

ECE300 – Project 1

Jonathan Lam

October 16, 2020

Simulating analog modulation schemes in MATLAB

Contents

1 Overview	4
1.1 Procedure	4
1.2 Execution environments and resource usage	4
2 Review of modulation schemes	6
2.1 Conventional (CONV) AM modulation	6
2.1.1 Overview	6
2.1.2 Analysis	6
2.1.3 Implementation details	7
2.2 SSB AM modulation	10
2.2.1 Overview	10
2.2.2 Analysis	10
2.2.3 Implementation details	10
2.3 PM Modulation	13
2.3.1 Overview	13
2.3.2 Analysis	14
2.3.3 Implementation details	14
2.4 FM Modulation	17
2.4.1 Overview	17
2.4.2 Analysis	18
2.4.3 Implementation details	18
2.5 Modulation schemes comparison	21
2.6 “Fairly” comparing different modulation schemes	21
3 Results	23
4 Discussion	41
4.1 The message signal	41
4.2 Verification of the modulation schemes	41
4.2.1 Conventional AM	41
4.2.2 SSB AM	41

4.2.3	PM	41
4.2.4	FM	42
4.3	Verifying Carson's rule for angle modulation	42
4.4	Effect of varying noise variance on SNR	42
4.5	Effect of varying modulation index on SNR	44
4.6	Effect of higher carrier and sampling frequencies	44
5	Conclusions	45
6	Acknowledgments	46
7	References	47
8	Appendix I: Audio	48
9	Appendix II: Auxiliary functions	49
9.1	Estimating signal power in MATLAB	49
9.2	Hilbert transform	49
9.3	Lowpass signal	49
9.4	Fourier transform plotter	50
9.5	Saving figures to PDFs	50
10	Appendix III: Calculating noise power in the digital domain	52
11	Appendix IV: Source code	53

List of Listings

1	Conventional AM modulation	7
2	Conventional AM demodulation	9
3	SSB AM modulation	11
4	SSB AM demodulation	12
5	PM modulation	15
6	PM demodulation	16
7	FM modulation	19
8	FM demodulation	20
9	Loading and preprocessing the audio signal	48
10	Hilbert transform function	49
11	Lowpass function	50
12	Fourier transform plotting function	50
13	Figure saver helper function	51

List of Figures

1	Original message signal	25
2	Conventional AM modulated and demodulated signals	26

3	SSB AM modulated and demodulated signals	27
4	PM modulated and demodulated signals	28
5	FM modulated and demodulated signals	29
6	Close-up comparison of conventional AM baseband and modulated spectra	30
7	Close-up comparison of SSB AM baseband and modulated spectra	31
8	Demonstration of Carson's rule	32
9	Effect of varying noise variance on conventional AM	33
10	Effect of varying noise variance on SSB AM	34
11	Effect of varying noise variance on PM	35
12	Effect of varying noise variance on FM	36
13	Effect of varying modulation index on conventional AM	37
14	Effect of varying modulation index on PM	38
15	Effect of varying modulation index on FM	39
16	Increased distortion with higher frequencies with FM ($f_c = 1\text{MHz}$, $f_{s,sc} = 10\text{MHz}$) .	40
17	Untitled. Commissioned by Anthony Belladonna.	46

List of Tables

1	Comparison of modulation schemes	21
2	Parameters for modulation schemes	23
3	Expected vs. theoretical power of modulated (transmitted) and demodulated signals	23
4	Percent power of signal in Carson's bandwidth for angle-modulated signals	23
5	Effect of varying noise variance on SNR	24
6	Effect of varying modulation index on SNR	24
7	Audio message signal statistics	48

1 Overview

Analog modulation schemes are at the core of all modern communication schemes. In this project, four common modulation schemes – conventional amplitude modulation, single side-band amplitude modulation, phase modulation, and frequency modulation – are implemented in MATLAB. In this report, we first review some of the modulation theory. We demonstrate that these signals faithfully recreate the original message after modulation and demodulation, and that the modulation is correct. Then, after adding additive white gaussian noise (AWGN) random processes, simulations are performed to analyze the effect of modulation scheme parameters and noise variance on the signal-to-noise ratio (SNR) on the different schemes. The relationship between SNR and noise power, and the relationships between SNR and the various parameters of the modulation schemes, generally agreed with the theory. The theoretical SNR calculations are compared with the experimentally-computed SNR values. SNR values comparable to theoretical values are obtained for all but FM. In addition to the requirements of this project, some additional checks were performed to verify the behavior of the modulated signals and also generally agreed with the theory.

1.1 Procedure

The assignment called for the following:

1. Read in an audio signal (and its sampling rate).
2. Write modulator/demodulator implementations for each modulation scheme. Make sure that the behavior seems correct. Verify Carson's rule.
3. Generate three zero-mean, different-variance AWGN noise processes. Add these to the modulated outputs of each system, demodulate them, and calculate the SNR of the demodulated signals experimentally. Compare these with the theoretical results.
4. Choose a noise process for each of conventional AM, FM, and PM which makes the demodulated output qualitatively noisy (but recognizable). Vary the modulation indices of each of these schemes by a factor of two in both directions, and calculate the SNR of the demodulated signals experimentally. Compare these with the theoretical results.

Reading in the audio file is described in 8. The implementations of the modulator and demodulator functions for each scheme are described in 2. Plots of the results appear in 3 and are discussed in 4.

1.2 Execution environments and resource usage

In order to emulate reasonable frequencies for the modulation schemes, I originally had the carrier and sampling frequencies set fairly high. The carrier frequency was set around 600kHz for amplitude modulation and around 1MHz for angle modulation, and the sampling rates were set to ten times those frequencies. Even though this AM carrier is on the lower end of the AM spectrum, and the FM carriers are far lower than real carriers, the lower end of which is at 88MHz [1], this was quite heavy on my computers. This caused many vectors with dozens of millions of double floating-point samples, which took dozens of seconds to run and took over 30GB of RAM when running on my Intel S5520UR server.

Another problem I encountered with these high frequencies is that there was often significant numerical error, especially with my FM signal. Interestingly, this happened very clearly when advancing the sampling rate to over 5MHz, or the carrier frequency to over 500kHz.

As a result of both the resource consumption issues and numerical errors associated with the multi-megahertz frequency ranges, I lowered the carrier and sampling frequencies for the angle modulation. While this becomes even less unrealistic for angle modulation, it did help alleviate both of these problems. A plot of the FM demodulated signal is Figure 16, and a plot of the FM demodulated signal with the lower carrier and sampling frequency is Figure 5.

2 Review of modulation schemes

Conventional AM, SSB AM, FM, and PM schemes are covered in this section. We also covered DSB AM and vestigial AM in class, but these are not covered here.

2.1 Conventional (CONV) AM modulation

2.1.1 Overview

Conventional AM (henceforth denoted CONV) is amplitude modulation with an offset, such that the modulation envelope is always positive and thus can be cheaply extracted with an envelope detector. Given a message signal $m(t)$, where $|m(t)| \leq 1$, modulation index $0 \leq a \leq 1$ (greater values of a are possible but cause overmodulation, or clipping), carrier frequency f_c , and carrier amplitude A_c , the modulation involves scaling and shifting m , and then mixing this with a carrier tone. The modulated signal $u(t)$ is:

$$u_{conv}(t) = A_c(1 + am(t)) \cos(2\pi f_c t) \quad (1)$$

Demodulation can be performed in two ways:

1. By mixing with the carrier signal (i.e., an I-Q demodulator with only an I component), which can be obtained by using a narrowband filter similar to in DSB-SC AM demodulation.
2. By rectifying and low-pass-filtering the signal (i.e., performing an envelope detection). Either full-wave or half-wave rectification can be used. This method is ubiquitous because it is cheap to implement in hardware.

Additionally, the demodulator has to:

- DC block (e.g., with a HPF) to remove the offset
- Divide the output by aA_c to get back to the original output amplitude (power)
- Scale the output up in order to compensate for energy lost during the LPF/HPF

2.1.2 Analysis

Let P_u be the total transmitted (bandpass) (modulated signal) power, P_c is the power in the transmitted signal sent in the carrier, P_y is the power in the transmitted signal for the message signal, and P_m is the power of the original normalized message signal.

$$P_u = P_c + P_m \quad (2)$$

$$P_c = \frac{A_c^2}{2} \quad (3)$$

$$P_y = \frac{a^2 A_c^2}{2} P_m \quad (4)$$

Since $P_m < 1$ (since $|m(t)| < 1$) and $|a| < 1$, at least half of the power is always sent in the carrier.

The SNR can be estimated by assuming an I-Q demodulation scheme. This doesn't account for rectification noise and is thus not perfect, but it is good enough. Let $r(t)$ be the noiseless received

message signal after mixing with the carrier tone (and ignoring the carrier power, which can be removed with a DC block). Let $y(t)$ denote the message part of the signal, and $n(t)$ denote the additive noise incurred during transmission.

$$r(t) = \frac{1}{2}A_c(1 + am(t))\cos(2\pi f_c t) + n_I(t)\cos(2\pi f_c t) + n_Q(t)\sin(2\pi f_c t) \quad (5)$$

$$\rightarrow \frac{1}{2}A_c(am(t)) + n_I(t) \quad (\text{after DC block and mixing with cosine})$$

$$P_y = \frac{a^2 A^2}{4} P_m \quad (6)$$

$$P_r = \frac{2N_0 W}{4} \quad (7)$$

$$\text{SNR} = \frac{a^2 A_c^2}{2N_0 W} P_m \quad (8)$$

The calculation for noise power was derived in the analog domain. See 10 for details about the digital domain calculation.

2.1.3 Implementation details

The modulation function is fairly straightforward. Notes on demodulation:

```

1 % modulates a signal using the conventional AM scheme
2 % params:
3 % f_c = frequency of carrier (Hz)
4 % A_c = carrier amplitude
5 % a = modulation index
6 % sig_m = message signal
7 % f_s_m = message sampling frequency (Hz)
8 % f_s_c = carrier sampling frequency (Hz)
9 % returns:
10 % sig_c = conventional AM-modulated signal
11 function sig_c = conv_mod(f_c, A_c, a, sig_m, f_s_m, f_s_c)
12     duration = length(sig_m) / f_s_m;
13     t_m = linspace(0, duration, length(sig_m));
14     t_c = linspace(0, duration, f_s_c * duration);
15
16     % upsample sig_m
17     sig_m_us = interp1(t_m, sig_m, t_c);
18
19     % generate conventional AM-modulated signal
20     sig_c = A_c * (1 + sig_m_us * a) .* cos(2 * pi * f_c * t_c);
21 end

```

Listing 1: Conventional AM modulation

- I perform the latter demodulation scheme because it is more common.
- Half-wave rectification was arbitrarily chosen for my implementation of the demodulation.
- The LPF and HPF are the frequency response of first-order systems.
- A HPF with an arbitrarily low cutoff frequency was used to help alleviate a low-frequency noise caused by rectification.
- An additional “DC block” is performed by subtracting the mean of the filtered signal, since the HPF doesn’t always perfectly center the signal.
- The constant 3.3 was empirically determined to recover the energy lost from the rectification and LPF. This constant scalar seemed to work pretty well for a variety of input samples. I wasn’t able to find a way to do this analytically, especially because rectification is a nonlinear function.
- The original signal amplitude was recovered by dividing by $A_c a$.

Note that the SNR equation for conventional AM doesn’t assume that the original amplitude is restored (i.e., by dividing by $A_c a$); thus, when computing SNR experimentally, we have to remultiply the scaled output by $A_c a$.

```

1 % demodulates a signal modulated using the conventional AM scheme
2 % params:
3 % sig_c = conventional-AM modulated signal
4 % A_c = carrier amplitude
5 % a = modulation index
6 % f_s_c = carrier sampling rate (Hz)
7 % tau = LPF cutoff (Hz)
8 % returns:
9 % sig_m = demodulated signal at lower frequency
10 function sig_m = conv_demod(sig_c, A_c, a, f_s_m, f_s_c, tau)
11     % half-wave rectification
12     sig_c(sig_c < 0) = 0;
13
14     % low-pass filtering in freq. domain
15     wd = linspace(-pi, pi, length(sig_c));
16     f_c = wd * f_s_c / (2 * pi);
17     lpf = (1 + f_c/tau*1j).^-1;           % LPF frequency response
18     hpf = (1 - 20*f_c.^-1*1j).^-1;       % HPF to get rid of offset
19                                         % and rectifier noise
20     ft_c = fftshift(fft(sig_c)) / f_s_c; % freq. domain rectified signal
21     ft_c = lpf .^ 2 .* hpf .* ft_c;      % apply filter in freq. domain
22
23     sig_c = ifft(ifftshift(ft_c) * f_s_c);
24
25     % downsample
26     duration = length(sig_c) / f_s_c;
27     t_c = linspace(0, duration, f_s_c * duration);
28     t_m = linspace(0, duration, f_s_m * duration);
29     sig_m = real(interp1(t_c, sig_c, t_m));
30
31     % DC block
32     sig_m = sig_m - mean(sig_m);
33
34     % scaling; the extra factor is empirically determined to account for
35     % the power in the carrier lost when filtering
36     sig_m = 3.3 * sig_m / A_c / a;
37 end

```

Listing 2: Conventional AM demodulation

2.2 SSB AM modulation

2.2.1 Overview

Single-sideband amplitude modulation aims to be more bandwidth efficient than the other AM schemes. It does this by employing the Hilbert transform to encode some of its information into a quadrature component. Given a message signal $m(t)$, the transmitted modulated signal looks like (some of the terms used in conventional AM will not be redefined here):

$$u_{(u|l)ssb}(t) = A_c (m(t) \cos(2\pi f_c t) \mp \hat{m}(t) \sin(2\pi f_c t)) + A_p \cos(2\pi f_c t) \quad (9)$$

Both modulation and demodulation occur using an I-Q (de)modulator. A pilot tone is necessary to ensure coherence, and a narrow-band filter and mixer are required in the demodulator to employ the pilot tone. These additions make SSB-AM also power-inefficient (as, like conventional AM, much of the power is in the carrier signal), and expensive (as it requires a mixer circuit and narrow-band filter.)

