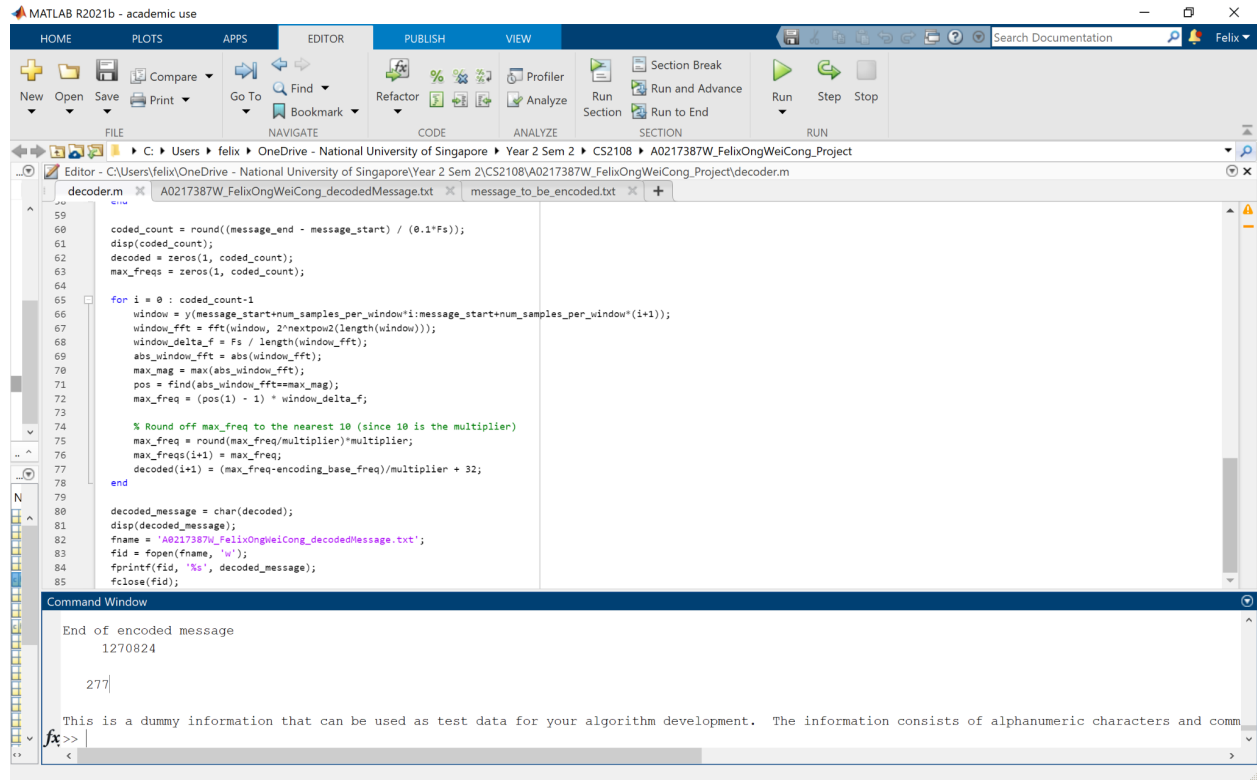


CS2108 Project Report

The report below details the process used to encode a given message into the given source music and to decode the music back into a message by sampling it from the air. It is assumed that the length of the message to encode will not be more than 277 characters as provided in the dummy text.

Should there be a need to encode more than 277 characters, the encoding and decoding process still works but may require a change in the durations and frequencies used.

