ECE310 – Project 4

Jonathan Lam

December 20, 2020

1 Linear chirp

The linear chirp is defined by the equation:

$$x(t) = \cos(2\pi\mu t^2)$$

Generate a linear chirp with $\mu = 4.0 \times 10^9$, for a 200μs with $f_s = 5$MHz:

```
mu = 4e9;
duration = 200e-6;
fs = 5e6;

t = 0:(1/fs):duration;
x = cos(2 * pi * mu * t.^2);
```

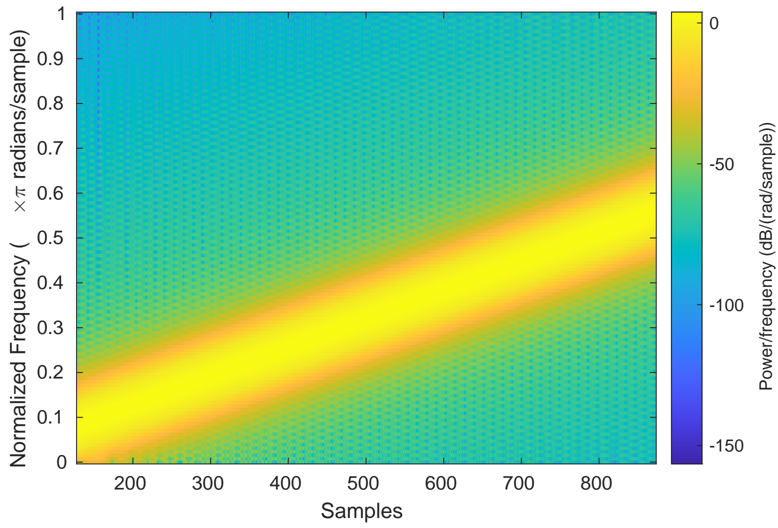
Generate a spectrogram for this signal using 256-point FFT's, a 256-point triangular window, and an overlap of 255 samples between sections.

```
N_fft = 256;
N_overlap = 255;
spectrogram(x, triang(N_fft), N_overlap, 'yaxis');
```

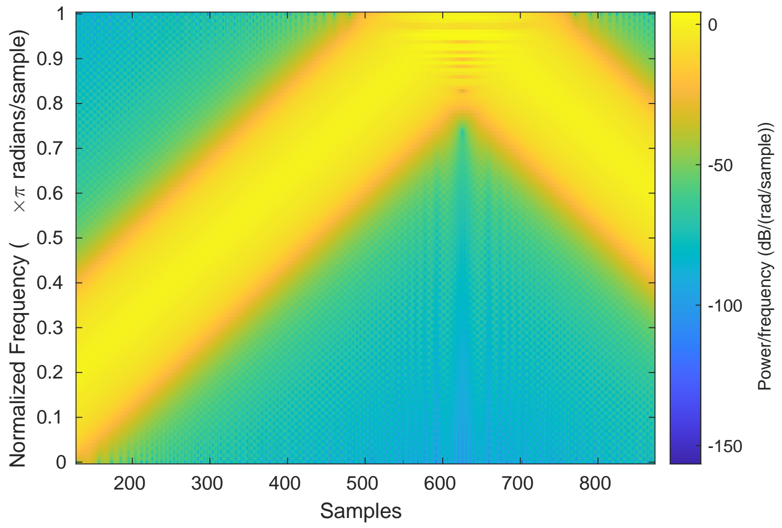
This leads to the spectrogram in Figure 1a. If we try to take the instantaneous frequencies, using the two definitions:

$$x(t) = \cos(2\pi f_1(t)t)$$
$$f_2(t) = \frac{1}{2\pi}\frac{d}{dt}\phi(t)$$

then, by the first definition f_1, the instantaneous frequency is μt; in the second definition the instantaneous frequency is $2\mu t$. These are plotted on top of the spectrogram in Figure 2. It is clear from this figure that the second definition (using the derivative) is correct; the first definition is consistent with the second only for constant frequency values, but the second is necessary for non-constant frequencies. We can compare this with the chirp signal when $\mu = 1.0 \times 10^{10}$ (slightly higher). This is shown in Figure 1b. The slope is steeper, and we see that the spectrogram line "bounces" back after reaching the top ($\pi = 1$). This is due to the Nyquist bandwidth and the fact that going x radians above the Nyquist bandwidth appears in the digital domain the same as going x below the Nyquist bandwidth.



