Robotix Induction Task 1

EE24BTECH11002 Agamjot Singh
IIT Hyderabad

May 17, 2025

Task 1- Mathematical Modelling

Write a MATLAB/Python program to simulate a 2 DOF robotic arm moving in a 2-D space. The end effector traces a circle defined by $(x-a)^2+(y-b)^2=r^2$ with uniform speed v. Each link has a length of 1000mm = 1m. The program takes a, b, r and v as input parameters and generates an animation of the links with the end effector starting at the coordinate (a+r, b). The program should also generate the angle the links make with x axis, angular velocity and angular acceleration required at the actuators vs time. If any point on the circle is out of reach of the arm, the program should return "Out of reach".

Mathematical Model

Inverse Kinematics Problem

For our 2-link planar arm, we need to find joint angles θ_1 and θ_2 that position the end-effector at a desired point (x, y) on the circle. This is trickier than it sounds because there are usually two possible configurations (elbow-up or elbow-down) for any reachable point.
We will use the elbow-down configuration. Inverse Kinematics for the same are given below, derivations for which are excluded.

$$r = \sqrt{x^2+y^2} \tag{1}$$

$$\cos\theta_2 = \frac{r^2 - L_1^2 - L_2^2}{2L_1L_2} \tag{2}$$

$$\theta_2 = \text{atan2}(-\sqrt{1-\cos^2\theta_2}, \cos\theta_2) \tag{3}$$

$$\beta = \text{atan2}(y, x) \tag{4}$$

$$\gamma = \text{atan2}(L_2\sin\theta_2, L_1 + L_2\cos\theta_2) \tag{5}$$

$$\theta_1 = \beta - \gamma \tag{6}$$

The negative square root in the θ_2 calculation forces the elbow-down configuration.

Code Explanation

Setting Up the Simulation

(a, b) is the center of the circle to be traced, v is the velocity of the end effector and radius of course is the radius of the circle.
Set $\omega = v/r$ to ensure constant tangential velocity along the circle.

```
function robotic_arm_circle(a, b, radius, v)
    % Link lengths = 1 m
    L1 = 1;
    L2 = 1;

    % Angular Velocity of end effector = omega
    omega = v/radius;

    figure;
    hold on;
    axis equal;
    grid on;
    title('2-Link Robotic Arm Circle Path Following');
    xlabel('X');
    ylabel('Y');
```

Tracking previous values to calculate angular velocities and accelerations of each of the two actuators later on.

```
    prev_theta1 = 0;
    prev_omega1 = 0;
    prev_theta2 = 0;
    prev_omega2 = 0;
    dt = 0.1; % time step
```

Inverse Kinematics Implementation

The core of the program is the inverse kinematics calculation,

```
    % (x, y) is the coordinate of the end effector
    x = a + radius*cos(omega*t);
    y = b + radius*sin(omega*t);

    % r distance from origin (not center)
    r = sqrt(x^2 + y^2);

    % Checking for out of reach points
    if r > L1 + L2
        fprintf("Out of reach: (%f, %f)\n", x, y);
        continue;
```

```
end

% Applying inverse kinematics to get the angles
cos_theta2 = (r^2 - L1^2 - L2^2) / (2 * L1 * L2);
theta2 = atan2(-sqrt(1 - cos_theta2^2), cos_theta2
    );
beta = atan2(y, x);
gamma = atan2(L2 * sin(theta2), L1 + L2 * cos(
    theta2));
theta1 = beta - gamma;
```

Here t goes from 0 to 1000 for the animation sake, with a step size of dt. The condition $r > L1 + L2$ checks if the point is beyond the arm's maximum reach.

Calculating Angular Velocities and Accelerations of the Actuators

Using finite differences to approximate derivatives. This is because analytical methods for calculating both velocity and acceleration are hard to get. I mean, velocity is still doable, but acceleration is where you eventually give up.

```
% Finding angular velocity and acceleration
omega1 = (theta1 - prev_theta1)/dt;
omega2 = (theta2 - prev_theta2)/dt;

alpha1 = (omega1 - prev_omega1)/dt;
alpha2 = (omega2 - prev_omega2)/dt;

prev_theta1 = theta1;
prev_theta2 = theta2;
prev_omega1 = omega1;
prev_omega2 = omega2;
```

This isn't the most accurate method, but it works well enough for visualization purposes.

Animation/Plotting

The animation/plotting code updates the arm configuration at each time step,

```
% Plotting the arm and circle
x1 = L1 * cos(theta1);
y1 = L1 * sin(theta1);
x2 = x1 + L2 * cos(theta1 + theta2);
y2 = y1 + L2 * sin(theta1 + theta2);
cla;
```

```
plot(x_circle, y_circle, 'k');
plot(0, 0, 'ks', 'MarkerSize', 10, '
    MarkerFaceColor', 'k');
plot([0, x1], [0, y1], 'b', 'LineWidth', 6);
plot([x1, x2], [y1, y2], 'r', 'LineWidth', 8);
```

Quirky Observations

When running the simulation with various parameters, I noticed several interesting behaviors.

- Near singularities (when the arm is fully extended), the angular velocities spike dramatically
- The elbow-down configuration sometimes causes the arm to take unexpected paths which are pretty counter-intuitive

The error calculation shows how accurately the end-effector follows the desired path:

```
err = sqrt((x2-x)^2 + (y2-y)^2);
text(-18, 18, sprintf('Error: %.2f units', err));
```

In practice, this error should be near zero, but numerical issues can cause small deviations.