Forward and Inverse Velocity Kinematics and Trajectory Tracking

For 7-DOF Manipulator using ROS2 & MuJoCo

1. Introduction

This report presents the theoretical background and implementation of **forward** and **inverse velocity kinematics** for a 7-DOF manipulator (KUKA iiwa14) simulated in **MuJoCo** and controlled via **ROS2**. The objective is to compute the end-effector velocity (forward kinematics) and the required joint velocities (inverse kinematics) for a desired motion. Additionally, basic methods for trajectory tracking are discussed.

2. Forward Velocity Kinematics

2.1. Concept

Forward velocity kinematics describes how joint velocities map to the end-effector's linear and angular velocities. Mathematically, the relationship is given by:

$$\dot{x} = \begin{bmatrix} v \\ \omega \end{bmatrix} = J(q)\,\dot{q} \tag{1}$$

Where:

- $J(q) \in \mathbb{R}^{6\times n}$ is the **Jacobian matrix**,
- $v \in \mathbb{R}^3$ is the linear velocity of the end-effector,
- $\omega \in \mathbb{R}^3$ is the angular velocity,
- $\dot{q} \in \mathbb{R}^n$ is the vector of joint velocities.

2.2. Implementation Steps

1. **Load the MuJoCo model:**

```
self.model = mujoco.MjModel.from_xml_path(model_path)
self.data = mujoco.MjData(self.model)
```

2. **Read joint positions and velocities:**

```
q = np.array([float(a) for a in angles_str.split(",")])
qdot = np.array([float(v) for v in qdot_str.split(",")])
```

3. **Update the simulation state:**

```
for i in range(7):
    self.data.qpos[i] = q[i]
    self.data.qvel[i] = qdot[i]
mujoco.mj_forward(self.model, self.data)
```

4. **Compute the Jacobian for the end-effector:**

```
jacp, jacr = self.get_jacobian('link7')
```

5. **Calculate end-effector velocities:**

$$v = J_p(q)\,\dot{q} \tag{2}$$
$$\omega = J_r(q)\,\dot{q} \tag{3}$$

```
v_linear = jacp @ qdot
v_angular = jacr @ qdot
```

3. Inverse Velocity Kinematics

3.1. Concept

Inverse velocity kinematics determines the joint velocities $\dot{q}$ required to achieve a desired end-effector velocity $\dot{x}_{des}$:

$$\dot{q} = J^{+}(q)\,\dot{x}_{des} \tag{4}$$

Where J^{+} is the **pseudo-inverse** of the Jacobian matrix:

$$J^{+} = J^{T}(JJ^{T})^{-1} \tag{5}$$

This formulation is used when J is not square or is singular.

3.2. Implementation Steps

1. **Read the joint configuration:**

```
q = np.array([float(a) for a in angles_str.split(",")])
```

2. **Compute Jacobian:**

```
jacp, jacr = self.get_jacobian('link7')
J = np.vstack((jacp, jacr))
```

3. **Read desired end-effector velocity:**

```
v_linear = np.array([float(v) for v in v_lin_str.split(",")])
v_angular = np.array([float(w) for w in v_ang_str.split(",")])
xdot = np.hstack((v_linear, v_angular))
```

4. **Compute joint velocities using pseudo-inverse:**

```
qdot = np.linalg.pinv(J) @ xdot
```

5. **Publish joint commands to simulator:**

```
msg = Float64MultiArray()
msg.data = qdot.tolist()
self.joint_pub.publish(msg)
```

4. Trajectory Tracking Methods

Trajectory tracking aims to control the robot so that the end-effector follows a desired path over time.

4.1. Joint-Space Tracking

A reference trajectory is defined in the joint space as:

$$q_d(t),\ \dot{q}_d(t),\ \ddot{q}_d(t)$$

A simple velocity control law can be:

$$\dot{q} = \dot{q}_d + K_p(q_d - q) \tag{6}$$

where K_p is a diagonal gain matrix.

4.2. Task-Space Tracking

In task space, the control aims to follow a desired end-effector trajectory $x_d(t)$. The velocity-level control law is:

$$\dot{q} = J^{+}(q)[\dot{x}_d + K_p(x_d - x)] \tag{7}$$

To avoid singularities, a **Damped Least Squares (DLS)** inverse can be used:

$$J^{+} = J^T(JJ^T + \lambda^2 I)^{-1} \tag{8}$$

where $\lambda > 0$ is the damping factor.

4.3. Acceleration-Level Control (Optional)

For acceleration-based control:

$$\ddot{x} = J(q)\ddot{q} + \dot{J}(q, \dot{q})\dot{q} \tag{9}$$

which can be inverted to obtain $\ddot{q}$, then integrated to get velocity and position commands.

5. Simulation Workflow

1. Launch the ROS2 node:

   ```
   ros2 run mujoco_ros2 fk_ik_velocity_node
   ```

2. Choose mode from terminal:

   - `fk_vel` – Compute end-effector velocity from $(q, \dot{q})$
   - `ik_vel` – Compute required $\dot{q}$ for desired end-effector motion
   - `quit` – Exit the node

3. The results are printed and published to the `/joint_commands` topic.

6. Conclusion

This work demonstrates a ROS2-MuJoCo pipeline for performing velocity-level kinematics on a 7-DOF manipulator. Forward velocity kinematics computes end-effector motion from joint velocities, while inverse velocity kinematics uses the pseudo-inverse of the Jacobian to find required joint velocities for a desired end-effector trajectory. The same framework can be extended to acceleration-level control or combined with feedback control for precise trajectory tracking.