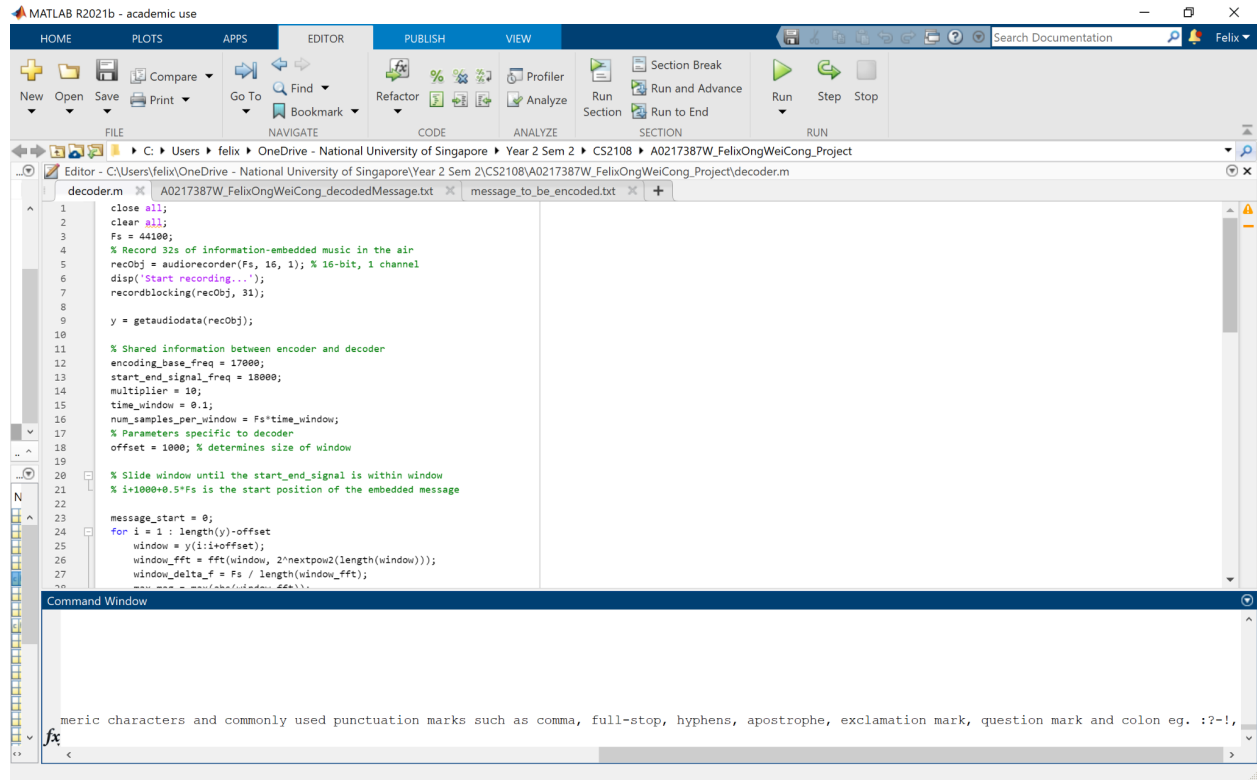


The image shows the MATLAB R2021b interface. The editor window displays a script named 'decoder.m' with the following code:

```
59 coded_count = round((message_end - message_start) / (0.1*Fs));
60 disp(coded_count);
61 decoded = zeros(1, coded_count);
62 max_freqs = zeros(1, coded_count);
63
64 for i = 0 : coded_count-1
65     window = y(message_start+num_samples_per_window*i:message_start+num_samples_per_window*(i+1));
66     window_fft = fft(window, 2*nextpow2(length(window)));
67     window_delta_f = Fs / length(window_fft);
68     abs_window_fft = abs(window_fft);
69     max_mag = max(abs_window_fft);
70     pos = find(abs_window_fft==max_mag);
71     max_freq = (pos(1) - 1) * window_delta_f;
72
73     % Round off max_freq to the nearest 10 (since 10 is the multiplier)
74     max_freq = round(max_freq/multiplier)*multiplier;
75     max_freqs(i+1) = max_freq;
76     decoded(i+1) = (max_freq-encoding_base_freq)/multiplier + 32;
77 end
78
79 decoded_message = char(decoded);
80 disp(decoded_message);
81 fname = 'A0217387W_FelixOngWeiCong_decodedMessage.txt';
82 fid = fopen(fname, 'w');
83 fprintf(fid, '%s', decoded_message);
84 fclose(fid);
```

