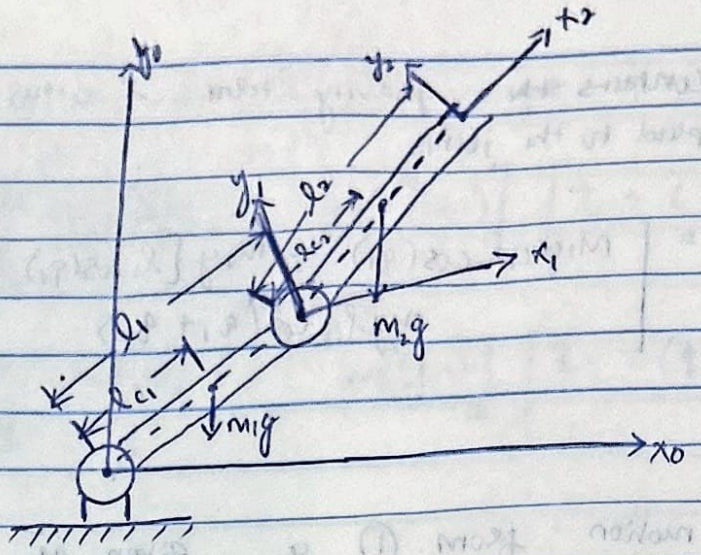
ROB-599 . HW-3

Date :- 11/10/2024



The eqn of motion of this robot is

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + N(q) = \tau \quad \text{--- (1)}$$

Where

→ q is the generalized co-ordinate vector, in this case $q = [q_1, q_2]^T$

→ $M(q)$ is the mass/Inertia Matrix of the robot, defined as

$$M(1,1) = m_1 l_{c1}^2 + m_2\left[l_1^2 + l_{c2}^2 + 2 l_1 l_{c2}\cos(q_2)\right] + I_1 + I_2$$

$$M(1,2) = m_2\left[l_{c2}^2 + l_1 l_{c2}\cos(q_2)\right] + I_2$$

$$M(2,1) = M(1,2) = m_2\left[l_{c2}^2 + l_1 l_{c2}\cos(q_2)\right] + I_2$$

$$M(2,2) = m_2 l_{c2}^2 + I_2$$

where I_1 & I_2 are the moment of Inertia of both Rinks

→ $C(q,\dot{q})$ is the coriolis matrix of the robot, defined as

$$C(1,1) = -m_2 l_1 l_{c2}\sin(q_2)\dot{q}_2 \qquad C(1,2) = -m l_1 l_{c2}\sin(q_2)(\dot{q}_1+\dot{q}_2)$$

$$C(2,1) = m_2 l_1 l_{c2}\sin(q_2)\dot{q}_1 \qquad C(2,2) = 0$$

$N(q)$ ⇒ contains the gravity term & other conservative forces applied to the joints

$$N(q) = \begin{bmatrix} M_1 g l_{c1} \cos(q_1) + m_2 g [l_1 \cos(q_1) + l_{c2} \cos(q_1 + q_2)] \\ m_2 g l_{c2} \cos(q_1 + q_2) \end{bmatrix}$$

problem-1 Eqⁿ of motion from (1) is given as

$$M(q)\ddot{q} + C(q,\dot{q})\cdot\dot{q} + N(q) = \tau$$

$$M(q)\ddot{q} = \tau - C(q,\dot{q})\dot{q} - N(q)$$

$$\Rightarrow \ddot{q} = M^{-1}(q)\left[\tau - C(q,\dot{q})\cdot\dot{q} - N(q)\right] \quad \text{---} (2)$$

Now $M^{-1}(q) = M^{-1}$ ⇒ $\ddot{q}_1 = \frac{1}{\det(M)}\left[M^{-1}(1,1) \quad M^{-1}(1,2)\right]\left[\tau - C(q,\dot{q})\cdot\dot{q} - N(q)\right]$

(i.e ⇒ this is the first row of 2) — (3)

$$\ddot{q}_2 = \frac{1}{\det(M)}\left[M^{-1}(2,1) \quad M^{-1}(2,2)\right]\left[\tau - C(q,\dot{q})\cdot\dot{q} - N(q)\right]$$

(i.e ⇒ this is the 2nd row of (2)) — (4)

$$M^{-1} = \frac{1}{\det(M)}\begin{bmatrix} M(2,2) & -M(2,1) \\ -M(1,2) & M(1,1) \end{bmatrix} \quad \text{---} (5)$$

Sub (5) in eq^n (3) & (4) we get.

$$\ddot{q}_1 = \frac{1}{\det(M)} \begin{bmatrix} M(2,2) & -M(2,1) \end{bmatrix} \begin{bmatrix} \tau - c(q,\dot{q})\cdot\dot{q} - N(q) \end{bmatrix}$$

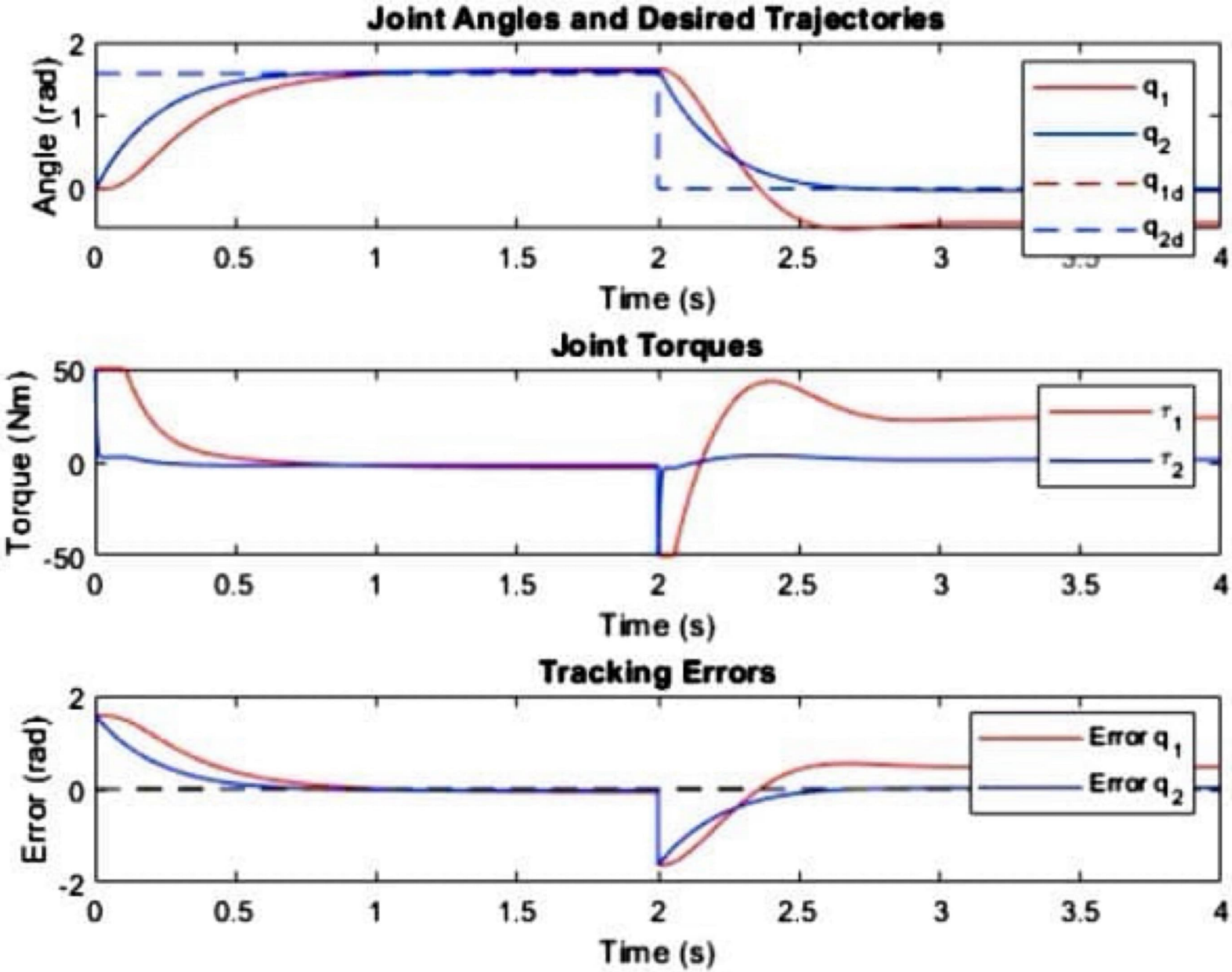
$$\ddot{q}_2 = \frac{1}{\det(M)} \begin{bmatrix} -M(1,2) & M(1,1) \end{bmatrix} \begin{bmatrix} \tau - c(q,\dot{q})\dot{q} - N(q) \end{bmatrix}$$

(2)

```
%P2

% Define the time span and initial conditions
tspan = [0 4]; % from 0 to 4 seconds
x0 = [0; 0; 0; 0]; % initial conditions for [q1, q1dot, q2, q2dot]

% Define the desired trajectory as a function of time
function qd = desired_trajectory(t)
    if t < 2
        qd = [pi/2; pi/2]; % step to π/2 radians
    else
        qd = [0; 0]; % step back to 0 radians
    end
end

% Define the system of differential equations
function xdot = robot_dynamics(t, x, para)
    % Extract joint positions and velocities
    q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

    % Desired trajectory at time t
    qd = desired_trajectory(t);
    q1d = qd(1);
    q2d = qd(2);

    % Mass/inertia matrix D(q)
    D = zeros(2, 2);
    D(1,1) = para.m1*para.lc1^2 + para.m2*(para.l1^2 + para.lc2^2 + 2*para.l1*para.
lc2*cos(q2)) + para.I1 + para.I2;
    D(1,2) = para.m2*(para.lc2^2 + para.l1*para.lc2*cos(q2)) + para.I2;
    D(2,1) = D(1,2);
    D(2,2) = para.m2*para.lc2^2 + para.I2;

    % Coriolis matrix C(q,qdot)
    C = zeros(2, 2);
    C(1, 1) = -para.m2*para.l1*para.lc2*sin(q2)*q2dot;
    C(1, 2) = -para.m2*para.l1*para.lc2*sin(q2)*(q1dot + q2dot);
    C(2, 1) = para.m2*para.l1*para.lc2*sin(q2)*q1dot;
    C(2, 2) = 0;

    % Gravity terms N(q)
    N = zeros(2, 1);
    N(1) = para.m1*para.g*para.lc1*cos(q1) + para.m2*para.g*(para.l1*cos(q1) + para.
lc2*cos(q1 + q2));
    N(2) = para.m2*para.g*para.lc2*cos(q1 + q2);

    % PD control torques
    tau = zeros(2, 1);
    tau(1) = para.Kp1*(q1d - q1) - para.Kd1*q1dot;
```

```
    tau(2) = para.Kp2*(q2d - q2) - para.Kd2*q2dot;

    % Limit the torques to the range [-50, 50]
    tau(1) = max(min(tau(1), 50), -50);
    tau(2) = max(min(tau(2), 50), -50);

    % Calculate accelerations
    a = tau - C * [q1dot; q2dot] - N;

    % Solve for joint accelerations using inverse of D matrix
    qddot = D \ a;

    % Return derivative of state vector
    xdot = [q1dot; qddot(1); q2dot; qddot(2)];
end

% Define parameters for the robot arm
para.m1 = 7.848;
para.m2 = 4.49;
para.l1 = 0.3;
para.lc1 = 0.1554;
para.lc2 = 0.0341;
para.I1 = 0.176;
para.I2 = 0.0411;
para.g = 9.81;
para.Kp1 = 50;
para.Kp2 = 50;
para.Kd1 = 10;
para.Kd2 = 10;

% Solve the differential equations using ODE45
[t, x] = ode45(@(t,x) robot_dynamics(t, x, para), tspan, x0);

% Calculate torques and tracking errors over time
tau1 = zeros(length(t), 1);
tau2 = zeros(length(t), 1);
tracking_error_q1 = zeros(length(t), 1);
tracking_error_q2 = zeros(length(t), 1);

for i = 1:length(t)
    qd = desired_trajectory(t(i));
    q1d = qd(1);
    q2d = qd(2);

    % Extract current states
    q1 = x(i, 1);
    q1dot = x(i, 2);
    q2 = x(i, 3);
    q2dot = x(i, 4);
```

```
    % Calculate PD control torques
    tau1(i) = para.Kp1 * (q1d - q1) - para.Kd1 * q1dot;
    tau2(i) = para.Kp2 * (q2d - q2) - para.Kd2 * q2dot;

    % Limit the torques to the range [-50, 50]
    tau1(i) = max(min(tau1(i), 50), -50);
    tau2(i) = max(min(tau2(i), 50), -50);

    % Calculate tracking errors
    tracking_error_q1(i) = q1d - q1;
    tracking_error_q2(i) = q2d - q2;
end

% Plot joint responses with desired trajectory
figure;
subplot(3, 1, 1);
plot(t, x(:, 1), 'r', t, x(:, 3), 'b');
hold on;
qd_plot = arrayfun(@desired_trajectory, t, 'UniformOutput', false);
qd1 = cellfun(@(qd) qd(1), qd_plot); % Extract q1d for all time steps
qd2 = cellfun(@(qd) qd(2), qd_plot); % Extract q2d for all time steps
plot(t, qd1, '--r', t, qd2, '--b'); % Plot desired trajectories
hold off;
title('Joint Angles and Desired Trajectories');
xlabel('Time (s)');
ylabel('Angle (rad)');
legend('q_1', 'q_2', 'q_{1d}', 'q_{2d}');

% Plot joint torques
subplot(3, 1, 2);
plot(t, tau1, 'r', t, tau2, 'b');
title('Joint Torques');
xlabel('Time (s)');
ylabel('Torque (Nm)');
legend('\tau_1', '\tau_2');
ylim([-50 50]);  % Set y-axis limits from -100 to +100


% Plot tracking errors
subplot(3, 1, 3);
plot(t, tracking_error_q1, 'r', t, tracking_error_q2, 'b');
hold on;
plot(t, zeros(size(t)), '--k'); % Plot zero line for perfect tracking
hold off;
title('Tracking Errors');
xlabel('Time (s)');
ylabel('Error (rad)');
legend('Error q_1', 'Error q_2');
```


Joint Angles and Desired Trajectories
Angle (rad)
Time (s)
q1
q2
q1d
q2d
Joint Torques
Torque (Nm)
Time (s)
τ1
τ2
Tracking Errors
Error (rad)
Time (s)
Error q1
Error q2

In theory, the answer is yes, but with certain conditions and limitations:

Proportional Gain (Kp):

Increasing Kp can reduce the steady-state error. A higher Kp will make the system more responsive by increasing the corrective torque based on the error in position.

However, a Kp that is too high can cause overshooting and oscillations, where the system overshoots the desired value and keeps oscillating around it before settling down.

Derivative Gain (Kd):

Increasing Kd improves the damping of the system. It reduces oscillations and overshoots by responding to the rate of change of the error (i.e., velocity).

However, if Kd is too high, it can make the system sluggish or even unstable in the presence of noise, since the derivative term amplifies high-frequency signals (like noise or rapid changes).

Eliminating the Error:

There is no integral term for a PD controller, so steady-state error might not completely disappear unless the proportional and derivative gains are perfectly tuned.

To completely eliminate steady-state errors, a PI (Proportional-Integral) or PID controller (adding an integral term) is typically required.