2.2.2 Analysis

Examining the power transmitted by this signal:

$$P_u = P_c + P_y \quad (10)$$

$$P_c = \frac{A_p^2}{2} \quad (11)$$

$$P_y = \frac{A_c^2}{2} P_m + \frac{A_c^2}{2} P_{\hat{m}} = A_c^2 P_m \quad (12)$$

Again, a large portion of the power is transmitted in the carrier signal. Twice as much power is transmitted in the message part of the signal as in conventional AM (assuming that $a = 1$), but there is also twice as much noise power, which contributes to a SNR that is equal to that of DSB SC (and roughly equal to that of conventional AM, assuming that $a = 1$). Let $r(t)$ again be the received signal,

$$r(t) = (A_c m(t) + n_I(t)) \cos(2\pi f_c t) - (\pm A_c \hat{m}(t) + n_Q(t)) \sin(2\pi f_c t) \quad (13)$$

$$\rightarrow \frac{A_c}{2} m(t) + \frac{1}{2} n_I(t) \quad (\text{after mixing with cosine})$$

$$P_y = \frac{A_c^2}{4} P_m \quad (14)$$

$$P_n = \frac{1}{4} N_0 W \quad (15)$$

$$\text{SNR} = \frac{A_c^2}{N_0 W} P_m \quad (16)$$

Again, the calculation for noise power was derived in the analog domain. See 10 for details about the digital domain calculation.

2.2.3 Implementation details

See 9.2 for the implementation of the Hilbert transform. The modulator follows the definition of SSB fairly closely, and easily allows for both USSB and LSSB. Notes on demodulation:

```

1 % modulates a signal using the SSB AM scheme; currently assumes phase
2 % coherence (no pilot tone in modulated signal)
3 % params:
4 % f_c = carrier frequency (Hz)
5 % A_c = carrier amplitude
6 % sig_m = message signal
7 % f_s_m = message sampling frequency (Hz)
8 % f_s_c = carrier sampling frequency (Hz)
9 % ussb = whether it is USSB (true for USSB, false for LSSB)
10 % returns:
11 % sig_c = SSB AM-modulated signal
12 function sig_c = ssb_mod(f_c, A_c, sig_m, f_s_m, f_s_c, is_ussb)
13     duration = length(sig_m) / f_s_m;
14     t_m = linspace(0, duration, length(sig_m));
15     t_c = linspace(0, duration, f_s_c * duration);
16
17     % upsample sig_m
18     sig_m_us = interp1(t_m, sig_m, t_c);
19
20     % generate in-phase component (same as DSB-SC AM-modulated signal)
21     sig_dsbsc = sig_m_us .* cos(2 * pi * f_c * t_c);
22
23     % generate quadrature component
24     sig_mhat = hilbert_transform(sig_m_us);
25     sig_quad = sig_mhat .* sin(2 * pi * f_c * t_c);
26
27     if is_ussb
28         sig_c = A_c * (sig_dsbsc - sig_quad);
29     else
30         sig_c = A_c * (sig_dsbsc + sig_quad);
31     end
32
33     % add pilot tone
34     A_p = 1;
35     sig_c = sig_c + A_p * cos(2 * pi * f_c * t_c);
36 end

```

Listing 3: SSB AM modulation

- A narrow-band filter would be hard to realistically construct in MATLAB, and I did not get very far with my approximations for one (using only first-order filters). Thus, we abstract away these hardware details, assume phase coherence, and use a pure generated cosine and sine with no phase offset for the I-Q demodulation.
- A first-order LPF and HPF were designed for this function.

- The LPF is largely intended to filter out the double-carrier-frequency signal generated from I-Q mixing, and the HPF reduces any low-frequency noise and the DC component caused by the I-Q mixing.

```

1 % demodulates a signal modulated using the SSB AM scheme; currently
2 % assumes phase coherence (no pilot tone in modulated signal)
3 % params:
4 % sig_c = SSB AM-modulated signal
5 % f_c = carrier frequency (Hz)
6 % A_c = carrier amplitude
7 % f_s_m = baseband sampling frequency
8 % f_s_c = carrier sampling frequency (Hz)
9 % tau = LPF cutoff frequency
10 % returns:
11 % sig_m = demodulated signal at baseband sampling frequency
12 function sig_m = ssb_demod(sig_c, f_c, A_c, f_s_m, f_s_c, tau)
13     duration = length(sig_c) / f_s_c;
14     t_c = linspace(0, duration, f_s_c * duration);
15     t_m = linspace(0, duration, f_s_m * duration);

16
17     % demodulate in-phase component
18     sig_c = sig_c .* cos(2 * pi * f_c * t_c);

19
20     % low-pass filtering in freq. domain; set cutoff frequency to carrier
21     % frequency
22     wd = linspace(-pi, pi, length(sig_c));
23     f_c = wd * f_s_c / (2 * pi);
24     lpf = (1 + f_c/tau*1j).^-1;           % LPF frequency response
25     hpf = (1 - 20*f_c.^-1*1j).^-1;       % HPF to remove low frequency
26                                         % distortion from pilot
27     ft_c = fftshift(fft(sig_c)) / f_s_c;   % freq. domain rectified signal
28     ft_c = lpf .* hpf .* ft_c;            % apply filter in freq. domain
29     sig_c = ifft(ifftshift(ft_c) * f_s_c);

30
31     % scale and downsample
32     sig_m = real(interp1(t_c, sig_c, t_m));

33
34     % scale back up to original amplitude
35     sig_m = 2 * sig_m / A_c;
36 end

```

Listing 4: SSB AM demodulation

2.3 PM Modulation

2.3.1 Overview

Phase modulation involves encoding the message into the phase $\phi(t)$ of the carrier signal:

$$\phi(t) = k_p m(t) \quad (17)$$

where k_p is an arbitrary scaling constant. We also define the modulation index, β_p :

$$\beta_p := k_p \max |m(t)| \quad (18)$$

This modulation index indicates the maximum phase deviation, $\Delta\phi_{max}$. If we adjust the message signal so that its maximum value has unity amplitude, it makes the maximum deviation simply k_p . The modulated signal looks like:

$$u_{pm}(t) = A_c \cos(2\pi f_c t + \phi(t)) \quad (19)$$

$$= A_c (\cos \phi \cos(2\pi f_c t) - \sin \phi \sin(2\pi f_c t)) \quad (20)$$

$$\approx A_c (\cos(2\pi f_c t) - \phi \sin(2\pi f_c t)) \quad (\phi \ll 1) \quad (21)$$

These three forms of the equation lend them to three modes of demodulation (and also multiple modes of modulation).

- If we differentiate the form in (19) (e.g., using an inductor), then we get:

$$u'_{pm}(t) = -A_c (2\pi f_c + \phi'(t)) \sin(2\pi f_c t + \phi(t)) \quad (22)$$

This has the same form as conventional AM, where the offset is $2\pi f_c$ (sufficiently large) and the message signal is $\phi'(t)$. By rectifying and LPF-ing, this can produce the envelope, $\phi'(t)$. This can then be integrated (e.g., by using a capacitor), then divided by k_p in order to recover the original message. This method would introduce some rectification noise, but is cheap to implement like conventional AM. This method also does not require sending an additional pilot tone.

- The form in (20) is an I-Q decomposition of the PM-modulated signal, which can be I-Q demodulated to reproduce $I = \cos \phi$ and $Q = \sin \phi$. ϕ can be recovered with $\arctan(Q/I)$, and then we can divide by k_p . I haven't tried implementing this, but I've heard from peers that the arctangent may sometimes introduce numeric instabilities using this method.
- (20) can be approximated as (21) with the assumption that $|\phi| \ll 1$. This is called the **narrow band approximation**. When demodulating, we only need recover the Q-component, and don't require the use of an arctangent function, so this is simpler than the second method.

For phase coherence, an additional pilot tone term $A_p \cos(2\pi f_c t)$ should be added. This is required for the latter two methods (I-Q demodulation), but is not required for the former (envelope detection).

PM-signal generation can be implemented as an I-Q demodulation (one of the two latter equation forms); the last form is easiest to implement, but only can be used when the modulation index is small. However, there are ways to use this method even with a wide band signal, namely by frequency division and multiplication.

Unlike amplitude-modulated signals, angle-modulated signals (PM and FM) take up infinite bandwidth because of their continuous frequency (phase) deviations. This can be seen mathematically as the Fourier transform of angle-modulated signals involves the Bessel functions, and thus have infinite bandwidth. However, Carson's rule states that the majority of an angle-modulated signal's power ($\geq 98\%$) lies in a bandwidth of:

$$W_{carson} = 2(\beta + 1)W \quad (23)$$

where $\beta = \beta_p$ for phase modulation and $\beta = \beta_f$ for frequency modulation.

2.3.2 Analysis

When assuming the narrow band approximation, we can estimate that both $A_c + A_p$ is the amplitude of the in-phase carrier, and $A_c\beta_p$ is the amplitude of the quadrature carrier. Thus:

$$P_u = \frac{(A_c + A_p)^2}{2} + \frac{A_c^2\beta_p^2 P_m}{2} \quad (24)$$

Since the modulated signal is narrow band (by assumption), then β_p is small and the first term dominates. I defer the derivation of the noise power and SNR to the class notes, or section 5.3 of [2].

$$P_m = k_p^2 P_m \quad (25)$$

$$P_n = \frac{2WN_0}{A_c^2} \quad (26)$$

$$\text{SNR} = \frac{k_p^2 A_c^2}{2} \frac{P_m}{N_0 W} = \frac{A_c^2}{2} \left(\frac{\beta_p}{\max |m(t)|} \right)^2 \frac{P_m}{N_0 W} \quad (27)$$

Again, the calculation for noise power was derived in the analog domain. See 10 for details about the digital domain calculation.

2.3.3 Implementation details

The modulation is fairly straightforward. We calculate the phase function, and add this phase pointwise to the carrier signal. (In practice, it would be easier to implement by mixing a narrow band signal with the carrier frequency using the narrow band approximation, but this is MATLAB.) A pilot tone is added to the signal.

Notes on demodulation:

- We use the narrow band approximation here in order to simplify the calculation. This means an I-Q demodulation (the third PM demodulation method described) by only mixing with the quadrature carrier.
- Similar to in SSB, we generated a pilot tone in the modulation “for show”: this would be necessary for coherence with the I-Q demodulation, but actually extracting this tone in MATLAB with a realistic filter would be hard. Thus, again we don't use this pilot tone in the demodulator, but we generate a quadrature carrier (sine wave) and assume coherence.

```

1 % modulates a signal using the PM scheme
2 % params:
3 % f_c = frequency of carrier
4 % A_c = amplitude of carrier
5 % sig_m = message signal
6 % k = k_p, phase deviation coefficient
7 % f_s_m = downsampled sampling rate
8 % f_s_c = upsampled sampling rate
9 % returns:
10 % sig_c = PM-modulated signal at upsampled rate
11 function sig_c = pm_mod(f_c, A_c, sig_m, k, f_s_m, f_s_c)
12     duration = length(sig_m) / f_s_m;
13     t_m = linspace(0, duration, length(sig_m));
14     t_c = linspace(0, duration, f_s_c * duration);
15
16     % upsample sig_m, multiply by k to get phase
17     sig_phi = k * interp1(t_m, sig_m, t_c);
18
19     % generate phase-modulated signal w/ pilot tone
20     A_p = 1;
21     sig_c = A_c * cos(2 * pi * f_c * t_c + sig_phi) ...
22             + A_p * cos(2 * pi * f_c * t_c);
23 end

```

Listing 5: PM modulation

- As with the previous modulation schemes, a first-order LPF and HPF were used to filter the components. The LPF was applied twice to get rid of some extra noise at higher frequencies.
- To regain the original amplitude, we multiply by a factor of 2. This can be explained by the power losses in the demodulation. Firstly, half of the power should be in the I component, which we completely disregard. Another half of the power is lost when mixing with the carrier. Losing power by a factor of 4 means an amplitude attenuation by a factor of 2.

```

1 % demodulates a signal modulated using the PM scheme
2 % params:
3 % f_c = frequency of carrier
4 % A_c = amplitude of carrier
5 % sig_c = PM-modulated signal
6 % k = k_p, phase deviation coefficient
7 % f_s_m = downsampled sampling rate
8 % f_s_c = upsampled sampling rate
9 % tau = LPF cutoff frequency
10 % returns:
11 % sig_m = demodulated signal at downsampled rate
12 function sig_m = pm_demod(f_c, A_c, sig_c, k, f_s_m, f_s_c, tau)
13     duration = length(sig_c) / f_s_c;
14     t_c = linspace(0, duration, f_s_c * duration);
15     t_m = linspace(0, duration, f_s_m * duration);

16
17     % mix with sine; negative because quadrature component is negative
18     sig_c = -sig_c .* sin(2 * pi * f_c * t_c);

19
20     % LPF and HPF
21     wd = linspace(-pi, pi, length(sig_c));
22     f_c = wd * f_s_c / (2 * pi);
23     hpf = (1 - 20*f_c.^-1*1j).^-1;           % HPF
24     lpf = (1 + f_c/tau*1j).^-1;               % LPF frequency response
25     ft_c = fftshift(fft(sig_c));              % freq. domain rectified signal
26     ft_c = lpf .^ 2 .* hpf .* ft_c;          % apply filter in freq. domain
27     sig_c = ifft(ifftshift(ft_c));

28
29     % downsample
30     sig_m = real(interp1(t_c, sig_c, t_m) / (k * A_c));
31
32     % restore amplitude
33     sig_m = sig_m * 2;
34 end

```

Listing 6: PM demodulation

2.4 FM Modulation

2.4.1 Overview

Frequency modulation involves encoding the message into the frequency of the carrier signal. This first requires the concept of instantaneous frequency, which can be defined as a function of the phase ϕ :

$$f_{inst} = \frac{1}{2\pi} \frac{d\phi}{dt} \quad (28)$$

Now, say that we wish to encode the message signal as a in the frequency, i.e.:

$$f_{inst} = k_f m(t) \quad (29)$$

Here, k_f is the frequency deviation coefficient from the carrier (I.e., if the message has an amplitude of a , then the instantaneous frequency of the modulated signal will be $f_c + a$ at that moment; intuitively, $k_f \max |m(t)| = \Delta f_{max}$) Then we can express the phase as a function of the message signal:

$$\phi(t) = 2\pi k_f \int_{-\infty}^t m(t) dt \quad (30)$$

Then, we can apply the same principles behind PM to modulate this phase. This generates the modulated signal:

$$u_{fm}(t) = A_c \cos \left(2\pi f_{ct} t + 2\pi k_f \int_{-\infty}^t m(t) dt \right) \quad (31)$$

Similar to PM, we can define a modulation factor k_f that indicates the phase shift:

$$\beta_f := \frac{k_f \max |m(t)|}{W} = \frac{\Delta f_{max}}{f_{m,max}} \quad (32)$$

where W is the bandwidth of $m(t)$ (i.e., the highest frequency of the baseband signal $m(t)$). This can intuitively be interpreted to mean the bandwidth efficiency of a modulated signal: a higher β_f indicates a higher carrier modulation for the same input bandwidth (and thus lower bandwidth efficiency). β_f can also be roughly interpreted as the maximum phase deviation (just like β_p for PM), since, for the pure tone $m_{pure}(t) = \cos(2\pi f_0 t)$ ($\max |m(t)| = 1$, $\beta = k_f/f_0$):

$$\begin{aligned} u(t) &= A_c \cos \left(2\pi f_{ct} t + 2\pi k_p \int_{-\infty}^{\infty} \cos(2\pi f_0 t) dt \right) \\ &= A_c \cos \left(2\pi f_{ct} t + \frac{2\pi k_p}{2\pi f_0} \sin(2\pi f_0 t) \right) = A_c \cos(2\pi f_{ct} t + \beta_f \sin(2\pi f_0 t)) \end{aligned} \quad (33)$$

Clearly, $\beta_f = \Delta\phi_{max}$ in this case.

