Lab1

January 21, 2025

Tufan Pandey

1 Theory

2 Signal Processing with Python

Sound waves are fundamental to signal processing and can be represented as signals for analysis and manipulation. This study focuses on the generation, manipulation, and analysis of waveforms using Python, emphasizing key concepts such as signal synthesis, frequency manipulation, and noise reduction. Python's robust libraries, including **numpy**, **matplotlib**, and **scipy**, are leveraged to streamline these processes.

2.1 Signal Generation and Synthesis

Sine waves, represented mathematically as: y(t)=Asin(2 ft+) were synthesized with specified frequencies f, amplitudes A , and phases . Square waves, characterized by their sharp transitions, were generated using Python's signal generation tools. These waveforms served as the building blocks for experiments in additive synthesis, where sine and square waves were combined to create complex audio signals. This technique is widely used in music production and telecommunications.

2.2 Frequency Manipulation

Frequency manipulation techniques, such as pitch shifting, were demonstrated by altering waveform parameters. For instance, doubling the frequency f resulted in an octave increase, while halving it lowered the pitch by an octave. These principles are critical in audio engineering applications, including synthesizers and sound effects.

2.3 Noise Introduction and Filtering

To simulate real-world scenarios, high-frequency noise was added to clean signals, creating noisy waveforms. A low-pass Butterworth filter, designed using the transfer function: H(s)= 1/(1+(s/ c)^2n) was implemented to remove noise. Here, c is the cutoff frequency, and n is the filter order. This approach demonstrated the effectiveness of filtering in restoring signal quality. Visualizations of clean, noisy, and filtered signals highlighted the transformative impact of noise reduction.

2.4 Applications and Insights

These experiments underscore the foundational importance of signal generation, frequency manipulation, and noise reduction across domains such as audio engineering, sound design, and telecom-

munications. Python's capabilities as a versatile tool for signal processing allow for the seamless integration of theoretical concepts with practical applications, enhancing both technical understanding and skill development.

The use of Python libraries such as **numpy** for mathematical operations, **matplotlib** for waveform visualization, and **scipy** for filter design ensures that these tasks are executed efficiently. These methods are particularly useful in creating and analyzing audio signals, developing music production tools, and designing communication systems.

2.5 Looking at the sampling rate of audio file

[3]:
```
import audiofile
signal, sampling_rate = audiofile.read("/Users/Tufan File/vdo/Happy Birthday To␣
 ↪You Ji - Funny Hindi Birthday Song (Part 1) - Funzoa Mimi Teddy, Krsna Solo.
 ↪mp3")
sampling_rate,signal
```

[3]:
```
(44100,
 array([[ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00, …,
          3.0823270e-05,  6.5984117e-05, -8.1392966e-05],
        [ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00, …,
          7.9334815e-05,  4.9850347e-05,  1.4509988e-04]],
       shape=(2, 8083456), dtype=float32))
```

