_base = R_i @ I_link @ R_i.T
```

Linear and angular contributions:
The contributions of linear and angular velocities to the inertia matrix are calculated separately:

$$\text{Linear: } m_i \cdot J_p^T J_p$$

$$\text{Angular: } J_o^T I_{\text{base}} J_o$$

```
        linear_contribution = m_i * (J_p.T @ J_p)
        angular_contribution = J_o.T @ I_base @ J_o
```

Summation of contributions:
The contributions are accumulated in the inertia matrix B:

$$B = \sum_{i=1}^{n} \left(m_i \cdot J_p^T J_p + J_o^T I_{\text{base}} J_o \right)$$

```
        B += linear_contribution + angular_contribution
```

c. Initialize variables:
The Christoffel symbols h_{ijk} and Coriolis matrix C are initialized as zero matrices. A small perturbation $\delta = 1 \times 10^{-8}$ is used for numerical differentiation.

```
    delta = 1e-8
    n = len(joint_readings)
    h_ijk = np.zeros((n, n, n))  # Christoffel symbols
    C = np.zeros((n, n))         # Coriolis matrix
```

Compute Christoffel symbols h_{ijk}:
For each combination of i, j, k, the partial derivatives of $B(q)$ are computed numerically using finite differences:

$$h_{ijk} = \frac{\partial B_{ij}}{\partial q_k} - \frac{1}{2} \frac{\partial B_{jk}}{\partial q_i}$$

```
    for k in range(n):
        delta_q_k = np.zeros(n)
        delta_q_k[k] = delta

        for j in range(n):
            for i in range(n):
                delta_q_i = np.zeros(n)
                delta_q_i[i] = delta

                dB_ij_dqk = (
                    self.get_B((np.array(joint_readings) + delta_q_k).tolist())[i, j] -
                    self.get_B(joint_readings)[i, j]
                ) / delta

                dB_jk_dqi = (
                    self.get_B((np.array(joint_readings) + delta_q_i).tolist())[j, k] -
                    self.get_B(joint_readings)[j, k]
                ) / delta

                h_ijk[i, j, k] = dB_ij_dqk - 0.5 * dB_jk_dqi
```

Construct the Coriolis matrix C:
The Coriolis matrix is constructed by summing over the Christoffel symbols and joint velocities:

$$C_{ij} = \sum_k h_{ijk}\dot{q}_k$$

```
    for k in range(n):
        C += h_ijk[:, :, k] * joint_velocities[k]
```

Compute the final Coriolis term:
The final result is computed as:

$$C \cdot \dot{q}$$

```
    C = C @ joint_velocities
```

d. Initialize variables:
The gravity matrix g is initialized as a zero vector, and the gravitational acceleration vector is defined in the base frame as g $= [0, 0, -g]^T$. A cache is also initialized to store the Jacobian matrices of each link's CoM, avoiding redundant computations.

```
    g = np.zeros(7)
    grav = np.array([0, 0, -self.g])
    jacobian_cache = {}
```

Main loop:
The code iterates over each link of the robot, and for each link:

- Compute or retrieve the cached Jacobian matrix at the CoM of the link.
- Compute the force wrench F_i, which includes the gravitational force acting on the CoM and no angular force components:

$$F_i = [m_i \cdot g, 0, 0, 0]^T$$

- Calculate the torque contribution to each joint as:

$$\tau_i = J_{\text{com},i}^T \cdot F_i$$

- Accumulate τ_i into the gravity matrix g.

