Project 3

DTMF Signaling

Matthew Lopez Tarsky
[Redacted Member #1]
[Redacted Member #2]

ENGR 285

Objective #1: What the encoding program does and how.

This program generates a DTMF (Dual-tone multi-frequency) signal that encodes a sequence of digits (from 0 to 9) into sound and saves it as a **.wav** file. DTMF signals are commonly used in telecommunication systems, such as the tones you hear when pressing keys on a telephone. The program works as follows:

Libraries Used:

- **numpy:** Used for numerical operations and generating sound waveforms.
- **wave:** A Python standard library used to write the **.wav** audio file.
- **struct:** Used to pack data into binary format, which is required to write the sound data into the **.wav** file.

File Setup:

- **fileName = "EncodedSound.wav":** This specifies the name of the output **.wav** file.
- **numberList = [1,1,2,1,0,1]:** This is the list of digits (0-9) that will be encoded into sound. You can change this list to encode different numbers.

Audio Parameters:

- **sampleRate = 44100**: The sample rate for the audio. 44100 samples per second is a common standard for audio files.
- **soundLevel = 4096:** The amplitude of the sound. This controls how loud the sound is.
- **soundLength = 400:** The duration of each tone, in milliseconds.
- **pauseLength = 200:** The duration of the pause between each tone, in milliseconds.

Tone Creation (DTMF Frequencies): The program defines DTMF tones based on pairs of frequencies. A combination of two frequencies represents each digit (0-9). The following frequencies are used in the program:

- **697 Hz, 770 Hz, 852 Hz, and 941 Hz** for the low frequencies.
- **1209 Hz, 1336 Hz, and 1477 Hz** for the high frequencies.

The function **createPureToneData(freq)** generates a pure tone at a given frequency using a sine wave equation:

$$sin(\frac{2\pi f x}{sampleRate})$$

- This generates a series of values corresponding to a sine wave at the specified frequency. The tone is created for one second (i.e., the sample rate).
- **Tone Mapping for Each Digit: toneList** maps each digit to a combination of two frequencies:
    - Each pair of frequencies corresponds to a specific key on a telephone keypad. For example, the digit "1" corresponds to the pair (697 Hz, 1209 Hz), "2" corresponds to (697 Hz, 1336 Hz), and so on.
    - The **toneList** is created by combining the sine waves for each pair of frequencies.
- **Sound Data Generation:**
    - **soundData**: An empty list to store the final waveform data for all the tones and pauses.
    - For each digit in **numberList**, the corresponding tone from **toneList** is added to **soundData** for the duration specified by **soundLength** (400 milliseconds). The code takes the first part of the tone data, equal to the number of samples corresponding to 400 milliseconds.
    - After each tone, a pause (silent part) is added to separate the tones. The pause length is specified as **pauseLength** (200 milliseconds), and this silence is represented by zeros in the sound data.
- **Writing the WAV File:** The program then writes the audio data to a **.wav** file:
    - **wav_file = wave.open(fileName, "w"):** Opens a new **.wav** file for writing.
    - **wav_file.setparams**: Sets parameters for the **.wav** file, such as the number of channels (mono, **nchannels = 1**), sample width (2 bytes per sample), frame rate (sample rate), and total number of frames **(nframes**).
    - The number of frames is calculated based on the length of each tone and pause multiplied by the number of digits in **numberList.**
    - **wav_file.writeframes**: Writes the data to the file using the **struct.pack('h', int(s))** to pack each sample as a 16-bit signed integer (the format required for **.wav** files).
- **Completion:**
    - The **wav_file.close()** statement closes the file once all the data has been written.
    - The program prints **Writing EncodedSound.wav complete!** to confirm that the file has been created successfully.

Overview

- The program generates a sequence of DTMF tones based on the **numberList** provided (for example, the sequence [1, 1, 2, 1, 0, 1]).
- Each digit is encoded by combining two sine wave frequencies corresponding to that digit.
- The tones are generated for 400 milliseconds, followed by a 200-millisecond pause.
- The generated audio data is saved as a **.wav** file (**EncodedSound.wav**).

Example Use:

If you run the program with the numberList = [1, 1, 2, 1, 0, 1], it will generate a **.wav** file with the sequence of DTMF tones for the digits "1", "1", "2", "1", "0", and "1". This audio file can then be played on a speaker or used for other purposes that require encoding numeric data in sound.

