Please help me implement an encoder and a decoder. The encoder encodes the message to a WAV file and plays with speakers, and the decoder decodes from the microphone to the message.

**ChatGPT 说：**

ChatGPT

Certainly! To implement an encoder and decoder for encoding and decoding a message using audio signals, you'll need to follow these general steps:

1. **Encoder (Message to WAV and Sound Output)**:
   * The encoder will convert a message into an audio signal.
   * We'll use a **modulation** technique to encode the message into sound (e.g., frequency modulation or amplitude modulation).
   * The message is then saved as a WAV file.
   * After saving, we can play the WAV file through the speakers.
2. **Decoder (Microphone Input to Message)**:
   * The decoder will capture the sound using a microphone.
   * Then it will decode the frequency or amplitude pattern back into the original message.

To keep things simple, let’s use **frequency modulation** (FM) for encoding the message and decode the message by analyzing the frequency of the received sound.

**Requirements**

1. **Libraries**:
   * numpy: For handling arrays and signal processing.
   * scipy: For generating and reading WAV files.
   * sounddevice: For playing and recording audio.
   * matplotlib: (Optional) For visualizing signals.

**Step-by-Step Implementation**

**1. Install Required Libraries**

First, you need to install the necessary Python libraries. You can install them with pip:

bash

复制编辑

pip install numpy scipy sounddevice matplotlib

**Encoder: Message to WAV and Sound Output**

python

复制编辑

import numpy as np

import sounddevice as sd

from scipy.io.wavfile import write

import time

def encode\_message(message, sample\_rate=44100, duration\_per\_char=0.5):

# Convert the message to ASCII values and then to binary

binary\_message = ''.join(format(ord(char), '08b') for char in message)

# Parameters for the FM signal

carrier\_frequency = 1000 # Frequency for carrier wave (Hz)

modulation\_index = 100 # Modulation depth (Hz per bit)

# Time settings

total\_time = len(binary\_message) \* duration\_per\_char

t = np.linspace(0, total\_time, int(sample\_rate \* total\_time))

# Initialize an empty waveform to store the signal

waveform = np.zeros\_like(t)

for i, bit in enumerate(binary\_message):

# Frequency shift depending on the bit (1 or 0)

shift = modulation\_index if bit == '1' else -modulation\_index

# Create the FM signal for the current bit

waveform[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)] = np.sin(2 \* np.pi \* (carrier\_frequency + shift) \* t[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)])

# Normalize waveform to fit in 16-bit PCM format

waveform = np.int16(waveform / np.max(np.abs(waveform)) \* 32767)

# Save the audio file as a WAV

write("encoded\_message.wav", sample\_rate, waveform)

# Play the audio through speakers

print("Playing encoded message...")

sd.play(waveform, sample\_rate)

sd.wait() # Wait for the audio to finish playing

print("Playback finished.")

# Example of encoding and playing a message

message = "Hello"

encode\_message(message)

**Decoder: Microphone Input to Message**

python

复制编辑

import numpy as np

import sounddevice as sd

from scipy.io.wavfile import read

import time

def decode\_message(filename="encoded\_message.wav", sample\_rate=44100, duration\_per\_char=0.5):

# Read the WAV file

\_, waveform = read(filename)

# Parameters for decoding

carrier\_frequency = 1000

modulation\_index = 100

# Time array corresponding to the audio signal

t = np.arange(len(waveform)) / sample\_rate

# Frequency analysis (using FFT or zero-crossing detection)

bit\_duration = int(sample\_rate \* duration\_per\_char)

decoded\_message = ''

for i in range(0, len(waveform), bit\_duration):

# Extract the segment of the signal

segment = waveform[i:i + bit\_duration]

if len(segment) < bit\_duration:

break

# Apply FFT to determine the dominant frequency in the segment

freqs = np.fft.fftfreq(len(segment), 1 / sample\_rate)

fft\_vals = np.fft.fft(segment)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Determine whether it is a 1 or 0 based on frequency shift

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

decoded\_message += '0'

else:

decoded\_message += '1'

