Implementation of a Plate Cleaning Robot using ABB IRB 120

*A Comprehensive Approach to Dishwashing Automation

1st Jerin Peter
Robotics Department
University of California
Riverside, CA
jerinpeter@gmail.com

2nd Prof. Jonathan Realmuto
Mechanical Department
University of California
Riverside, CA
jrealmut@ucr.edu

***Abstract*—This paper presents a comprehensive implementation of a plate cleaning robot utilizing the ABB IRB 120 robotic arm. The robot mimics human dishwashing motions through a sophisticated spiral trajectory, ensuring thorough cleaning of plates placed randomly within a predefined workspace. The system leverages numerical inverse kinematics (IK) and Proportional-Derivative (PD) control strategies for precise end-effector positioning and trajectory following. Detailed explanations of the task space definition, plate position generation, trajectory planning, and control methodologies are provided, highlighting the complexity and intricacies involved in developing such a robotic system.**

***Index Terms*—robotics, inverse kinematics, PD control, trajectory planning, automation**

I. Introduction

The automation of routine tasks, such as dishwashing, has seen significant advancements with the integration of robotic systems. This report delves into the meticulous implementation of a robotic system designed to mimic human dishwashing motions. The ABB IRB 120 robotic arm is employed in this project to perform dishwashing tasks by executing a spiral motion to clean plates, emulating human-like washing techniques. This report provides detailed explanations of the system's components, including task space definition, inverse kinematics, control strategies, and trajectory planning.

II. System Overview

The robotic system comprises an ABB IRB 120 robotic arm programmed to navigate a defined workspace, identify plate positions, and perform cleaning operations. The core components of the system include the inverse kinematics method, task space boundaries, plate position generation, and control strategies.

A. System Flowchart

B. Inverse Kinematics Method

Inverse kinematics (IK) is pivotal in determining the required joint angles for the robotic arm to achieve a specific end-effector position and orientation. The numerical IK method is chosen for its robustness in handling the complex configurations of the robotic arm. The IK problem is formulated as:

$$\text{Find } \mathbf{q} \text{ such that } \mathbf{T}(\mathbf{q}) = \mathbf{T}_d \tag{1}$$

where $\mathbf{q}$ is the vector of joint angles, $\mathbf{T}(\mathbf{q})$ is the transformation matrix representing the end-effector pose, and $\mathbf{T}_d$ is the desired end-effector pose.

C. Task Space Definition

The task space is defined to exclude the robot's base area, ensuring safe operation. The boundaries are set as follows:

- **X-axis:** $x_{\min} = 0.0$ m, $x_{\max} = 0.32$ m
- **Y-axis:** $y_{\min} = -0.38$ m, $y_{\max} = 0.32$ m
- **Z-axis:** $z_{\text{plate}} = 0.03$ m (fixed plate height)

D. Plate Position Generation

A function is employed to generate random plate positions within the task space. The generated positions must ensure no overlap between plates and maintain a minimum distance from the robot's base. This is achieved through the following steps:

1) **Random Position Generation:** A new position $\mathbf{P}_{\text{new}}$ is generated within the task space boundaries.
2) **Distance Calculation:** The Euclidean distances between $\mathbf{P}_{\text{new}}$ and existing plate positions $\mathbf{P}_i$ are calculated.
3) **Validation:** The new position is validated if the distances satisfy the minimum distance constraint (minDistance $= 0.28$ m) and the distance from the base (baseRadius $= 0.25$ m).

III. Control and Motion Planning

The robot's motion planning involves two primary phases: cleaning the plates using a spiral trajectory and transitioning between plates.

A. Spiral Trajectory for Cleaning

A spiral trajectory is generated above each plate to simulate the cleaning motion. The trajectory is defined using cylindrical coordinates and converted to Cartesian coordinates as follows:

