DSP Quiz 2 Solutions

Question 1: Fourier Series and DFT of a Square Wave

(a) Derivation of Fourier Series

Let the square wave $x(t)$ have period T and amplitude ± 1, defined over one period as:

$$x(t) = \begin{cases} 1, & 0 < t < \frac{T}{2} \\ -1, & -\frac{T}{2} < t < 0 \end{cases}$$

This is an odd, periodic function. Its Fourier series will contain only sine terms:

$$x(t) = \sum_{n=1}^{\infty} b_n \sin\left(\frac{2\pi n t}{T}\right)$$

We compute the sine coefficients b_n as:

$$b_n = \frac{2}{T} \int_0^{T/2} x(t) \sin\left(\frac{2\pi n t}{T}\right) dt$$

Since $x(t) = 1$ for $0 < t < T/2$, this becomes:

$$b_n = \frac{2}{T} \int_0^{T/2} \sin\left(\frac{2\pi n t}{T}\right) dt = \frac{2}{T} \left[-\frac{T}{2\pi n} \cos\left(\frac{2\pi n t}{T}\right) \right]_0^{T/2}$$

$$= -\frac{2}{2\pi n} \left[\cos(\pi n) - \cos(0)\right] = -\frac{1}{\pi n} \left[(-1)^n - 1\right]$$

So:

$$b_n = \begin{cases} \frac{4}{\pi n}, & \text{if } n \text{ is odd} \\ 0, & \text{if } n \text{ is even} \end{cases}$$

Hence, the final Fourier series is:

$$x(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{1}{2k-1} \sin\left((2k-1)\omega_0 t\right), \quad \omega_0 = \frac{2\pi}{T}$$

(b) Time Series Generation in Octave

We generate a square wave with fundamental frequency $f_0 = 1.4$ MHz over a time window covering 10 periods. The signal is approximated using the first 5 odd harmonics:

```
f0 = 1.4e6;
T = 1 / f0;
Tmax = 10 * T;
fs = 100e6;
dt = 1 / fs;
t = 0:dt:Tmax;
N = 5;
x = zeros(size(t));
for k = 1:N
    n = 2*k - 1;
    bn = 4 / (pi * n);
    x = x + bn * sin(2 * pi * n * f0 * t);
end
```

(c) Computing the DFT

We compute the Discrete Fourier Transform (DFT) using the `fft` function:

```
X = fft(x);
Nfft = length(X);
f = (0:Nfft-1) * fs / Nfft;
X_mag = abs(X);
f_half = f(1:Nfft/2);
X_half = X_mag(1:Nfft/2);
```

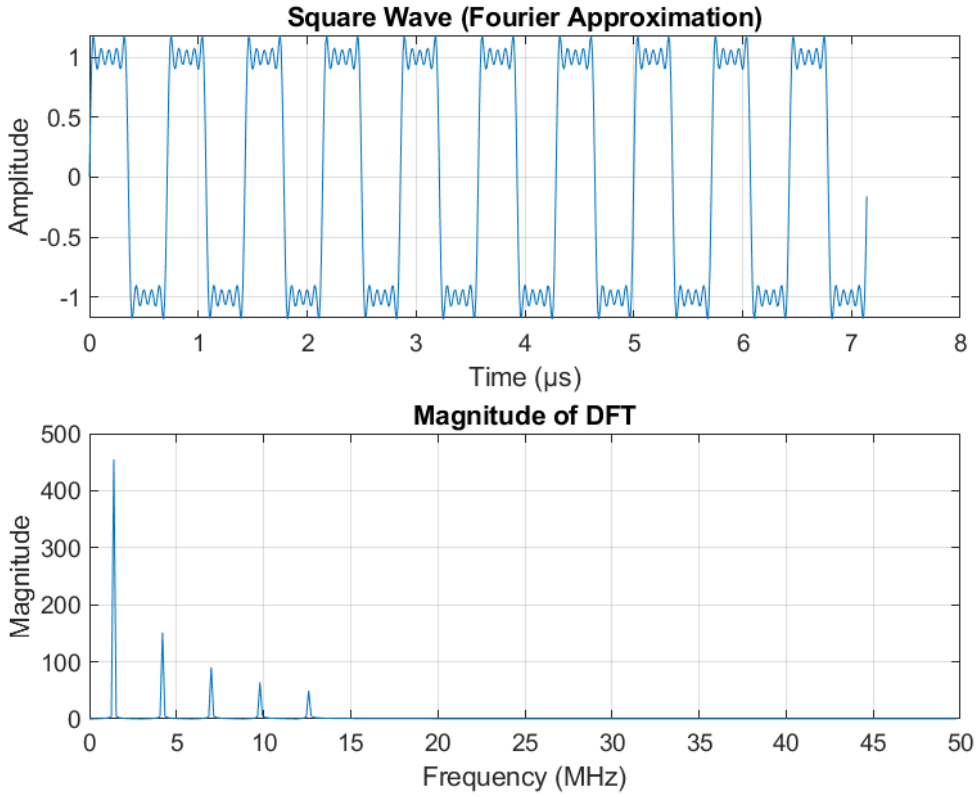
This provides the magnitude spectrum of the signal.

(d) Plotting the Time Series and DFT Magnitude

We use `subplot()` to visualize both the time-domain signal and the DFT magnitude, and save the figure:

```
figure;
subplot(2,1,1);
plot(t * 1e6, x);
xlabel('Time (s)');
ylabel('Amplitude');
title('Square Wave (Fourier Approximation)');

subplot(2,1,2);
plot(f_half / 1e6, X_half);
```

```
xlabel('Frequency (MHz)');
ylabel('Magnitude');
title('Magnitude of DFT');

print('square_wave_plots.png', '-dpng');
```



The time-domain plot shows the square wave synthesized from sine harmonics. The frequency-domain plot shows spectral peaks at odd multiples of the fundamental frequency, as expected.

Question 2: Gibbs Effect Observation

A Gibbs effect appears in the square wave generated during the previous question. The signal displays tiny disturbances like ripples around the time when it switches from +1 to –1 or from –1 to +1. The resulting ripples stem from the fact that the Fourier series uses sine waves that produce continuous smooth curves in order to create the square wave. The Fourier-series method falls short when it comes to exactly reproducing square wave corners. No matter how many extra terms we include to the Fourier series the ripples persist near the waveform ends. The time-domain signals acquire diminishing size and approach the different states of the binary code without altering their heights. A typical jump overshoot belongs to around 9% of the total jump duration. A natural phenomenon known as Gibbs effect

occurs because sine waves are used to generate square waves resulting in its clear visibility in time-domain analog signals.

Question 3: DFT and Group Delay of Impulse Signal

(a) Define a $\delta[n]$delta[n] Signal in Octave

We define a discrete-time impulse signal $\delta[n]$ of length $N = 1000$, where only the first sample is non-zero. This represents a unit impulse at $n = 0$.

Octave code:

```
N = 1000;
x = zeros(1, N);
x(1) = 1;  % Impulse at n = 0
```

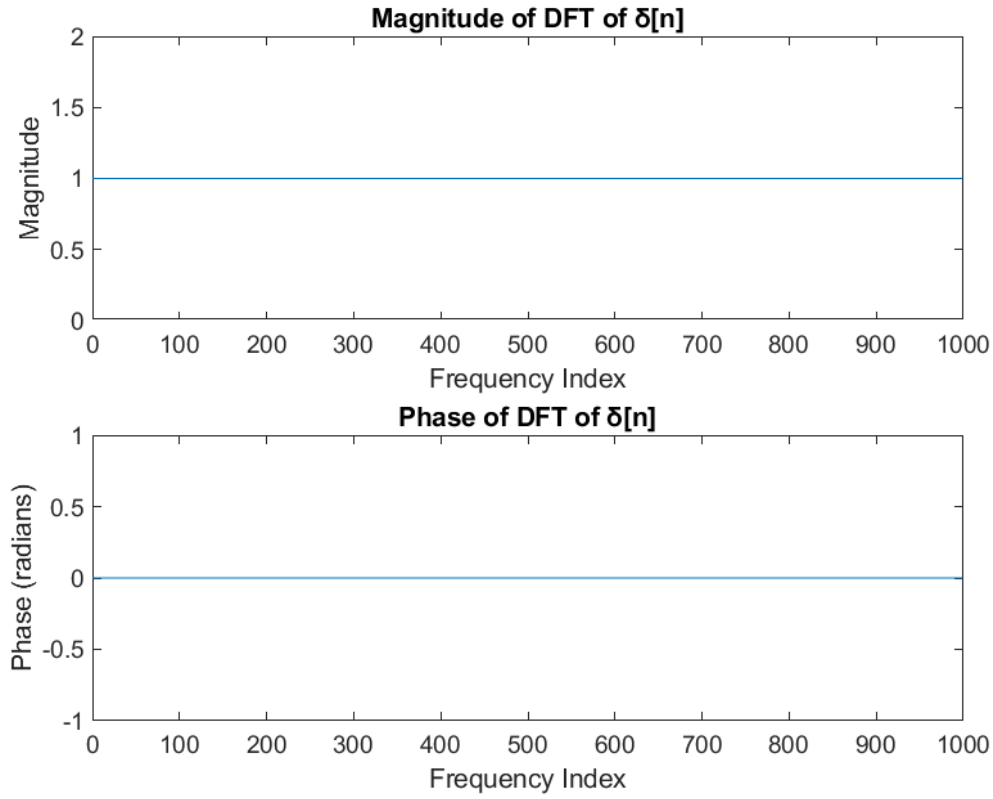
(b) Compute the Magnitude and Phase of the DFT

We compute the Discrete Fourier Transform (DFT) of the signal using `fft`, and then plot both magnitude and phase versus frequency index.