The Command Window shows the output of the script:

```
End of encoded message
1270824
277
This is a dummy information that can be used as test data for your algorithm development. The information consists of alphanumeric characters and comm
```



Encoding

Firstly, I read the source music file to obtain the sampled data and the sampling frequency. Since the sampling frequency is 44100 Hz, this means that the maximum frequency that can be present in the data is $44100 \text{ Hz} \times 0.5 = 22050 \text{ Hz}$ according to the Nyquist rate, to prevent aliasing.

However, testing of my decoding device's mics showed that they are only sensitive to frequencies of up to 18000 Hz. Taking this hardware constraint into account as well as the fact that the human audible range is up to 20000 Hz, with decreasing sensitivity as the frequency increases, I decided to set 18000 Hz as the upper limit for the frequencies used to embed the message into the music. Testing of my output device also showed that the amplitudes were very low past 17000 Hz. As a result, each of the sine waves used to encode a character had to be multiplied by 2.5 to increase the amplitude. Multiplication by a factor too high distorted the frequency produced and 2.5 was determined to be the optimum factor.

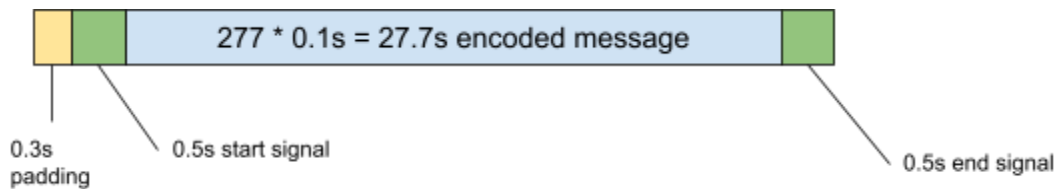
As decoding is done by sampling the music in the air using a single channel, only the left channel of the source music data was used to embed the encoded message.

The encoding process is as follows:

1. The message to be encoded is read from the file and converted from a character array into an array of ASCII values.

2. Each character will be represented by a sine wave of a specific frequency (henceforth referred to as '*encoded sine wave*') obtained using ' $encoding_base_frequency + coded_value * multiplier_value$ ', where each value of 1 of the *coded_value* corresponds to 1 Hz, and the *encoding_base_frequency* is the lowest frequency of an *encoded sine wave*.
 - a. Since ASCII values 0 to 31 are not alphanumeric characters or commonly used punctuation marks, they will not appear in the message to be decoded. This means that the range of values used to encode the characters can be reduced from 128 values(0 to 127) to 96 values(taken to be 0 to 95), reducing the frequency range required for encoding and hence increasing the lowest frequency of the *encoded sine wave*, making them less likely to be heard by the human ear.
 - b. My experiments showed that frequencies from 17000 Hz to 18000 Hz tend to deviate by around ± 5 Hz when analyzed using Fast Fourier Transform(FFT). Therefore, the *multiplier_value* is set as 10 allowing 10 Hz of difference between the frequencies of the *encoded sine waves* of characters differing by 1, preventing overlap due to deviations.
 - c. With a *multiplier_value* of 10 and the range of *coded_value* being 0 to 95, the range of frequency required to encode the differences between the characters is 950 Hz. With a maximum limit of 18000 Hz, the *encoding_base_frequency* is set at 17000 Hz.
 - d. Therefore, the range of frequencies of *encoded sine waves* used is 17000 Hz to 17950 Hz, which is barely audible to adult human ears.
3. A lowpass filter with a passband frequency of 15000 Hz is used on the left channel source music data. I set a 2000 Hz difference between the passband frequency and the *encoding_base_frequency* since the frequencies are not cut off sharply at 15000 Hz.
4. Since there is a need to synchronize the start of playing the message-embedded music and the start of the decoding process, a sine wave of 18000 Hz is used to signal the start and the end of the encoded segment of the music (*start_end_signal*). 18000 Hz was chosen because it is easy to identify and debug, and is 50 Hz outside the range of the encoding frequencies and hence will not interfere with the encoding.
5. Each *encoded sine wave* occupies a fixed time window of 0.1s and *encoded sine waves* are multiplied by 2.5 and then concatenated to form the encoded message signal. For 277 characters in the message, this will take 27.7s, leaving 1.3s since the source music has a duration of 29s. Each of the two *start_end signals* is given a time window of 0.5s, and the remaining 0.3s is added as padding to the front of the embedded message signal. These durations can be adjusted according to the requirements such as number of characters.