Demodulation of an FM-modulated signal can be performed identically to a PM demodulator, except that it would require an additional division by $2\pi k_f$ at the conclusion and a differentiation stage. Note that the first demodulation scheme involves a differentiation, envelope detection, and an integration stage; if we add another differentiation stage at the end of this, it would cancel out the integration stage and make it a simple two-step process very similar to conventional AM. Since differentiation and envelope detection are both easy to implement in hardware, this makes conventional AM and FM the most commonly implemented modulation schemes.

2.4.2 Analysis

As with PM, I defer the derivation of the noise power and SNR to the class notes, or section 5.3 of [2].

$$P_m = k_f^2 P_m \quad (34)$$

$$P_n = \frac{2W^3 N_0}{3A_c^2} \quad (35)$$

$$\text{SNR} = \frac{3k_f^2 A_c^2}{2W^2} \frac{P_m}{N_0 W} = \frac{3A_c^2}{2} \left(\frac{\beta_f}{\max |m(t)|} \right)^2 \frac{P_m}{N_0 W} \quad (36)$$

Again, the calculation for noise power was derived in the analog domain. See 10 for details about the digital domain calculation. What is a little confusing is that the bandwidth W in the first term in (36) can be in analog units (it should be the same value used when defining k_f), whereas the W in the second term indicates the cutoff frequency and should be in digital radian units when working in a sampled space. The equivalent expression using β_f is a little nicer because β_f is invariant to the domain (digital or analog), and so it hides this ugliness.

2.4.3 Implementation details

This uses the cheap method involving differentiation and conventional AM demodulation (envelope detection). The modulation implementation is pretty straightforward. The only design decision was to use `cumsum` as a simple running integral.

Notes about demodulation:

- Differentiation is performed in the time domain using a difference equation approximation. It is padded with a single zero to maintain the length.
- Like in the conventional AM case, half-wave rectification is arbitrarily chosen (as opposed to full-wave rectification).
- A first-order LPF and HPF were used to filter the circuit. As in the previous examples, the LPF is primarily to filter out the double frequency carrier and the DC components created by the carrier mixing.
- The LPF was arbitrarily applied multiple times, similar to the SSB case, in order to reduce additional noise at higher frequencies.

```

1 % modulates a signal using the FM scheme
2 % params:
3 % f_c = frequency of carrier
4 % A_c = amplitude of carrier
5 % sig_m = message signal
6 % k = k_f, frequency deviation coefficient
7 % f_s_m = downsampled sampling rate
8 % f_s_c = upsampled sampling rate
9 % returns:
10 % sig_c = FM-modulated signal at upsampled rate
11 function sig_c = fm_mod(f_c, A_c, sig_m, k, f_s_m, f_s_c)
12     duration = length(sig_m) / f_s_m;
13     t_m = linspace(0, duration, length(sig_m));
14     t_c = linspace(0, duration, f_s_c * duration);
15
16     % upsample sig_m, multiply by k, integrate and multiply by 2pi to get
17     % phase (in rad)
18     sig_phi = 2 * pi * k * interp1(t_m, cumsum(sig_m) / f_s_m, t_c);
19
20     % generate phase-modulated signal (since FM is basically PM)
21     sig_c = A_c * cos(2 * pi * f_c * t_c + sig_phi);
22 end

```

Listing 7: FM modulation

```

1 % demodulates a signal modulated using the FM scheme
2 % params:
3 % sig_m = message signal
4 % A_c = amplitude of carrier
5 % f_s_m = downsampled sampling rate
6 % f_s_c = upsampled sampling rate
7 % k = k_f, frequency deviation coefficient
8 % tau = LPF cutoff frequency
9 % returns:
10 % sig_m = FM-demodulated signal at downsampled rate
11 function sig_m = fm_demod(sig_c, A_c, f_s_m, f_s_c, k, tau)
12     % differentiate (same as multiplying by jw in freq domain)
13     sig_c = [(diff(sig_c) * f_s_c) 0];
14
15     % rectify signal
16     sig_c(sig_c < 0) = 0;
17
18     % lpf and hpf
19     wd = linspace(-pi, pi, length(sig_c));
20     f_c = wd * f_s_c / (2 * pi);
21     hpf = (1 - 20*f_c.^-1*1j).^-1;           % HPF
22     lpf = (1 + f_c/tau*1j).^-1;               % LPF frequency response
23     ft_c = fftshift(fft(sig_c));              % freq. domain rectified signal
24
25     % apply LPF multiple times in a row to completely get rid of carrier
26     % frequency
27     ft_c = lpf .^ 2 .* hpf .* ft_c;          % apply filter in freq. domain
28     sig_c = ifft(ifftshift(ft_c));
29
30     % factor of 3.4 found empirically (similar to conventional AM demod)
31     sig_c = 3.4 * real(sig_c) / (2 * pi * k * A_c);
32
33     % downsample sig_c
34     duration = length(sig_c) / f_s_c;
35     t_m = linspace(0, duration, f_s_m * duration);
36     t_c = linspace(0, duration, length(sig_c));
37     sig_m = interp1(t_c, sig_c, t_m);
38 end

```

Listing 8: FM demodulation

2.5 Modulation schemes comparison

See Table 1 for a general comparison between modulation schemes.

	CONV	SSB	FM
PM	PM more robust to noise CONV cheaper	Similar I-Q demod scheme SSB more bandwidth efficient PM more robust to noise	FM cheaper FM more common
FM	FM more robust to noise Similar cheap demod scheme	FM cheaper SSB more bandwidth efficient PM more robust to noise	
SSB	CONV cheaper SSB use half bandwidth SSB transmits double signal power		

Table 1: Comparison of modulation schemes

Power Conventional AM uses half the power of SSB. Angle-modulation schemes are usually more power-efficient because they can increase β to increase SNR instead of increasing A_c .

Bandwidth Conventional AM uses double the bandwidth of SSB. Angle-modulation schemes are usually less bandwidth-efficient because they use a larger bandwidth (larger β) to increase SNR without increasing A_c .

SNR (and modulation index) In all of the schemes, increasing the carrier amplitude increases the SNR (but is less power efficient). In SSB AM, this is the only way to affect the SNR (assuming the message power is fixed). In conventional AM, increasing a also increases the SNR, but this also decreases power efficiency, and it is limited by the fact that $0 \leq a \leq 1$ so that no clipping occurs at demodulation. In the angle modulation schemes, in addition to the option of increasing the carrier amplitude, increasing the modulation index β increases SNR at no power cost, and is not as limited as a is; however, this uses a larger bandwidth.

2.6 “Fairly” comparing different modulation schemes

The “design problem” of this assignment is that there are multiple parameters to vary, and most do not affect the noiseless demodulated output (but affect the SNR when noise is added to the modulated signal). For example, the demodulated signal of a no-noise modulated signal is invariant to increases in A_c , but the SNR is directly related. In other words, if we set the criterion for choosing parameters to be such that the demodulated signal has the same power, there are infinitely-many possible combinations of parameters to choose from. This can be seen to make the estimated SNRs exactly the same in the simulations below (see Table 5), and the experimental SNR values are very close.

The assignment asks us to “ensure that the signal power will be equal at the output in each [modulated] scheme.” This means that the criterion for choosing parameters is such that the *power of the component of the modulated signal encoding the message* is equal between each modulated scheme. This is easily done with conventional AM and SSB AM, where the power of the message

signal component P_y are given by (derived in their respective sections):

$$P_{y,conv} = \frac{A_c^2 a^2}{2} P_m \quad (37)$$

$$P_{y,ssb} = A_c^2 P_m \quad (38)$$

In this case, the modulation index a conventional AM is arbitrarily set to 0.5 (which allows us to double it later without worrying about clipping), and the carrier amplitude A_c is arbitrarily set to 4. To match this, in the SSB case $A_{c,ssb} \leftarrow A_{c,conv}/\sqrt{8}$.

For the angle-modulated cases, I am not sure how to solve it because of the complex nature of the Fourier transform and PSD (i.e., with the Bessel functions). Thus we give up on this idea of fair comparison and just give the FM and PM the same A_c as the conventional AM case.

The assumption that signal power will be equal in each modulation scheme is unfair because this is (probably) not what happens in the real world: different modulation schemes probably get transmitted at vastly different signal powers and different transmitted powers. In other words, I give up on the idea of directly trying to compare the different modulation schemes to each other with some absolute measure (e.g., comparing their SNRs at some given noise variance level to determine which is overall more resistant to noise), but rather observe the general trends w.r.t. changes in their parameters.

3 Results

	CONV	SSB	PM	FM
modulation index	$a = 0.5$	—	$k_p = 0.1$	$k_f = 10W \approx 60650\text{Hz}$
carrier amplitude A_c	4	$\sqrt{2}$	4	4
carrier frequency f_c	500000	500000	500000	400000
sampling rate of modulated signal $f_{s,c}$	2000000	2000000	5000000	5000000
USSB/LSSB	—	LSSB	—	—
noise variance (when varying modulation index)	1	—	1	0.5

Table 2: Parameters for modulation schemes

		CONV	SSB	PM	FM
Total transmitted power	Experimental	8.1044	0.6031	12.4990	8.000
	Theoretical	8.1042	0.6044	12.5000	8.000
	% error	0.00%	0.22%	0.01%	0.00%
Demodulated power	Experimental	0.0546	0.0513	0.0498	0.0553
	Theoretical	0.0522	0.0522	0.0522	0.0522
	% error	4.60%	1.72%	4.60%	5.94%

Table 3: Expected vs. theoretical power of modulated (transmitted) and demodulated signals

	PM	FM
% power of signal in Carson's bandwidth	100.00%	100.00%

Table 4: Percent power of signal in Carson's bandwidth for angle-modulated signals

		$\sigma = 0.1$	$\sigma = 0.5$	$\sigma = 1$
CONV SNR (dB)	Experimental	22.52	15.12	9.83
	Theoretical	31.99 (28.98 †)	18.01 (15.00 †)	11.99 (8.98 †)
SSB SNR (dB)	Experimental	21.40	13.55	8.11
	Theoretical	31.99 (28.98 †)	18.01 (15.00 †)	11.99 (8.98 †)
PM SNR (dB)	Experimental	20.93	9.66	3.79
	Theoretical	21.99	8.01	1.99
FM SNR (dB)	Experimental	19.46	9.97	2.32
	Theoretical	66.76	52.78	47.76

Table 5: Effect of varying noise variance on SNR. † using an extra scaling factor of 2

		Modulation index (a , β_p , or β_f)		
		0.5*original	original	2*original
CONV SNR (dB)	Experimental	4.06	9.83	14.79
	Theoretical	5.97 (2.96 †)	11.99 (8.98 †)	18.01 (15.00 †)
PM SNR (dB)	Experimental	-2.26	3.72	9.64
	Theoretical	-4.03	1.98	8.01
FM SNR (dB)	Experimental	3.43	9.97	13.54
	Theoretical	45.93	51.95	57.97

Table 6: Effect of varying modulation index on SNR. † using an extra scaling factor of 2

