Alexandria University

Faculty of Engineering

Mehatronics and Robotics Program

Signals and Systems

Final Project Report

5th Term

Submitted by:

Name:	Mohamed Moataz
ID:	9156
Name:	Marwan Bekhit
ID:	9023

Instructors:
Prof. Hassan El Ragal, Prof. Ahmed Eltrass, Prof. Mohamed Rizk

Teaching Assistants:
Eng. Hossam Hassan, Eng. Menna Taha, Eng. Yehia Ehab, Eng. Merna Mansour, Eng. Aya Gebril, Eng. Salma Magdy, Eng. Mohammed Rizk, Eng. Manar Amer

December 2025

Contents

1 Part I: Handwritten Analysis and MATLAB Simulation

1.1 Question 1: Piecewise Signal Analysis

1.1.1 Problem Statement

Given the following signal in time domain:

$$x(t) = \begin{cases} e^{-2t}u(t), & t < 2 \\ \cos(4\pi t)e^{-0.5t}, & t \geq 2 \end{cases}$$

Tasks:

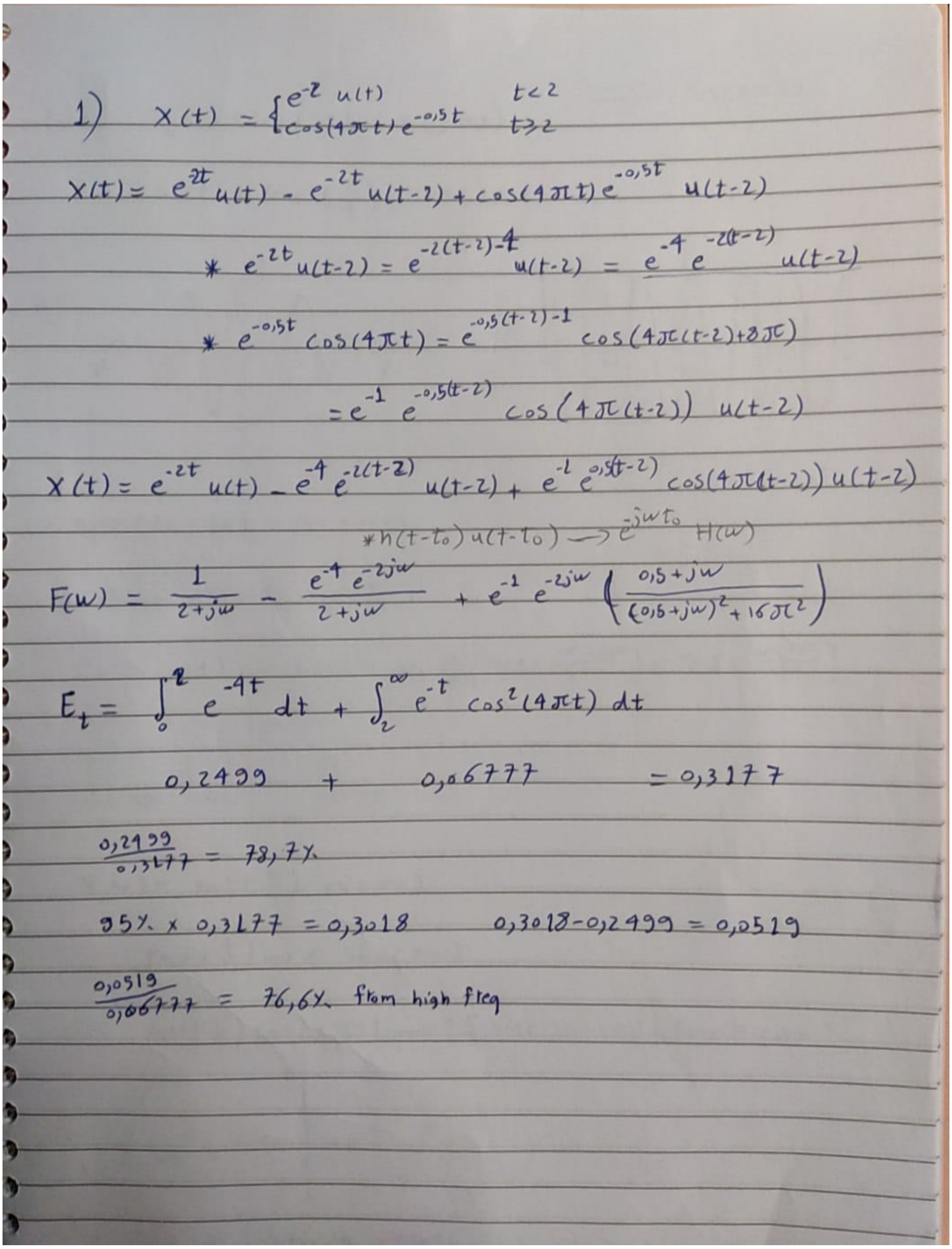
a. Derive a complete mathematical expression for $x(t)$, including all necessary shifted/exponential terms.

b. Compute and plot the Fourier Transform $X(\omega)$ using MATLAB.

c. Determine the bandwidth containing 95% of the signal energy.

d. Comment on how the presence of the shifted exponential affects the spectral shape.

1.1.2 Handwritten Solution



1) $x(t) = \begin{cases} e^{-2} u(t) & t<2 \\ \cos(4\pi t)\, e^{-0,5t} & t\geq 2 \end{cases}$

$x(t) = e^{2t} u(t) - e^{-2t} u(t-2) + \cos(4\pi t)\, e^{-0,5t}\, u(t-2)$

$* \; e^{-2t} u(t-2) = e^{-2(t-2)-4} u(t-2) = e^{-4} e^{-2(t-2)} u(t-2)$

$* \; e^{-0,5t} \cos(4\pi t) = e^{-0,5(t-2)-1} \cos(4\pi(t-2)+8\pi)$

$= e^{-1} e^{-0,5(t-2)} \cos(4\pi(t-2))\, u(t-2)$

$x(t) = e^{-2t} u(t) - e^{-4} e^{-2(t-2)} u(t-2) + e^{-1} e^{0,5(t-2)} \cos(4\pi(t-2))\, u(t-2)$

$* \; h(t-t_0)\, u(t-t_0) \longrightarrow e^{-j\omega t_0} H(\omega)$

$F(\omega) = \frac{1}{2+j\omega} - \frac{e^{-4} e^{-2j\omega}}{2+j\omega} + e^{-1} e^{-2j\omega} \left(\frac{0,5+j\omega}{(0,5+j\omega)^2 + 16\pi^2} \right)$

$E_t = \int_0^2 e^{-4t}\, dt + \int_2^\infty e^{-t} \cos^2(4\pi t)\, dt$

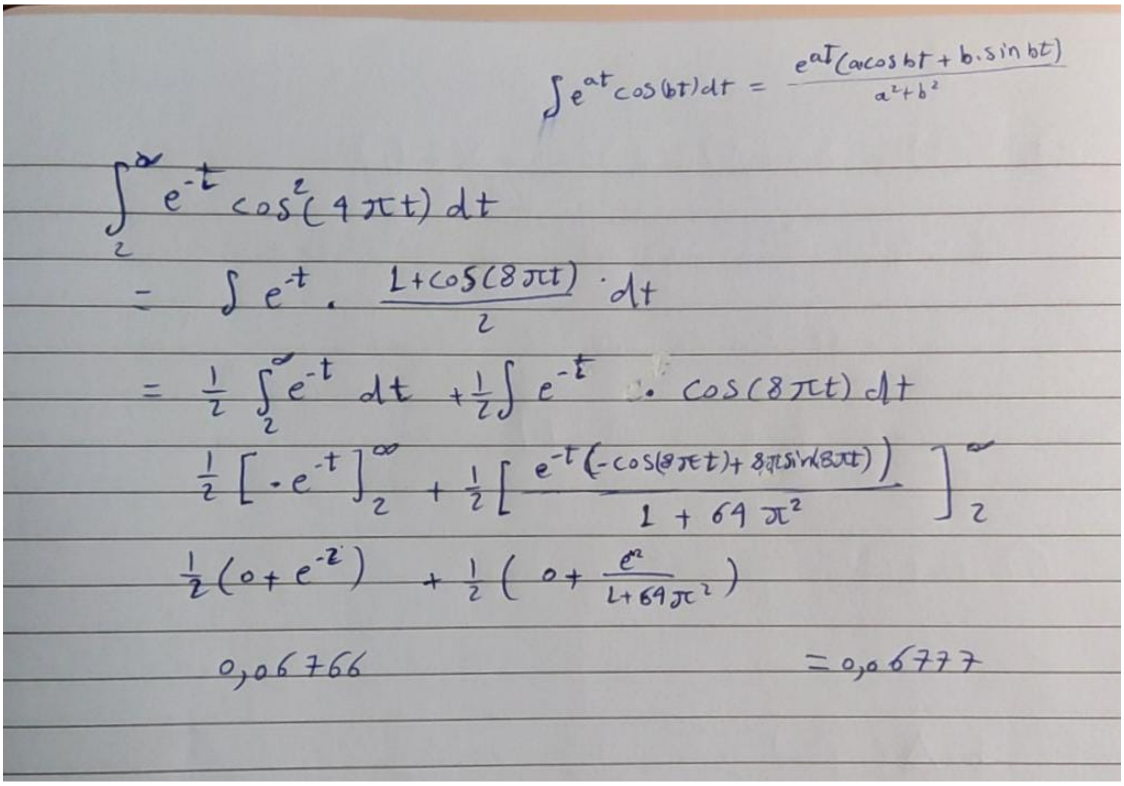
0,2499 + 0,06777 = 0,3177

$\frac{0,2499}{0,3177} = 78,7\%$

95% x 0,3177 = 0,3018   0,3018 - 0,2499 = 0,0519

$\frac{0,0519}{0,06777} = 76,6\%$ from high freq

Figure 1: Question 1 - Handwritten Solution (Page 1)



$$\int e^{at}\cos(bt)\,dt = \frac{e^{at}(a\cos bt + b\cdot\sin bt)}{a^2+b^2}$$

$$\int_2^{\infty} e^{-t}\cos^2(4\pi t)\,dt$$

$$= \int e^{-t}\cdot\frac{1+\cos(8\pi t)}{2}\cdot dt$$

$$= \frac{1}{2}\int_2^{\infty} e^{-t}\,dt + \frac{1}{2}\int e^{-t}\cdot\cos(8\pi t)\,dt$$

$$\frac{1}{2}\left[-e^{-t}\right]_2^{\infty} + \frac{1}{2}\left[\frac{e^{-t}(-\cos(8\pi t)+8\pi\sin(8\pi t))}{1+64\pi^2}\right]_2^{\infty}$$

$$\frac{1}{2}(0+e^{-2}) + \frac{1}{2}\left(0+\frac{e^{2}}{1+64\pi^2}\right)$$

$$0{,}06766 \qquad = 0{,}06777$$

Figure 2: Question 1 - Handwritten Solution (Page 2)

1.1.3 MATLAB Implementation

```
Fs = 10000;
t = linspace(0, 10,Fs);
x = zeros(size(t));
dt = t(2) - t(1);

mask1 = (t < 2);
x(mask1) = exp(-2 * t(mask1));

mask2 = (t >= 2);
x(mask2) = cos(4 * pi * t(mask2)) .* exp(-0.5 * t(mask2));

figure;
plot(t, x, 'LineWidth', 1.5);
grid on;
xlabel('Time (s)');
ylabel('Amplitude');
title('Signal x(t)');

%% Fourier Transform
frq = linspace(-50, 50, 10000);
omega = 2*pi*frq;
X1 = (1-exp(-4) * exp(-1j*2*omega))./ (2 + 1j*omega);
X2 = (exp(-1) * exp(-1j*2*omega)  .* ( 0.5+1j*omega) ./  (0.25 +  1j*omega -omega.^2 + 16*pi^2) );
X = X1 + X2 ;

figure;
subplot(2,1,1);
plot(omega, abs(X));
xlabel('\omega (rad/s)');
ylabel('|X(\omega)|');
title('Analytical Fourier Transform Magnitude');
grid on;xlim([-50,50]);

subplot(2,1,2);
plot(omega, angle(X));
xlabel('\omega (rad/s)');
ylabel('Phase (rad)');
title('Phase Spectrum');
grid on;xlim([-50,50]);

%% Part 1(c): 95% Energy Bandwidth
L = length(t);
f = linspace(-Fs/2, Fs/2, L);
w = 2 * pi * f;
X_f = fft(x);
X_w = fftshift(X_f) * dt;
energy_density = (abs(X_w).^2) / (2*pi);
total_energy = sum(abs(x).^2) * dt;
pos_mask = w >= 0;
w_pos = w(pos_mask);
density_pos = energy_density(pos_mask);
c_energy = cumsum(density_pos) * dw * 2;
c_energy_percent = c_energy / total_energy;
target_index = find(c_energy_percent >= 0.95, 1);
bandwidth_95 = w_pos(target_index);

fprintf('Total Energy: %.4f Joules\n', total_energy);
fprintf('95%% Energy Bandwidth: %.4f rad/s\n', bandwidth_95);
```

Listing 1: Question 1 - MATLAB Code

1.1.4 Results and Analysis

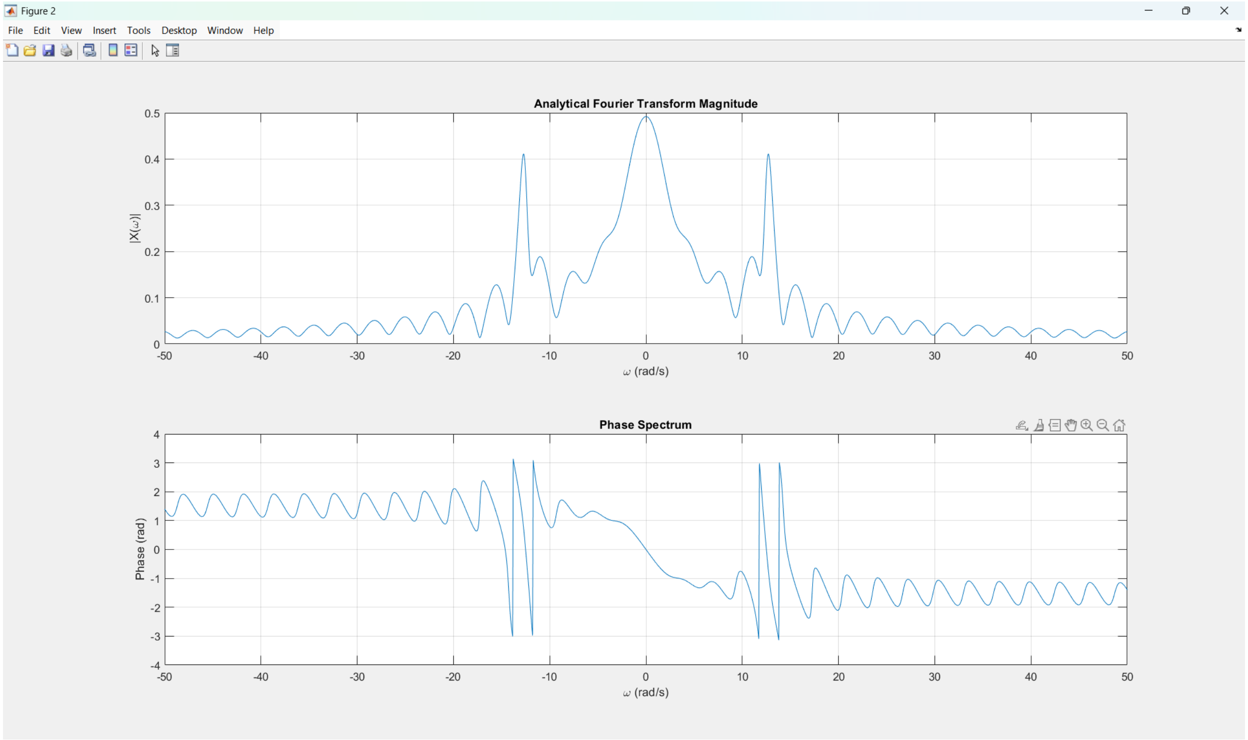


Figure 3: Question 1 - Signal in Time Domain

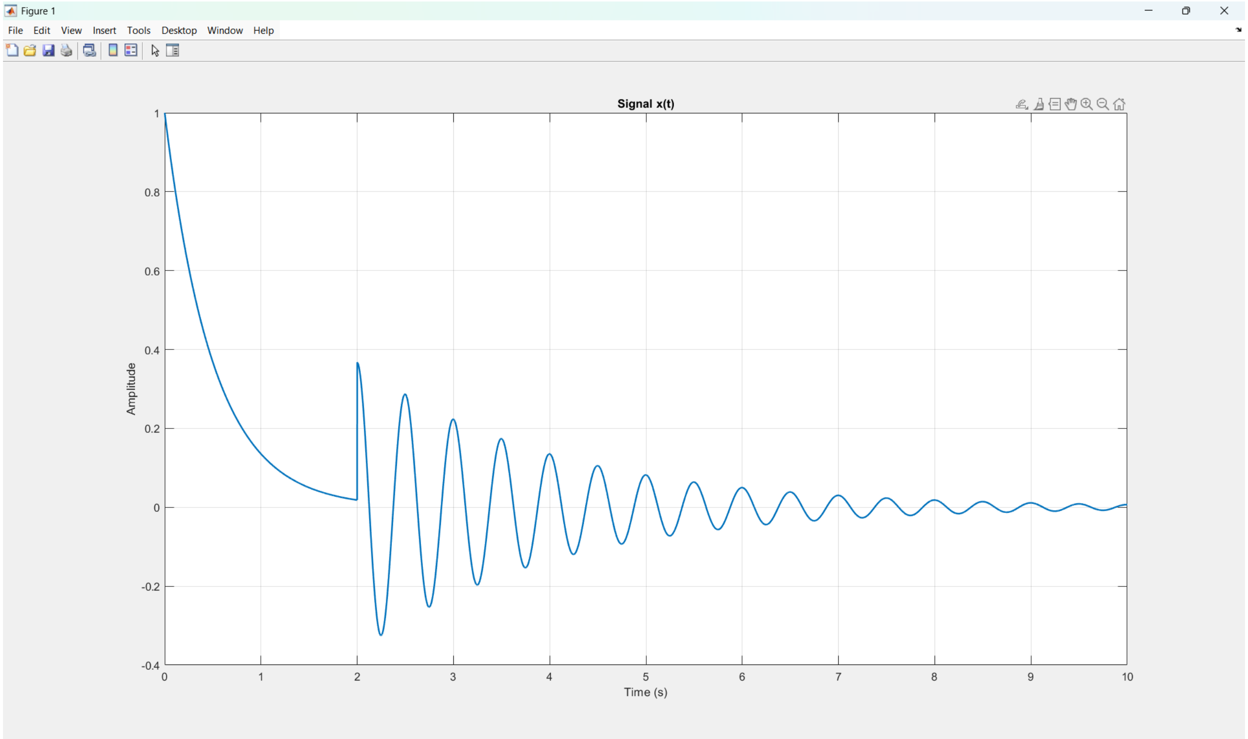


Figure 4: Question 1 - Fourier Transform Magnitude and Phase

1.2 Question 2: Signal Comparison and Bandwidth Efficiency

1.2.1 Problem Statement

Consider the following two signals:

$$x_1(t) = \operatorname{sinc}(2t)\cos(3\pi t)$$
$$x_2(t) = \operatorname{rect}\left(\frac{t}{4}\right) * \cos(3\pi t)$$

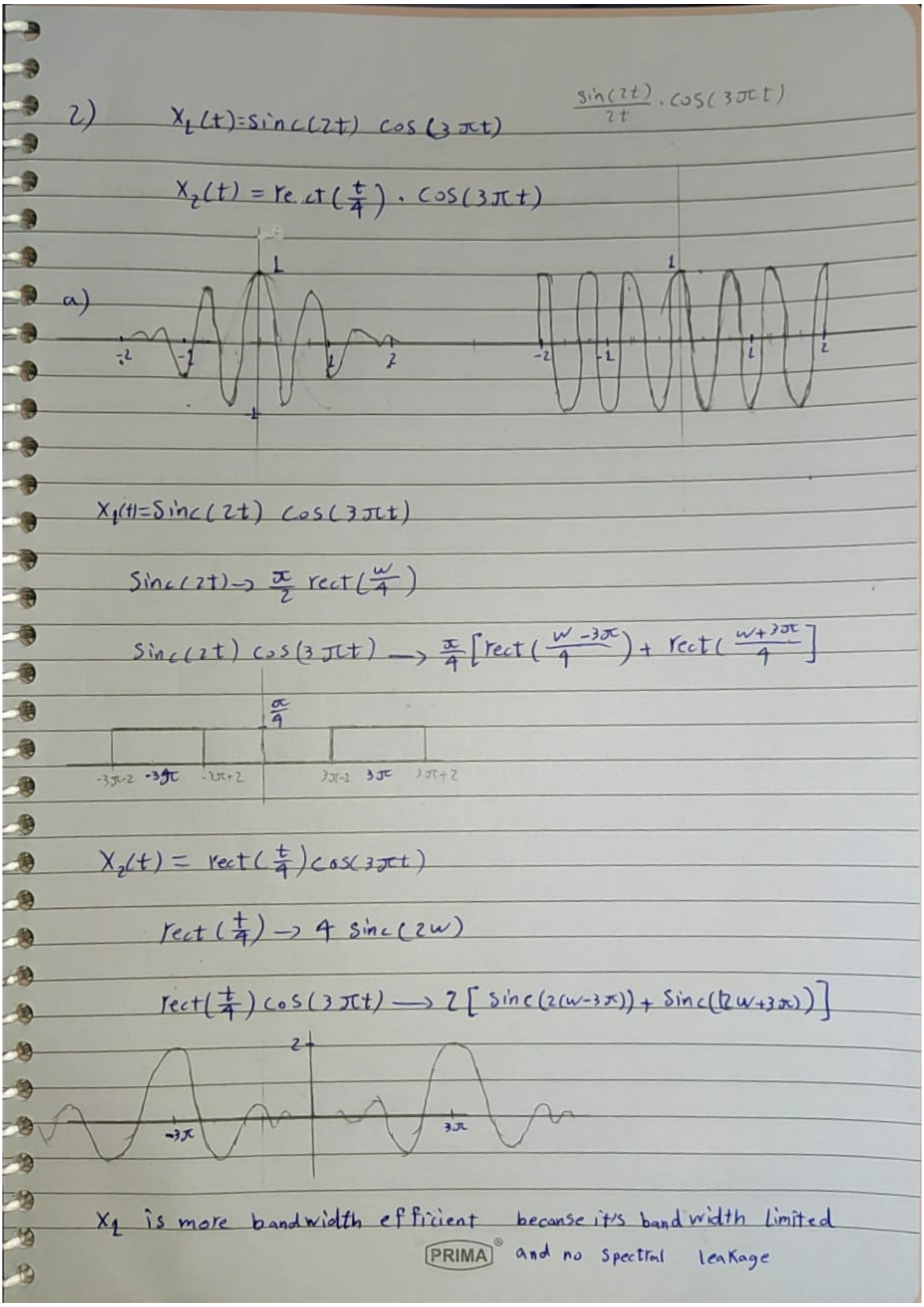
Tasks:

a. Plot both signals in time domain.

b. Derive and plot their Fourier Transform using properties (convolution, modulation, scaling).

c. Compare the magnitude spectra and discuss which signal is more bandwidth efficient and why.