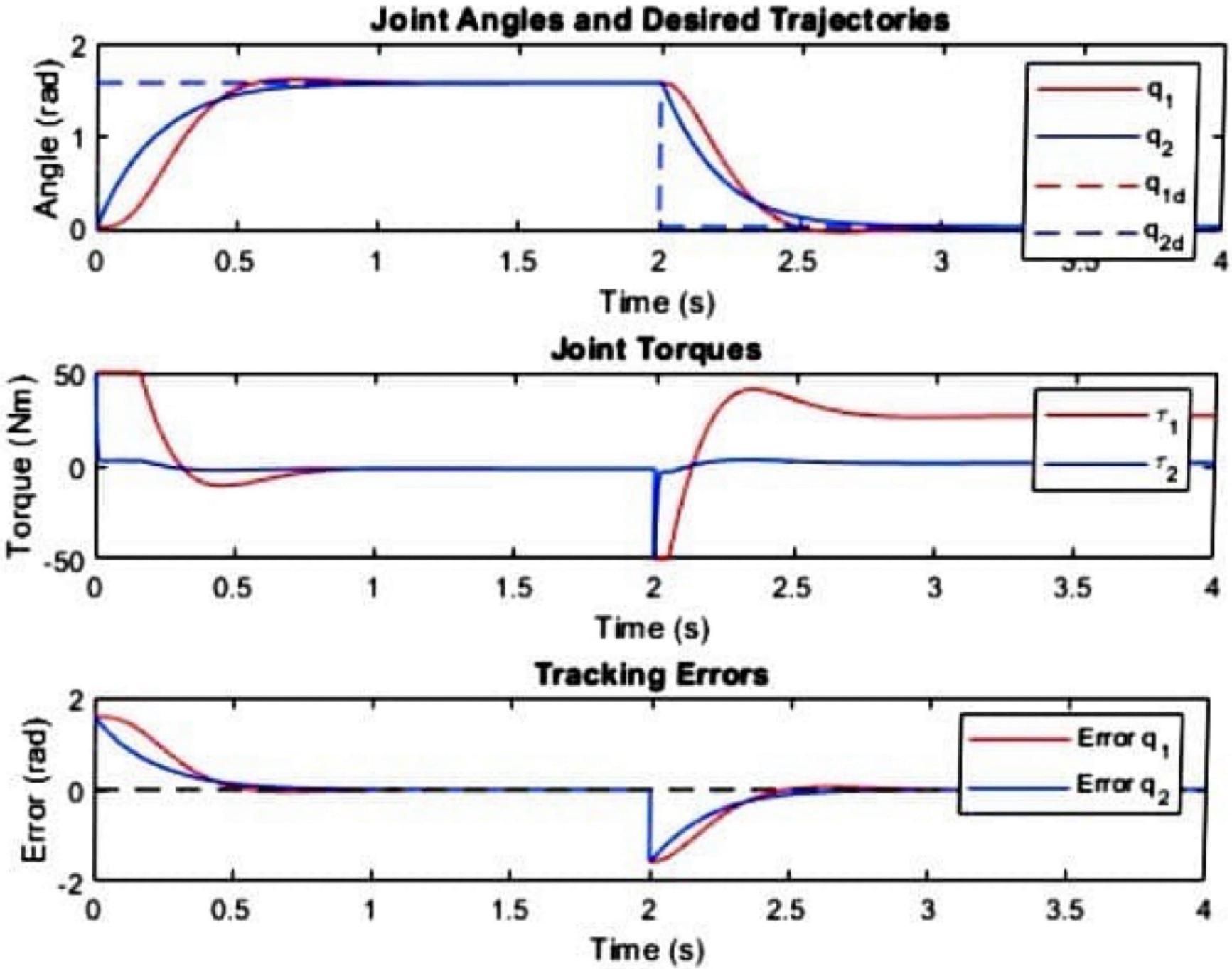
With PD control, reducing the tracking error largely depends on how well-tuned Kp and Kd are. The best you can achieve is minimizing the error, but it may not be possible to eliminate it without integral action.

```
%P3

% Define the time span and initial conditions
tspan = [0 4]; % from 0 to 4 seconds
x0 = [0; 0; 0; 0]; % initial conditions for [q1, q1dot, q2, q2dot]

% Define the desired trajectory as a function of time
function qd = desired_trajectory(t)
    if t < 2
        qd = [pi/2; pi/2]; % step to π/2 radians
    else
        qd = [0; 0]; % step back to 0 radians
    end
end

% Define the system of differential equations
function xdot = robot_dynamics(t, x, para)
    % Extract joint positions and velocities
    q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

    % Desired trajectory at time t
    qd = desired_trajectory(t);
    q1d = qd(1);
    q2d = qd(2);

    % Mass/inertia matrix D(q)
    D = zeros(2, 2);
    D(1,1) = para.m1*para.lc1^2 + para.m2*(para.l1^2 + para.lc2^2 + 2*para.l1*para.
lc2*cos(q2)) + para.I1 + para.I2;
    D(1,2) = para.m2*(para.lc2^2 + para.l1*para.lc2*cos(q2)) + para.I2;
    D(2,1) = D(1,2);
    D(2,2) = para.m2*para.lc2^2 + para.I2;

    % Coriolis matrix C(q,qdot)
    C = zeros(2, 2);
    C(1, 1) = -para.m2*para.l1*para.lc2*sin(q2)*q2dot;
    C(1, 2) = -para.m2*para.l1*para.lc2*sin(q2)*(q1dot + q2dot);
    C(2, 1) = para.m2*para.l1*para.lc2*sin(q2)*q1dot;
    C(2, 2) = 0;

    % Gravity terms N(q)
    N = zeros(2, 1);
    N(1) = para.m1*para.g*para.lc1*cos(q1) + para.m2*para.g*(para.l1*cos(q1) + para.
lc2*cos(q1 + q2));
    N(2) = para.m2*para.g*para.lc2*cos(q1 + q2);

    % PD control torques
    tau = zeros(2, 1);
    tau(1) = para.Kp1*(q1d - q1) - para.Kd1*q1dot + N(1);
```

```
    tau(2) = para.Kp2*(q2d - q2) - para.Kd2*q2dot + N(2);

    % Limit the torques to the range [-50, 50]
    tau(1) = max(min(tau(1), 50), -50);
    tau(2) = max(min(tau(2), 50), -50);

    % Calculate accelerations
    a = tau - C * [q1dot; q2dot] - N;

    % Solve for joint accelerations using inverse of D matrix
    qddot = D \ a;

    % Return derivative of state vector
    xdot = [q1dot; qddot(1); q2dot; qddot(2)]; % x_dot returns the velocities which
are required so that we can plug it to the ODE 45
end

% Define the parameters in the 'para' structure
para.m1 = 7.848;
para.m2 = 4.49;
para.l1 = 0.3;
para.lc1 = 0.1554;
para.lc2 = 0.0341;
para.I1 = 0.176;
para.I2 = 0.0411;
para.g = 9.81;
para.Kp1 = 50;
para.Kp2 = 50;
para.Kd1 = 10;
para.Kd2 = 10;

% Solve the differential equations using ODE45
[t, x] = ode45(@(t,x) robot_dynamics(t, x, para), tspan, x0); % we input x_dot which
is calculated using the robot_dynamics function

% Preallocate arrays for torques and tracking errors
tau1 = zeros(length(t), 1);
tau2 = zeros(length(t), 1);
tracking_error_q1 = zeros(length(t), 1);
tracking_error_q2 = zeros(length(t), 1);

% Loop over time points to calculate torques and tracking errors
for i = 1:length(t)
    qd = desired_trajectory(t(i));
    q1d = qd(1);
    q2d = qd(2);

    % Extract current states
    q1 = x(i, 1);
```

```
    q1dot = x(i, 2);
    q2 = x(i, 3);
    q2dot = x(i, 4);

    % Recompute gravity terms N1 and N2 based on current q1 and q2
    N1 = para.m1*para.g*para.lc1*cos(q1) + para.m2*para.g*(para.l1*cos(q1) + para.
lc2*cos(q1 + q2));
    N2 = para.m2*para.g*para.lc2*cos(q1 + q2);

    % Calculate PD control torques with gravity compensation
    tau1(i) = para.Kp1 * (q1d - q1) - para.Kd1 * q1dot + N1;
    tau2(i) = para.Kp2 * (q2d - q2) - para.Kd2 * q2dot + N2;

    % Limit the torques to the range [-50, 50]
    tau1(i) = max(min(tau1(i), 50), -50);
    tau2(i) = max(min(tau2(i), 50), -50);

    % Calculate tracking errors
    tracking_error_q1(i) = q1d - q1;
    tracking_error_q2(i) = q2d - q2;
end

% Plot joint responses with desired trajectory
figure;
subplot(3, 1, 1);
plot(t, x(:, 1), 'r', t, x(:, 3), 'b');
hold on;
qd_plot = arrayfun(@desired_trajectory, t, 'UniformOutput', false);
qd1 = cellfun(@(qd) qd(1), qd_plot); % Extract q1d for all time steps
qd2 = cellfun(@(qd) qd(2), qd_plot); % Extract q2d for all time steps
plot(t, qd1, '--r', t, qd2, '--b'); % Plot desired trajectories
hold off;
title('Joint Angles and Desired Trajectories');
xlabel('Time (s)');
ylabel('Angle (rad)');
legend('q_1', 'q_2', 'q_{1d}', 'q_{2d}');

% Plot joint torques
subplot(3, 1, 2);
plot(t, tau1, 'r', t, tau2, 'b');
title('Joint Torques');
xlabel('Time (s)');
ylabel('Torque (Nm)');
legend('\tau_1', '\tau_2');
ylim([-50 50]);  % Set y-axis limits from -100 to +100


% Plot tracking errors
subplot(3, 1, 3);
```

```
plot(t, tracking_error_q1, 'r', t, tracking_error_q2, 'b');
hold on;
plot(t, zeros(size(t)), '--k'); % Plot zero line for perfect tracking
hold off;
title('Tracking Errors');
xlabel('Time (s)');
ylabel('Error (rad)');
legend('Error q_1', 'Error q_2');
```


Joint Angles and Desired Trajectories
Angle (rad)
Time (s)
q1
q2
q1d
q2d
Joint Torques
Torque (Nm)
Time (s)
τ1
τ2
Tracking Errors
Error (rad)
Time (s)
Error q1
Error q2

1. Joint Angles and Desired Trajectories:
   - The second controller (with gravity compensation) shows closer alignment
     between q1, q2 and the desired trajectories q1d, q2d.
   - The first controller exhibits larger deviations from the desired trajectories,
     particularly in the presence of significant gravitational forces.

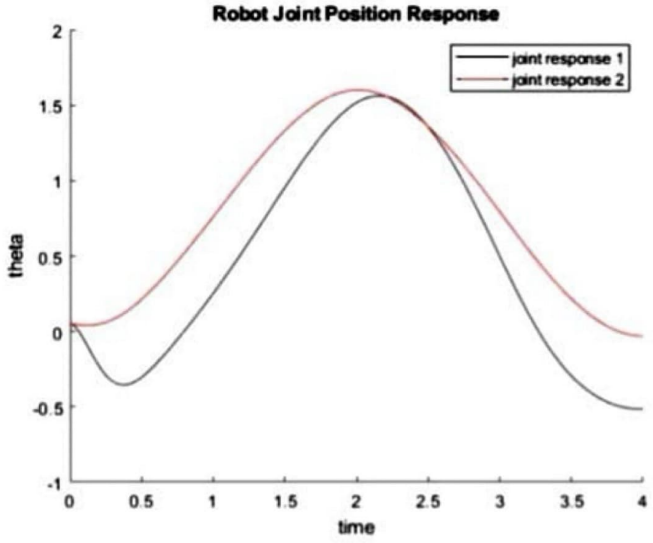
2. Joint Torques:
   - The second controller produces lower and smoother torques (tau1, tau2)
     due to gravity compensation.
   - The first controller requires higher and more variable torques to overcome
     unmodeled gravitational forces.

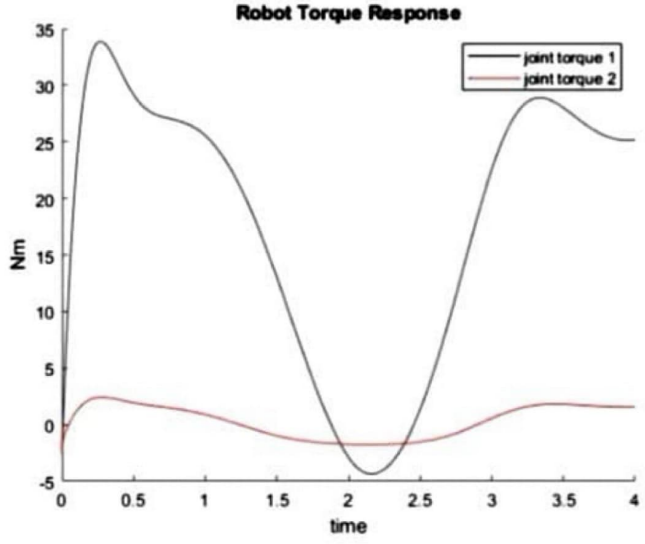
3. Tracking Errors:
   - The second controller results in smaller tracking errors and faster convergence
     to zero, compensating for gravitational forces.
   - The first controller shows larger tracking errors and slower convergence,
     particularly when gravitational forces are significant.

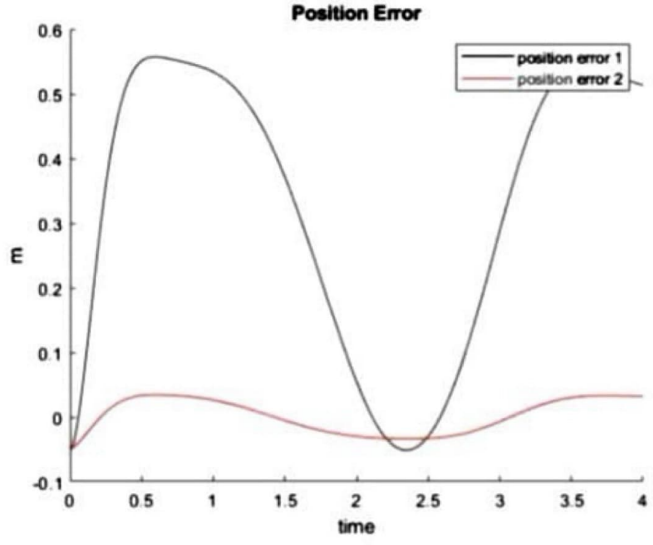
```
%HW 3 P4
clear
close all
clc
function [xdot,tau,error,cmd_val] = HW3_Q6(t,x)
q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

m1 = 7.848;m2 = 4.49; l1 = 0.3;lc1 = 0.1554;
lc2 = 0.0341;I1 = 0.176;I2 = 0.0411;
g = 9.81;

M(1,1) = m1*lc1^2 + m2*(l1^2 + lc2^2 + 2*l1*lc2*cos(q2)) + I1 + I2;
M(1,2) = m2*(lc2^2 + l1*lc2*cos(q2)) + I2;
M(2,1) = M(1,2);
M(2,2) = m2*lc2^2 + I2;

% Coriolis matrix
C(1, 1) = -m2*l1*lc2*sin(q2)*q2dot;
C(1, 2) = -m2*l1*lc2*sin(q2)*(q1dot + q2dot);
C(2, 1) = m2*l1*lc2*sin(q2)*q1dot;
C(2, 2) = 0;

%External forces (gravity etc)
N(1,1) = m1*g*lc1*cos(q1) + m2*g*(l1*cos(q1)+lc2*cos(q1 +q2));
N(2,1) = m2*g*lc2*cos(q1 + q2);

%defining the control law
[qd, vd, ad] = cubicpoly([0, 2, 4], [0, pi/2, 0], [0, 0, 0], t);
cmd_val = [qd, vd, ad];
Kp1 = 50 ; Kd1 =10;
Kp2 = 50; Kd2 = 10;

error = [qd - q1 qd - q2 vd - q1dot vd - q2dot]';

tau(1,1) = Kp1*(qd - q1) + Kd1*(vd - q1dot);
tau(2,1) = Kp2*(qd - q2) + Kd2*(vd - q2dot);


%employing the limitations
if (tau(1,1)<= -50)
tau(1,1) = -50;
end
if (tau(1,1) >= 50)
tau(1,1) = 50;
end
if (tau(2,1)<= -50)
tau(2,1) = -50;
end
if (tau(2,1) >= 50)
```

