# SF1: Data Analysis - Statistical Signal Processing with Application in Audio

Week 2 Report

Joseph Johnson

May 2021

## 1 Introduction

Week 2 builds on the windowing experimentation from week 1. More complex audio analysis is explored using the short time fourier transform (stft) and several audio samples are processed using frequency domain techniques for noise reduction.

## 2 Answers to the tasks

### 2.1 Task 1

My interpretation of this filter is a bandpass filter between 1250Hz and 2500Hz (i.e. from $\frac{\pi}{4}$ to $\frac{\pi}{2}$ in the $\theta$ domain). It has sharp cutoffs at the boundaries, but is not an ideal filter because it passes all frequencies - only reducing the magnitude of the components outside of the desired region. It also passes the first frequency component of the dft without scaling.

A time domain version of this filter would have the same output as this frequency domain filter, despite the frequency filter having discontinuities at the cutoff frequencies. The frequency filter can be represented as a linear sum of rectangular functions, thus the equivalent time domain filter will be a sum of *sinc* waves which is, of course, non-causal. Therefore the realisable time domain filter would be a shifted and truncated version of the ideal filter. One might expect for this reason that the frequency domain filter would be a better filter, however the output of this discontinuous filter is infinitely long and so to represent it in a finite memory computer requires some truncation. To reiterate, the time and frequency domain filters would be identical assuming the same data length was used.

In an online scenario, the dft can be implemented using a sliding dft algorithm. This algorithm updates the frequency components each time a new data point arrives using only the data points that are entering and leaving the window, plus the value of the frequency component in the previous iteration. It has complexity $\mathcal{O}(N)$ where N is the window sized used in the stft. Therefore, with N set to 512 we require N additions (for real data) and N multiplications - thus the minimum expected delay in steady state would be $512t_{MAC}$, where $t_{MAC}$ is the time taken for one multiply and accumulate operation.

The line in part (d) utilises the conjugate symmetric nature of the dft of a real signal to rearrange the output frequency response in a more convenient order. The output of the matlab fft function provides frequency components in the order $\{0, .., \frac{N}{2}, -\frac{N}{2}+1, ..., -1\}$, but for plotting and inverse transform purposes it is more practical in the form $\{-\frac{N}{2}+1, .., -1, 0, ..., \frac{N}{2}\}$.

### 2.2 Task 2

My implementation of the whole noise reduction is shown in figure (1). It uses the `scipy.signal` library to perform the stft and inverse stft operations. The function `noise_reduction` takes the audio signal and an estimate of the noise power (see later sections) and returns a processed version of the audio according to the parameters. It also makes sure that the choice of window, window length and overlap are compatible.

### 2.3 Task 3

To test the filters, the provided audio sample *piano.wav* was corrupted with additive gaussian white noise and passed through the filter with the noise estimate equal to the power of the generated noise. I experimented with parameters to see how mean squared error (mse) scaled with different settings.

- Increasing window size caused the signal to 'blur' together once the window became long enough that the audio signal could no longer be assumed stationary in each window. Short windows were able to reduce mse by about 25%, but they only dampened the noise rather than removing it. Most online resources suggested a window length of between 5-50ms, so I aimed for this range in subsequent sections.

- Increasing the overlap generally improved mse. Short windows with larger overlap generally quietened noise more upon listening. Long windows with large overlap just sounded even more blurred.

- The Wiener and Power Subtraction methods generally had similar mse values, whilst the Spectral subtraction method was much higher. The three methods produced otherwise indiscernible results for the listening tests.
- Generally the hamming and hanning windows were equally good choices and were better than all of the other, more esoteric, windows I tried. The boxcar window was by far the worst as it introduced a high pitched whine after processing.

I found that mse was typically a poor indicator of filter performance. It heavily favoured shorter window lengths. For the piano sample ($f_s = 16k\text{Hz}$), a window length of 8 did successfully reduce mse and upon listening the noise was slightly weaker but definitely still present. Therefore I decided to mostly disregard mse as a performance metric and discern filter quality through listening in later sections.

