```
pip install SpeechRecognition
```

```
Collecting SpeechRecognition
  Downloading SpeechRecognition-3.10.4-py2.py3-none-any.whl.metadata (28 kB)
Requirement already satisfied: requests>=2.26.0 in /usr/local/lib/python3.10/dist-packages (from SpeechRecognition) (2.32.3)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from SpeechRecognition) (4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.26.0->SpeechRec
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.26.0->SpeechRecognition) (3
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.26.0->SpeechRecogniti
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.26.0->SpeechRecogniti
Downloading SpeechRecognition-3.10.4-py2.py3-none-any.whl (32.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 32.8/32.8 MB 38.4 MB/s eta 0:00:00
Installing collected packages: SpeechRecognition
Successfully installed SpeechRecognition-3.10.4
```

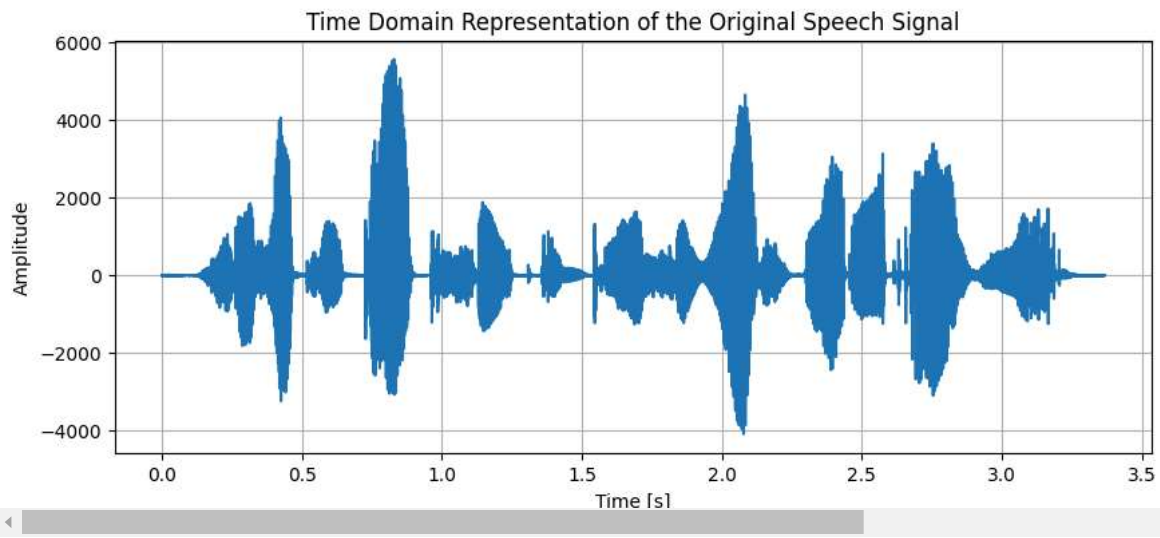
```
import speech_recognition as sr
```

Double-click (or enter) to edit

Question-1

1. Plot the Time Domain Representation of the Original Speech Signal

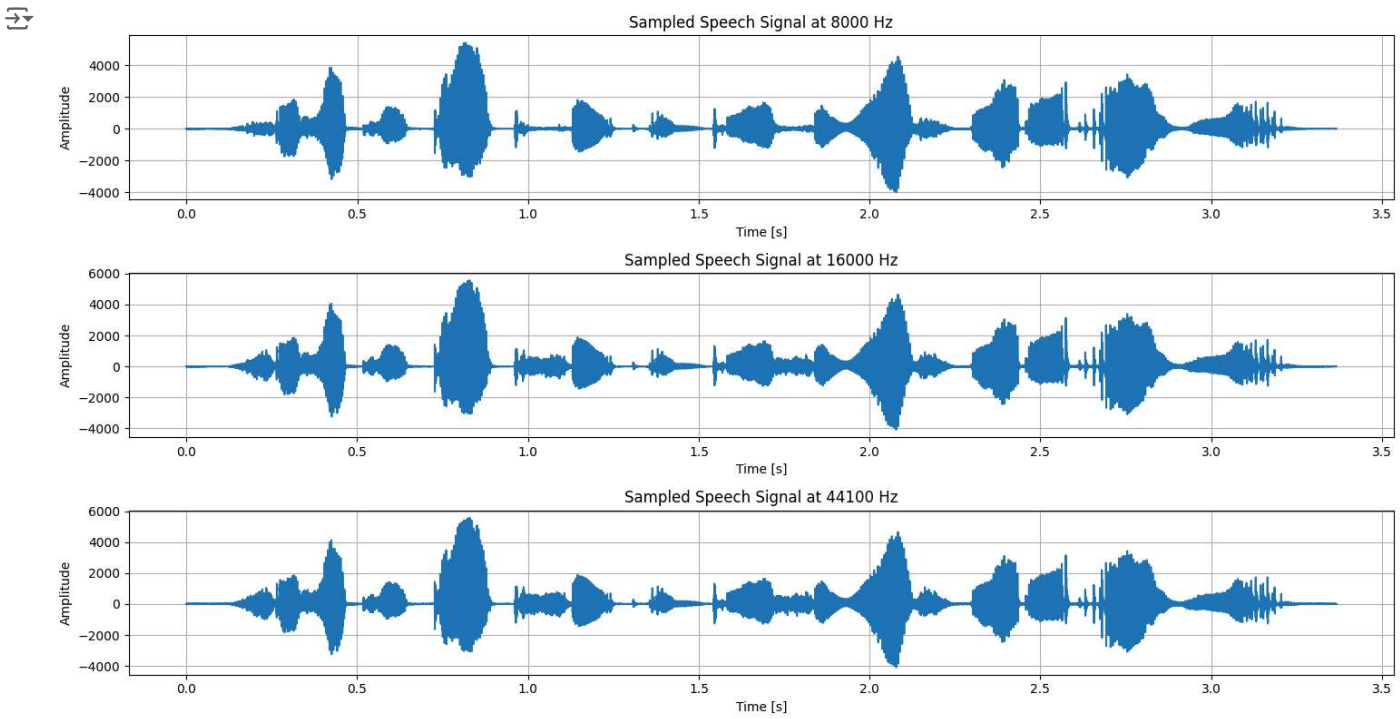
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile



sample_rate, data = wavfile.read('/content/sa1.wav')

time = np.linspace(0, len(data) / sample_rate, num=len(data))

# Plot the time domain representation
plt.figure(figsize=(10, 4))
plt.plot(time, data)
plt.title('Time Domain Representation of the Original Speech Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```



2. Sample the Speech Signal at Different Sampling Rates

```
from scipy.signal import resample

def resample_signal(data, original_rate, new_rate):
    num_samples = int(len(data) * new_rate / original_rate)
    return resample(data, num_samples)

# Define new sampling rates
```

