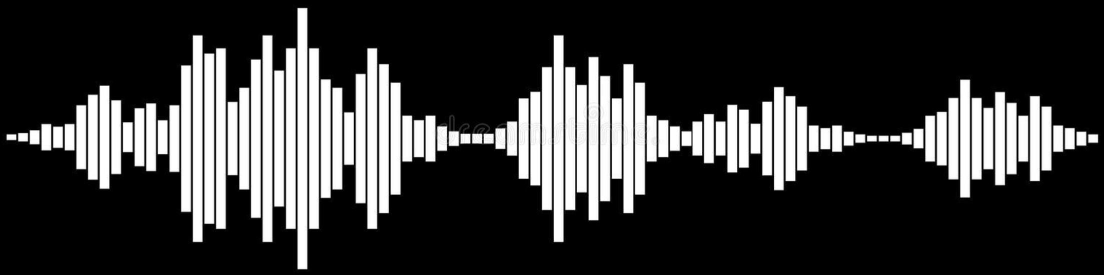


SOUND CARD

A sound card, also known as an audio card, is a computer expansion card that is responsible for processing audio signals in and out of a computer. It converts digital signals from the computer into analog signals that can be played through speakers or headphones, and converts analog signals from a microphone or other audio input device into digital signals that can be processed by the computer.

There are several types of sound cards, including:

1. Integrated sound card: This type of sound card is built into the motherboard of a computer and is typically the most basic type of sound card. It can handle basic audio tasks, but may not have as many features as other types of sound cards.
2. PCI sound card: This type of sound card is installed in a computer's PCI slot, which provides more bandwidth and better sound quality than an integrated sound card.
3. USB sound card: This type of sound card is external and connects to a computer via USB. It is often used for laptops or computers that don't have a PCI slot.
4. External sound card: This type of sound card is typically used by professionals in music production or other audio-intensive tasks. It connects to a computer via USB or another interface and often has more advanced features and higher quality components than other types of sound cards.

The basic hardware blocks of a sound card include:

1. Analog-to-digital converter (ADC): This block is responsible for converting analog audio signals from microphones, musical instruments, or other sources into digital signals that can be processed by a computer. The ADC typically includes a preamplifier, filter, and sampling circuitry.
2. Digital signal processor (DSP): This block is responsible for processing the digital audio signals, such as filtering, equalization, reverb, and other effects. The DSP may be a separate chip or integrated into the main processor of the sound card.
3. Digital-to-analog converter (DAC): This block is responsible for converting digital audio signals into analog signals that can be output to speakers or headphones. The DAC typically includes a reconstruction filter and amplifier.
4. Input/output (I/O) ports: These ports are used to connect the sound card to external devices such as microphones, headphones, speakers, and other audio equipment. Common I/O ports include microphone inputs, line inputs, headphone outputs, and speaker outputs.
5. Direct Memory Access (DMA): This hardware block allows the sound card to access the computer's memory without using the CPU. This allows the sound card to transfer data more efficiently, reducing the load on the CPU and improving performance.
6. Clock generator: This block provides a precise clock signal to synchronize the sampling rate of the ADC and DAC with the computer's clock. This ensures accurate and synchronized capture and playback of audio data.
7. Output amplifiers: These amplify the analog signals to a level that can drive headphones or speakers.

Data synchronization with the computer CPU and memory is critical for the proper functioning of a sound card. The DMA controller provides direct access to the computer's memory, allowing the sound card to read and write data without involving the CPU. This reduces the CPU load and ensures efficient transfer of data between the sound card and the computer's memory.

To synchronize data transfer between the sound card and the computer's memory, the sound card uses interrupts and a dedicated buffer. When the sound card is ready to transfer data, it sends an interrupt to the CPU, which temporarily stops executing other tasks and transfers the data to or from the sound card. The sound card stores the data in a buffer to avoid data loss or corruption due to timing issues.

DIRECT MEMORY ACCESS

Direct Memory Access (DMA) is a hardware feature that allows devices such as sound cards to transfer data between their own memory and the computer's memory without using the CPU. This improves performance by reducing the load on the CPU and allowing the device to transfer data more efficiently.

In the case of a sound card, DMA is used to transfer audio data between the sound card's buffer and the computer's memory. When the sound card is ready to transfer data, it sends a request to the DMA controller, which takes over the transfer process and transfers the data directly between the sound card's buffer and the computer's memory.

During the transfer, the CPU is free to execute other tasks, which improves system performance. Once the transfer is complete, the DMA controller sends an interrupt signal to the CPU to notify it that the transfer is finished.

The use of DMA in sound cards has several benefits. It allows for more efficient transfer of large amounts of data, such as when recording or playing back high-quality audio. It also reduces the load on the CPU, which can improve overall system performance.

However, there are some potential issues to consider when using DMA with sound cards. For example, if the sound card is not properly configured, it may overwrite other parts of memory, which can cause system crashes or other problems. Additionally, DMA transfers can cause timing issues, which can result in audio glitches or other anomalies.

SOUND SAMPLING

Sampling audio is the process of converting an analog audio signal, which is continuous in time and amplitude, into a digital format that can be processed and stored by a computer or digital device. The process involves taking measurements of the analog signal at regular intervals, called the sampling rate, and representing the amplitude of the signal at each interval as a digital value.

The sampling rate is usually expressed in Hertz (Hz), which represents the number of samples per second. Common sampling rates for audio are 44.1 kHz and 48 kHz, which are used for CD and DVD quality audio, respectively. Higher sampling rates, such as 96 kHz and 192 kHz, are also used in some professional recording applications.

The process of sampling audio involves the following steps:

1. Analog signal input: The analog audio signal is input to the sampling system, usually via a microphone or line-level input.
2. Sampling: The analog signal is sampled at a fixed rate, and the amplitude of the signal at each sample is quantized to a digital value.

3. Quantization: The quantization process involves assigning a digital value to each sample based on its amplitude. The number of bits used to represent each sample determines the resolution of the digital signal. Common bit depths for audio are 16-bit and 24-bit.
4. Encoding: The digital values are encoded into a digital audio file format, such as WAV or MP3.
5. Storage: The encoded digital audio file is stored on a computer or other digital device.

Sampling audio is a critical process in modern audio production, as it enables the manipulation, processing, and storage of audio signals in a digital format. The quality of the sampled audio is affected by several factors, including the sampling rate, bit depth, and the quality of the analog-to-digital conversion hardware.

STORAGE AND SOUND PROCESSING

A sound card stores and processes sound by converting analog sound waves into digital signals that can be processed and stored by a computer. This process involves several steps:

1. Sampling: The sound card samples the incoming analog signal at a specific rate, typically 44.1 kHz for CD-quality audio. The analog signal is measured at regular intervals, and the measurements are converted into digital values that can be stored in memory.
2. Quantization: The digital values are quantized into a specific number of bits, typically 16 or 24 bits per sample. This determines the resolution of the digital signal, with higher bit depths allowing for more precise representations of the original analog signal.
3. Processing: The digital signal is processed by the sound card's digital signal processor (DSP), which can apply various effects such as equalization, reverb, or compression. The DSP can also perform more advanced processing, such as noise reduction or pitch shifting.
4. Playback: Once the digital signal has been processed, it is converted back into an analog signal that can be played through speakers or headphones. This involves the digital-to-analog converter (DAC) and output amplifiers, which convert the digital signal back into an analog waveform and amplify it to a level suitable for playback.

In addition to processing sound, a sound card can also store and play back audio files. This involves reading the digital audio data from a storage device, such as a hard drive or CD, and converting it into an analog signal for playback. The sound card can also record audio from an external source, such as a microphone, by converting the analog signal into a digital format that can be stored on a computer.

