DTMF Signaling

Ali Mortada, Xavier Valencia, James Phommachanh

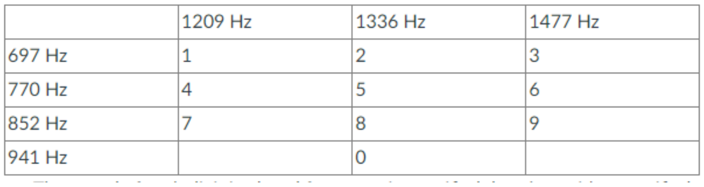
Mt. San Antonio College, ENGR 285, CRN 43464
June 14, 2024

1. Objective 1: Encoding Program

The short and sweet explanation as to what the encoding program does is that it makes a `.wav` file that contains a series of tones that are made from a list along with a series of pauses so that the original list of numbers fed into the program cannot be read without the proper decoding program.

From the top of the code, we import a few libraries to perform some math functions, then convert those outputs into `.wav` files. We make a file name for the `.wav` output, then a list of numbers that are between 0-9 to input into the program. The `sampleRate` variable is the step size of the wave file, so a higher sample rate would allow for a more accurate replication of that particular frequency. There is a limit to the frequencies a human can hear, so we will not change this value. The variable `soundLevel` represents the amplitude of the frequencies produced, so as the name of the variable indicates, a higher value will result in a higher sound level, with the contra-positive being true as well. The variable `pauseLength` is there to help encode the frequencies so that you cannot properly playback the `.wav` file.

We then create a function that will take in a frequency and create an array that loops for as long as the iterating variable is less than the sample rate value with the iterator incrementing at the end of each loop. The array is filled with a series of points that are representative of a sine wave of the input frequency. We do this for a series of frequencies and store them in variables, then the different frequencies are summed into a new list so that they correspond to the table given in the order of 0-9.



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

Figure 1. Keypad tones provided.

This list is essentially creating new tones by using constructive wave interference and associating a number to this new tone. This will be called `toneList` as provided. Now that we have a way to create a unique tone for each input and represent it in a list, we can now reference our `toneList` with the inputs of our `numberList`. We will make a new array called `soundData` that adds the tone list data along with pauses between each tone. This will be done in a for loop for the range of the `numberList`. This loops for however long the number list is, but it is noteworthy that we should increase the value of `soundLength` if we increase the length of the `numberList`.

Once that final for loop is done, we have all of our data to copy to the `.wav` file, the IDE opens the `.wav` file created prior and sets some parameters. These parameters include `nchannels`, which is the number of audio channels. I don't know how I feel about Dolby Atomos keypad tones. The variable `sampwidth` is the number of bytes used when writing data to the `.wav` file, we have 2 bytes so it would be 16 bits. The variable `framerate` is `sampleRate` but converted to an integer. The variable `nframes` represents the amount of steps that the `.wav` file will have, or how long it will be. The line `comptype = "NONE"` tells the program not to compress the output file, and `compname` is a description of the compression type. We set the parameters using a wave library function, then we use a for loop to write all of the data we have using a different wave library function. Finally, it closes the file and prints a message to the terminal indicating that the program has successfully written to the output file.

2. Objective 2: Slicing Data

The `slice_data()` function first divides the input data into equally sized parts, which depends on the number of inputs – this is stored in a variable called `slice_size`. Note that if the number of inputs changes, we must change the code so that it divides the input number into the correct number of equal parts. Figure 2 shows how the data is divided into different slices.

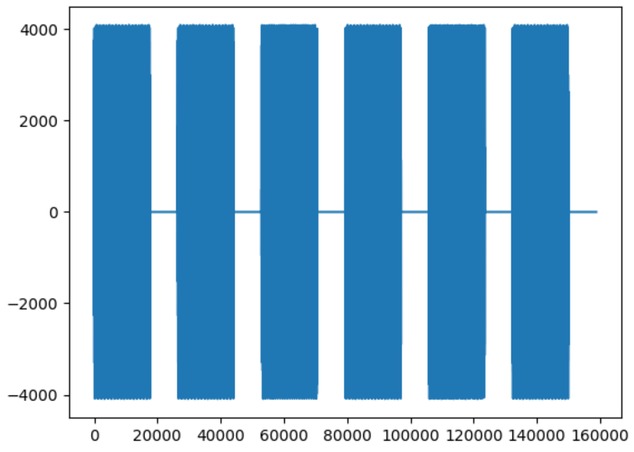
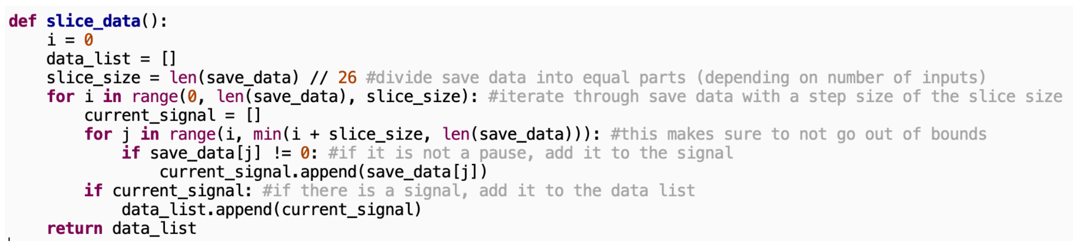


