Dual Tone Multi-Frequency (DTMF) Signaling

Vincent Edwards, Julia Corrales, and Rachel Gossard

June 10, 2025

Contents

1. DTMF Description

Dual-tone multi-frequency signals (DTMF) combine two pure tone sinusoids as a method of encoding or decoding digital information. This was first developed by the Bell System in 1963 and is commonly known as “Touch-Tone”, and it was used in push-button telephones. Each digit corresponds to a unique combination of one high frequency and one low frequency. Table 1 shows the low frequency in the first column and the high frequency in the first row associated with each digit from 0 to 9. For this project’s objectives, only numbers 0 to 9 were used.

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

Table 1: Digit Encoding Scheme

This matrix represents the most common arrangement of a DTMF telephone keypad, with the fourth row often reserved for symbols and 0. Pressing a key will send a superimposed combination of the low and high frequencies, where the sound of each dual-tone will play for a certain duration. A receiver will decode each dual-tone by performing Fourier analysis and determine the two most prominent frequencies.

`DTMFfrequencies.py` stores lists containing the high and low frequencies. This allows the frequencies to be accessed in the reading and writing programs without duplication. The `decode_matrix` stores the digit associated with each low and high frequency, similar to Table 1. -1 was put in place of unused frequency combinations. The `encode_dict` is a dictionary that maps each digit to its corresponding frequency pair. It was created by looping through the entries of the `decode_matrix`, searching for values between 0 and 9, then saving the position’s pair of frequencies in the dictionary using the current digit as the index.

DTMFfrequencies.py

```python
# Program to store the DTMF frequencies and the decoding matrix

available_characters = list(range(10))

low = [697, 770, 852, 941]
high = [1209, 1336, 1477]

decode_matrix = [
    [ 1, 2, 3],
    [ 4, 5, 6],
    [ 7, 8, 9],
    [-1, 0,-1],
]

encode_dict = {}
for i, row in enumerate(decode_matrix):
    low_freq = low[i]
```

```
    for j, digit in enumerate(row):
        high_freq = high[j]
        if digit not in available_characters:
            continue
        encode_dict[digit] = (low_freq, high_freq)
```

2. Encoding Program

DTMFwrite.py

```
# Program to encode a sequence of single digits into a DTMF sound (written to a .wav
file)

import DTMFfrequencies as freqs
import numpy as np
import wave # Necessary for writing the .wav file
import struct # Necessary for writing the .wav file

file_name = "media/TestSignals/TenDigits.wav" # Output file name (must include .wav)

number_list = [0,1,2,3,4,5,6,7,8,9] # List of digits (0-9) to be encoded into sound

sample_rate = 44100
sound_level = 4096
# Set the sound and pause lengths in milliseconds
sound_length = 400
pause_length = 200

# Use the sound/pause lengths and sample rate to calculate how many samples are need
for each
sound_samples = sample_rate * sound_length // 1000
pause_samples = sample_rate * pause_length // 1000

def create_pure_tone_data(freq):
    data = []
    amplitude = sound_level / 2
    omega = 2.0 * np.pi * freq
    for x in range(sound_samples):
        angle = omega * x / sample_rate
        value = amplitude * np.sin(angle)
        data.append(value)
    return np.array(data, dtype="int16")

pure_tone_data = {freq: create_pure_tone_data(freq) for freq in (freqs.low +
freqs.high)}

# Create a list that maps digits to their corresponding dual tone
tone_list = [[]] * 10
for digit in range(10):
    low_freq, high_freq = freqs.encode_dict[digit]
    tone_list[digit] = (pure_tone_data[low_freq] + pure_tone_data[high_freq]).tolist()
```

```
# Create a list with the tone and pause for each digit of the number list
sound_data = []
for digit in number_list:
    sound_data += tone_list[digit]
    sound_data += [0] * pause_samples

# Start to write the .wav file
wav_file = wave.open(file_name, "w")

# Parameters for the .wav file
nchannels = 1
sampwidth = 2
framerate = int(sample_rate)
nframes = (sound_samples + pause_samples) * len(number_list)
comptype = "NONE"
compname = "not compressed"

wav_file.setparams((nchannels, sampwidth, framerate, nframes,
    comptype, compname))

# Write the data to the file
for s in sound_data:
    wav_file.writeframes(struct.pack('h', int(s)))

wav_file.close() # Finish writing the .wav file

print("Writing " + file_name + " complete!")
```