ADC

An analog-to-digital converter (ADC) is a key component of a sound card that converts analog audio signals into digital signals that can be processed and stored by a computer. The ADC uses mathematical algorithms to convert the continuous analog waveform into discrete digital values that can be stored in memory.

The ADC works by taking the continuous analog waveform from a microphone or other audio source and converting it into a digital signal that can be stored on a computer. The process involves several steps:

1. Sampling: The sound card samples the incoming analog signal at a specific rate, typically 44.1 kHz for CD-quality audio. The analog signal is measured at regular intervals, and the measurements are stored as digital values.
2. Quantization: The digital values are quantized into a specific number of bits, typically 16 or 24 bits per sample. This determines the resolution of the digital signal, with higher bit depths allowing for more precise representations of the original analog signal.
3. Encoding: The digital values are encoded using a specific encoding scheme, such as pulse-code modulation (PCM). PCM encodes each digital value as a binary number, with the number of bits per sample determining the range of values that can be represented.
4. Output: Once the digital signal has been encoded, it can be stored in memory or processed by the sound card's digital signal processor (DSP).

The ADC follows some important mathematical rules, such as the Nyquist-Shannon sampling theorem, which states that in order to accurately sample a continuous analog signal, the sampling rate must be at least twice the highest frequency present in the signal. This means that the sound card must sample audio data at a rate of at least 2 times the highest frequency present in the audio signal, which is typically 20 kHz for human hearing.

DAC

A digital-to-analog converter (DAC) is a key component of a sound card that converts digital audio signals into analog signals that can be played back through speakers or headphones. The DAC uses mathematical algorithms to convert the digital audio data into an analog waveform that can be heard.

The DAC works by taking the digital audio data, which is represented as a series of binary numbers, and converting it into a continuous analog waveform that can be amplified and played back through speakers or headphones. The process involves several steps:

1. Sampling: The sound card samples the incoming digital audio data at a specific rate, typically 44.1 kHz for CD-quality audio. The digital audio data is measured at regular intervals, and the measurements are stored as binary numbers.
2. Quantization: The binary numbers are quantized into a specific number of bits, typically 16 or 24 bits per sample. This determines the resolution of the digital signal, with higher bit depths allowing for more precise representations of the original analog signal.
3. Reconstruction: The DAC uses mathematical algorithms to reconstruct the analog waveform from the digital audio data. This involves applying a low-pass filter to remove any high-frequency noise that may have been introduced during the sampling and quantization process.
4. Output Amplification: Once the analog waveform has been reconstructed, it is amplified to a level suitable for playback through speakers or headphones. The output amplifiers on the sound card are responsible for amplifying the analog signal to the appropriate level.

The DAC follows some important mathematical rules, such as the Nyquist-Shannon sampling theorem, which states that in order to accurately reproduce a continuous analog signal, it must be sampled at least twice the highest frequency present in the signal. This means that the sound card must sample audio data at a rate of at least 2 times the highest frequency present in the audio signal, which is typically 20 kHz for human hearing.

DAC & ADC MATH

The formulas used for digital-to-analog conversion (DAC) and analog-to-digital conversion (ADC) are based on the principles of sampling and quantization.

For DAC:

Digital Value = (Analog Value / Maximum Analog Value) x (Maximum Digital Value + 1)

where:

- Digital Value is the binary value representing the digital signal
- Analog Value is the amplitude of the analog signal
- Maximum Analog Value is the maximum amplitude of the analog signal
- Maximum Digital Value is the maximum binary value that can be represented by the digital signal (e.g., 255 for an 8-bit signal)

This formula converts a digital value into its corresponding analog value, by scaling the digital value to match the range of the analog signal.

For ADC:

Analog Value = (Digital Value / (Maximum Digital Value + 1)) x Maximum Analog Value

where:

- Analog Value is the amplitude of the analog signal
- Digital Value is the binary value representing the digital signal
- Maximum Digital Value is the maximum binary value that can be represented by the digital signal (e.g., 255 for an 8-bit signal)
- Maximum Analog Value is the maximum amplitude of the analog signal

This formula converts an analog signal into its corresponding digital value, by scaling the analog value to match the range of the digital signal.

ADC EXAMPLE

Let's assume that we have a microphone that captures an analog audio signal with a maximum voltage range of -1V to +1V. We want to convert this analog signal into a digital file with a bit depth of 16 bits and a sampling rate of 44.1 kHz using a sound card.

The conversion process involves several steps:

1. Amplification: The analog signal from the microphone may be too weak to be accurately digitized, so it may need to be amplified using an analog pre-amplifier.
2. Sampling: The sound card samples the amplified analog signal at a specific rate, typically 44.1 kHz for CD-quality audio. The analog signal is measured at regular intervals, and the measurements are stored as digital values.

3. Quantization: The digital values are quantized into a specific number of bits, typically 16 bits per sample. This determines the resolution of the digital signal, with higher bit depths allowing for more precise representations of the original analog signal.
4. Encoding: The digital values are encoded using a specific encoding scheme, such as pulse-code modulation (PCM). PCM encodes each digital value as a binary number, with the number of bits per sample determining the range of values that can be represented.
5. Storage: The resulting digital signal is stored in a digital file, such as a WAV or MP3 file, for playback or further processing.

To perform the analog-to-digital conversion, we can use the following formula to convert a single analog voltage value into a digital value:

Digital Value = (Analog Voltage + Maximum Voltage) / (2 x Maximum Voltage) x Maximum Digital Value

In this case, the maximum voltage is +1V and the minimum voltage is -1V, so the voltage range is 2V. The maximum digital value is 65,535 (2^16 - 1), so the formula becomes:

Digital Value = (Analog Voltage + 1) / 2 x 65,535

If the microphone captures an analog voltage of 0.5V, the formula would be:

Digital Value = (0.5 + 1) / 2 x 65,535 Digital Value = 0.75 x 65,535 Digital Value = 49,151

So, the sound card would convert the analog voltage of 0.5V into a digital value of 49,151 using this formula. This process is repeated for each sample in the analog audio signal, resulting in a digital file that represents the original analog waveform.

DAC EXAMPLE

Let's assume that we have a digital audio signal with a bit depth of 16 bits and a sampling rate of 44.1 kHz. We can represent each sample of the digital signal as a 16-bit binary number, with a range of values from 0 to 65,535.

To convert this digital signal into an analog waveform, we need to use a DAC that can convert the binary numbers into voltage values. For example, let's say that the DAC has a voltage range of -5V to +5V, and can represent 65,536 different voltage levels.

To convert a single 16-bit binary number into a voltage value, we can use the following formula:

Voltage = (Digital Value / Maximum Digital Value) x (Maximum Voltage - Minimum Voltage) + Minimum Voltage

In this case, the maximum digital value is 65,535 (2^16 - 1), and the maximum voltage is +5V, and the minimum voltage is -5V. If we want to convert a binary number of 32767, the formula would be:

Voltage = (32767 / 65535) x (5 - (-5)) + (-5) Voltage = 0.00001526 x 10 + (-5) Voltage = -4.9998474 V

So, the DAC would output a voltage of -4.9998474 V for the digital sample of 32767. This process is repeated for each sample in the digital audio signal, and the resulting voltage values are used to create a continuous analog waveform that can be played back through speakers or headphones.