Filters that successfully removed the noise tended to leave the desired audio sounding washed out and much less clear. If the noise estimate was significantly too high then the audio sounded much less natural, as if it had been poorly generated on a computer. Drastic overestimates could remove the signal entirely.

## 2.4 Task 4

### 2.4.1 Basic noise estimation

Task 4 involved trying my noise reduction function on some sample audio with different levels and types of noise. I focussed on the male speech sample and developed some methods for removing the different instances of noise. All of my subsequent methods used the Wiener filter to calculate gains. I decided to start with the soft noise case, as this was the most simple. I estimated the noise as the mean square of value of the first 1.7s of the audio (i.e. no speech), and checked that the mean was roughly zero when compared to the signal amplitude to validate this. I found that this method overestimated the noise quite significantly and left the signal sounding very washed out, but it was audible, and the noise was gone.

### 2.4.2 More complex noise estimation

In order to improve this, I tried some different methods of estimating the noise. I had the most success estimating the noise through a first-order low-pass filter:

$$|S_N|_i^2 = 0.85|S_N|^2{}_{i-1} + 0.15|N|_i^2$$

where the final term is the squared value of the *ith* noise sample. The estimate from this method was ten-fold smaller than the previous estimator, and left the speech much more audible and clear.

### 2.4.3 Filtering the spectrum

One method of dealing with the distortion introduced by the non-linearity of the *max* operation in the Wiener filter was to low-pass the processed spectrum across frequency at each time-frame before transforming back. The idea behind this is to soften the sharp troughs in areas of noise surrounded by signal frequencies and smooth out any isolated noise components. I used a slow first-order filter (coefficient 0.9) for the speech sample and found that it brightened the audio a noticeable amount.

### 2.4.4 Coloured noise

Dealing with the audio corrupted by non-white noise first required determining the form of the noise. A plot of the spectrum is shown in figure (2). Figure (2) also shows least-squares fit functions of the form $S_N(\theta) = \frac{A}{|\theta|^\alpha}$ for $\alpha = 1, 2$. These least-squares fits were performed on a smoothed sample of the noise spectrum, assuming that the noise was stationary. The best fit was $\alpha = 1$, so we proceed on the assumption that the added noise was pink noise (pink noise can be generated by application of an appropriate filter to white noise). Removing this noise was simply a case of updating the Wiener gain function to deal with values of $\alpha$ other than 0, ($\alpha = 0$ is white noise). The results were pleasing as this method removed the noise better than the same method assuming the noise was white.

#### 2.4.5 Noise oversubtraction

For the noisiest audio samples, I tried oversubtraction of the noise level with a floor of 0.01 and oversubtraction factor 2. I was unable to provide a satisfactory improvement to the audio quality with this method. In general, the loud samples were very difficult to clean as they had very poor signal-to-noise ratios.

## 3 Conclusion

Week two's work explored frequency domain noise reduction filters. I achieved the best results for the most number of cases using the following process:

1 Plot noise spectrum to determine type of noise (except for non-stationary case)

2 Estimate noise power through a low-pass filter for white noise, or a least-squares fit for non-white noise

3 Find stft of data and apply Wiener gains for the appropriate noise type

4 Low-pass filter the spectrum across frequency

5 Take inverse stft to recover processed signal.

All samples provided can be re-generated in the `week2.ipynb` notebook included. Chapter 11 of 'Advanced Digital Signal Processing and Noise Reduction, Second Edition' by Saeed V. Vaseghi was a very useful resource for methods to try in Task 4: http://dsp-book.narod.ru/304.pdf

```
def wiener_gains(stft_data, noise_power):
    # length of dft and number of bins
    nfs = stft_data.shape[0]
    nbins = stft_data.shape[1]
    # estimate of Power Spectrum
    Sy = np.square(np.abs(stft_data))
    # initialise gain matrix with the same shape as the stft data
    gains = np.zeros(stft_data.shape)
    # iterate over bins
    for i in range(nbins):
        # dft of the ith bin
        current_frame = Sy[:,i]
        # apply wiener rule, including maxing with zero
        gains[:,i] = np.maximum((current_frame - noise_power) / (current_frame), np.zeros(
            nfs))
    return gains

def noise_reduction(audio, noise_estimate, fsamp, M, red_func, ov=None, window_t='hanning
    '):
    # check NOLA and COLA conditions
    if (signal.check_COLA(window_t, M, ov) & signal.check_NOLA(window_t, M, ov)):
        # take stft of the audio signal
        _, _, Zxx = signal.stft(audio, fs=fsamp, window=window_t, nperseg=M, noverlap=ov)
        # calculate gain matrix according to our choice of red_func
        gains = red_func(Zxx, noise_estimate)
        # take inverse stft of gain data product to extract processed data
        _, clean = signal.istft(Zxx*gains, fs=fsamp, window=window_t, nperseg=M, noverlap=
            ov)
        return clean
    else:
        raise Exception("Window␣violates␣at␣least␣one␣of␣the␣NOLA␣or␣COLA␣conditions")
```