[5]:
```
len(signal[0])
```

[5]:
```
8083456
```

2.6 Playing the audio file

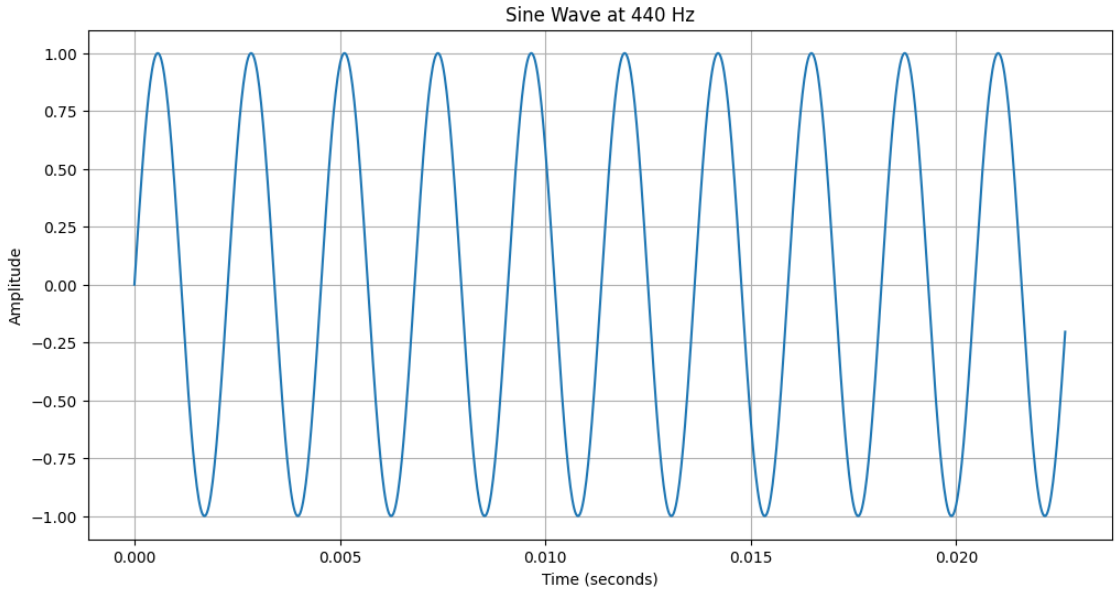
[6]:
```
import numpy as np
import audiofile as af
import sounddevice as sd

# Path to your audio file
audio_path = "/Users/Tufan File/vdo/Happy Birthday To You Ji - Funny Hindi␣
 ↪Birthday Song (Part 1) - Funzoa Mimi Teddy, Krsna Solo.mp3"

# Read the audio file
signal, sampling_rate = af.read(audio_path)
print(f"Original Signal Shape: {signal.shape}")
print(f"Sampling Rate: {sampling_rate} Hz")

# Handle multi-channel audio by converting it to mono
if len(signal.shape) > 1:  # If more than one channel
    signal = np.mean(signal, axis=1)  # Convert to mono
    print(f"Converted to Mono Signal Shape: {signal.shape}")
```

```
# Play the audio file
print("Playing audio...")
sd.play(signal, samplerate=sampling_rate)
sd.wait()  # Wait until playback finishes
print("Audio playback finished.")
```

```
Original Signal Shape: (2, 8083456)
Sampling Rate: 44100 Hz
Converted to Mono Signal Shape: (2,)
Playing audio…
Audio playback finished.
```

2.7 Plot the 440 sine wave

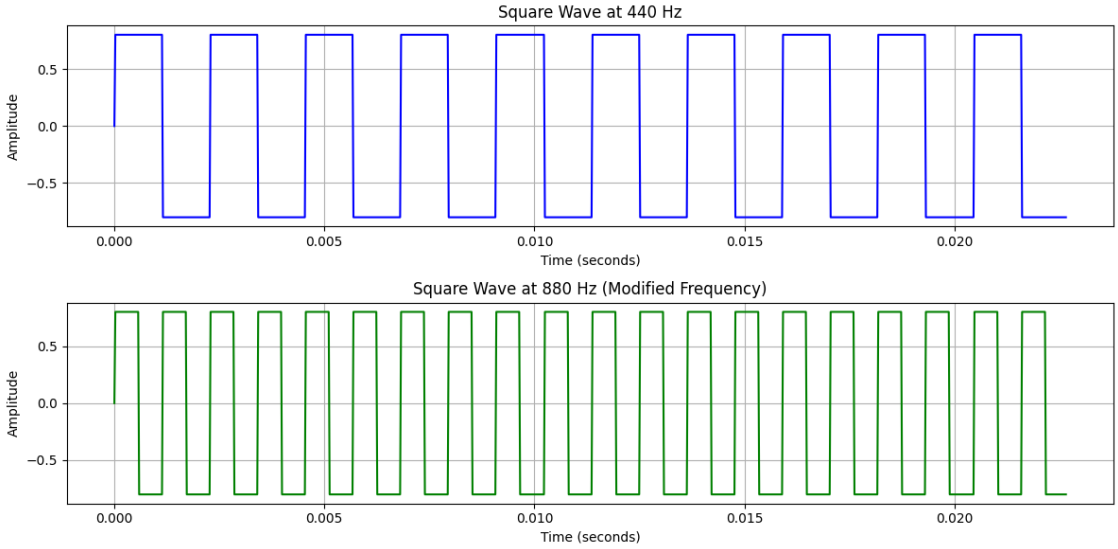
[9]:
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
frequency = 440  # Frequency in Hz
sampling_rate = 44100  # Sampling rate in Hz
duration = 5.0  # Duration in seconds