```python
for i in range(7):
    if i + 1 not in jacobian_cache:
        jacobian_cache[i + 1] = self.get_jacobian_centre_of_mass(joint_readings, i + 1)

    J_com_i = jacobian_cache[i + 1]
    m_i = self.mass[i]
    F_i = np.concatenate((m_i * grav, np.zeros(3)))  # [Fx, Fy, Fz, 0, 0, 0]^T
    tau_i = J_com_i.T @ F_i
    g += tau_i
```

3. **Hypothesis**

Huygens-Steiner theorem is a simple way to calculate the moment of inertia (I) of a rigid body about an axis that is not through its center of mass (CoM). If the moment of inertia about the CoM (I_{cm}) is known, you can use the formula:

$$I = I_{\text{cm}} + m \cdot d^2$$

Here:
- I: Moment of inertia about the new axis.
- I_{cm}: Moment of inertia about the CoM axis.
- m: Total mass of the body.
- d: Perpendicular distance between the CoM axis and the new axis.

This formula shows that as the distance d between the axes increases, the moment of inertia also increases, with $m \cdot d^2$ accounting for the ”offset effect.”

Derivation

The moment of inertia is defined as:

$$I = \int r^2 \, dm$$

where r is the perpendicular distance from a small mass element dm to the axis.

Now, let's say the new axis is d units away from the CoM axis. The distance r to the new axis can be split into:

$$r = r_{\text{cm}} + d$$

r_{cm} is the distance from dm to the CoM axis and d is the distance between the axes.

Substitute $r = r_{\text{cm}} + d$ into the inertia formula:

$$I = \int (r_{\text{cm}} + d)^2 \, dm$$

Expanding this gives:

$$I = \int (r_{\text{cm}}^2 + 2r_{\text{cm}}d + d^2) \, dm$$

Simplify the terms:
1. The first term $\int r_{\text{cm}}^2 \, dm$ is simply I_{cm}, the moment of inertia about the CoM axis.
2. The second term $\int 2r_{\text{cm}}d \, dm$ vanishes because the CoM balances out, making $\int r_{\text{cm}} \, dm = 0$.
3. The third term $\int d^2 \, dm = m \cdot d^2$, where m is the total mass.

So the final result is:

$$I = I_{\text{cm}} + m \cdot d^2$$

Importance in Robotics

This theorem is particularly useful in robotics because robots are made up of rotating links, and the axis of rotation often doesn't pass through the CoM of the links. Here's why it's so important:

1. Link Dynamics: Robot links rarely rotate around their CoM. For example, a joint might rotate a link at one end. Using this theorem, we can quickly calculate the moment of inertia about the actual axis of rotation.

2. Torque Calculations: To control robot joints, we need precise torque values, which depend heavily on the moment of inertia. Without the correct inertia, the robot might either overshoot its target or move sluggishly.

3. Energy Optimization: Efficient robot motion often involves minimizing unnecessary rotations around high-inertia axes. With accurate inertia values, we can optimize movement and save energy.

CONTINUED

In summary, the Huygens-Steiner theorem is a quick and reliable way to shift inertia calculations between axes, making it an essential tool for robotics. It simplifies calculations and ensures accurate modeling for control, planning, and energy efficiency.

4. **Forward Dynamics**

Forward dynamics involves determining the joint accelerations ($\ddot{q}$) of a robot given the applied joint torques (τ) and external forces. It answers the question: *"What motion results from these forces?"* The dynamics are governed by the equation:

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q)$$

This equation expresses how the torques (τ) applied to the robot are distributed among three main components: the inertial effects ($M(q)\ddot{q}$), the Coriolis and centrifugal forces ($C(q, \dot{q})\dot{q}$), and the gravitational effects ($G(q)$).

The main application of forward dynamics is in simulating robot motion. It is commonly used in:

- *Robot Simulation*: To predict how a robot will behave under certain forces and torques.
- *Trajectory Validation*: To verify whether a planned trajectory is physically feasible.
- *Training Controllers*: Forward dynamics provides the physical basis for training advanced controllers like reinforcement learning algorithms.

However, forward dynamics has its challenges:

- Solving the equation requires inverting the inertia matrix $M(q)$, which can be computationally expensive, especially for robots with many degrees of freedom.
- The equations are nonlinear and coupled, making real-time computation difficult.
- Accurate modeling of $M(q)$, $C(q, \dot{q})$, and $G(q)$ is essential, but in practice, obtaining precise parameters can be challenging due to friction, backlash, and other real-world uncertainties.

Inverse Dynamics

Inverse dynamics determines the joint torques (τ) required to produce a desired joint motion ($\ddot{q}$, $\dot{q}$, and q). It answers the question: *"What forces and torques are needed for this motion?"* The governing equation is the same:

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q)$$

This equation shows how the total torque (τ) required to achieve a specific motion is calculated by summing the contributions from inertial effects ($M(q)\ddot{q}$), Coriolis and centrifugal forces ($C(q, \dot{q})\dot{q}$), and gravitational effects ($G(q)$).

Inverse dynamics is widely used in robotics for:

- *Control*: Calculating the required torques to follow a specific trajectory (e.g., PID or model-based control).
- *Motion Planning*: Generating the necessary forces to execute a planned path.
- *Dynamic Compensation*: Accounting for gravitational, Coriolis, and inertial effects to improve motion accuracy.

The challenges of inverse dynamics include:

- It requires precise knowledge of the robot's dynamic parameters ($M(q)$, $C(q, \dot{q})$, $G(q)$), and errors in these parameters lead to incorrect torque predictions.
- Modeling external forces and disturbances accurately is difficult, especially in unstructured environments.
- For robots with many degrees of freedom, the computation becomes complex and time-sensitive in real-world control scenarios.

Comparison of Forward and Inverse Dynamics

- *Forward Dynamics* starts with forces (τ) and computes accelerations ($\ddot{q}$), whereas *Inverse Dynamics* starts with desired accelerations ($\ddot{q}$) and computes the forces (τ).
- Forward dynamics is primarily used for simulation, while inverse dynamics is central to control and motion planning.
- Both require accurate dynamic models, but forward dynamics often demands real-time computational efficiency for prediction, while inverse dynamics focuses on ensuring torque precision.

In summary, forward dynamics predicts motion from forces, while inverse dynamics determines forces for a given motion. Both are essential tools in robotics for understanding, simulating, and controlling robot behavior, though each comes with its computational and modeling challenges.

5. a. The bag file `cw3q5.bag` contains **1 message** of type `trajectory_msgs/JointTrajectory`, published on the topic `/iiwa/EffortJointInterface_trajectory_controller/command`.

   The message includes the following fields:

- joint_names: A list of joint names, such as ["iiwa_joint_1", ..., "iiwa_joint_7"].
- points: A sequence of trajectory points, each represented as a JointTrajectoryPoint object, containing:
  - positions: Joint positions at a specific trajectory point.
  - velocities: Joint velocities at the same trajectory point.
  - accelerations: Joint accelerations at the trajectory point.

The function _create_trajectory_from_bag is used to load and process the bag file. Below is the relevant code snippet:

```
def _create_trajectory_from_bag(self):
    traj_msg = JointTrajectory()
    traj_msg.joint_names = [
        "iiwa_joint_1", "iiwa_joint_2", "iiwa_joint_3",
        "iiwa_joint_4", "iiwa_joint_5", "iiwa_joint_6", "
           ↪ iiwa_joint_7"
    ]

    bag_file_path = os.path.join(self.pkg_path, "bag", "
       ↪ cw3q5.bag")
    try:
        with rosbag.Bag(bag_file_path, 'r') as bag:
            for topic, msg, t in bag.read_messages():
                for point in msg.points:
                    new_point = JointTrajectoryPoint()
                    new_point.positions = point.positions
                    new_point.velocities = point.velocities
                    new_point.accelerations = point.
                       ↪ accelerations
                    new_point.time_from_start = rospy.
                       ↪ Duration(
                        10 * (len(traj_msg.points) + 1))
                    traj_msg.points.append(new_point)

            rospy.loginfo("Bag␣file␣read␣successfully.␣Got␣%
               ↪ d␣trajectory␣points.",
                          len(traj_msg.points))
    except Exception as e:
        rospy.logerr("Error␣reading␣bag␣file:␣%s", e)
        return traj_msg

    traj_msg.header.stamp = rospy.Time.now()
    return traj_msg
```

This function:

- Loads the bag file and reads messages from the topic /iiwa/EffortJointInterface_trajectory_controller/command

- Extracts joint names, trajectory points, and corresponding positions, velocities, and accelerations.
- Assigns the extracted data to a `JointTrajectory` object for further processing or simulation.

b. This problem is a **forward dynamics** problem. We are given joint torques (τ) and we need to compute the resulting joint accelerations ($\ddot{q}$). In the equation:

$$\tau = M(q)\ddot{q} + C(q,\dot{q})\,\dot{q} + G(q),$$

we isolate $\ddot{q}$ as:

$$\ddot{q} = M(q)^{-1}\big[\tau - \big(C(q,\dot{q})\,\dot{q} + G(q)\big)\big].$$

Hence, given τ, we solve for $\ddot{q}$, which is the proof of forward dynamics.

c. The trajectory from the bag file `cw3q5.bag` is published to the topic `/iiwa/EffortJointInterface_trajectory_controller/command` to observe the robot's motion in simulation.

To achieve this, the function `_create_trajectory_from_bag` is used to load the trajectory from the bag file, and the `publish()` method of the ROS publisher is used to send the trajectory to the appropriate topic. Below is the relevant code snippet:

```python
def run(self):
    rospy.loginfo("Waiting␣for␣environment␣setup...")
    rospy.sleep(2.0)  # Allow time for environment to initialize

    # Load trajectory from the bag file
    trajectory = self._create_trajectory_from_bag()

    # Publish the trajectory to the topic
    self.trajectory_pub.publish(trajectory)
    rospy.loginfo("Trajectory␣published.")

    rospy.spin()  # Keep the node alive
```

This function:

- Waits for the environment to initialize using `rospy.sleep()`.
- Loads the trajectory from the bag file using `_create_trajectory_from_bag`.
- Publishes the trajectory to the topic `/iiwa/EffortJointInterface_trajectory_controller/command` using the ROS publisher.

- Keeps the node alive with `rospy.spin()` to allow the simulation to execute the trajectory.

Publishing the trajectory to the specified topic enables the robot's motion to be visualized in the simulation.

d. To calculate the joint accelerations throughout the trajectory using dynamic components, the function `_on_joint_state` subscribes to the topic `/iiwa/joint_states` and computes accelerations for each joint using the dynamic model. Below is the explanation and relevant code snippet:

```python
def _on_joint_state(self, joint_state_msg):
    curr_time = rospy.get_time()
    positions = joint_state_msg.position
    velocities = joint_state_msg.velocity
    efforts = joint_state_msg.effort

    # Compute joint accelerations
    accelerations = self._compute_acceleration(positions, velocities, efforts)

    # Log time and acceleration data to CSV files
    self.time_writer.writerow([curr_time])
    self.accel_writer.writerow(accelerations.flatten().tolist())
```

This function:

- Subscribes to the topic `/iiwa/joint_states` to receive joint state messages, which include positions, velocities, and efforts.
- Computes joint accelerations using the dynamic model via the `_compute_acceleration` method.
- Logs the computed accelerations and timestamps to CSV files for later analysis.

The dynamic model used for acceleration computation is based on the equation for forward dynamics:

$$\ddot{q} = B(q)^{-1} \left[\tau - (C(q, \dot{q})\dot{q} + G(q)) \right]$$

Where:

- $\ddot{q}$: Joint accelerations (to be computed)
- $B(q)$: Joint space mass/inertia matrix
- τ: Joint torques (from sensor input)

- $C(q,\dot{q})\dot{q}$: Coriolis and centrifugal forces
- $G(q)$: Gravitational torques

The implementation of this computation is shown below:

```
def _compute_acceleration(self, q, qdot, tau):
    # Forward dynamics equation: ddq = inv(B) @ (tau - (
        ↪ Cq_dot + G_vec))
    B_matrix = np.array(self.kdl_model.get_B(q))
    Cq_dot = np.array(self.kdl_model.get_C_times_qdot(q,
        ↪ qdot))
    G_vec = np.array(self.kdl_model.get_G(q))