Octave code:

```
X = fft(x);
f = 0:N-1;

X_mag = abs(X);
X_phase = angle(X);

figure;
subplot(2,1,1);
plot(f, X_mag);
xlabel('Frequency Index');
ylabel('Magnitude');
title('Magnitude of DFT of [n]');

subplot(2,1,2);
plot(f, X_phase);
xlabel('Frequency Index');
ylabel('Phase (radians)');
title('Phase of DFT of [n]');

print('impulse_dft_original.png', '-dpng');
```

The magnitude is constant (all ones), and the phase is zero throughout, which is expected for a $\delta[n]$ signal.



(c) Advance the Impulse by 100 Samples and Recompute the DFT

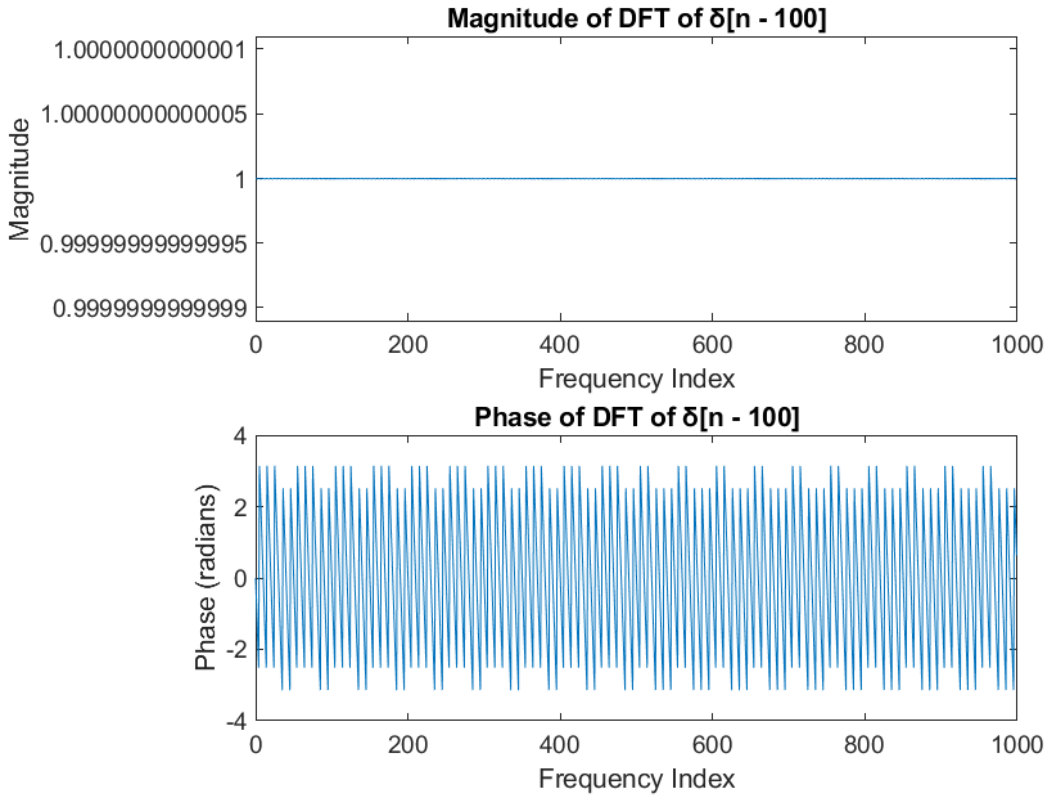
We shift the impulse to $n = 100$ (i.e., delay the signal), and recompute the DFT.

Octave code:

```
x_shifted = zeros(1, N);
x_shifted(101) = 1;

X_shifted = fft(x_shifted);
X_shifted_mag = abs(X_shifted);
X_shifted_phase = angle(X_shifted);

figure;
subplot(2,1,1);
plot(f, X_shifted_mag);
xlabel('Frequency Index');
ylabel('Magnitude');
title('Magnitude of DFT of [n - 100]');

subplot(2,1,2);
plot(f, X_shifted_phase);
xlabel('Frequency Index');
```

```
ylabel('Phase (radians)');
title('Phase of DFT of [n - 100]');

print('impulse_dft_shifted.png', '-dpng');
```

The magnitude remains the same, but the phase now varies linearly due to the delay.



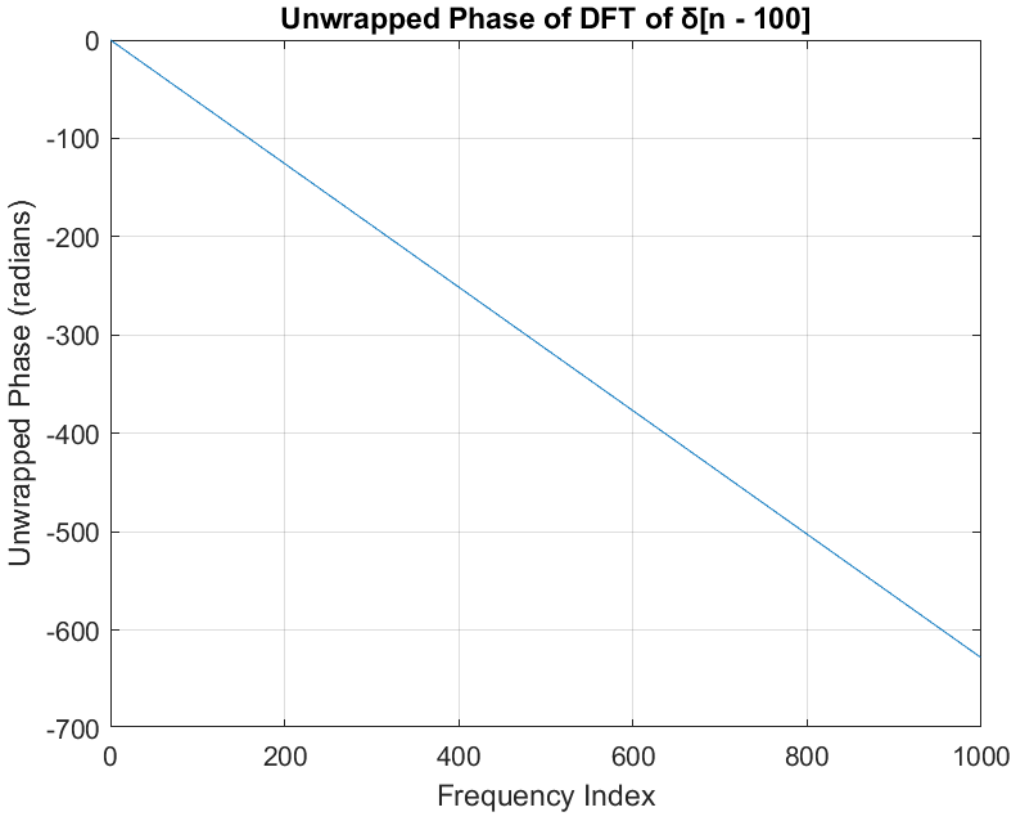
(d) Use `unwrap()` to Graph the Linear Phase

We use `unwrap()` to smooth the phase and observe its linear nature.

Octave code:

```
unwrapped_phase = unwrap(angle(X_shifted));

figure;
plot(f, unwrapped_phase);
xlabel('Frequency Index');
ylabel('Unwrapped Phase (radians)');
title('Unwrapped Phase of DFT of [n - 100]');
grid on;

print('impulse_unwrapped_phase.png', '-dpng');
```

The unwrapped phase is a straight line with negative slope, confirming the linear relationship due to delay.



