Student Name: Muhammad Mahdi Amirpour
Student ID: 40003033

PART 1: Basic Audio Manipulation

Code Description
This part of the script involves loading an audio file, plotting its waveform, doubling the sampling rate, and saving the modified audio file.

Detailed Steps

Step 1: Load the Audio File
The audio file `voice.wav` is loaded into the MATLAB workspace.

Code:
```matlab
[audio, Fs] = audioread('voice.wav');
```

Step 2: Plot the Audio Signal
The waveform of the audio signal is plotted. The code handles both mono and stereo audio files.

Code:
```matlab
figure;
if size(audio, 2) == 2
    % Stereo audio
    subplot(2, 1, 1);
    plot(audio(:, 1));
    title('Left Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');

    subplot(2, 1, 2);
    plot(audio(:, 2));
    title('Right Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');
else
    % Mono audio
    plot(audio);
    title('Mono Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');
end
```

Step 3: Double the Sampling Rate
The sampling rate of the audio file is doubled to make the audio play faster and at a higher pitch.

Code:
```matlab
Fs_2 = Fs * 2;
```

```

### Step 4: Save the Audio with Doubled Sampling Rate
The modified audio data is saved to a new file `out_doubled.wav`.

#### Code:
```matlab
audiowrite('out_doubled.wav', audio, Fs_2);
```

Explanation
The script ends with a message indicating that the audio data has been saved with a doubled sampling rate, explaining that this modification will make the audio play faster and at a higher pitch.

Code:
```matlab
disp('The audio data has been saved with a doubled sampling rate. This makes the audio play faster and at a higher pitch.');
```

Explanation
In this part of the code, we process an audio file to manipulate its playback speed and pitch by altering the sampling rate.

Key Points:
- **Loading Audio**: The `audioread` function loads the audio file into the workspace, along with its sampling rate.
- **Plotting Waveform**: The audio signal is plotted to visualize its amplitude over time. The script checks if the audio is stereo or mono and plots accordingly.
- **Doubling Sampling Rate**: The sampling rate is doubled. In digital audio, increasing the sampling rate without altering the data makes the audio play back at twice the speed, effectively raising its pitch.
- **Saving Modified Audio**: The modified audio is saved using `audiowrite`. This ensures the changes are preserved and can be played back for verification.

By following these steps, we can manipulate the speed and pitch of an audio file, demonstrating basic audio processing techniques in MATLAB.

PART 2: Applying an Exponential Decay to the Audio Signal

Code Description
This part of the script involves loading an audio file, creating a decreasing exponential signal, applying this exponential decay to the audio signal, plotting the modified audio signal, and saving the modified audio file.

Detailed Steps

Step 1: Load the Audio File
```

The audio file `voice.wav` is loaded into the MATLAB workspace.
Code:
```matlab
[audio, Fs] = audioread('voice.wav');
```

Step 2: Create a Decreasing Exponential Signal
A decreasing exponential signal is created to apply an exponential decay effect to the audio signal. The signal is generated based on the length of the audio file and its sampling frequency.

Code:
```matlab
n = length(audio);
t = (0:n-1)' / Fs;
exp_signal = exp(-t);
```

Step 3: Ensure the Exponential Signal Matches Both Channels
If the audio file is stereo, the exponential signal needs to be matched for both channels.

Code:
```matlab
if size(audio, 2) == 2
    exp_signal = [exp_signal, exp_signal];
end
```

Step 4: Multiply the Audio Signal by the Exponential Signal
The audio signal is multiplied by the exponential signal to apply the decay effect.

Code:
```matlab
modified_audio = audio .* exp_signal;
```
Step 5: Plot the Modified Audio Signal
The modified audio signal is plotted. The script handles both mono and stereo audio files.
Code:
```matlab
figure;
if size(modified_audio, 2) == 2
    % Stereo audio
    subplot(2, 1, 1);
    plot(modified_audio(:, 1));
    title('Modified Left Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');

    subplot(2, 1, 2);
    plot(modified_audio(:, 2));
    title('Modified Right Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');
else
    % Mono audio
    plot(modified_audio);
```

```
    title('Modified Mono Channel');
    xlabel('Sample Index');
    ylabel('Amplitude');
end
```

Step 6: Save the Modified Audio Signal
The modified audio data is saved to a new file `modified_audio.wav`.

Code:
```matlab
audiowrite('modified_audio.wav', modified_audio, Fs);
```

Explanation
The script ends with a message indicating that the modified audio data has been saved with the volume decreasing exponentially over time.