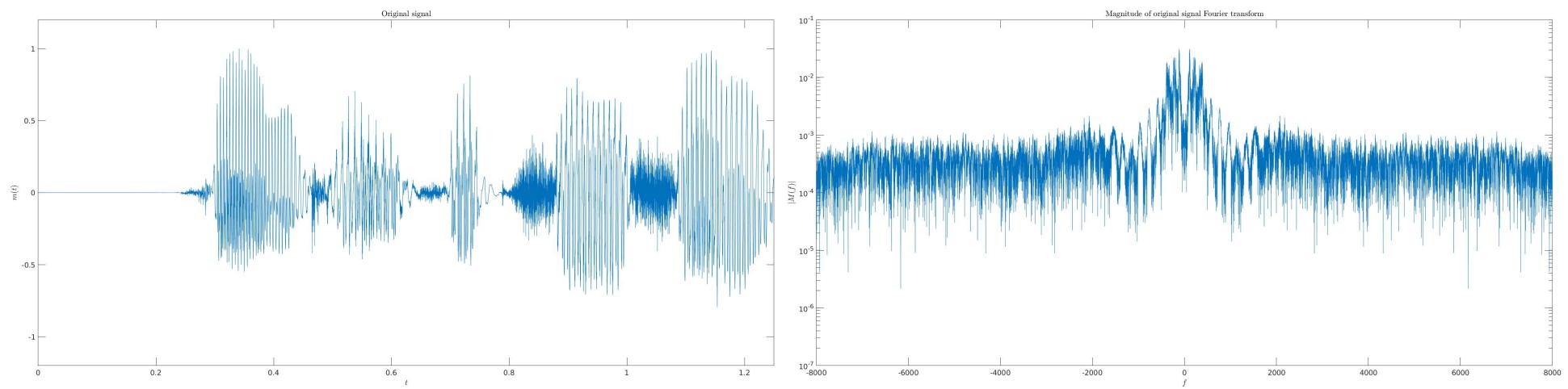


Figure 1: Original message signal

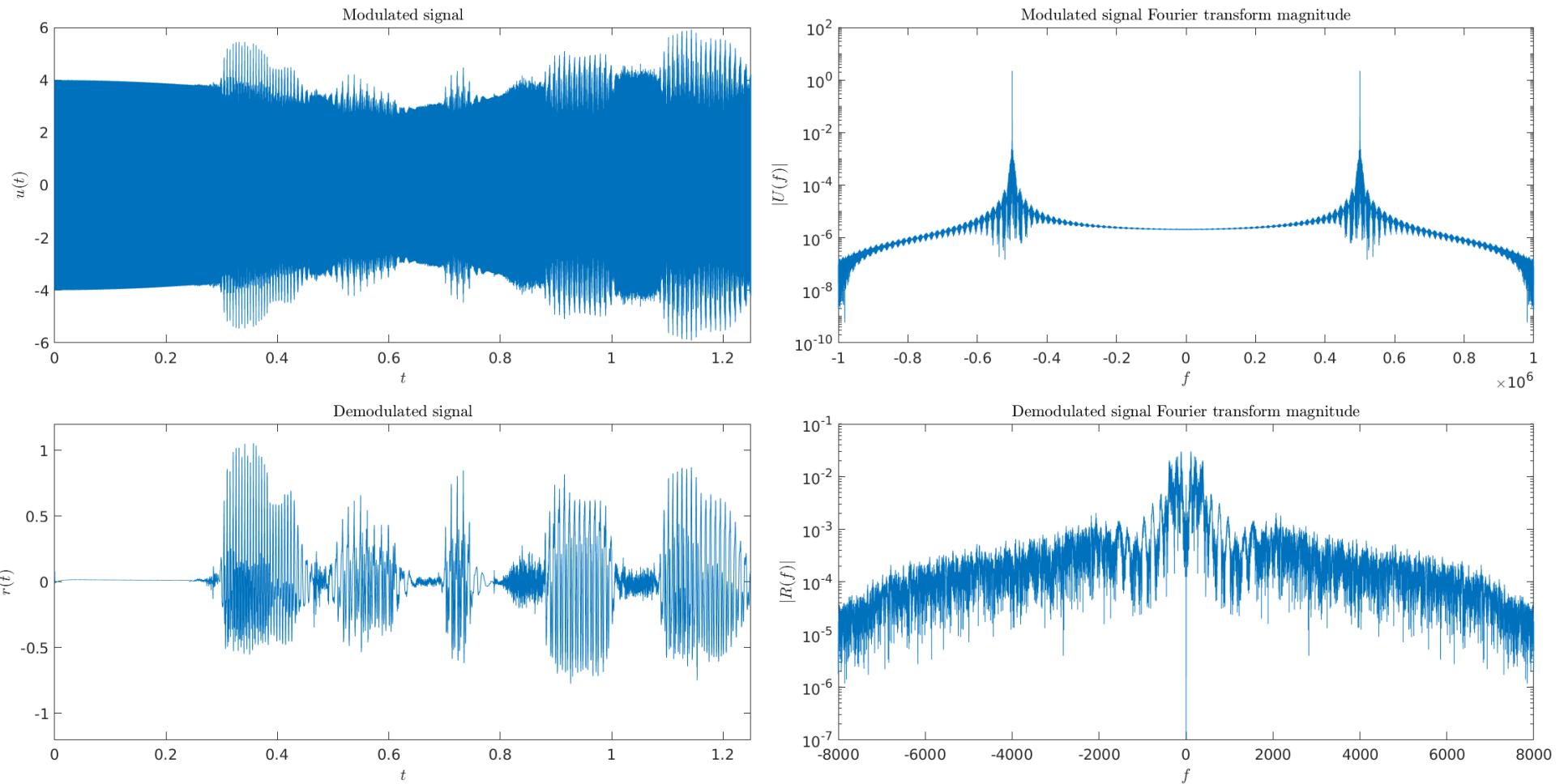


Figure 2: Conventional AM modulated and demodulated signals

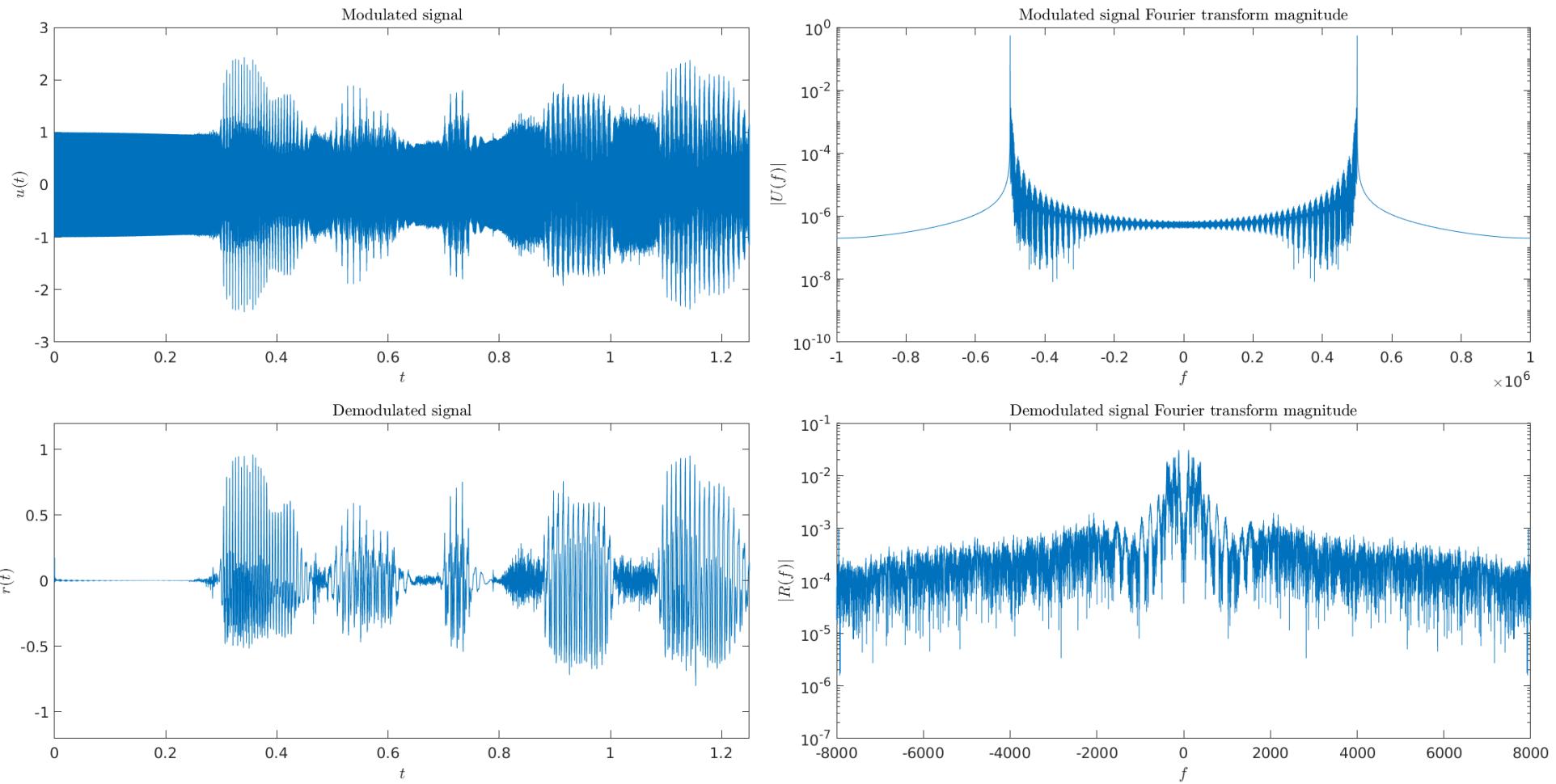


Figure 3: SSB AM modulated and demodulated signals

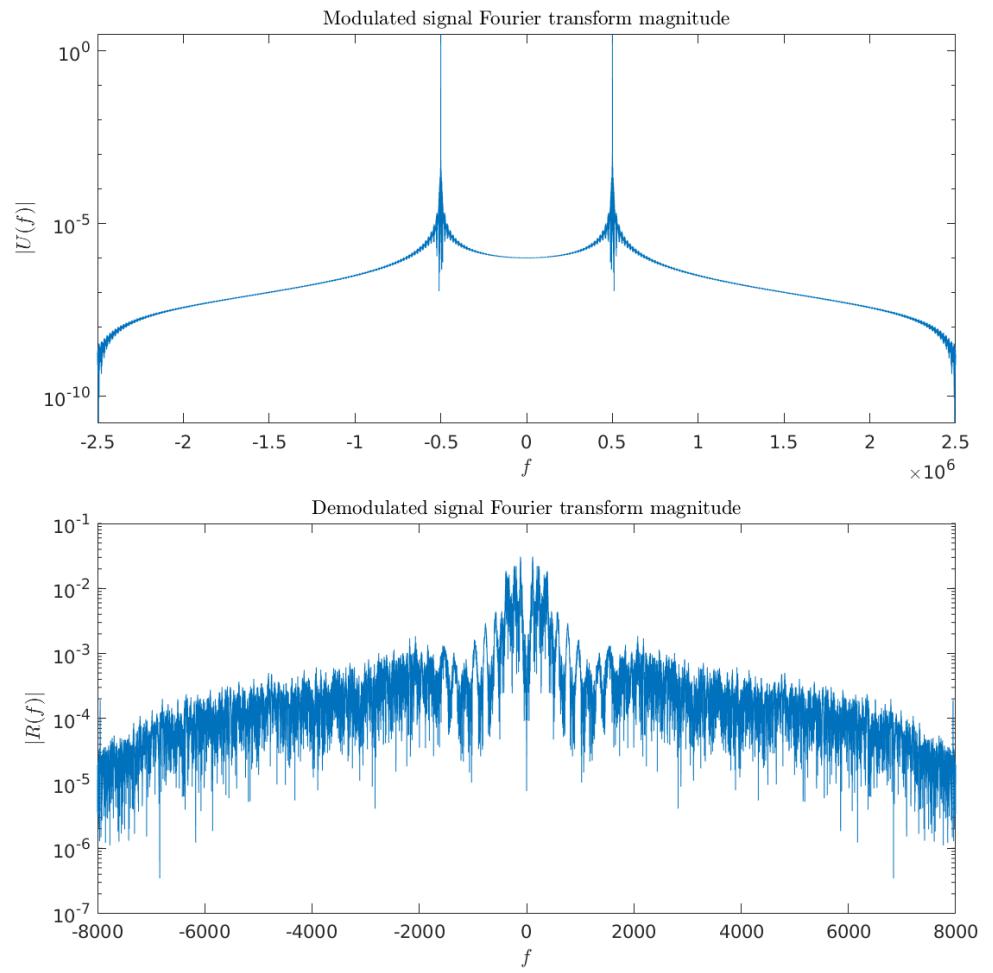
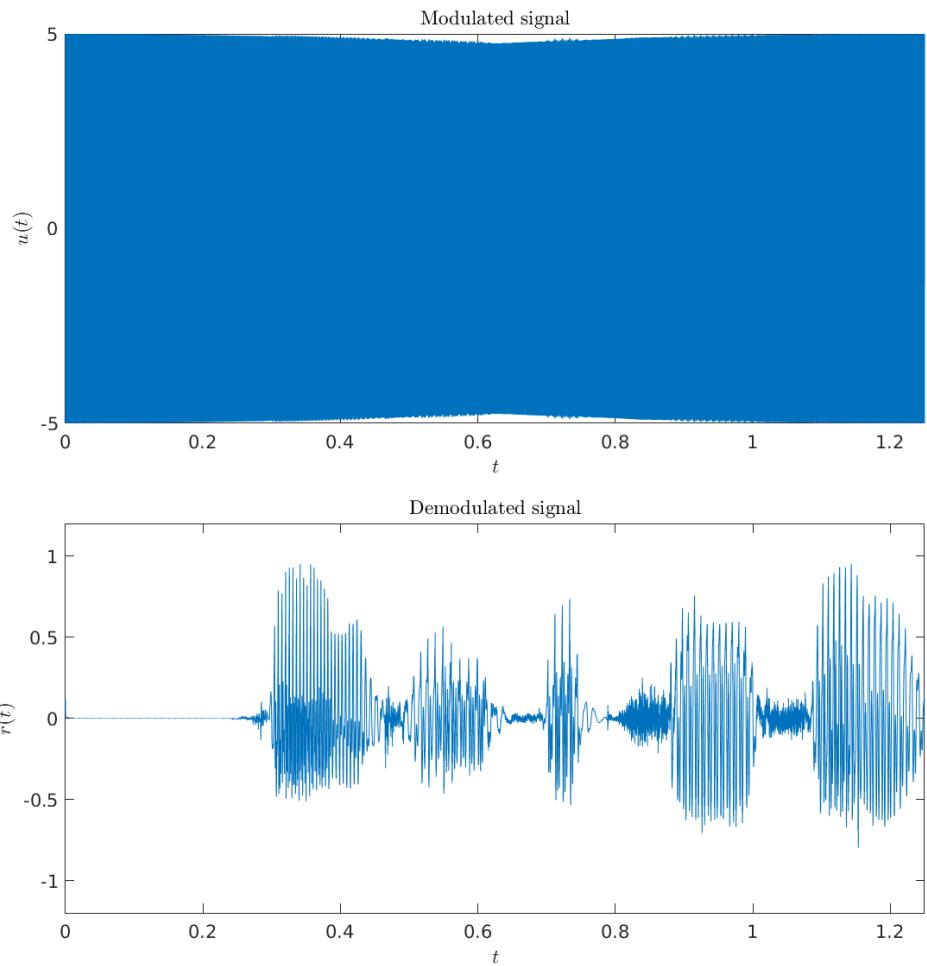