(e) Estimate Group Delay from the Phase Slope

Group delay is the negative slope of the unwrapped phase versus frequency. We approximate it using a numerical derivative.

Octave code:

```
w = 2 * pi * f / N;                    % Angular frequency
dphi_dw = diff(unwrapped_phase) ./ diff(w);   % Derivative
group_delay = -mean(dphi_dw);          % Estimate

fprintf('Estimated group delay: %.2f samples\n', group_delay);
```

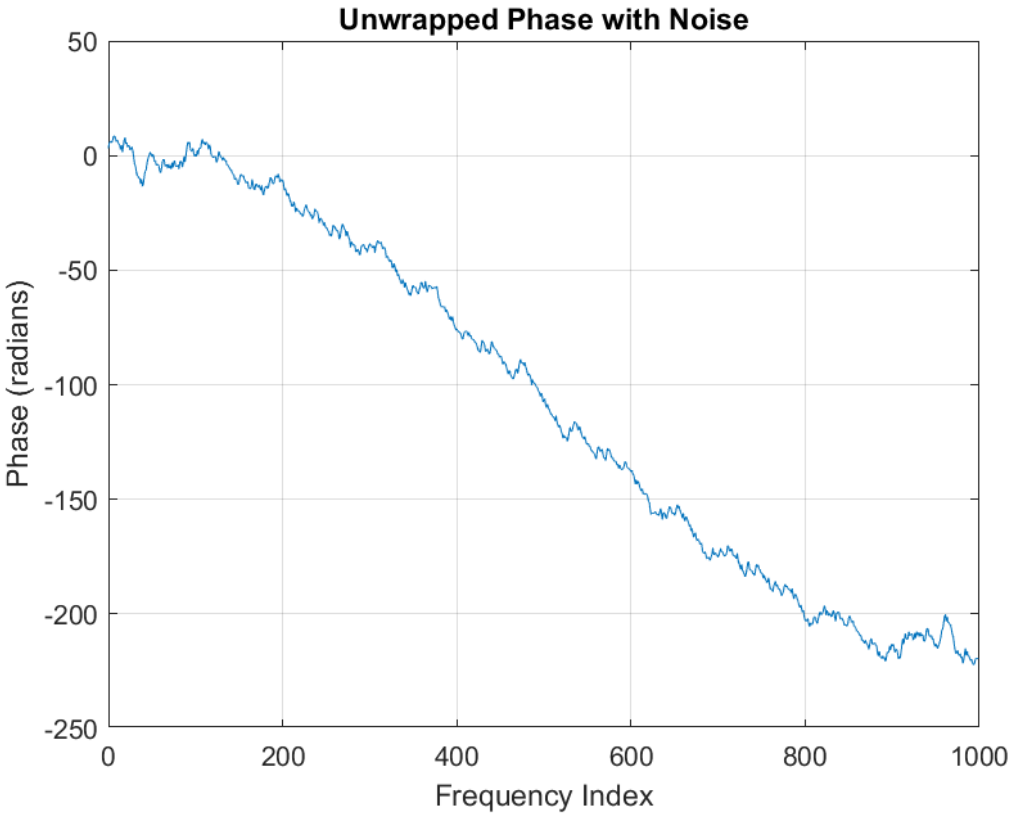
The estimated group delay is close to 100 samples, which matches the actual delay applied to $\delta[n]$.

(f) Bonus: Effect of Noise on Group Delay

If we add noise to the signal, the phase becomes less smooth, and the group delay estimate becomes less accurate. This is especially visible at higher frequencies.

Octave code:

```
x_noisy = x_shifted + 0.05 * randn(1, N);  % Add noise
X_noisy = fft(x_noisy);
phase_noisy = unwrap(angle(X_noisy));

figure;
plot(f, phase_noisy);
xlabel('Frequency Index');
ylabel('Phase (radians)');
title('Unwrapped Phase with Noise');
grid on;

print('impulse_noisy_phase.png', '-dpng');
```



The plot shows how random fluctuations in phase (due to noise) can distort the slope, making the group delay calculation less reliable.

Question 4: Clipping in DSP Data

(a) Generating the Sine Wave

We create a sine wave with:

- Sampling rate: 10 MHz
- Frequency: 100 kHz
- Duration: 6 ms
- Amplitude: 1.0

This gives us more than 10^4 samples.

Octave code:

```
fs = 10e6;              % 10 MHz sampling rate
f0 = 100e3;             % 100 kHz signal frequency
Tmax = 6e-3;            % Duration of 6 ms
t = 0:1/fs:Tmax;
A = 1.0;
x = A * sin(2 * pi * f0 * t);
```

(b) Clipping the Signal

We clip the signal so that any value above 0.75 is set to 0.75, and any value below –0.75 is set to –0.75.

Octave code:

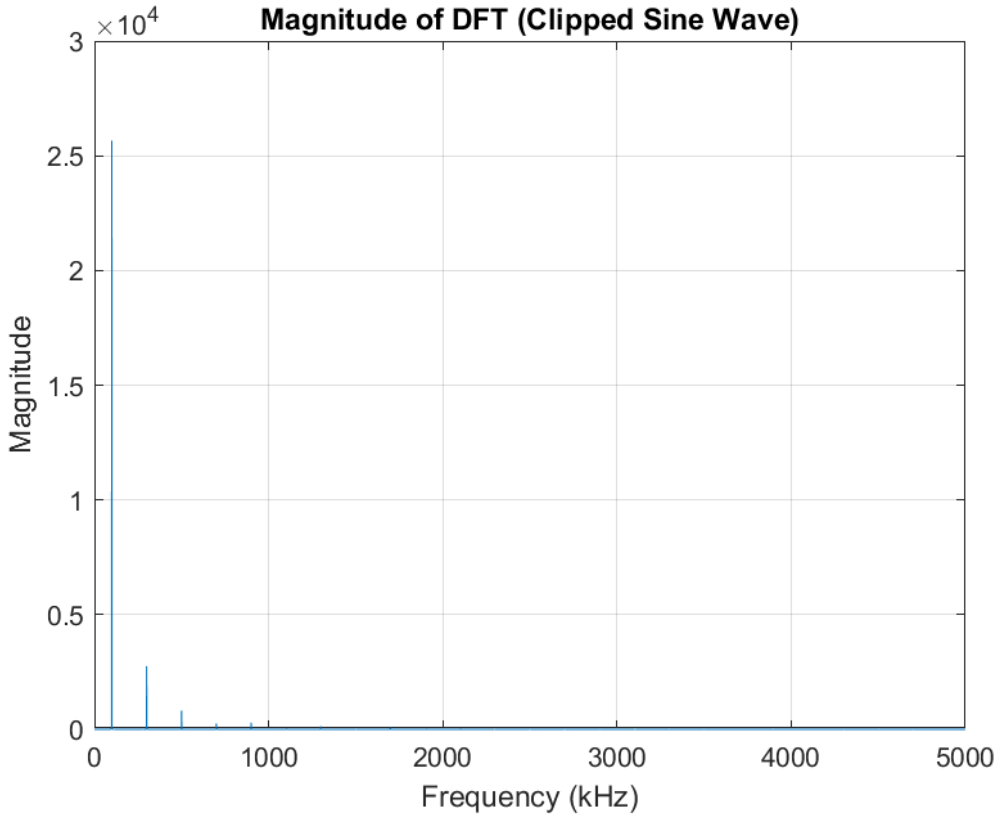
```
x(x > 0.75) = 0.75;
x(x < -0.75) = -0.75;
```

(c) Plot the DFT Magnitude and Locate the Spike

We compute the DFT of the clipped signal and plot its magnitude:

Octave code:

```
N = length(x);
X = fft(x);
f = (0:N-1) * fs / N;
X_mag = abs(X);

% First half of spectrum
f_half = f(1:N/2);
X_half = X_mag(1:N/2);

figure;
plot(f_half/1e3, X_half);  % Frequency in kHz
xlabel('Frequency (kHz)');
ylabel('Magnitude');
title('Magnitude of DFT (Clipped Sine Wave)');
grid on;

print('clipped_dft.png', '-dpng');
```



(d) Interpretation: Do You Observe Harmonics?

Yes, we observe harmonics. The peak occurs exactly at 100 kHz because this frequency matches the original sine wave frequency. Clipping creates new frequency elements which tend to occur at even frequencies that are multiple times of 100 kHz for example 300 kHz and 500 kHz and so on. Clipping effects the input signal because it creates non-linear behavior that deforms the sine wave shape. The distorted form resembles a square wave that produces strong odd harmonics due to its nature. A distortion of the signal becomes evident through its generation of additional frequency spikes. Clipping transforms a signal's shape by producing harmonics which were absent from the initial sine wave.