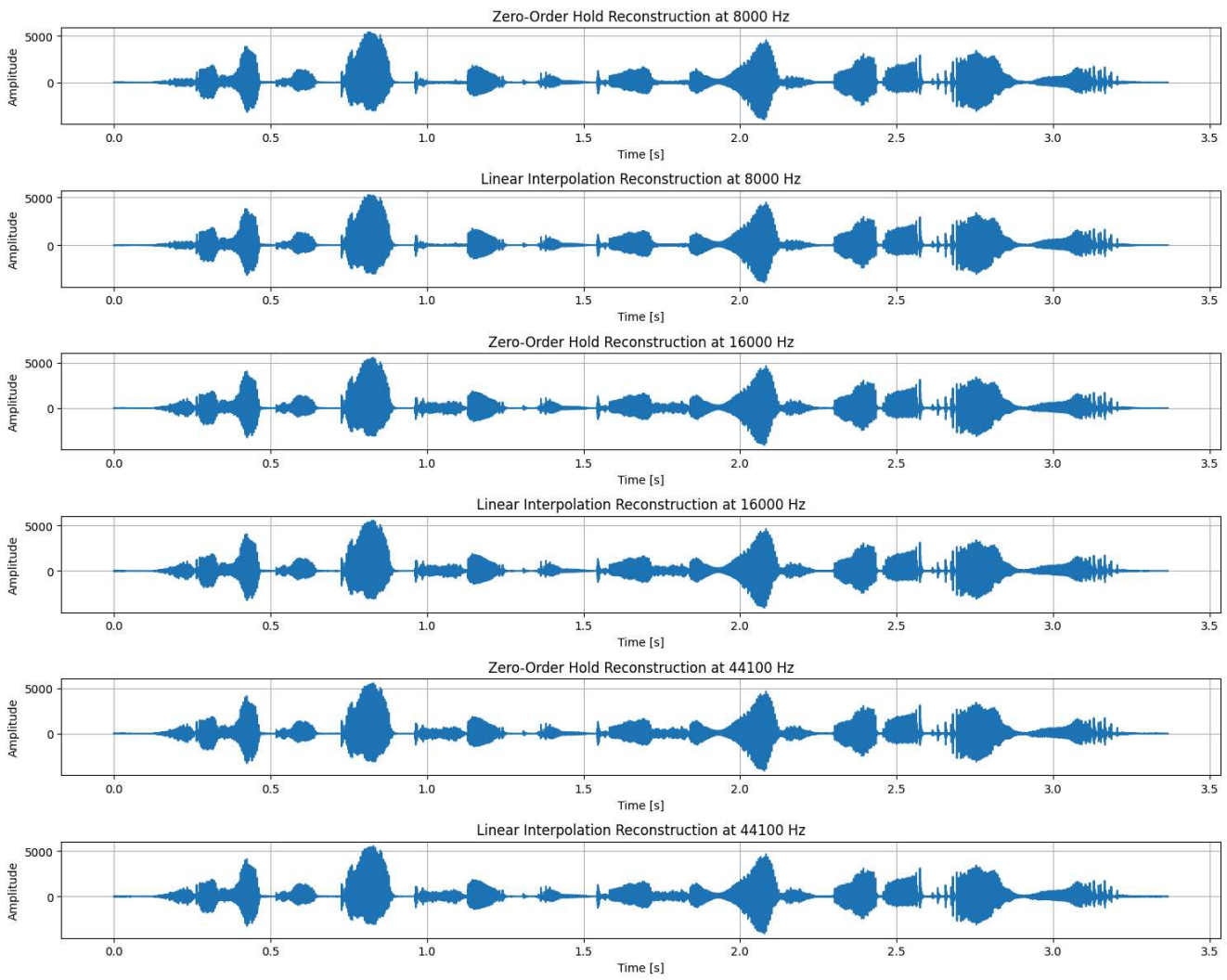
```
sampling_rates = [8000, 16000, 44100]

# Resample the signal
resampled_signals = {rate: resample_signal(data, sample_rate, rate) for rate in sampling_rates}



# Plot the sampled signals
plt.figure(figsize=(15, 8))

for i, rate in enumerate(sampling_rates):
    time_resampled = np.linspace(0, len(resampled_signals[rate]) / rate, num=len(resampled_signals[rate]))
    plt.subplot(len(sampling_rates), 1, i + 1)
    plt.plot(time_resampled, resampled_signals[rate])
    plt.title(f'Sampled Speech Signal at {rate} Hz')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')
    plt.grid(True)

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



3. Reconstruct the Signal Using Zero-Order Hold and Linear Interpolation

```
from scipy.interpolate import interp1d

def zero_order_hold_interpolation(data, original_rate, new_rate):
    # Create a new time array for the new sampling rate
    num_samples = int(len(data) * new_rate / original_rate)
    new_time = np.linspace(0, len(data) / original_rate, num=num_samples)

    # Original time array
    old_time = np.linspace(0, len(data) / original_rate, num=len(data))

    # Zero-order hold interpolation
    interp_func = interp1d(old_time, data, kind='nearest', fill_value="extrapolate")
    return interp_func(new_time)

# Reconstruct signals using zero-order hold interpolation
```

```
reconstructed_signals_zero_order = {rate: zero_order_hold_interpolation(resampled_signals[rate], rate, sample_rate)
                                    for rate in sampling_rates}



def linear_interpolation(data, original_rate, new_rate):
    # Create a new time array for the new sampling rate
    num_samples = int(len(data) * new_rate / original_rate)
    new_time = np.linspace(0, len(data) / original_rate, num=num_samples)

    # Original time array
    old_time = np.linspace(0, len(data) / original_rate, num=len(data))

    # Linear interpolation
    interp_func = interp1d(old_time, data, kind='linear', fill_value="extrapolate")
    return interp_func(new_time)

# Reconstruct signals using linear interpolation
reconstructed_signals_linear = {rate: linear_interpolation(resampled_signals[rate], rate, sample_rate)
                                for rate in sampling_rates}