Figure 4: PM modulated and demodulated signals

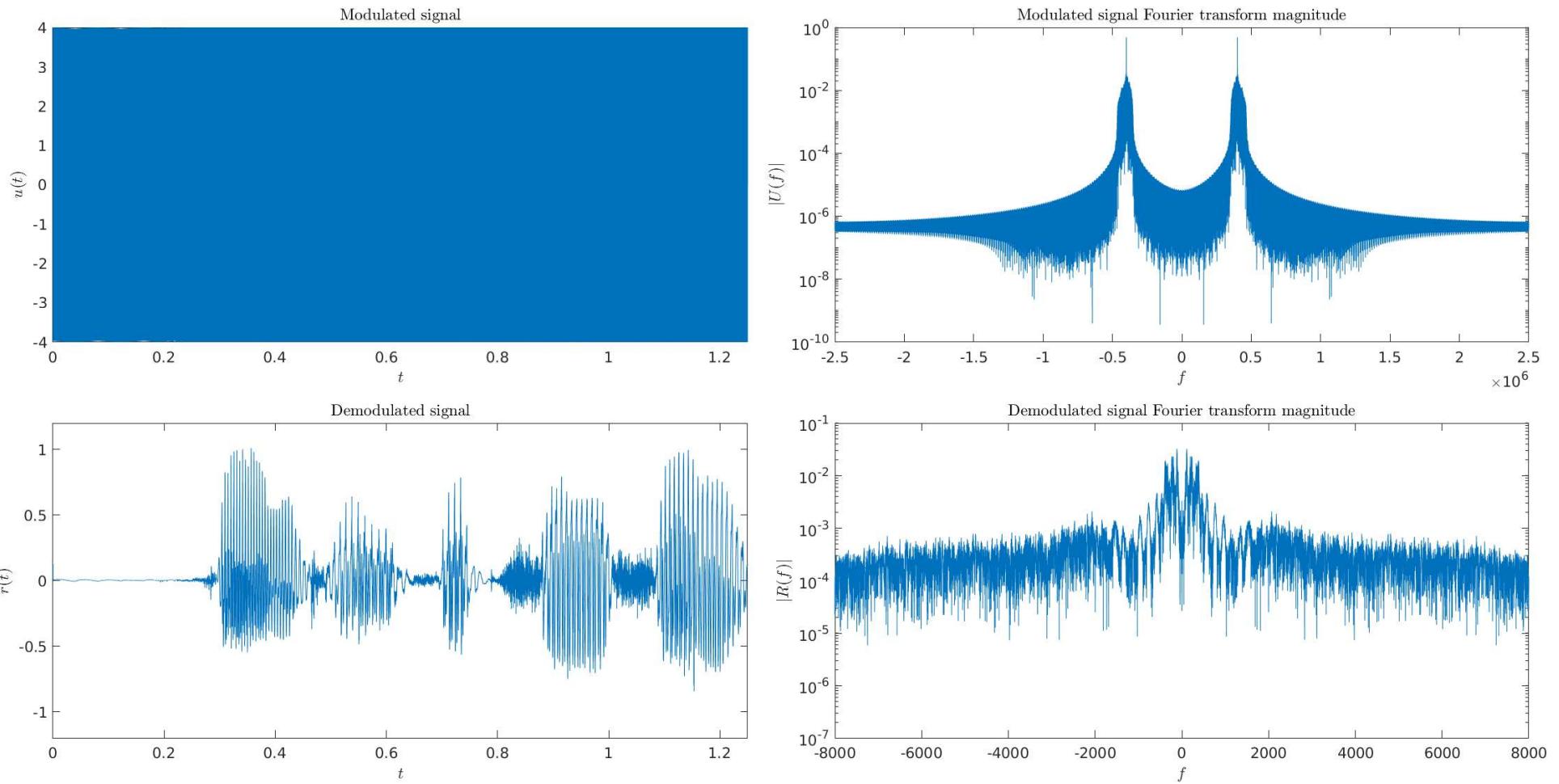


Figure 5: FM modulated and demodulated signals

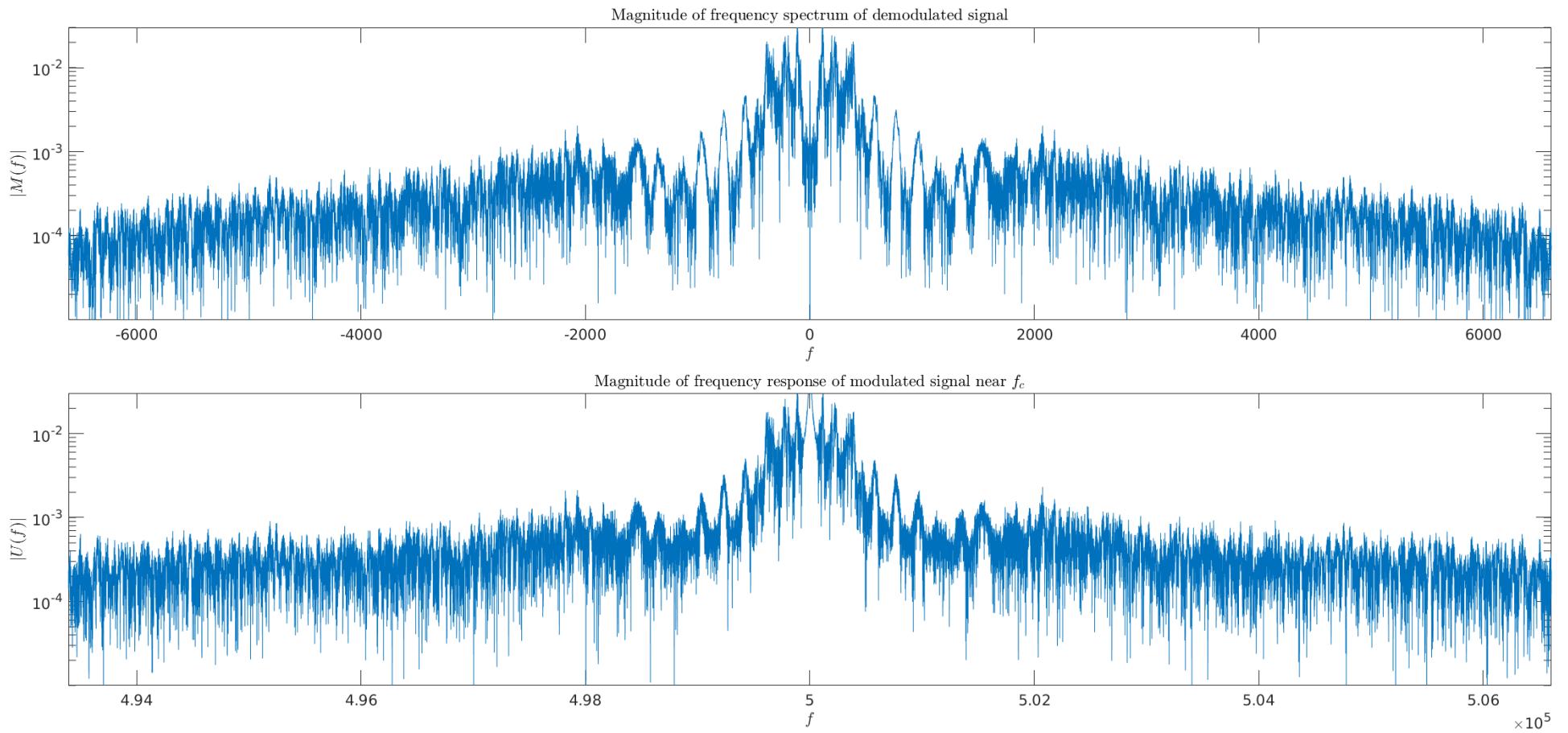


Figure 6: Close-up comparison of conventional AM baseband and modulated spectra

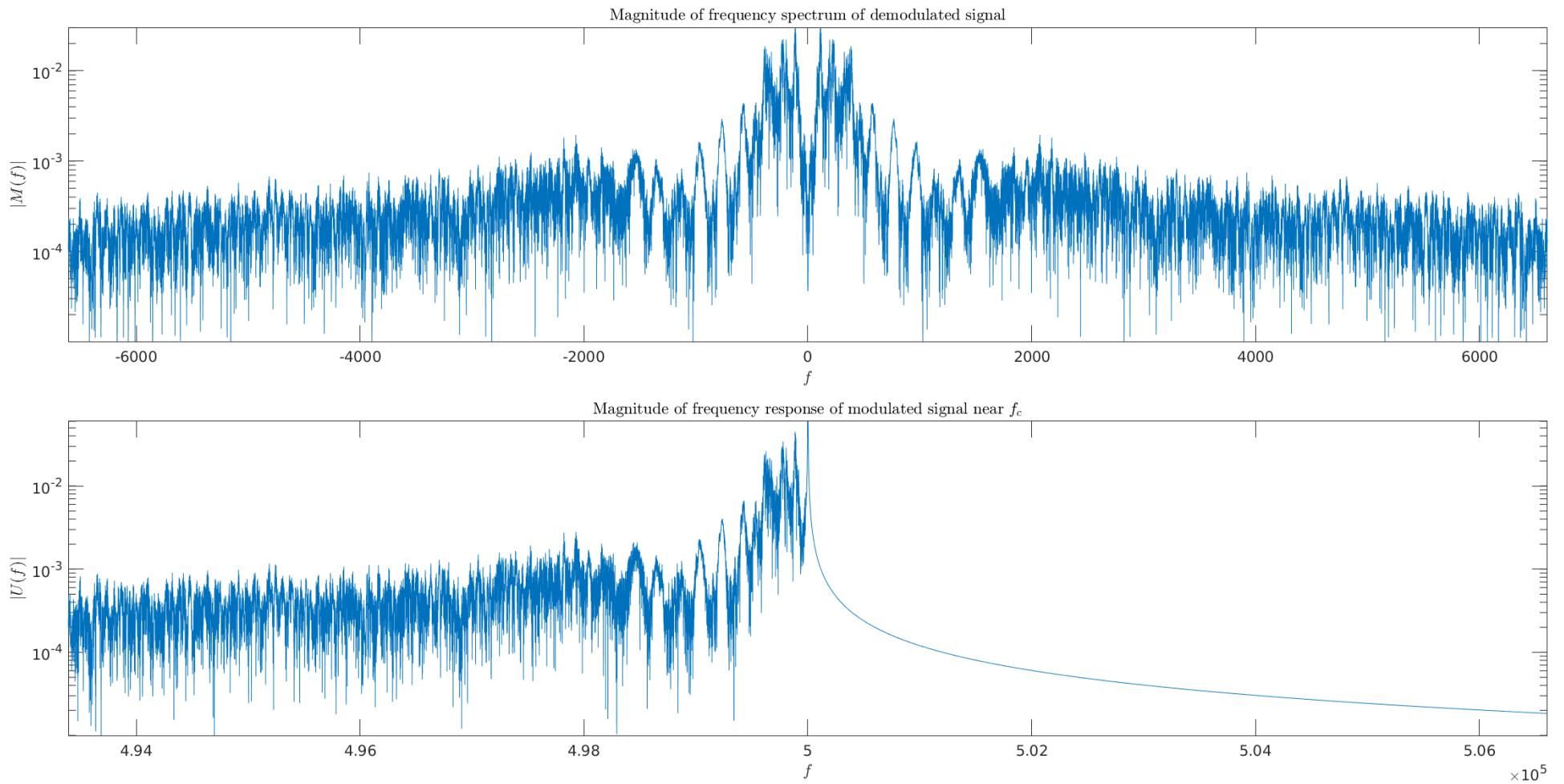


Figure 7: Close-up comparison of SSB AM baseband and modulated spectra

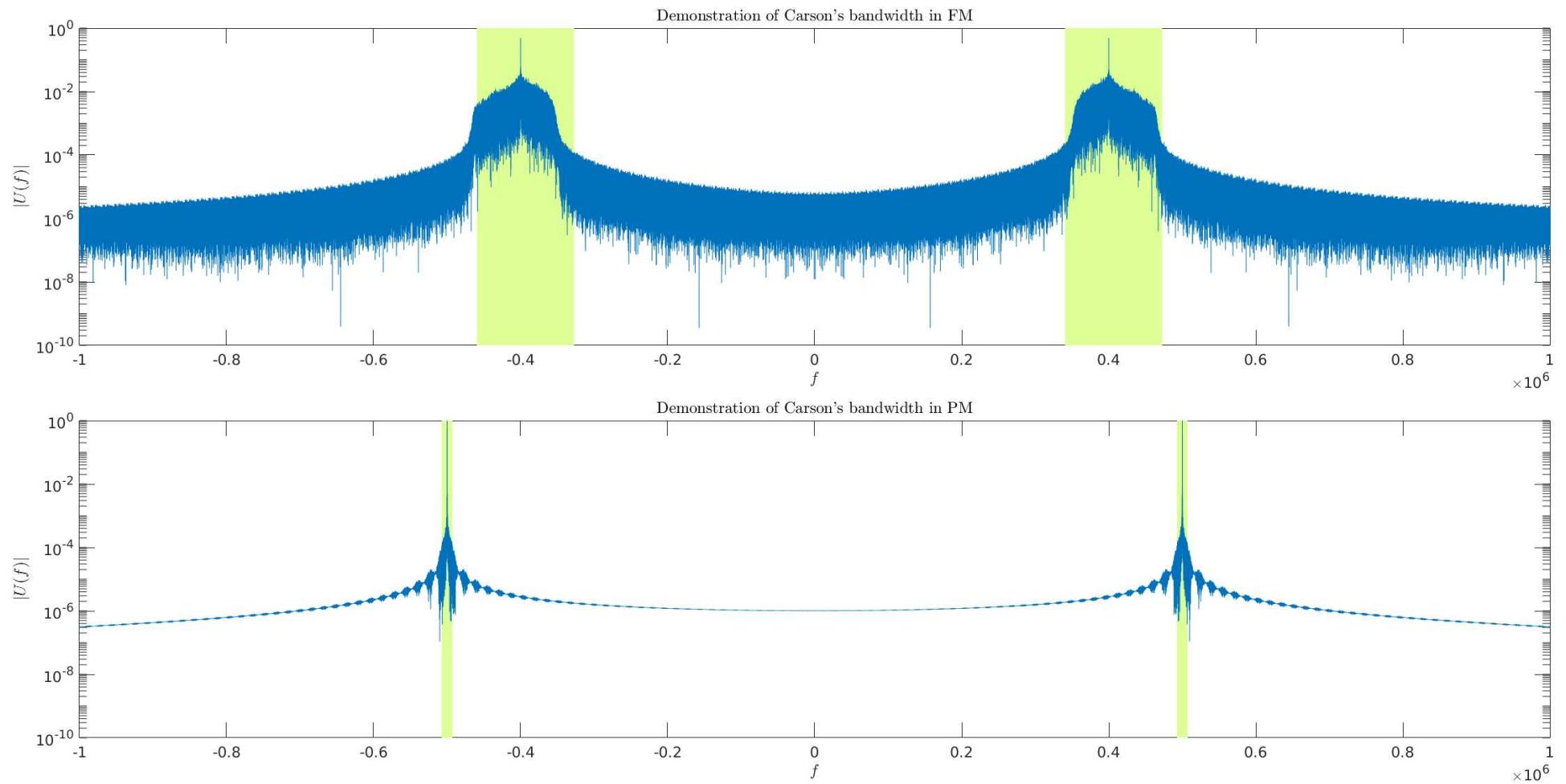


Figure 8: Demonstration of Carson's rule

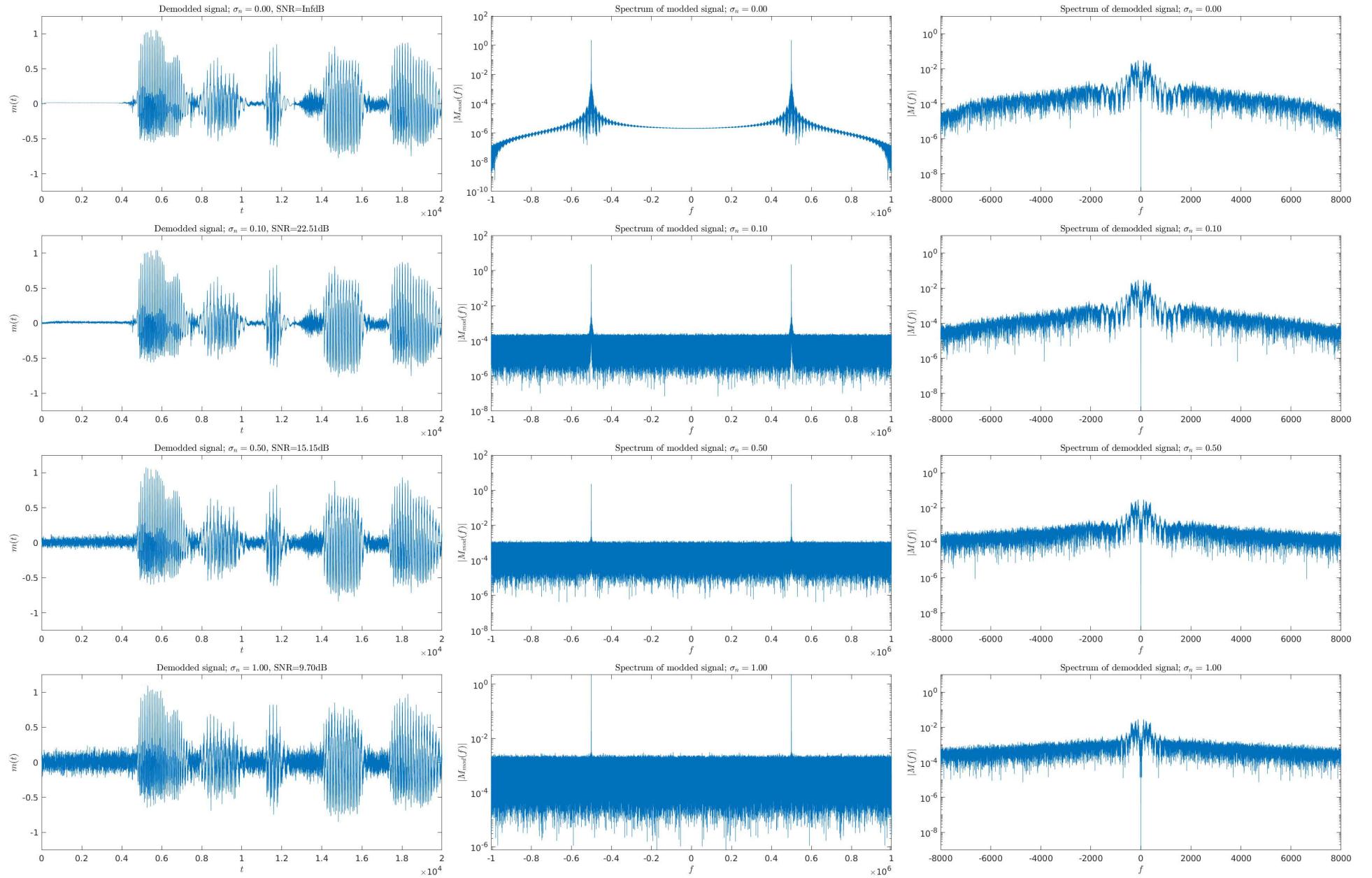