1.2.2 Handwritten Solution



2) $x_1(t) = \text{sinc}(2t)\cos(3\pi t)$ $\quad \frac{\sin(2t)}{2t} \cdot \cos(3\pi t)$

$x_2(t) = \text{rect}\left(\frac{t}{4}\right) \cdot \cos(3\pi t)$

a)

$x_1(t) = \text{Sinc}(2t)\cos(3\pi t)$

$\text{Sinc}(2t) \to \frac{\pi}{2}\,\text{rect}\left(\frac{w}{4}\right)$

$\text{Sinc}(2t)\cos(3\pi t) \to \frac{\pi}{4}\left[\text{rect}\left(\frac{w-3\pi}{4}\right) + \text{rect}\left(\frac{w+3\pi}{4}\right)\right]$

$x_2(t) = \text{rect}\left(\frac{t}{4}\right)\cos(3\pi t)$

$\text{rect}\left(\frac{t}{4}\right) \to 4\,\text{sinc}(2w)$

$\text{rect}\left(\frac{t}{4}\right)\cos(3\pi t) \to 2\left[\text{sinc}(2(w-3\pi)) + \text{Sinc}((2w+3\pi))\right]$

x_1 is more bandwidth efficient because it's bandwidth limited and no spectral leakage

Figure 5: Question 2 - Handwritten Solution

1.2.3 MATLAB Implementation

```
fs=10000;
Ts=1/fs;
t=linspace(-10,10,20*fs);
x1=(sin(2*t)./(2*t)).*(cos(3*pi*t));
x2=(t>=-2 & t<=2).* (cos(3*pi*t));

subplot(2,2,1);
plot(t, x1);
xlabel('t'); ylabel('X1');
title('signal (1)');
grid on;
subplot(2,2,2);
plot(t, x2);
xlabel('t'); ylabel('X2');
title('signal (2)');
grid on;

rect = @(m) (m>=-0.5).*(m<=0.5);
f = linspace(-20, 20, 5000);
w=2*pi*f;
W1 = 0.25*pi*( rect((w-3*pi)/4) + rect((w+3*pi)/4));
W2 = 2*( ( sin(2*(w-3*pi)) ./ (2*(w-3*pi)) ) + (sin(2*(w+3*pi))./(2*(w+3*pi))) );

subplot(2,2,3);
plot(w , W1);
xlabel('W'); ylabel('W1');
title('fourier transform 1');
grid on; xlim([-20,20]);
subplot(2,2,4);
plot(w , W2);
xlabel('W'); ylabel('W2');
title('fourier transform 2');
grid on; xlim([-20,20]);

%% Phase & Magnitude
figure
subplot(2,2,1);
plot(w , abs(W1));
xlabel('W'); ylabel('W1');
title('magnitude 1');
grid on; xlim([-20,20]);

subplot(2,2,2);
plot(w , abs(W2));
xlabel('W'); ylabel('W2');
title('magnitude 2');
grid on; xlim([-20,20]);

subplot(2,2,3);
plot(w , angle(W1));
xlabel('W'); ylabel('W1');
title('phase 1');
grid on; xlim([-20,20]);

subplot(2,2,4);
plot(w , angle(W2));
xlabel('W'); ylabel('W2');
title('phase 2');
grid on; xlim([-20,20]);
```

Listing 2: Question 2 - MATLAB Code

1.2.4 Results and Analysis

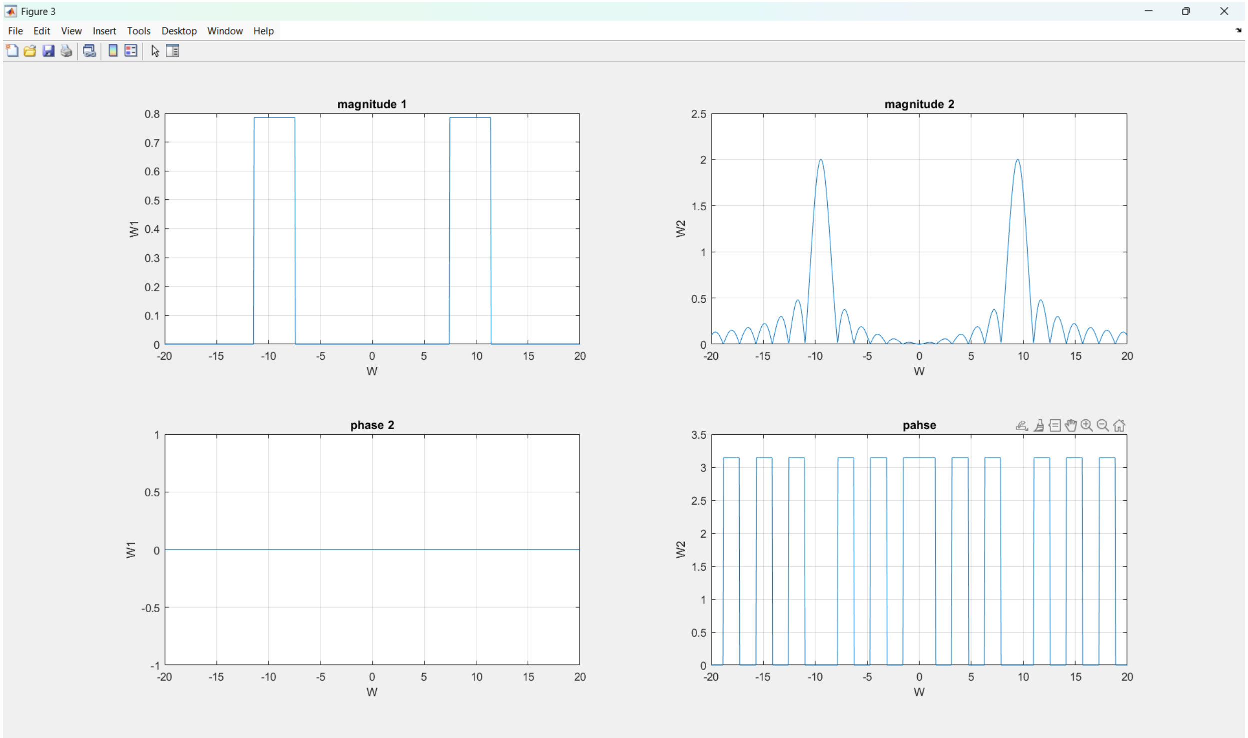


Figure 6: Question 2 - Time Domain Signals and Fourier Transforms

1.3 Question 3: Fourier Series Analysis

1.3.1 Problem Statement

A periodic signal $x(t)$ of period $T = 4$ is defined as a triangular pulse followed by a flat top:

$$x(t) = \begin{cases} t+2, & -2 \le t < 0 \\ 2, & 0 \le t < 2 \end{cases}$$

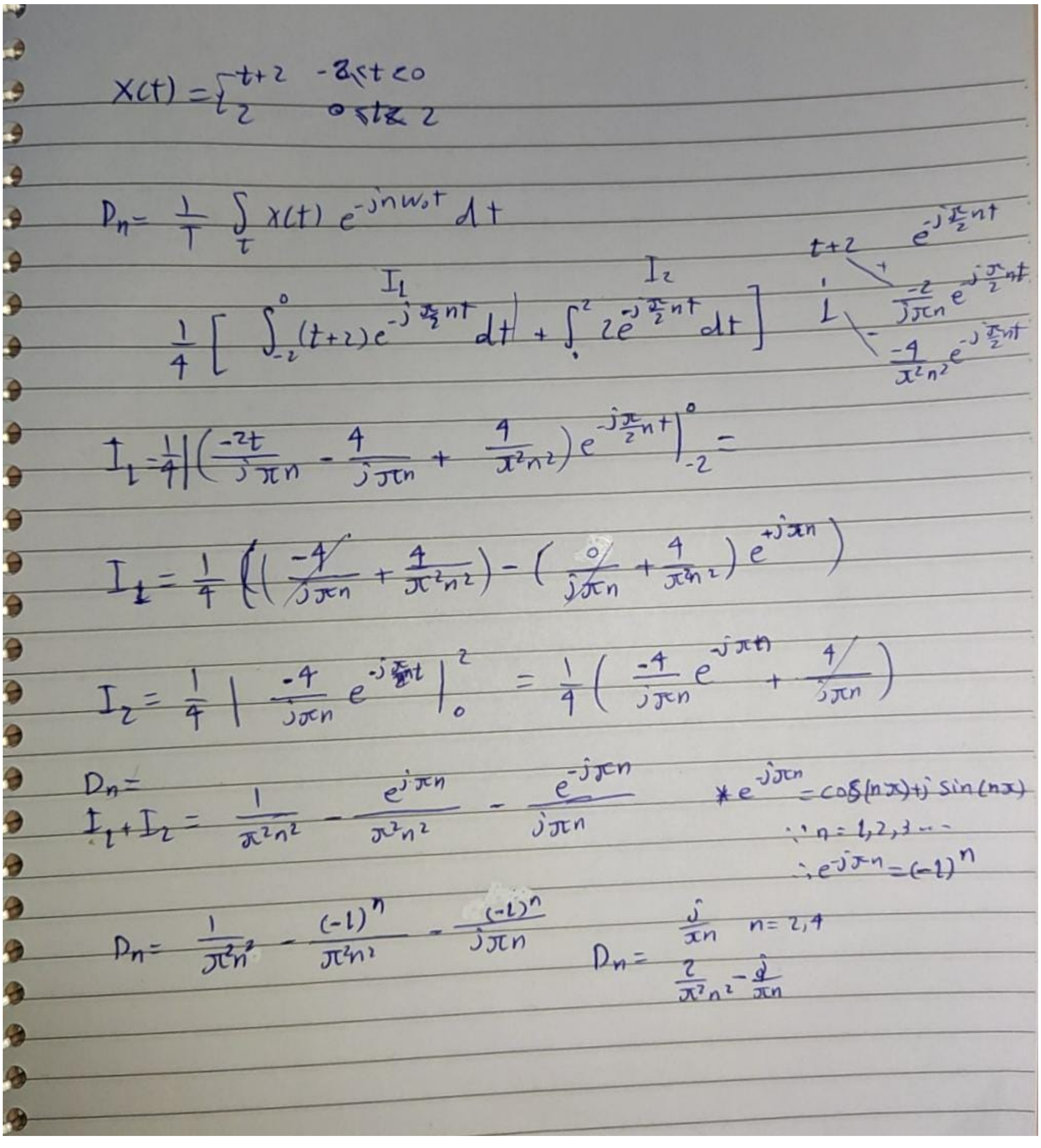
Tasks:

a. Derive the Fourier series coefficients D_n.

b. Plot the magnitude and phase of the Fourier series.

c. Using MATLAB, reconstruct the signal using the first 15 harmonics, then compare with the original $x(t)$.

1.3.2 Handwritten Solution



$$x(t) = \begin{cases} t+2 & -2 \le t < 0 \\ 2 & 0 \le t < 2 \end{cases}$$

$$D_n = \frac{1}{T}\int_T x(t)\, e^{-jn\omega_0 t}\, dt$$

$$\frac{1}{4}\Big[\underbrace{\int_{-2}^{0} (t+2) e^{-j\frac{\pi}{2}nt} dt}_{I_1} + \underbrace{\int_0^2 2e^{-j\frac{\pi}{2}nt} dt}_{I_2} \Big]$$

$t+2$ $\quad e^{-j\frac{\pi}{2}nt}$

1 $\quad \frac{-2}{j\pi n} e^{-j\frac{\pi}{2}nt}$

0 $\quad \frac{-4}{\pi^2 n^2} e^{-j\frac{\pi}{2}nt}$

$$I_1 = \frac{1}{4}\left|\left(\frac{-2t}{j\pi n} - \frac{4}{j\pi n} + \frac{4}{\pi^2 n^2}\right) e^{-j\frac{\pi}{2}nt}\right|_{-2}^{0} =$$

$$I_1 = \frac{1}{4}\left(\left(\frac{-4}{j\pi n} + \frac{4}{\pi^2 n^2}\right) - \left(\frac{0}{j\pi n} + \frac{4}{\pi^2 n^2}\right) e^{+j\pi n}\right)$$

$$I_2 = \frac{1}{4}\left|\frac{-4}{j\pi n} e^{-j\frac{\pi}{2}nt}\right|_0^2 = \frac{1}{4}\left(\frac{-4}{j\pi n} e^{-j\pi n} + \frac{4}{j\pi n}\right)$$

$$D_n = I_1 + I_2 = \frac{1}{\pi^2 n^2} - \frac{e^{j\pi n}}{\pi^2 n^2} - \frac{e^{-j\pi n}}{j\pi n}$$

$* \; e^{-j\pi n} = \cos(n\pi) + j\sin(n\pi)$

$\because n = 1, 2, 3 \ldots$

$\therefore e^{-j\pi n} = (-1)^n$

$$D_n = \frac{1}{\pi^2 n^2} - \frac{(-1)^n}{\pi^2 n^2} - \frac{(-1)^n}{j\pi n}$$

$$D_n = \begin{cases} \frac{j}{\pi n} & n = 2, 4 \\ \frac{2}{\pi^2 n^2} - \frac{j}{\pi n} \end{cases}$$

Figure 7: Question 3 - Handwritten Solution

1.3.3 MATLAB Implementation

```
N = 15;
n = -N:N;
Dn = zeros(size(n));

for i = 1:length(n)
    nv = n(i);
    if nv == 0
        Dn(i) = 1.5;
    elseif mod(nv, 2) == 0
        Dn(i) = 1j/(nv*pi);
    else
        Dn(i) =  2/(nv^2*pi^2) - 1j/(nv*pi);
    end
end

figure('Name', 'Fourier Series Spectra', 'Color', 'w');

subplot(2,1,1);
stem(n, abs(Dn));
grid on;
title('Magnitude Spectrum |D_n|');
xlabel('Harmonic Number n');
ylabel('Amplitude');

subplot(2,1,2);
stem(n, angle(Dn));
grid on;
title('Phase Spectrum \angle D_n (radians)');
xlabel('Harmonic Number n');
ylabel('Phase (rad)');

%% Reconstruction
T=4;
w0=2*pi/T;

t=-2:0.01:2;
Smain = (t+2).*(t<=0) + 2.*(t>0);
SDn = zeros(size(t));

for k = 1:length(n)
    SDn = SDn + Dn(k)*exp(1j*n(k)*w0*t);
end

SDn = real(SDn);
figure
plot(t, SDn, 'r', 'LineWidth', 2)
hold on
plot(t, Smain, 'k--', 'LineWidth', 2)
grid on
xlabel('t')
ylabel('x(t)')
legend('Reconstructed from D_n','Original Signal')
title(['Fourier Series Reconstruction, N = ', num2str(N)])
```

Listing 3: Question 3 - MATLAB Code

1.3.4 Results and Analysis

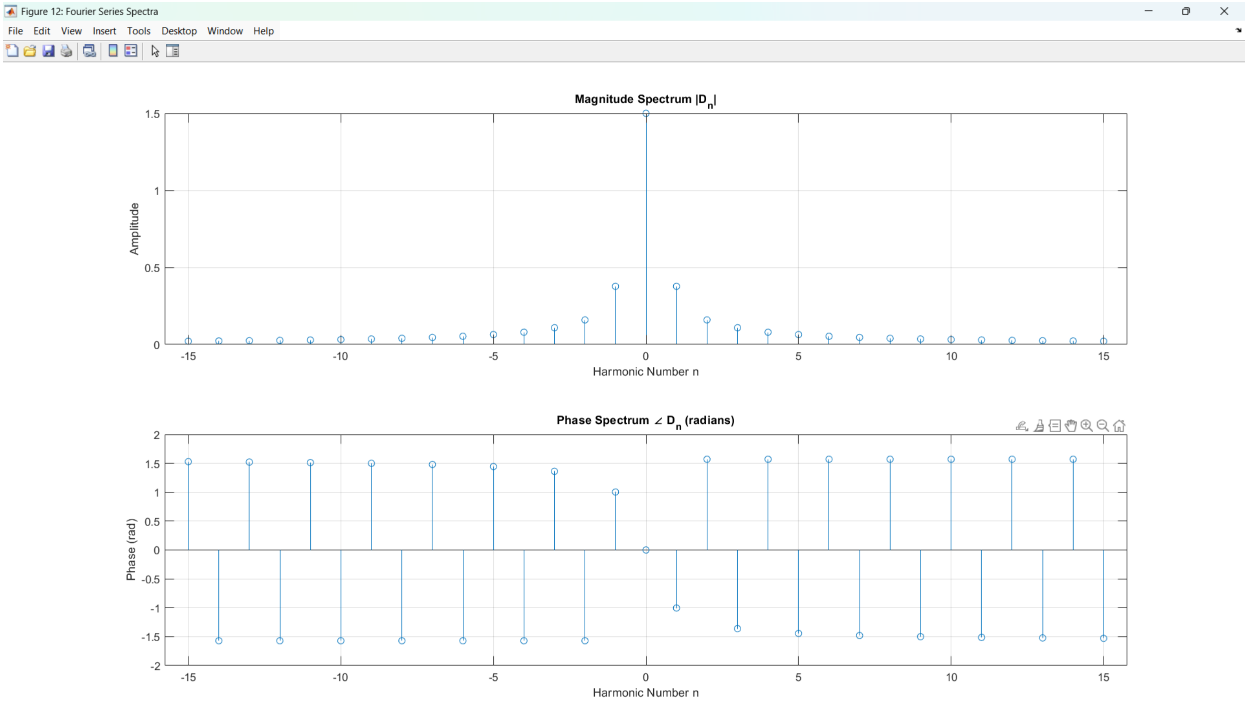


Figure 8: Question 3 - Fourier Series Magnitude and Phase Spectrum

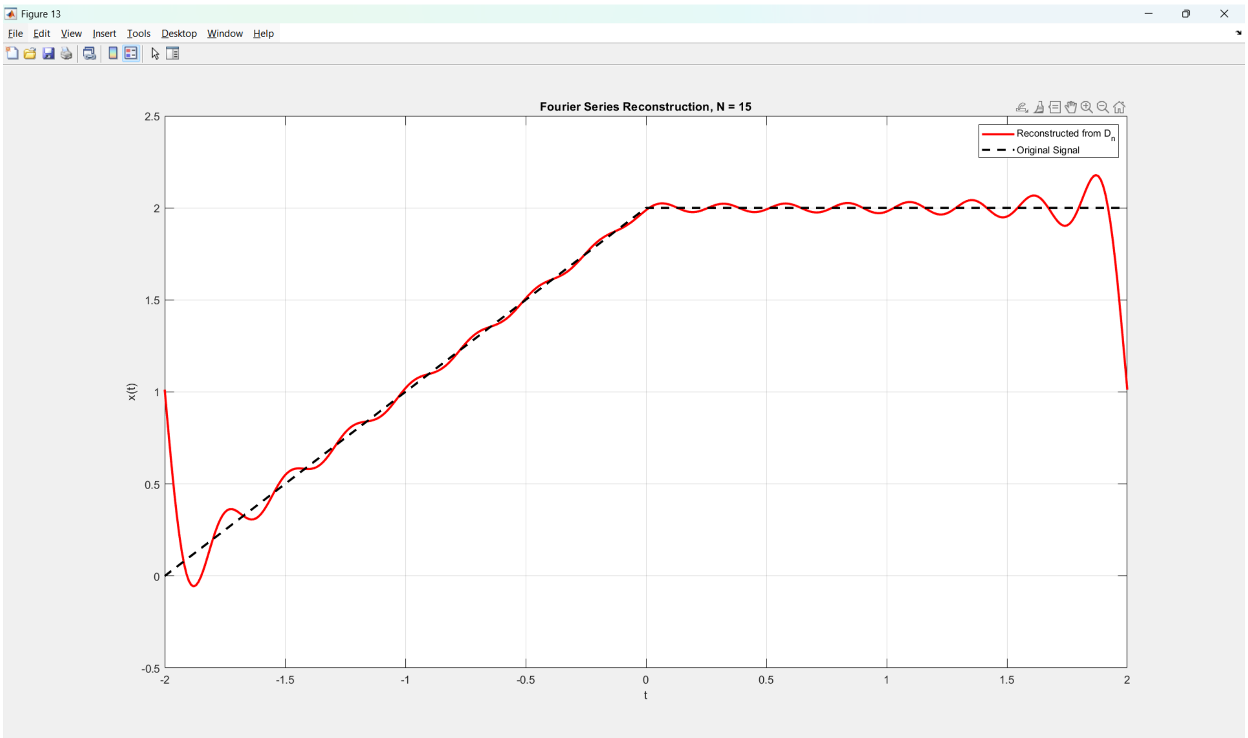


Figure 9: Question 3 - Signal Reconstruction using First 15 Harmonics

2 Part II: Modular Signal Generator in MATLAB

2.1 Overview

This section presents a comprehensive MATLAB program that generates complex piecewise signals with user-defined specifications. The program includes:

1. **Signal Generator**: Creates custom signals (single or combined with breakpoints)
2. **Signal Operations**: Applies various transformations to generated signals
3. **Random Generation**: Automatically generates multiple signals with random parameters

2.2 Main Program Structure

```
clc;
clear;
close all;
% Display welcome banner
fprintf('\n');
fprintf('===========================================================================\n');
fprintf('                        SIGNAL GENERATOR PROGRAM\n');
fprintf('===========================================================================\n\n');
% TIME CONFIGURATION
fprintf('--- TIME CONFIGURATION ---\n');
t_start = input('Start Time (s) = ');
t_end = input('End Time (s) = ');

while t_end <= t_start
    fprintf('Error: End time must be greater than start time!\n');
    t_start = input('Start Time (s) = ');
    t_end = input('End Time (s) = ');
end
F_s = input('Sampling Frequency (Hz) = ');
while F_s <= 0
    fprintf('Error: Sampling frequency must be positive!\n');
    F_s = input('Sampling Frequency (Hz) = ');
end

T_s = 1 / F_s;
t = t_start:T_s:t_end;

fprintf('\n--- TIME CONFIGURATION SUMMARY ---\n');
fprintf('Duration: %.4f seconds\n', t_end - t_start);
fprintf('Sampling Period (Ts): %.6f seconds\n', T_s);
fprintf('Sampling Frequency (Fs): %.2f Hz\n', F_s);
fprintf('Number of Samples: %d\n', length(t));

mode = get_generation_mode();
if mode == 1
    fprintf('\nWould you like to combine multiple signals at breakpoints?\n');
    combine_signals_flag = input('Enter 1 for Yes, 0 for No: ');
else
    combine_signals_flag = 0;
end

try
    if mode == 1
      if combine_signals_flag == 1
            [x_t, t] = combine_signals(t_start, t_end, F_s);
      else
        choice = get_signal_choice();

        switch choice
            case 1: x_t = DC(t);
            case 2: x_t = Ramp(t);
            case 3: x_t = Exponential(t);
            case 4: [x_t, t, F_s] = Sinusoidal(t, F_s, t_start, t_end);
            case 5: x_t = Gaussian_Pulse(t);
            case 6: [x_t, t, F_s] = Sawtooth_Wave(t, F_s, t_start, t_end);
            case 7: x_t = Polynomial(t);
        end
      end

        [x_final, t_final, operation_history] = apply_operations(x_t, t);

    else
        num_signals = input('\nEnter the number of random signals to generate: ');
        signals = Random(t, num_signals);
    end

    fprintf('\nPROGRAM COMPLETED SUCCESSFULLY!\n');

catch ME
    fprintf('\nError occurred: %s\n', ME.message);
    rethrow(ME);
end
```

Listing 4: Main Program - main.m

2.3 Time Configuration

```
--- TIME CONFIGURATION SUMMARY ---
Duration: 20.0000 seconds
Sampling Period (Ts): 0.000167 seconds
Sampling Frequency (Fs): 6000.00 Hz
Number of Samples: 120001
```

2.4 Signal Generator - Individual Signal Types

2.4.1 1. DC Signal (Constant)

```
function x_t = DC(t)
    amplitude = input('The Amplitude (DC Level) = ');
    x_t = amplitude * ones(size(t));

    figure;
    plot(t, x_t, 'r-', 'LineWidth', 2);
    xlabel('Time (s)'); ylabel('Amplitude');
    title(sprintf('DC Signal: x(t) = %g', amplitude));
    grid on;
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3, 'Label', 'Zero');
    yline(amplitude, 'r--', 'LineWidth', 1, 'Alpha', 0.5);
    hold off;

    fprintf('\n--- DC Signal Properties ---\n');
    fprintf('DC Level (Amplitude): %.4f\n', amplitude);
    fprintf('Signal Type: Constant\n');
    fprintf('- Frequency: 0 Hz (no oscillation)\n');
    fprintf('- RMS Value: %.4f\n', abs(amplitude));
    fprintf('- Average Value: %.4f\n', amplitude);
end
```

Listing 5: DC Signal Generator

```
--- DC Signal Properties ---
DC Level (Amplitude): 50.0000
Signal Type: Constant
Time Duration: 20.0000 s (from -10.0000 to 10.0000 s)
Characteristics:
- Frequency: 0 Hz (no oscillation)
- RMS Value: 50.0000
- Average Value: 50.0000
- Peak-to-Peak: 0 (constant)
```

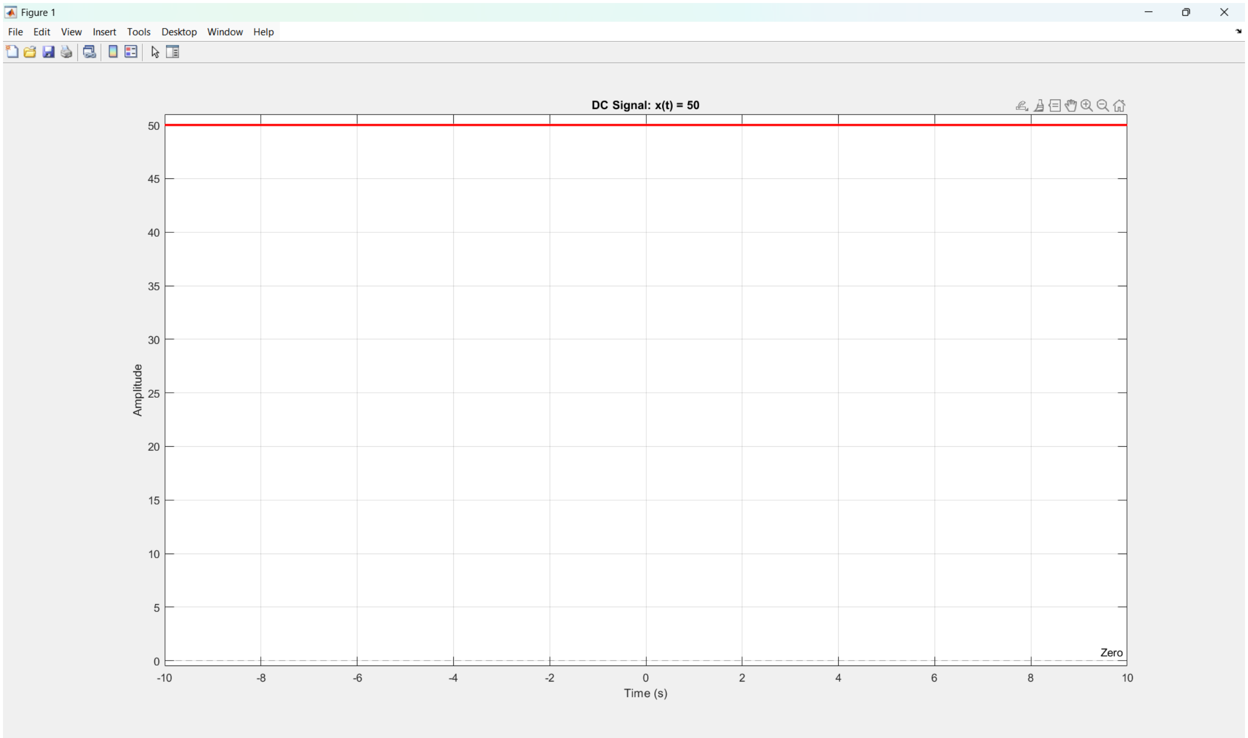


Figure 10: DC Signal Output

2.4.2 2. Ramp Signal (Linear)

```
function x_t = Ramp(t)
    slope = input('The Slope = ');
    intercept = input('The Intercept = ');
    % Generate ramp signal
    x_t = slope * t + intercept;
    % Determine signal type based on slope
    if slope > 0
        signal_type = 'Increasing';
    elseif slope < 0
        signal_type = 'Decreasing';
    else
        signal_type = 'Constant (Zero Slope)';
    end

    % Plot the signal
    figure;
    plot(t, x_t, 'g-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Ramp Signal: x(t) = %g*t + %g', slope, intercept));
    grid on;
    % Add reference lines
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3, 'Label', 'Zero');
    yline(intercept, 'r--', 'LineWidth', 1, 'Alpha', 0.5, 'Label', 'Intercept');
    % Mark initial point
    if ~isempty(t)
        plot(t(1), x_t(1), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
        plot(t(end), x_t(end), 'go', 'MarkerSize', 8, 'MarkerFaceColor', 'g');
    end
    hold off;
    % Display signal properties
    fprintf('\n--- Ramp Signal Properties ---\n');
    fprintf('Signal Type: %s Ramp\n', signal_type);
    fprintf('Slope: %.4f\n', slope);
    fprintf('Intercept: %.4f\n', intercept);

    if ~isempty(t)
        fprintf('\nTime Range: %.4f to %.4f s (Duration: %.4f s)\n', ...
                t(1), t(end), t(end) - t(1));
        fprintf('Initial value at t=%.4f: %.4f\n', t(1), x_t(1));
        fprintf('Final value at t=%.4f: %.4f\n', t(end), x_t(end));
        fprintf('Total change: %.4f\n', x_t(end) - x_t(1));
        if slope ~= 0
            % Calculate when signal crosses zero
            t_zero = -intercept / slope;
            if t_zero >= t(1) && t_zero <= t(end)
                fprintf('\nZero crossing at t = %.4f s\n', t_zero);
            elseif t_zero < t(1)
                fprintf('\nZero crossing occurred before t = %.4f s (at t = %.4f s
                    )\n', t(1), t_zero);
            else
                fprintf('\nZero crossing will occur after t = %.4f s (at t = %.4f
                    s)\n', t(end), t_zero);
            end
        end
        % Average value over the time range
        avg_value = mean(x_t);
        fprintf('Average value: %.4f\n', avg_value);
    end
end
```

Listing 6: Ramp Signal Generator

```
--- Ramp Signal Properties ---
Signal Type: Increasing Ramp
Slope: 2.0000
Intercept: 0.0000
Time Range: -10.0000 to 10.0000 s (Duration: 20.0000 s)
Initial value at t=-10.0000: -20.0000
Final value at t=10.0000: 20.0000
Total change: 40.0000
Zero crossing at t = -0.0000 s
Average value: 0.0000
```

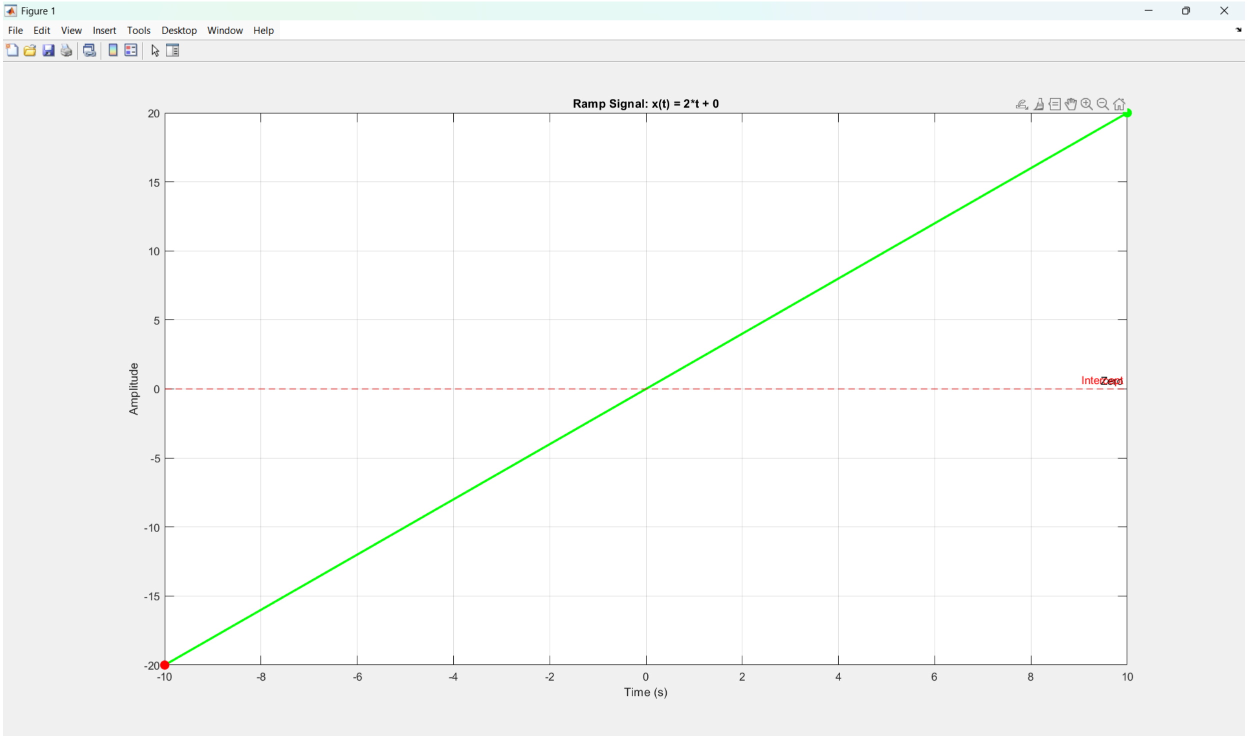


Figure 11: Ramp Signal Output

2.4.3 3. Polynomial Signal

```
function x_t = Polynomial(t)
    amplitude = input('The Amplitude = ');
    intercept = input('The Intercept = ');
    % Validate order
    order = input('The Order = ');
    while ~isnumeric(order) || order < 0 || order ~= floor(order)
        fprintf('    Error: Order must be a non-negative integer!\n');
        order = input('The Order = ');
    end
    % Display the polynomial formula
    fprintf('\nEnter coefficients for: x(t) = amplitude * (');
    for i = order:-1:0
        if i == order
            fprintf('a%d*t^%d', i, i);
        elseif i > 1
            fprintf(' + a%d*t^%d', i, i);
        elseif i == 1
            fprintf(' + a1*t');
        else  % i == 0
            fprintf(' + a0');
        end
    end
    fprintf(') + intercept\n\n');
    % Get coefficients
    coefficients = zeros(1, order + 1);
    for i = 1:(order + 1)
        power = order + 1 - i;
        if power == 0
            coefficients(i) = input(sprintf('Enter a%d (constant term): ', power))
                ;
        elseif power == 1
            coefficients(i) = input(sprintf('Enter a%d (coefficient for t): ',
                power));
        else
            coefficients(i) = input(sprintf('Enter a%d (coefficient for t^%d): ',
                power, power));
        end
    end
    % Calculate the polynomial signal
    x_t = amplitude * polyval(coefficients, t) + intercept;
    % Build dynamic title string (shortened for readability)
    if order <= 3
        % For low-order polynomials, show full formula
        titleStr = sprintf('Polynomial Signal (Order %d): x(t) = %g*(', order,
            amplitude);
        for i = 1:(order + 1)
            power = order + 1 - i;
            coeff = coefficients(i);
            if i == 1
                % First term
                if power == 0
                    titleStr = [titleStr, sprintf('%g', coeff)];
                elseif power == 1
                    titleStr = [titleStr, sprintf('%g*t', coeff)];
                else
                    titleStr = [titleStr, sprintf('%g*t^%d', coeff, power)];
                end
            else
                % Subsequent terms
                if power == 0
                    if coeff >= 0
                        titleStr = [titleStr, sprintf(' + %g', coeff)];
```

```
                    else
                        titleStr = [titleStr, sprintf(' - %g', abs(coeff))];
                    end
                elseif power == 1
                    if coeff >= 0
                        titleStr = [titleStr, sprintf(' + %g*t', coeff)];
                    else
                        titleStr = [titleStr, sprintf(' - %g*t', abs(coeff))];
                    end
                else
                    if coeff >= 0
                        titleStr = [titleStr, sprintf(' + %g*t^%d', coeff, power)];
                    else
                        titleStr = [titleStr, sprintf(' - %g*t^%d', abs(coeff), power)];
                    end
                end
            end
        end
        titleStr = [titleStr, sprintf(') + %g', intercept)];
    else
        % For high-order polynomials, use compact notation
        titleStr = sprintf('Polynomial Signal (Order %d): x(t) = %g*P_%d(t) + %g', ...
                          order, amplitude, order, intercept);
    end

    % Plot the signal
    figure;
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(titleStr);
    grid on;

    % Add zero line for reference
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3);
    if intercept ~= 0
        yline(intercept, 'r--', 'LineWidth', 1, 'Label', 'Intercept');
    end
    hold off;

    % Display polynomial properties
    fprintf('\n--- Polynomial Signal Properties ---\n');
    fprintf('Order: %d\n', order);
    fprintf('Amplitude multiplier: %.2f\n', amplitude);
    fprintf('Intercept (DC offset): %.2f\n', intercept);
    fprintf('Coefficients (highest to lowest power):\n');
    for i = 1:(order + 1)
        power = order + 1 - i;
        if power == 0
            fprintf('  a%d (constant): %.4f\n', power, coefficients(i));
        elseif power == 1
            fprintf('  a%d (linear): %.4f\n', power, coefficients(i));
        else
            fprintf('  a%d (t^%d): %.4f\n', power, power, coefficients(i));
        end
    end
    fprintf('Range of x(t): [%.4f, %.4f]\n', min(x_t), max(x_t));
    fprintf('Mean value: %.4f\n', mean(x_t));
end
```

Listing 7: Polynomial Signal Generator

```
--- Polynomial Signal Properties ---
Order: 3
Amplitude multiplier: 5.00
Intercept (DC offset): 2.00
Coefficients (highest to lowest power):
  a3 (t^3): 1.0000
  a2 (t^2): 1.0000
  a1 (linear): 1.0000
  a0 (constant): 0.0000
Range of x(t): [-4548.0000, 5552.0000]
Mean value: 168.6694
```

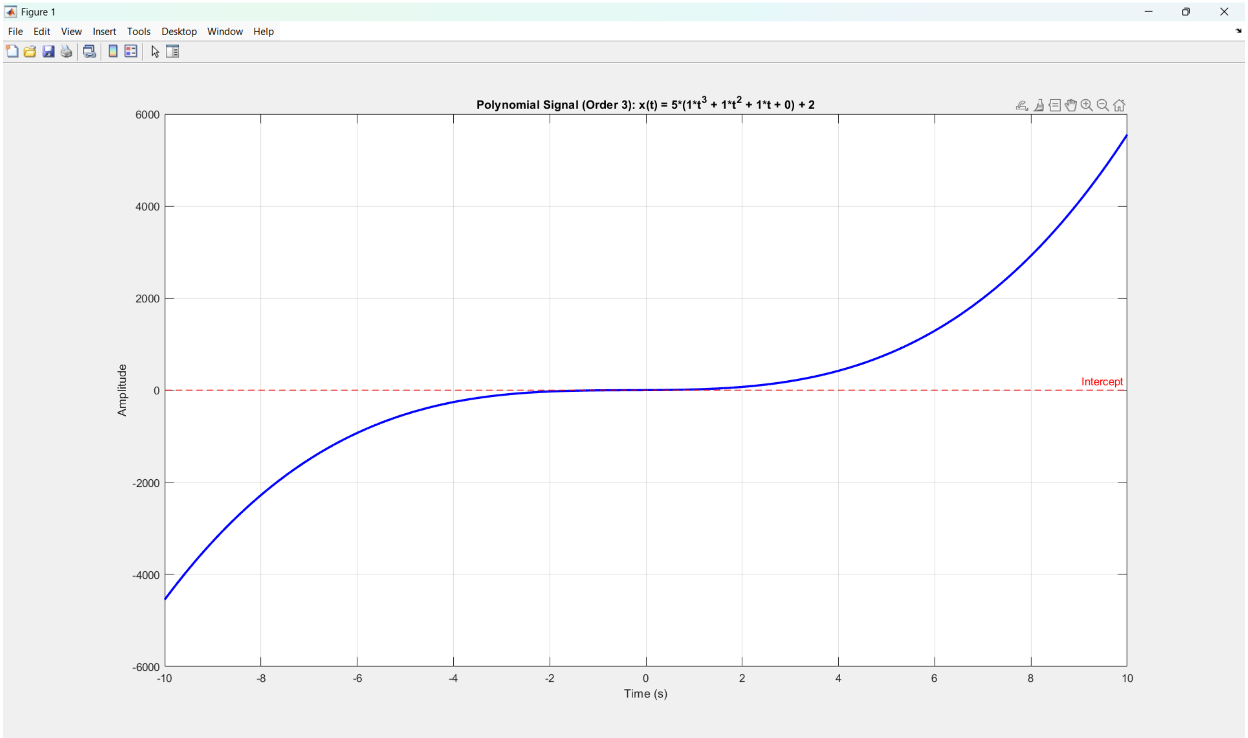


Figure 12: Polynomial Signal Output

2.4.4 4. Exponential Signal

```
function x_t = Exponential(t)
    amplitude = input('The Amplitude = ');
    exponent = input('The Exponent (decay rate if negative, growth rate if
       positive) = ');
    dc_offset = input('The DC Offset = ');
    % Generate exponential signal
    x_t = amplitude * exp(exponent * t) + dc_offset;
    % Determine signal type for display
    if exponent > 0
        signal_type = 'Growing';
    elseif exponent < 0
        signal_type = 'Decaying';
    else
        signal_type = 'Constant';
    end

    % Plot the signal
    figure;
    plot(t, x_t, 'm-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Exponential Signal: x(t) = %g*e^{%g*t} + %g', amplitude,
       exponent, dc_offset));
    grid on;

    % Add reference lines
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3);
    if dc_offset ~= 0
        yline(dc_offset, 'r--', 'LineWidth', 1, 'Label', 'DC Offset');
    end
    hold off;

    % Display signal properties
    fprintf('\n--- Exponential Signal Properties ---\n');
    fprintf('Signal Type: %s Exponential\n', signal_type);
    fprintf('Amplitude: %.2f\n', amplitude);
    fprintf('Exponent: %.4f\n', exponent);
    fprintf('DC Offset: %.2f\n', dc_offset);

    if exponent ~= 0
        time_constant = abs(1/exponent);
        fprintf('Time Constant (  ): %.4f s\n', time_constant);

        if exponent < 0
            fprintf('At t=  , signal decays to %.2f%% of initial value\n', 100/exp
               (1));
        else
            fprintf('At t=  , signal grows to %.2f%% more than initial value\n', (
               exp(1)-1)*100);
        end
    end

    % Display initial and final values (if within time range)
    if ~isempty(t)
        fprintf('\nInitial value at t=%.2f: %.4f\n', t(1), x_t(1));
        fprintf('Final value at t=%.2f: %.4f\n', t(end), x_t(end));
    end
end
```