# Convert binary message back to text

decoded\_text = ''.join(chr(int(decoded\_message[i:i + 8], 2)) for i in range(0, len(decoded\_message), 8))

print(f"Decoded message: {decoded\_text}")

# Example of decoding the message from microphone input or the saved file

decode\_message("encoded\_message.wav")

**How It Works:**

* **Encoder**:
  + The message is converted to binary, and each bit is encoded by changing the frequency of a sine wave (FM modulation).
  + The waveform is saved as a WAV file and played through speakers.
* **Decoder**:
  + The microphone captures the audio, and the signal is analyzed using FFT (Fast Fourier Transform) to determine the dominant frequency.
  + If the frequency shift indicates a modulation pattern, it’s decoded as a binary 1 or 0.
  + The binary message is then converted back to text.

**Key Points to Consider:**

* **Noise and Distortions**: This is a simple frequency modulation method, and real-world noise could cause decoding errors, so further techniques like error correction might be needed for a more robust system.
* **Sampling Rate**: Ensure that the sampling rate is consistent between encoding and decoding.
* **Modulation Scheme**: The frequency modulation scheme used here is basic. You can explore other techniques, like Phase Shift Keying (PSK), Amplitude Modulation (AM), or even more sophisticated methods for better noise immunity.

This should get you started on building an encoder-decoder system using audio signals! Let me know if you need any further clarifications or modifications.

4o

**您说：**

We need to run encoder and decoder on different computer simultaneously. We can't use WAV file as input/output. We are only use speaker and microphone to transfer message.

**ChatGPT 说：**

ChatGPT

Ah, got it! If you need to run the encoder and decoder on different computers simultaneously and cannot use files (such as WAV) for input/output, you'll need to stream the audio directly between the two systems. In this case, the encoder will send the audio (as a live stream) to the decoder using the speakers and microphone as input/output devices.

**Here's how we can break this down:**

* **Encoder**: The encoder will modulate the message into an audio signal (using frequency modulation, for example) and transmit it via the speakers.
* **Decoder**: The decoder will capture the audio from the microphone, process it, and decode the message.

We can achieve this using **UDP sockets** to stream the audio data between the two computers over the network. The encoder will encode the message and send the audio via UDP, and the decoder will receive this stream and decode it in real time.

**Outline of the Plan:**

1. **Encoder**:
   * Encode the message into audio using frequency modulation (FM).
   * Send the audio in chunks over a UDP socket to the decoder.
2. **Decoder**:
   * Listen to the UDP socket for incoming audio.
   * Decode the message in real time by analyzing the frequencies in the audio stream.

We'll use sounddevice to handle real-time audio input/output and socket for network communication. Additionally, the encoder will continuously stream the audio, and the decoder will process the incoming stream.

**1. Encoder: Sending Audio via UDP**

Here's how the **encoder** can stream audio to a decoder using UDP:

python

复制编辑

import numpy as np

import sounddevice as sd

import socket

import time

# Encoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

host = 'localhost' # IP of the decoder

port = 5000 # UDP port

def encode\_message(message):

# Convert the message to binary

binary\_message = ''.join(format(ord(char), '08b') for char in message)

# Prepare the audio stream buffer

total\_time = len(binary\_message) \* duration\_per\_char

t = np.linspace(0, total\_time, int(sample\_rate \* total\_time))

# Initialize an empty waveform

waveform = np.zeros\_like(t)

# Create the FM signal for each bit in the binary message

for i, bit in enumerate(binary\_message):

shift = modulation\_index if bit == '1' else -modulation\_index

waveform[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)] = np.sin(2 \* np.pi \* (carrier\_frequency + shift) \* t[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)])

# Normalize the waveform

waveform = np.int16(waveform / np.max(np.abs(waveform)) \* 32767)

# Send the waveform over UDP

with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as sock:

for i in range(0, len(waveform), sample\_rate): # Send in chunks

chunk = waveform[i:i + sample\_rate] # 1 second per chunk

sock.sendto(chunk.tobytes(), (host, port)) # Send the chunk to the decoder

time.sleep(1) # Wait for 1 second before sending the next chunk

print("Message encoded and sent.")

