Code Report: 7-DOF Robot Inverse Kinematics Solver

July 12, 2025

Contents

1 Project Overview

This project implements an Inverse Kinematics (IK) solver for a 7-degree-of-freedom (DOF) robotic system. The system is composed of a 6-DOF robotic arm (ABB IRB 6700) mounted on a linear axis. The primary goal of the solver is to determine the joint configurations required to position the robot's end-effector at a specified target position and orientation in 3D space.

The project leverages the PyBullet physics engine for simulation and visualization, allowing for a realistic depiction of the robot's movement. It also includes a URDF parser to extract kinematic and dynamic properties of the robot from its URDF file.

Key Features

- **7-DOF Inverse Kinematics:** The core of the project is the IK solver that finds solutions for the combined 7-DOF system.
- **URDF Parsing:** The code includes a parser for URDF files to extract joint limits, link origins, and other essential robot parameters.
- **PyBullet Visualization:** A visualizer class provides a real-time 3D simulation of the robot's motion.
- **End-Effector Trajectory Plotting:** The system plots the trajectory of the end-effector and saves it as a PNG image.
- **Modular Design:** The code is well-structured into classes for the IK solver, URDF parser, and visualizer, promoting modularity and reusability.

2 Code Structure

The project is organized into the following key files:

- `main.py`: The main entry point of the application. It handles command-line argument parsing for the target pose, initializes the IK solver and visualizer, and orchestrates the overall workflow.
- `src/ik_solver.py`: Contains the `IKSolver7DOF` class, which implements the inverse kinematics algorithm.
- `src/urdf_parser.py`: Implements the `URDFParser` class, responsible for parsing URDF files.
- `src/visualizer.py`: Contains the `Visualizer` class, which manages the PyBullet simulation and trajectory plotting.
- `robot-urdfs/`: This directory contains the URDF files for the robotic arm and the linear axis.

2.1 Detailed Component Analysis

2.1.1 `main.py`

The `main.py` script serves as the orchestrator of the IK solving and visualization process. Its main responsibilities are:

- **Argument Parsing:** It uses the `argparse` module to accept the target end-effector position and orientation from the command line. If no arguments are provided, it uses default values.
- **Initialization:** It initializes the `Visualizer` and `IKSolver7DOF` classes.
- **Logging:** It configures logging to output information to both the console and a file (`robot_ik.log`).
- **IK Solving:** It calls the `inverse_kinematics` method of the `IKSolver7DOF` class to find a set of possible solutions.

- **Solution Selection:** It selects the "best" solution from the returned list using the `find_best_solution` method, which considers both position and orientation errors.
- **Visualization:** It runs the PyBullet simulation to visualize the robot moving to the target pose, logging joint and linear axis positions at each iteration.
- **Trajectory Plotting:** After the simulation, it calls the `plot_end_effector_trajectory` method to generate a plot of the end-effector's path, including the start and target positions.

2.1.2 `src/urdf_parser.py` - `URDFParser` Class

The `URDFParser` class is responsible for reading and parsing URDF files. It uses the `xml.etree.ElementTree` module to parse the XML structure of the URDF file.

- `__init__(self, urdf_content)`: The constructor takes the URDF file content as a string and initializes the parsing process.
- `_parse_urdf(self)`: This private method iterates through the URDF structure and extracts information about joints and links, including their types, origins, axes, and limits.
- `get_joint_limits(self)`: This method returns a list of tuples, where each tuple contains the lower and upper limits of a joint.
- `get_link_origins(self)`: This method returns the origins of each link.

2.1.3 `src/visualizer.py` - `Visualizer` Class

The `Visualizer` class encapsulates all the logic related to the PyBullet simulation.