Listing 8: Exponential Signal Generator

```
--- Exponential Signal Properties ---
Signal Type: Growing Exponential
Amplitude: 10.00
Exponent: 2.0000
DC Offset: 3.00
Time Constant (tau): 0.5000 s
At t=tau, signal grows to 171.83% more than initial value
Initial value at t=-10.00: 3.0000
Final value at t=10.00: 4851651957.0979
```

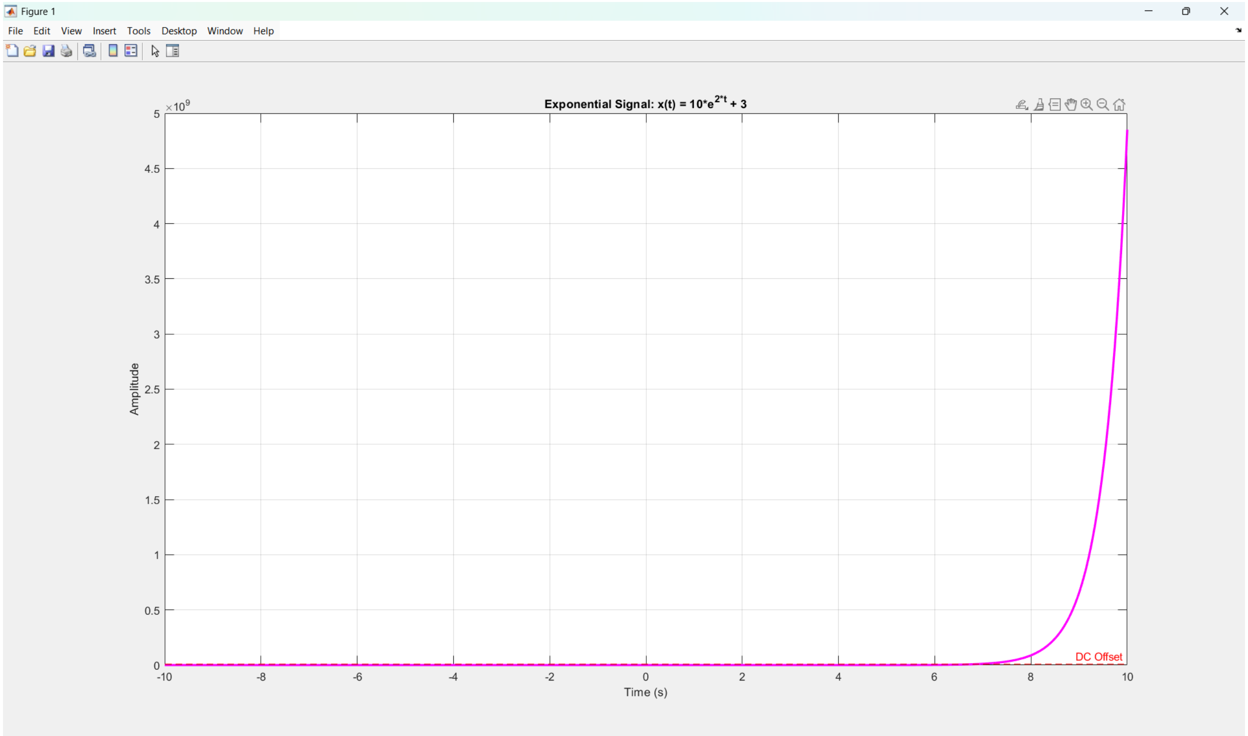


Figure 13: Exponential Signal Output

2.4.5 5. Sinusoidal Signal

```
function [x_t, t_new, F_s_new] = Sinusoidal(t,F_s , t_start, t_end)
    amplitude = input('The Amplitude = ');
    frequency = input('The Frequency (Hz) = ');
    phase = input('The Phase Shift (degrees) = ');
    dc_offset = input('The DC Offset = ');

    % Calculate actual sampling frequency from time vector
    F_s = 1 / (t(2) - t(1));

    % Validate Nyquist criterion
    nyquist_rate = 2 * frequency;
    samples_per_cycle = F_s / frequency;

    fprintf('\n--- Sampling Analysis ---\n');
    fprintf('Signal Frequency: %.2f Hz\n', frequency);
    fprintf('Sampling Frequency: %.2f Hz\n', F_s);
    fprintf('Nyquist Rate (minimum): %.2f Hz\n', nyquist_rate);
    fprintf('Samples per cycle: %.2f\n', samples_per_cycle);

    % Check if resampling is needed
    need_reconfig = false;

    if F_s < nyquist_rate
        warning('    ALIASING WARNING: Sampling frequency (%.2f Hz) is below Nyquist rate (%.2f Hz)!', F_s, nyquist_rate);
        fprintf('   The signal will be aliased and distorted!\n');
        fprintf('   Increase sampling frequency to at least %.2f Hz\n\n', nyquist_rate);
        need_reconfig = true;
    elseif samples_per_cycle < 10
        warning('    LOW QUALITY: Only %.1f samples per cycle.', samples_per_cycle);
        fprintf('   For better visualization, increase sampling frequency to %.2f Hz or higher.\n\n', frequency * 10);
        user_choice = input('\nContinue with current sampling rate? (y/n): ', 's');
        if ~strcmpi(user_choice, 'y')
            need_reconfig = true;
        end
    else
        fprintf('    Sampling frequency is adequate.\n\n');
    end

    % Reconfigure sampling if needed
    if need_reconfig
        fprintf('\n--- RECONFIGURE SAMPLING FREQUENCY ---\n');
        fprintf('Recommended minimum: %.2f Hz (Nyquist rate)\n', nyquist_rate);
        fprintf('Recommended for good quality: %.2f Hz (10 samples/cycle)\n', frequency * 10);
        fprintf('Recommended for excellent quality: %.2f Hz (100 samples/cycle)\n', frequency * 100);

        F_s_new = input('Enter new Sampling Frequency (Hz) = ');

        % Validate new sampling frequency
        while F_s_new <= 0 || F_s_new < nyquist_rate
            if F_s_new <= 0
                fprintf('    Error: Sampling frequency must be positive!\n');
            else
                fprintf('    Error: Still below Nyquist rate (%.2f Hz)!\n', nyquist_rate);
            end
```

```
            F_s_new = input('Enter new Sampling Frequency (Hz) = ');
        end

        % Recalculate time vector with new sampling frequency
        T_s_new = 1 / F_s_new;
        t_new = t_start:T_s_new:t_end;

        fprintf('\ n   Sampling frequency updated successfully!\n');
        fprintf('New samples per cycle: %.2f\n', F_s_new / frequency);
        fprintf('New number of samples: %d\n\n', length(t_new));
    else
        t_new = t;
        F_s_new = F_s;
    end


    % Get sinusoidal type
    type = get_sinusoidal_signal();

    % Convert phase from degrees to radians
    phase_rad = phase * pi / 180;

    % Generate signal based on type
    switch type
        case 1
            x_t = amplitude * sin(2 * pi * frequency * t_new + phase_rad) +
                dc_offset;
            signal_name = 'Sine';
            fprintf('\nSine Signal Has Been Chosen\n');
        case 2
             x_t = amplitude * cos(2 * pi * frequency * t_new + phase_rad) +
                 dc_offset;
            signal_name = 'Cosine';
            fprintf('\nCosine Signal Has Been Chosen\n');
    end

    % Create dynamic title based on signal type
    if type == 1
        titleStr = sprintf('Sinusoidal Signal: x(t) = %g*sin(2  *%g*t + %g  ) + %g
            ', ...
                           amplitude, frequency, phase, dc_offset);
    else
        titleStr = sprintf('Sinusoidal Signal: x(t) = %g*cos(2  *%g*t + %g  ) + %g
            ', ...
                           amplitude, frequency, phase, dc_offset);
    end

    % Plot the signal
    figure;
    plot(t_new, x_t, 'c-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(titleStr);
    grid on;
    ylim([dc_offset - amplitude - 0.5, dc_offset + amplitude + 0.5]);

    % Add zero line and DC offset line for reference
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3);
    if dc_offset ~= 0
        yline(dc_offset, 'r--', 'LineWidth', 1, 'Label', 'DC Offset');
    end
    hold off;

```

```
    % Display signal properties
    fprintf('\n--- %s Wave Properties ---\n', signal_name);
    fprintf('Amplitude: %.2f\n', amplitude);
    fprintf('Frequency: %.2f Hz\n', frequency);
    fprintf('Period: %.4f s\n', 1/frequency);
    fprintf('Phase Shift: %.2f degrees (%.4f radians)\n', phase, phase_rad);
    fprintf('DC Offset: %.2f\n', dc_offset);
    fprintf('Peak-to-Peak: %.2f\n', 2*amplitude);
    fprintf('Actual Sampling Frequency: %.2f Hz\n', F_s_new);
    fprintf('Actual Samples per Cycle: %.2f\n', F_s_new / frequency);
end
```

Listing 9: Sinusoidal Signal Generator

```
function type = get_sinusoidal_signal()
    menu_text = sprintf(['Choose sinusoidal type:\n' ...
                         '1. Sine wave\n' ...
                         '2. Cosine wave\n' ...
                         '\nEnter your choice (1-2): ']);
    valid_choices = [1, 2];

    while true
        type = input(menu_text);
        if isnumeric(type) && isscalar(type) && ismember(type, valid_choices)
            break;
        else
            fprintf('Invalid choice! Please enter 1 for sine or 2 for cosine.\n\n');
        end
    end
end
```

Listing 10: Sinusoidal Signal Choice

```
--- Sine Wave Properties ---
Amplitude: 10.00
Frequency: 20.00 Hz
Period: 0.0500 s
Phase Shift: 0.00 degrees (0.0000 radians)
DC Offset: 5.00
Peak-to-Peak: 20.00
Sampling Frequency: 6000.00 Hz
Samples per Cycle: 300.00
```

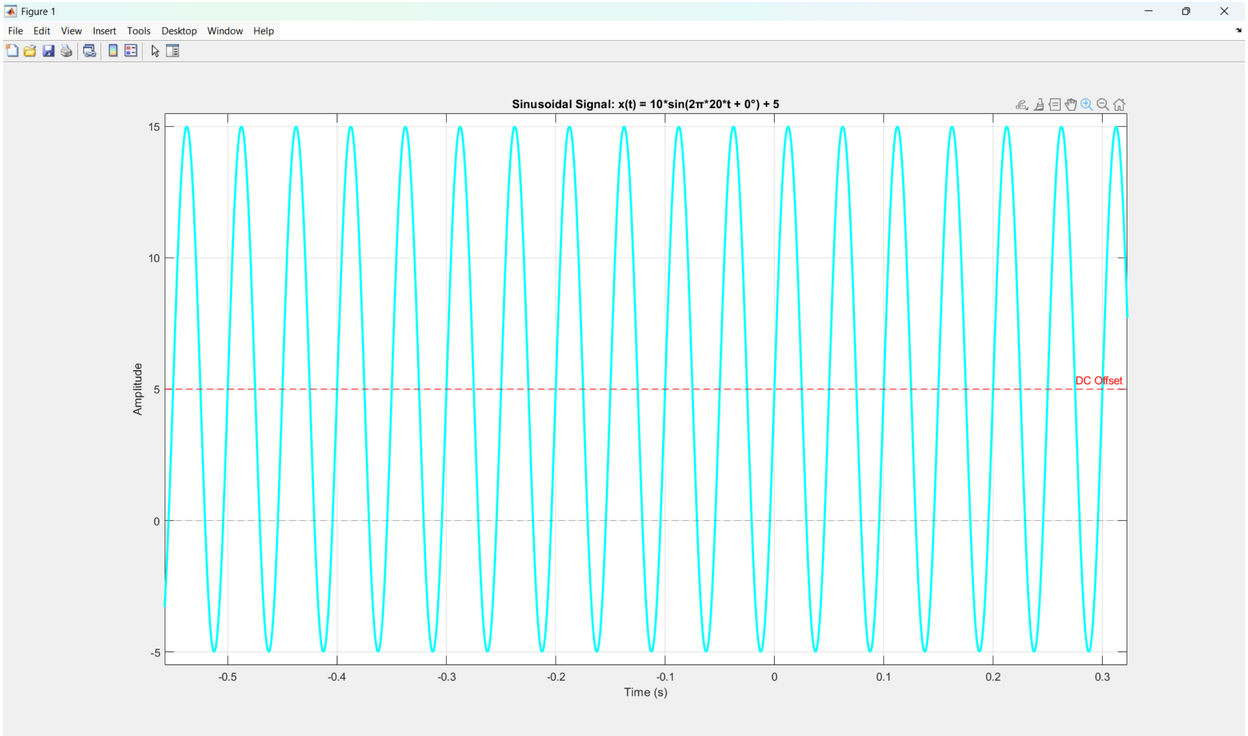


Figure 14: Sine Wave Output

```
--- Cosine Wave Properties ---
Amplitude: 10.00
Frequency: 20.00 Hz
Period: 0.0500 s
Phase Shift: 0.00 degrees (0.0000 radians)
DC Offset: 2.00
Peak-to-Peak: 20.00
Sampling Frequency: 6000.00 Hz
Samples per Cycle: 300.00
```

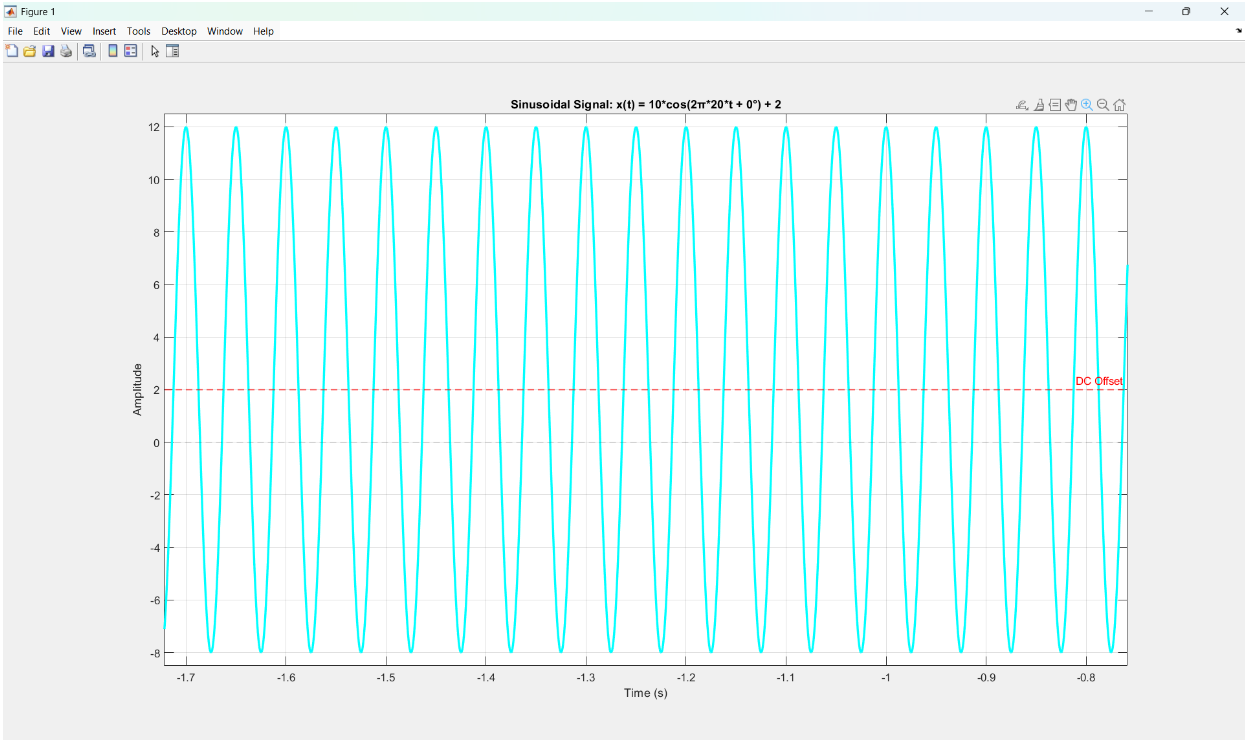


Figure 15: Cosine Wave Output

2.4.6 6. Gaussian Pulse

```
function x_t = Gaussian_Pulse(t)
    amplitude = input('The Amplitude = ');
    center = input('The Center Position (mean) = ');
    width = input('The Width (standard deviation   ) = ');
    dc_offset = input('The DC Offset = ');

    % Generate Gaussian pulse signal
    x_t = amplitude * exp(-(t - center).^2 / (2 * width^2)) + dc_offset;

    % Calculate key pulse characteristics
    fwhm = 2 * sqrt(2 * log(2)) * width;  % Full Width at Half Maximum
    half_max = amplitude / 2 + dc_offset;

    % Find pulse boundaries at half maximum
    if ~isempty(t)
        idx_peak = find(abs(t - center) == min(abs(t - center)), 1);
        left_half = find(x_t(1:idx_peak) >= half_max, 1, 'first');
        right_half = find(x_t(idx_peak:end) >= half_max, 1, 'last') + idx_peak - 1;
    end

    % Plot the signal
    figure;
    plot(t, x_t, 'k-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Gaussian Pulse: x(t) = %g*e^{-(t-%g)^2/(2*%g^2)} + %g', ...
                  amplitude, center, width, dc_offset));
    grid on;

    % Add reference lines and markers
    hold on;
    yline(0, 'k--', 'LineWidth', 0.5, 'Alpha', 0.3);
    if dc_offset ~= 0
        yline(dc_offset, 'r--', 'LineWidth', 1, 'Alpha', 0.5, 'Label', 'DC Offset');
    end

    % Mark peak
    plot(center, amplitude + dc_offset, 'ro', 'MarkerSize', 10, 'MarkerFaceColor', 'r');

    % Mark FWHM points
    yline(half_max, 'b--', 'LineWidth', 1, 'Alpha', 0.5, 'Label', 'Half Maximum');

    % Mark     ,  2  ,  3   boundaries
    for i = 1:3
        xline(center - i*width, 'g:', 'LineWidth', 1, 'Alpha', 0.3);
        xline(center + i*width, 'g:', 'LineWidth', 1, 'Alpha', 0.3);
    end
    hold off;

    % Display signal properties
    fprintf('\n--- Gaussian Pulse Properties ---\n');
    fprintf('Amplitude: %.4f\n', amplitude);
    fprintf('Center Position (  ): %.4f s\n', center);
    fprintf('Width (  ): %.4f s\n', width);
    fprintf('DC Offset: %.4f\n', dc_offset);
    fprintf('\nPulse Characteristics:\n');
    fprintf('Peak Value: %.4f (at t = %.4f s)\n', amplitude + dc_offset, center);
    fprintf('FWHM (Full Width at Half Max): %.4f s\n', fwhm);
    fprintf('Half Maximum Level: %.4f\n', half_max);
```

```

    % Percentage of energy contained within   n
    fprintf('\nEnergy Distribution:\n');
    fprintf('Within  1  (%.4f to %.4f s): 68.27%% of energy\n', ...
            center - width, center + width);
    fprintf('Within  2  (%.4f to %.4f s): 95.45%% of energy\n', ...
            center - 2*width, center + 2*width);
    fprintf('Within  3  (%.4f to %.4f s): 99.73%% of energy\n', ...
            center - 3*width, center + 3*width);

    % Bandwidth (approximate for Gaussian pulse)
    bandwidth = 1 / (2 * pi * width);
    fprintf('\nApproximate Bandwidth: %.4f Hz\n', bandwidth);

    if ~isempty(t)
        fprintf('\nTime Range: %.4f to %.4f s\n', t(1), t(end));

        % Calculate pulse energy in the visible range
        dt = mean(diff(t));
        energy = sum(x_t.^2) * dt;
        fprintf('Signal Energy (in range): %.4f\n', energy);
    end
end
```

Listing 11: Gaussian Pulse Generator

```
--- Gaussian Pulse Properties ---
Amplitude: 20.0000, Center Position (mu): 5.0000 s
Width (sigma): 10.0000 s, DC Offset: 0.0000
Peak Value: 20.0000 (at t = 5.0000 s)
FWHM (Full Width at Half Max): 23.5482 s
Energy Distribution:
Within ±1sigma (-5.0000 to 15.0000 s): 68.27% of energy
Within ±2sigma (-15.0000 to 25.0000 s): 95.45% of energy
Within ±3sigma (-25.0000 to 35.0000 s): 99.73% of energy
Approximate Bandwidth: 0.0159 Hz
```