# Example of encoding and streaming a message

message = "Hello"

encode\_message(message)

**2. Decoder: Receiving Audio and Decoding the Message**

Now, the **decoder** will receive the audio stream from the microphone and analyze the frequency shifts to decode the message:

python

复制编辑

import numpy as np

import sounddevice as sd

import socket

# Decoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

port = 5000 # Same port as encoder

def decode\_message():

# Create a UDP socket to receive data

with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as sock:

sock.bind(('0.0.0.0', port)) # Listen on all interfaces

# Initialize an empty buffer to store received data

waveform = np.array([], dtype=np.int16)

while True:

# Receive data from encoder (audio stream)

chunk, \_ = sock.recvfrom(2048) # Receive audio chunk (1 second at a time)

new\_chunk = np.frombuffer(chunk, dtype=np.int16)

waveform = np.concatenate((waveform, new\_chunk))

# Process every 'duration\_per\_char' seconds of data for decoding

if len(waveform) > int(sample\_rate \* duration\_per\_char):

segment = waveform[:int(sample\_rate \* duration\_per\_char)] # Take the first segment

waveform = waveform[int(sample\_rate \* duration\_per\_char):] # Remove the processed segment

# Apply FFT to find the dominant frequency

freqs = np.fft.fftfreq(len(segment), 1 / sample\_rate)

fft\_vals = np.fft.fft(segment)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Decode the bit based on frequency shift

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

decoded\_bit = '0'

else:

decoded\_bit = '1'

print(decoded\_bit, end='', flush=True)

# Start the decoder to listen for messages

decode\_message()

**How the System Works:**

1. **Encoder**:
   * The encoder encodes the message into a frequency-modulated audio signal.
   * The signal is split into chunks, and each chunk is sent over a UDP socket.
2. **Decoder**:
   * The decoder listens on a specified UDP port and receives the audio chunks.
   * Each chunk is processed to identify the frequency shift (corresponding to binary 0 or 1).
   * The decoded bits are displayed as the message is received.

**Key Points:**

* **UDP Streaming**: The encoder sends small chunks of the audio stream to the decoder using UDP. This allows the decoder to receive the audio in real time and decode it.
* **Real-time Decoding**: The decoder processes each chunk as it arrives, so the message is decoded live.
* **Frequency Modulation**: The encoding relies on frequency modulation, where each bit is represented by a frequency shift (e.g., +100 Hz for 1 and -100 Hz for 0).
* **Network**: Ensure that both the encoder and decoder are on the same network or have proper network configurations (such as port forwarding) if they are on different networks.

**Potential Improvements:**

* **Error Handling**: In real-world scenarios, you might want to add checks and error correction to handle issues like packet loss or noise.
* **Synchronization**: For long messages, consider adding synchronization markers so the decoder can correctly segment the data stream.
* **Compression**: If you're working with larger messages or need a more efficient encoding scheme, you could explore using compression algorithms.

This system allows you to send real-time audio messages from one computer to another without the need for files, using only the microphone and speakers! Let me know if you have any questions or need further help!

4o

**您说：**

We can't use socket. There isn't a network.

**ChatGPT 说：**

ChatGPT

Got it! Since there’s no network, you’ll need to use **local audio input/output** (via speakers and microphones) to transfer the message. In this case, the encoder will play the audio through the speakers, and the decoder will listen to the microphone.

This requires two main things:

1. **Encoder**: To generate the modulated audio and play it through the speakers.
2. **Decoder**: To capture the audio via the microphone and decode it in real-time.

We’ll use sounddevice to handle real-time audio input and output, and instead of sending audio via a network, we’ll rely on the audio environment (i.e., speakers and microphones).

**Plan:**

* **Encoder** will convert the message into an audio signal using **frequency modulation** (FM) and play it through the speakers.
* **Decoder** will listen to the microphone, capture the audio, and decode it in real-time based on the frequency shifts in the signal.

**Updated Solution**

Since there's no network, we'll use the **microphone** and **speakers** to capture and transmit the signal directly between the two systems. The encoder will play a sound, and the decoder will record and decode it.

