# ME 5250 Project 2 - UR5e Collaborative Robot

*Ahmad Hassan*

## 1. Introduction

This project implements forward kinematics, a numerical Jacobian, and Newton's method inverse kinematics solver for the **Universal Robots UR5e 6-DOF collaborative robot**. The robot follows a square trajectory in task space while maintaining fixed end-effector orientation, visualized through a 3D stick-figure animation.

## 2. Robot Description and DH Parameters

The UR5e consists of six revolute joints in a waist-shoulder-elbow configuration followed by a three-axis wrist. While Universal Robots' official documentation provides DH parameters using the standard (Paul) convention, this project uses parameters from Williams (2024), which presents the **modified (Craig) DH convention**. This choice aligns with the convention taught in this course, where coordinate frames are attached at the joint rather than distal to it.

## 3. Forward Kinematics

The forward kinematics computes end-effector pose by chaining homogeneous transformation matrices. A `dh_transform()` function was written to compute the transformation matrix for each joint given the four DH parameters (α, a, d, θ).

Initially, the implementation computed T_0_6 by multiplying the six individual joint transforms. However, this only gives the pose of frame {6} relative to frame {0}—not the actual tool tip position. Referring to Figure 6 in Williams, the base frame {B} is offset below {0} by L_B, and the tool plate frame {TP} extends beyond {6} by L_TP.

The final `forward_kinematics()` function computes the complete transformation as:

$$[{}^{B}_{TP}T] = [{}^{B}_{0}T] \,.\, [{}^{0}_{6}T] . [{}^{6}_{TP}T]$$

where T_B_0 and T_6_TP are constant translation matrices along the Z-axis by L_B and L_TP respectively. This ensures the output represents the true tool tip pose in the base frame. The implementation was verified against numerical examples from Williams (2024) using UR3e parameters.

## 4. Jacobian Matrix Computation

Rather than deriving the analytical Jacobian (which involves computing z-axis unit vectors and position vectors for each joint frame), a **numerical finite-difference approach** was

implemented. This method perturbs each joint angle by a small increment $\delta = 10^{-6}$ rad and observes the resulting change in end-effector pose:

For the translational component (columns 1-3 of each Jacobian column):

$$J_{pos}(:, i) = \frac{p(q + \delta e_i) - p(q)}{\delta}$$

For the rotational component, the angular velocity is extracted from the differential rotation matrix:

$$R_{diff} = R(q + \delta e_i) \cdot R(q)^T$$

The skew-symmetric portion of R_diff encodes the angular velocity components, extracted as:

$$J_{rot}(:, i) = \frac{1}{\delta}[R_{diff}(3,2),\ R_{diff}(1,3),\ R_{diff}(2,1)]^T$$

The numerical Jacobian approach offers several advantages: implementation simplicity, immunity to derivation errors, and automatic correctness verification through the FK function. The perturbation size $\delta = 10^{-6}$ balances numerical precision against floating-point truncation errors.

## 5. Inverse Kinematics Solver

The IK problem seeks joint angles q such that FK(q) = T_target. This is solved iteratively using Newton's method, which linearizes the forward kinematics using the Jacobian matrix and solves joint corrections that reduce the pose error.. At each iteration, the position error is computed as $e\_pos = p\_target - p\_current$, and the orientation error is extracted from $R\_error = R\_target \cdot R\_current^T$ using the same skew-symmetric extraction as the Jacobian computation. The joint update follows:

$$q_{k+1} = q_k + J^{\dagger}(q_k) \cdot e$$

The pseudoinverse J† (via MATLAB's `pinv()`) was chosen over direct matrix inversion because the Jacobian can become singular or near-singular at certain configurations. The pseudoinverse provides a least-squares solution that handles these cases gracefully without causing numerical instability.

During testing, joint angles were observed to accumulate to very large values over successive iterations. To address this, `wrapToPi()` was applied after each update to constrain angles to [−π, π]. The solver terminates when $||e|| < 10^{-6}$ or after 100 iterations. Additionally, each waypoint uses the previous solution as its initial guess, exploiting trajectory continuity for faster convergence.