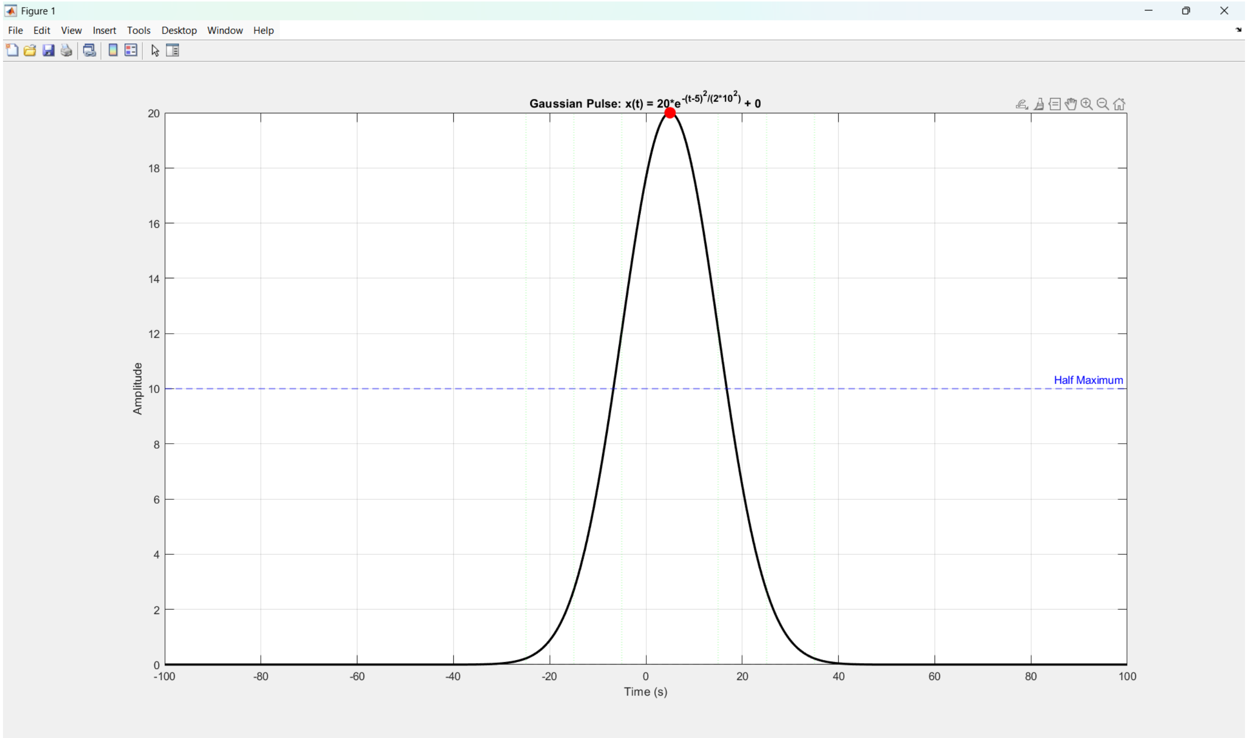


Figure 16: Gaussian Pulse Output

2.4.7 7. Sawtooth Wave

```
function [x_t, t_new, F_s_new] = Sawtooth_Wave(t, F_s, t_start, t_end)
    amplitude = input('The Amplitude = ');
    frequency = input('The Frequency (Hz) = ');
    phase = input('The Phase Shift (degrees) = ');
    dc_offset = input('The DC Offset = ');

    % Convert phase from degrees to radians
    phase_rad = phase * pi / 180;

    % Calculate sampling metrics
    nyquist_rate = 2 * frequency;  % Minimum for fundamental only
    samples_per_cycle = F_s / frequency;

    % For sawtooth, we need many more samples due to harmonics
    recommended_min = frequency * 50;   % 50 samples per cycle
    recommended_good = frequency * 100;  % 100 samples per cycle

    fprintf('\n--- Sampling Analysis for Sawtooth Wave ---\n');
    fprintf('Signal Frequency (fundamental): %.2f Hz\n', frequency);
    fprintf('Sampling Frequency: %.2f Hz\n', F_s);
    fprintf('Nyquist Rate (fundamental only): %.2f Hz\n', nyquist_rate);
    fprintf('Samples per cycle: %.2f\n', samples_per_cycle);

    % Determine sampling adequacy
    need_reconfig = false;

    if F_s < nyquist_rate
        warning('    CRITICAL: Below Nyquist rate for fundamental frequency!\n');
        fprintf('   Severe aliasing will occur.\n');
        need_reconfig = true;
    elseif samples_per_cycle < 20
        fprintf('    VERY POOR: Only %.1f samples/cycle.\n', samples_per_cycle);
        fprintf('   Sawtooth shape will be completely lost.\n');
        fprintf('   Signal will look like a triangle or sinusoid.\n');
        need_reconfig = true;
    elseif samples_per_cycle < 50
        fprintf('    POOR QUALITY: %.1f samples/cycle.\n', samples_per_cycle);
        fprintf('   Sawtooth will appear very blocky and distorted.\n');
        fprintf('   High-frequency harmonics will be lost.\n');

        user_choice = input('\nContinue with current sampling rate? (y/n): ', 's')
            ;
        if ~strcmpi(user_choice, 'y')
            need_reconfig = true;
        end
    elseif samples_per_cycle < 100
        fprintf('    ACCEPTABLE: %.1f samples/cycle.\n', samples_per_cycle);
        fprintf('   Sawtooth shape visible but somewhat blocky.\n');
        fprintf('   Consider increasing for smoother appearance.\n');

        user_choice = input('\nContinue with current sampling rate? (y/n): ', 's')
            ;
        if ~strcmpi(user_choice, 'y')
            need_reconfig = true;
        end
    else
        fprintf('       Good sampling frequency (%.1f samples/cycle).\n',
            samples_per_cycle);
    end

    % Reconfigure if needed
    if need_reconfig
```

```
        fprintf('\n--- RECONFIGURE SAMPLING FREQUENCY ---\n');
        fprintf('    NOTE: Sawtooth waves contain many high-frequency harmonics!\n');
        fprintf('   They require MUCH higher sampling rates than sinusoids.\n\n');
        fprintf('Absolute minimum: %.2f Hz (Nyquist for fundamental)\n', nyquist_rate);
        fprintf('Minimum acceptable: %.2f Hz (20 samples/cycle)\n', frequency * 20);
        fprintf('Recommended minimum: %.2f Hz (50 samples/cycle)\n', recommended_min);
        fprintf('Good quality: %.2f Hz (100 samples/cycle)\n', recommended_good);
        fprintf('Excellent quality: %.2f Hz (500 samples/cycle)\n\n', frequency * 500);

        F_s_new = input('Enter new Sampling Frequency (Hz) = ');

        % Validate new sampling frequency
        while F_s_new <= 0 || F_s_new < nyquist_rate
            if F_s_new <= 0
                fprintf('    Error: Sampling frequency must be positive!\n');
            else
                fprintf('    Error: Still below Nyquist rate (%.2f Hz)!\n', nyquist_rate);
            end
            F_s_new = input('Enter new Sampling Frequency (Hz) = ');
        end

        % Warn if still low quality
        new_samples_per_cycle = F_s_new / frequency;
        if new_samples_per_cycle < 50
            fprintf('\ n   WARNING: %.1f samples/cycle is still low for sawtooth.\n', new_samples_per_cycle);
            fprintf('   Signal quality will be limited.\n');
        end

        % Recalculate time vector
        T_s_new = 1 / F_s_new;
        t_new = t_start:T_s_new:t_end;

        fprintf('\ n   Sampling frequency updated!\n');
        fprintf('New samples per cycle: %.2f\n', new_samples_per_cycle);
        fprintf('New number of samples: %d\n\n', length(t_new));
    else
        t_new = t;
        F_s_new = F_s;
    end

    % Generate sawtooth wave
    x_t = amplitude * sawtooth(2 * pi * frequency * t_new + phase_rad) + dc_offset;

    % Plot the signal
    figure;
    plot(t_new, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Sawtooth Wave: A=%g, f=%g Hz, \\phi=%g , DC=%g', ...
                  amplitude, frequency, phase, dc_offset));
    grid on;

    % Add zero line for reference
    hold on;
    yline(dc_offset, 'r--', 'LineWidth', 1, 'Label', 'DC Offset');
    hold off;
```

```

    % Display signal properties
    fprintf('\n--- Sawtooth Wave Properties ---\n');
    fprintf('Amplitude: %.2f\n', amplitude);
    fprintf('Frequency: %.2f Hz\n', frequency);
    fprintf('Period: %.4f s\n', 1/frequency);
    fprintf('Phase Shift: %.2f degrees (%.4f radians)\n', phase, phase_rad);
    fprintf('DC Offset: %.2f\n', dc_offset);
    fprintf('Peak-to-Peak: %.2f\n', 2*amplitude);
    fprintf('Actual Sampling Frequency: %.2f Hz\n', F_s_new);
    fprintf('Actual Samples per Cycle: %.2f\n', F_s_new / frequency);

    % Display harmonic information
    fprintf('\n--- Harmonic Content ---\n');
    fprintf('A sawtooth wave contains infinite harmonics at:\n');
    fprintf('  f, 2f, 3f, 4f, ...\n');
    fprintf('With current sampling, harmonics up to %.0f Hz can be represented.\n'
       , F_s_new/2);
    fprintf('This includes approximately the first %.0f harmonics.\n', floor((
       F_s_new/2)/frequency));
end
```

Listing 12: Sawtooth Wave Generator

```
--- Sawtooth Wave Properties ---
Amplitude: 20.00
Frequency: 1.00 Hz
Period: 1.0000 s
Phase Shift: 0.00 degrees (0.0000 radians)
DC Offset: 4.00
Peak-to-Peak: 40.00
Sampling Frequency: 6000.00 Hz
Samples per Cycle: 6000.00
Harmonic Content: Contains infinite harmonics at f, 2f, 3f, 4f...
With current sampling, harmonics up to 3000 Hz can be represented.
This includes approximately the first 3000 harmonics.
```

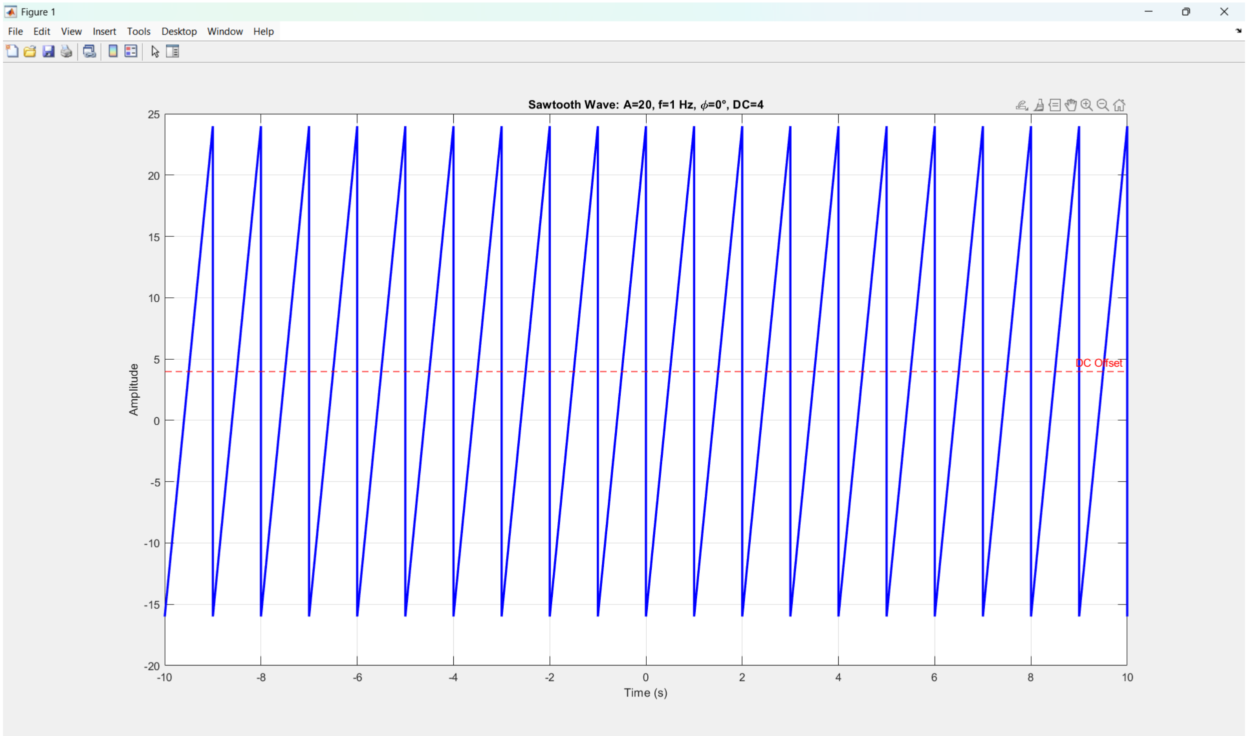


Figure 17: Sawtooth Wave Output

2.5 Combined Signals with Breakpoints

The program can combine multiple signal types at specified breakpoints to create complex piecewise signals.

```
function [x_combined, t_combined] = combine_signals(t_start, t_end, F_s)
% COMBINE_SIGNALS - Combine multiple signals at specified breakpoints
%
% Syntax: [x_combined, t_combined] = combine_signals(t_start, t_end, F_s)
%
% Inputs:
%   t_start - Start time
%   t_end - End time
%   F_s - Sampling frequency
%
% Outputs:
%   x_combined - Combined signal
%   t_combined - Time vector for combined signal

    fprintf('\n');
    fprintf('======================================================================\n');
    fprintf('                    SIGNAL COMBINATION MODULE\n');
    fprintf('======================================================================\n');

    % Get number of segments
    fprintf('\nHow many signal segments do you want to combine?\n');
    num_segments = input('Enter number of segments (minimum 2): ');

    % Validate number of segments
    while ~isnumeric(num_segments) || num_segments < 2 || num_segments ~= floor(num_segments)
        fprintf('   Invalid input! Please enter an integer     2.\n');
        num_segments = input('Enter number of segments: ');
    end

    % Get breakpoints
    breakpoints = get_breakpoints(t_start, t_end, num_segments);

    % Display segment information
    fprintf('\n--- SEGMENT BREAKDOWN ---\n');
    for i = 1:num_segments
        fprintf('Segment %d: [%.4f, %.4f] s (Duration: %.4f s)\n', ...
                i, breakpoints(i), breakpoints(i+1), ...
                breakpoints(i+1) - breakpoints(i));
    end
    fprintf('======================================================================\n');

    % Generate each segment
    segments = cell(num_segments, 1);

    for i = 1:num_segments
        fprintf('\n');
        fprintf('
            \n');
        fprintf('                    SEGMENT %d of %d\n', i, num_segments);
        fprintf('                    Time Range: [%.4f, %.4f] s\n', ...
                breakpoints(i), breakpoints(i+1));
        fprintf('
```

```
            \n');

        % Create time vector for this segment
        T_s = 1 / F_s;
        t_segment = breakpoints(i):T_s:breakpoints(i+1);

        % Get signal choice for this segment
        choice = get_signal_choice();

        fprintf('\  n      Generating signal for Segment %d...\n\n', i);

        % Generate signal based on choice
        switch choice
            case 1
                x_segment = DC(t_segment);
            case 2
                x_segment = Ramp(t_segment);
            case 3
                x_segment = Exponential(t_segment);
            case 4
                [x_segment, t_segment, ~] = Sinusoidal(t_segment, F_s, breakpoints
                    (i), breakpoints(i+1));
            case 5
                x_segment = Gaussian_Pulse(t_segment);
            case 6
                [x_segment, t_segment, ~] = Sawtooth_Wave(t_segment, F_s,
                    breakpoints(i), breakpoints(i+1));
            case 7
                x_segment = Polynomial(t_segment);
        end

        % Store segment
        segments{i}.signal = x_segment;
        segments{i}.time = t_segment;
        segments{i}.type = choice;
        segments{i}.start = breakpoints(i);
        segments{i}.end = breakpoints(i+1);

        fprintf('\ n   Segment %d completed!\n', i);
    end

    % Combine all segments
    fprintf('\n');
    fprintf('
        ======================================================================\
        n');
    fprintf('                       COMBINING SEGMENTS\n');
    fprintf('
        ======================================================================\
        n');

    [x_combined, t_combined] = merge_segments(segments);

    % Plot combined signal
    plot_combined_signal(segments, x_combined, t_combined, breakpoints);

    % Display statistics
    display_combination_summary(segments, x_combined, t_combined, breakpoints);

    fprintf('\n
        ======================================================================\
        n');
    fprintf('    Signal combination completed successfully!\n');
```

```
    fprintf('============================================================================\n\n');
end
```

Listing 13: Sawtooth Wave Generator

```
--- SEGMENT BREAKDOWN ---
Segment 1: [-10.0000, -8.0000] s - DC Signal (Amplitude: 10.0000)
Segment 2: [-8.0000, -7.0000] s - Ramp Signal (Slope: 9.0000)
Segment 3: [-7.0000, -5.0000] s - Sine Wave (Amp: 5.00, f: 1.00 Hz)
Segment 4: [-5.0000, 0.0000] s - Sawtooth Wave (Amp: 5.00, f: 1.00 Hz)
Segment 5: [0.0000, 2.0000] s - Gaussian Pulse (Amp: 7.0000, mu: 4.0000)
Segment 6: [2.0000, 6.0000] s - Polynomial (Order: 2, Amp: 4.00)
Segment 7: [6.0000, 10.0000] s - Exponential (Amp: 5.00, exp: -3.0000)
```