# Plot the reconstructed signals
plt.figure(figsize=(15, 12))

# Plot zero-order hold reconstructed signals
for i, rate in enumerate(sampling_rates):
    time_reconstructed = np.linspace(0, len(resampled_signals[rate]) / rate, num=len(reconstructed_signals_zero_order[rate]))
    plt.subplot(len(sampling_rates) * 2, 1, 2 * i + 1)
    plt.plot(time_reconstructed, reconstructed_signals_zero_order[rate])
    plt.title(f'Zero-Order Hold Reconstruction at {rate} Hz')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')
    plt.grid(True)

# Plot linear interpolation reconstructed signals
for i, rate in enumerate(sampling_rates):
    time_reconstructed = np.linspace(0, len(resampled_signals[rate]) / rate, num=len(reconstructed_signals_linear[rate]))
    plt.subplot(len(sampling_rates) * 2, 1, 2 * i + 2)
    plt.plot(time_reconstructed, reconstructed_signals_linear[rate])
    plt.title(f'Linear Interpolation Reconstruction at {rate} Hz')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')
    plt.grid(True)

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



4. Calculate the Mean Squared Error (MSE) Between the Original and the Reconstructed Signals

```
from sklearn.metrics import mean_squared_error

# Function to calculate MSE
def calculate_mse(original_signal, reconstructed_signal):
    return mean_squared_error(original_signal, reconstructed_signal)

# Calculate MSE for zero-order hold interpolation
mse_zero_order = {rate: calculate_mse(data[:len(reconstructed_signals_zero_order[rate])],
                                      reconstructed_signals_zero_order[rate])
                  for rate in sampling_rates}

# Calculate MSE for linear interpolation
mse_linear = {rate: calculate_mse(data[:len(reconstructed_signals_linear[rate])],
                                  reconstructed_signals_linear[rate])
              for rate in sampling_rates}

# Print MSE for Zero-Order Hold Interpolation
print("MSE for Zero-Order Hold Interpolation:")
```

```
for rate, mse in mse_zero_order.items():
    print(f" {rate}: {mse:.2f}")

# Print MSE for Linear Interpolation
print("MSE for Linear Interpolation:")
for rate, mse in mse_linear.items():
    print(f" {rate}: {mse:.2f}")
```

```
MSE for Zero-Order Hold Interpolation:
 8000: 15274.80
 16000: 0.00
 44100: 29953.92
MSE for Linear Interpolation:
 8000: 6586.56
 16000: 0.00
 44100: 29075.99
```

The Zero-Order Hold Interpolation

It performed poorly at 8000 Hz with a high MSE of 15274.80.

It achieved perfect interpolation at 16000 Hz with an MSE of 0.00, indicating no error.

However, at 44100 Hz, the MSE increased significantly to 29953.92, indicating poorer performance compared to 16000 Hz.

The Linear Interpolation It outperformed Zero-Order Hold Interpolation at 8000 Hz with a lower MSE of 6586.56.

Similar to Zero-Order Hold, Linear Interpolation achieved perfect interpolation at 16000 Hz with an MSE of 0.00.

At 44100 Hz, the MSE for Linear Interpolation was 29075.99, indicating slightly better performance compared to Zero-Order Hold at the same sampling rate.

Based on the MSE values provided, Linear Interpolation generally performed better than Zero-Order Hold Interpolation across the different sampling rates, with lower errors in most cases.

QUESTION -2

(2) Implement the source-filter Model for a given speech signal and analyze the impact of sampling and reconstruction on the quality of the speech signal.

(a) Generate a synthetic speech signal using the source-filter model.

(i) Create a source signal (e.g., a glottal pulse train for voiced sounds or white noise for unvoiced sounds).

(ii) Apply a filter that models the vocal tract, represented by an all-pole filter or an FIR filter with formants (resonances of the vocal tract).

(b) Plot the generated speech signal and analyze the effect of the filter on the original source.

(c) Sample the speech signal generated in the above task at different sampling rates (e.g., 8 kHz, 16 kHz, 44.1 kHz).

(d) Reconstruct the signal using a suitable interpolation method (e.g., zero- order hold, linear interpolation).

(e) Compute the Mean Squared Error (MSE) between the original and re- constructed speech signals.

Write an inference on tasks such as creating the source filter model, different sampling rates, and reconstruction of the sampled signals.