Figure 9: Effect of varying noise variance on conventional AM

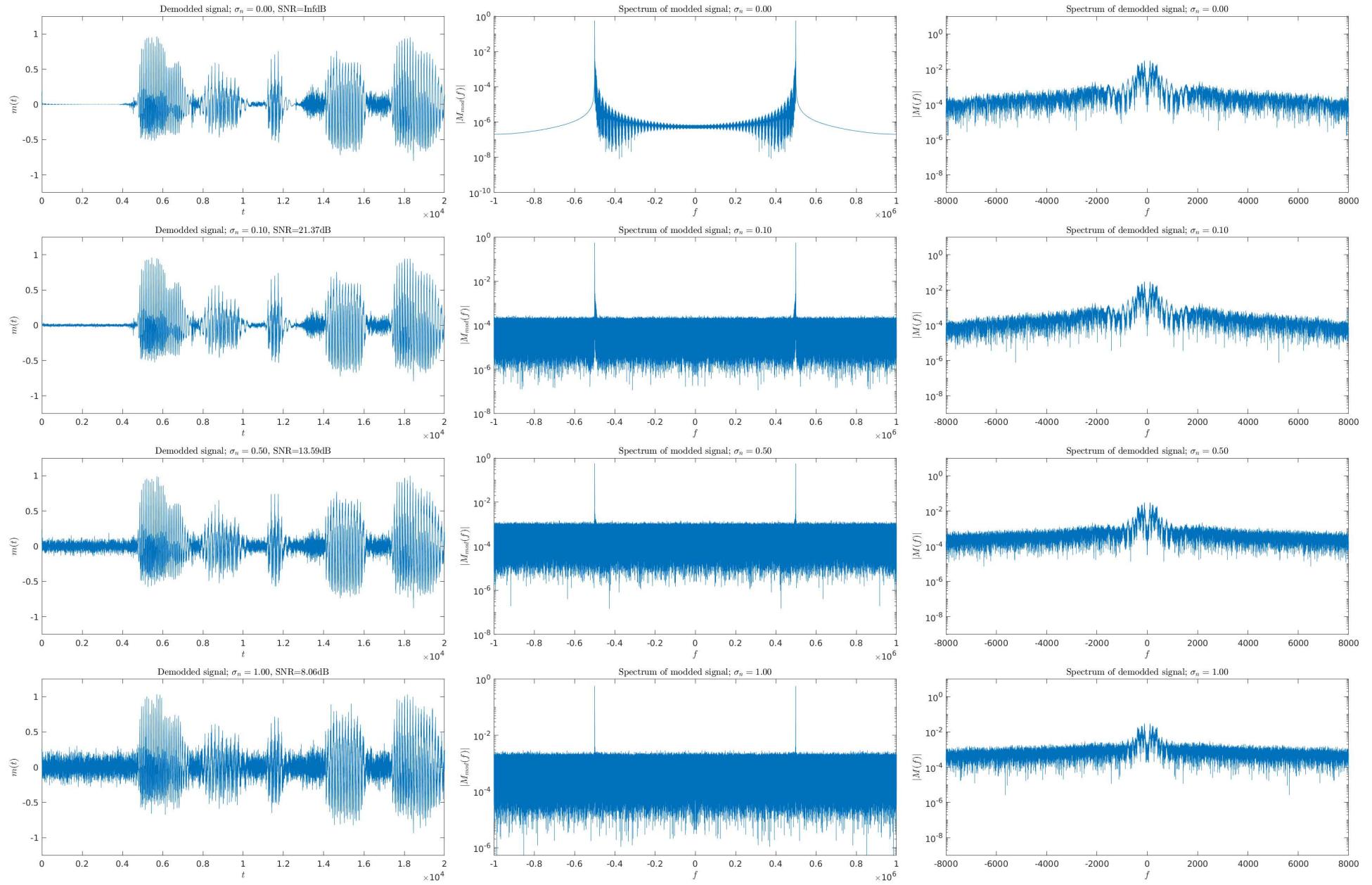


Figure 10: Effect of varying noise variance on SSB AM

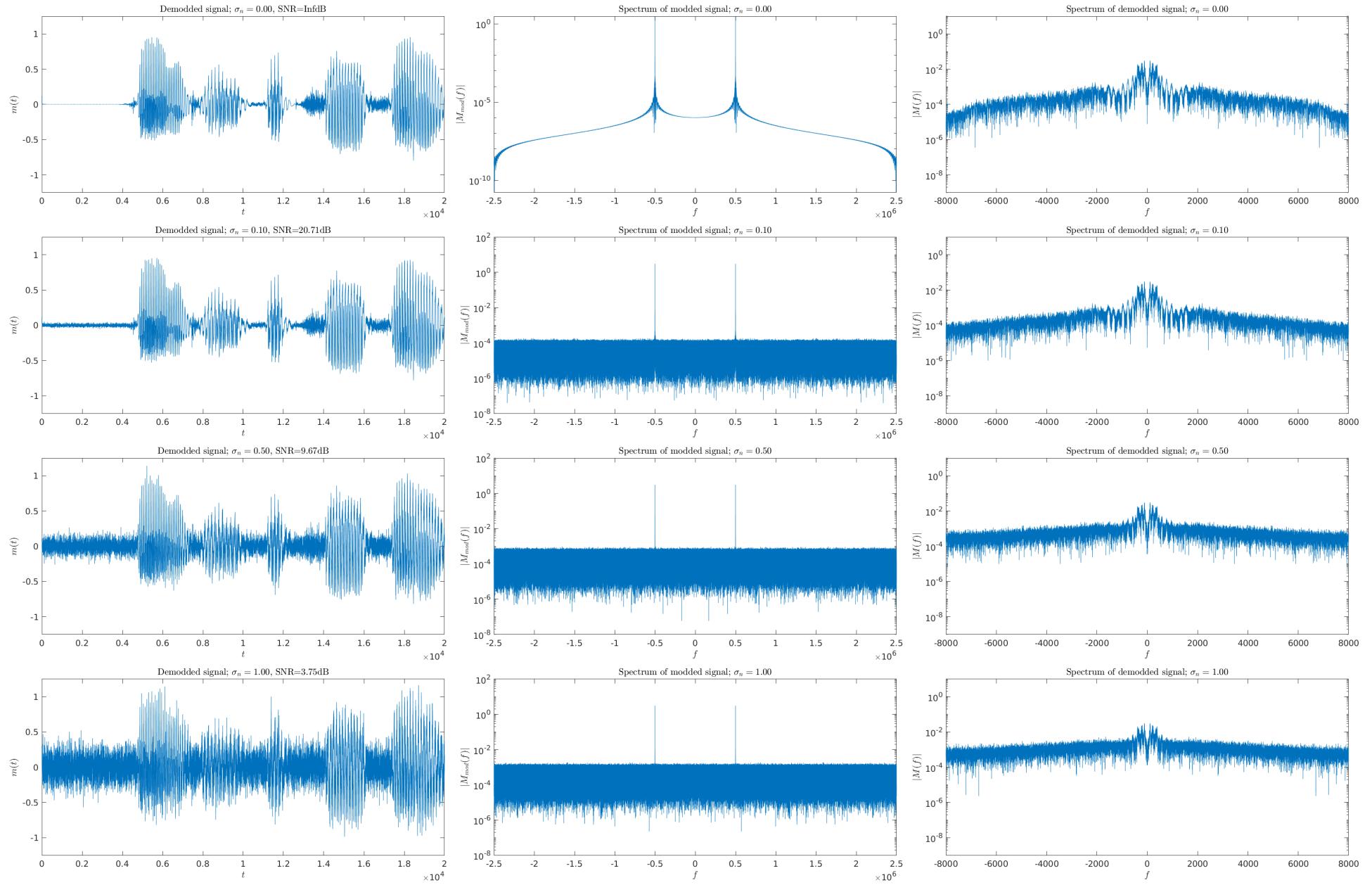


Figure 11: Effect of varying noise variance on PM

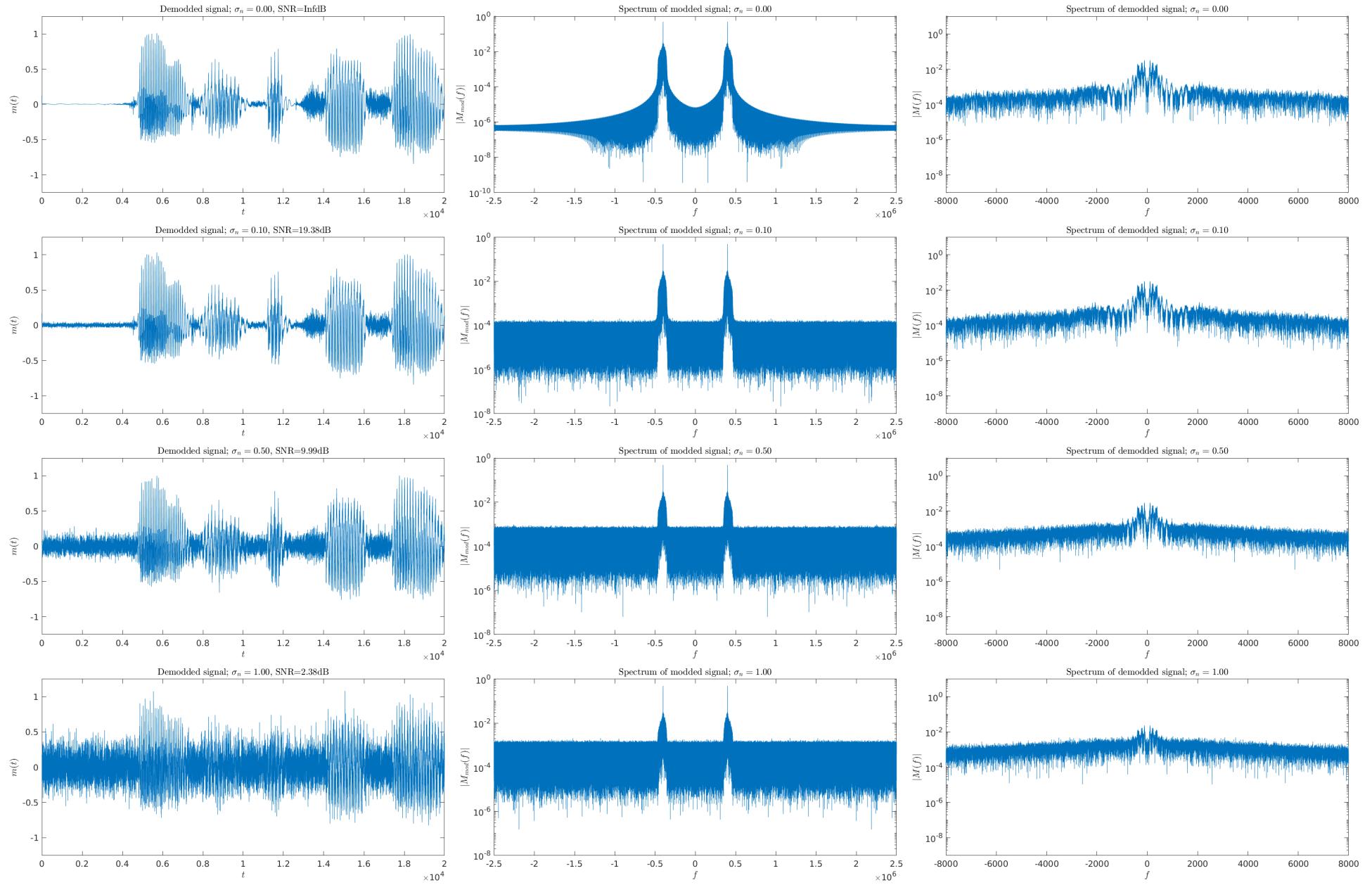


Figure 12: Effect of varying noise variance on FM

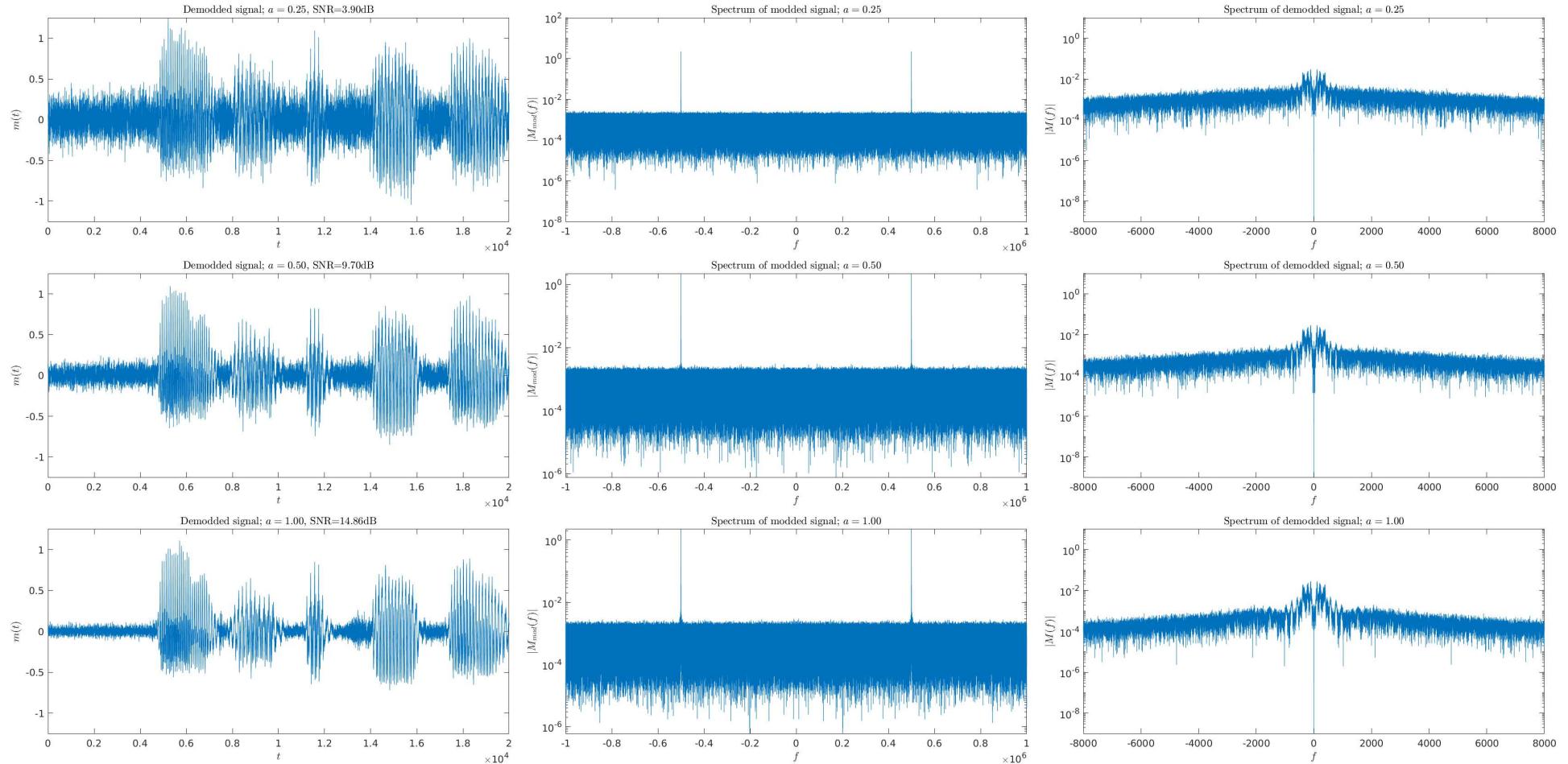


Figure 13: Effect of varying modulation index on conventional AM

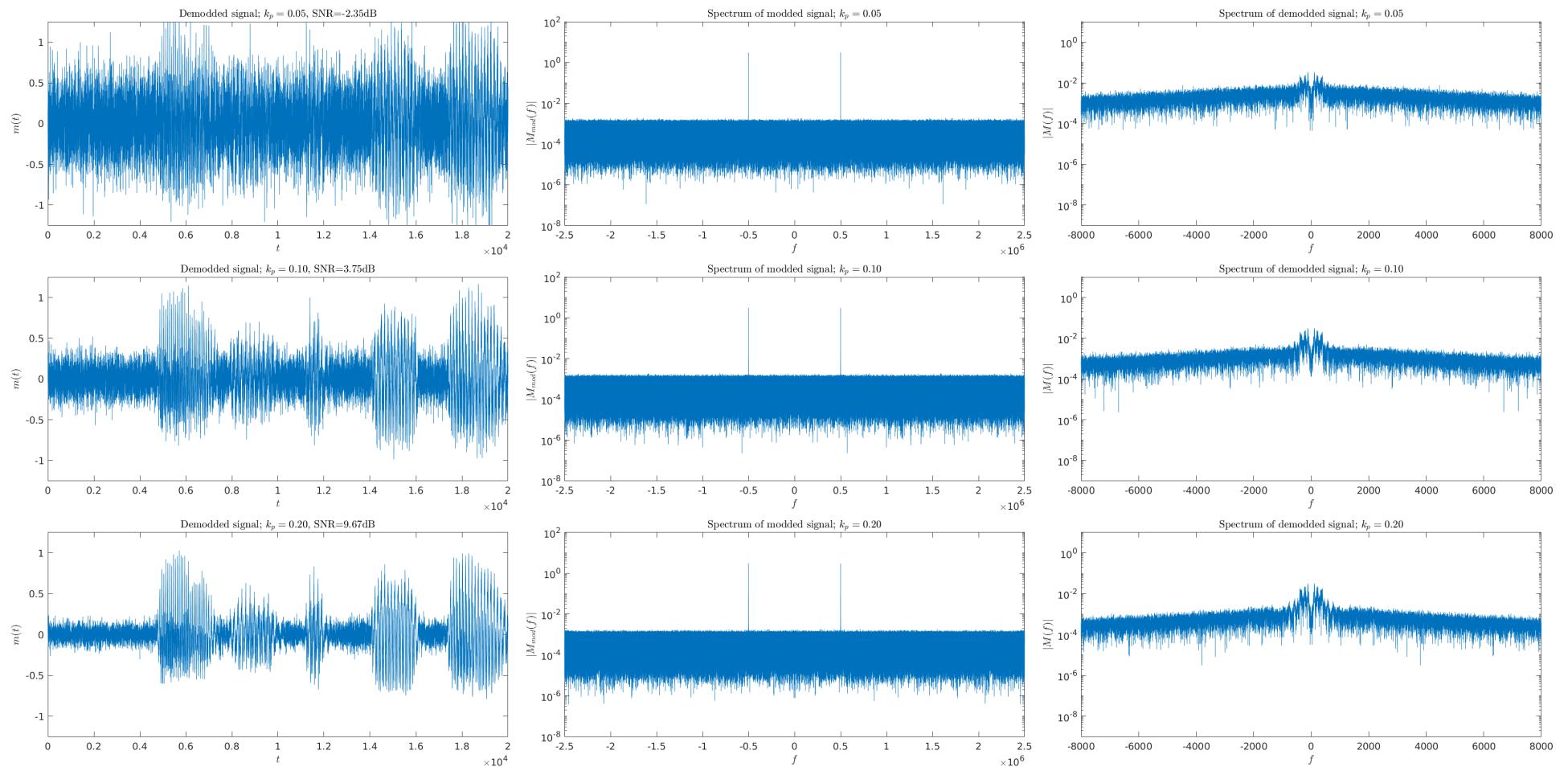