The diagram below shows how the encoded message signal is divided (not drawn to scale).



6. The encoded message signal which is now the same length is added to the lowpass-filtered source music signal to form the message-embedded music and written to the given file.

Decoding

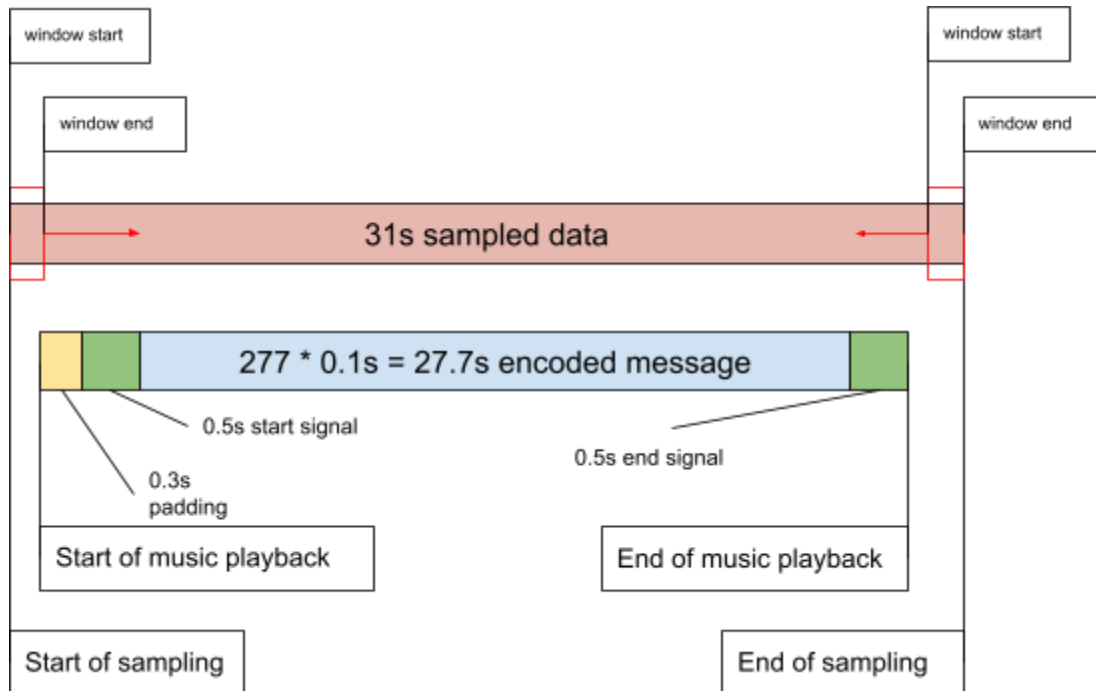
Firstly, the message-embedded music file is played and sampled from the air for a duration of 31s at a sampling frequency of 44100 Hz, 16-bit and single channel. 31s was used because although the music is 29s long, it is almost impossible to start the message-embedded music playback and the sampling at the same time, which may result in some parts of the music not being recorded. Therefore, I start the music playback slightly later after starting the sampling such that the entire music duration including the *start_end signals* can be recorded. This helps to resolve the synchronization issue.

Since it is assumed that the decoder is aware of the encoding process, the *encoding_base_frequency*, *start_end_signal_frequency*, *multiplier*, *time_window_duration* and *sampling_frequency* values, as well as the algorithm used to convert the frequencies back into ASCII values are shared with the decoder.