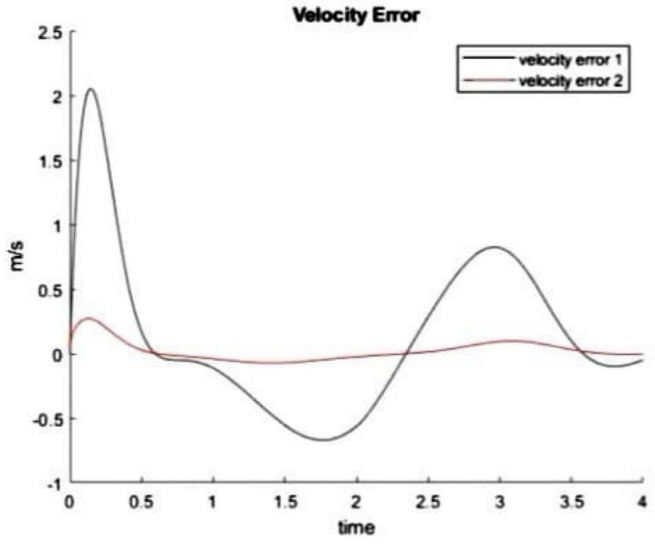
```
tau(2,1) = 50;
end

xdot = inv(M)*(tau - N -C*[q1dot;q2dot]);
xdot(4,1) = xdot(2,1);
xdot(2,1) = xdot(1,1);
xdot(1,1) = x(2);
xdot(3,1) = x(4);

end %function end

q0 = [0.05 0 0.05 0]';
[t,x]=ode45(@(t,x) HW3_Q6(t,x),0:0.01:4,q0);
tau = zeros(length(t), 2);
error = zeros(length(t), 4);
cmd_val = zeros(length(t), 3);
for i = 1:length(t)
[~, tau(i, :),error(i,:),cmd_val(i,:)] = HW3_Q6(t(i), x(i, :));
end

figure(1)
hold on
plot(t,x(:,1),'black')
plot(t,x(:,3),'red')
xlabel('time')
ylabel('theta')
title('Robot Joint Position Response')
legend('joint response 1','joint response 2')
hold off

figure(2)
hold on
plot(t,tau(:,1),'black')
plot(t,tau(:,2),'red')
xlabel('time')
ylabel('Nm')
title('Robot Torque Response')
legend('joint torque 1','joint torque 2')
hold off

figure(3)
hold on
plot(t,error(:,1),'black')
plot(t,error(:,2),'red')
title('Position Error')
xlabel('time')
ylabel('m')
legend('position error 1','position error 2')
hold off
```

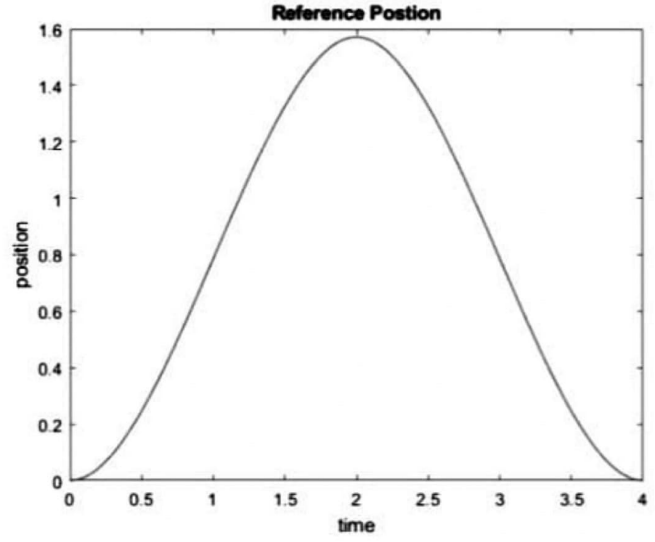
```
figure(4)
hold on
plot(t,error(:,3),'black')
plot(t,error(:,4),'red')
title('Velocity Error')
xlabel('time')
ylabel('m/s')
legend('velocity error 1','velocity error 2')
hold off

figure(5)
plot(t,cmd_val(:,1),'black')
xlabel('time')
ylabel('position')
title('Reference Postion')
figure(6)
plot(t,cmd_val(:,2),'black')
xlabel('time')
ylabel('velocity')
title('Reference Velocity')
figure(7)
plot(t,cmd_val(:,3),'black')
xlabel('time')
ylabel('acceleration')
title('Reference Acceleration')
```

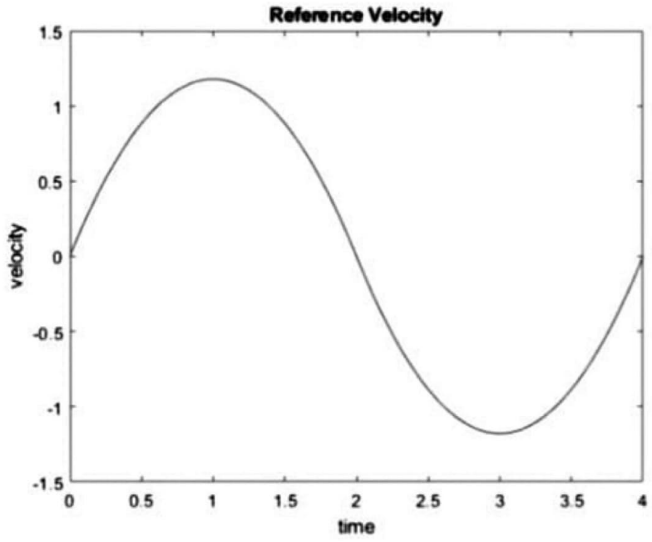

Robot Joint Position Response
joint response 1
joint response 2
theta
time

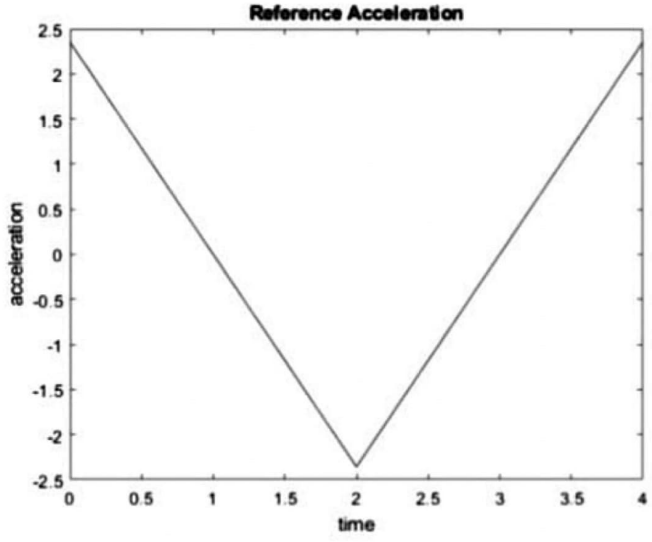

Robot Torque Response
joint torque 1
joint torque 2
Nm
time


Position Error
position error 1
position error 2
m
time


Velocity Error
velocity error 1
velocity error 2
m/s
time


Reference Postion
position
time





Published with MATLAB® R2024a

Controller Gains (Kp, Kd) Tuning:
If the proportional gain (Kp) is too low, the system may not respond aggressively enough to follow the desired trajectory closely.
If the derivative gain (Kd) is too low, it can result in poor damping and cause oscillations or a lag in tracking, as seen in the actual joint positions.
On the other hand, if Kp or Kd is too high, it could lead to high-frequency oscillations or even instability (which seems visible in the high-frequency ripples in acceleration plots).

Actuator Limits:
The system may have physical limitations, such as actuator speed or torque constraints, that prevent it from following rapid changes in position or velocity accurately. This is evident if the torque output saturates (seen in the sharp torque transitions).

Nonlinear Dynamics:
PD controllers are generally effective for linear systems or small deviations around an operating point. However, if your system has nonlinearities (e.g., friction, backlash, or time-varying dynamics), a simple PD controller may not suffice for precise tracking of complex trajectories.

```
%P5

function [qd1, vd1, ad1, qd2, vd2, ad2] = generate_trajectory(t)
    % Time breakpoints
    t_waypoints = [0 2 4];

    % Waypoints for position q(t) for both joints
    waypoints = [0, pi/2, 0];    % Desired positions for both joints

    % Desired velocities for both joints
    velocities = [0, 0, 0];      % Desired velocities for both joints

    % Generate cubic polynomial trajectory
    [qd, vd, ad] = cubicpolytraj(waypoints, t_waypoints, t, ...
                                 'VelocityBoundaryCondition', velocities);

    % Assign the same trajectory to both joints
    qd1 = qd; vd1 = vd; ad1 = ad;   % For joint 1
    qd2 = qd; vd2 = vd; ad2 = ad;   % For joint 2
end

function xdot = robot_dynamics(t, x, para)
    % Extract joint positions and velocities
    q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

    % Desired trajectory at time t
    [qd, vd, ad] = generate_trajectory(t);  % Get desired position, velocity,
acceleration
    q1d = qd;  % Same trajectory for both q1 and q2
    q2d = qd;

    % Ensure that vd and ad are vectors with two identical elements
    Vd = [vd; vd];  % Both joints have the same desired velocity
    Ad = [ad; ad];  % Both joints have the same desired acceleration

    % Mass/inertia matrix D(q)
    D = zeros(2, 2);
    D(1,1) = para.m1 * para.lc1^2 + para.m2 * (para.l1^2 + para.lc2^2 + 2 * para.l1 *
para.lc2 * cos(q2)) + para.I1 + para.I2;
    D(1,2) = para.m2 * (para.lc2^2 + para.l1 * para.lc2 * cos(q2)) + para.I2;
    D(2,1) = D(1,2);
    D(2,2) = para.m2 * para.lc2^2 + para.I2;

    % Coriolis matrix C(q,qdot)
    C = zeros(2, 2);
    C(1, 1) = -para.m2 * para.l1 * para.lc2 * sin(q2) * q2dot;
    C(1, 2) = -para.m2 * para.l1 * para.lc2 * sin(q2) * (q1dot + q2dot);
    C(2, 1) = para.m2 * para.l1 * para.lc2 * sin(q2) * q1dot;
    C(2, 2) = 0;
```

```
    % Gravity terms N(q)
    N = zeros(2, 1);
    N(1) = para.m1 * para.g * para.lc1 * cos(q1) + para.m2 * para.g * (para.l1 * cos
(q1) + para.lc2 * cos(q1 + q2));
    N(2) = para.m2 * para.g * para.lc2 * cos(q1 + q2);

    % Feed Forward
    Tff = D*Ad + C*Vd + N;

    % PD control torques with desired velocity feedback
    tau1 = Tff(1)+para.Kp1 * (q1d - q1) + para.Kd1 * (Vd(1) - q1dot);   % Same vd for
both
    tau2 = Tff(2)+para.Kp2 * (q2d - q2) + para.Kd2 * (Vd(2) - q2dot);   % Same vd for
both

    % Limit the torques to the range [-50, 50]
    tau1 = max(min(tau1, 50), -50);
    tau2 = max(min(tau2, 50), -50);

    % Calculate accelerations
    a = [tau1; tau2] - C * [q1dot; q2dot] - N;

    % Solve for joint accelerations using inverse of D matrix
    qdoubledot = D \ a;

    % Return derivative of state vector
    xdot = [q1dot; qdoubledot(1); q2dot; qdoubledot(2)];
end

% Define the parameters in the 'para' structure
para.m1 = 7.848;
para.m2 = 4.49;
para.l1 = 0.3;
para.lc1 = 0.1554;
para.lc2 = 0.0341;
para.I1 = 0.176;
para.I2 = 0.0411;
para.g = 9.81;
para.Kp1 = 50;
para.Kp2 = 50;
para.Kd1 = 10;
para.Kd2 = 10;

% Define the simulation time span and initial conditions
tspan = [0 4];  % Define the time span for simulation
x0 = [0; 0; 0; 0];  % Initial conditions for [q1; q1dot; q2; q2dot]

% Run the simulation using ODE45
```

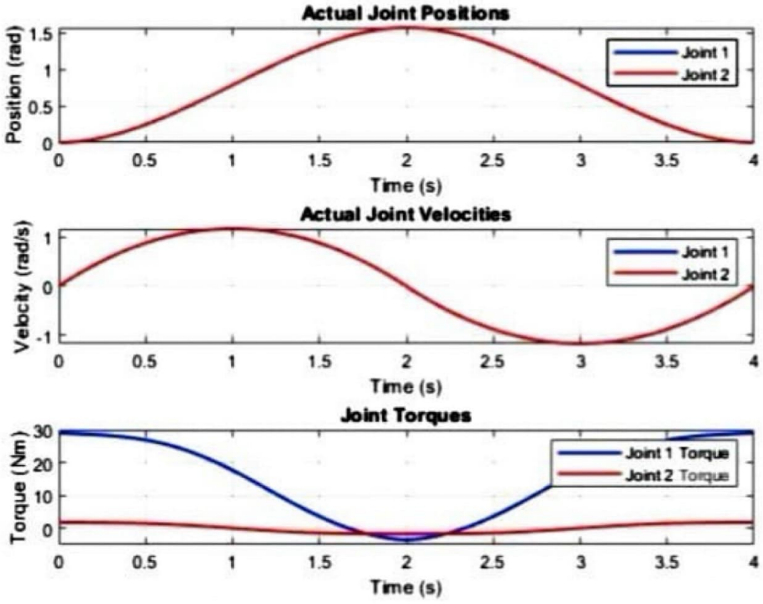
```
[t, x] = ode45(@(t, x) robot_dynamics(t, x, para), tspan, x0);

% Initialize arrays for accelerations and tracking errors
joint_accelerations = zeros(length(t), 2); % For both joints

% Preallocate tracking error arrays
tau1 = zeros(length(t), 1);
tau2 = zeros(length(t), 1);
tracking_error_q1 = zeros(length(t), 1);
tracking_error_q2 = zeros(length(t), 1);
tracking_error_v1 = zeros(length(t), 1); % Preallocated
tracking_error_v2 = zeros(length(t), 1); % Preallocated
tracking_error_a1 = zeros(length(t), 1); % Preallocated
tracking_error_a2 = zeros(length(t), 1); % Preallocated

for i = 1:length(t)
    [qd, vd, ad] = generate_trajectory(t(i));  % Get desired trajectory
    q1d = qd;  % Desired position for joint 1
    q2d = qd;  % Desired position for joint 2

    % Since vd and ad are scalars, you don't need to index them like arrays
    Vd = [vd; vd];  % Same velocity for both joints
    Ad = [ad; ad];  % Same acceleration for both joints

    % Extract current states
    q1 = x(i, 1);
    q1dot = x(i, 2);
    q2 = x(i, 3);
    q2dot = x(i, 4);

    % Calculate the matrices D, C, N for the current state
    D_current = zeros(2, 2);
    D_current(1,1) = para.m1 * para.lc1^2 + para.m2 * (para.l1^2 + para.lc2^2 + 2 * ↙
para.l1 * para.lc2 * cos(q2)) + para.I1 + para.I2;
    D_current(1,2) = para.m2 * (para.lc2^2 + para.l1 * para.lc2 * cos(q2)) + para.I2;
    D_current(2,1) = D_current(1,2);
    D_current(2,2) = para.m2 * para.lc2^2 + para.I2;

    C_current = zeros(2, 2);
    C_current(1, 1) = -para.m2 * para.l1 * para.lc2 * sin(q2) * q2dot;
    C_current(1, 2) = -para.m2 * para.l1 * para.lc2 * sin(q2) * (q1dot + q2dot);
    C_current(2, 1) = para.m2 * para.l1 * para.lc2 * sin(q2) * q1dot;
    C_current(2, 2) = 0;

    N_current = zeros(2, 1);
    N_current(1) = para.m1 * para.g * para.lc1 * cos(q1) + para.m2 * para.g * (para. ↙
l1 * cos(q1) + para.lc2 * cos(q1 + q2));
    N_current(2) = para.m2 * para.g * para.lc2 * cos(q1 + q2);
```

