Trajectory

Peter Jan Sincak

April 2025

Exercise 1: Joint Space Trajectory Planning Using Cubic Interpolation

Overview

In joint space trajectory planning, the goal is to compute how each joint of a robot should move over time to achieve a desired end configuration. The trajectory is defined as a time-varying vector:

$$\mathbf{q}(t) = [q_1(t), q_2(t), \ldots, q_n(t)]$$

where $q_i(t)$ is the position of joint i at time t, and n is the number of joints.

Problem Setup

Given:

- Initial joint configuration: $\mathbf{q}_0$
- Final joint configuration: $\mathbf{q}_f$
- Total duration of the motion: T

We wish to generate a smooth trajectory $\mathbf{q}(t)$ from $\mathbf{q}_0$ to $\mathbf{q}_f$, ensuring continuity in position and velocity.

Cubic Interpolation

Each joint trajectory is represented by a cubic polynomial:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

We impose the following boundary conditions:

$$q(0) = q_0, \quad q(T) = q_f, \quad \dot{q}(0) = 0, \quad \dot{q}(T) = 0$$

These four constraints are sufficient to solve for the four coefficients of the cubic polynomial.

Alternatively, we can use a normalised time parameter $\tau = \frac{t}{T} \in [0, 1]$, and apply the following cubic interpolation formula:

$$q(t) = (2\tau^3 - 3\tau^2 + 1)q_0 + (-2\tau^3 + 3\tau^2)q_f$$

This formulation guarantees zero velocity at the start and end of the motion, resulting in smooth transitions.

Simulation Process

1. Load the KUKA iiwa robot in PyBullet.
2. Define the initial and final joint configurations $\mathbf{q}_0$ and $\mathbf{q}_f$.
3. Generate a joint trajectory using cubic interpolation over a duration T.
4. At each time step, send joint positions to the simulator and record the end-effector pose using forward kinematics.
5. Visualise:
    - The joint trajectories $q_i(t)$ over time.
    - The path traced by the end-effector in 3D task space.

Expected Results

- **Joint Space Trajectories:** Each joint's angle $q_i(t)$ should follow a smooth S-shaped curve characteristic of cubic interpolation with zero start and end velocity.
- **Task Space Trajectory:** The end-effector path will typically appear curved and nonlinear in 3D space due to the robot's forward kinematics. It will not be a straight line unless Cartesian interpolation is explicitly performed.

Conclusion

Cubic interpolation in joint space is a simple and effective method for generating smooth robot motions. It is particularly useful for point-to-point movements where smoothness is desired and precise Cartesian paths are not required.

Exercise 2: Task Space Trajectory Planning Using Inverse Kinematics

Objective

The goal of this exercise is to move the robot's end-effector along a desired trajectory in **task space** (Cartesian space), using inverse kinematics (IK) to compute the corresponding joint configurations. The trajectory is defined as a straight line between a start and goal position.

Task Space vs. Joint Space

- **Joint space:** A configuration is represented by a vector of joint angles $\mathbf{q}(t) = [q_1(t), q_2(t), \ldots, q_n(t)]$.
- **Task space:** A configuration is defined by the position and orientation of the end-effector $\mathbf{x}(t) = [x(t), y(t), z(t)]$.

In this exercise, the trajectory is specified in task space, and joint angles are obtained via inverse kinematics.

Linear Interpolation in Task Space

To move the end-effector along a straight line from a starting point $\mathbf{x}_0$ to a goal point $\mathbf{x}_f$, we interpolate linearly over a time interval $[0, T]$:

$$\mathbf{x}(t) = (1 - \tau)\,\mathbf{x}_0 + \tau\,\mathbf{x}_f, \quad \text{where } \tau = \frac{t}{T}$$

This provides a smooth set of Cartesian waypoints at each time step.

Inverse Kinematics (IK)

For each Cartesian waypoint $\mathbf{x}(t_i)$, the inverse kinematics algorithm is used to compute joint angles $\mathbf{q}(t_i)$ such that:

$$f_{\text{FK}}(\mathbf{q}(t_i)) \approx \mathbf{x}(t_i)$$

where f_{FK} is the forward kinematics function of the robot. The resulting joint angles are then sent to the simulator using direct joint control (e.g., `resetJointState` in PyBullet).

Tracking Error

Due to numerical approximation and redundancy in the robot kinematics, the actual end-effector path $\mathbf{x}_{\text{actual}}(t)$ may differ from the desired path $\mathbf{x}_{\text{desired}}(t)$. We define the Euclidean tracking error as:

$$e(t) = \|\mathbf{x}_{\text{desired}}(t) - \mathbf{x}_{\text{actual}}(t)\|$$

This error can be visualised over time to assess the accuracy of the motion.

Simulation Process

1. Define the start and goal positions in Cartesian space.
2. Linearly interpolate between them to generate task space waypoints.
3. For each waypoint:
    - Compute the joint configuration using inverse kinematics.
    - Apply the joint angles to the robot.
    - Record the actual end-effector position.
4. Plot:
    - The desired vs. actual end-effector path in 3D.
    - The joint angles over time.
    - The tracking error.

Expected Results

- **Task Space Path:** The end-effector should approximately follow a straight line between the start and goal positions.
- **Joint Trajectories:** The joint angles will vary non-linearly, depending on the robot's kinematics.
- **Tracking Error:** The Euclidean distance between the desired and actual end-effector positions should remain small, except possibly for an initial spike if the robot does not start at the intended position.