To generate a voiced speech signal using an impulse train generator

Steps:

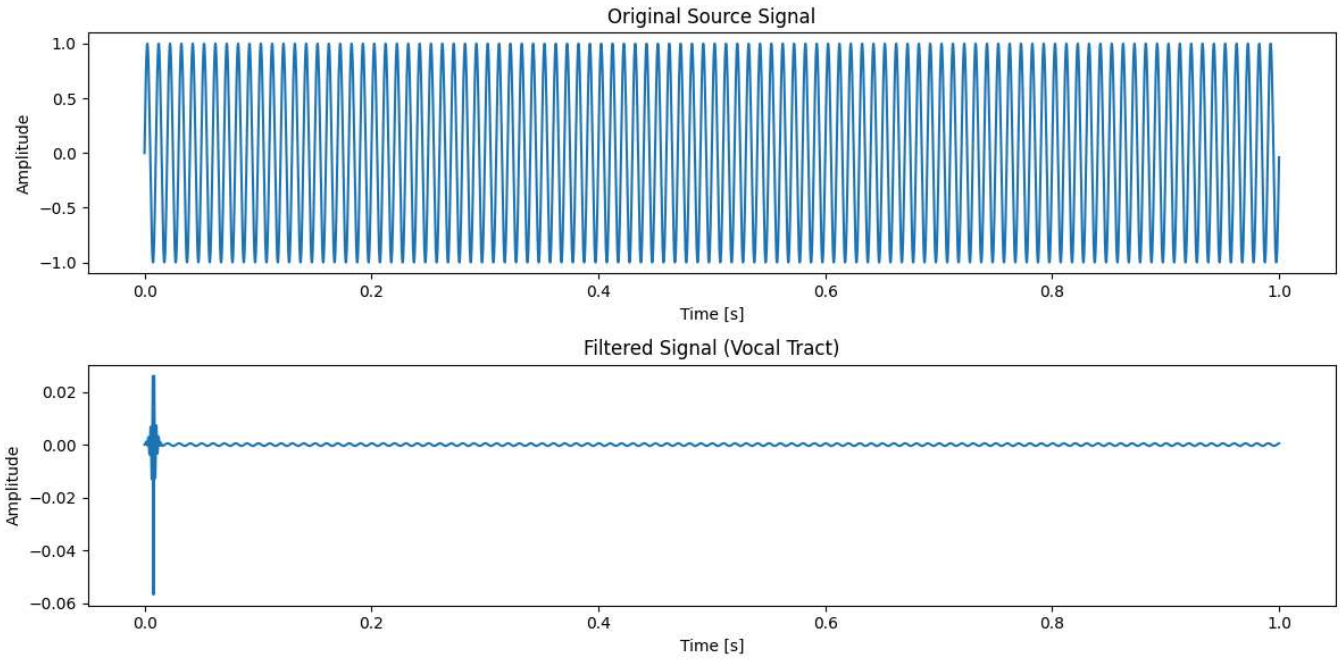
Impulse Train: A voiced speech signal can be modeled by an impulse train generator, where impulses occur at** regular intervals** based on the pitch of the voice.

Filtering: After generating the impulse train, you can apply a vocal tract filter (an all-pole filter) to simulate the resonances of the vocal tract.