## 6. Code Structure

The implementation consists of modular functions written from scratch in MATLAB without using any robotics toolboxes. The core functions are:

- `dh_transform(alpha, a, d, theta)` — Computes a single 4×4 transformation matrix from DH parameters
- `forward_kinematics(q)` Chains all transforms to compute tool tip pose from joint angles
- `compute_jacobian(q)` Numerically computes the 6×6 Jacobian using finite differences
- `ik_solver(T_target, q_init, max_iter, tol)` — Iteratively solves for joint angles using Newton's method
- `generate_square_trajectory(center, side_length)` — Creates waypoints along a square path
- `plot_robot_frame(q)` — Renders the robot stick figure with end-effector frame

The main script initializes the robot at $q_0$, computes the initial end-effector position, generates the square trajectory centered at that location, then loops through each waypoint solving IK and updating the animation. Each IK call uses the previous solution as its starting guess, and the traced path is accumulated for display.

## 7. Trajectory Generation and Simulation

A square trajectory was generated in the XZ plane (vertical plane at constant Y), centered at the initial end-effector position [0; -45; -90; -45; 90; 0]. The square has a side length of 0.1 m with waypoints spaced at 1 mm increments as specified in the requirements, resulting in approximately 400 total waypoints. The end-effector orientation is held fixed throughout the trajectory, demonstrating independent control of position and orientation.

The `plot_robot_frame()` function renders the robot as a 3D stick figure by computing nine key points along the kinematic chain (base, waist, shoulder, elbow, three wrist points, and tool tip). The end-effector frame is visualized using RGB arrows (red=X, green=Y, blue=Z) to show the SE(3) pose.

The animation displays: the robot configuration, the target square trajectory (dashed black), the traced path (solid red), and the current target point (green marker).

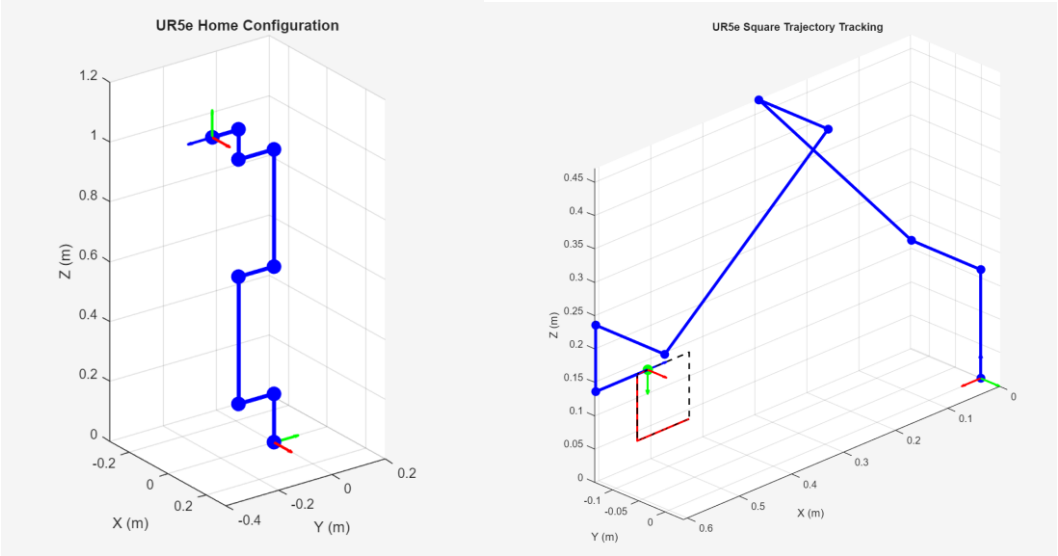



***Figure 1:*** *UR5e at home configuration [0°, 0°, 0°, 0°, 0°, 0°] (left) and tracking a 0.1 m square trajectory in the XZ plane with fixed end-effector orientation (right). RGB arrows indicate the end-effector frame axes.*

## 8. Results and Discussion

Starting from the initial configuration $q_0$ = [0°, −45°, −90°, −45°, 90°, 0°], the IK solver successfully converged for all waypoints along the square trajectory. The robot exhibited smooth joint motion throughout, with no discontinuous jumps between configurations. This is because the solver naturally tracks the nearest solution when starting from the previous waypoint's result, keeping the robot in a consistent "elbow-up" configuration.

The chosen trajectory and initial configuration also kept the robot away from the three singularity conditions identified by Williams: the elbow-extended singularity ($\theta_3$ = 0°, ±180°), the wrist singularity ($\theta_5$ = 0°, ±180°), and the shoulder singularity where the wrist origin lies on the plane formed by the first two joint axes. Avoiding these regions ensured stable Jacobian inversion and reliable tracking performance.