(a) $\mu = 4.0 \times 10^{9}$



(b) $\mu = 1.0 \times 10^{10}$

Figure 1: Chirp spectrograms

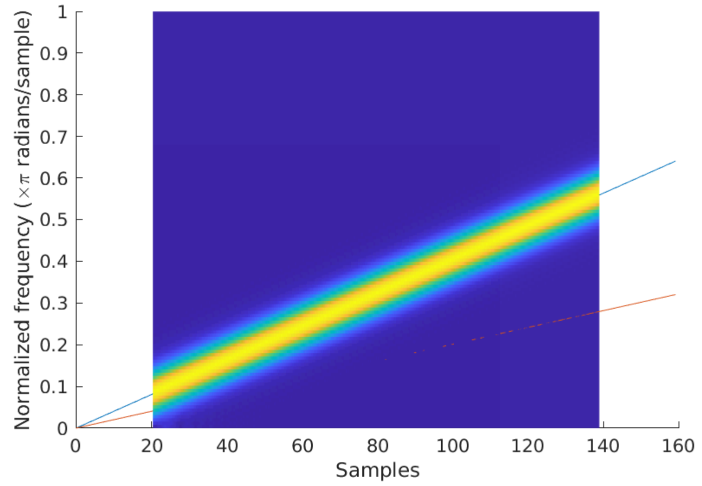


Figure 2: Chirp spectrogram slope

2 Narrowband and wideband spectrograms

We can produce narrow-band spectrograms of a speech signal to see the fundamental frequencies (broad yellow regions) and get good temporal resolution (to see when tones start and finish):

```
spectrogram(s1, triang(64), 63, 'yaxis');
spectrogram(s5, triang(128), 127, 'yaxis');
```

We can also produce wide-band spectrograms of the speech to see harmonics:

```
spectrogram(s1, triang(1024), 1023, 'yaxis');
spectrogram(s5, triang(1024), 1023, 'yaxis');
```