Figure 1: Noise reduction function with example of Wiener gains

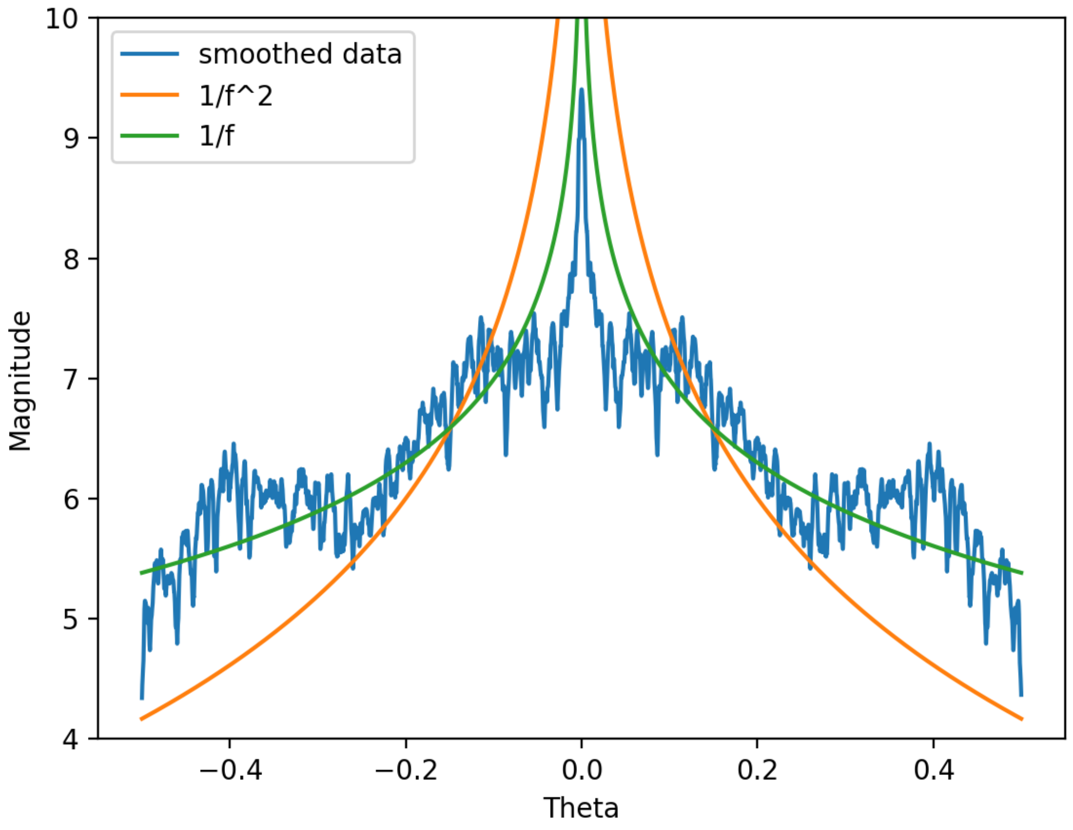

Figure 2: Least squares fits of different potential noise spectra based on the smoothed sample noise spectrum

```
def wiener_gains(stft_data, noise_power, f, exponen, alph):
    # length of dft and number of bins
    nfs = stft_data.shape[0]
    nbins = stft_data.shape[1]
    # estimate of Power Spectrum
    Sy = np.square(np.abs(stft_data))
    # initialise gain matrix with the same shape as the stft data
    gains = np.zeros(stft_data.shape)
    # iterate over bins
    for i in range(nbins):
        # dft of the ith bin
        current_frame = Sy[:,i]
        # apply wiener rule, including maxing with zero
        gains[:,i] = np.maximum((current_frame - alph*noise_power/np.power(f, exponen)) /
            (current_frame), np.zeros(nfs))
    return gains

def noise_reduction(audio, noise_estimate, fsamp, M, red_func, ov, window_t='hanning',
    expon=0, filter_spec=False, delt=0.5, alpha=1):
    # check NOLA and COLA conditions
    if (signal.check_COLA(window_t, M, ov) & signal.check_NOLA(window_t, M, ov)):
        # take stft of the audio signal
        freqs, _, Zxx = signal.stft(audio, fs=fsamp, window=window_t, nperseg=M, noverlap=
            ov)
        norm_freqs = 2*np.pi * freqs / audio.shape[0]
        # calculate gain matrix according to our choice of red_func
        gains = red_func(Zxx, noise_estimate, norm_freqs, expon, alpha)

        if filter_spec:
            gains = lpf(gains, delt)

        # take inverse stft of gain data product to extract processed data
        _, clean = signal.istft(Zxx*gains, fs=fsamp, window=window_t, nperseg=M, noverlap=
            ov)
        return clean
    else:
        raise Exception("Window␣violates␣at␣least␣one␣of␣the␣NOLA␣or␣COLA␣conditions")

def estimate_noise_est(noise_samp):
    # return mean square value
    return np.mean(np.square(noise_samp))

def estimate_noise_lpf(noise_samp, gam):
    # initialise average square noise value
    ave_n = 0
    gam_com = 1-gam
    # first order low pass filter
    for i in range(noise_samp.shape[0]):
        ave_n = gam * ave_n + gam_com * np.square(noise_samp[i])

    return ave_n
```

```
def lpf(stft_data, delta):
    # length of dft and number of bins
    nfs = stft_data.shape[0]
    nbins = stft_data.shape[1]
    # interested in the magnitudes of the dft only
    abs_data = np.abs(stft_data)
    # filter each time bin seperately
    for i in range(nfs):
        current_frame = abs_data[i,:]
        # filter the jth time bin accross frequency
        for j in range(1, nbins):
            current_frame[j] = delta*current_frame[j-1] + (1-delta)*current_frame[j]
        abs_data[i,:] = current_frame
    return abs_data

def fit_least_squares_1(noise_sample , w, range_low, range_high, M):
    # frequency axis
    axis = np.fft.fftshift(np.fft.fftfreq(noise_sample.shape[0]))
    # window the noise
    wind = signal.windows.hann(M)
    window_padded = np.pad(wind, (0, noise_sample.shape[0]-M))
    # find windowed noise specturm
    ft = np.fft.fftshift((np.abs(np.fft.fft(noise_sample*window_padded))))
    # smooth the spectrum
    Sn = np.convolve(ft, np.ones(w), 'same')/w
    # least squares solution on specified range
    A = np.sum(np.divide(Sn[range_low:range_high], np.abs(axis[range_low:range_high]))) /
         np.sum(np.divide(1, np.square(axis[range_low:range_high])))
    return A
```

Figure 3: Full suite of functions for noise reduction. Note that the `wiener_gains` and `noise_reduction` functions have been updated to account for the additional methods added in task 4.