MICROPHONE SOUND TO DIGITAL EXAMPLE

The process of converting sound from a microphone through a sound card to a digital file involves several steps:

1. Microphone input: The first step is to connect the microphone to the sound card's input port. The microphone converts sound waves into electrical signals, which are sent to the sound card's analog-to-digital converter (ADC).
2. Analog-to-digital conversion: The ADC converts the analog electrical signals from the microphone into digital data that can be processed by the sound card and computer. This involves sampling the analog signals at regular intervals and quantizing the sampled values into discrete digital values.
3. Digital signal processing: Once the analog signals are converted to digital data, the sound card's digital signal processor (DSP) can perform various processing tasks such as filtering, equalization, and noise reduction. These tasks can be performed in real-time or post-processing, depending on the specific sound card and software being used.
4. Storage: The processed digital data is then stored in a buffer on the sound card or transferred directly to the computer's memory using Direct Memory Access (DMA). From there, the data can be stored on the computer's hard drive as a digital audio file, such as WAV or MP3.
5. Playback: To play back the audio file, the computer retrieves the data from the hard drive and sends it to the sound card's digital-to-analog converter (DAC). The DAC converts the digital data back into analog electrical signals, which can be amplified and sent to speakers or headphones for playback.

Throughout this process, timing is critical to ensure accurate and synchronized capture and playback of audio data. The sound card's clock generator provides a precise clock signal to synchronize the sampling rate of the ADC and DAC with the computer's clock. Interrupts and buffers are also used to synchronize data transfer between the sound card and computer's memory.

PYTHON INSTALLATION & SETUP

Download python3 - https://www.python.org/downloads/

Add python to environment variables if **python3** path is not defined - https://datatofish.com/add-python-to-windows-path/

DOWNLOAD & INSTALL LIBRARIES

Open command line (win+R, type “cmd”, press enter)

pip install numpy

pip install matplotlib

pip install scipy

RESOURCES, DOCUMENTATION & LINKS:

Python3 documentation - https://docs.python.org/3/ , https://docs.python.org/3.11/tutorial/index.html

Numpy documentation - https://numpy.org/doc/

Matplotlib documentation – https://matplotlib.org/stable/index.html

Scipy documentation - https://docs.scipy.org/doc/scipy/

PREVIOUS CODE IMPROVEMENTS & UPDATES

We will update our equalizer code to support gain inputs in dB

To convert db. gain to linear gain we can use formula like this

```
linear_gain = 10 ** (gain / 20)
```

Example: how to increase frequencies in human speech specter

```
# perform equalizing
    bands = [(0, 300), (300, 500), (500, 2000), (2000, 3400), (3400, 4800)]
    gains = [-12.0, -6.0, 6.0, 3.0, -6.0]

    audio_analyzer.equalize(bands, gains)
```

We will separate information functions to another class called AudioInfoProvider. Move methods from AudioAnalyzer class to AudioInfoProvider, add variable to set reference to analyzer class that handles data.

```
class AudioInfoProvider:
    def __init__(self, audio_analyzer) -> None:
        self.audio_analyzer = audio_analyzer

    def display_info(self):
        print(f"Number of channels: { self.audio_analyzer.audio.shape[1] }")
        print(f"Sample rate: { self.audio_analyzer.samplerate}")

    def display_waveform(self):
        # retrive time in seconds
        time = np.linspace(0., self.audio_analyzer.length, self.audio_analyzer.audio.shape[0])

        plt.plot(time, self.audio_analyzer.audio[:, 0], label="Left Channel")
        plt.plot(time, self.audio_analyzer.audio[:, 1], label = "Right Channel")

        plt.legend()

        plt.xlabel("Time [s]")
        plt.ylabel("Amplitude")

        plt.show()

    def display_fft_spectr(self):
        plt.figure()
        plt.specgram(self.audio_analyzer.audio[:, 0], NFFT=1024,
Fs=self.audio_analyzer.samplerate, noverlap=900 )
        plt.xlabel("Time [s]")
        plt.ylabel("Frequency (Hz)")
        plt.colorbar()
        plt.show()
```

Example usage

```
audio_analyzer = AudioAnalyzer("stlkr_ex.wav")

    audio_info_provider = AudioInfoProvider(audio_analyzer=audio_analyzer)
    audio_info_provider.display_info()
    audio_info_provider.display_waveform()
    audio_info_provider.display_fft_spectr()
```

DIGITAL SIGNAL PROCESSING

The Digital Signal Processing (DSP) is a broad field that involves the manipulation and analysis of digital signals, particularly discrete-time signals obtained by sampling continuous-time signals such as sound waves. DSP has become an essential technology in various domains, including audio processing, speech processing, image processing, telecommunications, and control systems.

In audio processing, DSP is used for a variety of purposes:

1. Noise reduction: DSP techniques can be used to remove or reduce background noise, interference, and artifacts in audio signals, resulting in clearer and more intelligible speech or music.
2. Equalization and filtering: DSP allows the adjustment of the spectral balance of an audio signal to improve its tonal quality or to emphasize or attenuate specific frequency bands, which can be useful for enhancing speech intelligibility or musical balance.
3. Audio effects: DSP is widely used to create various audio effects, such as reverb, delay, pitch-shifting, and time-stretching, which can enhance the artistic quality or add creative elements to music and other audio content.
4. Compression: Audio data can be compressed using DSP techniques to reduce file sizes or transmission bandwidth requirements while maintaining acceptable levels of audio quality.

In speech processing and speech recognition, DSP plays a crucial role:

1. Feature extraction: DSP techniques are used to extract relevant features from speech signals that can be used as inputs to speech recognition algorithms. These features typically represent the spectral, temporal, or other characteristics of the speech signal that carry information about the spoken words or phonemes.
2. Noise reduction and enhancement: As mentioned earlier, DSP can be used to remove background noise and enhance the quality of speech signals, which can lead to improved speech recognition performance, especially in noisy or adverse environments.
3. Voice activity detection (VAD) and segmentation: DSP algorithms can be employed to identify and separate speech segments from non-speech segments (e.g., silence or background noise) in an audio signal. This can help reduce the computational complexity of speech recognition systems by focusing processing resources on the relevant portions of the signal.
4. Speaker identification and verification: DSP techniques can be used to extract speaker-specific features from speech signals, allowing for the identification or verification of a speaker's identity based on their unique vocal characteristics.
5. Pitch detection and prosody analysis: DSP can be used to analyze the pitch and prosodic features of speech, which can be important for understanding the speaker's emotional state, intent, or linguistic information such as stress patterns and intonation.

Overall, DSP plays a critical role in audio and speech processing by providing tools and techniques to analyze, manipulate, and enhance signals in various ways. This enables a wide range of applications, from improving audio quality and enabling creative effects to facilitating speech recognition and understanding, ultimately leading to more natural and effective human-computer interaction.

NOISE REDUCTION ALGORITHMS

Wiener Filter for Noise Reduction:

The Wiener filter is a noise reduction algorithm that aims to minimize the mean square error between the desired signal and the filtered output. It is an adaptive filter that uses statistical information about the signal and noise to estimate the best filter coefficients.

Pros:

- Simple to implement and computationally efficient.
- Can adapt to changing signal and noise characteristics.
- Works well when the noise is relatively stationary and has a known statistical distribution.

Cons:

- Performance may degrade when noise characteristics are unknown or change rapidly.
- May cause speech distortion if the filter is not properly adjusted.

Spectral Subtraction for Noise Reduction:

Spectral subtraction is a noise reduction algorithm that works by estimating the noise spectrum and subtracting it from the signal spectrum. This is typically done in the frequency domain using the Short-Time Fourier Transform (STFT).

Pros:

- Simple to implement and understand.
- Works well for stationary noise or when the noise spectrum can be accurately estimated.
- Can be extended to handle non-stationary noise with advanced techniques, such as adaptive spectral subtraction.

Cons:

- Sensitive to the accuracy of the noise spectrum estimation; errors can lead to speech distortion or residual noise.
- May introduce artifacts, such as "musical noise," when the noise subtraction is not smooth across frequency bins.

In summary, the Wiener filter and spectral subtraction algorithms are both simple and effective noise reduction techniques. The Wiener filter adapts to the statistical properties of the signal and noise, making it well-suited for stationary noise with known characteristics. On the other hand, spectral subtraction directly estimates and subtracts the noise spectrum, making it a good choice when the noise spectrum can be accurately estimated. Both methods have their pros and cons, and the choice between them depends on the specific application and requirements.