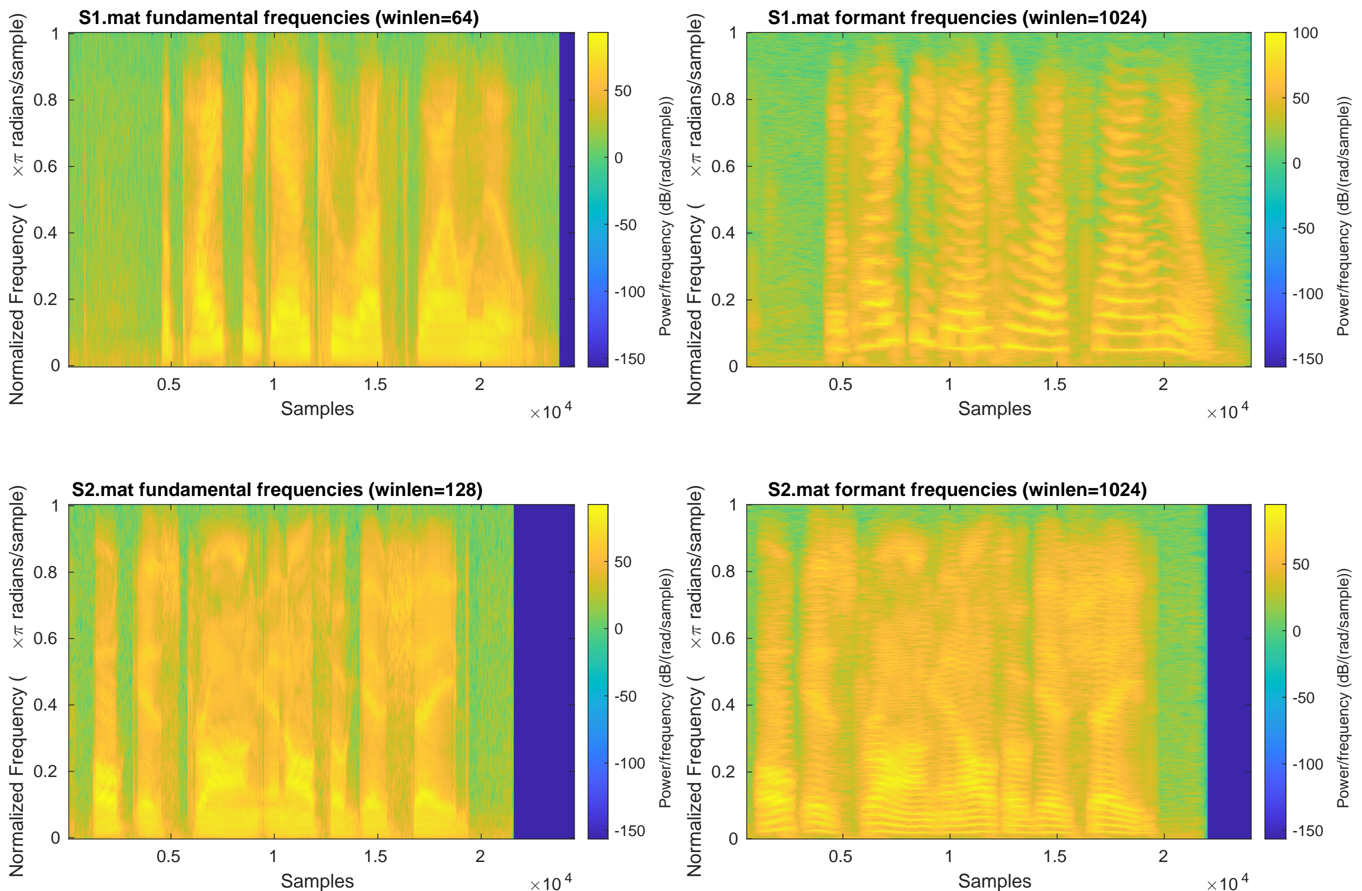


Figure 3: Speech analysis spectrograms

3 Modified STFTs

3.1 `stft2sig` function implementation

A potential pipeline for real-time (or otherwise) signal processing is to take the STFT, perform some processing on the STFT, and then reconstruct the signal from this modified STFT. Since adjacent samples of the STFT slices share a lot of redundant information between its samples (there is usually some overlap of signal data in adjacent STFT slices), modifying the STFT may ruin these redundancies and make it not a valid STFT anymore. However, we can do our best to reconstruct the signal from a modified (but potentially invalid) STFT by averaging the supposedly redundant parts (which would have no effect if they were indeed redundant, but has the effect of smoothing over irregularities if the STFT is invalid). My implementation is shown in Figure 4.

The assignment says to make some assumptions about the FFT length, window length, and sample overlap. I decided to implement it a little more generally, and all of these are parameters to the function. My function also infers the FFT length and number of samples from the input (it assumes the input spectrogram has the same STFT form as the output of the `spectrogram` function).

The function first augments the modified STFT with the negative frequencies (since the `spectrogram` function only returns the positive frequencies of the DFT), and then it takes the IFFT of each column (each spectrogram "slice").

To keep the function general, there are three parameters: `mod_stft`, `win_len`, and `overlap_len` for the modified STFT, window length, and overlap length, respectively. This takes the first `win_len` points from each slice's IFFT and places it at the correct position in the output signal, adding it to the current values there. However, since this may result in (one or more) overlaps, I keep track of how many times a particular point in the output signal has been overlaid. The returned signal is that output signal divided by the array keeping track of how many overlaps there are at each point, thus averaging all overlapped signals. This should work for more general cases than the specific case given in the assignment.

3.2 Example usage

We can test the function as follows (as a basic form of doubling the speed of the signal `vowels` while preserving frequency content). Playing back the reconstructed signal does sound like double the speed of the original signal at the same pitch (same frequency content).

```
win_len = 256;
overlap_len = 128;
fft_len = 1024;

sgram = spectrogram(vowels, rectwin(win_len), overlap_len, fft_len);
sgram_faster = sgram(:, 1:2:size(sgram, 2));
reconstructed_faster = stft2sig(sgram_faster, win_len, overlap_len);
```

```matlab
% Given a modified STFT, estimate the signal (perform an estimate
% of the inverse Fourier transform, even if the STFT is not valid)
%
% This function is implemented more generally than the problem set asks
% for.
%
% params:
% mod_stft      = modified STFT
% win_len       = length of window
% overlap_len   = length of window overlap
%
% returns:
% stft          = corrected stft
function stft = stft2sig(mod_stft, win_len, overlap_len)
    % infer FFT length and samples from STFT dimensions; note that
    % this FFT length is equal to half what it should be plus one
    % due to the nature of the spectrogram command
    [fft_len, samples] = size(mod_stft);
    fft_len = (fft_len-1) * 2;

    % non-overlap length
    nol = win_len - overlap_len;

    % augment STFT with negative frequencies
    mod_stft = [mod_stft(1:end-1, :); flip(mod_stft(2:end, :))];

    % take IFFT columnwise
    ifft_sig = real(ifft(mod_stft, fft_len, 1));

    % output array contains the results, as well as counting the number
    % of overlaps for the averaging process; probably a more efficient
    % way to do this but this is pretty general
    results = zeros(win_len + (samples-1) * nol, 2);

    % grab each sample, add it to the correct position, and increase
    % the overlap count for those samples
    for i = 1:samples
        current_range = ((i-1)*nol+1):((i-1)*nol+win_len);
        results(current_range, :) ...
            = [ifft_sig(1:win_len, i), ones(win_len, 1)] ...
            + results(current_range, :);
    end

    % return averaged samples (each sample is divided by its count)
    stft = results(:, 1) ./ results(:, 2);
end
```

Figure 4: Function to estimate a signal from a modified STFT