UR3 Inverse Kinematics Implementation

Course: ME5250 - Robot Mechanics and Control

Author: Prathmesh Barapatre

1. Abstract

This project focuses on the kinematic modeling and control simulation of a 6-DOF Universal Robots UR3 manipulator. The objective was to develop a simulation environment from scratch using Python, without relying on high-level robotics packages (e.g., ROS, MoveIt, or Robotics Toolbox). The project implements analytical Forward Kinematics (FK) based on the Standard Denavit-Hartenberg (DH) convention and a numerical Inverse Kinematics (IK) solver using the Newton-Raphson method with Damped Least Squares (DLS). The system is visualized using a custom Matplotlib 3D interface, demonstrating the robot's capability to track parametric task-space trajectories (square, circle) and respond to interactive manual control.

2. Parametric Modeling and Validation

To ensure high-fidelity simulation, the geometric parameters were extracted from the official UR3 kinematic calibration data and mapped to the Standard Denavit-Hartenberg convention.

2.1 Parameter Selection and Link Lengths

The kinematic parameters were derived from the official UR3 technical specifications. The physical link lengths were extracted from the UR3 calibration data and mapped to the Denavit-Hartenberg (DH) convention.

- **Base Height (d_1)**: 0.1519 m (Distance from base mounting to shoulder axis).
- **Upper Arm (a_2)**: -0.24365 m (Length of the first link).
- **Forearm (a_3)**: -0.21325 m (Length of the second link).
- **Wrist Offsets (d_4, d_5, d_6)**: The distinct wrist complex of the UR series requires vertical offsets of 0.11235 m, 0.08535 m, and 0.0819 m respectively.

2.2 Denavit-Hartenberg (DH) Parameters

The simulation utilizes the Standard DH Convention. The transformation from frame $i-1$ to frame i is defined by four parameters:

1. Θ_i : Joint angle (variable).
2. d_i : Link offset (along $z_{\{i-1\}}$).
3. a_i : Link length (along x_i).
4. α_i : Link twist (angle between $z_{(i-1)}$, z_i , and x_i).

DH Parameters Used:

Joint	d (m)	a (m)	α (rad)	Θ offset	Description
1	0.1519	0	pi/2	0	Base to Shoulder
2	0	-0.24365	0	0	Shoulder to Elbow
3	0	-0.21325	0	0	Elbow to Wrist 1
4	0.11235	0	pi/2	0	Wrist 1 to Wrist 2
5	0.08535	0	-pi/2	0	Wrist 2 to Wrist 3
6	0.0819	0	0	0	Wrist 3 to Flange

3. Forward Kinematics (FK)

The Forward Kinematics problem determines the pose of the end-effector (frame {6}) relative to the base (frame {0}) given joint angles $\mathbf{q} = [q_1, q_2, q_3, q_4, q_5, q_6]^T$.

The homogeneous transformation matrix ${}^{i-1}\boldsymbol{T}_i$ for a single link is calculated as:

$$
{}^{i-1}\mathbf{T}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

The global transformation of the end-effector is the product of these individual transformations:

$$
{}^0\mathbf{T}_6(q) = {}^0\mathbf{T}_1(q_1) \cdot {}^1\mathbf{T}_2(q_2) \cdot {}^2\mathbf{T}_3(q_3) \cdot {}^3\mathbf{T}_4(q_4) \cdot {}^4\mathbf{T}_5(q_5) \cdot {}^5\mathbf{T}_6(q_6)
$$

This results in a 4 X 4 matrix containing the rotation matrix $R \in SO(3)$ and position vector $p \in \mathbb{R}^3$. In the implementation, intermediate positions were stored to visualize the "stick figure" robot links.

4. Inverse Kinematics (IK) Implementation

An analytical solution for 6-DOF robots can be complex and sensitive to configuration changes. Therefore, a numerical approach using the **Newton-Raphson method** was implemented. This allows for iterative convergence toward a target pose.

