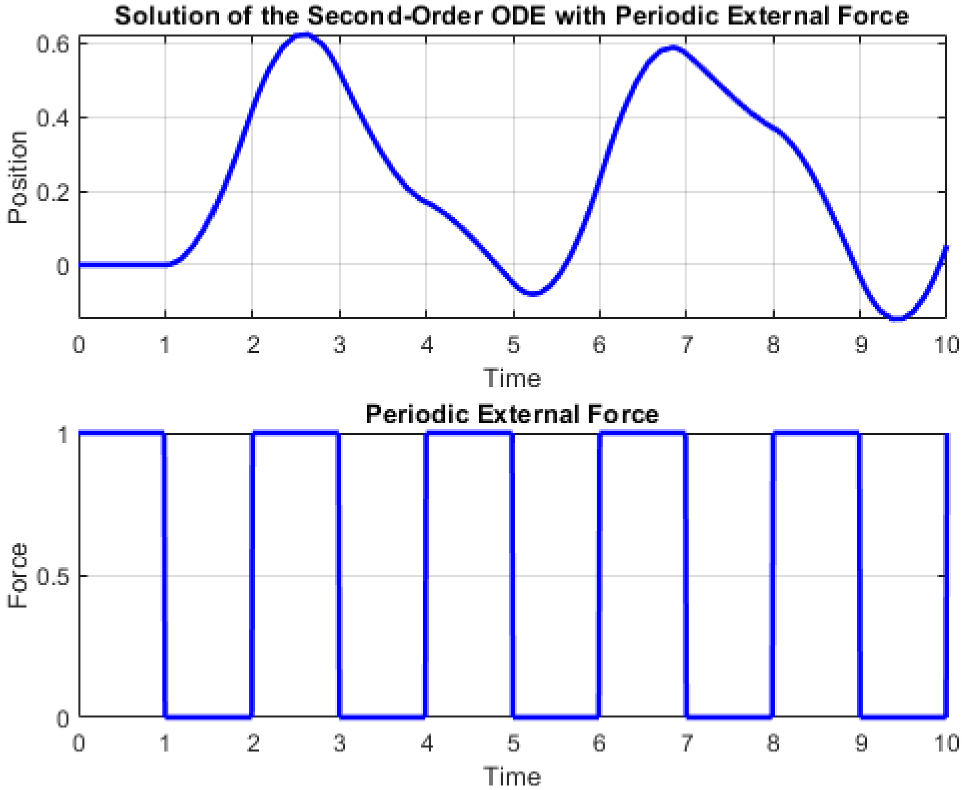
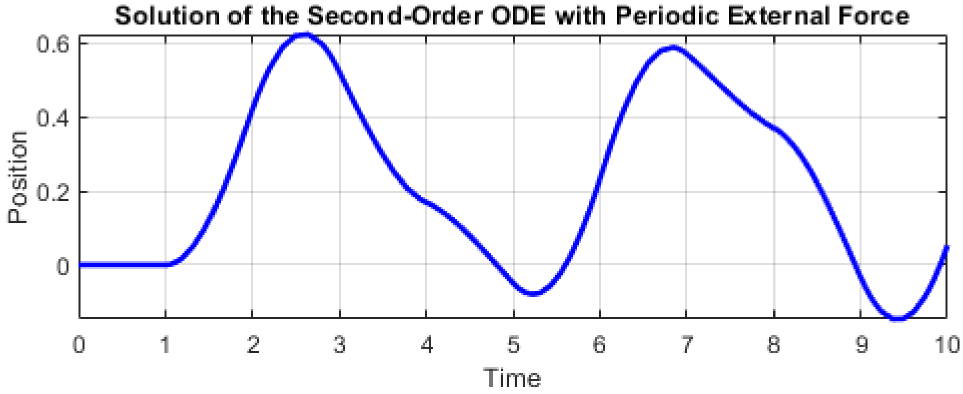
```
clc, clear, close all;
 %Question 1
 % Define parameters
    m = 1;       % Mass
    c = 0;       % Damping coefficient
    k = 2;       % Spring constant

    % Define time span
    tspan = [0 10];  % Time interval for simulation

    % Define initial conditions [x0, x'0]
    x0 = [0; 0];  % Initial position and velocity

    % Solve the ODE system
    [t, x] = ode45(@(t, x) odefun(t, x, m, c, k), tspan, x0);


    subplot(2,1,1)
    % Plot the solution
    plot(t, x(:, 1), 'b-', 'LineWidth', 2);  % Plot position vs. time
    xlabel('Time');
    ylabel('Position');
    title('Solution of the Second-Order ODE with Periodic External Force');
    grid on;
    subplot(2,1,2)
    t1 = 0:0.01:10;
    f = zeros(size(t1));
    for i = 1:numel(t1)
        if mod(t1(i), 2) < 1
            f(i) = 1;
        else
            f(i) = 0;
        end
    end
    plot(t1,f, 'b-', 'LineWidth', 2)%plot the force vs. time
    xlabel('Time');
    ylabel('Force');
    title('Periodic External Force');
    grid on;
```

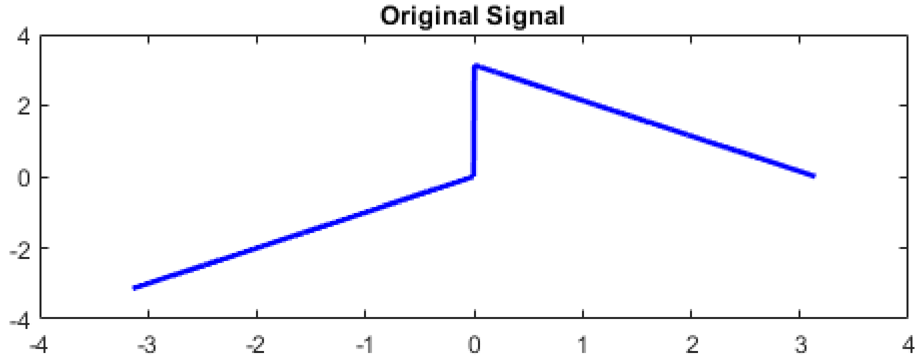