## References

1. Williams, R.L. II, "Universal Robot URe-Series Kinematics & Dynamics," Ohio University, January 2024.
2. Universal Robots, "DH Parameters for Calculations of Kinematics and Dynamics," UR Support Articles. Available: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/

Video Link: https://drive.google.com/file/d/1OnpnlW_fe-pRKoMz0PPgpak4YGyb0VTS/view?usp=sharing

# APPENDIX: MATLAB Code | Ahmad Hassan

```
clear; clc; close all;

% Initial Configuration
q_init = deg2rad([0; -45; -90; -45; 90; 0]);

% Find where end-effector is at initial config
T_init = forward_kinematics(q_init);
p_init = T_init(1:3, 4);
R_init = T_init(1:3, 1:3);

fprintf('Initial end-effector position: [%.3f, %.3f, %.3f]\n', p_init);

% Generate Square Trajectory
% Center the square at current position, in XZ plane
side_length = 0.1;  % 10 cm
positions = generate_square_trajectory(p_init, side_length);
num_points = size(positions, 2);

% Use the current orientation
R_fixed = R_init;

% IK Parameters
max_iter = 100;
tol = 1e-6;

% Solve IK for All Waypoints
fprintf('Solving IK for %d waypoints...\n', num_points);

q_trajectory = zeros(6, num_points);
q_current = q_init;

for i = 1:num_points
    T_target = [R_fixed, positions(:,i); 0 0 0 1];
    [q_sol, success] = ik_solver(T_target, q_current, max_iter, tol);

    if ~success
        warning('IK failed at waypoint %d', i);
    end

    q_trajectory(:, i) = q_sol;
    q_current = q_sol;
end
fprintf('IK complete.\n');

% Animate
figure('Position', [100, 100, 1000, 800]);

skip = 4;
ee_path = [];
```

```
for i = 1:skip:num_points
    cla;

    q = q_trajectory(:, i);
    T_current = forward_kinematics(q);
    ee_path = [ee_path, T_current(1:3, 4)];

    plot_robot_frame(q);
    hold on;

    % Target trajectory
    plot3(positions(1,:), positions(2,:), positions(3,:), 'k--',
'LineWidth', 1.5);

    % Traced path
    if size(ee_path, 2) > 1
        plot3(ee_path(1,:), ee_path(2,:), ee_path(3,:), 'r-', 'LineWidth',
2);
    end

    % Current target
    plot3(positions(1,i), positions(2,i), positions(3,i), 'go',
'MarkerSize', 10, 'MarkerFaceColor', 'g');

    hold off;

    grid on;
    axis equal;
    xlabel('X (m)'); ylabel('Y (m)'); zlabel('Z (m)');
    title('UR5e Square Trajectory Tracking');
    view(135, 25);

    drawnow;
end

function T = dh_transform(alpha, a, d, theta)
% Inputs:
%   alpha - twist angle α_{i-1} (radians)
%   a     - link length a_{i-1} (meters)
%   d     - link offset d_i (meters)
%   theta - joint angle θ_i (radians)
% Output:
%   T - 4x4 homogeneous transformation matrix

    c_theta = cos(theta);
    s_theta = sin(theta);
    c_alpha = cos(alpha);
    s_alpha = sin(alpha);

    T = [c_theta           -s_theta            0           a;
         s_theta*c_alpha    c_theta*c_alpha   -s_alpha    -d*s_alpha;
         s_theta*s_alpha    c_theta*s_alpha    c_alpha     d*c_alpha;
         0                  0                  0           1];
end
```

```
function T = forward_kinematics(q)
% Inputs:
%   q - 6x1 joint angles (radians)
% Outputs:
%   T - 4x4 transformation matrix (base to tool plate)

    % UR5e parameters (meters)
    l_b  = 0.163;
    a2   = 0.425;
    a3   = 0.392;
    d4   = 0.133;
    d5   = 0.100;
    l_t = 0.100;

    % DH parameters: [alpha, a, d, theta]
    DH = [    0,   0,   0,  q(1);
           pi/2,   0,   0,  q(2) + pi/2;
              0,  a2,   0,  q(3);
              0,  a3,  d4,  q(4) - pi/2;
          -pi/2,   0,  d5,  q(5);
           pi/2,   0,   0,  q(6)];

    % Compute T_0^6
    T_0_6 = eye(4);
    for i = 1:6
        T_0_6 = T_0_6 * dh_transform(DH(i,1), DH(i,2), DH(i,3), DH(i,4));
    end

    % Add base and tool plate
    T_B_0  = [eye(3), [0; 0; l_b];  0 0 0 1];
    T_6_TP = [eye(3), [0; 0; l_t]; 0 0 0 1];

    T = T_B_0 * T_0_6 * T_6_TP;
end

function plot_robot(q)
% Input:
%   q - 6x1 vector of joint angles (radians)
% Draws a 3D stick figure of the UR5e robot

    % UR5e link lengths (meters)
    l_b = 0.163;
    a2 = 0.425;
    a3 = 0.392;
    d4 = 0.133;
    d5 = 0.100;
    l_t = 0.100;

    % DH Parameters: [alpha, a, d, theta]
    DH = [0        0    0    q(1);
          pi/2     0    0    q(2) + pi/2;
          0        a2   0    q(3);
          0        a3   d4   q(4) - pi/2;
```

```
         -pi/2     0     d5    q(5);
          pi/2     0     0     q(6)];

    % Compute transforms to each frame
    T = cell(7,1);
    T{1} = eye(4);
    for i = 1:6
        T_i = dh_transform(DH(i,1), DH(i,2), DH(i,3), DH(i,4));
        T{i+1} = T{i} * T_i;
    end

    % Extract rotation matrices
    R1 = T{2}(1:3, 1:3);
    R2 = T{3}(1:3, 1:3);
    R3 = T{4}(1:3, 1:3);
    R5 = T{6}(1:3, 1:3);
    R6 = T{7}(1:3, 1:3);

    % Compute positions incrementally
    P0 = [0; 0; 0];                         % Base
    P1 = P0 + l_b * [0; 0; 1];              % Waist
    P2 = P1 - d4 * R1(:,2);                 % Shoulder
    P3 = P2 + a2 * R2(:,1);                 % Elbow
    P4 = P3 - d4 * R2(:,3);
    P5 = P4 + a3 * R3(:,1);                 % Wrist 1
    P6 = P5 + d4 * R3(:,3);                 % Wrist 2
    P7 = P6 + d5 * R5(:,3);                 % Wrist 3
    P8 = P7 + l_t * R6(:,3);                % Tip

    % Combine points
    points = [P0, P1, P2, P3, P4, P5, P6, P7, P8];

    % Plot robot links
    figure;
    plot3(points(1,:), points(2,:), points(3,:), 'b-o', ...
          'LineWidth', 3, 'MarkerSize', 8, 'MarkerFaceColor', 'b');
    hold on;

    % Axis length for frames
    len = 0.1;

    % Base frame (RGB at origin)
    R0 = eye(3);
    quiver3(P0(1), P0(2), P0(3), R0(1,1)*len, R0(2,1)*len, R0(3,1)*len, 'r',
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P0(1), P0(2), P0(3), R0(1,2)*len, R0(2,2)*len, R0(3,2)*len, 'g',
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P0(1), P0(2), P0(3), R0(1,3)*len, R0(2,3)*len, R0(3,3)*len, 'b',
'LineWidth', 2, 'MaxHeadSize', 0.5);

    % End-effector frame (RGB at tool tip)
    quiver3(P8(1), P8(2), P8(3), R6(1,1)*len, R6(2,1)*len, R6(3,1)*len, 'r',
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P8(1), P8(2), P8(3), R6(1,2)*len, R6(2,2)*len, R6(3,2)*len, 'g',
```