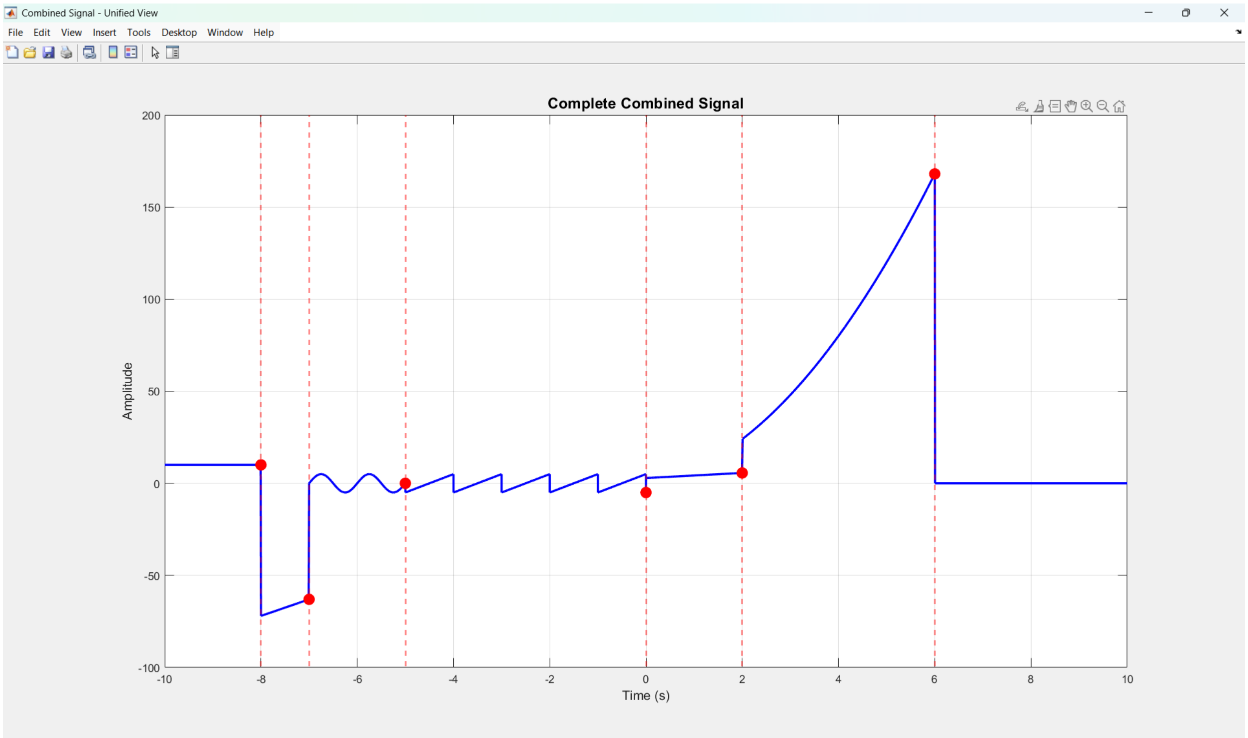


Figure 18: Combined Signal - Individual Segments

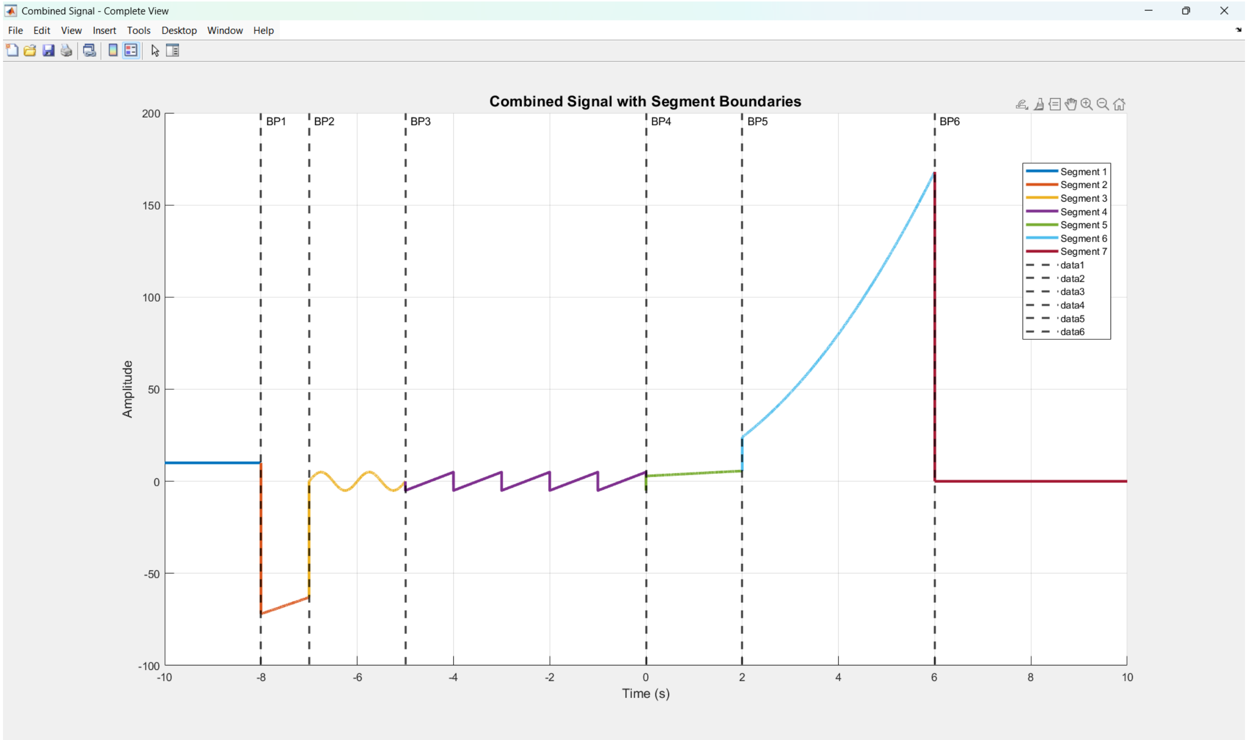


Figure 19: Combined Signal - Complete View with Color-Coded Segments

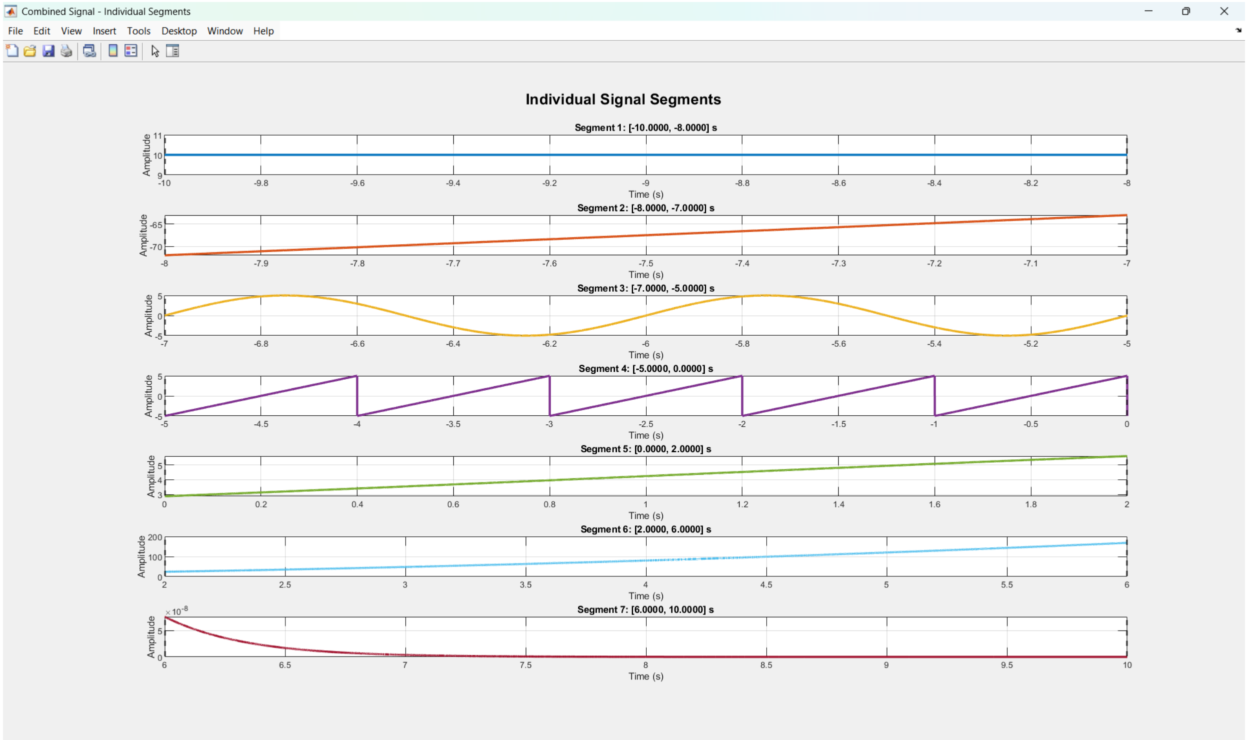


Figure 20: Combined Signal - Unified View

2.6 Random Signal Generation

The program can automatically generate multiple signals with random parameters.

```
function signals = Random(t, num_signals)
% RANDOM - Generate multiple random signals with random parameters
%
% Syntax: signals = Random(t, num_signals)
%
% Inputs:
%   t - Time vector
%   num_signals - Number of random signals to generate
%
% Outputs:
%   signals - Cell array containing generated signals and their info

    % Initialize cell array to store signals
    signals = cell(num_signals, 1);

    % Signal type names
    signal_names = {
        'DC Signal', ...
        'Ramp Signal', ...
        'Exponential Signal', ...
        'Sinusoidal Signal', ...
        'Gaussian Pulse', ...
        'Sawtooth Wave', ...
        'Polynomial Signal'
    };

    fprintf('\n========================================================================\n');
    fprintf('                  GENERATING %d RANDOM SIGNALS\n', num_signals);
    fprintf('========================================================================\n\n');

    % Generate each random signal
    for i = 1:num_signals
        fprintf('--- Generating Signal %d/%d ---\n', i, num_signals);

        % Randomly select signal type (1-7)
        signal_type = randi([1, 7]);
        fprintf('Signal Type: %s\n', signal_names{signal_type});

        % Generate signal based on type
        switch signal_type
            case 1
                [x_t, params] = generate_random_DC(t);
            case 2
                [x_t, params] = generate_random_Ramp(t);
            case 3
                [x_t, params] = generate_random_Exponential(t);
            case 4
                [x_t, params] = generate_random_Sinusoidal(t);
            case 5
                [x_t, params] = generate_random_Gaussian(t);
            case 6
                [x_t, params] = generate_random_Sawtooth(t);
            case 7
                [x_t, params] = generate_random_Polynomial(t);
        end

        % Store signal data
```

```
        signals{i}.type = signal_type;
        signals{i}.type_name = signal_names{signal_type};
        signals{i}.data = x_t;
        signals{i}.params = params;
        signals{i}.time = t;

        fprintf('\n');
    end

    % Plot all signals together
    plot_all_random_signals(signals);

    % Display summary
    display_random_signals_summary(signals);

    fprintf('
       ======================================================================\
       n');
    fprintf('    Successfully generated %d random signals!\n', num_signals);
    fprintf('
       ======================================================================\
       n\n');
end
```

Listing 14: Random

```
function [x_t, params] = generate_random_DC(t)
% GENERATE_RANDOM_DC - Generate random DC signal

    % Random amplitude between -10 and 10
    amplitude = (rand() * 20) - 10;

    % Generate DC signal
    x_t = amplitude * ones(size(t));

    % Store parameters
    params.amplitude = amplitude;

    % Display parameters
    fprintf('  Amplitude: %.4f\n', amplitude);
end
```

Listing 15: Random DC Generator

```
function [x_t, params] = generate_random_Sinusoidal(t)
% GENERATE_RANDOM_SINUSOIDAL - Generate random sinusoidal signal

    % Random amplitude between 1 and 10
    amplitude = rand() * 9 + 1;

    % Random frequency between 0.1 and 5 Hz
    frequency = rand() * 4.9 + 0.1;

    % Random phase between 0 and 360 degrees
    phase = rand() * 360;

    % Random DC offset between -5 and 5
    dc_offset = (rand() * 10) - 5;

    % Randomly choose sine or cosine
    type = randi([1, 2]);

    % Convert phase to radians
    phase_rad = phase * pi / 180;
```

```

    % Generate signal
    if type == 1
        x_t = amplitude * sin(2 * pi * frequency * t + phase_rad) + dc_offset;
        signal_name = 'Sine';
    else
        x_t = amplitude * cos(2 * pi * frequency * t + phase_rad) + dc_offset;
        signal_name = 'Cosine';
    end

    % Store parameters
    params.amplitude = amplitude;
    params.frequency = frequency;
    params.phase = phase;
    params.dc_offset = dc_offset;
    params.type = type;
    params.signal_name = signal_name;

    % Display parameters
    fprintf('  Type: %s\n', signal_name);
    fprintf('  Amplitude: %.4f\n', amplitude);
    fprintf('  Frequency: %.4f Hz\n', frequency);
    fprintf('  Phase: %.4f degrees\n', phase);
    fprintf('  DC Offset: %.4f\n', dc_offset);
end
```

Listing 16: Random Sinusoidal Generator

```
function [x_t, params] = generate_random_Exponential(t)
% GENERATE_RANDOM_EXPONENTIAL - Generate random exponential signal

    % Random amplitude between 0.5 and 5
    amplitude = rand() * 4.5 + 0.5;

    % Random exponent between -2 and 2 (excluding very small values near 0)
    exponent = (rand() * 4) - 2;
    if abs(exponent) < 0.1
        exponent = sign(exponent) * 0.1;
    end

    % Random DC offset between -5 and 5
    dc_offset = (rand() * 10) - 5;

    % Generate exponential signal
    x_t = amplitude * exp(exponent * t) + dc_offset;

    % Store parameters
    params.amplitude = amplitude;
    params.exponent = exponent;
    params.dc_offset = dc_offset;

    % Display parameters
    fprintf('  Amplitude: %.4f\n', amplitude);
    fprintf('  Exponent: %.4f\n', exponent);
    fprintf('  DC Offset: %.4f\n', dc_offset);
end
```

Listing 17: Random Exponential Generator

```
function [x_t, params] = generate_random_Gaussian(t)
% GENERATE_RANDOM_GAUSSIAN - Generate random Gaussian pulse

    % Random amplitude between 1 and 10
    amplitude = rand() * 9 + 1;
```

```

    % Random center within the time range
    t_range = t(end) - t(1);
    center = t(1) + rand() * t_range;

    % Random width between 5% and 20% of time range
    width = (rand() * 0.15 + 0.05) * t_range;

    % Random DC offset between -2 and 2
    dc_offset = (rand() * 4) - 2;

    % Generate Gaussian pulse
    x_t = amplitude * exp(-(t - center).^2 / (2 * width^2)) + dc_offset;

    % Store parameters
    params.amplitude = amplitude;
    params.center = center;
    params.width = width;
    params.dc_offset = dc_offset;

    % Display parameters
    fprintf('  Amplitude: %.4f\n', amplitude);
    fprintf('  Center: %.4f s\n', center);
    fprintf('  Width (  ): %.4f s\n', width);
    fprintf('  DC Offset: %.4f\n', dc_offset);
end
```

Listing 18: Random Gaussian Generator

```
function [x_t, params] = generate_random_Sawtooth(t)
% GENERATE_RANDOM_SAWTOOTH - Generate random sawtooth wave

    % Random amplitude between 1 and 10
    amplitude = rand() * 9 + 1;

    % Random frequency between 0.1 and 5 Hz
    frequency = rand() * 4.9 + 0.1;

    % Random phase between 0 and 360 degrees
    phase = rand() * 360;

    % Random DC offset between -5 and 5
    dc_offset = (rand() * 10) - 5;

    % Convert phase to radians
    phase_rad = phase * pi / 180;

    % Generate sawtooth wave
    x_t = amplitude * sawtooth(2 * pi * frequency * t + phase_rad) + dc_offset;

    % Store parameters
    params.amplitude = amplitude;
    params.frequency = frequency;
    params.phase = phase;
    params.dc_offset = dc_offset;

    % Display parameters
    fprintf('  Amplitude: %.4f\n', amplitude);
    fprintf('  Frequency: %.4f Hz\n', frequency);
    fprintf('  Phase: %.4f degrees\n', phase);
    fprintf('  DC Offset: %.4f\n', dc_offset);
end
```

Listing 19: Random Sawtooth Wave Generator

```
function [x_t, params] = generate_random_Polynomial(t)
% GENERATE_RANDOM_POLYNOMIAL - Generate random polynomial signal

    % Random amplitude between 0.1 and 2
    amplitude = rand() * 1.9 + 0.1;

    % Random intercept between -10 and 10
    intercept = (rand() * 20) - 10;

    % Random order between 2 and 4
    order = randi([2, 4]);

    % Generate random coefficients
    coefficients = zeros(1, order + 1);
    for i = 1:(order + 1)
        % Scale coefficients to avoid extreme values
        power = order + 1 - i;
        scale_factor = 10 / (power + 1);  % Higher powers get smaller coefficients
        coefficients(i) = (rand() * 2 - 1) * scale_factor;
    end

    % Generate polynomial signal
    x_t = amplitude * polyval(coefficients, t) + intercept;

    % Store parameters
    params.amplitude = amplitude;
    params.intercept = intercept;
    params.order = order;
    params.coefficients = coefficients;

    % Display parameters
    fprintf('  Amplitude: %.4f\n', amplitude);
    fprintf('  Intercept: %.4f\n', intercept);
    fprintf('  Order: %d\n', order);
    fprintf('  Coefficients: ');
    fprintf('%.4f ', coefficients);
    fprintf('\n');
end
```

Listing 20: Random Polynomial Generator

```
function [x_t, params] = generate_random_Ramp(t)
% GENERATE_RANDOM_RAMP - Generate random ramp signal

    % Random slope between -5 and 5
    slope = (rand() * 10) - 5;

    % Random intercept between -10 and 10
    intercept = (rand() * 20) - 10;

    % Generate ramp signal
    x_t = slope * t + intercept;

    % Store parameters
    params.slope = slope;
    params.intercept = intercept;

    % Display parameters
    fprintf('  Slope: %.4f\n', slope);
    fprintf('  Intercept: %.4f\n', intercept);
end
```

Listing 21: Random Ramp Generator

```
--- RANDOM SIGNALS SUMMARY ---
Signal 1: Sawtooth Wave (Min: -7.8284, Max: 10.4757, Mean: 1.3845)
Signal 2: DC Signal (Min: -4.4300, Max: -4.4300, Mean: -4.4300)
Signal 3: Sinusoidal Signal (Min: -4.9116, Max: 14.3235, Mean: 4.7230)
Signal 4: Sinusoidal Signal (Min: -4.0452, Max: 12.3599, Mean: 4.1999)
Signal 5: Polynomial Signal (Min: -13.2771, Max: 24819.1919)
Signal 6: Gaussian Pulse (Min: -0.8923, Max: 1.6484, Mean: -0.5435)
Signal 7: DC Signal (Min: -8.0574, Max: -8.0574, Mean: -8.0574)
Signal 8: Sawtooth Wave (Min: -11.9090, Max: 2.5978, Mean: -4.6570)
Signal 9: Sinusoidal Signal (Min: -7.5653, Max: 1.3028, Mean: -3.1301)
Signal 10: Sinusoidal Signal (Min: -9.0566, Max: 4.5771, Mean: -2.2406)
Signal 11: Gaussian Pulse (Min: 1.8390, Max: 4.3025, Mean: 2.4783)
Signal 12: Exponential Signal (Min: 2.5127, Max: 196762.1982)
```

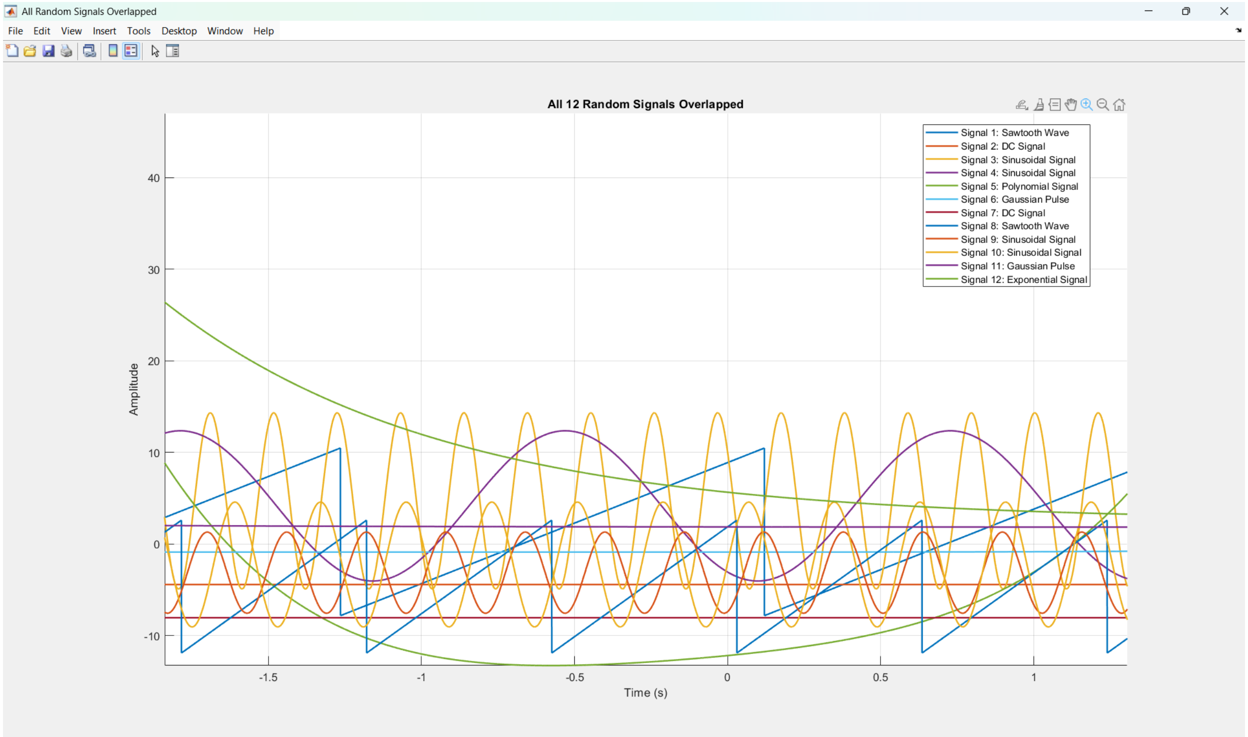


Figure 21: Random Signals - Individual Plots

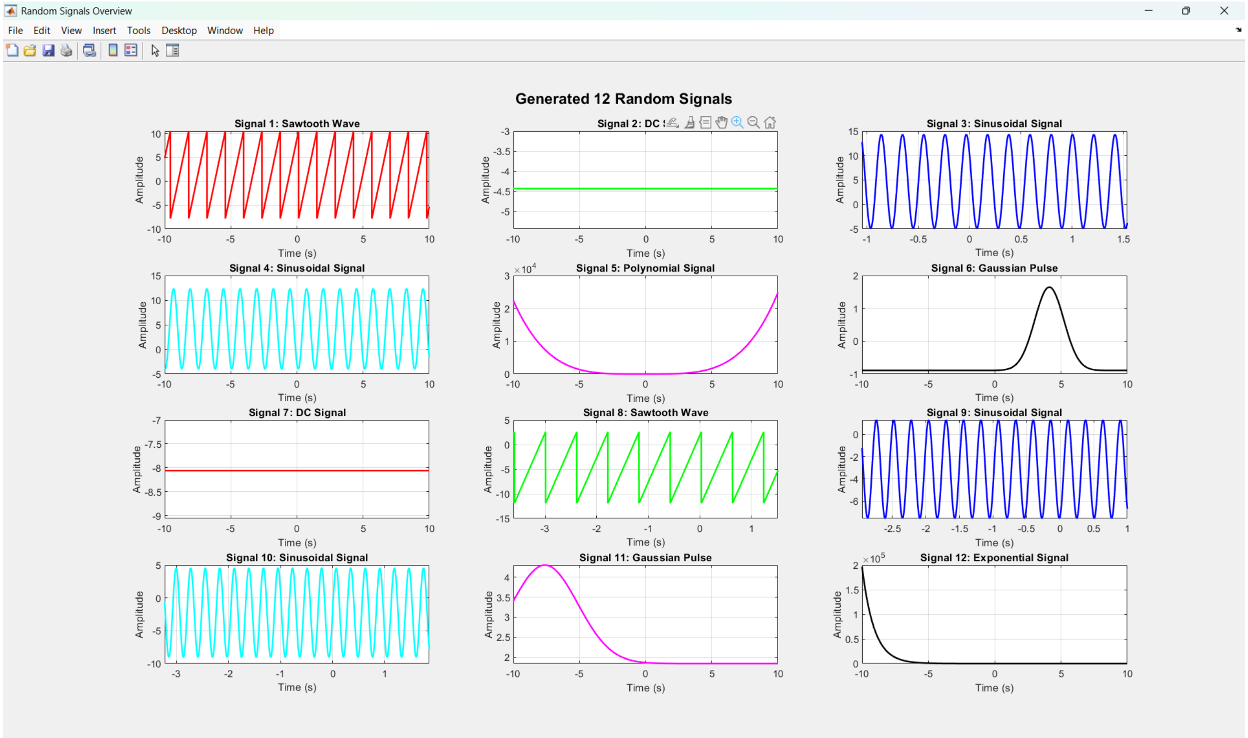


Figure 22: Random Signals - Overlapped View

2.7 Signal Operations

The program provides seven different signal transformation operations that can be applied to any generated signal.