4.1 The Jacobian Matrix

The geometric Jacobian $\mathbf{J(q)} \in \mathbb{R}^{6x6}$ relates joint velocities to the end-effector spatial velocity:

$$\mathbf{V_{spatial} = J(q)\ \dot{q}}$$

The code constructs **J** column by column. For a revolute joint **i**, the ***i-th*** column is given by:

$$\mathbf{J_i = [\ [\ \ z_{i-1} X (p_{ee} - p_{i-1})\ ]}$$

$$\mathbf{[\ \ \ z_{i-1}\ \ \ ]\ ]}$$

Where $\mathbf{z_{i-1}}$ is the axis of rotation and p_{i-1} is the joint position, extracted from the Forward Kinematics chain.

4.2 Newton-Raphson Iteration

The goal is to find **q** such that the error between the current pose x(q) and target pose x_d is minimized. The error vector **e** consists of:

1. **Position Error:** $\mathbf{e_p = p_d - p_{curr}}$
2. **Orientation Error**: Calculated using the vector cross-product approximation for small angles:

   $$\mathbf{e_0 = \tfrac{1}{2} \sum_{i=1}^{3} (\ n_{curr,i}\ X\ n_{target,i}\ )}$$

   Where **n** are the column vectors of the rotation matrices.

The joint update rule is:

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

$$\mathbf{\Delta\ q = J\dagger\ \ e}$$

4.3 Damped Least Squares (Singularity Robustness)

To prevent instability near singularities (where det(J) ≈ 0), the standard pseudo-inverse $\mathbf{J\dagger}$ is replaced by the Damped Least Squares (DLS) inverse:

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

The damping factor $\boldsymbol{\lambda}$ was set to 0.01. This ensures that velocities do not explode when the robot is fully extended or in a singular configuration, although at the cost of slight positional accuracy in those specific regions.

5. Trajectory Generation and Simulation

5.1 Visualization

The robot is visualized in a 3D Matplotlib environment.

- **Links:** Represented by line segments connecting joint origins derived from FK.
- **End-Effector:** Visualized with an RGB quiver plot representing the SE(3) frame (Red=X, Green=Y, Blue=Z).
- **Trace:** A history of the end-effector position is plotted to visualize the path.

5.2 Trajectories

Two autonomous trajectories were implemented:

1. **Parametric Circle:** Defined in the Y-Z plane with a radius of 10cm. The orientation is kept fixed (identity) to demonstrate the robot's ability to decouple position and orientation.
2. **Square Path:** A linear interpolation between 4 corner points in Cartesian space. The trajectory is discretized into 1mm increments, and the IK solver runs for every point to generate smooth motion.

5.3 Manual Control

The simulation includes an interactive "Drive" mode. Buttons allow the user to apply incremental delta vectors ($\Delta x = [\pm 0.02, 0, 0]^T$) to the current target pose. The Inverse Kinematics solver computes the new configuration in real-time, effectively allowing teleoperation in task space.

6. Conclusion

This project successfully demonstrated the kinematic implementation of a UR3 robot from first principles. By deriving the DH parameters and implementing a Newton-Raphson solver with Damped Least Squares, the simulation achieved robust trajectory tracking and interactive control. The code highlights the mathematical relationship between the joint space $\mathbb{R}^6$ and task space SE(3) without reliance on external "black box" libraries.