- `__init__(self, ...)`: The constructor initializes the PyBullet environment, loads the robot and linear axis from their URDF files, and sets up the simulation parameters.
- `_init_pybullet(self)`: This method handles the setup of the PyBullet environment, including gravity, camera position, and loading the plane and robot models. It also creates a constraint to attach the robot to the linear axis and changes the color of the robot and linear axis for better visualization.
- `visualize_pybullet(self, joint_angles, linear_pos)`: This method sets the joint positions of the robot and the linear axis in the simulation and steps the simulation forward. It also records the end-effector's position at each step.
- `plot_end_effector_trajectory(self)`: This method uses `matplotlib` to create a 3D plot of the end-effector's trajectory and saves it to a file. The plot is detailed, with labels, a title, a grid, a legend, and an equal aspect ratio.
- `close_pybullet(self)`: This method disconnects from the PyBullet simulation.

2.1.4 `src/ik_solver.py` - `IKSolver7DOF` Class

This is the core component of the project, responsible for solving the inverse kinematics problem.

- `__init__(self, ...)`: The constructor initializes the solver with the URDF paths and the visualizer instance. It also extracts robot parameters using the `URDFParser`.
- `_extract_robot_parameters(self)`: This method retrieves the joint limits for the robot arm and the movement range of the linear axis.
- `inverse_kinematics(self, ...)`: This method solves the inverse kinematics for the 7-DOF system by iterating over the linear axis. It discretizes the linear axis into 20 points.
- `find_best_solution(self, ...)`: This method takes a list of solutions and finds the one that minimizes a combined position and orientation error.

3 Inverse Kinematics Solution

The inverse kinematics problem for a 7-DOF redundant manipulator is challenging because there are infinitely many solutions for a given end-effector pose. The implemented solution in `IKSolver7DOF.inverse_kinematics` addresses this by simplifying the problem. Instead of a purely analytical or complex numerical optimization approach, it uses a **discretized search over the redundant degree of freedom combined with a numerical IK solver**.

Here's a step-by-step breakdown of the method:

1. **Discretization of the Redundant Joint:** The 7th DOF is the linear axis. The solver discretizes the continuous range of the linear axis into a set of discrete points. In the current implementation, it samples 20 points along the linear axis's range.

```
for linear_pos in np.linspace(self.linear_range[0], self.linear_range[1], 20):

```

2. **Reducing to a 6-DOF Problem:** For each discrete position of the linear axis, the base of the 6-DOF robot arm is fixed at a specific location in the world frame. This effectively reduces the problem to a standard 6-DOF inverse kinematics problem for the arm.

3. **Coordinate Frame Transformation:** The target position of the end-effector is given in the world frame. To solve the 6-DOF IK for the arm, the target position needs to be transformed into the robot's base frame. This is done by subtracting the robot's base position (which is determined by the `linear_pos`) from the world target position.

```
robot_base_pos = np.array([linear_pos, 0, 0]) + self.linear_offset
target_pos_in_robot_frame = target_pos - robot_base_pos

```

4. **Numerical 6-DOF IK Solver:** With the problem reduced to 6-DOF, the code uses PyBullet's built-in numerical IK solver, `calculateInverseKinematics`. This function uses a Damped Least Squares (DLS) method, which is an iterative approach to find a joint configuration that minimizes the error between the current and target end-effector pose.

```
joint_angles = self.visualizer.p.calculateInverseKinematics(...)

```

5. **Solution Validation:** The numerical solver in PyBullet does not inherently respect joint limits. Therefore, after a potential solution is found, the code checks if the calculated joint angles are within the specified limits for each joint.

```
valid_solution = True
for i, angle in enumerate(joint_angles[:len(self.joint_limits)]):
    lower, upper = self.joint_limits[i]
    if not (lower <= angle <= upper):
        valid_solution = False
        break

```

6. **Aggregation of Solutions:** If a valid solution is found for a particular `linear_pos`, the combination of the 6-DOF joint angles and the `linear_pos` is stored as a valid 7-DOF solution.

```
if valid_solution:
    solutions.append((list(joint_angles[:len(self.visualizer.joint_indices)]),
    linear_pos))

```

This process is repeated for all the discretized points of the linear axis, resulting in a list of potential solutions.