2.7.1 Original Signal

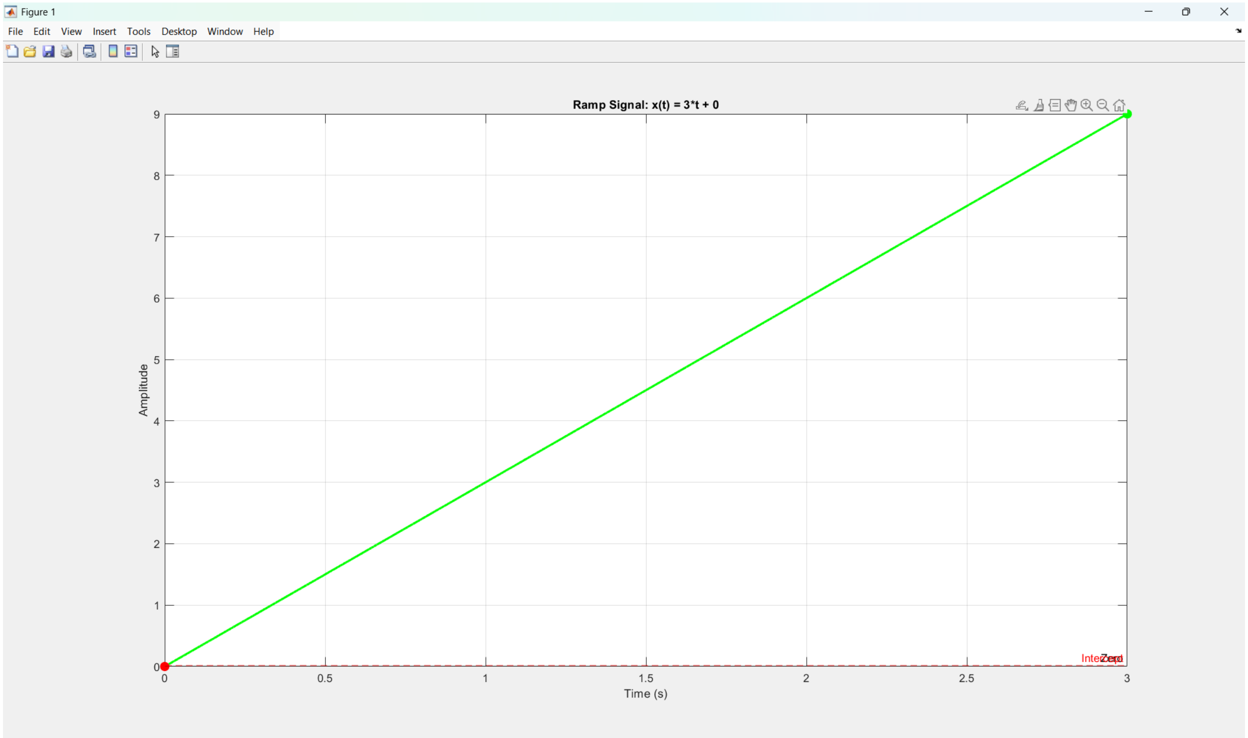


Figure 23: Original Signal Before Operations

2.7.2 Operation 1: Amplitude Scaling

This operation multiplies the signal amplitude by a user-specified scaling factor.

```
function [x_scaled, t_new, params] = amplitude_scaling(x_t, t)
% AMPLITUDE_SCALING - Scale the amplitude of a signal
%
% Syntax: [x_scaled, t_new, params] = amplitude_scaling(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_scaled - Scaled signal
%   t_new - Time vector (unchanged)
%   params - Structure containing operation parameters

    fprintf('\n--- AMPLITUDE SCALING ---\n');
    scale_factor = input('Enter the scaling factor (e.g., 2 for double, 0.5 for
        half): ');

    % Validate scaling factor
    while ~isnumeric(scale_factor) || ~isscalar(scale_factor)
        fprintf('    Invalid input! Please enter a numeric value.\n');
        scale_factor = input('Enter the scaling factor: ');
    end

    % Apply amplitude scaling
    x_scaled = scale_factor * x_t;
    t_new = t;

    % Store parameters
    params.operation = 'Amplitude Scaling';
    params.scale_factor = scale_factor;

    % Plot comparison
    figure('Name', 'Amplitude Scaling', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal');
    grid on;

    subplot(2, 1, 2);
    plot(t_new, x_scaled, 'r-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Scaled Signal (Factor = %.2f)', scale_factor));
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Scaling Factor: %.4f\n', scale_factor);
    fprintf('Original Signal - Min: %.4f, Max: %.4f, Mean: %.4f\n', ...
            min(x_t), max(x_t), mean(x_t));
    fprintf('Scaled Signal - Min: %.4f, Max: %.4f, Mean: %.4f\n', ...
            min(x_scaled), max(x_scaled), mean(x_scaled));
    fprintf('    Amplitude scaling completed!\n');
end
```

Listing 22: Amplitude Scaling

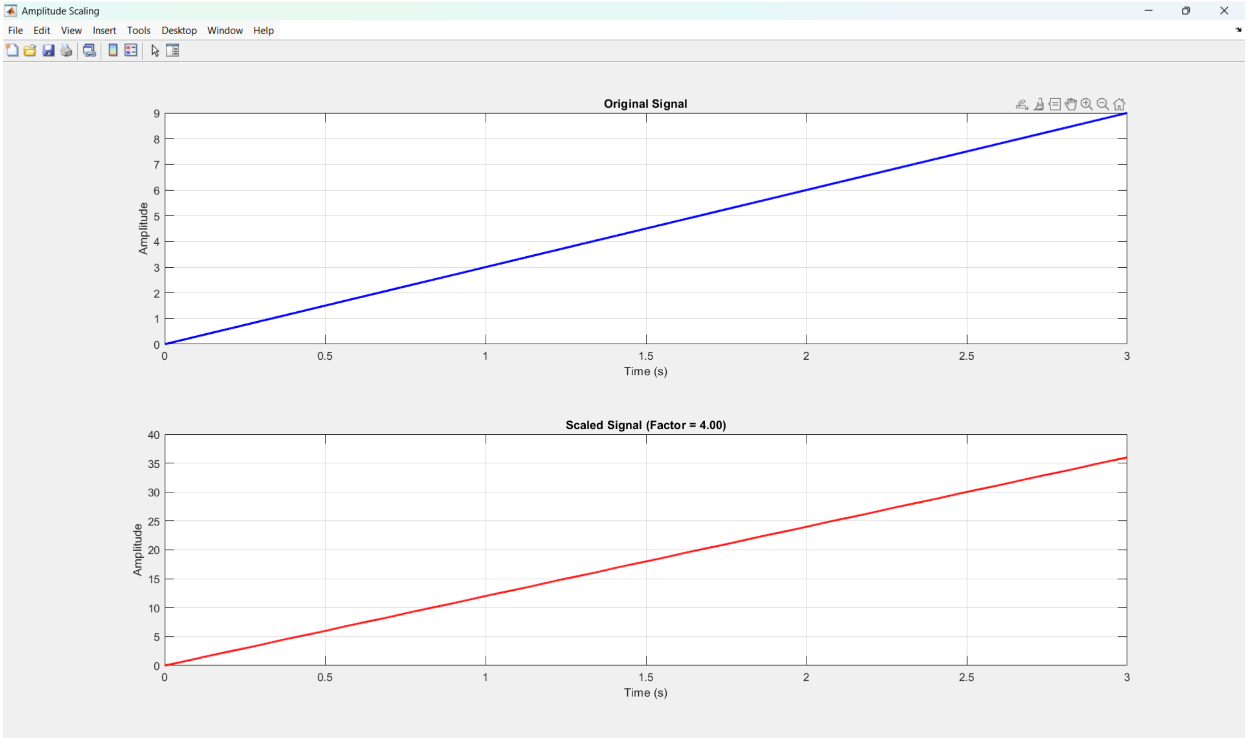


Figure 24: Amplitude Scaling Operation: $y(t) = A \cdot x(t)$

2.7.3 Operation 2: Time Reversal

This operation reverses the signal in time: $x(t) \to x(-t)$.

```
function [x_reversed, t_new, params] = time_reversal(x_t, t)
% TIME_REVERSAL - Reverse the signal in time
%
% Syntax: [x_reversed, t_new, params] = time_reversal(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_reversed - Time-reversed signal
%   t_new - Reversed time vector
%   params - Structure containing operation parameters

    fprintf('\n--- TIME REVERSAL ---\n');
    fprintf('Reversing signal in time: x(t)     x(-t)\n');

    % Apply time reversal
    x_reversed = fliplr(x_t);
    t_new = -fliplr(t);

    % Store parameters
    params.operation = 'Time Reversal';

    % Plot comparison
    figure('Name', 'Time Reversal', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal: x(t)');
    grid on;

    subplot(2, 1, 2);
    plot(t_new, x_reversed, 'r-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Time-Reversed Signal: x(-t)');
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Original time range: [%.4f, %.4f] s\n', t(1), t(end));
    fprintf('Reversed time range: [%.4f, %.4f] s\n', t_new(1), t_new(end));
    fprintf('    Time reversal completed!\n');
end
```

Listing 23: Time Reversal

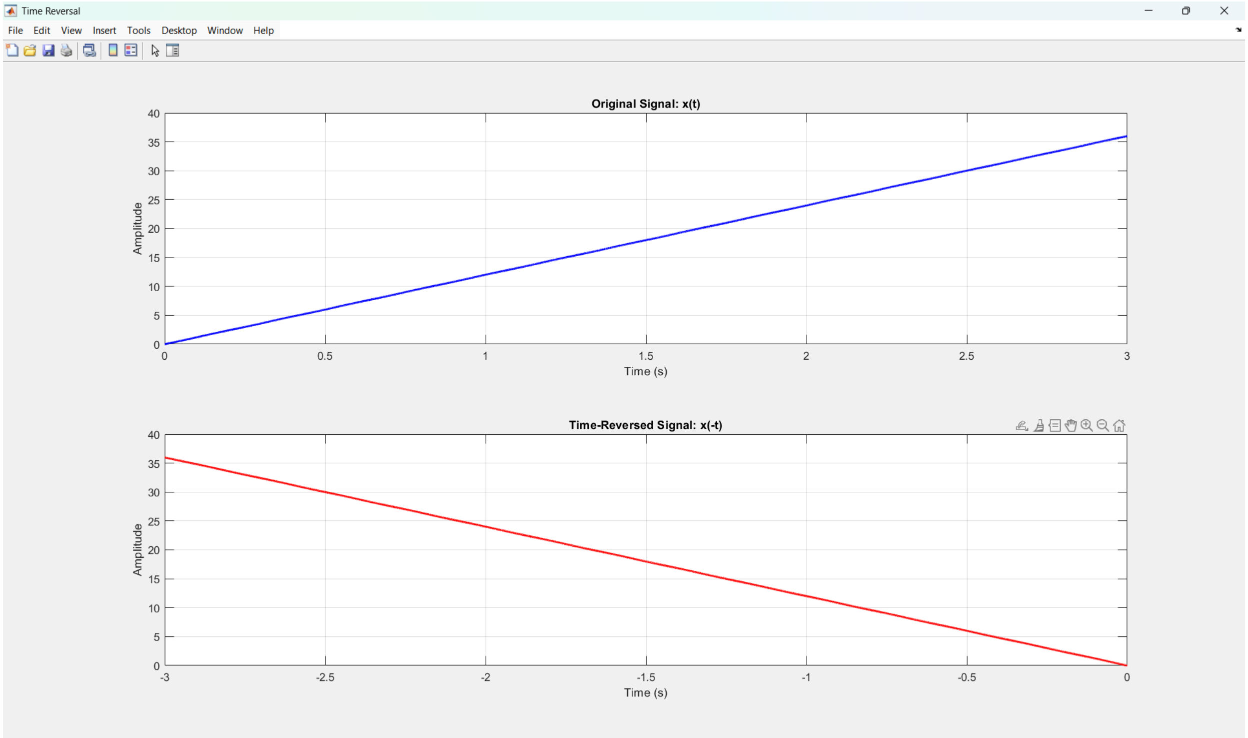


Figure 25: Time Reversal Operation: $y(t) = x(-t)$

2.7.4 Operation 3: Time Shift

This operation shifts the signal in time by a user-specified value: $x(t) \to x(t - t_0)$.

```
function [x_shifted, t_new, params] = time_shift(x_t, t)
% TIME_SHIFT - Shift the signal in time
%
% Syntax: [x_shifted, t_new, params] = time_shift(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_shifted - Time-shifted signal
%   t_new - Shifted time vector
%   params - Structure containing operation parameters

    fprintf('\n--- TIME SHIFT ---\n');
    shift_value = input('Enter the time shift value in seconds (positive = right,
        negative = left): ');

    % Validate shift value
    while ~isnumeric(shift_value) || ~isscalar(shift_value)
        fprintf('    Invalid input! Please enter a numeric value.\n');
        shift_value = input('Enter the time shift value in seconds: ');
    end

    % Apply time shift
    t_new = t + shift_value;
    x_shifted = x_t;  % Signal values remain the same, only time axis shifts

    % Store parameters
    params.operation = 'Time Shift';
    params.shift_value = shift_value;

    % Plot comparison
    figure('Name', 'Time Shift', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal: x(t)');
    grid on;

    subplot(2, 1, 2);
    plot(t_new, x_shifted, 'r-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Time-Shifted Signal: x(t - %.2f)', shift_value));
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Shift Value: %.4f seconds\n', shift_value);
    if shift_value > 0
        fprintf('Direction: Right (delayed)\n');
    elseif shift_value < 0
        fprintf('Direction: Left (advanced)\n');
    else
        fprintf('Direction: No shift\n');
    end
    fprintf('Original time range: [%.4f, %.4f] s\n', t(1), t(end));
    fprintf('Shifted time range: [%.4f, %.4f] s\n', t_new(1), t_new(end));
```

```
    fprintf('    Time shift completed!\n');
end
```

Listing 24: Time Shift

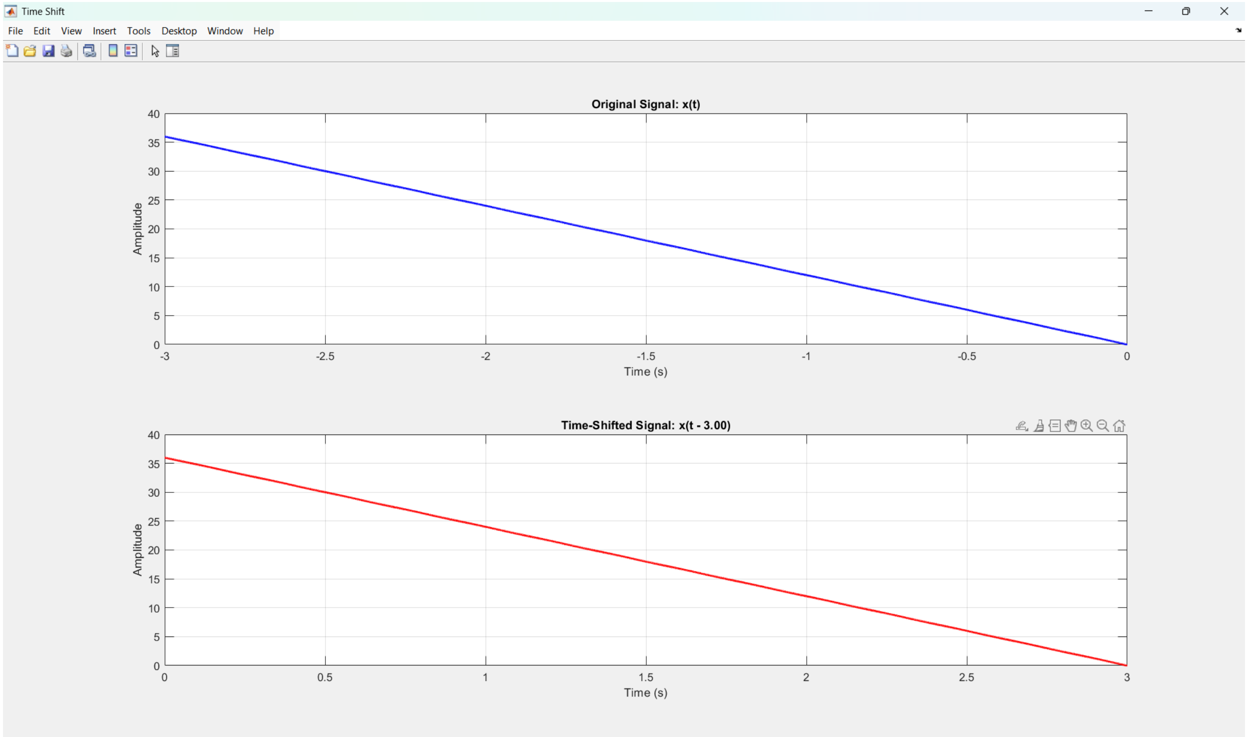


Figure 26: Time Shift Operation: $y(t) = x(t - t_0)$

2.7.5 Operation 4: Time Expansion

This operation expands (slows down) the signal: $x(t) \to x(t/a)$ where $a > 1$.

```
function [x_expanded, t_new, params] = time_expansion(x_t, t)
% TIME_EXPANSION - Expand the signal in time (slow down)
%
% Syntax: [x_expanded, t_new, params] = time_expansion(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_expanded - Time-expanded signal
%   t_new - Expanded time vector
%   params - Structure containing operation parameters

    fprintf('\n--- TIME EXPANSION ---\n');
    expansion_factor = input('Enter the expansion factor (e.g., 2 to make signal
        twice as slow): ');

    % Validate expansion factor
    while ~isnumeric(expansion_factor) || ~isscalar(expansion_factor) ||
        expansion_factor <= 0
        fprintf('    Invalid input! Please enter a positive numeric value.\n');
        expansion_factor = input('Enter the expansion factor: ');
    end
    % Apply time expansion: x(t)     x(t/a) where a > 1 expands the signal
    t_new = t * expansion_factor;
    x_expanded = x_t;  % Signal values remain the same
    % Store parameters
    params.operation = 'Time Expansion';
    params.expansion_factor = expansion_factor;
    % Plot comparison
    figure('Name', 'Time Expansion', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal: x(t)');
    grid on;
    subplot(2, 1, 2);
    plot(t_new, x_expanded, 'r-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Time-Expanded Signal: x(t/%.2f)', expansion_factor));
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Expansion Factor: %.4f\n', expansion_factor);
    fprintf('Original time range: [%.4f, %.4f] s (Duration: %.4f s)\n', ...
            t(1), t(end), t(end) - t(1));
    fprintf('Expanded time range: [%.4f, %.4f] s (Duration: %.4f s)\n', ...
            t_new(1), t_new(end), t_new(end) - t_new(1));
    fprintf('Signal is now %.2fx slower\n', expansion_factor);
    fprintf('    Time expansion completed!\n');
end
```

Listing 25: Time Expansion

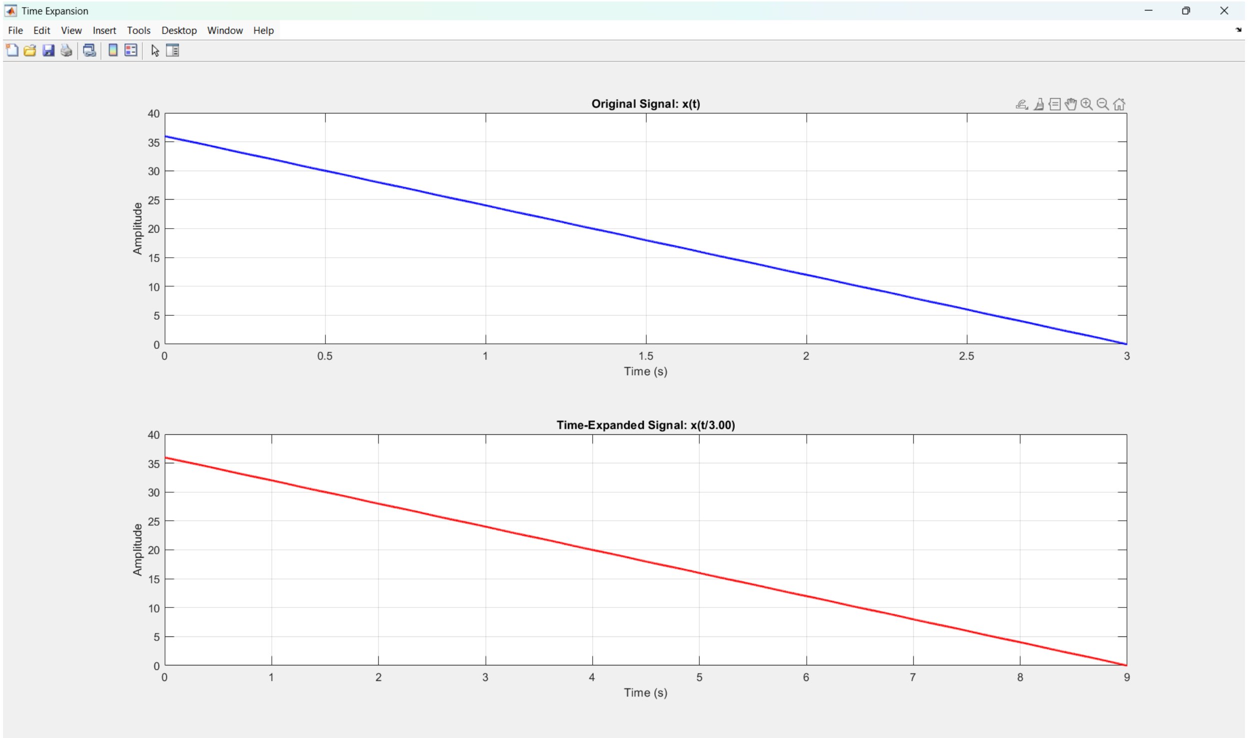


Figure 27: Time Expansion Operation: $y(t) = x(t/a)$

2.7.6 Operation 5: Time Compression

This operation compresses (speeds up) the signal: $x(t) \to x(at)$ where $a > 1$.

```
function [x_compressed, t_new, params] = time_compression(x_t, t)
% TIME_COMPRESSION - Compress the signal in time (speed up)
%
% Syntax: [x_compressed, t_new, params] = time_compression(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_compressed - Time-compressed signal
%   t_new - Compressed time vector
%   params - Structure containing operation parameters

    fprintf('\n--- TIME COMPRESSION ---\n');
    compression_factor = input('Enter the compression factor (e.g., 2 to make signal twice as fast): ');
    % Validate compression factor
    while ~isnumeric(compression_factor) || ~isscalar(compression_factor) || compression_factor <= 0
        fprintf('    Invalid input! Please enter a positive numeric value.\n');
        compression_factor = input('Enter the compression factor: ');
    end
    % Apply time compression: x(t)     x(a*t) where a > 1 compresses the signal
    t_new = t / compression_factor;
    x_compressed = x_t;  % Signal values remain the same
    % Store parameters
    params.operation = 'Time Compression';
    params.compression_factor = compression_factor;
    % Plot comparison
    figure('Name', 'Time Compression', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal: x(t)');
    grid on;

    subplot(2, 1, 2);
    plot(t_new, x_compressed, 'r-', 'LineWidth', 2);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Time-Compressed Signal: x(%.2f*t)', compression_factor));
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Compression Factor: %.4f\n', compression_factor);
    fprintf('Original time range: [%.4f, %.4f] s (Duration: %.4f s)\n', ...
            t(1), t(end), t(end) - t(1));
    fprintf('Compressed time range: [%.4f, %.4f] s (Duration: %.4f s)\n', ...
            t_new(1), t_new(end), t_new(end) - t_new(1));
    fprintf('Signal is now %.2fx faster\n', compression_factor);
    fprintf('    Time compression completed!\n');
end
```