Figure 14: Effect of varying modulation index on PM

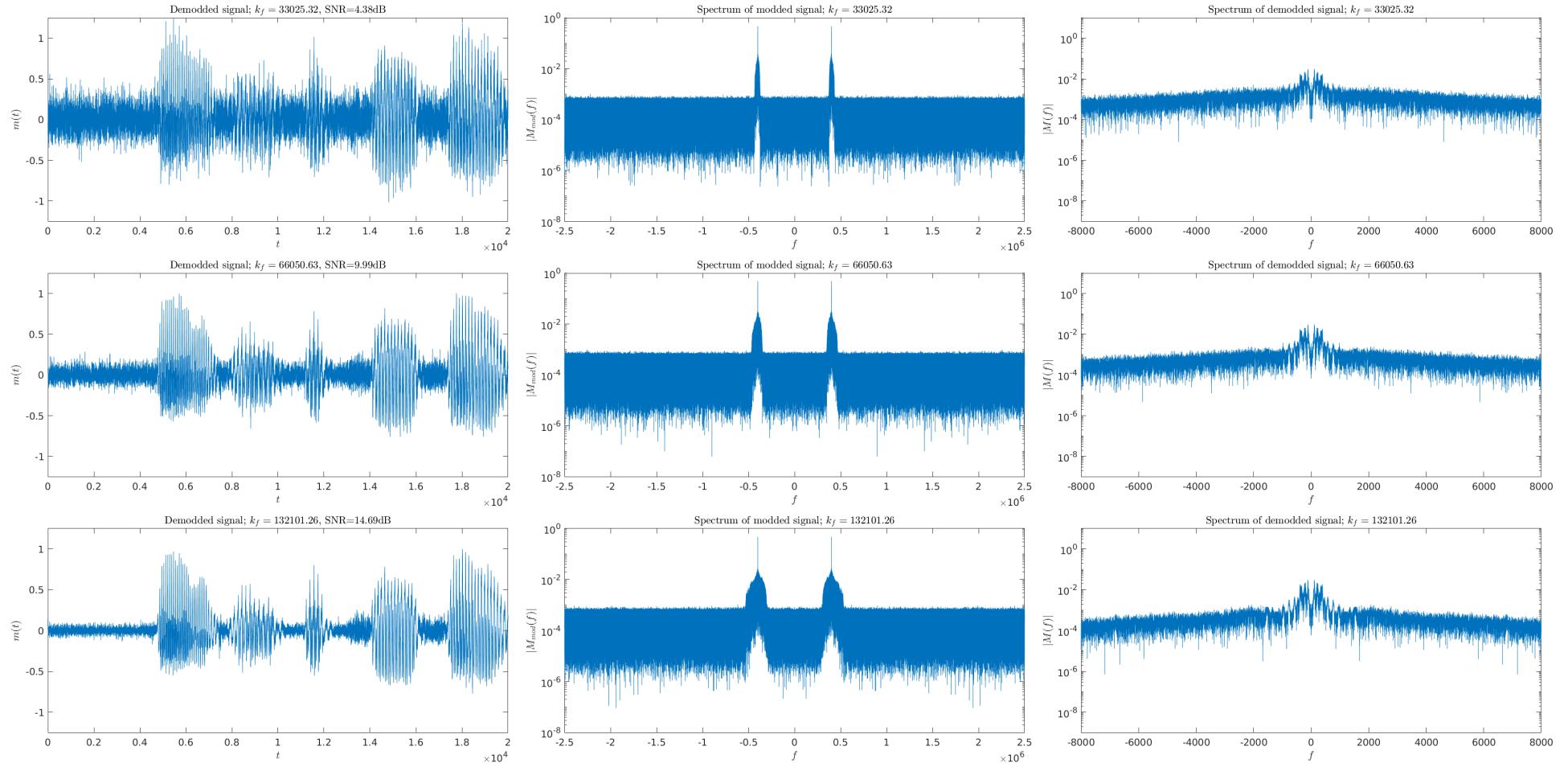


Figure 15: Effect of varying modulation index on FM

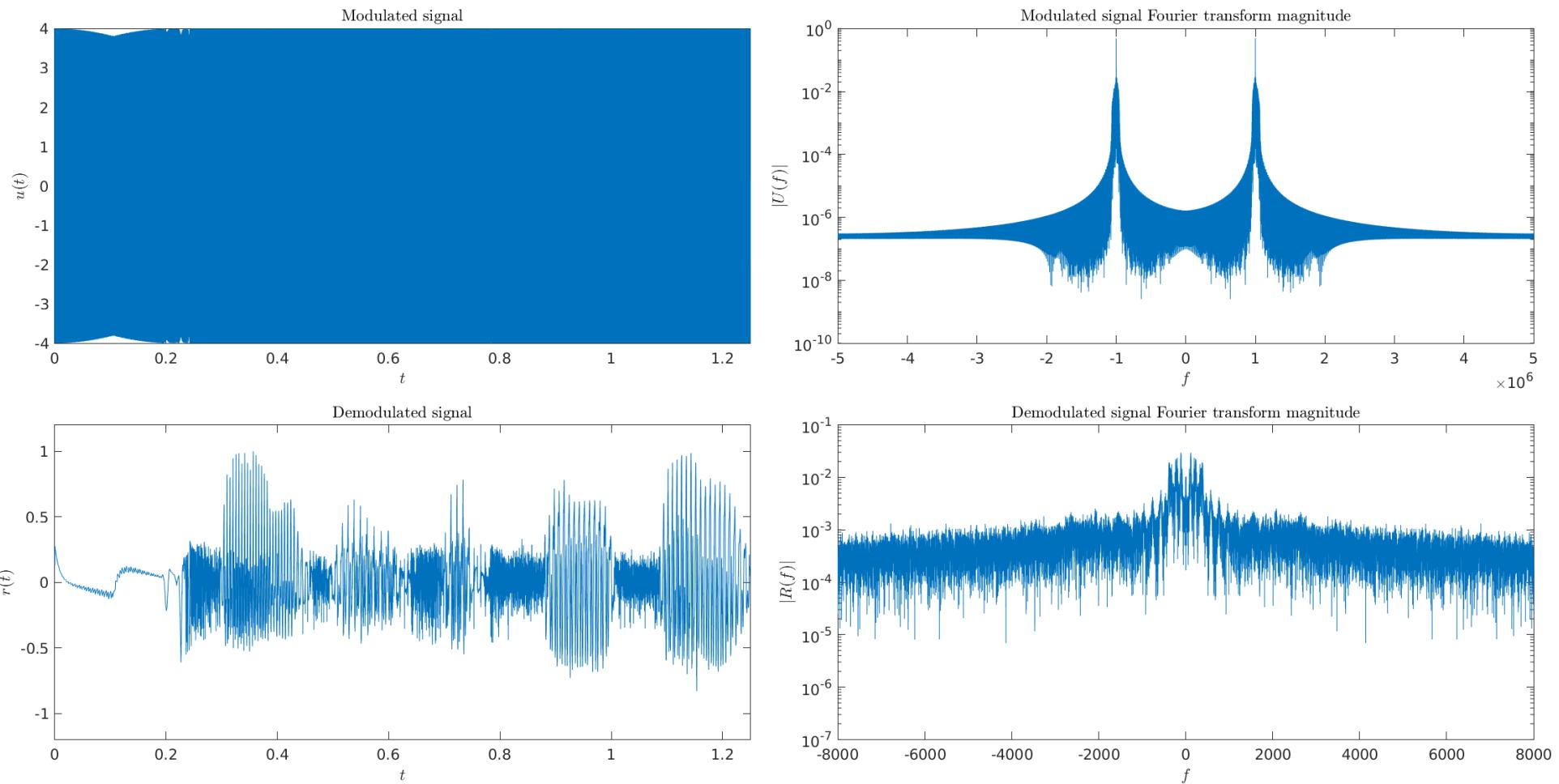


Figure 16: Increased distortion with higher frequencies with FM ($f_c = 1\text{MHz}$, $f_{s,sc} = 10\text{MHz}$)

4 Discussion

4.1 The message signal

The original message signal was scaled to have unity maximum amplitude, and it has a bandwidth (calculated using `obw`) of 6605Hz. Figure 1 is a plot of this signal and its Fourier transform.