Code:
```matlab
disp('The modified audio data has been saved with the volume decreasing exponentially over time.');
```

Explanation
In this part of the code, we process an audio file to apply an exponential decay effect, making the volume gradually decrease over time.

Key Points:
- **Loading Audio**: The `audioread` function loads the audio file into the workspace.
- **Creating Exponential Signal**: An exponential signal is generated based on the length and sampling frequency of the audio. This signal will be used to modulate the audio signal.
- **Matching Channels**: For stereo audio files, the exponential signal is duplicated to match both channels.
- **Applying Exponential Decay**: The audio signal is multiplied by the exponential signal, applying the decay effect.
- **Plotting Modified Signal**: The modified audio signal is plotted to visualize the effect.
- **Saving Modified Audio**: The modified audio is saved using `audiowrite`, ensuring the changes are preserved.

By following these steps, we can apply an exponential decay to an audio file, demonstrating basic audio signal processing techniques in MATLAB.

PART 3: Adding Echo to the Audio Signal

Code Description
This part of the script involves adding echo to an audio signal by convolving it with an impulse response. The steps include loading the audio file, defining the echo parameters, creating the impulse response, convolving the audio signal with this response, normalizing the modified audio to avoid clipping, and saving the result.

Detailed Steps
Step 1: Load the Audio File
The audio file `voice.wav` is loaded into the MATLAB workspace.

Code:
```matlab
[audio, Fs] = audioread('voice.wav');
```

Step 2: Define Echo Parameters
Echo parameters such as delay and intensity for the first and second echoes are defined.

Code:
```matlab
first_echo_delay = 1.0; % seconds
second_echo_delay = 2.0; % seconds
first_echo_intensity = 0.8;
second_echo_intensity = 0.5;
```

Step 3: Create the Impulse Response
An impulse response is created to simulate the echo effect. This involves defining the length of the impulse response and setting the appropriate values for the original signal and the echoes.

Code:
```matlab
n = length(audio);
first_echo_samples = round(first_echo_delay * Fs);
second_echo_samples = round(second_echo_delay * Fs);

% Impulse response length
h_length = max([n, first_echo_samples + n, second_echo_samples + n]);
impulse_response = zeros(h_length, 1);
impulse_response(1) = 1; % Original signal
impulse_response(first_echo_samples + 1) = first_echo_intensity; % First echo
impulse_response(second_echo_samples + 1) = second_echo_intensity; % Second echo
```

Step 4: Convolve the Audio with the Impulse Response
The audio signal is convolved with the impulse response to add the echo effect. For stereo audio files, each channel is processed separately.

Code:
```matlab
% If stereo, process each channel separately
if size(audio, 2) == 2
    modified_audio_left = conv(audio(:, 1), impulse_response);
    modified_audio_right = conv(audio(:, 2), impulse_response);
    modified_audio = [modified_audio_left(1:n), modified_audio_right(1:n)];
else
```

```
    modified_audio = conv(audio, impulse_response);
    modified_audio = modified_audio(1:n); % Ensure the length matches the original audio
end
```

Step 5: Normalize the Modified Audio to Avoid Clipping
The modified audio signal is normalized to ensure it does not clip.

Code:
```matlab
modified_audio = modified_audio / max(abs(modified_audio));
```

Step 6: Save the Modified Audio
The modified audio data is saved to a new file `modified_audio_with_echo.wav`.

Code:
```matlab
audiowrite('modified_audio_with_echo.wav', modified_audio, Fs);
```

Explanation
The script ends with a message indicating that the audio data has been convolved with the echo filter and saved.

Code:
```matlab
disp('The audio data has been convolved with the echo filter and saved to "modified_audio_with_echo.wav".');
```

Explanation
In this part of the code, we add an echo effect to an audio file by creating and applying an impulse response.

Key Points:
- **Loading Audio**: The `audioread` function loads the audio file into the workspace.
- **Defining Echo Parameters**: Parameters for the delay and intensity of the echoes are set.
- **Creating Impulse Response**: An impulse response is constructed to simulate the echo effect, with specific samples representing the delayed echoes.
- **Convolution**: The audio signal is convolved with the impulse response to apply the echo. For stereo files, each channel is processed separately.
- **Normalization**: The modified audio signal is normalized to prevent clipping.
- **Saving Modified Audio**: The modified audio is saved using `audiowrite`, ensuring the changes are preserved.

By following these steps, we can add an echo effect to an audio file, demonstrating convolution and audio processing techniques in MATLAB.