    # Calculate joint accelerations
    ddq = inv(B_matrix).dot(np.array(tau) - Cq_dot - G_vec)
    return ddq.reshape(-1, 1)
```

This method:

- Retrieves the mass matrix $B(q)$, Coriolis forces $C(q,\dot{q})\dot{q}$, and gravity vector $G(q)$ from the KDL dynamic model.
- Solves the forward dynamics equation:

$$\ddot{q} = B(q)^{-1}\left[\tau - (C(q,\dot{q})\dot{q} + G(q))\right]$$

- Returns the accelerations as a column vector.

This process allows joint accelerations to be dynamically calculated for the entire trajectory.

e. The joint accelerations computed throughout the trajectory were plotted as a function of time. Each curve represents the acceleration of a specific joint over the entire trajectory.

The plotting function reads the logged time and acceleration data from the CSV files generated during the trajectory execution. Below is the relevant code snippet used to create the plot:

```
def _plot_accelerations(self):
    time_log_path = os.path.join(self.csv_dir, ’time_log.csv
        ↪ ’)
    accel_log_path = os.path.join(self.csv_dir, ’accel_log.
        ↪ csv’)

    times = []
    accel_data = [[] for _ in range(7)]  # Assuming 7 joints

    # Read time data
    with open(time_log_path, ’r’) as tfile:
        treader = csv.reader(tfile)
        for row in treader:
            if row:
                times.append(float(row[0]))

    # Read acceleration data
    with open(accel_log_path, ’r’) as afile:
        areader = csv.reader(afile)
        for row in areader:
            if len(row) == 7:  # Assuming 7 joints
                for j in range(7):
                    accel_data[j].append(float(row[j]))

    # Plot acceleration data
    plt.figure()
    for joint_idx in range(7):
        plt.plot(times, accel_data[joint_idx], label=f"Joint_{joint_idx+1}")
    plt.title("Joint␣Accelerations")
    plt.xlabel("Time")
    plt.ylabel("Acceleration")
    plt.legend()
    plt.grid(True)
    plt.show()
```

This function:

- Reads logged data for time and joint accelerations from CSV files.
- Plots each joint’s acceleration as a function of time.
- Displays the plot for visualization purposes.

Below is the resulting plot generated:

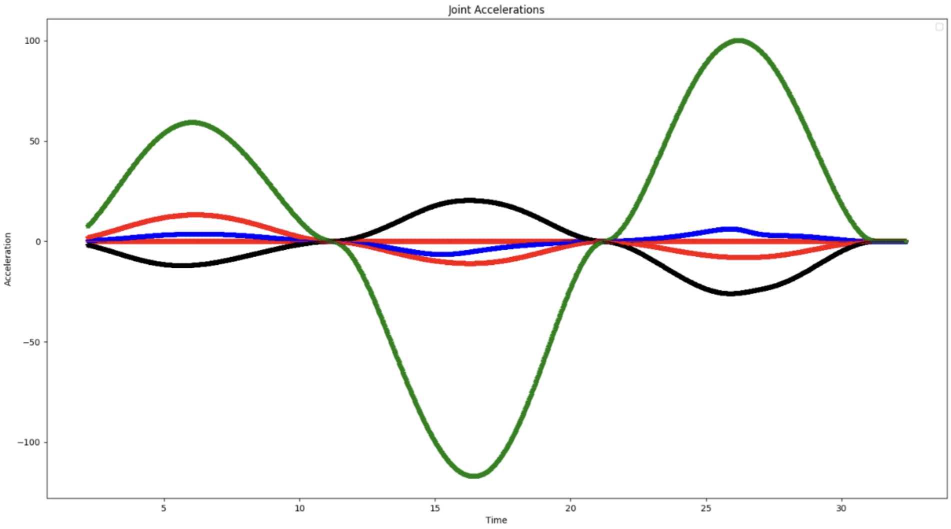

Figure 1: Joint Accelerations as a Function of Time

The plot clearly shows the acceleration variations of all joints over time, providing valuable insights into the robot's motion dynamics.

END OF COURSEWORK