Question 5: NASDAQ Prices and Moving Average Filter

(a) Plotting the Data and Applying a Moving Average

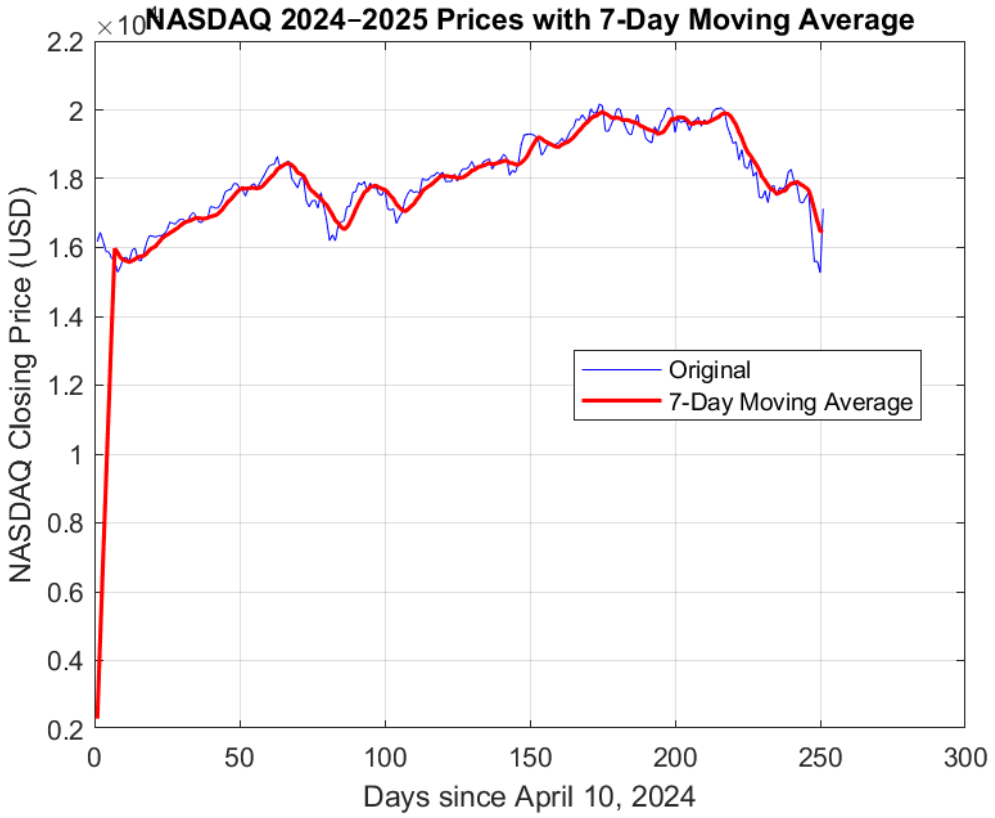
The data was obtained from the file `NASDAQ_2024_2025_converted.csv`, which contains:

- Column 1: Number of days since April 10, 2024
- Column 2: NASDAQ closing prices in US dollars

A 7-day moving average was applied to smooth short-term fluctuations and highlight longer-term trends.

Complete Octave code:

```
data = csvread('NASDAQ_2024_2025.csv', 1, 0);
days = data(:, 1);
prices = data(:, 2);

window_size = 7;
smoothed = filter(ones(1, window_size) / window_size, 1, prices);

figure;
plot(days, prices, 'b', 'DisplayName', 'Original');
hold on;
plot(days, smoothed, 'r', 'LineWidth', 1.5, 'DisplayName', '7-Day Moving Average');
xlabel('Days since April 10, 2024');
ylabel('NASDAQ Closing Price (USD)');
legend('Location', 'best');
title('NASDAQ 2024{2025 Prices with 7-Day Moving Average');
grid on;

print('nasdaq_moving_avg.png', '-dpng');

lag = (window_size - 1) / 2;
fprintf('Moving average lag: %.1f days\n', lag);
```



(b) Sharp Drop from Tariff Announcement

A sharp decline in NASDAQ values is visible between days 240 and 250, likely corresponding to the announcement of U.S. tariffs. This event causes immediate investor reaction, resulting in a significant dip in the index.

(c) Sharp Rise from Tariff Pause (April 2025)

Around day 180 to 200, the index shows a noticeable recovery, which aligns with the U.S. President's announcement of a 90-day pause in tariffs. The market responds positively, reflecting renewed confidence.

(d) Does the Moving Average Capture Rapid Shifts?

No. The moving average smooths the signal, reducing its ability to track sharp changes quickly. As a low-pass filter, it delays recognition of sudden events, such as economic announcements or shocks.

(e) Bonus: Lag of the Moving Average Filter

The lag introduced by an N-point moving average filter is calculated as:

$$\text{Lag} = \frac{N-1}{2} = \frac{7-1}{2} = 3 \text{ days}$$

This lag represents the delay in how quickly the filter reflects changes in the underlying data.

Question 6: Isolating an AM Station with Windowed-Sinc Filters

(a) Design a Low-Pass Filter to Suppress Noise Above 745 kHz

Octave code:

```
fs = 10e6;
fc = 740e3;

M = 101;
fc_lp = 745e3 / fs;
n = 0:M-1;
h_lp = sinc(2 * fc_lp * (n - (M-1)/2));
w = hamming(M)';
h_lp = h_lp .* w;
h_lp = h_lp / sum(h_lp);
```

(b) Design a High-Pass Filter to Eliminate Noise Below 735 kHz Using Spectral Inversion

Octave code:

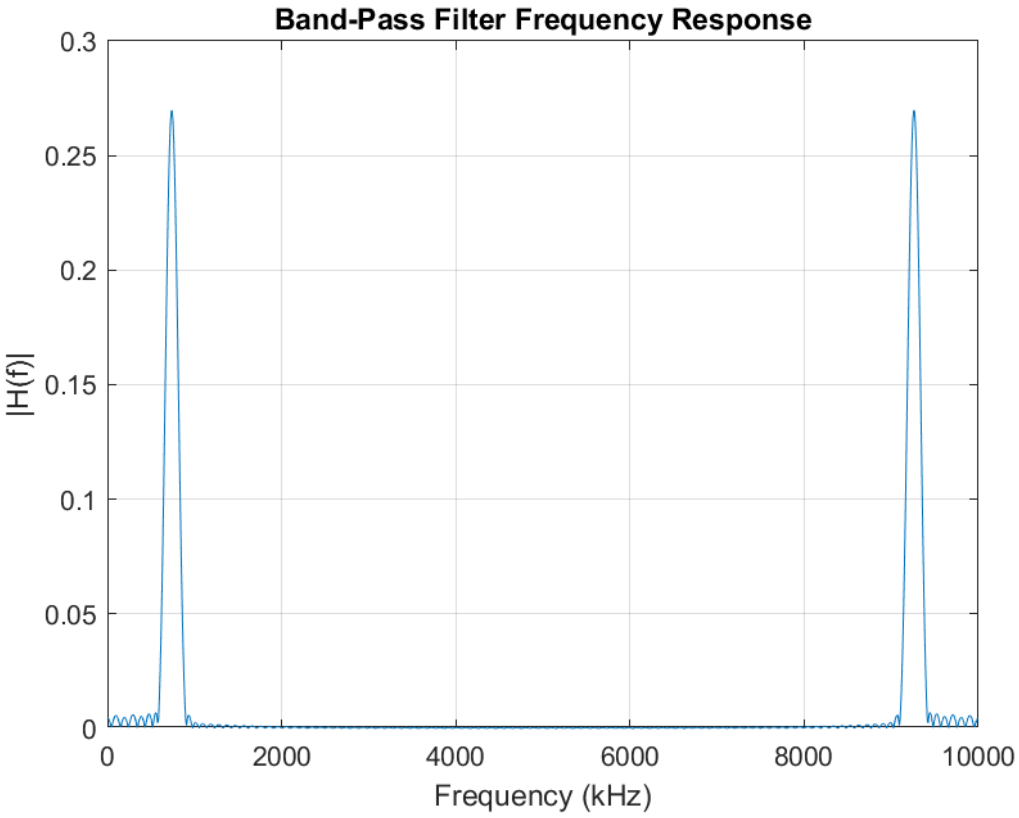
```
fc_hp = 735e3 / fs;
h_hp = sinc(2 * fc_hp * (n - (M-1)/2));
h_hp = h_hp .* w;
h_hp = -h_hp / sum(h_hp);
h_hp((M+1)/2) = h_hp((M+1)/2) + 1;
```