PART 4: Convolution with Impulse Responses

Code Description
This part of the script involves applying convolution of an audio file with impulse responses (IRs) of different environments (concert hall and iron bucket) to simulate the respective acoustic effects. The modified audio files are then saved.

Detailed Steps

Step 1: Load the Impulse Response for the Concert Hall
The impulse response for the concert hall is loaded and plotted.

Code:
```matlab
[ir_concert_hall, Fs_ir_concert_hall] = audioread('concert_hall_IR.wav');

% Ensure the impulse response is a column vector
ir_concert_hall = ir_concert_hall(:);

% Plot the impulse response for the concert hall
figure;
plot(ir_concert_hall);
title('Impulse Response - Concert Hall');
xlabel('Sample Index');
ylabel('Amplitude');
```

Step 2: Convolve the Audio with the Concert Hall Impulse Response
The audio is convolved with the concert hall impulse response. The result is normalized to avoid clipping and saved to a new file.

Code:
```matlab
if size(audio, 2) == 2
    % Stereo audio
    modified_audio_concert_hall_left = conv(audio(:, 1), ir_concert_hall, 'same');
    modified_audio_concert_hall_right = conv(audio(:, 2), ir_concert_hall, 'same');
    modified_audio_concert_hall = [modified_audio_concert_hall_left, 
modified_audio_concert_hall_right];
else
    % Mono audio
    modified_audio_concert_hall = conv(audio, ir_concert_hall, 'same');
end

% Normalize the modified audio to avoid clipping
modified_audio_concert_hall = modified_audio_concert_hall / 
max(abs(modified_audio_concert_hall), [], 'all');

% Save the modified audio for the concert hall impulse response
audiowrite('modified_audio_concert_hall.wav', modified_audio_concert_hall, Fs);
```

Step 3: Load the Impulse Response for the Iron Bucket
The impulse response for the iron bucket is loaded and plotted.

Code:
```matlab
[ir_iron_bucket, Fs_ir_iron_bucket] = audioread('iron_bucket_IR.wav');

% Ensure the impulse response is a column vector
ir_iron_bucket = ir_iron_bucket(:);

% Plot the impulse response for the iron bucket
figure;
plot(ir_iron_bucket);
title('Impulse Response - Iron Bucket');
xlabel('Sample Index');
ylabel('Amplitude');
```

Step 4: Convolve the Audio with the Iron Bucket Impulse Response
The audio is convolved with the iron bucket impulse response. The result is normalized to avoid clipping and saved to a new file.

Code:
```matlab
if size(audio, 2) == 2
    % Stereo audio
    modified_audio_iron_bucket_left = conv(audio(:, 1), ir_iron_bucket, 'same');
    modified_audio_iron_bucket_right = conv(audio(:, 2), ir_iron_bucket, 'same');
    modified_audio_iron_bucket = [modified_audio_iron_bucket_left, 
modified_audio_iron_bucket_right];
else
    % Mono audio
    modified_audio_iron_bucket = conv(audio, ir_iron_bucket, 'same');
end

% Normalize the modified audio to avoid clipping
modified_audio_iron_bucket = modified_audio_iron_bucket / 
max(abs(modified_audio_iron_bucket), [], 'all');

% Save the modified audio for the iron bucket impulse response
audiowrite('modified_audio_iron_bucket.wav', modified_audio_iron_bucket, Fs);

disp('The audio has been convolved with both impulse responses and saved to 
respective files.');
```

Explanation
In this part of the code, we process an audio file to simulate different acoustic environments by convolving it with the impulse responses of a concert hall and an iron bucket. The convolution operation applies the effect of the impulse response to the audio, making it sound as if it were played in the specified environment.

Key Points:
- **Loading Impulse Responses**: We load the IR files using `audioread` and plot them to visualize the impulse responses.
- **Convolution**: The `conv` function is used to convolve the audio signal with the impulse response. This operation effectively applies the acoustic characteristics of the environment to the audio.

- **Normalization**: The resulting audio signal is normalized to prevent clipping, ensuring the audio stays within the valid range.
- **Saving**: The modified audio files are saved using `audiowrite`, allowing us to listen to the effects.

By following these steps, we can effectively simulate how the audio would sound in different environments, providing an interesting and immersive listening experience.

PART 5: Fourier Series Analysis and Reconstruction

Code Description
This part of the script involves defining a periodic function, computing its Fourier series coefficients using the trapezoidal rule, and plotting the Fourier series approximation. Additionally, it explores the theoretical relation between old and new Fourier coefficients after shifting the function.