A parameter *offset* is used to control the size of the sliding window which is used to find the start and end positions of the encoded message in the message-embedded music data. As the sliding window moves from the start of the sampled data, *offset* represents the offset of the end of the window from the start of the sampled data initially.

As the sliding window moves from the end of the sampled data, *offset* represents the offset of the start of the window from the end of the sampled data initially. This is set at 1000 positions according to my experimentation taking into account efficiency and accuracy and can be changed depending on different conditions, such as duration of *start_end signal*.

The diagram below visualizes the decoding process. The red rectangles represent the sliding windows initially.



The decoding process after the sampling is complete is as follows:

1. Let the sampled data be y . Let the start and end positions of a sliding window be $window_start$ and $window_end$ respectively.
2. The sliding window covers $y(1:1001)$ initially, which should not contain the $start_end_signal$ if the playback of the message-embedded music and the decoding program is started simultaneously.
3. FFT is done on the window and the $position(s)$ of the maximum magnitude of the FFT result is found. The $maximum_frequency$ present can then be obtained by taking ' $(position(1) - 1) * frequency_resolution$ ' whereby $frequency_resolution$ is $sampling_frequency / length(FFT_result)$.
4. If the absolute value of ' $maximum_frequency - start_end_signal_frequency$ ' is within 5 Hz which is the range of deviation, the $start_end_signal$ is present in the window and the start position of the encoded message, $message_start$, is set to ' $window_start + offset + 0.5 * sampling_frequency$ ' since the $start_end_signal$ has a duration of 0.5s.
5. Else, the $start_end_signal$ is not present in the window and the sliding window moves by 1 position to the right (start and end positions increased by 1) and the process repeats from step 3.
6. To obtain the end position of the encoded message, $message_end$, a similar approach is taken, but with the sliding window covering $y(length(y)-1000:length(y))$ initially.
7. Same as step 3.

8. If the absolute value of ' $maximum_frequency - start_end_signal_frequency$ ' is within 5 Hz which is the range of deviation, the $start_end_signal$ is present in the window and $message_end$ is set to ' $window_start - 0.5 * sampling_frequency$ '.
9. Else, the $start_end_signal$ is not present in the window and the sliding window moves by 1 position to the left(start and end positions decreased by 1) and the process repeats from step 7.
10. The number of encoded character segments in the message is obtained by taking ' $round((message_end - message_start) / (time_window_duration * sampling_frequency))$ '.
11. To obtain the character within each time window of 0.1s, FFT is done for each segment from $message_start$ to $message_end$ to find $maximum_frequency$. Since the $maximum_frequency$ in each segment should be equal to ' $encoding_base_frequency + coded_value * multiplier_value$ ' where $multiplier_value$ is 10 as defined by the encoder, $maximum_frequency$ is rounded off to the nearest 10 by taking ' $round(maximum_frequency / 10) * 10$ '.
12. The ASCII value of the character in each segment is therefore ' $(maximum_frequency - encoding_base_frequency) / multiplier + 32$ '.
13. The array of ASCII values is converted back into a character array and the decoded message is displayed and written to the given file.

While the decoding process may seem to be computationally intensive due to having to run FFT many times, the process is actually rather fast due to the use of powers of 2 for the number of points used for FFT and selecting an appropriate window size. Clicking sounds can be heard in the playback of the message-embedded music, although I attempted to set the time window for each *encoded sine wave* to the fundamental period of all the *encoded sine waves*. Increasing their amplitudes by multiplication of 2.5 also made the clicking more audible but since this ensured the accuracy of decoding by making it easier to be picked up by my device's mics, I decided not to resolve the issue of clicking sounds.

References and Acknowledgements

While this project was done individually, some ideas were introduced by referencing discussions in the CS2108 LumiNUS forum, such as the idea of having a signal to indicate the start of the encoded message to solve synchronization issues.