DTMF Signaling

Ali Mortada, Xavier Valencia, James Phommachanh

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

1. Objective 1: Encoding Program

This is the section for the first objective.

What to talk about:

- What `createPureToneData` does
- What `toneList` is
- What the for loop does

2. Objective 2: Slicing Data

This is the section for the second objective.

What to talk about:

- Function has to divide the data into equally sized parts which depends on the number of inputs
- Iterates through the save data with a step size of the slice size
  - Uses another nested for loop that only adds signals that are not pauses (signal is not equal to 0)
  - If a signal is found, it is added to the data list that the function returns

3. Objective 3: Calculating Approximate Fourier Coefficients

This is the section for the third objective.

What to talk about:

- T is the length of the slice we are analyzing
- Integral is just a sum, so we can approximate it by multiplying each data point in the slice with its respective trig function
  - t is the data point divided by the frame rate
- Coefficient returned is the Euclidean norm of a and b

4. Objective 4: Outputting Decoded Digits

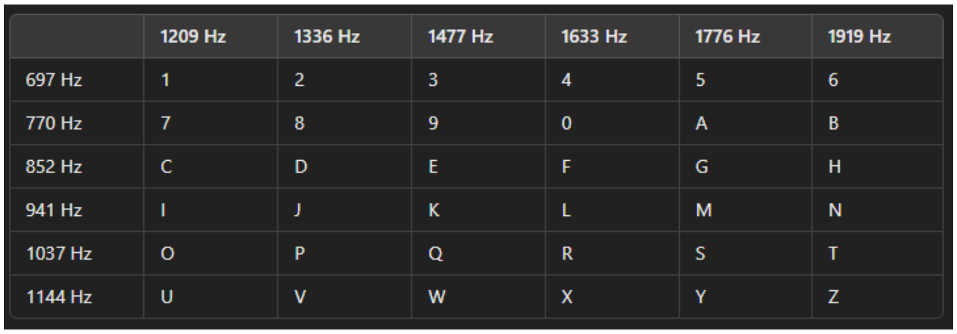
This is the section for the fourth objective.

What to talk about:

- Iterate through each data point in the sliced data
  - Create a list of low coefficients using the `calculate_coefficient` function and passing in the data point and each low frequency
  - Repeat for the high coefficients
- Find the low frequency from the index of the first appearance of the maximum low coefficients value
- Repeat for the high frequency
- Pass in the previously found low and high frequencies into `decode_freqs`, and if it is not -1, append it to the list of stored numbers

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 1 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`.



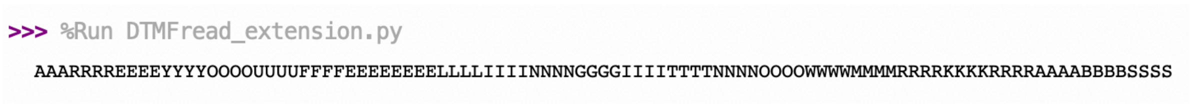
	1209 Hz	1336 Hz	1477 Hz	1633 Hz	1776 Hz	1919 Hz
697 Hz	1	2	3	4	5	6
770 Hz	7	8	9	0	A	B
852 Hz	C	D	E	F	G	H
941 Hz	I	J	K	L	M	N
1037 Hz	O	P	Q	R	S	T
1144 Hz	U	V	W	X	Y	Z

Figure 1. 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.

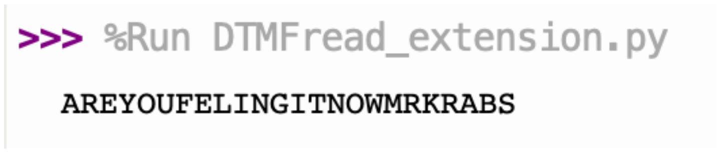
The code was now theoretically ready to handle alphanumeric messages. However, when it was tested, we found that the program would output multiple copies of the same symbol for each symbol in the message, as shown in Figure 2. To fix this, code was added to check if the current character is equal to 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 3.

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



```
>>> %Run DTMFread_extension.py
  AAARRRREEEEYYYYOOOOUUUUFFFFEEEEEEEEELLLLIIIINNNNGGGGIIIITTTTNNNNOOOOWWWWMMMMRRRRKKKKRRRRAAAABBBBSSSS
```

Figure 2. Initial output with test message ”AREYOUFEELINGITNOWMRKRABS”. Notice how each letter has 3-4 multiples in the output.



```
>>> %Run DTMFread_extension.py
  AREYOUFELINGITNOWMRKRABS
```

Figure 3. Initial output with test message ”AREYOUFEELINGITNOWMRKRABS” after fixing repeated symbol issue. Notice how the double E in ”FEELING” only has one E outputted.

to assume that the time complexity of each program increases quadratically with an increasing number of symbols.