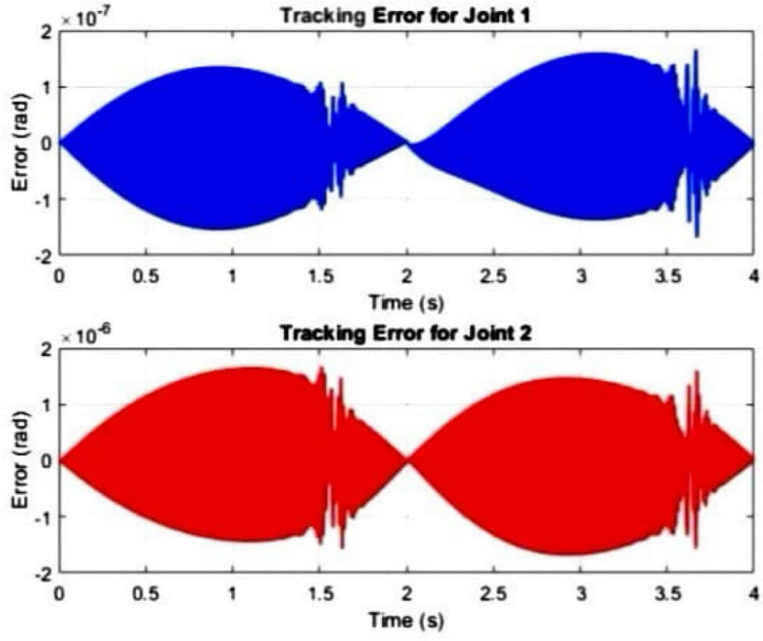
```
    Tff = D_current*Ad + C_current*Vd + N_current;

    % Calculate PD control torques
    tau1(i) = Tff(1)+para.Kp1 * (q1d - q1) + para.Kd1 * (Vd(1) - q1dot);
    tau2(i) = Tff(2)+para.Kp2 * (q2d - q2) + para.Kd2 * (Vd(2) - q2dot);

    % Limit the torques to the range [-50, 50]
    tau1(i) = max(min(tau1(i), 50), -50);
    tau2(i) = max(min(tau2(i), 50), -50);

    % Compute joint accelerations
    a_current = [tau1(i); tau2(i)] - C_current * [q1dot; q2dot] - N_current;
    joint_accelerations(i, :) = D_current \ a_current;   % Joint accelerations


    % Track errors
    tracking_error_q1(i) = q1d - q1;
    tracking_error_q2(i) = q2d - q2;
    tracking_error_v1(i) = Vd(1) - q1dot;
    tracking_error_v2(i) = Vd(2) - q2dot;
    tracking_error_a1(i) = Ad(1) - joint_accelerations(i, 1);
    tracking_error_a2(i) = Ad(2) - joint_accelerations(i, 2);

end

% Plot joint positions
figure;

subplot(3, 1, 1);
plot(t, x(:, 1), 'b', 'LineWidth', 1.5);
hold on;
plot(t, x(:, 3), 'r', 'LineWidth', 1.5);
plot(t, qd, 'k--', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Position (rad)');
title('Actual Joint Positions');
legend('Joint 1', 'Joint 2');
grid on;

% Plot joint velocities
subplot(3, 1, 2);
plot(t, x(:, 2), 'b', 'LineWidth', 1.5);
hold on;
plot(t, x(:, 4), 'r', 'LineWidth', 1.5);
plot(t, vd, 'k--', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Velocity (rad/s)');
```

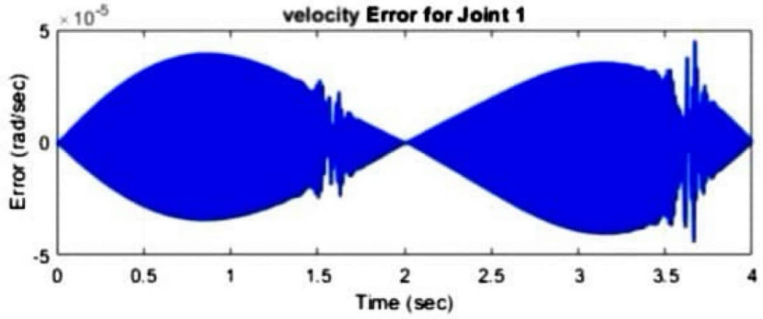
```
title('Actual Joint Velocities');
legend('Joint 1', 'Joint 2');
grid on;

% Plot joint torques
subplot(3, 1, 3);
plot(t, tau1, 'b', 'LineWidth', 1.5);
hold on;
plot(t, tau2, 'r', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Torque (Nm)');
title('Joint Torques');
legend('Joint 1 Torque', 'Joint 2 Torque');
ylim([-5 30]);  % Set y-axis limits from -100 to +100
grid on;

% Plot tracking errors
figure;

subplot(2, 1, 1);
plot(t, tracking_error_q1, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error (rad)');
title('Tracking Error for Joint 1');
grid on;

subplot(2, 1, 2);
plot(t, tracking_error_q2, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error (rad)');
title('Tracking Error for Joint 2');
grid on;

% plot velocity error
figure;

subplot(2, 1, 1);
plot(t, tracking_error_v1, 'b', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec)');
title('velocity Error for Joint 1');
grid on;

subplot(2, 1, 2);
plot(t, tracking_error_v2, 'r', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec)');
title('velocity Error for Joint 2');
```

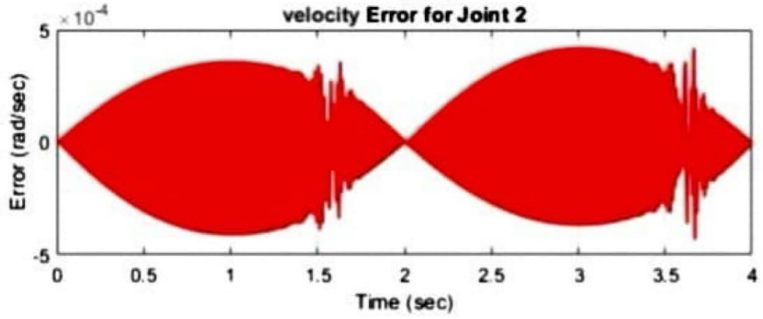
```
grid on;

%Plot acceleration error
figure;

subplot(2, 1, 1);
plot(t, tracking_error_a1, 'b', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec^2)');
title('acceleration Error for Joint 1');
grid on;

subplot(2, 1, 2);
plot(t, tracking_error_a2, 'r', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec^2)');
title('acceleration Error for Joint 2');
grid on;

% Plot joint accelerations
figure;
subplot(2, 1, 1);
plot(t, joint_accelerations(:, 1), 'b', 'LineWidth', 1.5); % Joint 1 acceleration
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
title('Joint 1 Acceleration');
grid on;

subplot(2, 1, 2);
plot(t, joint_accelerations(:, 2), 'r', 'LineWidth', 1.5); % Joint 2 acceleration
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
title('Joint 2 Acceleration');
grid on;

% Define the time span for the trajectory
tspan = linspace(0, 4, 100);  % 100 points from 0 to 4 seconds

% Preallocate arrays for joint trajectories
qd1 = zeros(size(tspan));
vd1 = zeros(size(tspan));
ad1 = zeros(size(tspan));
qd2 = zeros(size(tspan));
vd2 = zeros(size(tspan));
ad2 = zeros(size(tspan));

% Generate the trajectory for each time point
for i = 1:length(tspan)
    [qd1(i), vd1(i), ad1(i), qd2(i), vd2(i), ad2(i)] = generate_trajectory(tspan(i));
```

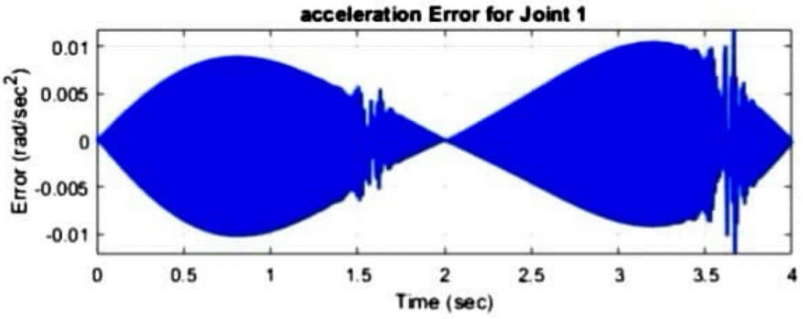
```
end

% Plot desired positions for both joints
figure;
subplot(3, 1, 1);  % 3 rows, 1 column, 1st subplot
plot(tspan, qd1, 'LineWidth', 2);
hold on;
plot(tspan, qd2, 'LineWidth', 2);
title('Desired Position (q) for Both Joints');
xlabel('Time (s)');
ylabel('Position (rad)');
legend('Joint 1', 'Joint 2');
grid on;

% Plot desired velocities for both joints
subplot(3, 1, 2);  % 3 rows, 1 column, 2nd subplot
plot(tspan, vd1, 'LineWidth', 2);
hold on;
plot(tspan, vd2, 'LineWidth', 2);
title('Desired Velocity (v) for Both Joints');
xlabel('Time (s)');
ylabel('Velocity (rad/s)');
legend('Joint 1', 'Joint 2');
grid on;

% Plot desired accelerations for both joints
subplot(3, 1, 3);  % 3 rows, 1 column, 3rd subplot
plot(tspan, ad1, 'LineWidth', 2);
hold on;
plot(tspan, ad2, 'LineWidth', 2);
title('Desired Acceleration (a) for Both Joints');
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
legend('Joint 1', 'Joint 2');
grid on;
```

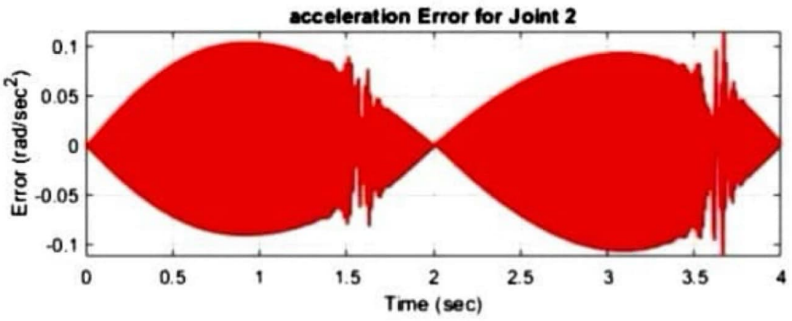

Actual Joint Positions
Position (rad)
Time (s)
Joint 1
Joint 2
Actual Joint Velocities
Velocity (rad/s)
Time (s)
Joint 1
Joint 2
Joint Torques
Torque (Nm)
Time (s)
Joint 1 Torque
Joint 2 Torque

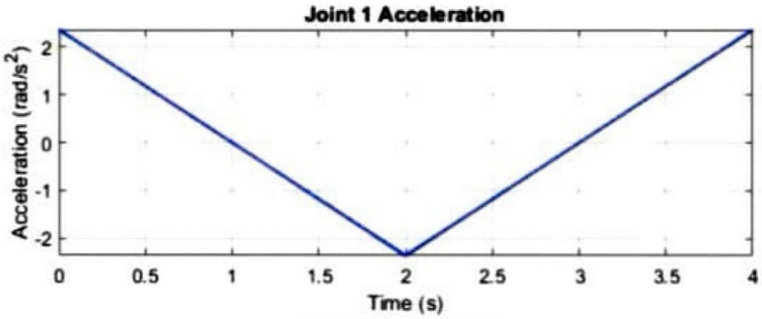

Tracking Error for Joint 1
× 10^-7
Error (rad)
Time (s)
Tracking Error for Joint 2
× 10^-6
Error (rad)
Time (s)

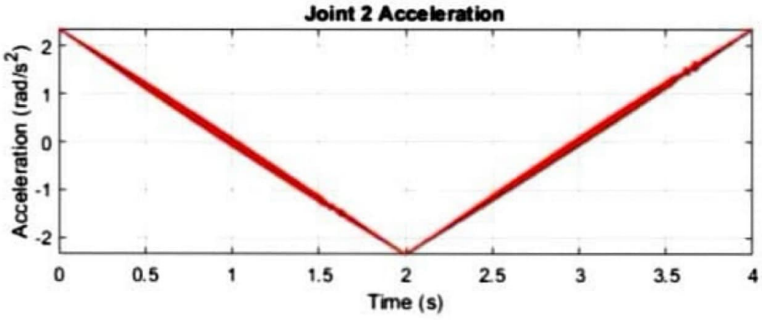

velocity Error for Joint 1
×10^-5
Error (rad/sec)
Time (sec)

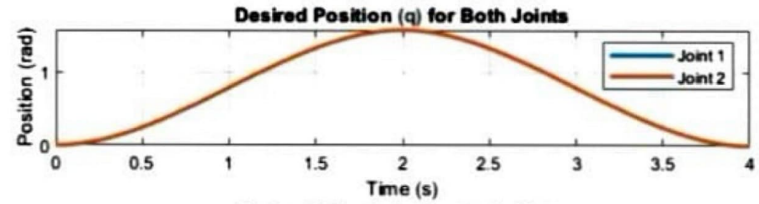

velocity Error for Joint 2
×10^-4
Error (rad/sec)
Time (sec)

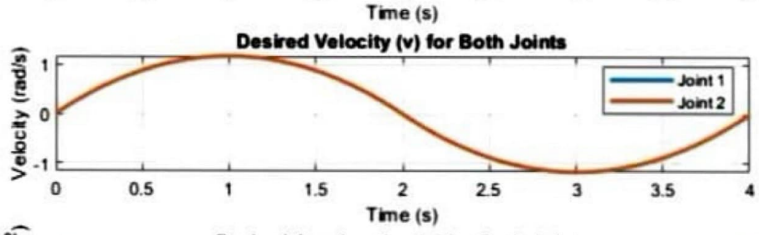

acceleration Error for Joint 1
Error (rad/sec^2)
Time (sec)

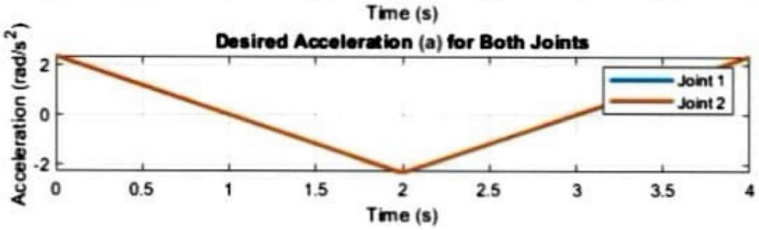

acceleration Error for Joint 2
Error (rad/sec^2)
Time (sec)