Both of this algorithms depend on Short-Time Fourier Transform (STFT) and Inverse Short-Time Fourier Transform (iSTFT).

SHORT-TIME FOURIER TRANSFORM

The Short-Time Fourier Transform (STFT) and its inverse, the Inverse Short-Time Fourier Transform (iSTFT), are mathematical techniques used to analyze and process signals, particularly non-stationary signals like speech and audio.

STFT is used for the following reasons:

1. Time-Frequency Representation: The Fourier Transform provides a frequency domain representation of a signal, but it does not provide any information about how the frequency content of the signal changes over time. The STFT, on the other hand, computes the Fourier Transform for overlapping short segments or "frames" of the input signal, resulting in a time-frequency representation. This enables the analysis and processing of non-stationary signals, where the frequency content varies over time.
2. Localization: The window function used in the STFT allows for better localization of the signal's frequency components. By adjusting the window size, a trade-off can be made between time resolution (the ability to resolve closely spaced events) and frequency resolution (the ability to resolve closely spaced frequency components). This is particularly useful when working with signals that have varying frequency content over time, such as speech and audio signals.
3. Signal Processing: The STFT is widely used in various signal processing applications, including noise reduction, speech enhancement, pitch detection, and time-scale modification. By working in the time-frequency domain, it is often easier to isolate and manipulate specific components of the signal or to apply adaptive processing techniques.

INVSERSE SHORT-TIME FOURIER TRANSFORM

iSTFT is used for the following reasons:

1. Signal Reconstruction: After processing the STFT representation of a signal (e.g., filtering, noise reduction, or modification of specific frequency components), the iSTFT is used to transform the processed time-frequency representation back into the time domain. This enables the synthesis of a modified or enhanced version of the original signal.
2. Overlap-Add: The iSTFT is based on the overlap-add method, which involves reconstructing the time-domain signal from overlapping frames. This ensures smooth transitions between frames and reduces artifacts that could be introduced by the processing performed in the time-frequency domain.

In summary, the STFT and iSTFT enable the analysis and processing of non-stationary signals, such as speech and audio, by providing a time-frequency representation of the signal. The STFT allows for better localization and manipulation of specific frequency components, while the iSTFT enables the reconstruction of the processed signal back into the time domain.

HANNING WINDOW

The Hanning window, also known as the Hann window or raised cosine window, is a type of window function used in signal processing and digital signal processing (DSP) applications. Window functions, in general, are used to taper or smooth the edges of a finite-length signal before applying a

Fourier Transform or other frequency-domain analysis. The Hanning window is widely used because of its good frequency resolution and smoothness properties.

The Hanning window is defined by the following equation:

w(n) = 0.5 * (1 - cos(2 * π * n / (N - 1)))

where n is the index of the window (0 <= n <= N-1) and N is the length of the window.

The main characteristics and advantages of the Hanning window include:

1. Smoothness: The Hanning window has a continuous first derivative, which reduces spectral leakage in the frequency domain compared to other window functions such as the rectangular window.
2. Frequency resolution: The Hanning window provides a good compromise between frequency resolution and sidelobe attenuation, making it suitable for many applications in signal processing.
3. Low sidelobes: The Hanning window has relatively low sidelobes in the frequency domain, which helps to minimize interference from adjacent frequency components when analyzing or processing a signal.

In the context of the Short-Time Fourier Transform (STFT) and other time-frequency analysis techniques, the Hanning window is applied to overlapping frames of the input signal to mitigate discontinuities at the frame boundaries. This leads to a smoother and more accurate representation of the signal's frequency content and helps to reduce artifacts introduced by the finite-length analysis.

STFT & iSTFT EXAMPLES

```
def stft(self, x, fft_size, hop_size):
        window = np.hanning(fft_size)
        return np.array([np.fft.rfft(window * x[i:i + fft_size]) for i in range(0, len(x) - 
fft_size, hop_size)])
```

x: The input signal (audio data) to be transformed.

fft_size: The size of the Fast Fourier Transform (FFT) to be applied.

hop_size: The step size between consecutive overlapping frames of the signal.

The function first creates a Hanning window of size fft_size. Then, it computes the STFT of the input signal x by applying the Hanning window to overlapping frames and performing an FFT on each frame. The output is an array of complex-valued frequency-domain representations of each frame.

```
    def istft(self, X, fft_size, hop_size):
        window = np.hanning(fft_size)
        x = np.zeros((X.shape[0] - 1) * hop_size + fft_size)
        for n, frame in enumerate(X):
            x[n * hop_size:n * hop_size + fft_size] += np.real(np.fft.irfft(frame)) * window
        return x
```

X: The STFT representation of a signal.

fft_size: The size of the FFT used in the original STFT computation.

hop_size: The step size between consecutive overlapping frames in the original STFT computation.

The function first creates a Hanning window of size fft_size. It initializes an empty signal x of appropriate length based on the number of frames and the hop size. Then, the function computes the iSTFT by iterating through each frame of the STFT representation X, performing an inverse FFT (iFFT) on the frame, and multiplying the result by the Hanning window. The windowed time-domain frames are overlapped and added together to reconstruct the original signal. The output is the reconstructed time-domain signal.

SPECTRAL SUBTRACTION EXAMPLE

```
def spectral_subtraction(self, alpha=2.0, beta=0.15):
        # Set FFT size and hop size
        fft_size = 512
        hop_size = int(fft_size / 2)

        # Compute the Short-Time Fourier Transform (STFT)
        stft_data = self.stft(self.audio, fft_size, hop_size)

        # Estimate the noise spectrum by averaging the magnitudes of the first few frames
        noise_spectrum = np.mean(np.abs(stft_data[:5]), axis=0)

        # Perform spectral subtraction
        de_noised_spectrum = np.abs(stft_data) - alpha * noise_spectrum[np.newaxis, :]
        de_noised_spectrum = np.maximum(de_noised_spectrum, beta * noise_spectrum[np.newaxis, :])

        # Compute the inverse STFT
        self.audio = self.istft(de_noised_spectrum * np.exp(1j * np.angle(stft_data)), fft_size,
hop_size)
```

The spectral_subtraction() function is a de-noising algorithm based on the spectral subtraction method. It removes background noise from an audio file to enhance speech quality and intelligibility.

Here's a detailed description of the function:

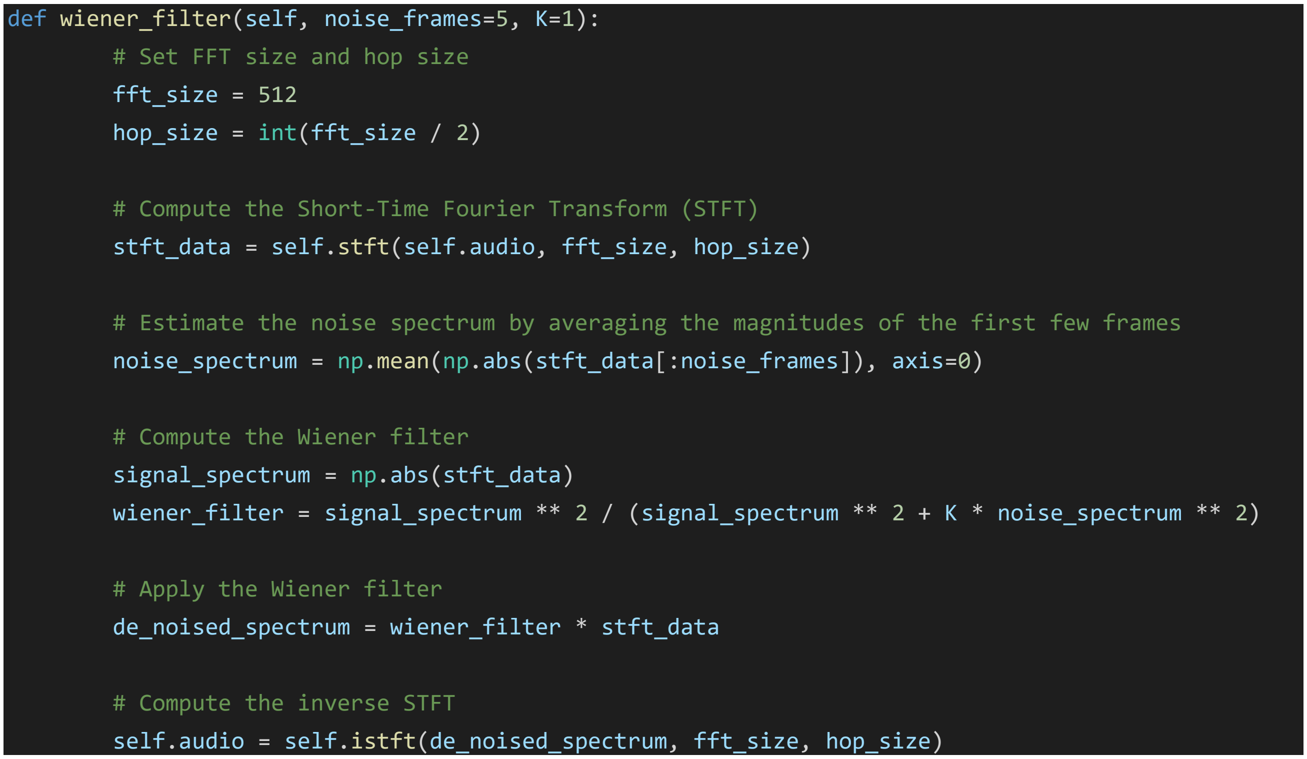
1. input_file: This is the path to the input audio file containing speech with background noise.
2. output_file: This is the path where the de-noised audio file will be saved after processing.
3. alpha: This is a parameter that determines the amount of noise reduction. Higher values result in stronger noise reduction but may also cause more distortion in the speech signal. The default value is 2.0, which provides a balance between noise reduction and speech quality.

4. beta: This is a parameter that controls the noise floor, which is the lowest level of noise that will be preserved in the output signal. Lower values result in a lower noise floor, while higher values preserve more noise. The default value is 0.15, which strikes a balance between reducing noise and preserving speech quality.

The function works as follows:

1. It loads the input audio file using the SoundFile library.
2. It computes the Short-Time Fourier Transform (STFT) of the audio data. The STFT represents the audio signal in the time-frequency domain, which makes it easier to manipulate and analyze.
3. It estimates the noise spectrum by averaging the magnitudes of the first few frames of the STFT. This assumes that the initial part of the audio signal mostly contains noise.
4. It performs spectral subtraction by subtracting the noise spectrum (scaled by the alpha parameter) from the audio spectrum. To prevent negative values, the result is clipped at the noise floor, which is determined by the beta parameter.
5. It computes the inverse STFT of the de-noised spectrum to obtain the de-noised audio signal in the time domain.
6. It saves the de-noised audio signal to the specified output file.

WIENER FILTER EXAMPLE



```python
def wiener_filter(self, noise_frames=5, K=1):
        # Set FFT size and hop size
        fft_size = 512
        hop_size = int(fft_size / 2)

        # Compute the Short-Time Fourier Transform (STFT)
        stft_data = self.stft(self.audio, fft_size, hop_size)

        # Estimate the noise spectrum by averaging the magnitudes of the first few frames
        noise_spectrum = np.mean(np.abs(stft_data[:noise_frames]), axis=0)

        # Compute the Wiener filter
        signal_spectrum = np.abs(stft_data)
        wiener_filter = signal_spectrum ** 2 / (signal_spectrum ** 2 + K * noise_spectrum ** 2)

        # Apply the Wiener filter
        de_noised_spectrum = wiener_filter * stft_data

        # Compute the inverse STFT
        self.audio = self.istft(de_noised_spectrum, fft_size, hop_size)
```

This code implements a noise reduction algorithm based on the Wiener filter, which is designed to minimize the mean square error between the desired signal and the filtered output. The implementation consists of several functions and steps as follows:

1. stft(x, fft_size, hop_size): This function computes the Short-Time Fourier Transform (STFT) of an input signal x. It takes an input signal x, FFT size fft_size, and hop size hop_size as arguments. The function applies a Hanning window to the input signal and computes the STFT by taking the FFT of overlapping frames.
2. istft(X, fft_size, hop_size): This function computes the inverse Short-Time Fourier Transform (iSTFT) of a given STFT X. It takes the STFT X, FFT size fft_size, and hop size hop_size as arguments. The function uses a Hanning window and the inverse FFT to convert the frequency domain representation back to the time domain.
3. wiener_filter(input_file, output_file, noise_frames=5, K=1): This function implements the Wiener filter noise reduction algorithm. It takes the following arguments:
    - input_file: The path to the input audio file containing speech with background noise.
    - output_file: The path where the de-noised audio file will be saved after processing.
    - noise_frames: The number of initial frames used to estimate the noise spectrum. The default value is 5.
    - K: A constant that controls the trade-off between noise reduction and speech distortion. The default value is 1.
4. The wiener_filter() function processes the input audio file as follows:
    - Loads the input audio file using the SoundFile library.
    - Computes the Short-Time Fourier Transform (STFT) of the audio data.
    - Estimates the noise spectrum by averaging the magnitudes of the first few frames of the STFT.
    - Computes the Wiener filter by dividing the square of the signal spectrum by the sum of the square of the signal spectrum and the product of the constant K and the square of the noise spectrum.
    - Applies the Wiener filter to the STFT data by multiplying the filter with the original STFT data.
    - Computes the inverse STFT of the de-noised spectrum to obtain the de-noised audio signal in the time domain.
    - Saves the de-noised audio signal to the specified output file.
5. In the if __name__ == "__main__": block, the input and output file paths are specified, and the wiener_filter() function is called to process the input audio file and save the de-noised output.

MICROPHONE RECORDING

A microphone device is an electronic device that converts sound waves into an electrical signal that can be recorded or processed by a computer or other digital device. Microphones are used in a wide variety of applications, including music recording, broadcasting, telecommunications, and speech recognition.

There are many types of microphones, each with its own strengths and weaknesses. Some of the most common types of microphones include:

1. **Dynamic microphones:** Dynamic microphones are rugged and durable, making them ideal for live performances and recording loud sounds. They work by using a diaphragm and a coil to generate an electrical signal in response to sound waves.
2. **Condenser microphones:** Condenser microphones are more sensitive and accurate than dynamic microphones, making them ideal for recording vocals and acoustic instruments. They work by using a diaphragm and a capacitor to generate an electrical signal in response to sound waves.
3. **Ribbon microphones:** Ribbon microphones are similar to dynamic microphones, but they use a thin strip of metal (the ribbon) instead of a coil to generate the electrical signal. Ribbon microphones are prized for their warm, natural sound and are often used to record stringed instruments and vocals.
4. **Lavalier microphones:** Lavalier microphones are small, discreet microphones that can be clipped onto clothing or hidden in a performer's hair. They are commonly used in broadcasting and live events.
5. **USB microphones:** USB microphones are designed to be plugged directly into a computer or other digital device, making them ideal for home recording and podcasting. They often include built-in preamps and other features to enhance the sound quality of the recording.

Microphone devices can be connected to a computer or other digital device using a variety of interfaces, including USB, XLR, and TRS. Many microphones require an external power source, such as phantom power or a battery, to operate.