Detailed Steps

Step 1: Define the Original Function and Parameters
The original function \(x(t) \) is defined as a repetitive step function with a period \(T = 4 \). The shifted function \(x_{\text{shifted}}(t) \) is also defined.

Code:
```matlab
T = 4;  % Period of the function
x = @(t) mod(t, T) < T / 4 | mod(t, T) >= 3 * T / 4;
x_shifted = @(t) x(t - T/4);
```

Step 2: Define the Interval for Integration
The lower limit (\(ll \)) and upper limit (\(ul \)) for the integration interval are defined based on the period of the function.

Code:
```matlab
ll = -T / 2;
ul = T / 2;
```

Step 3: Define the Range of Harmonics
Different ranges of harmonics (\(k \)) are defined to compute the Fourier series coefficients.
Code:
```matlab
k_ranges = {-20:20, -100:100, -500:500};
```

Step 4: Initialize Arrays to Store Results
Arrays are initialized to store the Fourier series results for different ranges of harmonics.

Code:

```matlab
fourier_series_results = cell(size(k_ranges));
t = linspace(-2*T, 2*T, 10000);  % Evaluate over a larger range for better
visualization
```

Step 5: Calculate Fourier Coefficients
The Fourier coefficients are calculated using the trapezoidal rule. A custom function `trapezoidal_rule` is defined to perform numerical integration.

Code:
```matlab
calc_fourier_coeffs = @(k_range) arrayfun(@(k) trapezoidal_rule(@(t) x(t) .* exp(-
1i * k * 2 * pi * t / T), ll, ul, 100000) / T, k_range);

for idx = 1:length(k_ranges)
    k_range = k_ranges{idx};
    coeffs = calc_fourier_coeffs(k_range);
    fourier_series = zeros(size(t));

    for k_idx = 1:length(k_range)
        k = k_range(k_idx);
        fourier_series = fourier_series + coeffs(k_idx) * exp(1i * k * 2 * pi * t /
T - 1i * k * pi / 2);
    end

    fourier_series_results{idx} = real(fourier_series);  % Store the real part
end
```

Step 6: Plot Results for Different Harmonic Ranges
The Fourier series approximation is plotted for different ranges of harmonics and compared to the shifted function.
Code:
```matlab
figure;
for idx = 1:length(k_ranges)
    subplot(length(k_ranges), 1, idx);
    plot(t, fourier_series_results{idx});
    hold on;
    plot(t, x_shifted(t), 'r');
    title(sprintf('Fourier Series with %d Harmonics', max(abs(k_ranges{idx}))));
    legend('Fourier Series', 'Shifted Function');
    hold off;
end
```

Step 7: Define the New Function
A new function \(x_{\text{new}}(t) \) is defined and its Fourier series coefficients are calculated.

Code:
```matlab
x_new = @(t) mod(t, T) < T / 4 | (mod(t, T) >= T / 2 & mod(t, T) < 3 * T / 4);

k_range = -500:500;
```

```
coeffs_new = arrayfun(@(k) trapezoidal_rule(@(t) x_new(t - T/4) .* exp(-1i * k * 2 * pi * t / T), ll, ul, 100000) / T, k_range);
```

Step 8: Theoretical Relation Between Old and New Coefficients
The theoretical relation between the old coefficients (a_k) and the new coefficients (c_k) is explored. The coefficients a_3 and c_3 are verified.

Code:
```matlab
ak_theoretical = @(k) (1 / pi) * (sin(k * pi / 4) - sin(3 * k * pi / 4)) / k;
ck = @(k) (1 / (1i * pi * k)) * (exp(-1i * k * pi / 4) - exp(1i * k * 3 * pi / 4));

a3_idx = find(k_range == 3);
a3 = coeffs_new(a3_idx);
c3 = ck(3);

fprintf('a3: %f\n', a3);
fprintf('c3: %f\n', c3);
fprintf('a3_theoretical: %f\n', ak_theoretical(3));
```

Step 9: Plot the Shifted Function and Its Fourier Series Approximation
The shifted function and its Fourier series approximation are plotted for comparison.