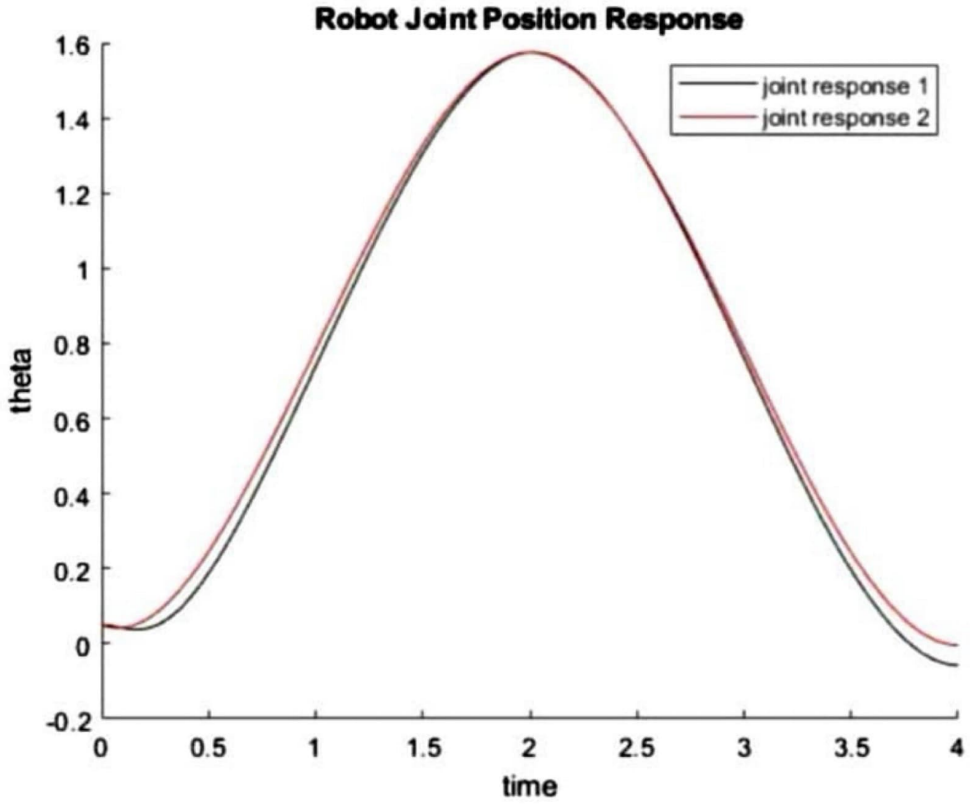


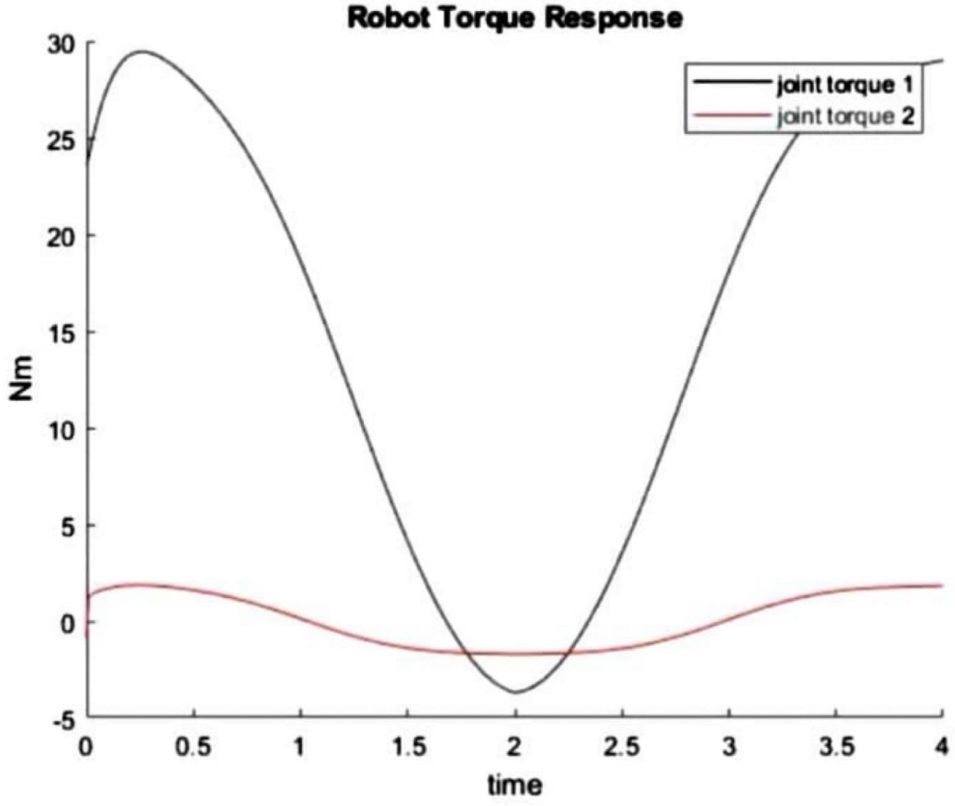
Published with MATLAB® R2024a

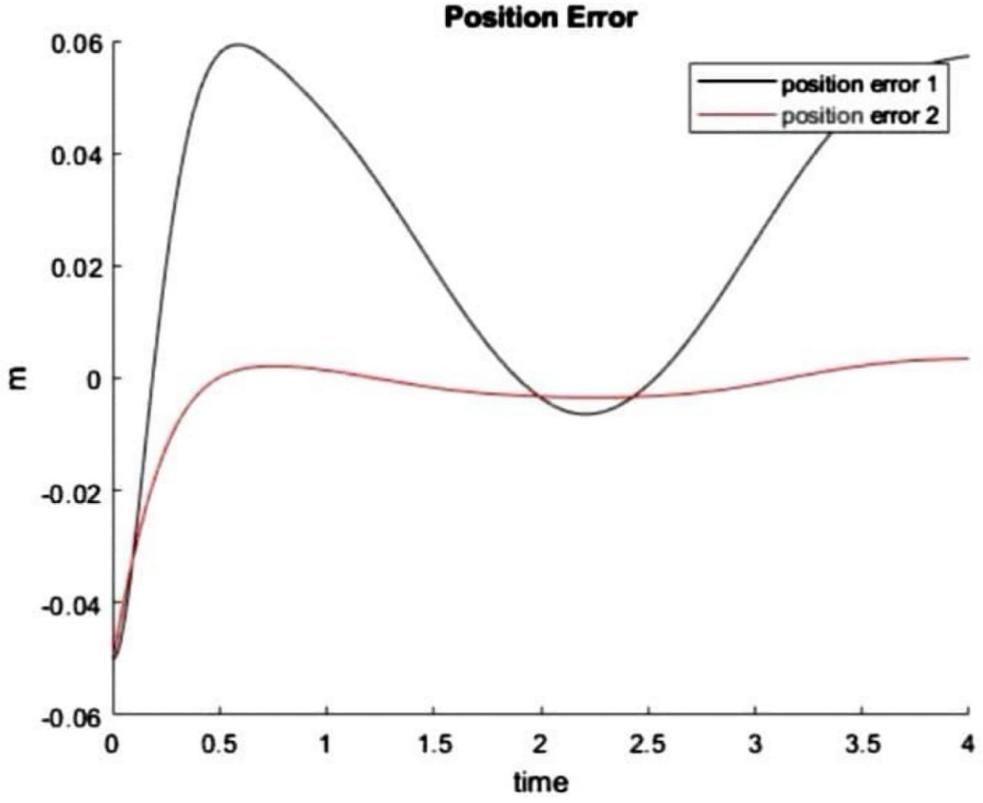
```
%HW 3 P5 b
clear
close all
clc
function [xdot,tau,error,cmd_val] = HW3_Q6(t,x)
q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

m1 = 7.848;m2 = 4.49; l1 = 0.3;lc1 = 0.1554;
lc2 = 0.0341;I1 = 0.176;I2 = 0.0411;
g = 9.81;

M(1,1) = m1*lc1^2 + m2*(l1^2 + lc2^2 + 2*l1*lc2*cos(q2)) + I1 + I2;
M(1,2) = m2*(lc2^2 + l1*lc2*cos(q2)) + I2;
M(2,1) = M(1,2);
M(2,2) = m2*lc2^2 + I2;

% Coriolis matrix
C(1, 1) = -m2*l1*lc2*sin(q2)*q2dot;
C(1, 2) = -m2*l1*lc2*sin(q2)*(q1dot + q2dot);
C(2, 1) = m2*l1*lc2*sin(q2)*q1dot;
C(2, 2) = 0;

%External forces (gravity etc)
N(1,1) = m1*g*lc1*cos(q1) + m2*g*(l1*cos(q1)+lc2*cos(q1 +q2));
N(2,1) = m2*g*lc2*cos(q1 + q2);

%defining the control law
[qd, vd, ad] = cubicpoly([0, 2, 4], [0, pi/2, 0], [0, 0, 0], t);
cmd_val = [qd, vd, ad];
Kp1 = 50 ; Kd1 =10;
Kp2 = 50; Kd2 = 10;


tau(1,1) = Kp1*(qd - q1) + Kd1*(vd - q1dot);
tau(2,1) = Kp2*(qd - q2) + Kd2*(vd - q2dot);
error = [qd - q1 qd - q2 vd - q1dot vd - q2dot]';

%declaring the estimated variables for the feed forward equation
Mq_hat = 0.9*M; Cq_hat = 0.9*C; Nq_hat = 0.9*N;

%feed forward torque equation
tau_FF = Mq_hat*[ad;ad] +Cq_hat*[vd;vd]+ Nq_hat;

%feed forward + feed back equation
tau = tau+ tau_FF;


%employing the limitations
if (tau(1,1)<= -50)
```

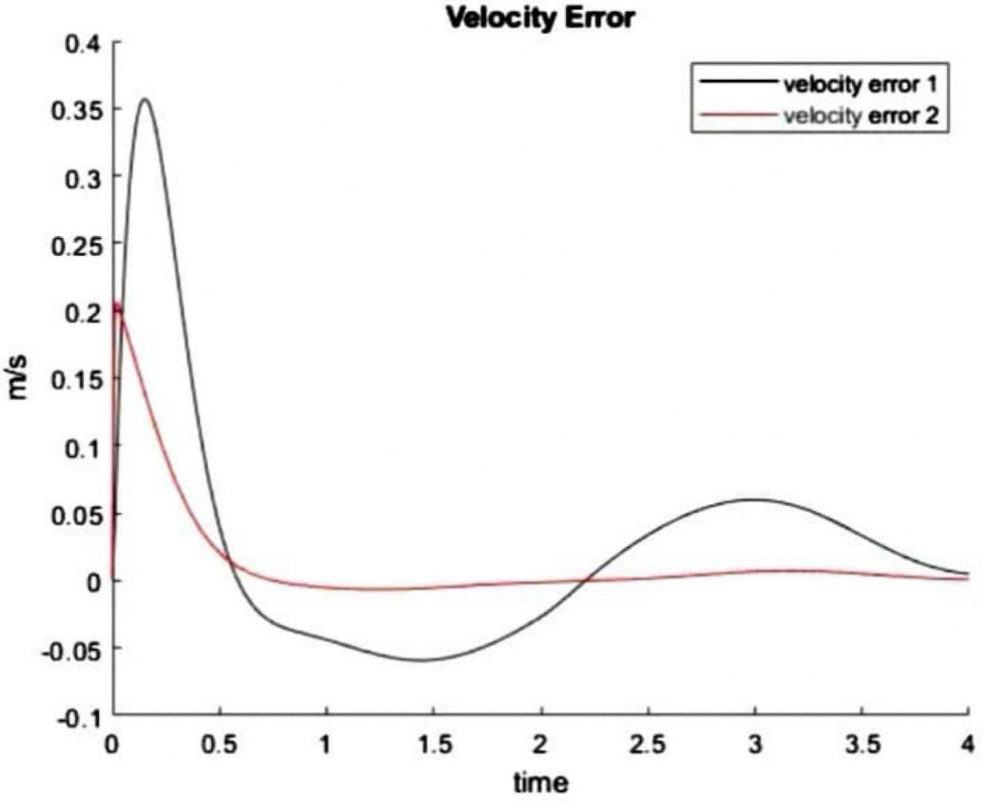
```
tau(1,1) = -50;
end
if (tau(1,1) >= 50)
tau(1,1) = 50;
end
if (tau(2,1)<= -50)
tau(2,1) = -50;
end
if (tau(2,1) >= 50)
tau(2,1) = 50;
end

xdot = inv(M)*(tau - N -C*[q1dot;q2dot]);
xdot(4,1) = xdot(2,1);
xdot(2,1) = xdot(1,1);
xdot(1,1) = x(2);
xdot(3,1) = x(4);

end %function end

q0 = [0.05 0 0.05 0]';
[t,x]=ode45(@(t,x) HW3_Q6(t,x),0:0.01:4,q0);
tau = zeros(length(t), 2);
error = zeros(length(t), 4);
cmd_val = zeros(length(t), 3);
for i = 1:length(t)
[~, tau(i, :),error(i,:),cmd_val(i,:)] = HW3_Q6(t(i), x(i, :));
end

figure(1)
hold on
plot(t,x(:,1),'black')
plot(t,x(:,3),'red')
xlabel('time')
ylabel('theta')
title('Robot Joint Position Response')
legend('joint response 1','joint response 2')
hold off

figure(2)
hold on
plot(t,tau(:,1),'black')
plot(t,tau(:,2),'red')
xlabel('time')
ylabel('Nm')
title('Robot Torque Response')
legend('joint torque 1','joint torque 2')
hold off
```

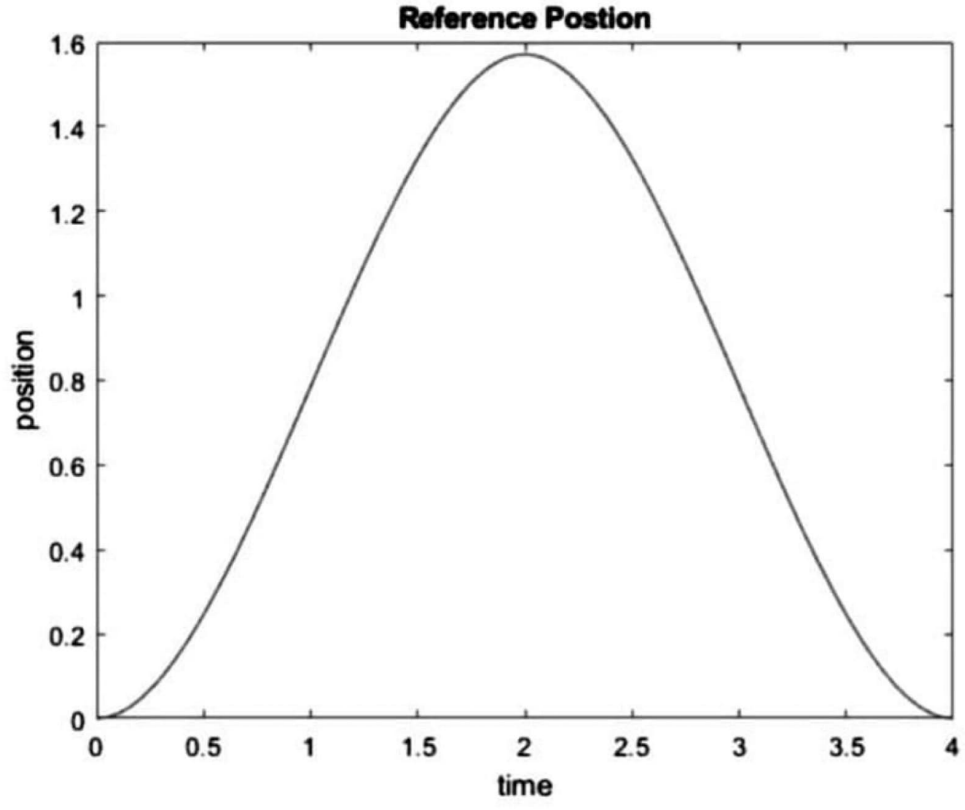
```
figure(3)
hold on
plot(t,error(:,1),'black')
plot(t,error(:,2),'red')
title('Position Error')
xlabel('time')
ylabel('m')
legend('position error 1','position error 2')
hold off

figure(4)
hold on
plot(t,error(:,3),'black')
plot(t,error(:,4),'red')
title('Velocity Error')
xlabel('time')
ylabel('m/s')
legend('velocity error 1','velocity error 2')
hold off

figure(5)
plot(t,cmd_val(:,1),'black')
xlabel('time')
ylabel('position')
title('Reference Postion')
figure(6)
plot(t,cmd_val(:,2),'black')
xlabel('time')
ylabel('velocity')
title('Reference Velocity')
figure(7)
plot(t,cmd_val(:,3),'black')
xlabel('time')
ylabel('acceleration')
title('Reference Acceleration')
```