Output:

```
Writing media/TestSignals/TenDigits.wav complete!
```

3. Decoding Program

DTMFread.py

```
# Program to read in and decode DTMF sound data from a .wav file

import DTMFfrequencies as freqs
import numpy as np
import matplotlib.pyplot as plt # Necessary if you want to plot the waveform
(commented out lines at the end)
import wave # Necessary for reading the .wav file
import struct # Necessary for reading the .wav file

# These first few blocks read in the .wav file to an ordinary integer data list
file_name = "media/TestSignals/TenDigits.wav"
plot_name = "media/TenDigitsPlot.svg"

wavefile = wave.open(file_name, 'r')
```

```
length = wavefile.getnframes()
framerate = wavefile.getframerate()
save_data = []
for i in range(0, length):
    wavedata = wavefile.readframes(1)
    data = struct.unpack("<h", wavedata)
    save_data.append(int(data[0]))
# At this point the sound data is saved in the save_data variable

# Slice up the save data into a list of each individual DTMF signal without pauses
def slice_data():
    i = 0
    data_list = []
    streak_length = 2
    while i < length:
        if not any(save_data[i:i+streak_length]):
            i += 1
        else:
            j = 0
            current_signal = []
            while any(save_data[i+j:i+j+streak_length]):
                current_signal.append(save_data[i+j])
                j += 1
            data_list.append(current_signal)
            i += j + 1
    return data_list

# Calculate the approximate Fourier coefficient of the input signal data for the given
frequency
def calculate_coefficient(data_sample, freq):
    a = 0
    b = 0
    N = len(data_sample)
    for i in range(N):
        y = data_sample[i]
        t = i / framerate
        a += y * np.cos(2 * np.pi * freq * t)
        b += y * np.sin(2 * np.pi * freq * t)
    return 2/N * np.sqrt(a**2 + b**2)

# Decode the given low and high frequencies to the corresponding digit
def decode_freqs(low_freq, high_freq):
    low_idx = freqs.low.index(low_freq)
    high_idx = freqs.high.index(high_freq)
    return freqs.decode_matrix[low_idx][high_idx]

sliced_data = slice_data()

# For each signal in the sliced data, find the dominant low and high frequencies
# Print the corresponding digit for each
```

```
for signal in sliced_data:
    low_coeffs = [calculate_coefficient(signal, freq) for freq in freqs.low]
    high_coeffs = [calculate_coefficient(signal, freq) for freq in freqs.high]
    low_freq = freqs.low[np.argmax(low_coeffs)]
    high_freq = freqs.high[np.argmax(high_coeffs)]

    print(decode_freqs(low_freq, high_freq), end="")
print()

# Plot the save data over time
fig, ax = plt.subplots()
ax.set(ylabel="$y$", xlabel="$t$ (s)")

time = np.arange(length) / framerate
ax.plot(time, save_data)

fig.savefig(plot_name)
```

Output:

```
0123456789
```

4. Handling More Complicated Messages

With the current encoding scheme, only digits (0-9) can be sent in messages. One might wish to expand the encoding and decoding programs to support messages containing more complicated information, such as letters and symbols. Let C refer to the number of different characters available.

To start, notice that Table 1 has two unused frequency combinations. Those unused combinations could be used to add two more characters to the encoding scheme. The desired characters would be put in the `decode_matrix`. But, there are still only $4 \times 3 = 12$ frequency pairs, and thus characters available. One technique to increase the number of characters available would be to increase the number of low frequencies and/or the number of high frequencies. These new frequencies would be added to the `low` and `high` frequency lists, and the size of the `decode_matrix` would be expanded to match the sizes of the frequency lists. If we use n_1 to refer to the number of low frequencies and n_2 to refer to the number of high frequencies, then number of characters available is

$$C = n_1 n_2 \tag{1}$$

If the goal is to be able to send digits and uncased letters, then $26 + 10 = 36$ characters are needed and it is required that $n_1 n_2 \geq 36$. One possible choice for n_1 and n_2 is $n_1 = n_2 = 6$. If the goal is to also include characters such as spaces, commas, and periods, then n_1 or n_2 can be increased to 7, giving 42 possible characters.

Another technique that could be used is mapping sequences of preliminary characters to different characters. Each frequency pair would correspond to a specific preliminary character as in the previous technique. Then, sequences of preliminary characters would be mapped to a final character. For example, a 4 followed by a 1 could map to `A`, a 4 followed by a 2 could map to `B`, and so on. The encoding program would need to be modified to have a function, dictionary, or matrix that maps message characters to sequences of preliminary characters. Those preliminary characters could then be encoding into the signal as before. The decoding program would function the same initially, turning signals into preliminary characters. Then, a function, dictionary, or matrix would be needed to map sequences of preliminary characters to message characters. Using sequences with a fixed length of l, the number of characters available is

$$C = (n_1 n_2)^l \tag{2}$$

If $n_1 = 4$ and $n_2 = 3$ as in the default scheme and $l = 2$, then that gives $(4 \times 3)^2 = 144$ possible characters. That is more than enough to cover any of the 128 ASCII characters. This technique has the advantage of not requiring any new frequencies, thought it has the disadvantage of making messages l times longer.

A third technique that could be used is introducing additional sets of frequencies, perhaps higher or lower than the current sets. If k frequency sets are used in total, then each character in the message would consist of k pure sinusoids. In order for the combined signal to have a maximum amplitude equal to `sound_level`, the individual sinusoids would need their amplitude set equal to the `sound_level / k`. Next, each of these new frequency sets would need to have their list of frequencies stored in a variable. To make all these frequency lists easier to manage, perhaps they could be put in a list. That way, the program could loop over the frequencies in each frequency list. Lastly, the `decode_matrix` would then need to be expanded to have `k` dimensions with lengths corresponding to the number of frequencies in each frequency list. The number of characters available when using k frequency sets each with length n_i is

$$C = \prod_{i=1}^{k} n_i \tag{3}$$

If $k = 3$, $n_1 = 4$, $n_2 = 3$, and $n_3 = 3$, then that gives $4 \times 3 \times 3 = 36$ characters, enough to send digits and uncased letters. Similar to the first technique, this technique does not increase the length of the message, though it does require introducing new frequencies. However, not as many distinct frequencies need to be used as the first technique. While the first technique required $6 + 6 = 12$ frequencies to encode 36 characters, this technique requires $4 + 3 + 3 = 10$ frequencies to encode the same amount of characters.