Code:
```matlab
figure;
plot(t, x_shifted(t), 'r', 'LineWidth', 2);
hold on;
fourier_series = zeros(size(t));
for k_idx = 1:length(k_range)
    k = k_range(k_idx);
    fourier_series = fourier_series + coeffs(k_idx) * exp(1i * k * 2 * pi * t / T - 1i * k * pi / 2);
end
fourier_series = real(fourier_series);
plot(t, fourier_series, 'b', 'LineWidth', 2);
legend('Shifted Function', 'Fourier Series Approximation', 'Location', 'Best');
xlabel('t');
ylabel('x(t)');
title('Shifted Function and Fourier Series Approximation');
```

Explanation
In this part of the code, we analyze and reconstruct a periodic function using Fourier series, exploring the theoretical relations between Fourier coefficients after shifting the function.

Key Points:
- **Original and Shifted Functions**: The original periodic step function and its shifted version are defined.

- **Integration Interval**: The integration interval is set based on the function's period.
- **Harmonic Ranges**: Different harmonic ranges are used to compute the Fourier series coefficients.
- **Trapezoidal Rule**: Fourier coefficients are computed using numerical integration (trapezoidal rule).
- **Fourier Series Approximation**: The Fourier series approximation is plotted for different harmonic ranges.
- **New Function**: A new function is defined and its Fourier series coefficients are calculated.
- **Theoretical Relation**: The theoretical relation between old and new Fourier coefficients is verified.
- **Plotting**: The shifted function and its Fourier series approximation are plotted for comparison.

By following these steps, we can analyze and reconstruct periodic functions using Fourier series, demonstrating the application of numerical integration and theoretical analysis in MATLAB.

PART 6: Creating 360° Audio Effect
Code Description
This part of the script creates a 360° audio effect by applying a phase shift to the original audio channels. The audio effect is achieved by modulating the audio signal with sinusoidal waves that simulate a rotational effect, giving the impression of sound moving around the listener.

Detailed Steps

Step 1: Load the Original Audio File
The original audio file (`original_audio.wav`) is loaded into the MATLAB workspace, and the sampling frequency (`Fs`) is obtained.

Code:
```matlab
[original_audio, Fs] = audioread('original_audio.wav');
```

Step 2: Define Rotation Speed and Time Vector
The rotation speed (in radians per second) is defined, and a time vector (`t`) is created based on the length of the original audio and the sampling frequency.

Code:
```matlab
rotation_speed = 0.1; % Adjust this value to control the speed of rotation
t = (0:length(original_audio)-1) / Fs;
```

Step 3: Define Sinusoidal Waves with Phase Shift for Left and Right Channels
Sinusoidal waves are created for both the left and right audio channels, with the left channel's phase shifted positively and the right channel's phase shifted negatively.

Code:
```matlab
phase_shift_left = 2*pi*rotation_speed*t;
phase_shift_right = -4*pi*rotation_speed*t; % Opposite direction for right channel

sin_wave_left = sin(phase_shift_left);
```

```
sin_wave_right = sin(phase_shift_right);
```

Step 4: Apply Phase Shift to Original Audio for Left and Right Channels
The original audio signals for the left and right channels are modulated with the respective sinusoidal waves to apply the phase shift.

Code:
```matlab
modified_audio_left = original_audio(:, 1) .* sin_wave_left(:);
modified_audio_right = original_audio(:, 2) .* sin_wave_right(:);
```

Step 5: Combine Channels to Create Stereo Audio
The modified left and right audio channels are combined to create the final stereo audio signal.

Code:
```matlab
modified_audio = [modified_audio_left, modified_audio_right];
```

Step 6: Save the Modified Audio
The modified audio with the 360° effect is saved to a new file (`360_audio.wav`).

Code:
```matlab
audiowrite('360_audio.wav', modified_audio, Fs);
```

Display a Message
A message is displayed to confirm the successful creation of the 360° audio file.

Code:
```matlab
disp('360° audio file has been created successfully.');
```

Explanation
In this part of the code, we create a 360° audio effect by modulating the original audio channels with phase-shifted sinusoidal waves. This effect simulates the sound rotating around the listener, enhancing the spatial audio experience.

Key Points:
- **Load Original Audio**: The original audio file is loaded, and the sampling frequency is obtained.
- **Define Rotation Speed and Time Vector**: A rotation speed is defined, and a time vector is created based on the audio length and sampling frequency.
- **Create Sinusoidal Waves**: Sinusoidal waves with phase shifts are created for both left and right audio channels.
- **Apply Phase Shift**: The original audio signals are modulated with the sinusoidal waves to apply the phase shift.

- **Combine Channels**: The modified left and right audio channels are combined to create the final stereo audio signal.
- **Save Modified Audio**: The modified audio with the 360° effect is saved to a new file.

By following these steps, we can create an immersive 360° audio effect that enhances the spatial perception of sound for the listener.