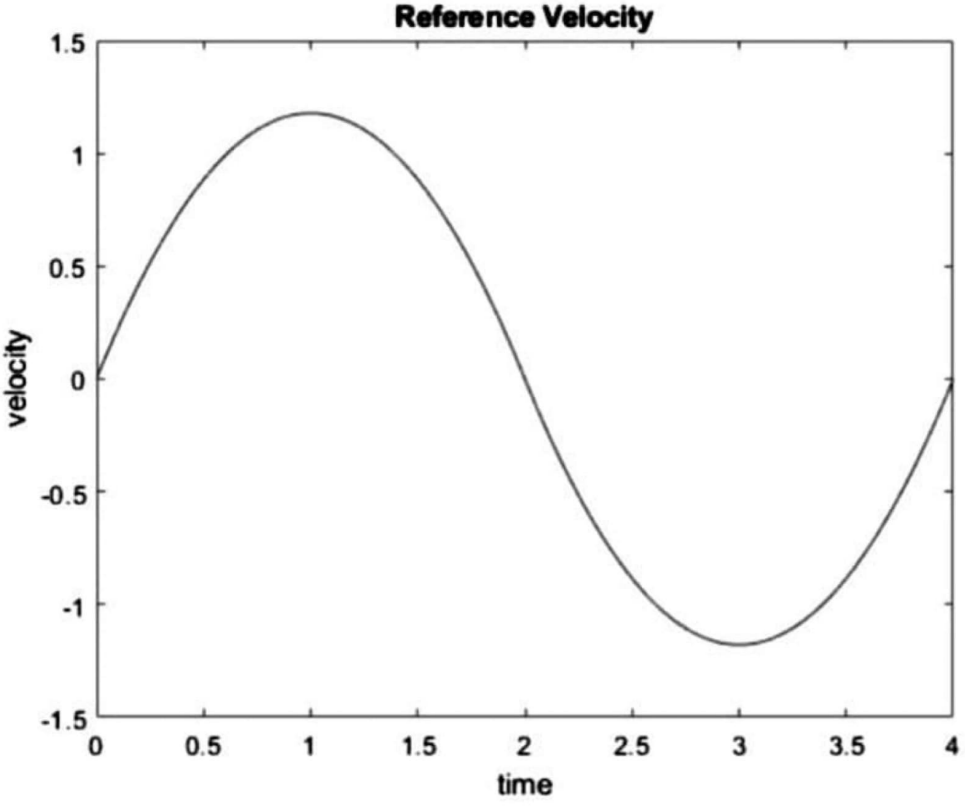

Robot Joint Position Response
joint response 1
joint response 2
theta
time
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.5
1
1.5
2
2.5
3
3.5
4

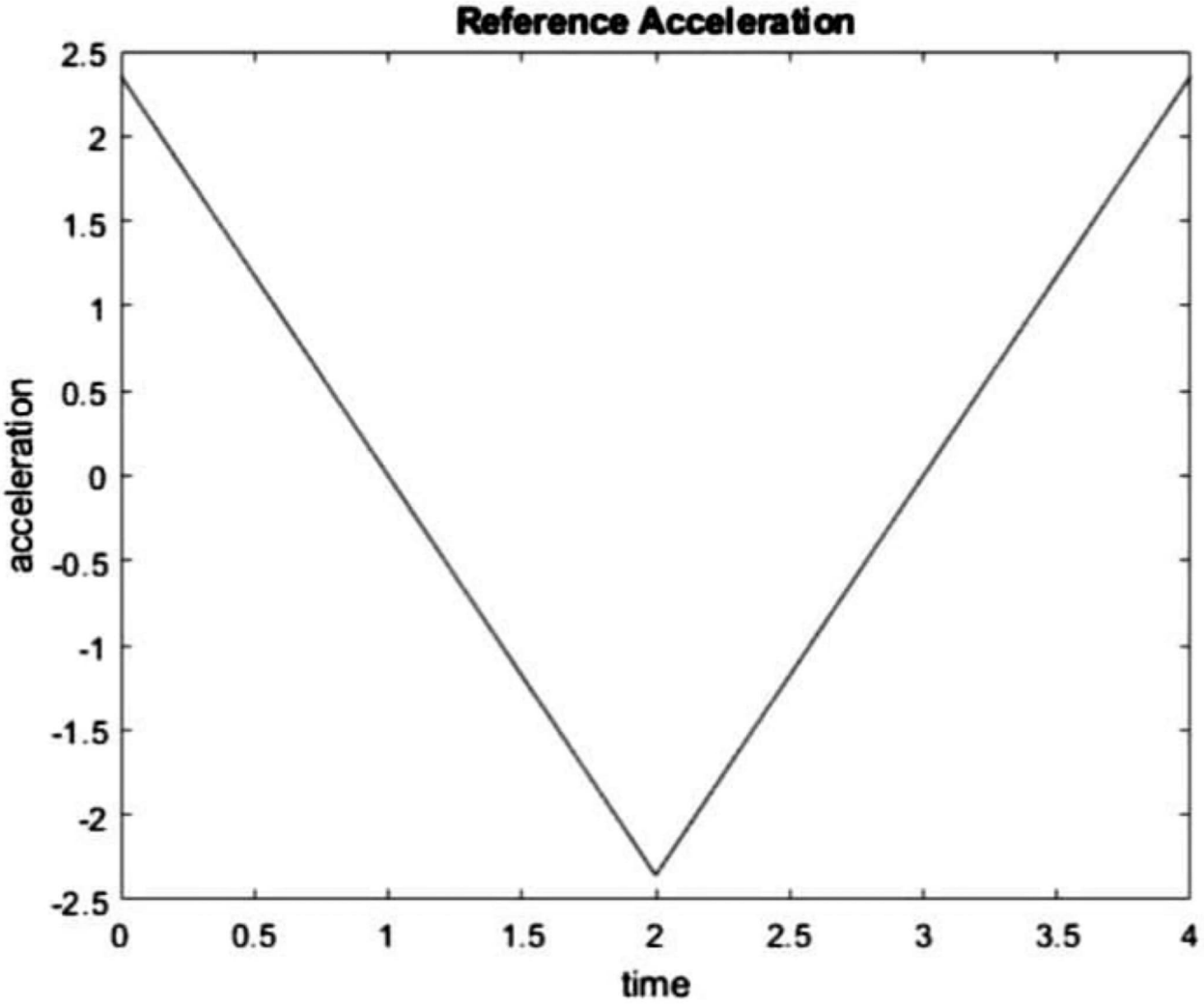

Robot Torque Response
joint torque 1
joint torque 2
Nm
time
30
25
20
15
10
5
0
-5
0
0.5
1
1.5
2
2.5
3
3.5
4


Position Error
position error 1
position error 2
m
time
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
0
0.5
1
1.5
2
2.5
3
3.5
4


Velocity Error
velocity error 1
velocity error 2
m/s
time
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
0
0.5
1
1.5
2
2.5
3
3.5
4


Reference Postion
position
time
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
0
0.5
1
1.5
2
2.5
3
3.5
4


Reference Velocity
velocity
time
-1.5
-1
-0.5
0
0.5
1
1.5
0
0.5
1
1.5
2
2.5
3
3.5
4



Published with MATLAB® R2024a

When comparing the results between the feed-forward control with PD (Problem 5(a)) and the regular PD controller (Problem 4), the key differences are:

1. Dynamic Compensation: In the feed-forward controller, errors are significantly smaller because it compensates for the robot's dynamics (inertia, Coriolis, and gravity forces). The regular PD controller lacks this, resulting in noticeably larger errors, especially in complex motions.

2. Error Magnitude and Response: The feed-forward controller shows minimal tracking, velocity, and acceleration errors throughout the trajectory, as it anticipates required torques. The regular PD controller has visibly larger position and torque errors since it only reacts after errors occur without dynamic pre-compensation.

3. Handling Complex Trajectories: The regular PD controller struggles with larger trajectory variations, leading to significant overshoots and delays, while the feed-forward controller maintains more accurate and stable tracking.

Conclusion:

The feed-forward control with PD results in far smaller errors and more stable performance, especially for complex trajectories, while the regular PD controller exhibits larger errors due to its reactive nature and lack of dynamic compensation.

```
%P6

function [qd1, vd1, ad1, qd2, vd2, ad2] = generate_trajectory(t)
    % Time breakpoints
    t_waypoints = [0 2 4];

    % Waypoints for position q(t) for both joints
    waypoints = [0, pi/2, 0];    % Desired positions for both joints

    % Desired velocities for both joints
    velocities = [0, 0, 0];      % Desired velocities for both joints

    % Generate cubic polynomial trajectory
    [qd, vd, ad] = cubicpolytraj(waypoints, t_waypoints, t, ...
                                 'VelocityBoundaryCondition', velocities);

    % Assign the same trajectory to both joints
    qd1 = qd; vd1 = vd; ad1 = ad;   % For joint 1
    qd2 = qd; vd2 = vd; ad2 = ad;   % For joint 2
end

function xdot = robot_dynamics(t, x, para)
    % Extract joint positions and velocities
    q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

    % Desired trajectory at time t
    [qd, vd, ad] = generate_trajectory(t);   % Get desired position, velocity, ↙
acceleration
    q1d = qd;   % Same trajectory for both q1 and q2
    q2d = qd;

    % Ensure that vd and ad are vectors with two identical elements
    Vd = [vd; vd];   % Both joints have the same desired velocity
    Ad = [ad; ad];   % Both joints have the same desired acceleration

    Aq = zeros(2,1);
    Aq(1)= Ad(1) + para.Kp1*(q1d - q1) + para.Kd1*(Vd(1)-q1dot);
    Aq(2)= Ad(2) + para.Kp2*(q2d - q2) + para.Kd2*(Vd(2)-q2dot);


    % Mass/inertia matrix D(q)
    D = zeros(2, 2);
    D(1,1) = para.m1 * para.lc1^2 + para.m2 * (para.l1^2 + para.lc2^2 + 2 * para.l1 * ↙
para.lc2 * cos(q2)) + para.I1 + para.I2;
    D(1,2) = para.m2 * (para.lc2^2 + para.l1 * para.lc2 * cos(q2)) + para.I2;
    D(2,1) = D(1,2);
    D(2,2) = para.m2 * para.lc2^2 + para.I2;

    % Coriolis matrix C(q,qdot)
```

```
    C = zeros(2, 2);
    C(1, 1) = -para.m2 * para.l1 * para.lc2 * sin(q2) * q2dot;
    C(1, 2) = -para.m2 * para.l1 * para.lc2 * sin(q2) * (q1dot + q2dot);
    C(2, 1) = para.m2 * para.l1 * para.lc2 * sin(q2) * q1dot;
    C(2, 2) = 0;

    % Gravity terms N(q)
    N = zeros(2, 1);
    N(1) = para.m1 * para.g * para.lc1 * cos(q1) + para.m2 * para.g * (para.l1 * cos
(q1) + para.lc2 * cos(q1 + q2));
    N(2) = para.m2 * para.g * para.lc2 * cos(q1 + q2);

    % Feed back new controller
    Tau = D*Aq + C*[q1dot; q2dot] + N;

    % Limit the torques to the range [-50, 50]
    Tau(1) = max(min(Tau(1), 50), -50);
    Tau(2) = max(min(Tau(2), 50), -50);

    % Calculate accelerations
    a = [Tau(1); Tau(2)] - C * [q1dot; q2dot] - N;

    % Solve for joint accelerations using inverse of D matrix
    qdoubledot = D \ a;

    % Return derivative of state vector
    xdot = [q1dot; qdoubledot(1); q2dot; qdoubledot(2)];
end

% Define the parameters in the 'para' structure
para.m1 = 7.848;
para.m2 = 4.49;
para.l1 = 0.3;
para.lc1 = 0.1554;
para.lc2 = 0.0341;
para.I1 = 0.176;
para.I2 = 0.0411;
para.g = 9.81;
para.Kp1 = 50;
para.Kp2 = 50;
para.Kd1 = 10;
para.Kd2 = 10;

% Define the simulation time span and initial conditions
tspan = [0 4];  % Define the time span for simulation
x0 = [0.05; 0; 0.05; 0];  % Initial conditions for [q1; q1dot; q2; q2dot]

% Run the simulation using ODE45
[t, x] = ode45(@(t, x) robot_dynamics(t, x, para), tspan, x0);
```

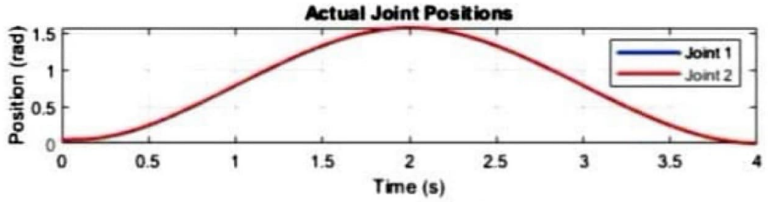
```
% Initialize arrays for accelerations
joint_accelerations = zeros(length(t), 2); % For both joints

% Calculate torques and tracking errors over time
tau1 = zeros(length(t), 1);
tau2 = zeros(length(t), 1);
tracking_error_q1 = zeros(length(t), 1);
tracking_error_q2 = zeros(length(t), 1);
tracking_error_v1 = zeros(length(t), 1); % Preallocated
tracking_error_v2 = zeros(length(t), 1); % Preallocated
tracking_error_a1 = zeros(length(t), 1); % Preallocated
tracking_error_a2 = zeros(length(t), 1); % Preallocated

for i = 1:length(t)
    % Desired trajectory at time t(i)
    [qd, vd, ad] = generate_trajectory(t(i));  % Get desired trajectory
    q1d = qd;  % Desired position for joint 1
    q2d = qd;  % Desired position for joint 2

    % Ensure that vd and ad are scalars
    Vd = [vd; vd];  % Same velocity for both joints
    Ad = [ad; ad];  % Same acceleration for both joints

    % Extract current states
    q1 = x(i, 1);
    q1dot = x(i, 2);
    q2 = x(i, 3);
    q2dot = x(i, 4);

    % Calculate Aq using the same method as in robot_dynamics
    Aq = zeros(2,1);
    Aq(1) = Ad(1) + para.Kp1 * (q1d - q1) + para.Kd1 * (Vd(1) - q1dot);
    Aq(2) = Ad(2) + para.Kp2 * (q2d - q2) + para.Kd2 * (Vd(2) - q2dot);

    % Calculate the mass matrix D for the current state
    D_current = zeros(2, 2);
    D_current(1,1) = para.m1 * para.lc1^2 + para.m2 * (para.l1^2 + para.lc2^2 + 2 * ↙
para.l1 * para.lc2 * cos(q2)) + para.I1 + para.I2;
    D_current(1,2) = para.m2 * (para.lc2^2 + para.l1 * para.lc2 * cos(q2)) + para.I2;
    D_current(2,1) = D_current(1,2);
    D_current(2,2) = para.m2 * para.lc2^2 + para.I2;

    % Coriolis matrix C for the current state
    C_current = zeros(2, 2);
    C_current(1, 1) = -para.m2 * para.l1 * para.lc2 * sin(q2) * q2dot;
    C_current(1, 2) = -para.m2 * para.l1 * para.lc2 * sin(q2) * (q1dot + q2dot);
    C_current(2, 1) = para.m2 * para.l1 * para.lc2 * sin(q2) * q1dot;
    C_current(2, 2) = 0;
```