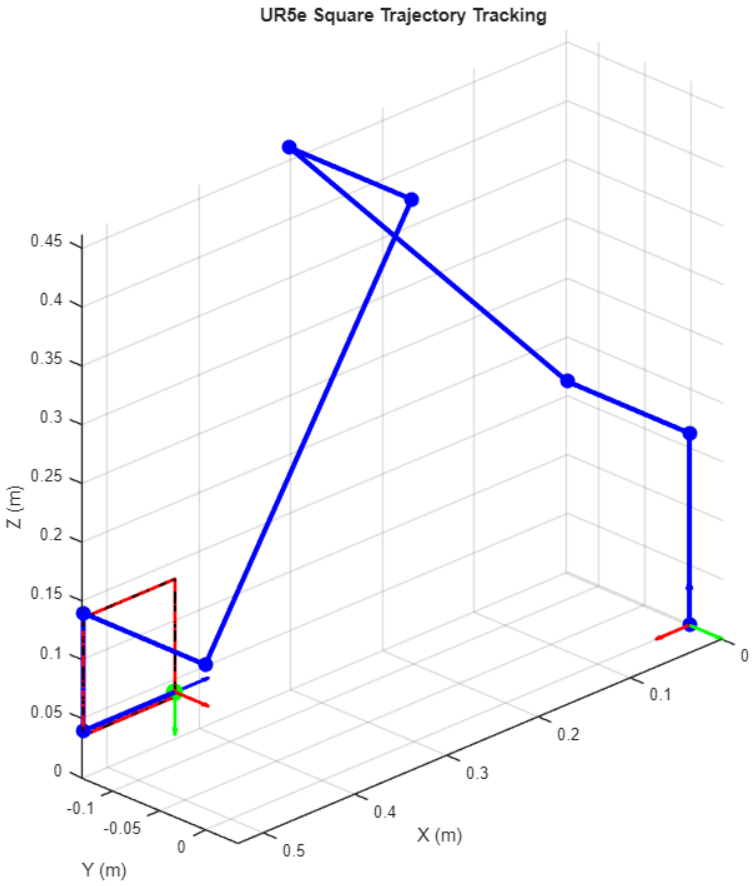
```
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P8(1), P8(2), P8(3), R6(1,3)*len, R6(2,3)*len, R6(3,3)*len, 'b',
'LineWidth', 2, 'MaxHeadSize', 0.5);

    hold off;
    grid on;
    axis equal;
    xlim([-0.3, 0.3]);
    ylim([-0.4, 0.2]);
    zlim([0, 1.2]);
    xlabel('X (m)');
    ylabel('Y (m)');
    zlabel('Z (m)');
    title('UR5e Home Configuration');
end

function J = compute_jacobian(q)
% Input:
%   q - 6x1 vector of joint angles (radians)
% Compute Numerical Jacobian using finite differences

    delta = 1e-6;

    T_current = forward_kinematics(q);
    pos_current = T_current(1:3, 4);
    R_current = T_current(1:3, 1:3);

    J = zeros(6, 6);

    for i = 1:6
        q_perturbed = q;
        q_perturbed(i) = q_perturbed(i) + delta;

        T_perturbed = forward_kinematics(q_perturbed);
        pos_perturbed = T_perturbed(1:3, 4);
        R_perturbed = T_perturbed(1:3, 1:3);

        J(1:3, i) = (pos_perturbed - pos_current) / delta;

        R_diff = R_perturbed * R_current';
        S = R_diff - eye(3);
        J(4:6, i) = [S(3,2); S(1,3); S(2,1)] / delta;
    end
end

function [q, success] = ik_solver(T_target, q_init, max_iter, tol)
% Inputs:
%   T_target - 4x4 homogeneous transformation matrix of desired end-effector
pose
%   q_init   - 6x1 vector of initial joint angles (radians)
%   max_iter - Maximum number of iterations
%   tol      - Convergence tolerance for position/orientation error (meters/
radians)
%
```

```
% Outputs:
%   q       - 6x1 vector of solved joint angles (radians)
%   success - Boolean flag: true if converged within tolerance, false
otherwise

    q = q_init;

    for i = 1:max_iter
        T_current = forward_kinematics(q);

        pos_error = T_target(1:3, 4) - T_current(1:3, 4);

        R_target = T_target(1:3, 1:3);
        R_current = T_current(1:3, 1:3);
        R_error = R_target * R_current';
        S = R_error - eye(3);
        orient_error = [S(3,2); S(1,3); S(2,1)];

        error = [pos_error; orient_error];

        if norm(error) < tol
            success = true;
            return;
        end

        J = compute_jacobian(q);
        q = q + pinv(J) * error;
        q = wrapToPi(q);
    end

    success = false;
end

function positions = generate_square_trajectory(center, side_length)
% Inputs:
%   center      - [x; y; z] center of square (meters)
%   side_length - length of each side (meters)
%
% Outputs:
%   positions - 3 x N matrix of positions

    half = side_length / 2;
    cx = center(1);
    cy = center(2);
    cz = center(3);

    % Number of points per side (1mm spacing)
    spacing = 0.001;
    num_points_per_side = round(side_length / spacing) + 1;

    % Four corners in XZ plane (Y is constant)
    c1 = [cx - half; cy; cz - half];
    c2 = [cx + half; cy; cz - half];
    c3 = [cx + half; cy; cz + half];
```

```
    c4 = [cx - half; cy; cz + half];

    % Generate points along each side