Microphone programming involves capturing audio data from a microphone and processing it in real-time or storing it for later use. Microphone programming can be used for a wide range of applications, such as speech recognition, audio recording, and live performance.

When working with microphones in programming, there are several techniques and concepts that are important to understand, including:

1. **Buffers:** When capturing audio data from a microphone, it is important to use a buffer to store the incoming data. A buffer is a temporary storage area that holds a certain amount of data before it is processed or transferred. Buffers are used in microphone programming to prevent data loss and ensure that the audio data is processed in real-time.
2. **Sample rate and bit depth:** The sample rate and bit depth are important parameters that define the quality of the audio data being captured by the microphone. The sample rate is the number of samples taken per second, while the bit depth is the number of bits used to represent each sample. Higher sample rates and bit depths result in higher quality audio data, but also require more storage space and processing power.

3. **Audio processing:** Once the audio data has been captured, it can be processed using various techniques and algorithms. Audio processing can be used to remove noise, apply effects, equalize the sound, or extract features for analysis. Many audio processing libraries and tools are available for programming languages like Python and C++, making it easier to work with audio data.
4. **Real-time processing:** Real-time processing involves processing the audio data as it is being captured by the microphone, without any delay. Real-time processing is often used for live performances and applications like speech recognition, where low latency is critical.
5. **Offline processing:** Offline processing involves processing the audio data after it has been captured and stored in a file or buffer. Offline processing is often used for applications like audio recording, where the audio data can be processed at a later time.

In microphone programming, the audio data is typically captured in a buffer and processed in real-time or stored for later use. The audio data can be processed using various techniques and algorithms, depending on the application. Once the audio data has been processed, it can be stored or transmitted to other devices for further analysis or playback.

IMPLEMENTATION OF MICROPHONE RECORDING CASS

Download & install libraries

Open command line (win+R, type “cmd”, press enter)

pip install `pyaudio`

Code

```
import pyaudio
import wave
import numpy as np
import csv
import time

class MicrophoneDataProvider:
    def __init__(self, chunk=1024, channels=1, rate=44100, input_device=None):
        self.chunk = chunk
        self.channels = channels
        self.rate = rate
        self.frames = []
        self.input_device = input_device

    def get_available_devices(self):
        audio = pyaudio.PyAudio()
        n_devices = audio.get_device_count()
        print("Available audio devices:")
        for i in range(n_devices):
            info = audio.get_device_info_by_index(i)
            print(f"Device {i}: {info['name']}")
        audio.terminate()

    def set_input_device(self, device_index):
        audio = pyaudio.PyAudio()
        devices = audio.get_device_count()
        if device_index >= 0 and device_index < devices:
            self.input_device = device_index
            audio.terminate()
            return True
        else:
            audio.terminate()
            return False

    def start(self):
        self.audio = pyaudio.PyAudio()
        if self.input_device is None:
            self.stream = self.audio.open(format=pyaudio.paInt16,
                                          channels=self.channels,
                                          rate=self.rate,
                                          input=True,
                                          frames_per_buffer=self.chunk)
        else:
            self.stream = self.audio.open(format=pyaudio.paInt16,
                                          channels=self.channels,
                                          rate=self.rate,
                                          input=True,
                                          frames_per_buffer=self.chunk,
                                          input_device_index=self.input_device)

    def stop(self):
        if self.stream:
            self.stream.stop_stream()
            self.stream.close()

        self.audio.terminate()

    def record(self, duration): # duration in sec
        self.frames = []
        start_time = time.time()
        while time.time() - start_time < duration:
            data = self.stream.read(self.chunk)
            self.frames.append(data)

    def write_to_wav(self, filename):
        wf = wave.open(filename, 'wb')
        wf.setnchannels(self.channels)
        wf.setsampwidth(self.audio.get_sample_size(pyaudio.paInt16))
        wf.setframerate(self.rate)
        wf.writeframes(b''.join(self.frames))
        wf.close()

    def get_audio_data(self):
        audio_data = np.frombuffer(b''.join(self.frames), dtype=np.int16)

        if self.channels > 1:
            audio_data = audio_data.reshape(-1, self.channels)

        if audio_data.ndim > 1:
            audio_data = audio_data[:, 0]

        return audio_data

    def get_framerate(self):
        return self.rate / self.chunk
```

```
def get_samplerate(self):
    return self.rate
```

The `MicrophoneDataProvider` class is a utility class that allows you to record audio from a microphone and save it to a WAV file or analyze it in real-time. The class is built on top of the PyAudio library, which provides a cross-platform interface for recording and playing audio.

Here's a breakdown of the class and its methods:

- `__init__(self, chunk=1024, channels=1, rate=44100)`: Initializes the class with the chunk size, number of channels, and sample rate. The default values are 1024 samples per chunk, 1 channel, and 44100 samples per second.
- `start(self)`: Starts the PyAudio audio stream for recording.
- `stop(self)`: Stops the PyAudio audio stream and terminates the audio object.
- `record(self, duration)`: Records audio data for a specified duration in seconds.
- `write_to_wav(self, filename)`: Saves the recorded audio data to a WAV file with the specified filename.
- `get_audio_data(self)`: Returns the recorded audio data as a NumPy array.
- `get_framerate(self)`: Returns the frame rate of the recorded audio data.
- `get_samplerate(self)`: Returns the sample rate of the recorded audio data.
- `get_available_devices(self)`: Print list of all input devices available
- `set_input_device(self, device_index)`: Sets the input device to the specified device index. Returns `True` if the device name is valid and the device was successfully set, and `False` otherwise.

The `get_available_devices` method uses the PyAudio library to retrieve information about available audio devices and their indices. It returns a dictionary that maps the device name to its index.

The `set_input_device` method takes a device index as an argument and attempts to set the input device to the specified device. It uses the `get_available_devices` method to retrieve list of devices.

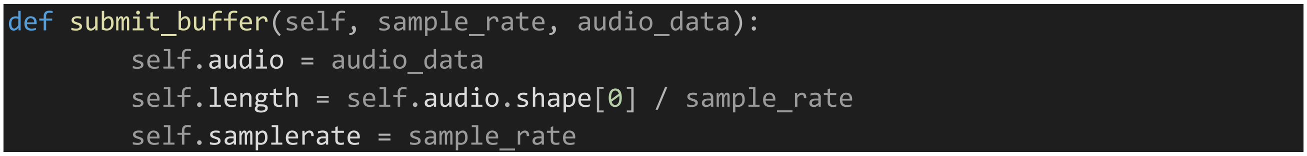
The `record` method records audio data for a specified duration by repeatedly reading audio data from the PyAudio stream and appending it to the `self.frames` attribute. The `write_to_wav` method saves the recorded audio data to a WAV file by opening a `wave` object, setting the parameters for the WAV file (number of channels, sample width, and sample rate), writing the audio data to the WAV file, and closing the WAV file.

The `get_audio_data` method returns the recorded audio data as a NumPy array. If the audio data has more than one channel, the method reshapes the array to have shape `(n_samples, n_channels)`.

The `get_framerate` method returns the frame rate of the recorded audio data by dividing the sample rate by the chunk size.

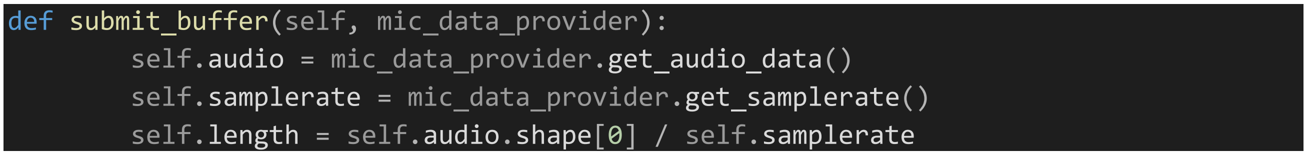
The `get_samplerate` method returns the sample rate of the recorded audio data.