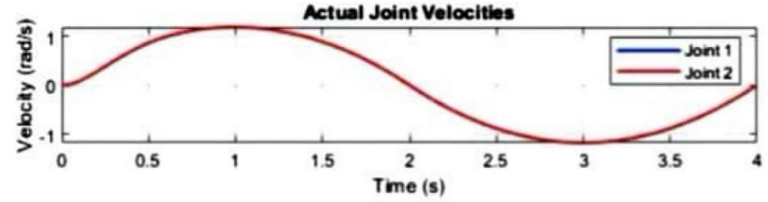
```
    % Gravity terms N for the current state
    N_current = zeros(2, 1);
    N_current(1) = para.m1 * para.g * para.lc1 * cos(q1) + para.m2 * para.g * (para.
l1 * cos(q1) + para.lc2 * cos(q1 + q2));
    N_current(2) = para.m2 * para.g * para.lc2 * cos(q1 + q2);

    % Calculate feed-forward torque
    Tau = D_current * Aq + C_current * [q1dot; q2dot] + N_current;

    % Compute control torques
    tau1(i) = Tau(1);
    tau2(i) = Tau(2);

    % Limit the torques to the range [-50, 50]
    tau1(i) = max(min(tau1(i), 50), -50);
    tau2(i) = max(min(tau2(i), 50), -50);

    % Compute joint accelerations using the current torque values
    a_current = [tau1(i); tau2(i)] - C_current * [q1dot; q2dot] - N_current;
    joint_accelerations(i, :) = D_current \ a_current;  % Joint accelerations

    % Track position and velocity errors
    tracking_error_q1(i) = q1d - q1;
    tracking_error_q2(i) = q2d - q2;
    tracking_error_v1(i) = Vd(1) - q1dot;
    tracking_error_v2(i) = Vd(2) - q2dot;
    tracking_error_a1(i) = Ad(1) - joint_accelerations(i, 1);
    tracking_error_a2(i) = Ad(2) - joint_accelerations(i, 2);
end

% Plot joint positions
figure;

subplot(3, 1, 1);
plot(t, x(:, 1), 'b', 'LineWidth', 1.5);
hold on;
plot(t, x(:, 3), 'r', 'LineWidth', 1.5);
plot(t, qd, 'k--', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Position (rad)');
title('Actual Joint Positions');
legend('Joint 1', 'Joint 2');
grid on;

% Plot joint velocities
subplot(3, 1, 2);
plot(t, x(:, 2), 'b', 'LineWidth', 1.5);
```

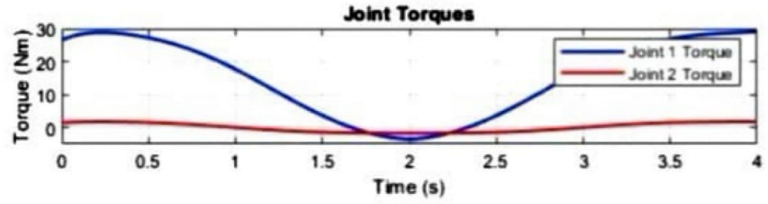
```
hold on;
plot(t, x(:, 4), 'r', 'LineWidth', 1.5);
plot(t, vd, 'k--', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Velocity (rad/s)');
title('Actual Joint Velocities');
legend('Joint 1', 'Joint 2');
grid on;

% Plot joint torques
subplot(3, 1, 3);
plot(t, tau1, 'b', 'LineWidth', 1.5);
hold on;
plot(t, tau2, 'r', 'LineWidth', 1.5);
hold off;
xlabel('Time (s)');
ylabel('Torque (Nm)');
title('Joint Torques');
legend('Joint 1 Torque', 'Joint 2 Torque');
ylim([-5 30]);  % Set y-axis limits from -100 to +100
grid on;

% Plot tracking errors
figure;

subplot(2, 1, 1);
plot(t, tracking_error_q1, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error (rad)');
title('Tracking Error for Joint 1');
grid on;

subplot(2, 1, 2);
plot(t, tracking_error_q2, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error (rad)');
title('Tracking Error for Joint 2');
grid on;

% plot velocity error
figure;

subplot(2, 1, 1);
plot(t, tracking_error_v1, 'b', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec)');
title('velocity Error for Joint 1');
grid on;
```

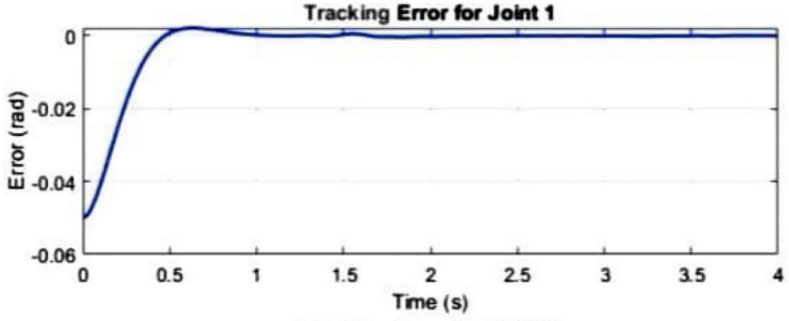
```
subplot(2, 1, 2);
plot(t, tracking_error_v2, 'r', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec)');
title('velocity Error for Joint 2');
grid on;

%Plot acceleration error
figure;

subplot(2, 1, 1);
plot(t, tracking_error_a1, 'b', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec^2)');
title('acceleration Error for Joint 1');
grid on;

subplot(2, 1, 2);
plot(t, tracking_error_a2, 'r', 'LineWidth', 1.5);
xlabel('Time (sec)');
ylabel('Error (rad/sec^2)');
title('acceleration Error for Joint 2');
grid on;

% Plot joint accelerations
figure;
subplot(2, 1, 1);
plot(t, joint_accelerations(:, 1), 'b', 'LineWidth', 1.5); % Joint 1 acceleration
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
title('Joint 1 Acceleration');
grid on;

subplot(2, 1, 2);
plot(t, joint_accelerations(:, 2), 'r', 'LineWidth', 1.5); % Joint 2 acceleration
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
title('Joint 2 Acceleration');
grid on;

% Define the time span for the trajectory
tspan = linspace(0, 4, 100);  % 100 points from 0 to 4 seconds

% Preallocate arrays for joint trajectories
qd1 = zeros(size(tspan));
vd1 = zeros(size(tspan));
ad1 = zeros(size(tspan));
qd2 = zeros(size(tspan));
```

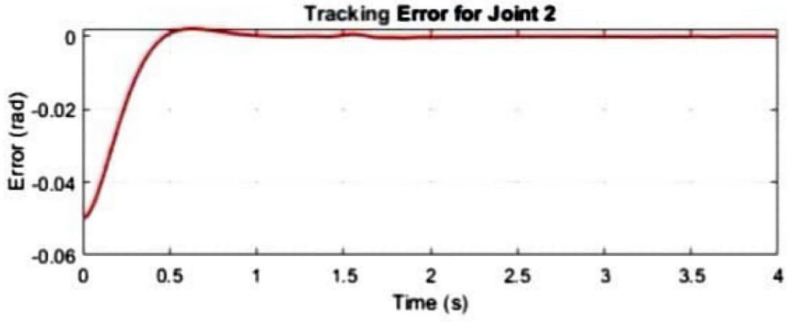
```
vd2 = zeros(size(tspan));
ad2 = zeros(size(tspan));

% Generate the trajectory for each time point
for i = 1:length(tspan)
    [qd1(i), vd1(i), ad1(i), qd2(i), vd2(i), ad2(i)] = generate_trajectory(tspan(i));
end

% Plot desired positions for both joints
figure;
subplot(3, 1, 1);  % 3 rows, 1 column, 1st subplot
plot(tspan, qd1, 'LineWidth', 2);
hold on;
plot(tspan, qd2, 'LineWidth', 2);
title('Desired Position (q) for Both Joints');
xlabel('Time (s)');
ylabel('Position (rad)');
legend('Joint 1', 'Joint 2');
grid on;

% Plot desired velocities for both joints
subplot(3, 1, 2);  % 3 rows, 1 column, 2nd subplot
plot(tspan, vd1, 'LineWidth', 2);
hold on;
plot(tspan, vd2, 'LineWidth', 2);
title('Desired Velocity (v) for Both Joints');
xlabel('Time (s)');
ylabel('Velocity (rad/s)');
legend('Joint 1', 'Joint 2');
grid on;

% Plot desired accelerations for both joints
subplot(3, 1, 3);  % 3 rows, 1 column, 3rd subplot
plot(tspan, ad1, 'LineWidth', 2);
hold on;
plot(tspan, ad2, 'LineWidth', 2);
title('Desired Acceleration (a) for Both Joints');
xlabel('Time (s)');
ylabel('Acceleration (rad/s^2)');
legend('Joint 1', 'Joint 2');
grid on;
```

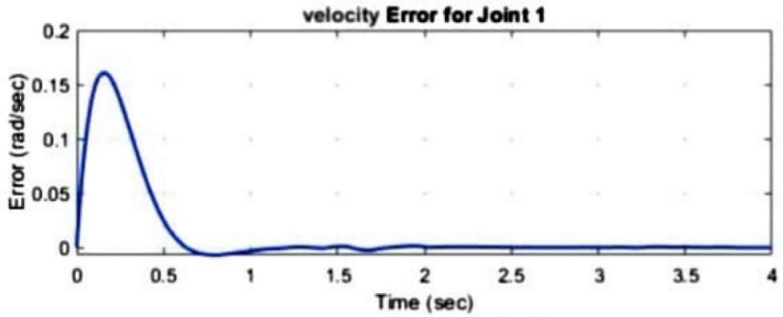

Actual Joint Positions
Position (rad)
Time (s)
Joint 1
Joint 2

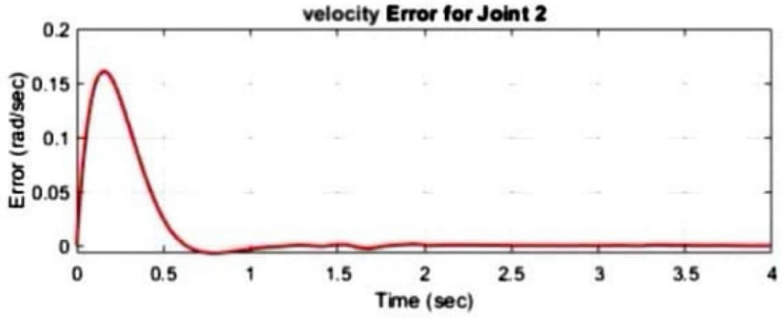

Actual Joint Velocities
Velocity (rad/s)
Time (s)
Joint 1
Joint 2

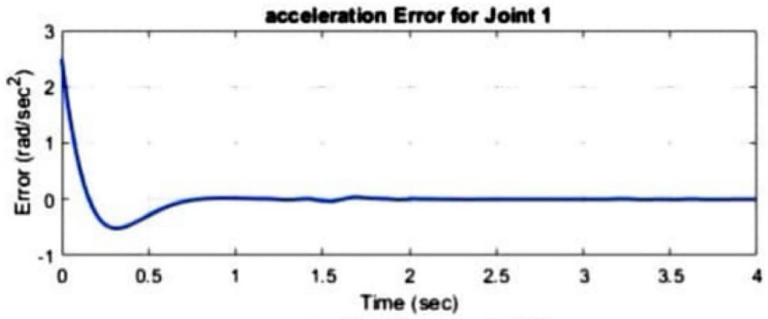

Joint Torques
Torque (Nm)
Time (s)
Joint 1 Torque
Joint 2 Torque

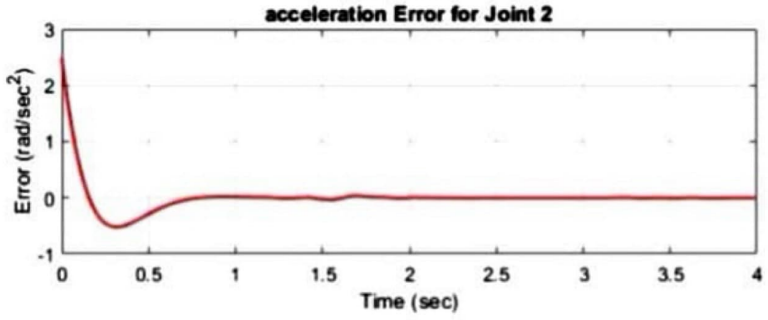

Tracking Error for Joint 1
Error (rad)
Time (s)

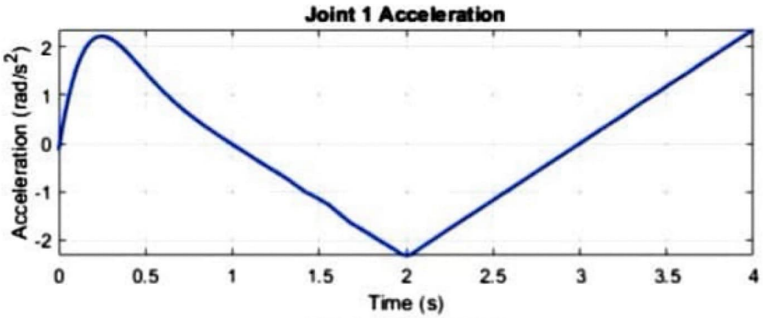

Tracking Error for Joint 2
Error (rad)
Time (s)

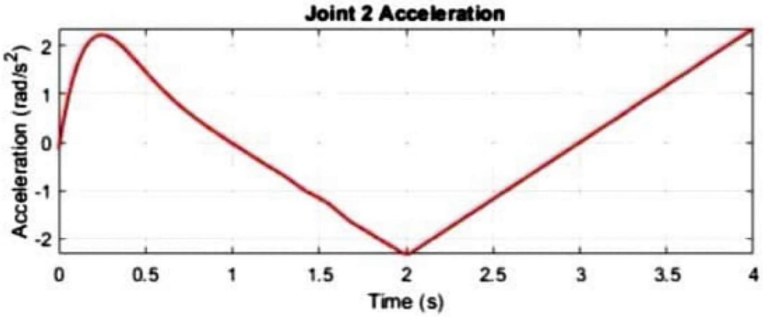

velocity Error for Joint 1
Error (rad/sec)
Time (sec)

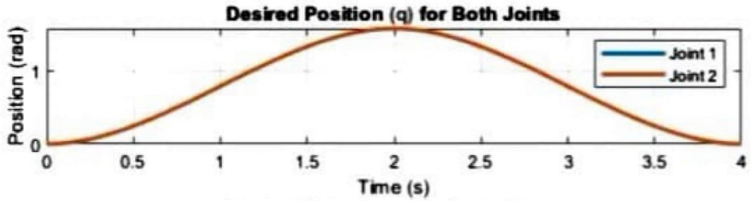

velocity Error for Joint 2
Error (rad/sec)
Time (sec)