Figure 2. Slice graph with input [1, 1, 2, 1, 0, 1]. The data is divided into six equal parts, regardless of what the input values actually are.

The function then contains a for-each loop that iterates through the length of `save_data` with a step size of `slice_size`. This loop contains another nested loop that checks each data point within the slice. If the signal is not a pause, it is added to the data list that is returned by the function, but if it is, it is skipped over.

3. Objective 3: Calculating Approximate Fourier Coefficients

Since decoding the DTMF signals requires Fourier analysis, the Fourier coefficients can be approximated using the different decoded frequencies. Recall the Fourier coefficients



```python
def slice_data():
    i = 0
    data_list = []
    slice_size = len(save_data) // 26 #divide save data into equal parts (depending on number of inputs)
    for i in range(0, len(save_data), slice_size): #iterate through save data with a step size of the slice size
        current_signal = []
        for j in range(i, min(i + slice_size, len(save_data))): #this makes sure to not go out of bounds
            if save_data[j] != 0: #if it is not a pause, add it to the signal
                current_signal.append(save_data[j])
        if current_signal: #if there is a signal, add it to the data list
            data_list.append(current_signal)
    return data_list
```

Figure 3. Code snippet for the slice_data() function.

for the sine and cosine basis:

$$a_n = \frac{2}{T}\int_0^T f(t)cos(\frac{2\pi n}{T}t)\ \mathrm{d}t \tag{1}$$

$$b_n = \frac{2}{T}\int_0^T f(t)sin(\frac{2\pi n}{T}t)\ \mathrm{d}t \tag{2}$$

The coefficients in the program correlate to each of these Fourier bases. Thus, an approximation of the coefficients can be calculated by taking the summation of each frequency from the sliced data for the decoded sound signal. The period for each Fourier Basis (T), is the length of the decoded and sliced frequencies. Then for each data point (n), the frequencies (t) within the trigonometric part of the basis are divided by the frame rate – which in this case is the period (T). In the program, this simplifies to:

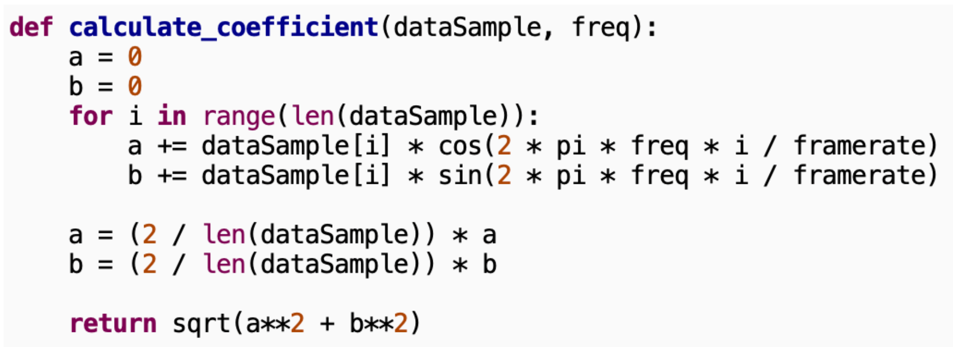
$$a_n = \frac{2}{T}\sum_0^T \text{frequency} * cos(\frac{2\pi * \text{datapoint}}{\text{framerate}} * \text{frequency}) \tag{3}$$

$$b_n = \frac{2}{T}\sum_0^T \text{frequency} * sin(\frac{2\pi * \text{datapoint}}{\text{framerate}} * \text{frequency}) \tag{4}$$

This results in the approximation of the Fourier coefficients for the DTMF signaling for each of the Fourier bases. The function returns the Euclidean norm of the Fourier basis in a plane representing each of the bases.