4 Selecting the Optimal Solution

The `inverse_kinematics` function can return multiple valid solutions. The `find_best_solution` method is responsible for evaluating this list of candidates and selecting the optimal one based on accuracy. This is crucial because numerical solvers may converge to solutions that are valid but not perfectly accurate. The selection process is as follows:

1. **Forward Kinematics Verification:** For each candidate solution (a set of joint angles and a linear position), the method first computes the *actual* end-effector pose using forward kinematics. This is done by setting the robot's state in the simulation and querying the link state of the end-effector. This step provides the ground truth for how accurate the candidate solution really is.

```
self.visualizer.set_robot_state(joint_angles, robot_base_pos)
ee_state = self.visualizer.p.getLinkState(
    self.visualizer.robot_id,
    self.visualizer.end_effector_link_index
)
ee_pos = np.array(ee_state[4])
ee_quat = np.array(ee_state[5])

```

2. **Error Calculation:** A composite error metric is calculated to quantify the deviation of the actual pose from the target pose.

   - **Position Error:** The Euclidean distance between the actual and target end-effector positions.

```
pos_error = np.linalg.norm(ee_pos - target_pos)

```

   - **Orientation Error:** The error between the actual and target orientations (represented as quaternions). This is calculated as $1-(q_{\text{actual}} \cdot q_{\text{target}})^2$. The dot product of two quaternions measures their similarity; squaring it handles the ambiguity where a quaternion q and its negative $-q$ represent the same orientation.

```
dot = np.dot(ee_quat, target_quat)
orn_error = 1.0 - dot**2

```

   - **Total Weighted Error:** The position and orientation errors are combined into a single score using tunable weights. This allows for prioritizing positional accuracy over orientational accuracy, or vice versa.

```
total_error = pos_weight * pos_error + orn_weight * orn_error

```

3. **Iterative Comparison:** The method iterates through all candidate solutions, calculating the total error for each. It keeps track of the solution that yields the minimum error found so far.

```
if total_error < min_error:
    min_error = total_error
    best_solution = (joint_angles, linear_pos)

```

4. **Tie-Breaking for Motion Smoothness:** In cases where two solutions have nearly identical errors, a tie-breaking rule is applied. If the robot's current joint state is provided, the method will prefer the solution that is "closer" to the current state. This distance is calculated as the sum of squared differences in joint angles and linear position. This heuristic promotes smoother, more efficient movements by minimizing large changes in the robot's configuration.

```python
if abs(total_error - min_error) < 1e-6 and current_joints is not None:
    joint_dist = np.sum((np.array(joint_angles) - np.array(current_joints))**2)
    linear_dist = (linear_pos - current_linear)**2
    # ... logic to select solution with smaller total_dist ...

```

By combining forward kinematics verification with a weighted error metric, this function robustly selects the solution that most accurately achieves the desired target pose from the list of candidates.

5 Results and Visualization

To validate the performance of the IK solver and the resulting motion, the trajectory of the end-effector was recorded and plotted. Figure 1 shows the path of the robot's end-effector as it moves from a specified initial position to a target position. This visualization is generated by the `plot_end_effector_trajectory` function, which captures the end-effector's world coordinates at each step of the simulation.