Objective #2: In the decoding program's "slice_data()" function, what conditions need to be added to the **if**-block and the second **while**-block in order for the function to successfully return a list of the individual DTMF signals' data without pauses *(Note of Caution: A sound signal can be zero momentarily without being silent!)*

In the `slice_data()` function, the conditions within the if-block and the while-loop are critical for isolating individual DTMF signals without mistakenly treating momentary zero values as silence. Here's how these conditions work and how they ensure proper slicing:

Conditions in the if-block:

```
if save_data[i] == 0:
    i += 1
```

This condition skips over any initial zero values, assuming they are part of silent sections. However, this alone is insufficient because sound signals can also momentarily pass through zero. Therefore, additional handling in the while-block is necessary.

Conditions in the second while block:

```
while(save_data[j+i+1] != 0 or save_data[j+i-1] != 0):
    current_signal.append(save_data[i+j])
    j += 1
```

Explanation:

1. `save_data[j+i+1] != 0` and `save_data[j+i-1] != 0`:

- These conditions check the values immediately before and after the current index.
- This ensures that a zero value within the signal (a momentary crossing of zero) is not misinterpreted as silence.
- It requires continuity of non-zero data in at least one direction to consider the segment as part of the same signal.

2. Continuity Checking:

- The while loop continues to append values to current_signal as long as there are adjacent non-zero values, avoiding prematurely ending the signal extraction.
- This prevents breaking the signal when the amplitude of the waveform temporarily crosses zero.

3. Advancing the Index:

- `j += 1` ensures the loop moves forward to capture the full segment of the DTMF signal without pauses.
- Once a true pause (sustained zeros in both directions) is encountered, the loop terminates.

4. Why These Conditions Work

- Momentary Zeros: The while loop's continuity check prevents zeros that are part of the waveform from being mistaken for silence.
- Silent Pauses: Sustained zeros on both sides indicate the end of a DTMF signal, allowing the function to correctly identify signal boundaries.

Objective #3: In the decoding program's "calculate_coefficient()" function, what code needs to be added to the **two** for-block lines for the function to calculate the <u>approximate</u> Fourier coefficients of the input signal data "dataSample" corresponding to the input frequency "freq." In other words, how you will adapt the terms in the equations $a_\nu = \frac{2}{T}\int_0^T f(t)cos(2\pi\nu t)dt$ and $b_\nu = \frac{2}{T}\int_0^T f(t)sin(2\pi\nu t)dt$ in order to approximately calculate the Fourier coefficients corresponding to a frequency ν from the data points $f(t)$.

$$a_\nu = \frac{2}{T}\int_0^T f(t)cos(2\pi\nu t)dt\sum \qquad b_\nu = \frac{2}{T}\int_0^T f(t)sin(2\pi\nu t)dt$$

$f(t)$: the signal as a function of t
T: Duration of the signal
v: The target frequency to analyze (i.e 697Hz and 770Hz)

Since we are dealing with discrete/non-continuous data, the integral must be approximated as a summation of the data points in "data sample." We will use the basic summation or trapezoidal rule for this, where the sampling rate dictates the step size.

$$a_v = \frac{2}{T}\sum_{i=0}^{N-1} f(t_i)cos(2\pi v t_i) \qquad b_v = \frac{2}{T}\sum_{i=0}^{N-1} f(t_i)sin(2\pi\nu t)$$

N: Total number of samples in the signal.
$t_i = i/framerate$: The time associated with sample i
$T = N/framerate$: The signal's overall duration, depending on the sampling rate(framerate) and sample count

With n representing the number of samples in "data sample" and "framerate" representing the number of samples per second, compute $T = N/framerate$.

Determine the time associated with the sample by computing ti=i/framerate. With $v = f$, multiply the sample value $f(t_i)$ by $cos(2\pi v t_i)$ and $sin(2\pi v t_i)$.

To estimate av and bv, add up all of the values.

The sums can be normalized by multiplying them by $2/N$,.