    side1 = linspace_3d(c1, c2, num_points_per_side);
    side2 = linspace_3d(c2, c3, num_points_per_side);
    side3 = linspace_3d(c3, c4, num_points_per_side);
    side4 = linspace_3d(c4, c1, num_points_per_side);

    % Combine
    positions = [side1, side2(:,2:end), side3(:,2:end), side4(:,2:end-1)];

    fprintf('Generated %d trajectory points\n', size(positions, 2));
end


function pts = linspace_3d(p1, p2, n)
% Linearly interpolate between two 3D points
    t = linspace(0, 1, n);
    pts = p1 + (p2 - p1) * t;
end

function plot_robot_frame(q)
% Input:
%   q - 6x1 vector of joint angles (radians)
% Draws UR5e stick figure for animation

    % UR5e link lengths (meters)
    l_b  = 0.163;
    a2   = 0.425;
    a3   = 0.392;
    d4   = 0.133;
    d5   = 0.100;
    l_t = 0.100;

    % DH Parameters: [alpha, a, d, theta]
    DH = [    0,   0,   0,  q(1);
           pi/2,   0,   0,  q(2) + pi/2;
              0,  a2,   0,  q(3);
              0,  a3,  d4,  q(4) - pi/2;
          -pi/2,   0,  d5,  q(5);
           pi/2,   0,   0,  q(6)];

    % Compute transforms
    T = cell(7,1);
    T{1} = eye(4);
    for i = 1:6
        T_i = dh_transform(DH(i,1), DH(i,2), DH(i,3), DH(i,4));
        T{i+1} = T{i} * T_i;
    end

    % Extract rotation matrices
    R1 = T{2}(1:3, 1:3);
    R2 = T{3}(1:3, 1:3);
    R3 = T{4}(1:3, 1:3);
```

```
    R5 = T{6}(1:3, 1:3);
    R6 = T{7}(1:3, 1:3);

    % Compute positions incrementally
    P0 = [0; 0; 0];                         % Base
    P1 = P0 + l_b * [0; 0; 1];              % Waist
    P2 = P1 - d4 * R1(:,2);                 % Shoulder
    P3 = P2 + a2 * R2(:,1);                 % Elbow
    P4 = P3 - d4 * R2(:,3);
    P5 = P4 + a3 * R3(:,1);                 % Wrist 1
    P6 = P5 + d4 * R3(:,3);                 % Wrist 2
    P7 = P6 + d5 * R5(:,3);                 % Wrist 3
    P8 = P7 + l_t * R6(:,3);                % Tip

    points = [P0, P1, P2, P3, P4, P5, P6, P7, P8];

    hold on;

    % Robot links
    plot3(points(1,:), points(2,:), points(3,:), 'b-o', ...
          'LineWidth', 3, 'MarkerSize', 6, 'MarkerFaceColor', 'b');

    % Coordinate frame size
    len = 0.04;

    % Base frame
    quiver3(P0(1), P0(2), P0(3), len, 0, 0, 'r', 'LineWidth', 2,
'MaxHeadSize', 0.5);
    quiver3(P0(1), P0(2), P0(3), 0, len, 0, 'g', 'LineWidth', 2,
'MaxHeadSize', 0.5);
    quiver3(P0(1), P0(2), P0(3), 0, 0, len, 'b', 'LineWidth', 2,
'MaxHeadSize', 0.5);

    % End-effector frame
    quiver3(P8(1), P8(2), P8(3), R6(1,1)*len, R6(2,1)*len, R6(3,1)*len, 'r',
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P8(1), P8(2), P8(3), R6(1,2)*len, R6(2,2)*len, R6(3,2)*len, 'g',
'LineWidth', 2, 'MaxHeadSize', 0.5);
    quiver3(P8(1), P8(2), P8(3), R6(1,3)*len, R6(2,3)*len, R6(3,3)*len, 'b',
'LineWidth', 2, 'MaxHeadSize', 0.5);

    hold off;
end
```

```
Initial end-effector position: [0.478, -0.133, 0.086]
Generated 400 trajectory points
Solving IK for 400 waypoints...
IK complete.
```




*Published with MATLAB® R2025a*