We'll proceed with **frequency modulation** as the method of encoding the binary message.

**1. Encoder (Message to Audio Stream via Speakers)**

The encoder will modulate the message (each character into a binary stream) and then use FM to encode the message. The sound will be played through the speakers.

python

复制编辑

import numpy as np

import sounddevice as sd

import time

# Encoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

def encode\_message(message):

# Convert the message to binary (ASCII to binary)

binary\_message = ''.join(format(ord(char), '08b') for char in message)

# Prepare the audio stream buffer

total\_time = len(binary\_message) \* duration\_per\_char

t = np.linspace(0, total\_time, int(sample\_rate \* total\_time))

# Initialize an empty waveform

waveform = np.zeros\_like(t)

# Create the FM signal for each bit in the binary message

for i, bit in enumerate(binary\_message):

shift = modulation\_index if bit == '1' else -modulation\_index

waveform[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)] = np.sin(2 \* np.pi \* (carrier\_frequency + shift) \* t[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)])

# Normalize the waveform

waveform = np.int16(waveform / np.max(np.abs(waveform)) \* 32767)

# Play the audio through the speakers

print("Playing encoded message...")

sd.play(waveform, sample\_rate)

sd.wait() # Wait until the audio finishes playing

print("Playback finished.")

# Example of encoding and playing a message

message = "Hello"

encode\_message(message)

**2. Decoder (Capture Audio from Microphone and Decode)**

The decoder will listen to the microphone, capture the audio stream, and decode the message by detecting the frequency shifts.

python

复制编辑

import numpy as np

import sounddevice as sd

# Decoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

def decode\_message():

# Listen to the microphone and capture the audio stream

print("Listening for incoming message...")

duration\_per\_segment = int(sample\_rate \* duration\_per\_char) # Duration of one message bit

# Initialize an empty buffer for the audio

audio\_buffer = np.array([], dtype=np.int16)

# Continuously capture audio and decode

with sd.InputStream(callback=callback):

sd.sleep(10000) # Capture for a period of time (e.g., 10 seconds)

print("Captured audio.")

def callback(indata, frames, time, status):

if status:

print(status, flush=True)

# Process the incoming audio buffer to detect frequency shifts

audio\_buffer = np.concatenate((audio\_buffer, indata[:, 0])) # Assume single-channel audio for simplicity

if len(audio\_buffer) > duration\_per\_segment:

segment = audio\_buffer[:duration\_per\_segment] # Take the first segment

audio\_buffer = audio\_buffer[duration\_per\_segment:] # Remove the processed segment

# Apply FFT to find the dominant frequency

freqs = np.fft.fftfreq(len(segment), 1 / sample\_rate)

fft\_vals = np.fft.fft(segment)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Decode the bit based on frequency shift

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

decoded\_bit = '0'

else:

decoded\_bit = '1'

print(decoded\_bit, end='', flush=True)

# Start the decoder to listen for messages

decode\_message()

**How It Works:**

* **Encoder**:
  + The encoder converts the message into binary, then modulates the bits using frequency modulation (FM) and plays it through the speakers.
  + Each bit of the message is represented by a slight shift in the frequency (e.g., +100 Hz for '1' and -100 Hz for '0').
* **Decoder**:
  + The decoder continuously listens to the microphone. It processes chunks of audio and analyzes the frequency shift in each chunk using the **Fast Fourier Transform (FFT)**.
  + The frequency shift indicates whether the bit is a 1 or 0, based on whether the dominant frequency is slightly above or below the carrier frequency.

**Key Points:**

* **Real-time Audio Streaming**: The encoder streams the modulated audio through the speakers, and the decoder listens to the microphone for real-time decoding.
* **FM Modulation**: The frequency shift between the two states (1 and 0) allows for encoding the binary data in sound.
* **Microphone/Speaker Use**: No network is needed. The systems rely on the **local microphone and speaker** to send and receive data.
* **Decoding**: The decoder listens to the audio, uses FFT to extract the dominant frequency, and decodes the bits based on the frequency shift.