The magnitude of the Fourier coefficient is $\sqrt{a_v^2 + b_v^2}$

Original code:

```
def calculate_coefficient(dataSample, freq):
    a = 0
    b = 0
    for i in range(len(dataSample)):
        a +=
        b +=
    return sqrt(a**2 + b**2)
```

To:

```
def calculate_coefficient(dataSample, freq):
    a = 0
    b = 0
    T = len(dataSample) / framerate
    N = len(dataSample)

    for i in range(N):
        t = i / framerate
        a += dataSample[i] * cos(2 * pi * freq * t)
        b += dataSample[i] * sin(2 * pi * freq * t)


    a *= 2 / N
    b *= 2 / N

    return sqrt(a**2 + b**2)
```

The variables "a" and "b" begin at 0 since the sine and cosine projections will be accumulated, respectively.

"*T*" is the overall time of the signal, which is determined by the sampling rate "framerate" and the number of samples `len(dataSample)`. Determine the relevant time `t=i/framerate` for each index "i" in the data sample. The sums "a" and "b" should be increased by the contributions of `f(ti)cos(2πvti)` and `f(ti)sin(2πvti)`.

To consider the number of samples and translate the sums into estimated Fourier coefficients, multiply "a" and "b" by $\frac{2}{N}$. The size of the signal at the desired frequency is ultimately represented by the square root of the sum of squares of a and b.

Objective #4: In the decoding program's main **for**-loop, what lines need to be added so that the program will output the decoded digits of the total sound.

Now that we have derived a method of extracting the Fourier coefficients, we can use it to extract the digits of our program. To do this, we must analyze each signal of the sliced data and determine where there is a dominant or very apparent frequency. We can do this by examining where each of the low frequencies or high frequencies is a very large number. First, we would calculate the value of the Fourier coefficient at each frequency. After calculating this, we would find the largest low-frequency coefficient. and the largest high-frequency coefficient along with their respective frequencies. After doing this, we use these two known frequencies, which one is a low frequency and one a high frequency to decode it with the `decode_freqs()` function to find the number that was pressed. Here is the modified for-loop:

```
for signal in sliced_data:

    highFreq = high_frequencies[0]
    lowFreq = low_frequencies[0]
    highValues = []
    lowValues = []

    for i in range(len(high_frequencies)):
        highValues.append(calculate_coefficient(signal, high_frequencies[i]))
#finds the value of the coefficient at each high frequency
        highFreq = high_frequencies[highValues.index(max(highValues))] #finds
the largest/most "apparent" high frequency

    for j in range(len(low_frequencies)):
        lowValues.append(calculate_coefficient(signal, low_frequencies[j]))
#finds the value of the coefficient at each low frequency
        lowFreq = low_frequencies[lowValues.index(max(lowValues))] #finds the
largest/most "apparent" low frequency


    print(decode_freqs(lowFreq, highFreq),end=" ")
```

What we are doing is similar to a Fourier transform or analytical method as a Fourier transformation uses a Dirac delta function to do the transformation.

Extension: Handling more complicated information: Consider expanding the encoding and decoding program to handle alphabetical messages.

The easiest way to expand the encoding and decoding program for alphabet messages is just to create a larger table/matrix. For example, we can expand the table below to include a, b, c, and so on:

	1209 Hz	1336 Hz	1477 Hz	…
697 Hz	1	2	3	…
770 Hz	4	5	6	…
852 Hz	7	8	9	…
941 Hz		0		…
…	…	…	…	…

That is one approach to expanding the program to alphabetic characters. If we just want to transpire only alphabetic characters, we can do this by replicating the dial pad of a phone:

	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2 (ABC)	3 (DEF)
770 Hz	4 (GHI)	5 (JKL)	6 (MNO)
852 Hz	7 (PGRS)	8 (TUV)	9 (WXYX)
941 Hz		0	

When a person wants to input a word or string of characters, they can just input the number of presses/taps for a particular number. We can differentiate between each character by adding a space button that indicates the beginning or ending of a string.

For example, someone can do:

44 (SPACE) 33 (SPACE) 555 (SPACE) 555 (SPACE) 666

To spell out the word “HELLO” using DTMF signals. To incorporate the SPACE instigator, we can use the empty slot of (941 Hz, 1209 Hz) or (941 Hz, 1477 Hz). Alternatively, we can also use 1’s frequency combination to incorporate this spacing. After doing this, the decoding program can be expanded to translate these number combinations into real words for us to read.