Dual Tone Multi-Frequency (DTMF) Signaling

Vincent Edwards, Julia Corrales, and Rachel Gossard

June 10, 2025

Contents

1. DTMF Description

DTMFfrequencies.py

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

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

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

encode_list = [(-1, -1)] * 10
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 range(10):
            continue
        encode_list[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_list[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. Extension