(c) Combine Filters into a Band-Pass Filter and Plot the Frequency Response

Octave code:

```
h_bp = conv(h_lp, h_hp, 'same');

H = abs(fft(h_bp, 1024));
f = (0:1023) * fs / 1024;
```

```
figure;
plot(f/1e3, H);
xlabel('Frequency (kHz)');
ylabel('|H(f)|');
title('Band-Pass Filter Frequency Response');
grid on;

print('bandpass_response.png', '-dpng');
```

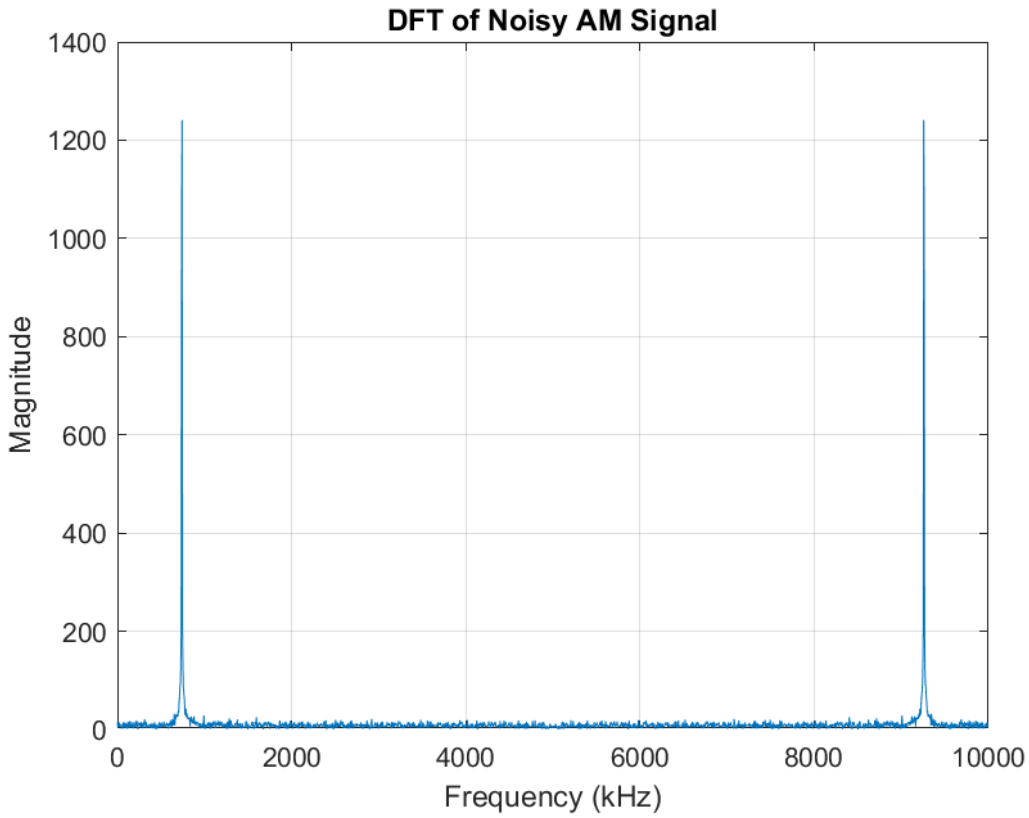


(d) Mix a 740 kHz Carrier with a 2.5 kHz Audio Signal and Add Noise

Octave code:

```
t = 0:1/fs:0.01;
audio = sin(2 * pi * 2.5e3 * t);
carrier = cos(2 * pi * fc * t);
am_signal = (1 + audio) .* carrier;
noisy_am = am_signal + 0.2 * randn(size(t));

X = abs(fft(noisy_am, 2048));
f2 = (0:2047) * fs / 2048;
```

```
figure;
plot(f2/1e3, X);
xlabel('Frequency (kHz)');
ylabel('Magnitude');
title('DFT of Noisy AM Signal');
grid on;

print('noisy_am_dft.png', '-dpng');
```

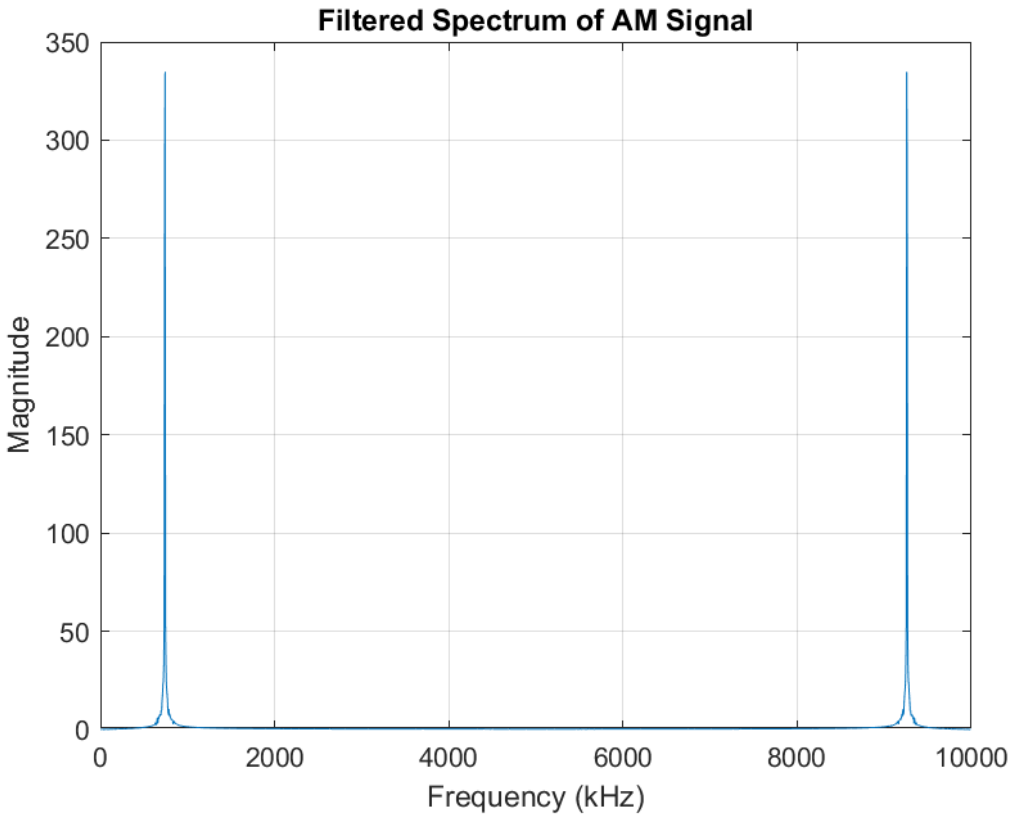


(e) Apply the Band-Pass Filter and Plot the Filtered Spectrum

Octave code:

```
filtered = conv(noisy_am, h_bp, 'same');
Xf = abs(fft(filtered, 2048));

figure;
plot(f2/1e3, Xf);
xlabel('Frequency (kHz)');
ylabel('Magnitude');
title('Filtered Spectrum of AM Signal');
```

```
grid on;

print('filtered_am_dft.png', '-dpng');
```



Question 7: FFT Convolution

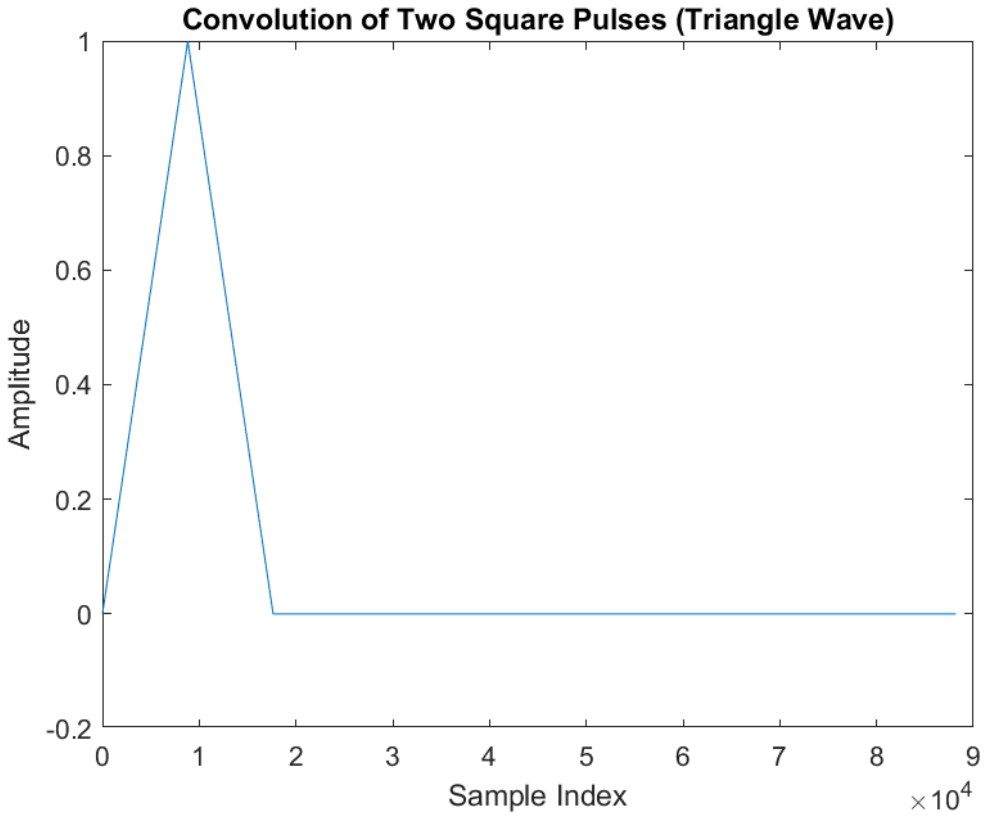
(a) Convolution of Two Square Pulses

Two square pulses of width 0.2 seconds are convolved using FFT. The result is a triangle wave.