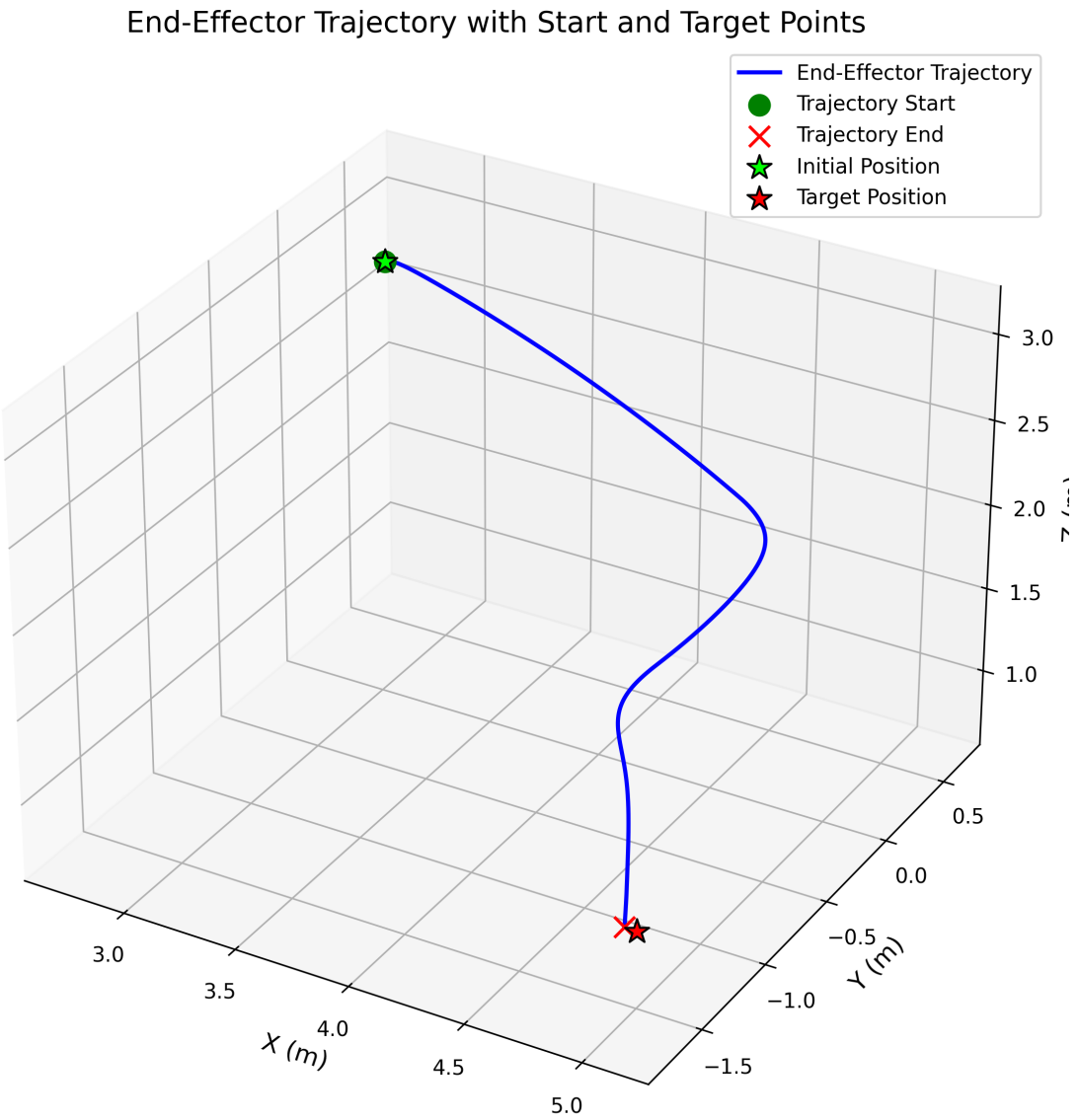

Figure 1: The 3D trajectory of the end-effector moving from the initial to the target pose. The plot visualizes the path generated by interpolating between the start and end joint configurations found by the IK solver.

The plot illustrates several key aspects of the system's performance:

- **Trajectory Path:** The solid blue line represents the continuous path traced by the end-effector in the world frame (in meters). The smooth, curved nature of the path indicates a coordinated motion across all seven degrees of freedom (the 6-axis arm and the linear track).
- **Initial and Start Points:** The green star (Initial Position) marks the desired starting pose for the motion. The green circle (Trajectory Start) marks the actual beginning of the plotted trajectory. The perfect overlap of these two points confirms that the robot was correctly positioned at the start pose found by the IK solver.
- **Target and End Points:** Similarly, the red star (Target Position) denotes the desired goal pose. The red 'x' (Trajectory End) marks where the end-effector actually stopped. The alignment of these markers demonstrates that the IK solver successfully found an accurate joint configuration to reach the target pose with high precision.