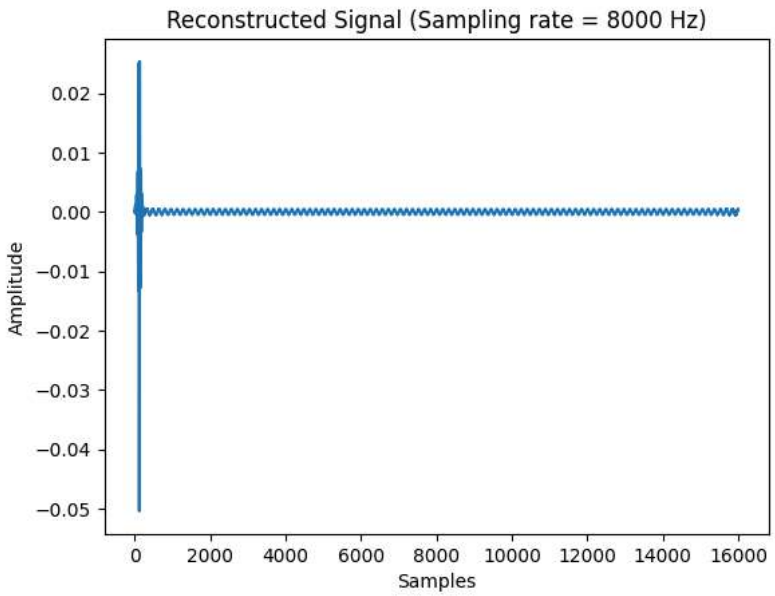
Pitch Period: The pitch period defines the interval between impulses. It can be calculated as Fs / f0, where Fs is the sampling rate and f0 is the fundamental frequency (pitch).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter, firwin, resample
from scipy.io import wavfile
from sklearn.metrics import mean_squared_error
```

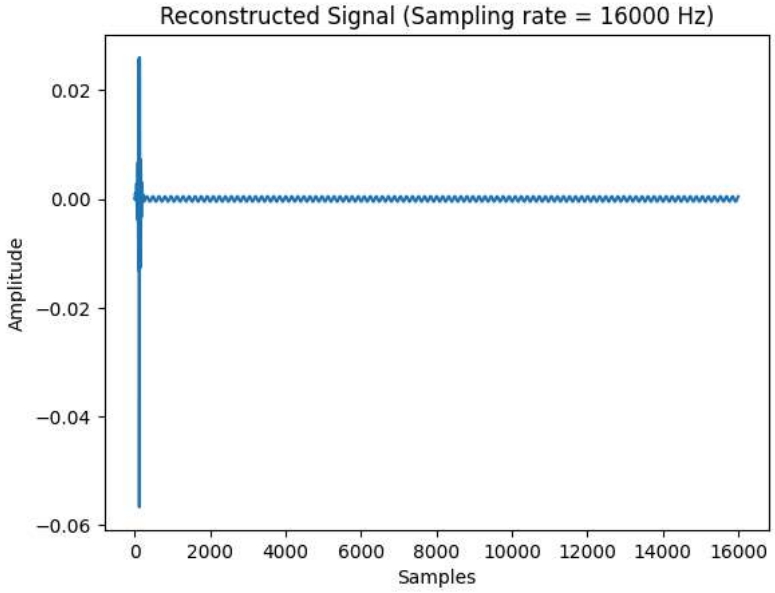
```
def generate_source_signal(duration=1.0, fs=16000, f0=100, voiced=True):
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)
    if voiced:
        # Glottal pulse train for voiced sound
        source_signal = np.sin(2 * np.pi * f0 * t)
    else:
        # White noise for unvoiced sound
        source_signal = np.random.randn(len(t))
    return source_signal, fs



def apply_vocal_tract_filter(source_signal, fs):
    # Formant frequencies in Hz (F1, F2, F3)
    formants = [500, 1500, 2500]
    bandwidths = [60, 90, 100]  # Bandwidth for each formant

    # Ensure that cutoff frequencies do not include Nyquist frequency
    nyquist = fs / 2
    formants_normalized = [f / nyquist for f in formants if f < nyquist]

    if len(formants_normalized) == 0:
        raise ValueError("Formant frequencies must be less than the Nyquist frequency.")

    # Design FIR filter using formant frequencies
    numtaps = 255  # Filter order, ensuring it's odd to avoid the Nyquist issue
    filter_coefs = firwin(numtaps, formants_normalized, pass_zero=False)

    # Filter the source signal to simulate vocal tract
    filtered_signal = lfilter(filter_coefs, 1.0, source_signal)
    return filtered_signal



def generate_source_signal(duration=1.0, fs=16000, f0=100, voiced=True):
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)
    if voiced:
        # Glottal pulse train for voiced sound
        source_signal = np.sin(2 * np.pi * f0 * t)
    else:
        # White noise for unvoiced sound
        source_signal = np.random.randn(len(t))
    return source_signal, fs



def plot_signals(original_signal, filtered_signal, fs):
    t = np.linspace(0, len(original_signal) / fs, num=len(original_signal))

    plt.figure(figsize=(12, 6))

    # Original source signal
    plt.subplot(2, 1, 1)
    plt.plot(t, original_signal)
    plt.title("Original Source Signal")
    plt.xlabel("Time [s]")
    plt.ylabel("Amplitude")