Octave code:

```
fs = 44100;
T = 1;
N = fs * T;
t = linspace(0, T, N);

pulse_width = 0.2;
square1 = double(t < pulse_width);
square2 = double(t < pulse_width);

len = length(square1) + length(square2) - 1;
```

```
Square1 = fft(square1, len);
Square2 = fft(square2, len);
triangle = real(ifft(Square1 .* Square2));
triangle = triangle / max(triangle);

figure;
plot(triangle);
title('Convolution of Two Square Pulses (Triangle Wave)');
xlabel('Sample Index');
ylabel('Amplitude');
print('square_convolution_triangle.png', '-dpng');
```



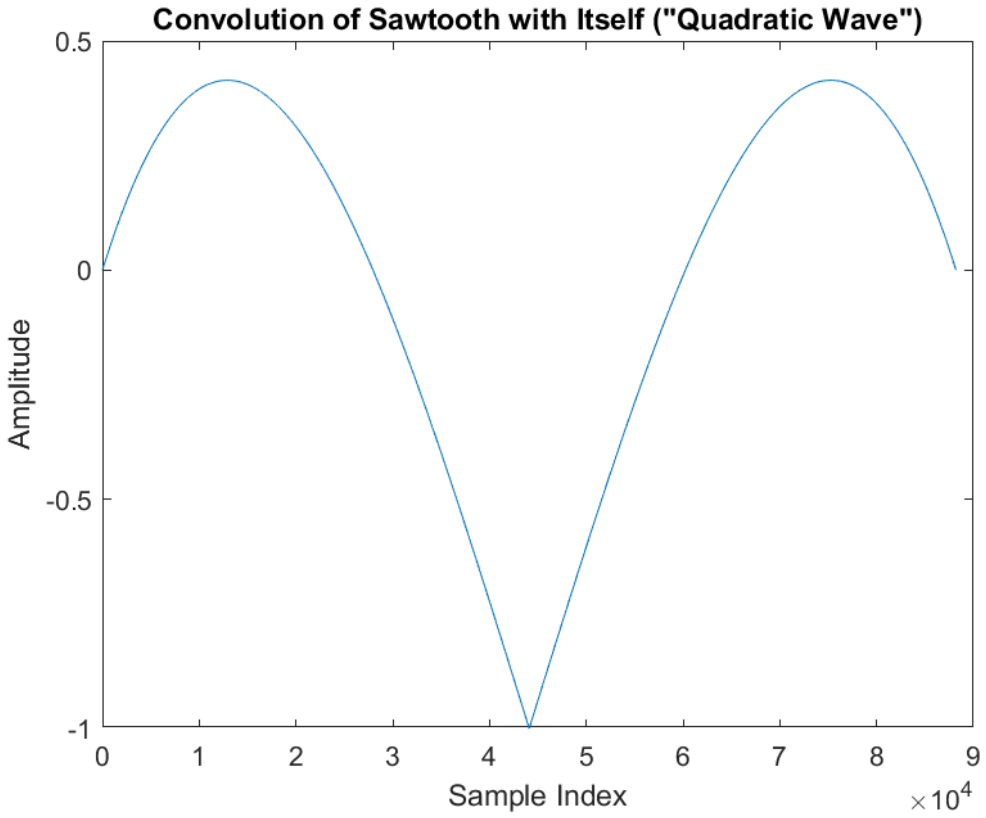
(b) Convolution of a Sawtooth Wave with Itself

We generate one period of a sawtooth wave and convolve it with itself using FFT. The result resembles a quadratic waveform.

Octave code:

```
saw = linspace(-1, 1, fs);
Saw1 = fft(saw, 2*fs-1);
Saw2 = fft(saw, 2*fs-1);
quadratic = real(ifft(Saw1 .* Saw2));
```

```
quadratic = quadratic / max(abs(quadratic));

figure;
plot(quadratic);
title('Convolution of Sawtooth with Itself ("Quadratic Wave")');
xlabel('Sample Index');
ylabel('Amplitude');
print('sawtooth_convolution_quadratic.png', '-dpng');
```



Question 8: Step Pulse Through a Recursive Low-Pass Filter

(a) Create a Step Pulse and Apply a Recursive LP Filter

We create a step signal and apply a recursive low-pass filter of the form:

$$y[n] = \alpha x[n] + (1 - \alpha)y[n - 1$$

where $\alpha = 0.05$ determines the smoothing strength.

Octave code:

```
N = 1024;
```

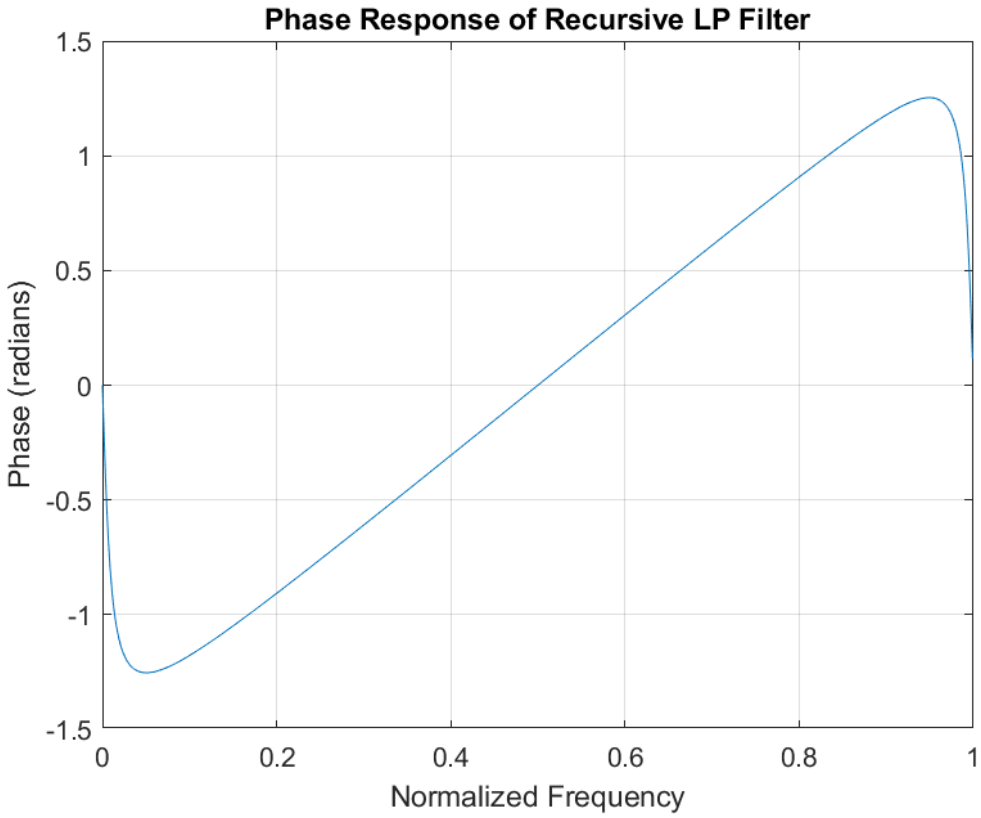
```
x = ones(1, N);
alpha = 0.05;
y = zeros(1, N);
y(1) = x(1);
for n = 2:N
    y(n) = alpha * x(n) + (1 - alpha) * y(n-1);
end
```

(b) Plot the Phase of the Recursive Filter Using freqz

We use Octave's `freqz` function to compute and unwrap the phase response of the filter across the frequency spectrum.