# Generate time points
t = np.linspace(0, duration, int(sampling_rate * duration), endpoint=False)

# Generate sine wave
sine_wave = np.sin(2 * np.pi * frequency * t)

# Plot the waveform
plt.figure(figsize=(12, 6))
plt.plot(t[:1000], sine_wave[:1000])  # Plot the first 1000 samples for better␣
 ↪visibility
plt.title("Sine Wave at 440 Hz")
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()
```



2.8 Square Wave Synthesis with Amplitude and Frequency Manipulation

[10]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
frequency = 440  # Frequency in Hz
amplitude = 0.8  # Amplitude (0 to 1)
sampling_rate = 44100  # Sampling rate in Hz
duration = 5.0  # Duration in seconds

# Generate time points
t = np.linspace(0, duration, int(sampling_rate * duration), endpoint=False)

# Generate square wave
square_wave = amplitude * np.sign(np.sin(2 * np.pi * frequency * t))

# Manipulate frequency (double it for demonstration)
new_frequency = 880  # New frequency in Hz
square_wave_new = amplitude * np.sign(np.sin(2 * np.pi * new_frequency * t))

# Plot the original and modified square waves
plt.figure(figsize=(12, 6))

# Original square wave
plt.subplot(2, 1, 1)
plt.plot(t[:1000], square_wave[:1000], color="blue")
```

```
plt.title("Square Wave at 440 Hz")
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude")
plt.grid()

# Modified square wave (new frequency)
plt.subplot(2, 1, 2)
plt.plot(t[:1000], square_wave_new[:1000], color="green")
plt.title("Square Wave at 880 Hz (Modified Frequency)")
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude")
plt.grid()

plt.tight_layout()
plt.show()
```