    # Filtered (speech) signal
    plt.subplot(2, 1, 2)
    plt.plot(t, filtered_signal)
    plt.title("Filtered Signal (Vocal Tract)")
    plt.xlabel("Time [s]")
    plt.ylabel("Amplitude")

    plt.tight_layout()
    plt.show()


source_signal, fs = generate_source_signal(duration=1.0, fs=16000, f0=100, voiced=True)
filtered_signal = apply_vocal_tract_filter(source_signal, fs)

# Task (b) - Plot signals and analyze effect of the filter
plot_signals(source_signal, filtered_signal, fs)
```



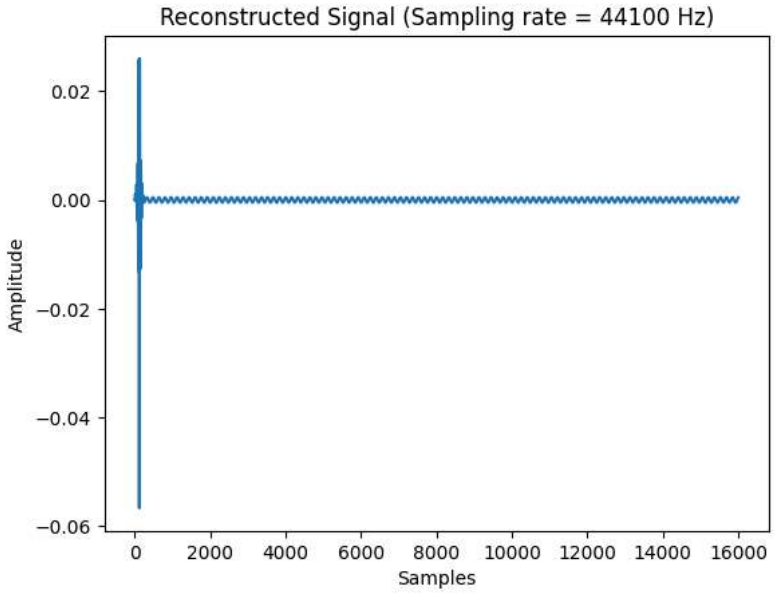

```
def sample_signal(signal, fs, new_fs):
    num_samples = int(len(signal) * new_fs / fs)
    sampled_signal = resample(signal, num_samples)
    return sampled_signal, new_fs

def reconstruct_signal(sampled_signal, original_fs, sampled_fs):
    num_samples = int(len(sampled_signal) * original_fs / sampled_fs)
    reconstructed_signal = resample(sampled_signal, num_samples)
    return reconstructed_signal


sampling_rates = [8000, 16000, 44100]
sampled_signals = [sample_signal(filtered_signal, fs, new_fs) for new_fs in sampling_rates]



def compute_mse(original_signal, reconstructed_signal):
    # Ensure signals are of the same length
    min_len = min(len(original_signal), len(reconstructed_signal))
    original_signal = original_signal[:min_len]
    reconstructed_signal = reconstructed_signal[:min_len]

    mse = mean_squared_error(original_signal, reconstructed_signal)
    return mse


# Task (d) - Reconstruct and calculate MSE for different sampling rates
for (sampled_signal, new_fs) in sampled_signals:
    reconstructed_signal = reconstruct_signal(sampled_signal, fs, new_fs)

    # Plot the reconstructed signal
    plt.figure()
    plt.plot(reconstructed_signal)
    plt.title(f"Reconstructed Signal (Sampling rate = {new_fs} Hz)")
    plt.xlabel("Samples")
    plt.ylabel("Amplitude")
    plt.show()

    mse = compute_mse(filtered_signal, reconstructed_signal)
    print(f"MSE between original and reconstructed signal (Sampling rate = {new_fs} Hz): {mse}")
```




```
MSE between original and reconstructed signal (Sampling rate = 8000 Hz): 5.134046406940019e-09
```




```
MSE between original and reconstructed signal (Sampling rate = 16000 Hz): 5.1022381622966205e-37
```




```
MSE between original and reconstructed signal (Sampling rate = 44100 Hz): 3.592538575888795e-37
```

1. The source signal is generated as either a glottal pulse train for voiced sounds .
2. The vocal tract filter introduces the effect of formants (resonant frequencies) on the raw source sound for forming into a speech-like audio.
3. The speech signal is then sampled at different rates (8 kHz, 16 kHz, and 44.1 kHz), where higher rates capture more detail.