Octave code:

```
[h, w] = freqz(alpha, [1 -(1-alpha)], N, 'whole');
f = w / (2*pi);

figure;
plot(f, unwrap(angle(h)));
xlabel('Normalized Frequency');
ylabel('Phase (radians)');
title('Phase Response of Recursive LP Filter');
grid on;

print('recursive_lp_phase.png', '-dpng');
```



(c) Are the Results Linear or Nonlinear?

The recursive filter is a linear time-invariant (LTI) system. Even though it uses feedback (recursion), it satisfies the properties of:

- **Superposition**: Output for the sum of inputs equals the sum of outputs
- **Scaling**: Output scales proportionally with the input

Hence, the results are linear. The phase response plot confirms expected behavior for a first-order IIR low-pass filter, with a smooth and continuous phase shift across frequencies.

Question 9: Reversal and Symmetry in Recursive LP Filter

This question extends the previous analysis by reversing signals and filtering symmetrically to observe phase effects.

(a) Reverse the Step Pulse

We begin with a step pulse, but instead of an all-ones signal (which gives meaningless phase), we use a short pulse (width 10 samples) to provide frequency content.

Octave code:

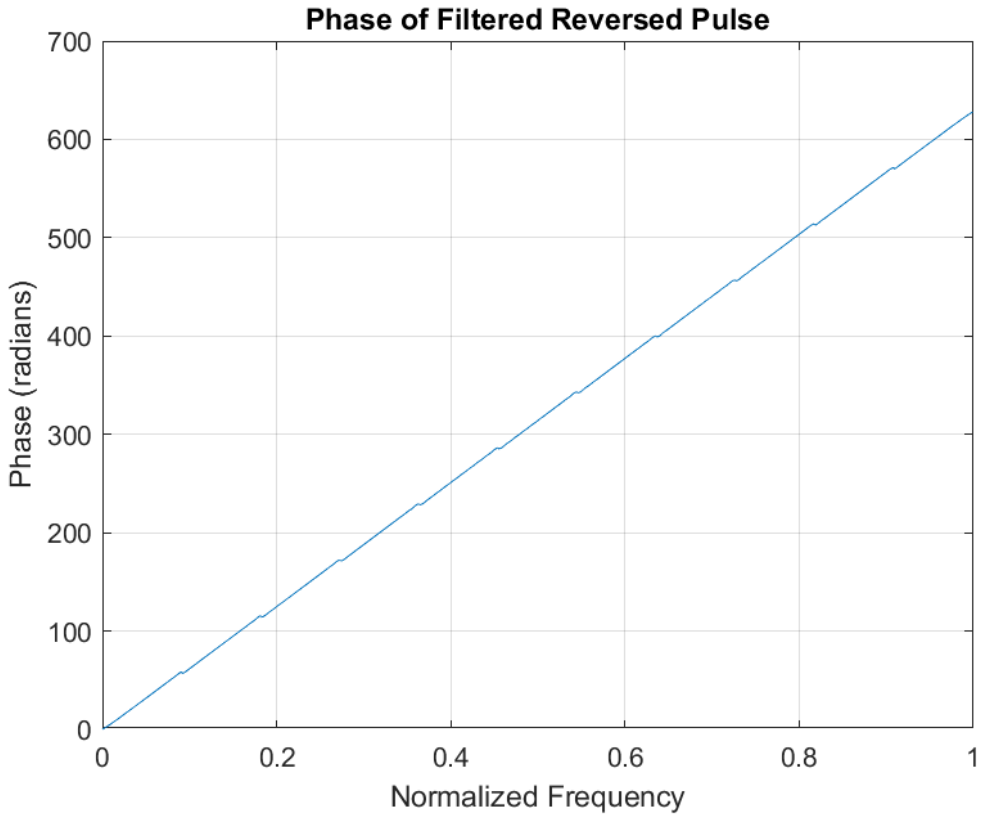
```
N = 1024;
alpha = 0.05;

% Create short pulse input
x = zeros(1, N);
x(100:110) = 1;

% Reverse the input pulse
x_rev = fliplr(x);
```

(b) Filter the Reversed Pulse and Plot Phase

We run the reversed pulse through the recursive low-pass filter and compute the phase of its FFT. This shows how filtering affects the phase of a reversed input.

Octave code:

```
% Filter reversed pulse
y_rev = zeros(1, N);
y_rev(1) = x_rev(1);
for n = 2:N
    y_rev(n) = alpha * x_rev(n) + (1 - alpha) * y_rev(n-1);
end

% Compute FFT and unwrap phase
Yrev = fft(y_rev);
f = (0:N-1)/N;
phase_rev = unwrap(angle(Yrev));

% Plot
figure;
plot(f, phase_rev);
xlabel('Normalized Frequency');
ylabel('Phase (radians)');
title('Phase of Filtered Reversed Pulse');
grid on;
print('reversed_input_phase.png', '-dpng');
```



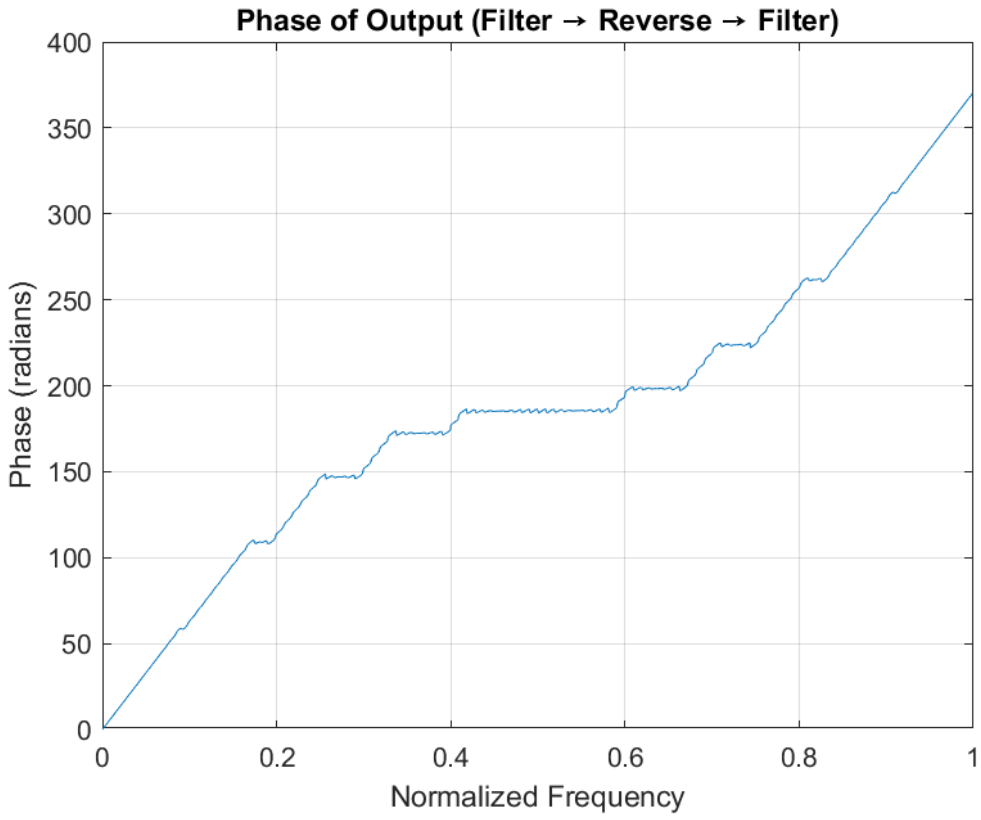
(c) Filter → Reverse → Filter Again and Plot Phase

We now apply the filter to the original pulse, reverse the filtered output, and filter again. This symmetry should reduce or balance the phase distortion introduced by the filter.

Octave code:

```
% First filtering of original pulse
y1 = zeros(1, N);
y1(1) = x(1);
for n = 2:N
    y1(n) = alpha * x(n) + (1 - alpha) * y1(n-1);
end

% Reverse the filtered output
y1_rev = fliplr(y1);

% Second filtering
y2 = zeros(1, N);
y2(1) = y1_rev(1);
for n = 2:N
    y2(n) = alpha * y1_rev(n) + (1 - alpha) * y2(n-1);
end
```

```
% Compute FFT and unwrap phase
Y2 = fft(y2);
phase_y2 = unwrap(angle(Y2));

% Plot
figure;
plot(f, phase_y2);
xlabel('Normalized Frequency');
ylabel('Phase (radians)');
title('Phase of Output (Filter → Reverse → Filter)');
grid on;
print('double_filter_symmetry_phase.png', '-dpng');
```



Conclusion

The first phase plot in part (b) shows a linearly increasing phase due to the asymmetry introduced by recursive filtering. However, after performing the symmetric operation of filtering, reversing, and filtering again, the phase response remains relatively linear with reduced asymmetry. This confirms the result hinted at in Figure 19-8 of the textbook, that filtering a signal in a forward and then reverse direction effectively cancels out phase distortion and leads to linear-phase-like behavior.