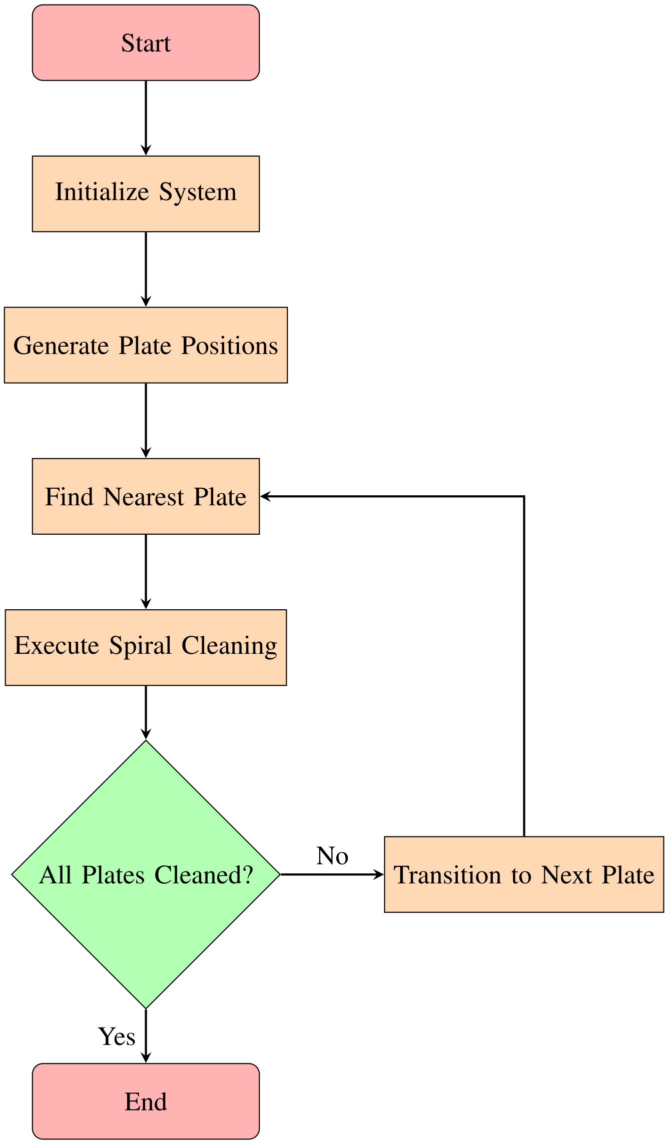

Fig. 1. System Flowchart

1) **Parameters Definition:**
   - Number of points (n_{points}): 55
   - Radius (r): 0.145 m
   - Number of turns (n_{turns}): 3
   - Height above plate ($z_{\text{above\_plate}}$): 0.05 m
2) **Trajectory Equations:**

$$\theta = \text{linspace}(0, 2\pi \cdot n_{\text{turns}}, n_{\text{points}}) \tag{2}$$

where θ is the angle in radians, n_{turns} is the number of turns in the spiral, and n_{points} is the number of points in the trajectory.

$$r_i = \text{linspace}(0, r, n_{\text{points}}) \tag{3}$$

where r_i is the radius at each point i.

$$x_i = r_i \cos(\theta_i) + x_{\text{plate}} \tag{4}$$

where x_i is the x-coordinate at each point i, $\cos$ is the cosine function, and x_{plate} is the x-coordinate of the plate center.

$$y_i = r_i \sin(\theta_i) + y_{\text{plate}} \tag{5}$$

where y_i is the y-coordinate at each point i, $\sin$ is the sine function, and y_{plate} is the y-coordinate of the plate center.

$$z_i = z_{\text{plate}} + z_{\text{above\_plate}} \tag{6}$$

where z_i is the z-coordinate at each point i, z_{plate} is the height of the plate, and $z_{\text{above\_plate}}$ is the height above the plate where the spiral trajectory is defined.

B. Transition Trajectory

Transitioning between plates requires generating smooth trajectories that avoid collisions with the robot's base and other obstacles. This is achieved using a cubic polynomial trajectory planning method, which ensures smooth acceleration and deceleration phases. The process involves several key steps:

1) **Waypoints Definition:** Define intermediate waypoints that guide the robot around the base, avoiding obstacles. These waypoints are strategically chosen to ensure smooth transitions and prevent sudden changes in direction.
2) **Midpoint Angle Calculation:** Calculate the angles for the waypoints using the atan2 function, which provides the angle between the x-axis and the line connecting the robot's current position to the waypoint:

$$\text{angle1} = \text{atan2}(y_{\text{current}}, x_{\text{current}}) \tag{7}$$

where angle1 is the angle of the current position in radians.