AudioProcessor class modifications



```
def submit_buffer(self, sample_rate, audio_data):
        self.audio = audio_data
        self.length = self.audio.shape[0] / sample_rate
        self.samplerate = sample_rate
```

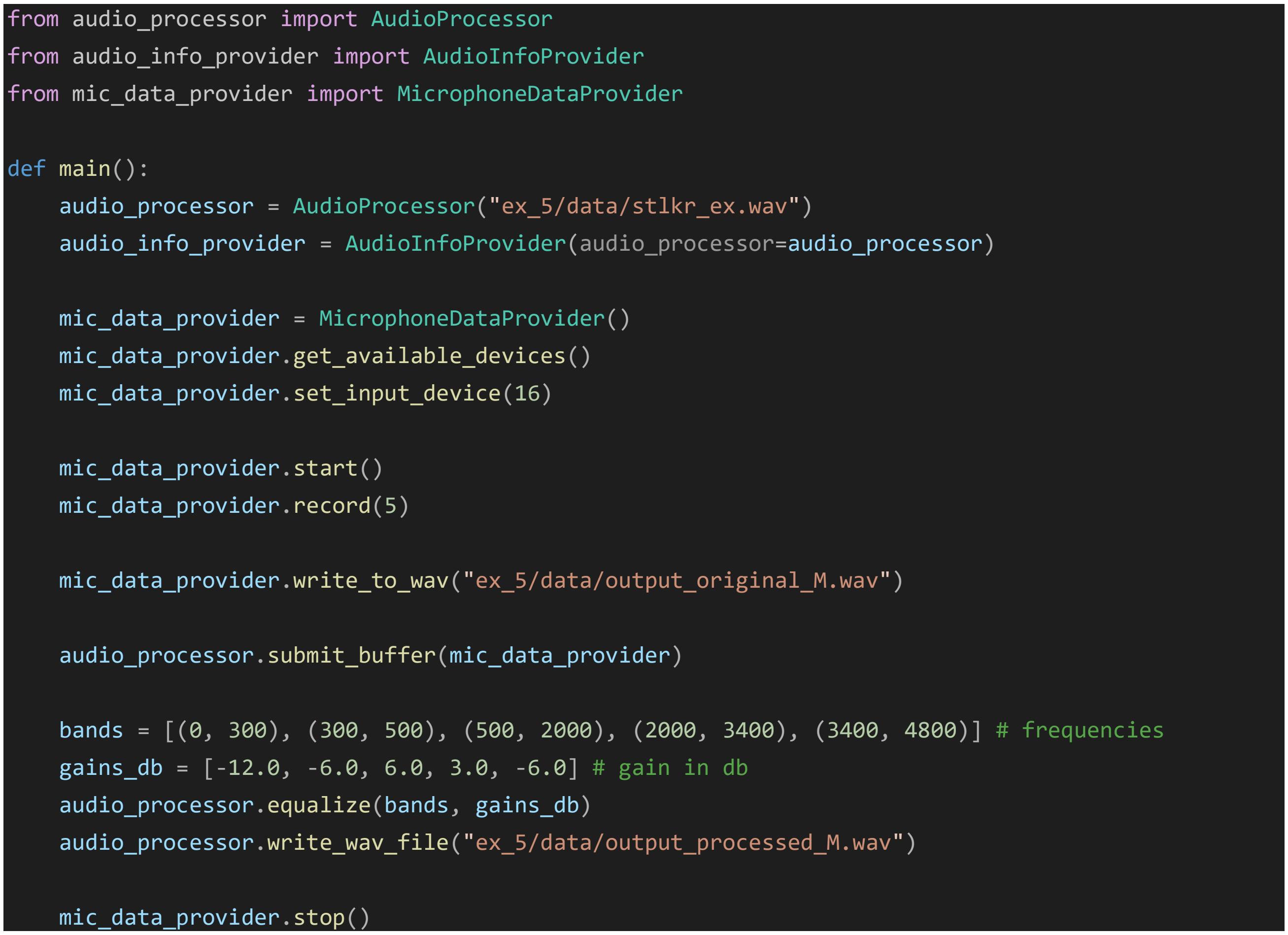
To be able to process data from microphone we need to add new method to AudioProcessor class. This method applies buffer data to AudioProcessor class. New data received from microphone wrapper class.



```
def submit_buffer(self, mic_data_provider):
        self.audio = mic_data_provider.get_audio_data()
        self.samplerate = mic_data_provider.get_samplerate()
        self.length = self.audio.shape[0] / self.samplerate
```

Another implementation of method. It uses direct access to microphone wrapper class.

Example usage of class



```
from audio_processor import AudioProcessor
from audio_info_provider import AudioInfoProvider
from mic_data_provider import MicrophoneDataProvider

def main():
    audio_processor = AudioProcessor("ex_5/data/stlkr_ex.wav")
    audio_info_provider = AudioInfoProvider(audio_processor=audio_processor)

    mic_data_provider = MicrophoneDataProvider()
    mic_data_provider.get_available_devices()
    mic_data_provider.set_input_device(16)

    mic_data_provider.start()
    mic_data_provider.record(5)

    mic_data_provider.write_to_wav("ex_5/data/output_original_M.wav")

    audio_processor.submit_buffer(mic_data_provider)

    bands = [(0, 300), (300, 500), (500, 2000), (2000, 3400), (3400, 4800)] # frequencies
    gains_db = [-12.0, -6.0, 6.0, 3.0, -6.0] # gain in db
    audio_processor.equalize(bands, gains_db)
    audio_processor.write_wav_file("ex_5/data/output_processed_M.wav")

    mic_data_provider.stop()
```

TODO: Lab 1

- **Додатковий метод для реверсії звукового сигналу** (розвертання масиву) та можливість подальшого зберігання. Метод потрібно додати до класу AudioProcessor.

- **Додатковий метод для вирізки звукового сигналу по заданих позиціях**. Наприклад, було завантажено моно файл з довжиною 10 секунд. За допомогою даного методу можна вирізати звуковий сигнал від 3 до 7 секунд та вподальшому використовувати його для інших маніпуляцій.

SOUND CLASIFICATION / RECOGNITION

Sound classification or recognition is the task of identifying sounds and assigning them into categories, or "classes". In the context of speech recognition, these categories could be different words or phonemes. In more general audio recognition, the categories could be things like different types of music, different environmental sounds, or different voices.

MEL-FREQUENCY CEPSTRAL COEFFICIENTS (MFCCs) AND MEL SPECTOGRAM

MFCCs and MEL spectrograms are two types of features that can be extracted from audio data. They both take into consideration the way humans perceive sound, specifically that human ears are more sensitive to certain frequencies.

MFCCs are a type of cepstral representation of the audio clip. A cepstrum is the result of taking the inverse Fourier transform (IFT) of the logarithm of the estimated spectrum of a signal.

A MEL spectrogram, on the other hand, is a spectrogram where the frequencies are converted to the Mel scale. A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time.

RANDOM FOREST CLASSIFIER

The Random Forest Classifier is a type of ensemble learning method, where a group of weak models come together to form a strong model. In Random Forest Classifier, the weak models are decision trees. Hence, in a random forest, multiple decision trees are grown and the prediction made by each decision tree is counted as a 'vote' for that class. The class receiving the most votes becomes our model's prediction.

LABEL ENCODING

Label Encoding refers to converting the labels into numeric form so as to convert it into the machine-readable form. Machine learning algorithms can then decide in a better way on how those labels must be operated. It is an important pre-processing step for the structured dataset in supervised learning.

TRAIN-TEST SPLIT

In Machine Learning, we create models to predict the outcome of certain events. Like the predictions of weather, the sales of a new product, etc. There is always a need to verify the stability of our machine learning model. We need to know how well our model performs on new unseen datasets.