This visualization serves as a crucial verification step, confirming that the combination of the discretized IK search (`inverse_kinematics`) and the optimal solution selection (`find_best_solution`) results in accurate and achievable robot movements in 3D space.

A Damped Least Squares (DLS) Method for Inverse Kinematics: Comprehensive Analysis

A.1 Introduction

The Damped Least Squares (DLS) method, also known as the Levenberg-Marquardt method for inverse kinematics, is a numerical optimization technique designed to solve the inverse kinematics problem while maintaining stability near kinematic singularities. It represents a significant improvement over the traditional Newton-Raphson method by introducing a damping term that prevents numerical instabilities.

A.2 Mathematical Foundation

The Inverse Kinematics Problem

For a robotic manipulator, the forward kinematics establishes the relationship:

$$x = f(q)$$

Where:

- $x \in \mathbb{R}^m$: End-effector pose (position and orientation)
- $q \in \mathbb{R}^n$: Joint configuration vector
- $f(q)$: Forward kinematics function

The inverse kinematics problem seeks to find q given a desired x_d:

$$\text{Find } q \text{ such that } f(q) = x_d$$

Iterative Solution Framework

Since analytical solutions are rarely available for complex manipulators, iterative methods are used:

$$q_{k+1} = q_k + \Delta q_k$$

Where Δq_k is computed to minimize the error:

$$e_k = x_d - f(q_k)$$

A.3 Traditional Newton-Raphson Method

Basic Formulation

The Newton-Raphson method linearizes the kinematics around the current configuration:

$$f(q_k + \Delta q) \approx f(q_k) + J(q_k)\Delta q$$

Where $J(q_k)$ is the Jacobian matrix:

$$J(q_k) = \left.\frac{\partial f}{\partial q}\right|_{q=q_k}$$

Joint Update Rule

To minimize the error, we set:

$$x_d = f(q_k) + J(q_k)\Delta q_k$$

Solving for Δq_k:

$$\Delta q_k = J(q_k)^{-1}(x_d - f(q_k)) = J(q_k)^{-1}e_k$$

The Singularity Problem

Mathematical Issue: When the manipulator approaches a kinematic singularity, the Jacobian matrix becomes rank-deficient:

$$\det(J) \to 0$$

Consequences:

1. **Numerical Instability**: J^{-1} becomes ill-conditioned or undefined.
2. **Infinite Joint Velocities**: Small task-space errors lead to huge joint motions.
3. **Convergence Failure**: Algorithm may diverge or oscillate.

Physical Interpretation: Singularities are configurations where the manipulator loses one or more degrees of freedom (e.g., fully extended arm, wrist singularities).

A.4 Damped Least Squares Solution

Core Innovation

The DLS method introduces a damping term λ to regularize the inversion:

$$\Delta q_k = J^T(JJ^T + \lambda I)^{-1}e_k$$

Where:

- $\lambda > 0$: Damping factor
- I: Identity matrix
- $JJ^T + \lambda I$: Always invertible (positive definite)

Alternative Formulation

The DLS update can also be written as:

$$\Delta q_k = (J^T J + \lambda I)^{-1}J^T e_k$$

Both formulations are mathematically equivalent but have different computational properties.

A.5 Detailed Algorithm Analysis

Step-by-Step DLS Process

1. Initialize: q_0, λ, tolerance ε, max_iterations
2. For $k = 0, 1, 2, \ldots,$ max_iterations:
   (a) Compute current pose: $x_k = f(q_k)$
   (b) Compute error: $e_k = x_d - x_k$
   (c) Check convergence: if $||e_k|| < \varepsilon$, stop
   (d) Compute Jacobian: $J_k = \frac{\partial f}{\partial q}\Big|_{q=q_k}$
   (e) Compute damped update: $\Delta q_k = J_k^T(J_k J_k^T + \lambda I)^{-1}e_k$
   (f) Update joints: $q_{k+1} = q_k + \Delta q_k$
3. Return q_k or failure