$$\text{angle2} = \text{atan2}(y_{\text{next}}, x_{\text{next}}) \tag{8}$$

where angle2 is the angle of the next plate position in radians.
3) **Shortest Arc Calculation:** To determine the shortest arc between two angles:
   - Normalize angles to the range $[0, 2\pi]$.
   - Calculate the clockwise and counterclockwise distances.
   - Select the direction with the smaller distance.
4) **Midpoint Waypoints:** Create intermediate waypoints along the arc at a safe distance from the base:

$$\text{midRadius} = \max(\text{baseRadius} + 0.1, 0.3) \tag{9}$$

where midRadius is the radius of the waypoints from the robot base.

$$\begin{aligned} \text{midPoints} = [&\text{midRadius} \cdot \cos(\text{midAngles}), \\ &\text{midRadius} \cdot \sin(\text{midAngles}), \\ &z_{\text{above\_plate}} + 0.2] \end{aligned} \tag{10}$$

where midPoints are the coordinates of the intermediate waypoints, midAngles are the angles of the waypoints, and $z_{\text{above\_plate}} + 0.2$ is the height of the waypoints above the plate.

5) **Trajectory Planning:** Utilize a cubic polynomial trajectory planning method, such as the cubicpolytraj function in MATLAB, to generate a smooth trajectory that connects these waypoints. This method ensures continuous velocity and acceleration profiles, enhancing the smoothness of the robot's movements. The polynomial coefficients are determined by the initial and final conditions, and the resulting trajectory minimizes jerky movements.

This detailed approach to trajectory planning ensures that the robot can move efficiently and safely between plates, maintaining high performance and reliability.

IV. Trajectory Generation

A. Cleaning Trajectory

The spiral trajectory for cleaning is generated using a parametric approach, where the radius increases linearly with the angle to form a spiral. The equations for the spiral are:

$$x_i = r_i \cos(\theta_i) + x_{\text{plate}} \tag{11}$$

$$y_i = r_i \sin(\theta_i) + y_{\text{plate}} \tag{12}$$

$$z_i = z_{\text{plate}} + z_{\text{above\_plate}} \tag{13}$$

where r_i and θ_i are the radial and angular coordinates, respectively.

B. Transition Trajectory

The transition trajectory ensures smooth movement from one plate to another. The cubic polynomial trajectory is defined by planning the trajectory between the waypoints generated earlier. This method ensures that the transition is smooth and collision-free, enhancing the overall efficiency and safety of the robot's operations. The cubic polynomial trajectory can be mathematically represented as:

$$p(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \tag{14}$$

where $p(t)$ is the position at time t, and a_3, a_2, a_1, a_0 are the polynomial coefficients determined by the initial and final conditions.

V. Experimental Setup and Results

A. Initial Setup

The robot starts at the origin with a zero joint configuration. Plates are placed randomly within the task space, ensuring compliance with spacing constraints.

B. Simulation Execution

The simulation proceeds with the robot identifying the nearest unvisited plate, executing the spiral cleaning motion, and then transitioning to the next plate. This loop continues until all plates are cleaned.

C. Shortest Path Calculation

To determine the shortest path to the next plate, the Euclidean distances between the current position and the unvisited plates are calculated:

$$\text{dist}(i) = \sqrt{(x_{\text{current}} - x_i)^2 + (y_{\text{current}} - y_i)^2} \tag{15}$$

where $\text{dist}(i)$ is the distance to the i-th plate, x_{current} and y_{current} are the current coordinates, and x_i and y_i are the coordinates of the i-th plate. The plate with the minimum distance is selected as the next target.

D. Visualization

The simulation provides real-time visualization of the robot's movements, including the spiral trajectories and transitions. This visualization aids in verifying the correctness of the implemented control and motion planning algorithms.

