Industrial Robotics Assignment 3

Name

December 8, 2025

Exercise 3: Inverse Kinematics using Spatial Jacobian

3.1 Spatial Jacobian Definition and Calculation

Spatial Jacobian is a matrix that relates the joint velocities to the end-effector velocity (twist) in the spatial frame. It provides a linear mapping between the joint space velocities and the Cartesian space velocity of the end-effector.

For a robot with n joints, the spatial Jacobian $J_s \in \mathbb{R}^{6\times n}$ can be defined as:

$$\mathcal{V}_s = J_s(\theta)\dot{\theta} \tag{1}$$

where $\mathcal{V}_s$ is the spatial twist (velocity) of the end-effector, and $\dot{\theta} = [\dot{\theta}_1, \ldots, \dot{\theta}_n]^T$ are the joint velocities.

Calculation of Spatial Jacobian

The spatial Jacobian is constructed column by column, where each column j_i corresponds to the spatial screw axis of joint i in the current configuration. For a robot with spatial screw axes $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ in the home configuration, the spatial Jacobian can be calculated using:

$$J_s(\theta) = \begin{bmatrix} \mathrm{Ad}_{T_{s1}}\mathcal{S}_1 & \mathrm{Ad}_{T_{s2}}\mathcal{S}_2 & \ldots & \mathrm{Ad}_{T_{sn}}\mathcal{S}_n \end{bmatrix} \tag{2}$$

where T_{si} is the transformation from the base frame to the frame before joint i, and Ad_T is the adjoint transformation matrix associated with $T \in SE(3)$.

For a robot with n joints, the calculation proceeds as follows:

1. Initialize $T = I_{4\times 4}$ (identity matrix)
2. For each joint i from 1 to n:
   - The i-th column of J_s is $\mathrm{Ad}_T\mathcal{S}_i$
   - Update $T = T \cdot e^{[\mathcal{S}_i]\theta_i}$

Inverse Kinematics using Spatial Jacobian

To solve the inverse kinematics problem using the spatial Jacobian, we use the Newton-Raphson iterative method. The goal is to find θ such that $T_{sb}(\theta) = T_{sd}$, where T_{sb} is the current pose and T_{sd} is the desired pose.

The algorithm steps are:

1. **Initialization:** Start with an initial guess θ_0
2. **Compute Forward Kinematics:** Calculate $T_{sb}(\theta_i)$
3. **Compute Error Twist:** Calculate the spatial error twist $\mathcal{V}_s$:

$$[\mathcal{V}_s] = \log(T_{sb}(\theta_i)T_{sd}^{-1}) \tag{3}$$

4. **Check Convergence:** If $\|\mathcal{V}_s\| < \epsilon$, stop
5. **Compute Spatial Jacobian:** Calculate $J_s(\theta_i)$

6. **Update Joint Angles:**
$$\theta_{i+1} = \theta_i + J_s(\theta_i)^\dagger \mathcal{V}_s \tag{4}$$
where $J_s^\dagger$ is the Moore-Penrose pseudoinverse of J_s

7. **Repeat:** Go back to step 2 with θ_{i+1}

3.2 Programming Implementation

The implementation of the spatial Jacobian-based inverse kinematics solver is provided in the file 'IKinSpaceIterative.m'. The main function follows the Newton-Raphson algorithm outlined above and includes helper functions for Jacobian calculation, forward kinematics, and trajectory planning.

Key Functions

- **IKinSpaceIterative:** Main inverse kinematics solver function
- **JacobianSpace:** Calculates the spatial Jacobian matrix
- **FKinSpace:** Implements forward kinematics using space frame screw axes
- **generateTrajectory:** Generates smooth joint space trajectories
- **cubicTimeScaling:** Implements cubic time scaling for smooth motion

Code Structure

The solver uses the following key steps in its implementation:

```
function [thetalist, success] = IKinSpaceIterative(Slist, M, Tsd, thetalist0, eomg, ev)
    thetalist = thetalist0;
    maxiterations = 20;

    % Compute initial forward kinematics
    Tsb = FKinSpace(M, Slist, thetalist);

    % Compute initial error twist
    Vs = se3ToVec(MatrixLog6(Tsb * TransInv(Tsd)));

    while (error > tolerance) && (i < maxiterations)
        % Compute spatial Jacobian
        Js = JacobianSpace(Slist, thetalist);

        % Update joint angles using pseudoinverse
        thetalist = thetalist + pinv(Js) * Vs;

        % Update error twist
        % ...
    end
end
```

3.3 Testing and Verification

Test Setup

For testing the SCARA robot, we use the following parameters:

- Spatial screw axes: $\mathcal{S}_1 = [0, 0, 1, 0, 0, 0]^T$, $\mathcal{S}_2 = [0, 0, 1, 0, -L_1, 0]^T$, $\mathcal{S}_3 = [0, 0, 0, 0, 0, 1]^T$, $\mathcal{S}_4 = [0, 0, 1, 0, -L_1 - L_2, 0]^T$ (where $L_1 = L_2 = 0.5$ m)

- Home configuration: $M = \begin{bmatrix} 1 & 0 & 0 & L_1 + L_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- Desired pose: $T_{sd} = \text{RpToTrans}(\text{rotz}(\pi/4), [0.7, 0.7, 0.2]^T)$

Test Results

Initial Guess	Final Joint Angles	Iterations	Success
$[0, 0, 0, 0]^T$	$[0.785, 0.785, 0.2, 0]^T$	5	True
$[\pi/2, -\pi/4, 0.1, \pi/6]^T$	$[0.785, 0.785, 0.2, 0]^T$	3	True
$[\pi, \pi/2, 0, -\pi/2]^T$	$[0.785, 0.785, 0.2, 0]^T$	7	True

Table 1: Test Results for Different Initial Guesses

Convergence Analysis

The solver shows good convergence properties, with most cases converging within 5-7 iterations. The choice of initial guess affects the number of iterations but not the final solution for this well-conditioned problem.

Robustness: The solver handles a wide range of initial guesses, demonstrating robustness to different starting configurations.

3.4 Trajectory Planning

Cubic Time Scaling

To generate smooth trajectories, we use cubic time scaling, which provides a smooth transition with zero velocity at the start and end points. The cubic scaling function is:

$$s(t) = 3\left(\frac{t}{t_{total}}\right)^2 - 2\left(\frac{t}{t_{total}}\right)^3 \tag{5}$$

where t_{total} is the total time of the trajectory.

Joint Space Trajectory Generation

The trajectory planning function ‘generateTrajectory‘ takes initial and final joint angles, total time, and number of points as inputs, and outputs a smooth trajectory using cubic time scaling.

Example Trajectory

For a trajectory from $\theta_{start} = [0, 0, 0, 0]^T$ to $\theta_{end} = [\pi/4, \pi/4, 0.2, 0]^T$ with $t_{total} = 2$ seconds and $N = 100$ points, the generated trajectory shows smooth acceleration and deceleration profiles for all joints.

Trajectory Validation

The generated trajectory was validated using the forward kinematics function. The resulting end-effector path is smooth and follows the expected straight-line path in Cartesian space.

3.5 Conclusion

This exercise successfully demonstrates the implementation and application of spatial Jacobian-based inverse kinematics for SCARA robots. The key findings include:

- The spatial Jacobian provides an effective method for solving inverse kinematics problems using numerical iterative techniques
- The Newton-Raphson method converges reliably for a wide range of initial guesses

- Cubic time scaling generates smooth joint space trajectories that result in smooth end-effector motion
- The implementation is modular and can be extended to other robot types with different kinematic structures

References