**Improvements and Notes:**

1. **Microphone and Speaker Quality**: Ensure that both the microphone and speakers are capable of handling the required frequency ranges. Distortion, noise, or limitations in the microphone’s frequency response could impact the accuracy of the decoder.
2. **Error Handling**: In real-world situations, ambient noise, distortions, or interruptions in the audio might affect the decoding. You could add error detection or retries to improve robustness.
3. **Synchronization**: The decoder should be synchronized to start capturing at the right moment. You can add a simple preamble (a known bit pattern) at the beginning of the transmission to help the decoder sync up.

This setup allows for **direct audio communication** between two systems using just speakers and microphones, without relying on any network connections or external file storage.

Let me know if you need any further modifications or explanations!

4o

**您说：**

Please handle the audio noise, and help the decoder sync up

**ChatGPT 说：**

ChatGPT

Sure! To make this system more robust, we need to address two main issues:

1. **Handling Audio Noise**: We need to filter out noise and ensure that the decoder can still correctly detect the frequency shifts, even in the presence of environmental noise.
2. **Synchronization**: The decoder needs to correctly sync up with the audio stream, especially if the encoder and decoder are not perfectly timed. We'll use a **preamble** or sync signal to help the decoder align and start decoding at the correct position.

**Plan for Improvements:**

1. **Audio Noise Handling**:
   * **Noise Filtering**: We'll use a **low-pass filter** to remove high-frequency noise that may interfere with the signal.
   * **Thresholding**: We'll apply a threshold to detect significant frequency shifts that correspond to the modulated bits (1 and 0).
   * **Decoding Confidence**: We can introduce a confidence measure to ensure the decoded bits are reliable.
2. **Synchronization**:
   * **Preamble**: We'll add a known "preamble" (a sequence of 1 and 0 bits) at the start of each transmission to help the decoder identify the start of the message.
   * **Time Synchronization**: The decoder should search for the preamble and begin decoding once it finds the correct starting point.

**Updated Solution**

Let's make the necessary changes to handle noise and improve synchronization:

**1. Encoder: Encoding with Preamble and Noise Robustness**

We'll add a simple **preamble** consisting of a known sequence of bits (e.g., a series of 1s) to help the decoder detect the start of the transmission. The encoder will also apply the modulation as before.

python

复制编辑

import numpy as np

import sounddevice as sd

import time

# Encoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

preamble = "11111111" # Preamble for synchronization (known sequence of bits)

def encode\_message(message):

# Convert the message to binary (ASCII to binary)

binary\_message = ''.join(format(ord(char), '08b') for char in message)

# Add preamble to help the decoder sync up

binary\_message = preamble + binary\_message

# Prepare the audio stream buffer

total\_time = len(binary\_message) \* duration\_per\_char

t = np.linspace(0, total\_time, int(sample\_rate \* total\_time))

# Initialize an empty waveform

waveform = np.zeros\_like(t)

# Create the FM signal for each bit in the binary message

for i, bit in enumerate(binary\_message):

shift = modulation\_index if bit == '1' else -modulation\_index

waveform[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)] = np.sin(2 \* np.pi \* (carrier\_frequency + shift) \* t[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)])

# Normalize the waveform

waveform = np.int16(waveform / np.max(np.abs(waveform)) \* 32767)

# Play the audio through the speakers

print("Playing encoded message with preamble...")

sd.play(waveform, sample\_rate)

sd.wait() # Wait until the audio finishes playing

print("Playback finished.")

# Example of encoding and playing a message

message = "Hello"

encode\_message(message)

**2. Decoder: Handling Noise and Synchronizing**

The decoder needs to:

* **Filter out noise**: We’ll use a **low-pass filter** to remove high-frequency noise.
* **Synchronize**: We’ll detect the preamble and start decoding once we find it.

**2.1. Noise Filtering with a Low-Pass Filter**

We'll use a simple **low-pass filter** (e.g., Butterworth filter) to remove high-frequency noise that may interfere with the frequency detection.