E. Results

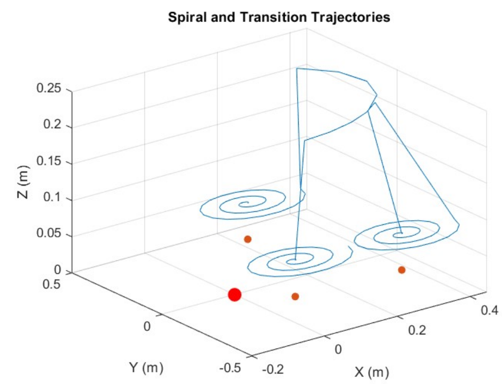

Fig. 2. Spiral and Transition Trajectories

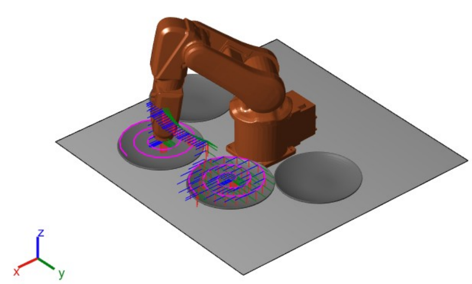

Fig. 3. Robot executing the spiral cleaning trajectory on plates

VI. Conclusion

The robotic system successfully demonstrates the capability to perform dishwashing tasks through a controlled spiral motion. The implementation highlights the effectiveness of

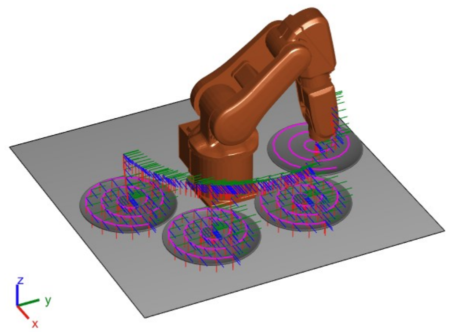

Fig. 4. Robot transitioning between plates using cubic polynomial trajectory

numerical IK methods and PD controllers in achieving precise and efficient robotic operations. Future work may include optimizing the control parameters and exploring advanced trajectory planning algorithms to enhance the system's performance.

Acknowledgment

I, Jerin Peter, the first author, would like to express my gratitude to Professor Jonathan Realmuto, the second author, for his exceptional teaching in the Kinematics and Dynamics (ME221) course. His insightful instruction provided the foundational understanding and intuition necessary for the concepts utilized in this project. I am sincerely thankful for his guidance and support.

References

[1] MathWorks, "Inverse Kinematics - MATLAB Simulink," Link: https://www.mathworks.com/help/robotics/ref/inversekinematics.html.

[2] MathWorks, "Analytical Solutions of the Inverse Kinematics," Link: https://www.mathworks.com/help/robotics/ug/solve-closed-form-inverse-kinematics.html.

[3] MathWorks, "Trajectory Control Modeling with Inverse Kinematics," Link: https://www.mathworks.com/help/robotics/ref/cubicpolytraj.html.

[4] MathWorks, "Implement PID Control," Link: https://www.mathworks.com/help/control/examples/implement-pid-control.html.

[5] MathWorks, "Constraint Objects," Link: https://www.mathworks.com/help/robotics/ug/constraint-objects.html.

[6] MathWorks, "Model and Control a Manipulator Arm with Robotics and Simscape," Link: https://www.mathworks.com/help/robotics/ug/model-and-control-a-manipulator-arm.html.

[7] MathWorks, "Multi-Loop PI Control of a Robotic Arm," Link: https://www.mathworks.com/help/control/ug/multi-loop-pi-control-of-a-robotic-arm.html.

[8] MathWorks, "Connect to Kinova Gen3 Robot and Manipulate the Arm Using MATLAB," Link: https://www.mathworks.com/help/robotics/ug/connect-to-kinova-gen3-robot-and-manipulate-the-arm-using-matlab.html.

[9] MathWorks, "Modeling Inverse Kinematics in a Robotic Arm," Link: https://www.mathworks.com/help/robotics/ug/modeling-inverse-kinematics-in-a-robotic-arm.html.

[10] MathWorks, "Sliding Mode Control Design for a Robotic Manipulator," Link: https://www.mathworks.com/help/robotics/ug/sliding-mode-control-design-for-a-robotic-manipulator.html.

[11] MathWorks, "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks," Link: https://www.mathworks.com/help/robotics/ug/safe-trajectory-tracking-control.html.