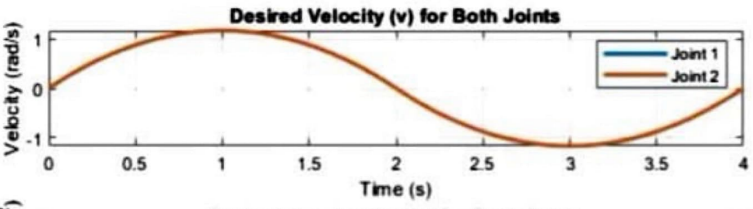

acceleration Error for Joint 1
Error (rad/sec2)
Time (sec)

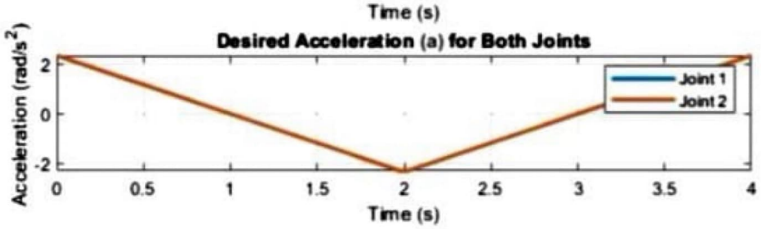

acceleration Error for Joint 2
Error (rad/sec2)
Time (sec)









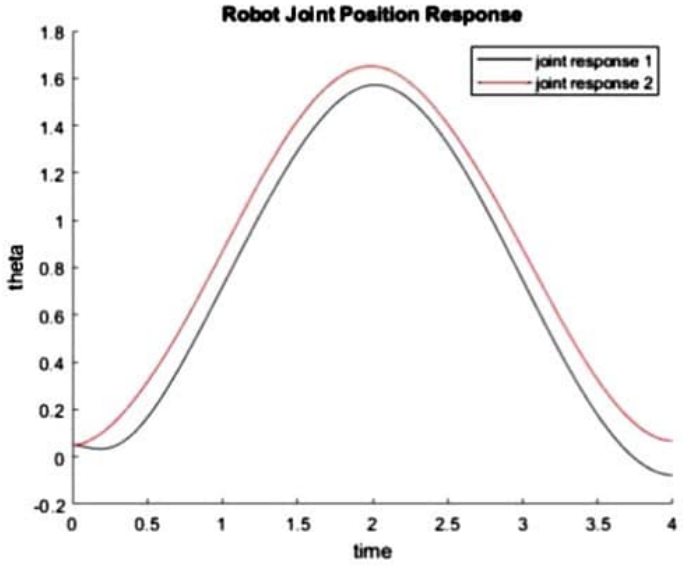


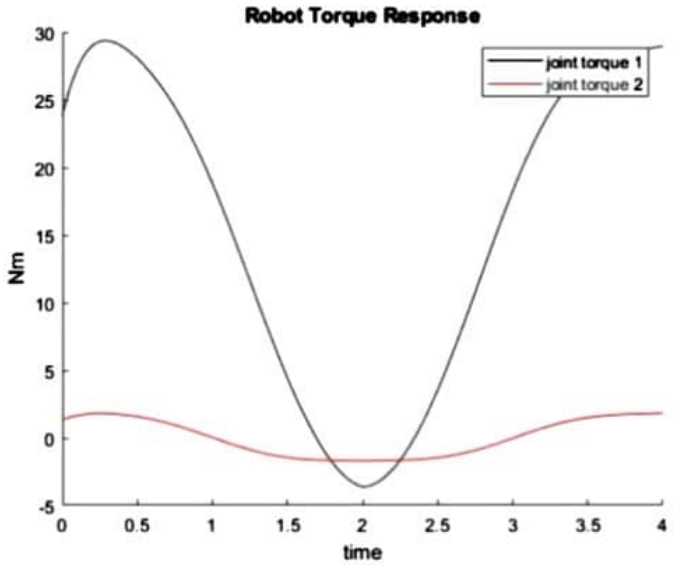
Published with MATLAB® R2024a

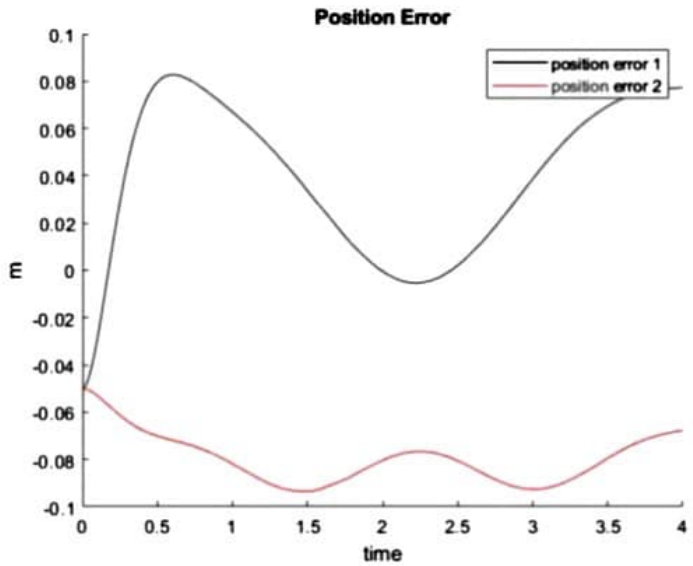
```
%HW 3 P6 b
clear
close all
clc
function [xdot,tau,error,cmd_val] = HW3_Q6(t,x)
q1 = x(1); q1dot = x(2); q2 = x(3); q2dot = x(4);

m1 = 7.848;m2 = 4.49; l1 = 0.3;lc1 = 0.1554;
lc2 = 0.0341;I1 = 0.176;I2 = 0.0411;
g = 9.81;

M(1,1) = m1*lc1^2 + m2*(l1^2 + lc2^2 + 2*l1*lc2*cos(q2)) + I1 + I2;
M(1,2) = m2*(lc2^2 + l1*lc2*cos(q2)) + I2;
M(2,1) = M(1,2);
M(2,2) = m2*lc2^2 + I2;

% Coriolis matrix
C(1, 1) = -m2*l1*lc2*sin(q2)*q2dot;
C(1, 2) = -m2*l1*lc2*sin(q2)*(q1dot + q2dot);
C(2, 1) = m2*l1*lc2*sin(q2)*q1dot;
C(2, 2) = 0;

%External forces (gravity etc)
N(1,1) = m1*g*lc1*cos(q1) + m2*g*(l1*cos(q1)+lc2*cos(q1 +q2));
N(2,1) = m2*g*lc2*cos(q1 + q2);

%defining the control law
[qd, vd, ad] = cubicpoly([0, 2, 4], [0, pi/2, 0], [0, 0, 0], t);
cmd_val = [qd, vd, ad];
Kp1 = 50 ; Kd1 =10;
Kp2 = 50; Kd2 = 10;

error = [qd - q1 qd - q2 vd - q1dot vd - q2dot]';

%declaring the estimated variables for the feed forward equation
Mq_hat = 0.9*M; Cq_hat = 0.9*C; Nq_hat = 0.9*N;

%inverse dynamics (computed torque) control equation
aq1 = ad + Kp1*(qd - q1)+Kd1*(vd - q1dot);
aq2 = ad + Kp2*(qd - q2)+Kd1*(vd - q2dot);
tau = Mq_hat*[aq1;aq2] +Cq_hat*[q1dot;q2dot]+ Nq_hat;


%employing the limitations
if (tau(1,1)<= -50)
tau(1,1) = -50;
end
if (tau(1,1) >= 50)
tau(1,1) = 50;
```

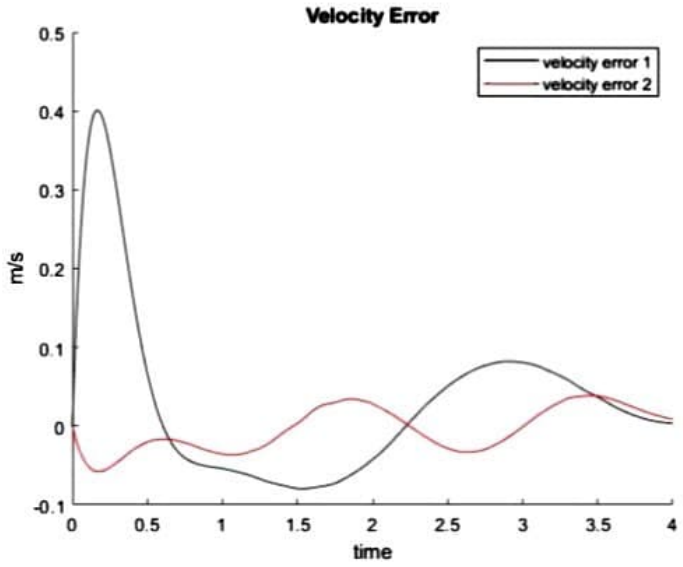
```
end
if (tau(2,1)<= -50)
tau(2,1) = -50;
end
if (tau(2,1) >= 50)
tau(2,1) = 50;
end

xdot = inv(M)*(tau - N -C*[q1dot;q2dot]);
xdot(4,1) = xdot(2,1);
xdot(2,1) = xdot(1,1);
xdot(1,1) = x(2);
xdot(3,1) = x(4);

end %function end

q0 = [0.05 0 0.05 0]';
[t,x]=ode45(@(t,x) HW3_Q6(t,x),0:0.01:4,q0);
tau = zeros(length(t), 2);
error = zeros(length(t), 4);
cmd_val = zeros(length(t), 3);
for i = 1:length(t)
[~, tau(i, :),error(i,:),cmd_val(i,:)] = HW3_Q6(t(i), x(i, :));
end

figure(1)
hold on
plot(t,x(:,1),'black')
plot(t,x(:,3),'red')
xlabel('time')
ylabel('theta')
title('Robot Joint Position Response')
legend('joint response 1','joint response 2')
hold off

figure(2)
hold on
plot(t,tau(:,1),'black')
plot(t,tau(:,2),'red')
xlabel('time')
ylabel('Nm')
title('Robot Torque Response')
legend('joint torque 1','joint torque 2')
hold off

figure(3)
hold on
plot(t,error(:,1),'black')
plot(t,error(:,2),'red')
```

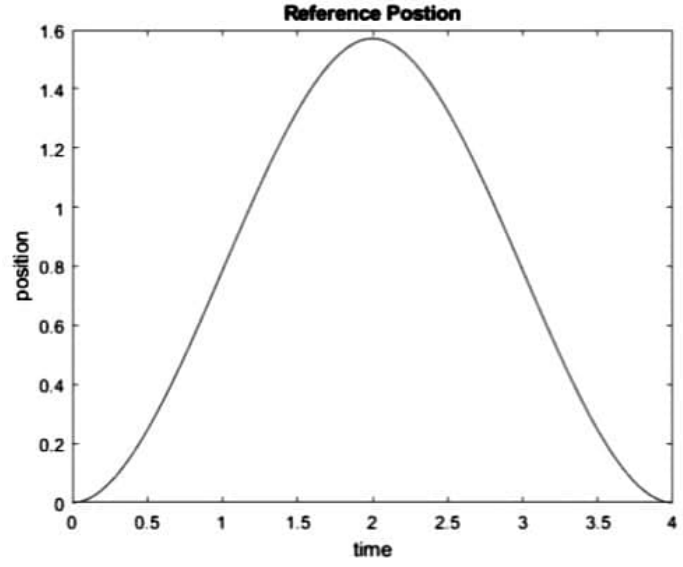
```
title('Position Error')
xlabel('time')
ylabel('m')
legend('position error 1','position error 2')
hold off

figure(4)
hold on
plot(t,error(:,3),'black')
plot(t,error(:,4),'red')
title('Velocity Error')
xlabel('time')
ylabel('m/s')
legend('velocity error 1','velocity error 2')
hold off

figure(5)
plot(t,cmd_val(:,1),'black')
xlabel('time')
ylabel('position')
title('Reference Postion')
figure(6)
plot(t,cmd_val(:,2),'black')
xlabel('time')
ylabel('velocity')
title('Reference Velocity')
figure(7)
plot(t,cmd_val(:,3),'black')
xlabel('time')
ylabel('acceleration')
title('Reference Acceleration')
```

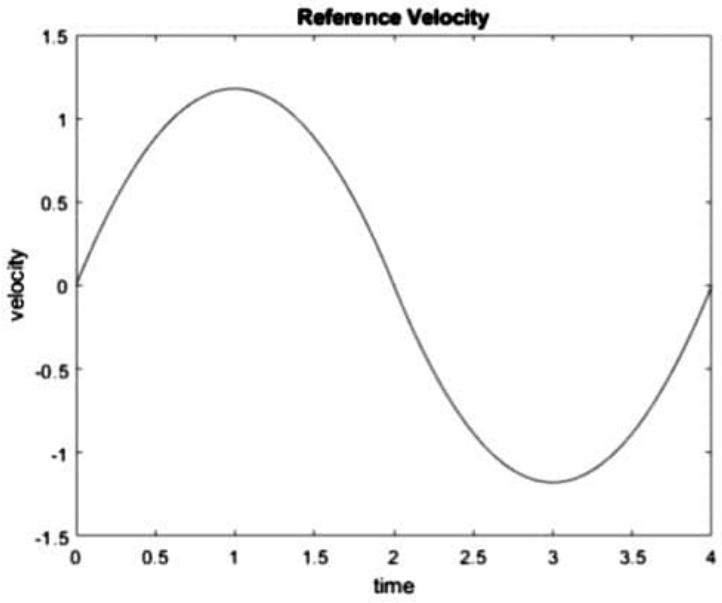

Robot Joint Position Response
joint response 1
joint response 2
theta
time

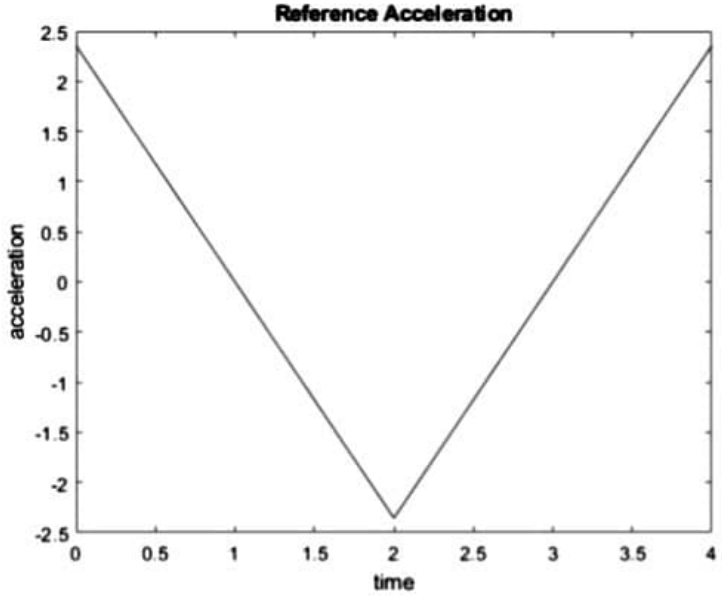

Robot Torque Response
joint torque 1
joint torque 2
Nm
time


Position Error
position error 1
position error 2
m
time


Velocity Error
velocity error 1
velocity error 2
m/s
time


Reference Postion
position
time





Published with MATLAB® R2024a

Initial Offset Impact (problem 6):
The small initial non-zero positions in problem 6 cause higher initial tracking errors and require larger joint torques to correct the positions.
This results in more pronounced oscillations and larger velocity/acceleration errors at the beginning.

Zero Start (problem 5(a)):
Starting from zero allows the system to track the desired trajectory with fewer oscillations, lower initial errors, and smoother torque profiles,
leading to better performance in the initial stages.

Overall System Stability:
While the overall system stabilizes in both cases, the initial conditions influence the magnitude of the error and the controller's required effort.
The zero-start case shows slightly better performance overall.