7. Appendix

Animation Video of Output and Code:
https://drive.google.com/drive/folders/1qVZIeu-B5T3UO-egUfQ2un9MUvWCy_DR?usp=drive_link`

Code Snippets:
Note: Attached Snippets are continuation of the entire code

```python
# UR3 Inverse Kinematics Simulation Code
# Author: Prathmesh Barapatre
# Project 2: Robot Mechanics and Control

#Description: This script simulates a UR3 robot performing task-space trajectories (Square, Circle)and manual control using a Newton-Raphson Inverse Kinematics solver from scratch.


import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.widgets import Button
import time

#Creating Robot Class
class UR3Robot:
    def __init__(self):
        #Standard DH Parameters for UR3 Robot
        #Derived from UR3 specifications.
        #Format: [theta_offset, d, a, alpha]
        #Units: meters (d, a), radians (alpha, theta)
        self.dh_params = [

            [0,         0.1519,   0,        np.pi/2], #Joint 1: Base to Shoulder
            [0,         0,       -0.24365,  0      ], #Joint 2: Shoulder to Elbow
            [0,         0,       -0.21325,  0      ], #Joint 3: Elbow to Wrist 1
            [0,         0.11235,  0,        np.pi/2], #Joint 4: Wrist 1 to Wrist 2
            [0,         0.08535,  0,       -np.pi/2], #Joint 5: Wrist 2 to Wrist 3
            [0,         0.0819,   0,        0      ]  #Joint 6: Wrist 3 to End-Effector
        ]
        self.num_joints = 6
        #Initialising 'Home' Configuration (Upright)
        self.q = np.array([0.0, -np.pi/2, 0.0, -np.pi/2, 0.0, 0.0])

    def dh_transform(self, theta, d, a, alpha):
        """
        Calculates the Homogeneous Transformation Matrix for a single link
        using Denavit-Hartenberg parameters.
        """
        ct, st = np.cos(theta), np.sin(theta)
        ca, sa = np.cos(alpha), np.sin(alpha)

        #Standard DH Matrix
        return np.array([
            [ct, -st*ca,  st*sa,  a*ct],
            [st,  ct*ca, -ct*sa,  a*st],
            [0,   sa,     ca,     d   ],
            [0,   0,      0,      1   ]
        ])

    def forward_kinematics(self, q):
        """
        Computes the Forward Kinematics to find the end-effector position and orientation.
        Returns:
            T: Final homogeneous transform (4x4)
            positions: List of joint origins for plotting
            z_vectors: List of Z-axes for Jacobian calculation
            origins: List of frame origins for Jacobian calculation
        """
        T = np.eye(4)
        positions = [T[:3, 3]]
        z_vectors = [T[:3, 2]]
        origins = [T[:3, 3]]

        for i in range(self.num_joints):
            theta_offset, d, a, alpha = self.dh_params[i]
            theta = q[i] + theta_offset

            T_i = self.dh_transform(theta, d, a, alpha)
            T = T @ T_i #Chain transformations: T_0_i = T_0_(i-1) @ T_(i-1)_i

            positions.append(T[:3, 3])

            #Store Z-vectors and Origins for Geometric Jacobian
            if i < self.num_joints - 1:
                z_vectors.append(T[:3, 2])
                origins.append(T[:3, 3])

        return T, np.array(positions), z_vectors, origins
```

```python
    def compute_jacobian(self, q, p_eff):
        """
        Computes the Geometric Jacobian Matrix J(q).
        J relates joint velocities to end-effector spatial velocity (v, w).
        dim(J) = 6 x 6
        """
        _, _, z_vectors, origins = self.forward_kinematics(q)
        J = np.zeros((6, self.num_joints))

        for i in range(self.num_joints):
            z_i = z_vectors[i]      #Axis of rotation for joint i
            p_i = origins[i]        #Position of joint i

            #Linear Velocity component (v = w x r)
            J[:3, i] = np.cross(z_i, (p_eff - p_i))

            #Angular Velocity component (w = z)
            J[3:, i] = z_i

        return J

    def inverse_kinematics_newton(self, target_T, max_iter=50, tol=1e-3):
        """
        Solves Inverse Kinematics using the Newton-Raphson numerical method.
        Iteratively updates q to minimize error between current and target pose.
        """
        q_current = self.q.copy()
        target_pos = target_T[:3, 3]
        target_rot = target_T[:3, :3]

        for _ in range(max_iter):
            #Forward Kinematics
            T_curr, _, _, _ = self.forward_kinematics(q_current)
            curr_pos = T_curr[:3, 3]
            curr_rot = T_curr[:3, :3]

            #Computing Error
            err_pos = target_pos - curr_pos

            #Orientation Error using Vector Cross Product method
            #Approximation for small angles
            err_rot = np.zeros(3)
            for i in range(3):
                err_rot += np.cross(curr_rot[:, i], target_rot[:, i])
            err_rot *= 0.5

            error_vector = np.concatenate((err_pos, err_rot))

            #Checking Convergence
            if np.linalg.norm(error_vector) < tol:
                return q_current, True #Converged

            #Computing Jacobian
            J = self.compute_jacobian(q_current, curr_pos)

            #Solving for dq using Damped Least Squares (DLS)
            #DLS is more robust near singularities than pure pseudo-inverse
            lambda_val = 0.01 #Damping factor
            #J_dls = J^T * (J * J^T + lambda^2 * I)^-1
            J_pinv = J.T @ np.linalg.inv(J @ J.T + lambda_val**2 * np.eye(6))
            dq = J_pinv @ error_vector

            #Update Joints
            q_current += dq

        return q_current, False


def rotation_matrix_to_rpy(R):
    """
    Converts a Rotation Matrix to Roll-Pitch-Yaw angles (XYZ convention).
    Returns angles in degrees for easier visualization.
    """
    sy = np.sqrt(R[0,0] * R[0,0] +  R[1,0] * R[1,0])
    singular = sy < 1e-6
    if not singular:
        x = np.arctan2(R[2,1], R[2,2])
        y = np.arctan2(-R[2,0], sy)
        z = np.arctan2(R[1,0], R[0,0])
    else:
        x = np.arctan2(-R[1,2], R[1,1])
        y = np.arctan2(-R[2,0], sy)
        z = 0
    return np.degrees([x, y, z])