For this, we use the Train-Test Split approach. We divide our data into two sets, a Training set, and a Test set. We use the Training Set to train our model, and we have the Test set to test and see how well our model has been trained.

MODE SERIALIZATION

Once we've trained our model, it's usually a good idea to save it, especially if the training process was time-consuming. This is called model serialization - saving the state of the machine learning model to a file. This allows us to load the model later and use it to make predictions without having to re-train it.

In Python, a popular library for model serialization is joblib. This allows us to save our models with a single line of code. We can then load the model later with another single line of code.

IMPLEMENTATION OF TRAINING CODE

Download & install libraries

Open command line (win+R, type “cmd”, press enter)

pip install librosa

pip install -U scikit-learn

pip install pandas

Importing the Necessary Libraries

```
import os
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
import joblib
```

In this block of code, we are importing all the libraries we will need for this script:

- os: This is a built-in Python library for interacting with the operating system, including reading and writing files, creating directories, etc.
- numpy: This is a fundamental package for scientific computing in Python. We'll use it for operations on arrays.
- librosa: This is a Python library for music and audio analysis. We will use it to load .wav files and extract features from them.
- sounddevice: This library provides functions for working with audio devices, such as recording and playing sounds.
- sklearn: This is one of the main Python libraries for machine learning. We're using several submodules:
  - sklearn.model_selection.train_test_split: This function is used to split our dataset into a training set and a test set.
  - sklearn.ensemble.RandomForestClassifier: This is the model we'll be using for our classification task.
  - sklearn.preprocessing.LabelEncoder: This class is used to convert categorical labels into numerical labels that can be used with our model.
- joblib: This is a library for saving and loading Python objects to and from disk. We'll use it to save our trained model and label encoder for later use.

Loading and Preprocessing the Data

```python
# Directory where your .wav files are organized in subfolders
main_directory = './ex_6/data/training/'

# List for features and labels
features = []
labels = []

# Iterate over subfolders in main directory
for subfolder in os.listdir(main_directory):
    subfolder_path = os.path.join(main_directory, subfolder)

    # Only process subfolders (skip any file that might be in the main directory)
    if os.path.isdir(subfolder_path):
        # Iterate over files in subfolder
        for filename in os.listdir(subfolder_path):
            if filename.endswith(".wav"):
                # Load .wav file
                y, sr = librosa.load(os.path.join(subfolder_path, filename))

                # Extract MEL spectrogram
                mel = librosa.feature.melspectrogram(y=y, sr=sr)

                # Extract MFCC (mel-frequency cepstral coefficients)
                mfcc = librosa.feature.mfcc(y=y, sr=sr)

                # Append features to the feature list
                features.append(np.hstack((np.mean(mel), np.mean(mfcc))))

                # Append the label (subfolder name) to the labels list
                labels.append(subfolder)
                print(subfolder)
```

In this section, we're loading and preprocessing our data. Here's what each part of the code does:

1. We specify the main directory where our .wav files are stored. Each subfolder in this directory represents a different category or class, and the .wav files in each subfolder are examples of that class.
2. We initialize two empty lists to hold our features and labels.
3. We iterate over each subfolder in the main directory. For each subfolder:

   a. We construct the full path to the subfolder.

   b. We check if this path is actually a directory (it might be a file).

   c. If it is a directory, we iterate over each file in the subfolder.

d. For each file that ends in .wav:

I. We load the audio data using librosa's `load` function, which returns the audio time series and the sample rate.
II. We extract the MEL spectrogram and the mel-frequency cepstral coefficients (MFCCs) from the audio data.
III. We calculate the mean of the MEL and MFCC features and concatenate them into a single feature vector.
IV. We append this feature vector to our list of features, and we append the name of the subfolder (which is our label) to our list of labels.

Preparing Data for the Model

```
# Encode labels
encoder = LabelEncoder()
labels = encoder.fit_transform(labels)

# Split dataset into training and test set
features_train, features_test, labels_train, labels_test =
train_test_split(features, labels, test_size=0.4)
```

In this section, we're preparing our data to be input into our model.

1. We create a LabelEncoder, which will convert our string labels into numerical labels that can be used with our model. We fit the encoder to our labels and transform our labels into numerical form.
2. We split our dataset into a training set and a test set. We're using 40% of our data for the test set, and the remaining 60% for the training set.

Training the Model and Saving the Results

```
# Train model
model = RandomForestClassifier()
model.fit(features_train, labels_train)

# Save model and encoder
joblib.dump(model, 'ex_6/data/models/model.joblib')
joblib.dump(encoder, 'ex_6/data/models/encoder.joblib')
```

Finally, we're training our model and saving the results.

1. We create a RandomForestClassifier, which is the model we'll be using to classify our audio files. We train the model on our training features and labels.
2. We save our trained model and our fitted label encoder to disk using joblib's dump function. We can load these objects later to make predictions on new data.

IMPLEMENTATION OF RECOGNIZER CODE

Importing the Necessary Libraries

```
import numpy as np
import librosa
import joblib
```

In this section, we are importing all the necessary libraries:

- numpy: This library provides support for arrays and matrices, mathematical functions on these structures, and much more.
- librosa: This is a Python library for music and audio analysis. It provides the necessary utilities for extracting features from audio data.
- joblib: This library is used for saving and loading Python objects that have large data arrays. We'll use it to load our previously trained model and label encoder.

Defining Utility Functions

```
def extract_features(file_path):
    y, sr = librosa.load(file_path)
    mel = librosa.feature.melspectrogram(y=y, sr=sr)
    mfcc = librosa.feature.mfcc(y=y, sr=sr)
    features = np.hstack((np.mean(mel), np.mean(mfcc)))
    return features
```

This function takes a file path to a .wav file as input and returns the extracted features. It uses the librosa.load function to load the .wav file. The MEL spectrogram and MFCCs are extracted from the audio data, and then the means of these features are calculated and concatenated together. The resulting feature vector is returned.

```
def classify_audio(file_path, model_path='model.joblib',
encoder_path='encoder.joblib'):
    # Load the model and encoder
    model = joblib.load(model_path)
    encoder = joblib.load(encoder_path)

    # Extract features from the audio file
    features = extract_features(file_path)

    # Make a prediction
    prediction = model.predict([features])

    # Decode the prediction
    label = encoder.inverse_transform(prediction)

    return label
```

This function takes a file path to a .wav file and optional paths to a trained model and label encoder. It uses the extract_features function to extract the features from the .wav file. Then, it uses the model to make a prediction on these features and decodes the numerical prediction into the original label using the encoder. Finally, it returns the decoded label.

Using the Functions to Classify an Audio File

```
# Usage:
file_path = './ex_6/data/testing/notepad_test.wav'
model_path = './ex_6/data/models/model.joblib'
encoder_path = './ex_6/data/models/encoder.joblib'

print(classify_audio(file_path=file_path, model_path=model_path,
encoder_path=encoder_path))
```

In this section, we are using the classify_audio function to classify a .wav file. We provide the paths to the .wav file, the trained model, and the label encoder. The classify_audio function uses these to extract features, make a prediction, and decode the prediction into a label. Finally, we print out the predicted label.

TODO: Lab 2

- **Натренувати модель для розпізнавання інших слів** – потрібно записати звукові файли в форматі WAV з певним словом або словосполученням по попередньому прикладу (ex_6) та натренувати модель на основі нових даних.

- **Об’єднати код розпізнавання з класом MicrophoneDataProvider**. Створити клас який буде містити в собі код для розпізнавання фраз (за основу взяти код з **ex_6_recognizer.py**), але замість завантаження уже записаних файлів використати клас для запису за допомогою мікрофону. Передачу даних між класами можна зробити по аналогії з методом **submit_buffer**.