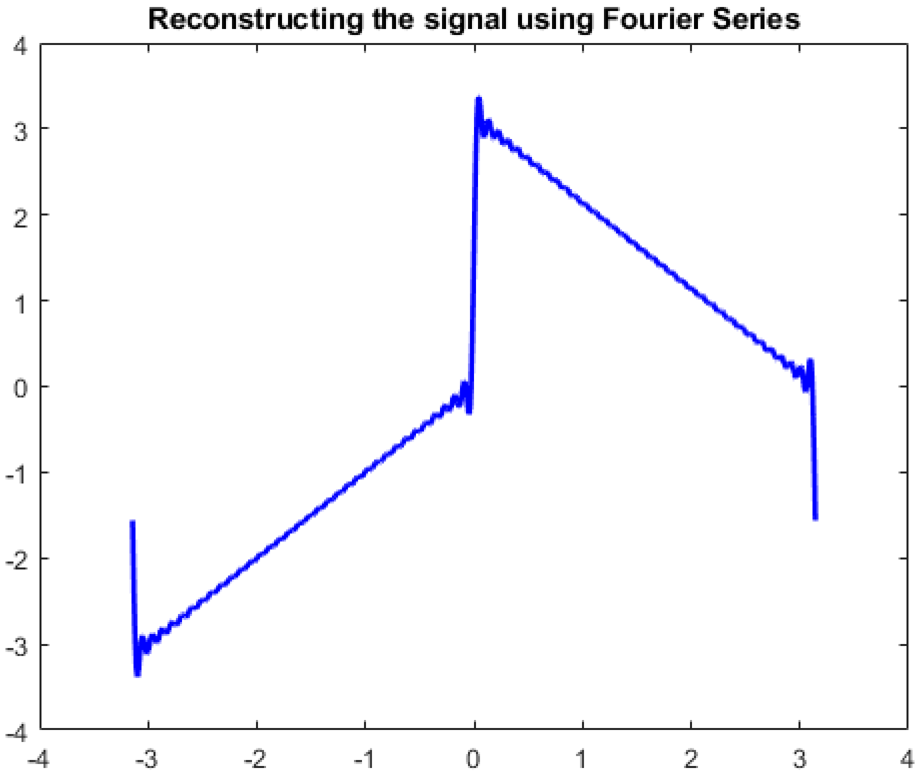

```
%Question 2
% Define the function f(x)
tolerance = 1.58;
f = @(x) (x .* (-pi <= x & x <= 0)) + ((pi - x) .* (0 <= x & x <= pi));

% Define the range of x
x_range = linspace(-pi, pi, 1000); % 1000 points between -pi and pi

% Initialize the Fourier series
fourier_series = zeros(size(x_range));
a0 = (1/(2*pi)) * integral(@(x) f(x), -pi, pi);
fourier_series = a0;
k=1;
while true
    % Compute the Fourier series

    % Compute the coefficients
    a_n = (1/pi) * integral(@(x) f(x) .* cos(k * x), -pi, pi);
    b_n = (1/pi) * integral(@(x) f(x) .* sin(k * x), -pi, pi);

    % Add the term to the Fourier series
    fourier_series = fourier_series + b_n * sin(k * x_range) + a_n * cos(k * x_range);

    % Compute the error
    error = abs(fourier_series - f(x_range));
    error_max = max(error);
    % Check if the maximum error is less than the tolerance
    if error_max < tolerance
        break;
    end
    k = k+1;

end
% Display the result
plot(x_range,f(x_range),'b-', 'LineWidth', 2)
title('Original Signal')
figure;
plot(x_range,fourier_series,'b-', 'LineWidth', 2)
```

```