**2.2. Synchronization with Preamble Detection**

The decoder will look for the **preamble** (a known sequence of 1s) in the incoming audio. Once it finds the preamble, it will start decoding.

Here’s the updated **decoder**:

python

复制编辑

import numpy as np

import sounddevice as sd

from scipy.signal import butter, filtfilt

import time

# Decoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

preamble = "11111111" # Preamble for synchronization (known sequence of bits)

preamble\_bits = ''.join(format(ord(char), '08b') for char in preamble) # Convert preamble to binary

# Low-pass filter for noise reduction

def butter\_lowpass(cutoff, fs, order=5):

nyquist = 0.5 \* fs

normal\_cutoff = cutoff / nyquist

b, a = butter(order, normal\_cutoff, btype='low', analog=False)

return b, a

def lowpass\_filter(data, cutoff, fs, order=5):

b, a = butter\_lowpass(cutoff, fs, order)

return filtfilt(b, a, data)

def decode\_message():

# Start listening to the microphone

print("Listening for incoming message...")

audio\_buffer = np.array([], dtype=np.float32) # Audio buffer

sync\_found = False # Flag to indicate if synchronization has been achieved

decoded\_message = ""

# Continuously capture audio and decode

with sd.InputStream(callback=callback):

sd.sleep(10000) # Capture for a period of time (e.g., 10 seconds)

print("Captured audio.")

# Check if we synchronized correctly

if sync\_found:

print(f"Decoded message: {decoded\_message}")

else:

print("Synchronization failed.")

def callback(indata, frames, time, status):

nonlocal audio\_buffer, sync\_found, decoded\_message

if status:

print(status, flush=True)

# Append incoming audio to the buffer

audio\_buffer = np.concatenate((audio\_buffer, indata[:, 0]))

# Apply a low-pass filter to remove high-frequency noise

filtered\_audio = lowpass\_filter(audio\_buffer, cutoff=1500, fs=sample\_rate)

# Check for synchronization (search for the preamble)

if not sync\_found:

# Check if the preamble is present in the filtered signal

preamble\_found = detect\_preamble(filtered\_audio)

if preamble\_found:

print("Preamble detected. Synchronizing...")

sync\_found = True # Start decoding after preamble

decoded\_message = "" # Reset message buffer

audio\_buffer = np.array([]) # Clear audio buffer to start fresh

# If synchronized, decode the message

if sync\_found:

# Process the audio in chunks and decode

chunk\_duration = int(sample\_rate \* duration\_per\_char)

if len(filtered\_audio) > chunk\_duration:

segment = filtered\_audio[:chunk\_duration] # Get the first segment

audio\_buffer = filtered\_audio[chunk\_duration:] # Keep the remaining audio

# Apply FFT to find the dominant frequency

freqs = np.fft.fftfreq(len(segment), 1 / sample\_rate)

fft\_vals = np.fft.fft(segment)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Decode the bit based on frequency shift

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

decoded\_message += '0'

else:

decoded\_message += '1'

# Print the decoded bits for feedback

print(decoded\_message[-1], end='', flush=True)

# Search for the preamble in the audio buffer

def detect\_preamble(audio\_data):

# Convert audio to frequency domain using FFT

freqs = np.fft.fftfreq(len(audio\_data), 1 / sample\_rate)

fft\_vals = np.fft.fft(audio\_data)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Check if the frequency shift matches the modulation pattern of the preamble (known pattern of 1s)

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

return True

return False

# Start the decoder

decode\_message()

**Explanation of Changes:**

1. **Low-Pass Filtering**:
   * The lowpass\_filter function uses a **Butterworth filter** to remove high-frequency noise from the captured audio. This helps isolate the frequency modulation and improve the decoder's ability to detect the modulated signal.
2. **Preamble Synchronization**:
   * The detect\_preamble function looks for the preamble (a known sequence of bits, e.g., 11111111) in the audio stream. Once the preamble is detected, the decoder starts decoding the message.
3. **FFT Frequency Detection**:
   * The **FFT (Fast Fourier Transform)** is used to identify the dominant frequency in each segment of the audio. The decoder compares this frequency with the expected carrier\_frequency and adjusts for any frequency shift, allowing it to decode the binary 0 or 1.
4. **Real-time Decoding**:
   * The decoder processes the audio stream in real-time,

4o

**您说：**

Handle "end of message"

**ChatGPT 说：**

ChatGPT

To handle the "end of message" (EOM) properly, we need to establish a clear indicator or signal that marks the end of the transmitted message. This will help the decoder know when to stop processing the stream and finalize the message.