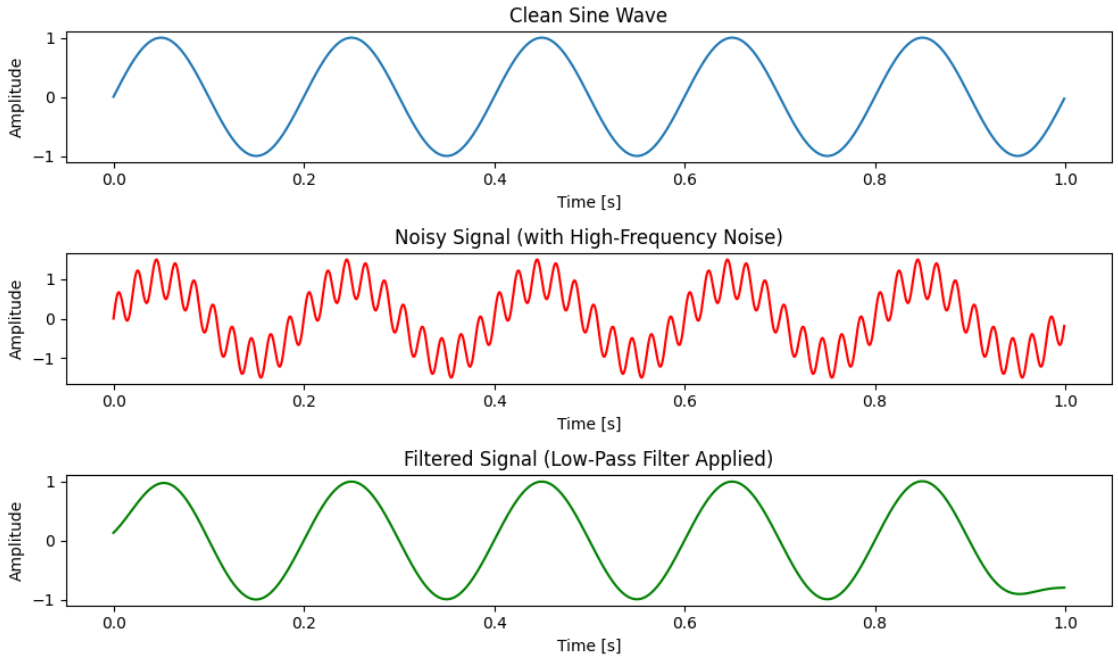


2.9 Implementing the low-pass filter on a noisy sine wave and observe the effect

[11]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt

# Step 1: Generate a noisy sine wave
fs = 1000  # Sampling frequency (samples per second)
t = np.arange(0, 1, 1/fs)  # Time vector
f_signal = 5  # Frequency of the sine wave in Hz
f_noise = 50  # Frequency of the noise in Hz
```

```
# Generate a clean sine wave
clean_signal = np.sin(2 * np.pi * f_signal * t)

# Add noise (high-frequency component)
noise = 0.5 * np.sin(2 * np.pi * f_noise * t)

# Noisy signal
noisy_signal = clean_signal + noise

# Step 2: Design a low-pass filter
def butter_lowpass(cutoff, fs, order=4):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

# Apply the low-pass filter to remove high-frequency noise
cutoff_frequency = 10  # Cutoff frequency for the low-pass filter
b, a = butter_lowpass(cutoff_frequency, fs)

# Apply the filter using filtfilt (zero-phase filtering)
filtered_signal = filtfilt(b, a, noisy_signal)

# Step 3: Plot the results
plt.figure(figsize=(10, 6))

# Plot the clean signal
plt.subplot(3, 1, 1)
plt.plot(t, clean_signal, label='Clean Sine Wave')
plt.title('Clean Sine Wave')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the noisy signal
plt.subplot(3, 1, 2)
plt.plot(t, noisy_signal, label='Noisy Signal', color='r')
plt.title('Noisy Signal (with High-Frequency Noise)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the filtered signal
plt.subplot(3, 1, 3)
plt.plot(t, filtered_signal, label='Filtered Signal', color='g')
plt.title('Filtered Signal (Low-Pass Filter Applied)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
```

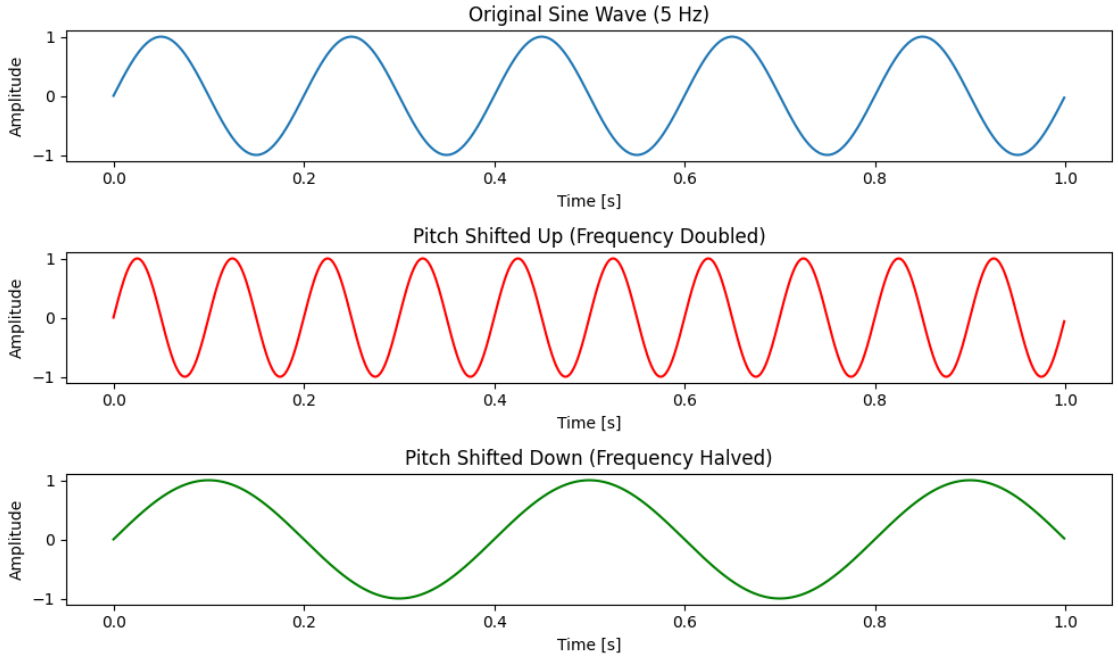
```
plt.tight_layout()
plt.show()
```



2.10 Applying pitch shifting to a sine wave and plot the resulting waveform

[12]:
```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Generate a sine wave
fs = 1000  # Sampling frequency (samples per second)
t = np.arange(0, 1, 1/fs)  # Time vector
f_original = 5  # Original frequency of the sine wave (in Hz)