    title('Reconstructing the signal using Fourier Series')
    disp(['Approximate value of k for tolerance ', num2str(tolerance), ': ', num2str(k)]);
```

```
Approximate value of k for tolerance 1.58: 69
```

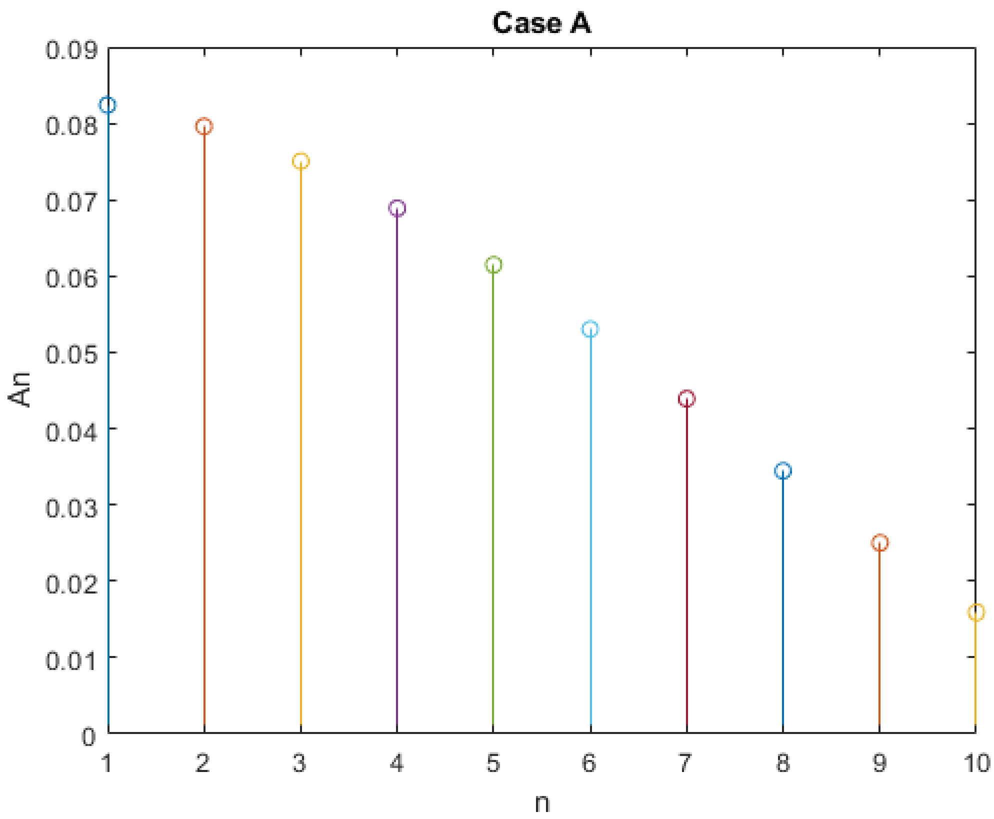


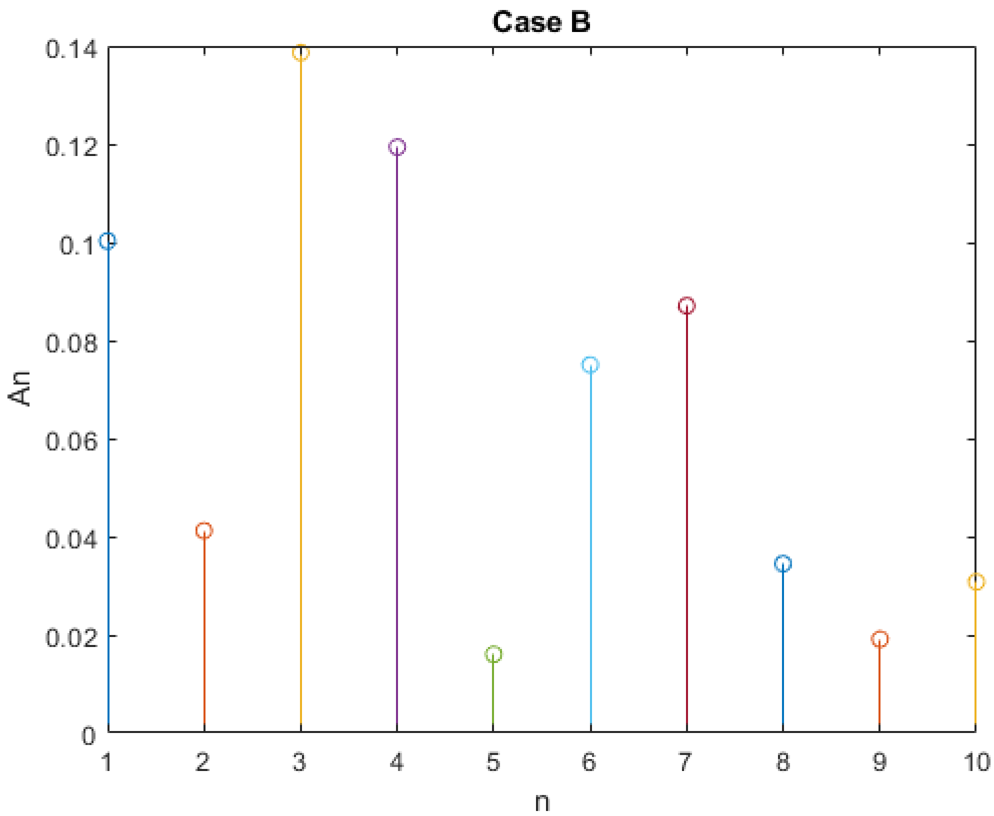





```
    %Question 3
    %section a
    % Define parameter
```

```
epsilon = pi/12;

% Create time vector
t = linspace(0, pi, 1000);

% Generate pulse
f = @(t) (0 <= t & t <= 2*epsilon);
N = 10;
n = 1:N;
for n=1:N
    % Compute the coefficients
    a_n = (1/pi) * integral(@(t) f(t) .* cos(n * t), 0, pi);
    b_n = (1/pi) * integral(@(t) f(t) .* sin(n * t), 0, pi);
    An = (1/2)*sqrt(a_n^2 + b_n^2);
    stem(n,An)
    title('Case A')
    xlabel('n')
    ylabel('An')
    hold on
end

% Generate pulse
f1 = @(t) ((0 <= t & t <= 2*epsilon) + ((7*pi/12) <= t & t <= (9*pi/12)));
figure;
for n=1:N
    % Compute the coefficients
    a_n1 = (1/pi) * integral(@(t) f1(t) .* cos(n * t), 0, pi);
    b_n1 = (1/pi) * integral(@(t) f1(t) .* sin(n * t), 0, pi);
    An = (1/2)*sqrt(a_n1^2 + b_n1^2);
    stem(n,An)
    title('Case B')
    xlabel('n')
    ylabel('An')
    hold on
end
%section b
```






```
%Question 4
% Step 1: Read the audio file

[y, Fs] = audioread('Audio01.wav');

% Step 2: Perform Fourier transform
Y = fft(y);

% Step 3: Reverse the order of frequency components
Y_reverse = Y(end:-1:1);
```

```
    % Step 4: Perform inverse Fourier transform
    y_reverse = ifft(Y_reverse);

    % Step 5: Write the reversed audio to a new file
    audiowrite('reversed_audio_file.wav', abs(y_reverse), Fs);
```

```
function dxdt = odefun(t, x, m, c, k)
    % Function defining the ODE system
    % Inputs:
    %   t: Time
    %   x: State vector [position; velocity]
    %   m: Mass
    %   c: Damping coefficient
    %   k: Spring constant

    % Define the periodic external force f(t)
    if mod(t, 2) < 1
        f = 0;
    else
        f = 1;
    end

    % Extract position and velocity from state vector
    pos = x(1);
    vel = x(2);

    % Compute acceleration (second derivative of position)
    acc = (f - c * vel - k * pos) / m;

    % Return the derivative of the state vector [velocity; acceleration]
    dxdt = [vel; acc];
end
```

Published with MATLAB® R2020b