```

```python
#Main Simulation
def run_simulation():
    #Setting up Figure and 3D Axis
    plt.ion()
    fig = plt.figure(figsize=(16, 9)) #Wide layout
    ax = fig.add_axes([0.05, 0.05, 0.65, 0.9], projection='3d') #Plot on Left

    # Title
    ax.set_title("UR3 Inverse Kinematics Trajectory (Red=X, Green=Y, Blue=Z)", fontsize=14)

    # Standard Plot Settings
    ax.set_xlim([-0.5, 0.5])
    ax.set_ylim([-0.5, 0.5])
    ax.set_zlim([0, 0.8])
    ax.set_xlabel('X (m)'); ax.set_ylabel('Y (m)'); ax.set_zlabel('Z (m)')

    # Visual Elements initialize
    robot_line, = ax.plot([], [], [], 'o-', lw=4, color='gray', markersize=8, label='Robot Link')
    path_trace, = ax.plot([], [], [], '--', lw=1, color='orange', label='Path Trace')

    #End-Effector Frame Quivers
    qx = ax.quiver(0,0,0,0,0,0, color='r', linewidth=2)
    qy = ax.quiver(0,0,0,0,0,0, color='g', linewidth=2)
    qz = ax.quiver(0,0,0,0,0,0, color='b', linewidth=2)

    #Initializing Robot
    robot = UR3Robot()
    path_x, path_y, path_z = [], [], []
    current_T_ee = np.eye(4)

    #Info Panel Text
    info_text = fig.text(0.75, 0.80, "Robot Status", fontsize=11, family='monospace',
                         verticalalignment='top', bbox=dict(facecolor='white', alpha=0.8))

    #Plot Update Function
    def update_plot(q, clear_trace=False):
        nonlocal qx, qy, qz, path_x, path_y, path_z

        if clear_trace:
            path_x, path_y, path_z = [], [], [] #Reset trace

        #Calculate FK
        T_ee, joints, _, _ = robot.forward_kinematics(q)

        #Update Robot Links
        robot_line.set_data(joints[:, 0], joints[:, 1])
        robot_line.set_3d_properties(joints[:, 2])

        #Update Trace
        p_ee = T_ee[:3, 3]
        path_x.append(p_ee[0])
        path_y.append(p_ee[1])
        path_z.append(p_ee[2])
        path_trace.set_data(path_x, path_y)
        path_trace.set_3d_properties(path_z)

        #Update End Effector Frame
        qx.remove(); qy.remove(); qz.remove() #Clear old quivers
        R = T_ee[:3, :3]
        L = 0.1 #Axis length
        qx = ax.quiver(p_ee[0], p_ee[1], p_ee[2], R[0,0], R[1,0], R[2,0], color='r', length=L)
        qy = ax.quiver(p_ee[0], p_ee[1], p_ee[2], R[0,1], R[1,1], R[2,1], color='g', length=L)
        qz = ax.quiver(p_ee[0], p_ee[1], p_ee[2], R[0,2], R[1,2], R[2,2], color='b', length=L)

        #Update Info Panel
        rpy = rotation_matrix_to_rpy(R)
        txt = f"End-Effector Pose:\n" + "-"*20 + "\n"
        txt += f"Pos [X, Y, Z] (m):\n[{p_ee[0]:.3f}, {p_ee[1]:.3f}, {p_ee[2]:.3f}]\n\n"
        txt += f"Rot [R, P, Y] (deg):\n[{rpy[0]:.1f}, {rpy[1]:.1f}, {rpy[2]:.1f}]\n"
        info_text.set_text(txt)

        fig.canvas.draw()
        fig.canvas.flush_events()
        return T_ee
```

```python
#Initial Draw
current_T_ee = update_plot(robot.q)

def run_trajectory(traj_points, label):
    nonlocal current_T_ee
    print(f"Executing {label} Trajectory...")

    #Move to Start Configuration first
    if len(traj_points) > 0:
        q_start, success = robot.inverse_kinematics_newton(traj_points[0])
        if success:
            robot.q = q_start
            update_plot(q_start, clear_trace=True) #Clear old trace
            plt.pause(0.2)

    #Execute Path
    for target_T in traj_points[1:]:
        q_sol, success = robot.inverse_kinematics_newton(target_T)
        #Update robot state
        robot.q = q_sol
        if success:
            update_plot(q_sol, clear_trace=False)
        else:
            print("IK Singularity/Limit reached.")
        plt.pause(0.01) # Animation Speed

    print(f"{label} Complete.")
    current_T_ee, _, _, _ = robot.forward_kinematics(robot.q)

def btn_circle_cb(event):
    #Parametric Circle in Y-Z plane
    center = np.array([0.25, 0.1, 0.3])
    radius = 0.10
    steps = 60
    trajectory = []
    for t in np.linspace(0, 2*np.pi, steps):