Appendix A
MATLAB Code

```matlab
clc;
clear all;
close all;

% {'analytic','numeric','generalized'}
ik_method = "numeric";

r = loadrobot('abbIrb120','DataFormat','column');
q0 = zeros(6,1);
q = q0;

% Define task space boundaries (excluding robot's base area)
x_min = 0.0; x_max = 0.3;
y_min = -0.38; y_max = 0.38;
z_plate = 0.03;  % Plate height
minDistance = 0.28;  % Minimum distance between plates
baseRadius = 0.25;  % Radius of the base exclusion zone

% Number of plates
numPlates = 4;

% Function to generate random plate positions within task space
generateRandomPosition = @() [x_min + (x_max-x_min)*rand, y_min + (y_max-y_min)*rand, z_plate];

% Generate random positions for plates ensuring no overlap and avoiding base
platePositions = zeros(numPlates, 3);
for i = 1:numPlates
    isValid = false;
    while ~isValid
        newPosition = generateRandomPosition();
        distances = sqrt(sum((platePositions(1:i-1,1:2) - newPosition(1:2)).^2, 2));
        distanceFromBase = norm(newPosition(1:2)); % Distance from robot's base
        if all(distances > minDistance) && distanceFromBase > baseRadius
            platePositions(i, :) = newPosition;
            isValid = true;
        end
    end
end

% Initialize plot
figure;
ax = show(r, q, ...
    'Visuals', 'on', ...
    'PreservePlot', 0, ...
    'Fastupdate', 1); hold all;
drawFloor();

% Show all plates first
for idx = 1:numPlates
    PlatePosition = platePositions(idx, :);
    body = rigidBody(['Plate', num2str(idx), '_link']);
    addVisual(body, "Mesh", 'Dinner_Plate_v1.stl', [[0.003*eye(3), zeros(3,1)]; 0 0 0 1]);
    setFixedTransform(body.Joint, trvec2tform(PlatePosition));
    addBody(r, body, r.BaseName);
    show(r, 'Visuals', 'on', 'PreservePlot', 0, 'Frames', 'off', 'Parent', ax);
    drawnow;
end

% BFGSGradientProjection IK object
ik = inverseKinematics('RigidBodyTree', r);

% Set the orientation to ensure the z-axis points down
orientation = eul2quat([0, pi/2, 0]);  % Rotate 180 degrees around y-axis to point z-axis down

% PD Controller parameters
Kp = 20; % Proportional gain
Kd = 0.1; % Derivative gain
dt = 0.01; % Time step

% Initialize visited plates array
visitedPlates = false(numPlates, 1);

% Start position
currentPosition = [0, 0, 0];  % Assuming the robot starts at the origin

% Loop until all plates are visited
for visitCount = 1:numPlates
    % Find the nearest unvisited plate
    distances = sqrt(sum((platePositions(:,1:2) - currentPosition(1:2)).^2, 2));
    distances(visitedPlates) = inf;  % Ignore already visited plates
    [~, idx] = min(distances);  % Find the index of the nearest plate
    PlatePosition = platePositions(idx, :);
    visitedPlates(idx) = true;  % Mark this plate as visited

    % Generate Spiral Trajectory for the plate
    nPoints = 55;
    radius = 0.145;  % Random radius for each plate
    turns = 3;
    theta = linspace(0, 2*pi*turns, nPoints);
    radii = linspace(0, radius, nPoints);  % Linear increment in radius to form a spiral
    z_above_plate = 0.05;  % Height above the plate where the spiral is shown
    z = ones(1, nPoints) * (z_plate + z_above_plate);  % Constant z-coordinate above the plate
    x = radii .* cos(theta) + PlatePosition(1);
    y = radii .* sin(theta) + PlatePosition(2);

    % Initialize error terms for PD control
    prevError = zeros(6,1);

    % Draw spiral above the plate
    for i = 1:nPoints
        % Desired position and orientation for the plate
        Td = trvec2tform([x(i), y(i), z(i)]) * quat2tform(orientation);

        % Find pose with numerical IK
        [q_desired, solnInfo] = ik('tool0', Td, ones(6,1), q);

        % Calculate error for PD control
        error = q_desired - q;
        dError = (error - prevError) / dt;

        % PD control law
        u = Kp * error + Kd * dError;

        % Update joint positions
        q = q + u * dt;
        prevError = error;

        % Update plot
        show(r, q, ...
```

```
            'Visuals', 'on', ...
            'PreservePlot', 0, ...
            'Frames', 'off', ...
            'Parent', ax);