4. Objective 4: Outputting Decoded Digits

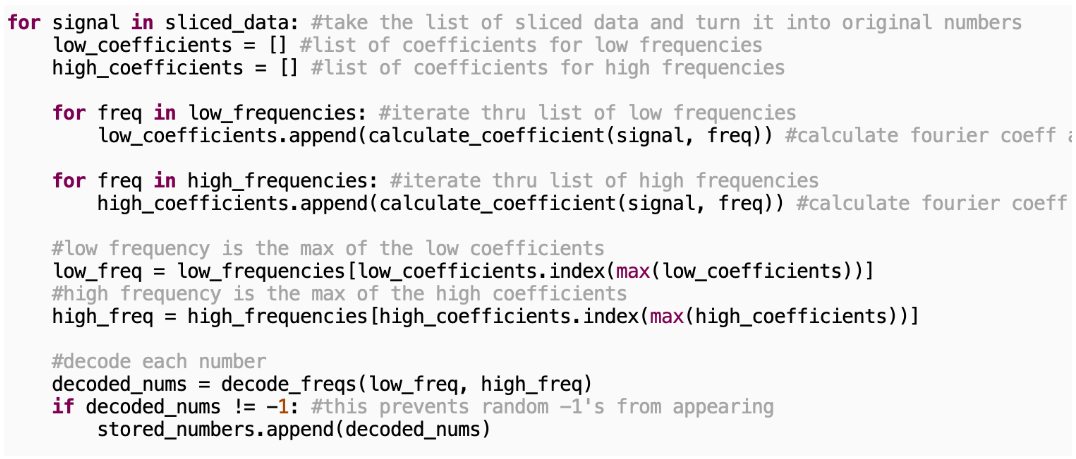
To output the decoded digits, a list of the Fourier coefficients of the decoded signal list can be iterated for each of the respective high and low frequencies of the program. Now for each signal from the data list, there is a list of low and high Fourier coefficients. The max of each of the coefficients represents the true occurrence of the high and low frequencies in the data signal. Thus, we can store the instances of these max coefficients for each signal to decipher the decoded digits. Once we have a list of where these instances occur, we find an instance in this list that correlates to the decoded digit originally in the decoded list of numbers. To check if an instance exists to the decoded



```python
def calculate_coefficient(dataSample, freq):
    a = 0
    b = 0
    for i in range(len(dataSample)):
        a += dataSample[i] * cos(2 * pi * freq * i / framerate)
        b += dataSample[i] * sin(2 * pi * freq * i / framerate)

    a = (2 / len(dataSample)) * a
    b = (2 / len(dataSample)) * b

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

Figure 4. Snippet of code for the calculate_coefficient() function.

list, we only stored digits that did not return -1 when put into the `decode_freqs()` function.

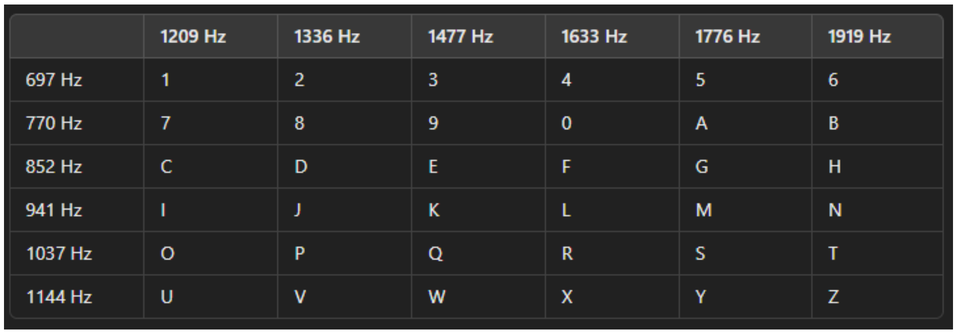


```python
for signal in sliced_data: #take the list of sliced data and turn it into original numbers
    low_coefficients = [] #list of coefficients for low frequencies
    high_coefficients = [] #list of coefficients for high frequencies

    for freq in low_frequencies: #iterate thru list of low frequencies
        low_coefficients.append(calculate_coefficient(signal, freq)) #calculate fourier coeff

    for freq in high_frequencies: #iterate thru list of high frequencies
        high_coefficients.append(calculate_coefficient(signal, freq)) #calculate fourier coeff