**Plan:**

1. **End of Message (EOM) Marker**:
   * The encoder will append a special sequence (e.g., a series of 0s or a specific frequency shift) to the message to signal the end.
   * The decoder will look for this special marker and stop decoding once it is detected.
2. **Handling End of Message in the Encoder**:
   * We will add a specific **EOM pattern** (e.g., a sequence of 00000000 or a long, silent period) at the end of the message.
3. **Handling End of Message in the Decoder**:
   * The decoder will check for the EOM pattern or frequency silence, indicating the end of the message.
   * Once the EOM pattern is detected, it will finalize the message and stop listening for further transmission.

**Updated Encoder: Adding End of Message (EOM) Marker**

We will append a specific EOM pattern to the message. For simplicity, let’s use a long sequence of 0s (or silence) to mark the end of the message.

Here’s how we can modify the encoder to add an EOM marker:

python

复制编辑

import numpy as np

import sounddevice as sd

import time

# Encoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

preamble = "11111111" # Preamble for synchronization (known sequence of bits)

eom\_pattern = "00000000" # EOM pattern (end of message)

def encode\_message(message):

# Convert the message to binary (ASCII to binary)

binary\_message = ''.join(format(ord(char), '08b') for char in message)

# Add preamble to help the decoder sync up

binary\_message = preamble + binary\_message + eom\_pattern # Append EOM pattern

# Prepare the audio stream buffer

total\_time = len(binary\_message) \* duration\_per\_char

t = np.linspace(0, total\_time, int(sample\_rate \* total\_time))

# Initialize an empty waveform

waveform = np.zeros\_like(t)

# Create the FM signal for each bit in the binary message

for i, bit in enumerate(binary\_message):

shift = modulation\_index if bit == '1' else -modulation\_index

waveform[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)] = np.sin(2 \* np.pi \* (carrier\_frequency + shift) \* t[i\*int(sample\_rate\*duration\_per\_char):(i+1)\*int(sample\_rate\*duration\_per\_char)])

# Normalize the waveform

waveform = np.int16(waveform / np.max(np.abs(waveform)) \* 32767)

# Play the audio through the speakers

print("Playing encoded message with preamble and EOM...")

sd.play(waveform, sample\_rate)

sd.wait() # Wait until the audio finishes playing

print("Playback finished.")

# Example of encoding and playing a message

message = "Hello"

encode\_message(message)

**Updated Decoder: Detecting End of Message (EOM)**

The decoder will:

1. Continuously listen for incoming audio.
2. Look for the EOM pattern (e.g., a sequence of 0s or a silent period) to indicate the end of the message.
3. Stop decoding when it detects the EOM marker.

Here’s the updated **decoder** that detects the EOM marker:

python

复制编辑

import numpy as np

import sounddevice as sd

from scipy.signal import butter, filtfilt

import time

# Decoder Configuration

carrier\_frequency = 1000 # Hz, carrier frequency for FM

modulation\_index = 100 # Hz, modulation depth

sample\_rate = 44100 # Hz

duration\_per\_char = 0.5 # seconds per character

preamble = "11111111" # Preamble for synchronization (known sequence of bits)

eom\_pattern = "00000000" # EOM pattern (end of message)

preamble\_bits = ''.join(format(ord(char), '08b') for char in preamble) # Convert preamble to binary

eom\_bits = ''.join(format(ord(char), '08b') for char in eom\_pattern) # Convert EOM to binary