Listing 26: Time Compression

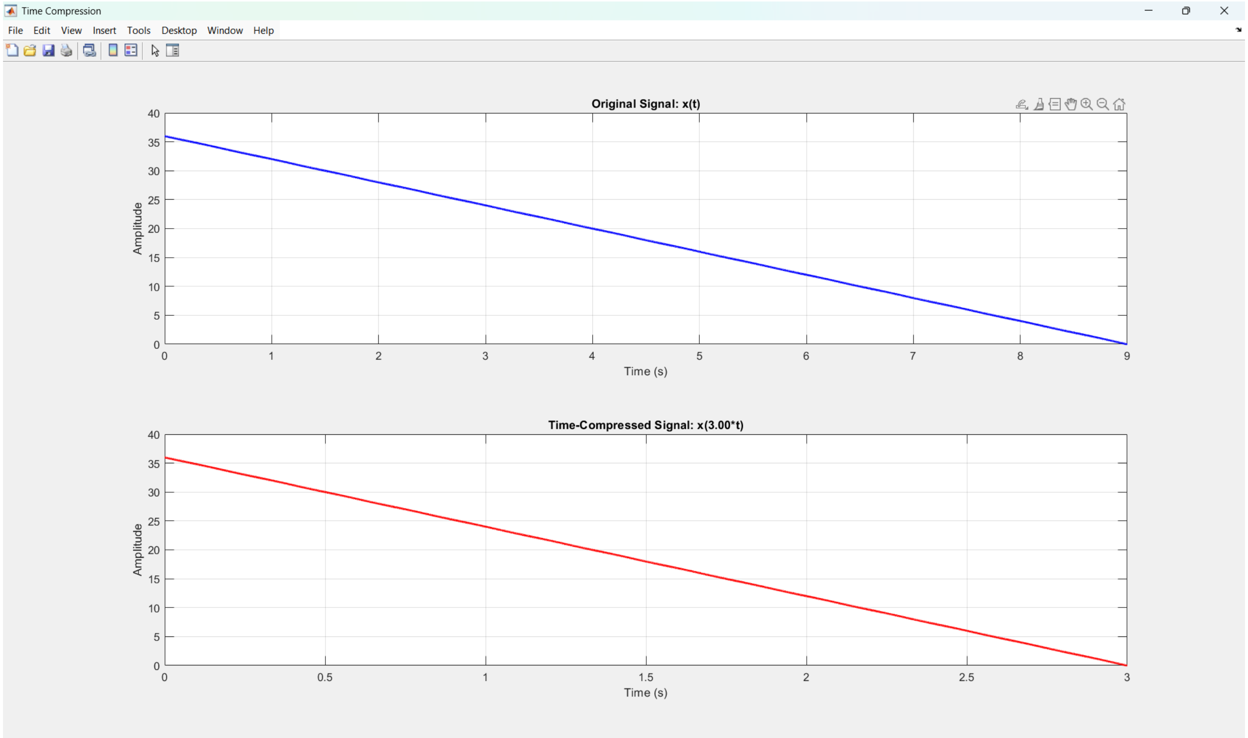


Figure 28: Time Compression Operation: $y(t) = x(at)$

2.7.7 Operation 6: Add Random Noise

This operation adds white Gaussian noise to the signal based on a specified Signal-to-Noise Ratio (SNR).

```
function [x_noisy, t_new, params] = add_noise(x_t, t)
% ADD_NOISE - Add random noise to the signal based on SNR
%
% Syntax: [x_noisy, t_new, params] = add_noise(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_noisy - Signal with added noise
%   t_new - Time vector (unchanged)
%   params - Structure containing operation parameters

    fprintf('\n--- ADD RANDOM NOISE ---\n');
    fprintf('SNR (Signal-to-Noise Ratio): Higher values = less noise\n');
    fprintf('Typical values: 10 dB (noisy), 20 dB (moderate), 30 dB (clean)\n');

    SNR_dB = input('Enter the desired SNR in dB: ');

    % Validate SNR
    while ~isnumeric(SNR_dB) || ~isscalar(SNR_dB)
        fprintf('    Invalid input! Please enter a numeric value.\n');
        SNR_dB = input('Enter the desired SNR in dB: ');
    end

    % Calculate signal power
    signal_power = mean(x_t.^2);

    % Calculate noise power from SNR
    % SNR (dB) = 10 * log10(signal_power / noise_power)
    SNR_linear = 10^(SNR_dB / 10);
    noise_power = signal_power / SNR_linear;

    % Generate white Gaussian noise
    noise = sqrt(noise_power) * randn(size(x_t));

    % Add noise to signal
    x_noisy = x_t + noise;
    t_new = t;

    % Calculate actual SNR
    actual_SNR_dB = 10 * log10(mean(x_t.^2) / mean(noise.^2));

    % Store parameters
    params.operation = 'Add Random Noise';
    params.SNR_dB = SNR_dB;
    params.actual_SNR_dB = actual_SNR_dB;
    params.noise_power = noise_power;

    % Plot comparison
    figure('Name', 'Add Random Noise', 'NumberTitle', 'off');

    subplot(3, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 1.5);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title('Original Signal');
    grid on;

```

```
    subplot(3, 1, 2);
    plot(t, noise, 'k-', 'LineWidth', 0.5);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Noise (Power = %.4f)', noise_power));
    grid on;

    subplot(3, 1, 3);
    plot(t_new, x_noisy, 'r-', 'LineWidth', 1.5);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Noisy Signal (SNR = %.2f dB)', actual_SNR_dB));
    grid on;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Target SNR: %.2f dB\n', SNR_dB);
    fprintf('Actual SNR: %.2f dB\n', actual_SNR_dB);
    fprintf('Signal Power: %.6f\n', signal_power);
    fprintf('Noise Power: %.6f\n', noise_power);
    fprintf('Original Signal - Mean: %.4f, Std: %.4f\n', mean(x_t), std(x_t));
    fprintf('Noisy Signal - Mean: %.4f, Std: %.4f\n', mean(x_noisy), std(x_noisy))
        ;
    fprintf('    Noise addition completed!\n');
end
```

Listing 27: Add Random Noise

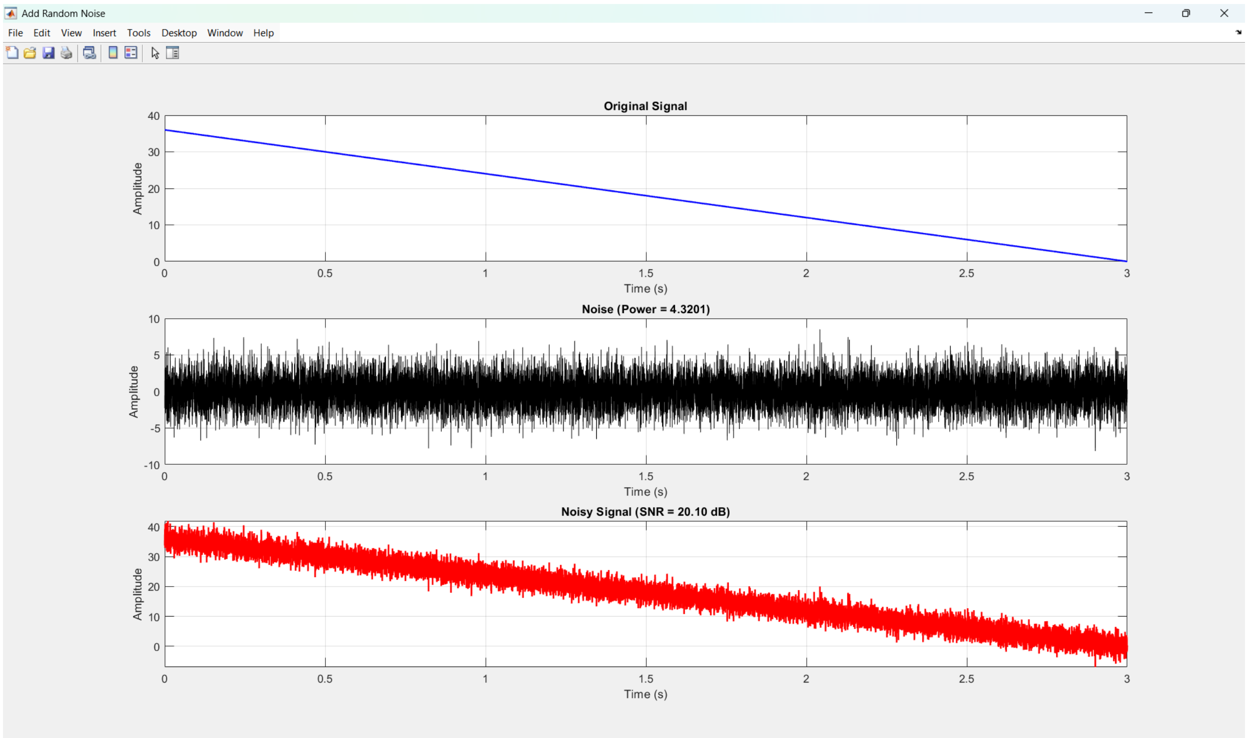


Figure 29: Add Random Noise Operation with SNR control

2.7.8 Operation 7: Smoothing (Moving Average)

This operation applies a moving average filter to smooth the signal.

```matlab
function [x_smoothed, t_new, params] = smoothing(x_t, t)
% SMOOTHING - Smooth the signal using moving average filter
%
% Syntax: [x_smoothed, t_new, params] = smoothing(x_t, t)
%
% Inputs:
%   x_t - Original signal
%   t - Time vector
%
% Outputs:
%   x_smoothed - Smoothed signal
%   t_new - Time vector (unchanged)
%   params - Structure containing operation parameters

    fprintf('\n--- SMOOTHING (MOVING AVERAGE) ---\n');
    fprintf('Window size determines smoothing strength: larger = smoother\n');
    fprintf('Recommended: 3-20 for subtle smoothing, 21-50 for strong smoothing\n');

    window_size = input('Enter the window size (odd number recommended): ');

    % Validate window size
    while ~isnumeric(window_size) || ~isscalar(window_size) || ...
          window_size < 1 || window_size ~= floor(window_size) || window_size > length(x_t)
        fprintf('    Invalid input! Please enter a positive integer less than signal length (%d).\n', length(x_t));
        window_size = input('Enter the window size: ');
    end

    % Make window size odd for symmetry
    if mod(window_size, 2) == 0
        window_size = window_size + 1;
        fprintf('    Window size adjusted to %d (odd number) for symmetry.\n', window_size);
    end

    % Apply moving average filter
    % Create moving average filter
    window = ones(1, window_size) / window_size;

    % Use convolution for moving average (with 'same' to keep original length)
    x_smoothed = conv(x_t, window, 'same');
    t_new = t;

    % Store parameters
    params.operation = 'Smoothing (Moving Average)';
    params.window_size = window_size;

    % Calculate smoothing metrics
    original_variance = var(x_t);
    smoothed_variance = var(x_smoothed);
    variance_reduction = (1 - smoothed_variance/original_variance) * 100;

    % Plot comparison
    figure('Name', 'Smoothing', 'NumberTitle', 'off');

    subplot(2, 1, 1);
    plot(t, x_t, 'b-', 'LineWidth', 1.5);
    xlabel('Time (s)');
    ylabel('Amplitude');
```

```
    title('Original Signal');
    grid on;

    subplot(2, 1, 2);
    plot(t_new, x_smoothed, 'r-', 'LineWidth', 1.5);
    hold on;
    plot(t, x_t, 'b:', 'LineWidth', 0.5, 'Color', [0.5 0.5 0.5]);
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Smoothed Signal (Window = %d)', window_size));
    legend('Smoothed', 'Original', 'Location', 'best');
    grid on;
    hold off;

    % Plot overlay comparison
    figure('Name', 'Smoothing Comparison', 'NumberTitle', 'off');
    plot(t, x_t, 'b-', 'LineWidth', 1.5, 'DisplayName', 'Original');
    hold on;
    plot(t_new, x_smoothed, 'r-', 'LineWidth', 2, 'DisplayName', 'Smoothed');
    xlabel('Time (s)');
    ylabel('Amplitude');
    title(sprintf('Original vs Smoothed Signal (Window = %d)', window_size));
    legend('Location', 'best');
    grid on;
    hold off;

    % Display operation info
    fprintf('\n--- Operation Results ---\n');
    fprintf('Window Size: %d samples\n', window_size);
    fprintf('Original Signal - Mean: %.4f, Variance: %.6f\n', mean(x_t),
        original_variance);
    fprintf('Smoothed Signal - Mean: %.4f, Variance: %.6f\n', mean(x_smoothed),
        smoothed_variance);
    fprintf('Variance Reduction: %.2f%%\n', variance_reduction);

    % Calculate difference
    difference = x_t - x_smoothed;
    fprintf('Mean Absolute Difference: %.6f\n', mean(abs(difference)));
    fprintf('Max Absolute Difference: %.6f\n', max(abs(difference)));
    fprintf('    Smoothing completed!\n');
end
```

Listing 28: Smoothing

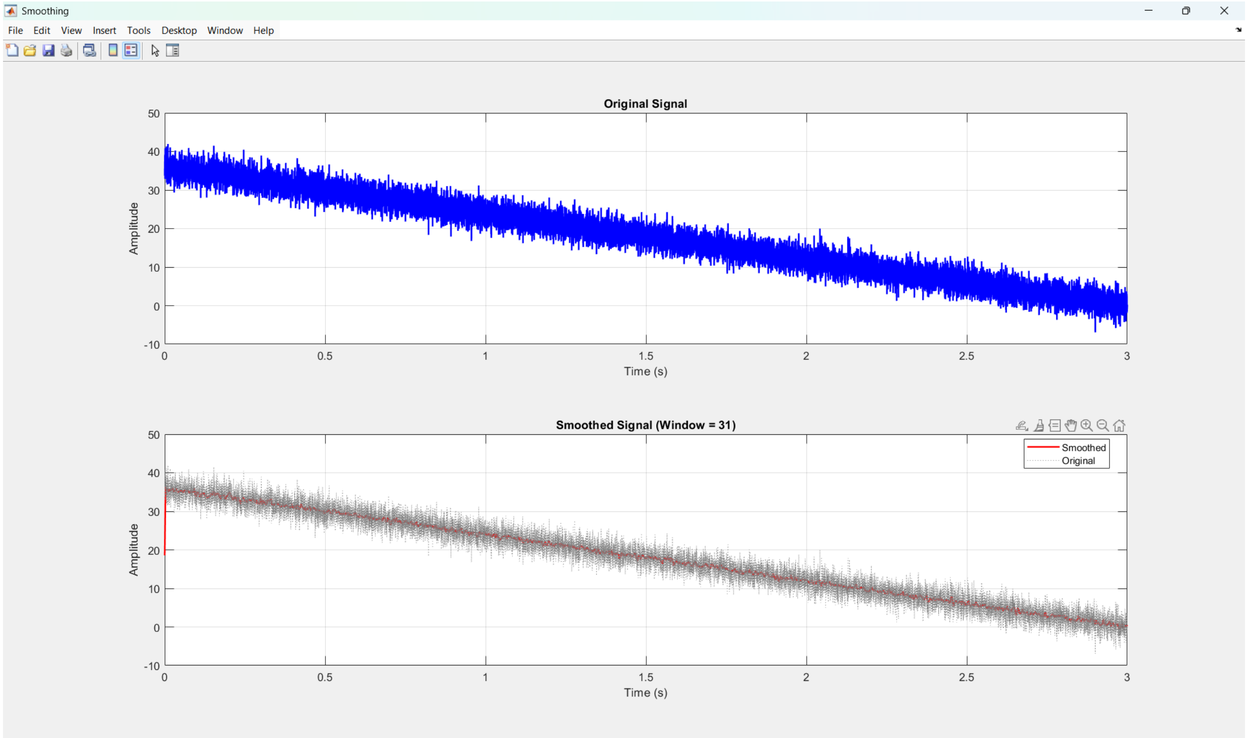


Figure 30: Smoothing Operation - Before and After Comparison

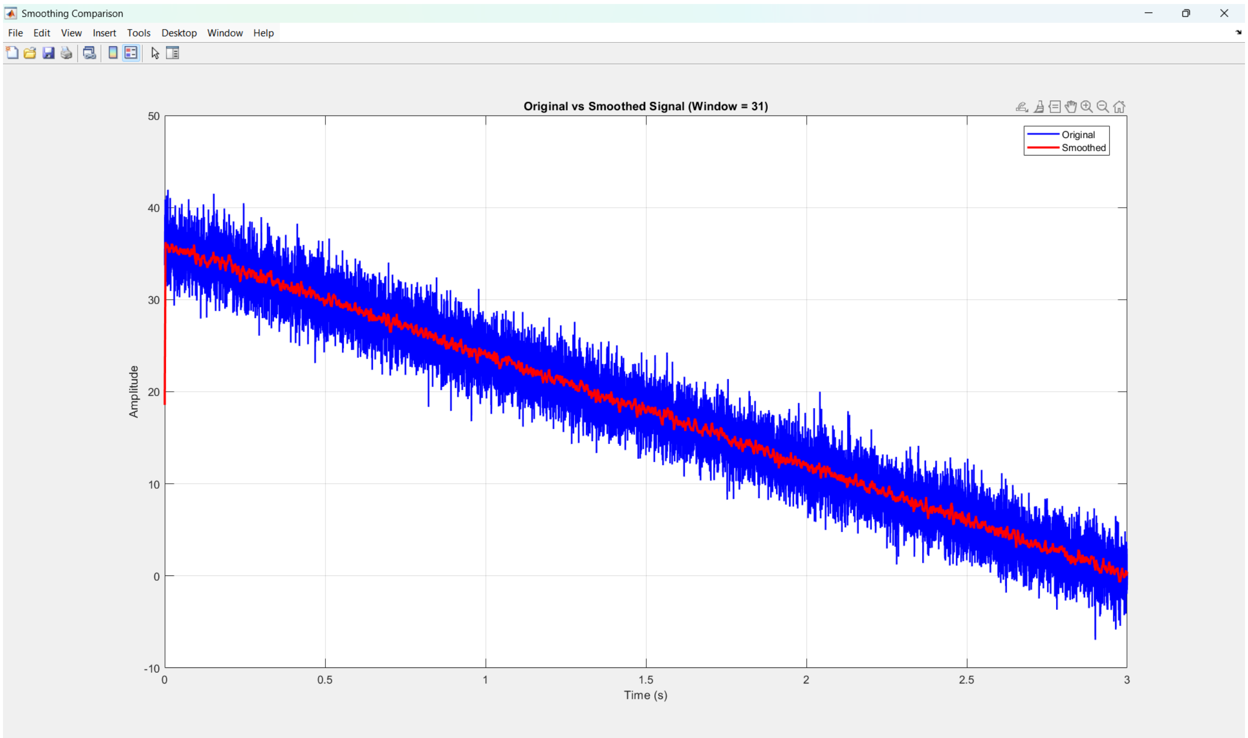


Figure 31: Smoothing Operation - Overlay Comparison

3 Conclusion

This project successfully implemented a comprehensive signal processing system with the following achievements:

3.1 Part I Achievements

- Successfully analyzed complex piecewise signals using both analytical (handwritten) and numerical (MATLAB) methods
- Computed Fourier Transforms and analyzed frequency domain characteristics
- Determined bandwidth requirements for 95% signal energy conservation
- Implemented Fourier Series analysis and signal reconstruction with harmonic synthesis
- Compared bandwidth efficiency of different signal types through spectral analysis

3.2 Part II Achievements

- Developed a comprehensive modular signal generator supporting 7 different signal types:
  - DC (Constant)
  - Ramp (Linear)
  - Polynomial (user-defined order)
  - Exponential (growth/decay)
  - Sinusoidal (Sine/Cosine)
  - Gaussian Pulse
  - Sawtooth Wave
- Implemented signal combination with user-defined breakpoints for complex piecewise signals
- Created 7 fundamental signal transformation operations:
  - Amplitude Scaling
  - Time Reversal
  - Time Shift
  - Time Expansion
  - Time Compression
  - Random Noise Addition (SNR-based)
  - Smoothing (Moving Average Filter)
- Developed automatic random signal generation capability with configurable parameters
- Built comprehensive visualization and analysis tools with real-time parameter display

3.3 Key Learning Outcomes

- Deep understanding of time-domain and frequency-domain signal analysis
- Proficiency in MATLAB programming for signal processing applications
- Experience with modular software design and user interface development
- Practical application of signal transformations and their effects on waveforms

- Understanding of sampling theory, Nyquist criterion, and aliasing
- Hands-on experience with Fourier analysis, both continuous and discrete
- Knowledge of energy and bandwidth relationships in signal processing

3.4 Technical Highlights

- Implementation of robust input validation and error handling
- Automatic sampling frequency adjustment for sinusoidal and sawtooth waves
- Real-time calculation and display of signal properties and statistics
- Support for cascading multiple operations on signals
- Comprehensive documentation with visual feedback at every step

4 References

1. Course Lecture Notes - Signals and Systems, Alexandria University, Faculty of Engineering
2. MATLAB Documentation - Signal Processing Toolbox, MathWorks Inc.
3. Project Requirements Document - Final Project 2025, Signals and Systems Course