        #x fixed, y and z vary
        x = center[0]
        y = center[1] + radius * np.cos(t)
        z = center[2] + radius * np.sin(t)
        T = np.eye(4)
        T[:3, 3] = [x, y, z] #Orientation stays Identity
        trajectory.append(T)
    run_trajectory(trajectory, "Circle")

def btn_square_cb(event):
    #Square Path
    center = np.array([0.25, 0.1, 0.3])
    side = 0.15
    x = center[0]
    #Define Corners
    c1 = [x, center[1]-side/2, center[2]-side/2]
    c2 = [x, center[1]+side/2, center[2]-side/2]
    c3 = [x, center[1]+side/2, center[2]+side/2]
    c4 = [x, center[1]-side/2, center[2]+side/2]
    corners = [c1, c2, c3, c4, c1] #Loop back to start

    trajectory = []
    steps_per_side = 20
    for k in range(4):
        start_p = np.array(corners[k])
        end_p = np.array(corners[k+1])
        for t in np.linspace(0, 1, steps_per_side):
            p = start_p + (end_p - start_p) * t
            T = np.eye(4)
            T[:3, 3] = p
            trajectory.append(T)
    run_trajectory(trajectory, "Square")

def move_callback(direction_vec):
    nonlocal current_T_ee
    #Manual Control Step
    target_T = current_T_ee.copy()
    delta = 0.02 #2cm step
    target_T[:3, 3] += direction_vec * delta

    q_sol, success = robot.inverse_kinematics_newton(target_T)

    if success:
        robot.q = q_sol
        #Keeping trace enabled to draw with keys
        current_T_ee = update_plot(q_sol, clear_trace=False)
    else:
        print("Cannot move: IK Solution not found.")

def btn_home_cb(event):
    nonlocal current_T_ee
    robot.q = np.array([0.0, -np.pi/2, 0.0, -np.pi/2, 0.0, 0.0]) # Home
    current_T_ee = update_plot(robot.q, clear_trace=True)
    print("Robot homed.")
```

```python
    #UI Layout
    #Coordinates:
    x_base = 0.70
    btn_w = 0.08
    btn_h = 0.05
    gap = 0.01

    #Manual Control Section
    y_manual = 0.45
    fig.text(x_base, y_manual + 0.08, "Manual Control", fontweight='bold', fontsize=12)

    #Row 1: X+, Y+, Z+
    y_row1 = y_manual
    btn_xp = Button(plt.axes([x_base, y_row1, btn_w, btn_h]), 'X+')
    btn_yp = Button(plt.axes([x_base + btn_w + gap, y_row1, btn_w, btn_h]), 'Y+')
    btn_zp = Button(plt.axes([x_base + 2*(btn_w + gap), y_row1, btn_w, btn_h]), 'Z+')

    #Row 2: X-, Y-, Z-
    y_row2 = y_manual - btn_h - gap
    btn_xn = Button(plt.axes([x_base, y_row2, btn_w, btn_h]), 'X-')
    btn_yn = Button(plt.axes([x_base + btn_w + gap, y_row2, btn_w, btn_h]), 'Y-')
    btn_zn = Button(plt.axes([x_base + 2*(btn_w + gap), y_row2, btn_w, btn_h]), 'Z-')

    #Trajectory Section
    y_traj = 0.15
    fig.text(x_base, y_traj + 0.08, "Trajectory Tasks", fontweight='bold', fontsize=12)

    #Horizontal Layout: Home | Circle | Square
    t_btn_w = 0.08

    btn_home = Button(plt.axes([x_base, y_traj, t_btn_w, btn_h]), 'Home')
    btn_circ = Button(plt.axes([x_base + t_btn_w + gap, y_traj, t_btn_w, btn_h]), 'Circle')
    btn_sq   = Button(plt.axes([x_base + 2*(t_btn_w + gap), y_traj, t_btn_w, btn_h]), 'Square')

    #Assigning Callbacks
    btn_xp.on_clicked(lambda x: move_callback(np.array([1, 0, 0])))
    btn_xn.on_clicked(lambda x: move_callback(np.array([-1, 0, 0])))
    btn_yp.on_clicked(lambda x: move_callback(np.array([0, 1, 0])))
    btn_yn.on_clicked(lambda x: move_callback(np.array([0, -1, 0])))
    btn_zp.on_clicked(lambda x: move_callback(np.array([0, 0, 1])))
    btn_zn.on_clicked(lambda x: move_callback(np.array([0, 0, -1])))

    btn_sq.on_clicked(btn_square_cb)
    btn_circ.on_clicked(btn_circle_cb)
    btn_home.on_clicked(btn_home_cb)

    fig._buttons = [btn_xp, btn_xn, btn_yp, btn_yn, btn_zp, btn_zn, btn_sq, btn_circ, btn_home]

    print("UR3 Simulation Ready.")
    print("Controls: Use arrows on screen to move. 'Z' controls height.")
    plt.ioff()
    plt.show()

if __name__ == "__main__":
    run_simulation()
```