        plotTransforms(Td(1:3,4)', tform2quat(Td), ...
            'Parent', ax, ...
            'framesize', 0.05);
        plot3(ax, x, y, z, 'm', 'LineWidth', 1);
        drawnow;
    end

    % Update current position to the position of the last point in the spiral
    currentPosition = [x(end), y(end), z(end)];

    % Generate Transition Trajectory using cubicpolytraj if it's not the last plate
    if visitCount < numPlates
        % Find the nearest unvisited plate again for the next target
        distances = sqrt(sum((platePositions(:,1:2) - currentPosition(1:2)).^2, 2));
        distances(visitedPlates) = inf;  % Ignore already visited plates
        [~, nextIdx] = min(distances);  % Find the index of the nearest plate
        nextPlatePosition = platePositions(nextIdx, :);

        nTransitionPoints = 50;
        transitionTime = linspace(0, 1, nTransitionPoints);

        % Calculate the angles for the waypoints around the robot
        angle1 = atan2(currentPosition(2), currentPosition(1));
        angle2 = atan2(nextPlatePosition(2), nextPlatePosition(1));

        % Normalize angles to range [0, 2*pi]
        if angle1 < 0
            angle1 = angle1 + 2*pi;
        end
        if angle2 < 0
            angle2 = angle2 + 2*pi;
        end

        % Calculate the shortest arc
        if angle2 < angle1
            angle2 = angle2 + 2*pi;
        end
        clockwiseDistance = angle2 - angle1;
        counterClockwiseDistance = 2*pi - clockwiseDistance;

        if clockwiseDistance <= counterClockwiseDistance
            midAngles = linspace(angle1, angle2, 5);
        else
            midAngles = linspace(angle1, angle2 - 2*pi, 5);
        end

        midRadius = max(baseRadius + 0.1, 0.3);  % Ensure radius is large enough to avoid the robot

        % Create waypoints around the robot
        midPoints = [midRadius*cos(midAngles') midRadius*sin(midAngles') repmat(z_above_plate + 0.2, length(midAngles), 1)];
        waypoints = [currentPosition; midPoints; nextPlatePosition(1), nextPlatePosition(2), z_plate + z_above_plate];
        waypointsTime = linspace(0, 1, size(waypoints, 1));

        [transitionTraj, ~, ~] = cubicpolytraj(waypoints', waypointsTime, transitionTime);
        xTransition = transitionTraj(1, :);
        yTransition = transitionTraj(2, :);
        zTransition = transitionTraj(3, :);

        % Initialize error terms for PD control
        prevError = zeros(6,1);

        % Draw transition trajectory
        for i = 1:nTransitionPoints
            % Desired position and orientation for the transition
            Td = trvec2tform([xTransition(i), yTransition(i), zTransition(i)]) * quat2tform(orientation);

            % Find pose with numerical IK
            [q_desired, solnInfo] = ik('tool0', Td, ones(6,1), q);

            % Calculate error for PD control
            error = q_desired - q;
            dError = (error - prevError) / dt;

            % PD control law
            u = Kp * error + Kd * dError;

            % Update joint positions
            q = q + u * dt;
            prevError = error;

            % Update plot
            show(r, q, ...
                'Visuals', 'on', ...
                'PreservePlot', 0, ...
                'Frames', 'off', ...
                'Parent', ax);
            plotTransforms(Td(1:3,4)', tform2quat(Td), ...
                'Parent', ax, ...
                'framesize', 0.05);
            drawnow;
        end

        % Update current position to the position of the last point in the transition
        currentPosition = [xTransition(end), yTransition(end), zTransition(end)];
    end
end

% ----------------------------------------------------------------
% Function to draw the floor
% ----------------------------------------------------------------
function drawFloor()
    ax = gca;
    ax.CameraViewAngle = 5;
    p = patch([1 -1 -1 1].*0.5, [1 1 -1 -1]*0.5, [0 0 0 0]);
    p.FaceColor = [0.8, 0.8, 0.8];
    axis off;
    xlim([-0.75, 0.75]);
```

```
    ylim([-0.75, 0.75]);
    zlim([0, 0.75]);
end
```