# Low-pass filter for noise reduction

def butter\_lowpass(cutoff, fs, order=5):

nyquist = 0.5 \* fs

normal\_cutoff = cutoff / nyquist

b, a = butter(order, normal\_cutoff, btype='low', analog=False)

return b, a

def lowpass\_filter(data, cutoff, fs, order=5):

b, a = butter\_lowpass(cutoff, fs, order)

return filtfilt(b, a, data)

def decode\_message():

# Start listening to the microphone

print("Listening for incoming message...")

audio\_buffer = np.array([], dtype=np.float32) # Audio buffer

sync\_found = False # Flag to indicate if synchronization has been achieved

decoded\_message = ""

# Continuously capture audio and decode

with sd.InputStream(callback=callback):

sd.sleep(10000) # Capture for a period of time (e.g., 10 seconds)

print("Captured audio.")

# Check if we synchronized correctly

if sync\_found:

print(f"Decoded message: {decoded\_message}")

else:

print("Synchronization failed.")

def callback(indata, frames, time, status):

nonlocal audio\_buffer, sync\_found, decoded\_message

if status:

print(status, flush=True)

# Append incoming audio to the buffer

audio\_buffer = np.concatenate((audio\_buffer, indata[:, 0]))

# Apply a low-pass filter to remove high-frequency noise

filtered\_audio = lowpass\_filter(audio\_buffer, cutoff=1500, fs=sample\_rate)

# Check for synchronization (search for the preamble)

if not sync\_found:

# Check if the preamble is present in the filtered signal

preamble\_found = detect\_preamble(filtered\_audio)

if preamble\_found:

print("Preamble detected. Synchronizing...")

sync\_found = True # Start decoding after preamble

decoded\_message = "" # Reset message buffer

audio\_buffer = np.array([]) # Clear audio buffer to start fresh

# If synchronized, decode the message

if sync\_found:

# Process the audio in chunks and decode

chunk\_duration = int(sample\_rate \* duration\_per\_char)

if len(filtered\_audio) > chunk\_duration:

segment = filtered\_audio[:chunk\_duration] # Get the first segment

audio\_buffer = filtered\_audio[chunk\_duration:] # Keep the remaining audio

# Apply FFT to find the dominant frequency

freqs = np.fft.fftfreq(len(segment), 1 / sample\_rate)

fft\_vals = np.fft.fft(segment)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Decode the bit based on frequency shift

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

decoded\_message += '0'

else:

decoded\_message += '1'

# Print the decoded bits for feedback

print(decoded\_message[-1], end='', flush=True)

# Check for EOM pattern (end of message)

if decoded\_message.endswith(eom\_bits):

print("\nEnd of message detected. Stopping decoding.")

sync\_found = False # Stop further decoding

# Search for the preamble in the audio buffer

def detect\_preamble(audio\_data):

# Convert audio to frequency domain using FFT

freqs = np.fft.fftfreq(len(audio\_data), 1 / sample\_rate)

fft\_vals = np.fft.fft(audio\_data)

idx = np.argmax(np.abs(fft\_vals))

dominant\_freq = np.abs(freqs[idx])

# Check if the frequency shift matches the modulation pattern of the preamble (known pattern of 1s)

if np.abs(dominant\_freq - carrier\_frequency) < modulation\_index / 2:

return True

return False

# Start the decoder

decode\_message()

**Key Updates:**

1. **EOM Pattern**:
   * The **EOM marker** (a series of 0s) is appended to the message at the encoder side.
   * The decoder checks for this pattern using decoded\_message.endswith(eom\_bits) to detect when the message ends.
2. **Synchronization and Stopping**:
   * The decoder first looks for the **preamble** to sync up.
   * Once synchronized, it starts decoding the message.
   * After decoding, the decoder continuously checks for the EOM pattern to detect the end of the message. When the pattern is detected, the decoding stops.
3. **Low-Pass Filtering**: The low-pass filter still helps reduce noise in the captured audio.

**Final Thoughts:**

This solution handles both **noise reduction** and **synchronization** with the addition of the EOM pattern. The decoder knows exactly when to stop decoding based on the EOM signal, making the system more reliable in noisy environments and ensuring that the message is properly terminated.

Let me know if you need further adjustments