    #low frequency is the max of the low coefficients
    low_freq = low_frequencies[low_coefficients.index(max(low_coefficients))]
    #high frequency is the max of the high coefficients
    high_freq = high_frequencies[high_coefficients.index(max(high_coefficients))]

    #decode each number
    decoded_nums = decode_freqs(low_freq, high_freq)
    if decoded_nums != -1: #this prevents random -1's from appearing
        stored_numbers.append(decoded_nums)
```

Figure 5. The program's main for loop.

5. Extension: Handling Alphabetical Messages

For the extension, we decided to implement handling alphanumeric messages (apparently we don't know how to read instructions properly). To do this, we had to extend the list of low and high frequencies, since the original four low frequencies and three high frequencies cannot make thirty-six unique combinations. Thus, we added two new frequencies to the low frequencies list, 1037 Hz and 1144 Hz, and three new frequencies to the high frequencies list, 1633 Hz, 1776 Hz, and 1919 Hz. These frequencies were chosen arbitrarily by our good friend Chad, and Figure 6 shows the mapping of each alphanumeric symbol to its pair of low and high frequencies. In the `DTMFwrite` program, the arrays for each frequency were replaced by a frequency map matching each symbol to its pair of frequencies, and this map was used to populate `toneList`.

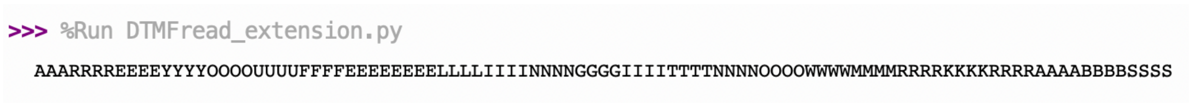


	1209 Hz	1336 Hz	1477 Hz	1633 Hz	1776 Hz	1919 Hz
697 Hz	1	2	3	4	5	6
770 Hz	7	8	9	0	A	B
852 Hz	C	D	E	F	G	H
941 Hz	I	J	K	L	M	N
1037 Hz	O	P	Q	R	S	T
1144 Hz	U	V	W	X	Y	Z

Figure 6. Extended frequency table matching each alphanumeric symbol to a pair of low and high frequencies.

The `slice_data()` function was also slightly modified to handle more symbols. The variable `slice_size` was redefined to be smaller to ensure a more accurate tone was recorded. New variables `overlap` and `sound_cutoff` were also defined to make sure the full tone was recorded and to filter out signals with no sound, respectively. The for loop now checked to see if the current signal was above the cutoff threshold before adding it to the data list – otherwise, it would skip over the overlap.

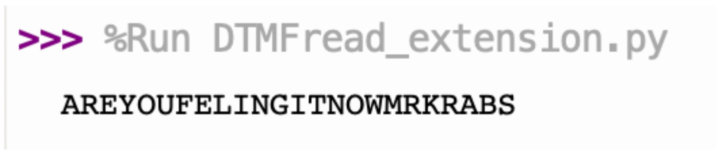
The code was now theoretically ready to handle alphanumeric messages. However, when it was tested, we found that the program would output multiple copies of the same symbol for each symbol in the message, as shown in Figure 7. To fix this, code was added to check if the current character is equal to the previously stored character – if it is not, then it is added to the list of stored characters. This seemed to fix the issue, but it added the limitation of being unable to read messages that have two or more of the same symbol consecutively, only outputting one copy of the letter. A sample output after the fix can be seen in Figure 8.



```
>>> %Run DTMFread_extension.py
  AAARRRREEEEYYYYOOOOUUUUFFFFEEEEEEEELLLLIIIINNNNGGGGIIIITTTTNNNNOOOOWWWWMMMMRRRRKKKKRRRRAAAABBBBSSSS
```

Figure 7. Initial output with test message ”AREYOUFEELINGITNOWMRKRABS”. Notice how each letter has 3-4 multiples in the output.

It is worthy to note that after implementing this extension, it takes noticeably longer for both the `DTMFread` and `DTMFwrite` programs to execute. This is likely because it has to map each symbol to its frequency pair instead of just reading from an array for each frequency. Both programs also contain multiple nested loops, so it is not unreasonable to assume that the time complexity of each program increases quadratically with an increasing number of symbols.



```
>>> %Run DTMFread_extension.py

  AREYOUFELINGITNOWMRKRABS
```

Figure 8. Initial output with test message ”AREYOUFEELINGITNOWMRKRABS” after fixing repeated symbol issue. Notice how the double E in ”FEELING” only has one E outputted.