# Generate the original sine wave
original_signal = np.sin(2 * np.pi * f_original * t)

# Step 2: Apply pitch shifting
# Shift the pitch up and down by a factor
pitch_up_factor = 2  # Shift the pitch up by a factor of 2 (10 Hz)
pitch_down_factor = 0.5  # Shift the pitch down by a factor of 0.5 (2.5 Hz)

# Create the pitch-shifted signals
shifted_up_signal = np.sin(2 * np.pi * f_original * pitch_up_factor * t)
shifted_down_signal = np.sin(2 * np.pi * f_original * pitch_down_factor * t)
```

```python
# Step 3: Plot the results
plt.figure(figsize=(10, 6))

# Plot the original sine wave
plt.subplot(3, 1, 1)
plt.plot(t, original_signal, label='Original Sine Wave')
plt.title('Original Sine Wave (5 Hz)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the pitch-shifted up signal
plt.subplot(3, 1, 2)
plt.plot(t, shifted_up_signal, label='Pitch Shifted Up (10 Hz)', color='r')
plt.title('Pitch Shifted Up (Frequency Doubled)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the pitch-shifted down signal
plt.subplot(3, 1, 3)
plt.plot(t, shifted_down_signal, label='Pitch Shifted Down (2.5 Hz)', color='g')
plt.title('Pitch Shifted Down (Frequency Halved)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()
```



2.11 Exolpring the effect of combining multiple waveform (eg., sine and square waves)

[13]:
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
fs = 1000  # Sampling frequency (samples per second)
t = np.arange(0, 1, 1/fs)  # Time vector

# Original frequencies
f_sine = 5  # Frequency of the sine wave (in Hz)
f_square = 15  # Frequency of the square wave (in Hz)

# Generate the individual waveforms
sine_wave = np.sin(2 * np.pi * f_sine * t)
square_wave = np.sign(np.sin(2 * np.pi * f_square * t))  # Square wave with 15␣
 ↪Hz

# Combine the sine and square waves (simple addition)
combined_wave = sine_wave + square_wave

# Plot the results
plt.figure(figsize=(10, 6))

# Plot the sine wave
plt.subplot(3, 1, 1)
plt.plot(t, sine_wave, label=f'Sine Wave ({f_sine} Hz)')
plt.title('Sine Wave (5 Hz)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the square wave
plt.subplot(3, 1, 2)
plt.plot(t, square_wave, label=f'Square Wave ({f_square} Hz)', color='r')
plt.title('Square Wave (15 Hz)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

# Plot the combined wave
plt.subplot(3, 1, 3)
plt.plot(t, combined_wave, label='Combined Waveform', color='g')
plt.title('Combined Sine and Square Waves')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
```

```
plt.tight_layout()
plt.show()
```