Question 10 (Bonus): Chirping Signals

Overview

We simulate a linear chirp signal with time-varying frequency:

$$f(t) = f_0 - \beta t \qquad s(t) = A\cos(2\pi f(t)t)$$

We analyze the signal by delaying it, mixing it with itself, and applying a low-pass filter to isolate the beat frequency. This technique is widely used in radar and sonar DSP.

(a) Mix Chirp and Delayed Chirp Signal

We define a chirp with $f_0 = 5\,\text{MHz}$, $\beta = 2\,\text{MHz}/\mu s$, and delay $t_d = 0.5\,\mu s$. Mixing $s(t) \cdot s(t - t_d)$ produces two frequencies: a high-frequency sum, and a low-frequency beat tone βt_d.

Octave code:

```
fs = 100e6;
dt = 1/fs;
T = 5e-6;
t = 0:dt:T-dt;
N = length(t);

f0 = 5e6;
beta = 2e6;
td = 0.5e-6;

% Chirp generation
f_t = f0 - beta * t;
phi = 2 * pi * (f0 * t - 0.5 * beta * t.^2);
s = cos(phi);
s_delayed = cos(2 * pi * (f0 * (t - td) - 0.5 * beta * (t - td).^2));

% Mix the chirp and delayed chirp
mixed = s .* s_delayed;
```

(b) Add Noise and Filter to Isolate Beat Frequency

We add white noise to the mixed signal and apply a low-pass FIR filter with a cutoff near 2 MHz to isolate the beat tone.

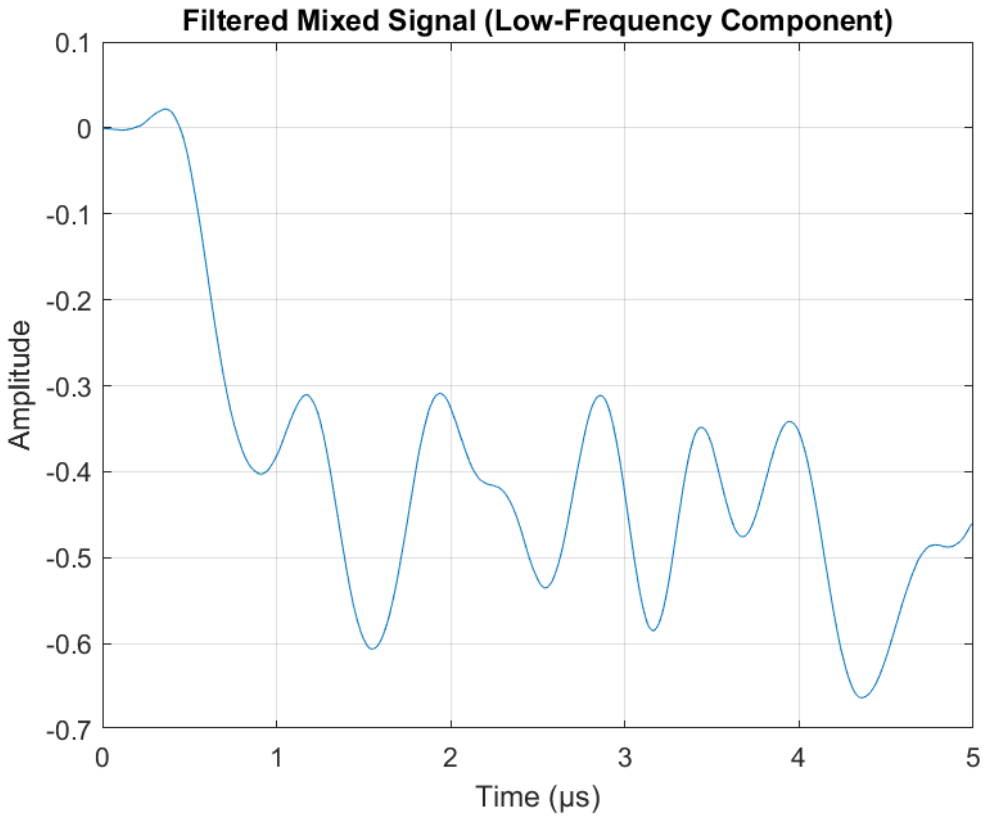
Octave code:

```
% Add noise
mixed_noisy = mixed + 0.5 * randn(size(mixed));

% Design low-pass filter (cutoff ~2 MHz)
```

```
fc = 2e6 / (fs/2);
b = fir1(128, fc);
filtered = filter(b, 1, mixed_noisy);
```

Expected beat frequency:

$$f_{\text{beat}} = \beta t_d = 2\,\text{MHz/s} \cdot 0.5\,\mu s = 1\,\text{MHz}$$



Filtered mixed signal showing beat waveform

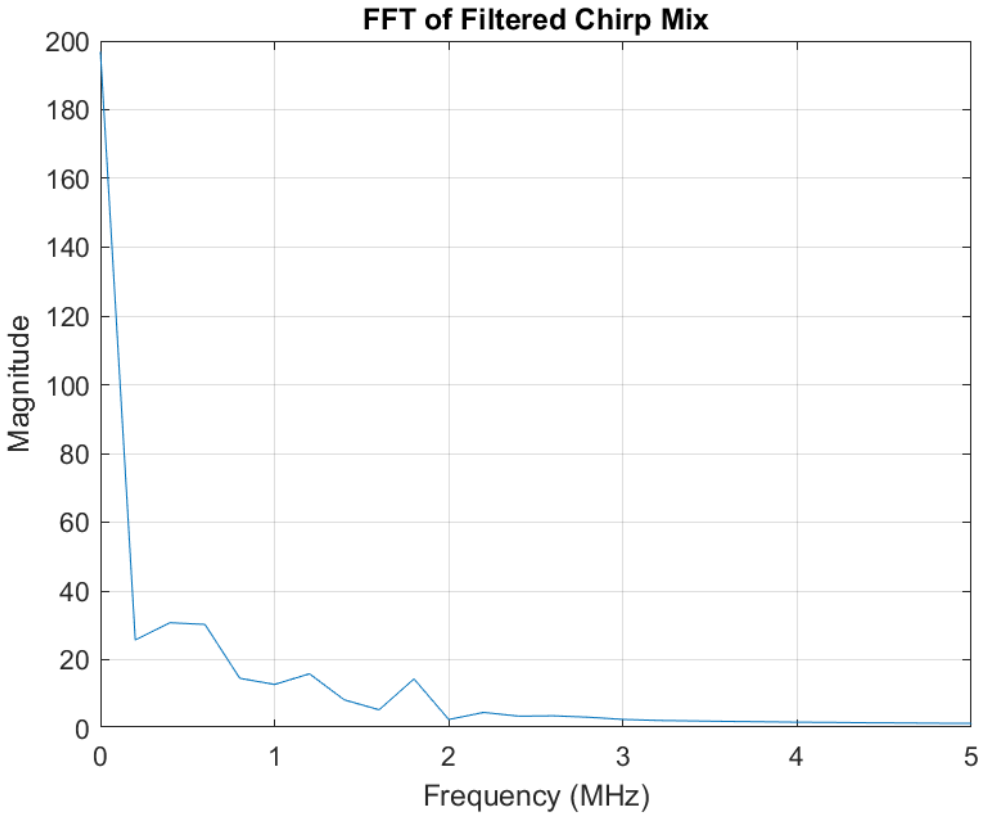
(c) Verify Beat Frequency Using FFT

We compute the FFT of the filtered output to confirm that the beat frequency is approximately 1 MHz.

Octave code:

```
F = abs(fft(filtered));
f = (0:N-1) * fs / N;

figure;
plot(f/1e6, F);
xlim([0 5]);
xlabel('Frequency (MHz)');
ylabel('Magnitude');
```

```
title('FFT of Filtered Chirp Mix');
grid on;
print('chirp_filtered_fft.png', '-dpng');
```



FFT of filtered signal — Peak observed near 1 MHz

Conclusion

We successfully verified the theoretical result:

$$f_{\text{beat}} = \beta t_d = 1\,\text{MHz}$$

Both the time-domain and frequency-domain plots confirm the presence of a clean low-frequency tone near 1 MHz after mixing and filtering, despite added noise. This technique is widely used in chirp radar systems to extract range or velocity from reflected signals.