4.2 Verification of the modulation schemes

The first order of business was to verify that each modulation and demodulation scheme works. The parameters for each of the modulation schemes is shown in Table 2. The modulated and demodulated signals were plotted in the time and frequency domains in Figures 2 (conventional AM), 3 (SSB AM), 4 (PM), and 5 (FM).

Also, the total transmitted power of the modulated signals, and the power of the demodulated signals, for each scheme are given in Table 3. It is clear from this table that the values match the theoretical values. (However, this doesn't mean much in the case of the demodulated conventional AM and demodulated FM cases, because there was an empirically-determined scalar multiplier in the demodulator used to restore the original amplitude.)

The outputs from each scheme are clearly audibly recognizable. Conventional AM and FM both have a very slight staticky sound, which is probably due to rectification noise.

4.2.1 Conventional AM

It is clear from the modulated signal plotted in the time domain that $A_c = 4$, and the message signal is clearly visible as the envelope of the modulated signal. In the Fourier transform of the modulated signal, we can see that there are two spikes at the carrier frequency as expected. The demodulated signal looks comparable to the original signal, as well as its Fourier transform; the only difference is that the DC offset is explicitly zero (from the demodulation function), so there is a (nonconsequential) negative spike at $f = 0$.

In addition, Figure 6 is a plot of the Fourier transform of the modulated signal shown zoomed horizontally and centered horizontally at $f = f_c$ and compared to the Fourier transform of the baseband message signal. This clearly demonstrates the function of mixing with a pure tone at the carrier frequency.

4.2.2 SSB AM

Similar to conventional AM, the envelope of the message signal is clearly visible in the carrier. The Fourier transform clearly shows a difference between the lower and upper sidebands, as expected. The output signal looks similar to the message signal in both time and frequency domain.

Similarly to conventional AM, Figure 7 is a plot of the Fourier transform of the modulate signal shown zoomed horizontally and centered horizontally at $f = f_c$ and compared to the Fourier transform of the baseband message signal. It is clear that all of the power is stored in the lower side band, as expected.

4.2.3 PM

As expected, the modulated signal is hard to see: the amplitude is more or less constant, with only a small visible distortion. The Fourier transform of the modulated signal shows narrow peaks at

the carrier frequency; they aren't quite as narrow as in the AM cases, but since $\beta_p = k_p = 0.1$ is small, they are fairly narrow. The demodulated output looks as expected.

4.2.4 FM

Much of the same arguments as PM apply. The main difference is that the Fourier transform of the modulated signal looks messier; this is because $\beta_p = 10$ is larger, so more bandwidth is used in this case.

4.3 Verifying Carson's rule for angle modulation

To do this, I simply overlaid a plot of the Carson bandwidth (shaded in green) on top of the plots of the modulated PM and FM signals in the frequency domain in Figure 8. These were calculated using the Carson bandwidth formula:

$$W_{carson} = 2(\beta + 1)W$$

It is visually apparent that most of the power lies in these bounds; to further verify this, I used the following snippet to experimentally calculate the amount of power in both Carson bandwidths (this is for PM, the result for FM is analogous):

```

1 | bandpower(sig_pm_modulated, f_s_c_pm, ...
2 |     [f_c_pm - (beta_p + 1) * W], f_c_pm + (beta_p + 1) * W) ...
3 |     / bandpower(sig_pm_modulated)

```

As shown in Table 4, both calculations give approximately 100% of the power is in the Carson bandwidth.

4.4 Effect of varying noise variance on SNR

First, to review the SNR results from 2:

$$\begin{aligned} SNR_{conv} &= \frac{a^2 A_c^2}{2} \frac{P_m}{N_0 W} \\ SNR_{ssb} &= A_c^2 \frac{P_m}{N_0 W} \\ SNR_{pm} &= \frac{A_c^2}{2} \left(\frac{\beta_p}{\max |m(t)|} \right)^2 \frac{P_m}{N_0 W} \\ SNR_{fm} &= \frac{3A_c^2}{2} \left(\frac{\beta_f}{\max |m(t)|} \right)^2 \frac{P_m}{N_0 W} \end{aligned}$$

A few things to note:

- As shown in 10, the noise power $N_0 W$ in all of these analog domain equations should be replaced with $N_0 W/f_s$ for use in the digital (sampled) domain.
- In the case of a fixed message signal, P_m and W are fixed.

- a , A_c , β_p , β_f , and $(1/\sqrt{N_0} = \frac{1}{\sigma\sqrt{2}})$ are all related to the SNR by a quadratic relationship. Thus, changing any of these should have the same result on the SNR (albeit different results on power, bandwidth, clipping, etc.).

Three standard deviations were chosen (arbitrarily) for the noise process: 0.1, 0.5, and 1. The overall results for this section are summarized in Table 5. Plots for the modulation schemes can be found at Figures 9 (conventional AM), 10 (SSB AM), 11 (PM), 12 (FM).

It is visually evident from the plots that increasing the variance of the noise decreases the SNR, in all cases. For conventional AM, SSB AM, and PM, the experimental and the theoretical SNRs agreed fairly well.

Due to the nature of the parameter setup for conventional and SSB AM, which were set up so that the signal power in the modulated signal was the same, incidentally the calculated SNRs were also exactly the same. Experimentally, the SNRs between these two schemes turned out pretty close.

For the AM schemes, the theoretical SNR is systematically higher than the experimental SNR. Dan Kim suggested that dividing the theoretical SNR by a factor of 2 (providing a uniform ≈ -3 dB shift) seems to make the experimental and theoretical results significantly closer. While the error doesn't seem to be a simple constant shift, the fact that there's an approximately constant decibel shift means that either the experimental noise power is systematically high or the experimental signal power is systematically low. I was debugging by probing the noise and signal powers at several points through the demodulation function, and found that the signal power was always more or less correct, so I suspect that the noise power was incorrect. I'm not sure where this error comes from, or if it's a constant, but here are a few possibilities:

- inherent quantization error
- rectification noise (in the case of conventional AM)
- half-wave rectification, which cuts the signal power in half (in the case of conventional AM)
- the LPF or HPF may cause the retained noise power to be disproportionately higher (w.r.t. signal power) than before filtering

The FM scheme experimental SNR was far from the theoretical value. Theoretically, the modulation index is large and the SNR should be very high. My guess is that there are many sources of numerical and other inherent error, e.g.:

- inherent quantization error
- numerical integration using `cumsum` during modulation
- numerical differentiation using `diff` during demodulation
- half-wave rectification noise during demodulation
- low-pass and high-pass filtering may cause distortions in the ratio of retained noise power to signal power

4.5 Effect of varying modulation index on SNR

The same notes from the previous section apply: decreasing σ by a factor of 2 or increasing modulation factor by a factor of 2 (ignoring clipping or other distortion) should cause the same change in SNR. In our specific case, having $\sigma = 0.5$, $a = 0.5$ in the conventional AM case would have the same SNR as $\sigma = 1$, $a = 1$. Indeed, the theoretical calculation produces the same number: 15.00 (with the 2 correction factor).

The overall results for this section are summarized in Table 6. Plots for the modulation schemes can be found at Figures 13 (conventional AM), 14 (PM), and 15 (FM).

Much of the analysis is exactly the same as in the previous section: the conventional AM, SSB AM, and PM had good agreement between experimental and theoretical SNRs. There was a similar systematic offset between the theoretical and experimental values that could be improved by adding the factor of 2.

4.6 Effect of higher carrier and sampling frequencies

I originally had a carrier frequency of 1MHz and a sampling frequency of 10MHz, but this caused a lot of distortion, which can be seen in Figure 16. It seems that higher frequencies can cause more errors with quantization errors than it might help by improving resolution. However, for the other modulation methods, I didn't have this problem. Therefore, my FM carrier and sampling frequencies are fairly low.

5 Conclusions

A reasonable implementation of multiple analog modulation schemes – conventional AM, SSB AM, FM, and PM – were implemented and simulated in MATLAB. Their operations were verified by a number of methods, including listening to the demodulated signal by ear, examining the modulated signal, checking the power of the modulated and demodulated signals, and comparing the demodulated signal to the original message signal.

These modulation and demodulation functions were used to empirically verify some of the theory. For example, the Carson bandwidth was loosely verified by a visual check and an estimated calculation. The effect of varying the variance of the AWGN noise, and of varying the modulation index, on the SNR was also explored and compared to theoretical values. Most of the calculated SNR values were fairly close to the theoretical values (i.e., most of the errors were less than 10dB); however, the empirical SNRs were systematically lower than the theoretical values. Most of the values were fairly close, but unfortunately I was unable to find a way to make the experimental SNR values for FM match the theoretical values.

These errors may be due to some combination of implementation errors, quantization/rounding errors, and inherent noise in the systems (especially in filtering, rectification, and numerical integration/differentiation stages). More time and effort would be needed to investigate these errors. Despite these errors, the general trends in SNR as a function of the changes in input parameters were correct.

6 Acknowledgments

I would like to acknowledge Prof. Brian Frost for answering my (and my peers') numerous late-night questions, hosting extra office hours, and extending the deadline in order to accommodate my peers and I for this project.

I would also like to acknowledge Anthony Belladonna, Dan Kim, Joshua Yoon, Thodoris Kapouranis, Paul Cucchiara, Philip Blumin, Derek Lee, Steven Lee, and other members of a particular Discord channel for discussing many aspects of the project and staying up with me multiple late nights to work on this project.

I would like to acknowledge Dan Kim for giving the suggestion that multiplying by two in the denominator of the conventional AM SNR equation, which gives closer estimates to the experimental values.

I would like to acknowledge Derek Lee for the following illustrious illustration:

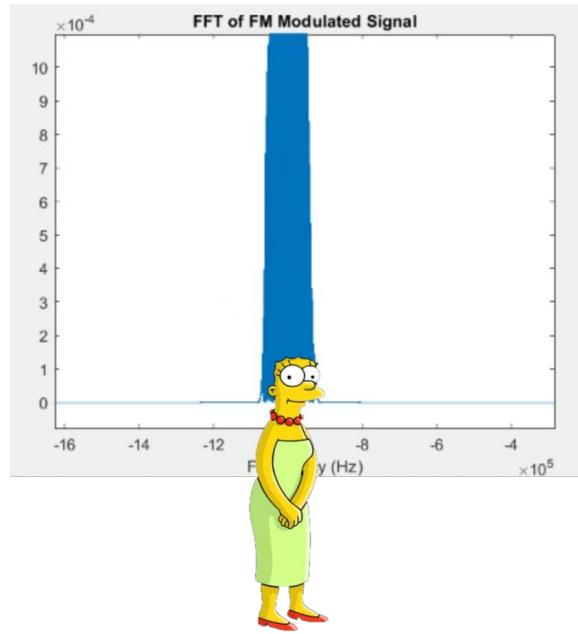


Figure 17: Untitled. Commissioned by Anthony Belladonna.

7 References

- [1] "Radio Broadcast Signals." <http://hyperphysics.phy-astr.gsu.edu/hbase/Audio/radio.html>.
- [2] Proakis, J. G., Salehi, M. (2001). *Communication Systems Engineering*. Upper Saddle River, NJ, USA: Prentice-Hall. ISBN: 0130617938
- [3] "Sample Audio Files." <https://www.opdsupport.com/downloads/miscellaneous/sample-audio-files>.

8 Appendix I: Audio

The original audio message signal used throughout this report is a speech sample downloaded from Olympus Professional Dictation Support [3]. The audio is truncated to reduce memory and processor strain. The message is also normalized so that its maximum amplitude is 1, which makes calculating modulation indices simpler.

Sampling rate	16kHz
Bandwidth	6.61kHz
Samples	20000 (after truncation)
Duration	1.25s (after truncation)
Signal power	0.0522
$\max m(t) $	1

Table 7: Audio message signal statistics

```
1 %% Load audio and preprocess
2
3 % audio details
4 audio_filename = 'welcome.wav';
5
6 % truncate to limit memory usage and processing time
7 audio_samples = 20000;
8
9 % loads the message and the sampling rate of the message
10 [sig_m, f_s_m] = audioread(audio_filename);
11 sig_m = sig_m(1:audio_samples)';
12
13 % make maximum magnitude 1 to avoid problems with conventional AM
14 sig_m = sig_m / max(abs(sig_m));
15
16 % get bandwidth
17 W = obw(sig_m, f_s_m);
18
19 % get signal power
20 P_m = bandpower(sig_m);
```

Listing 9: Loading and preprocessing the audio signal

9 Appendix II: Auxiliary functions

9.1 Estimating signal power in MATLAB

In MATLAB, we only have discrete-time signals, with N discrete samples, sampling frequency F_s , and duration $T = N/F_s$. We can estimate the power of a signal numerically using the root-mean-squared function (squared) `rms` or `bandpower` functions, where `rms(signal)^2=bandpower(signal)`. To show the equivalence of this mean-squared function to the power of a discrete signal:

$$\begin{aligned}\text{rms}^2\{x[n]\} &= \frac{1}{N} \sum x^2[n] = \frac{1}{TF_s} \sum x^2[n] = \frac{1}{T} \sum x^2[n] \frac{1}{F_s} \\ &= \frac{1}{T} \sum x^2[n] \delta t \approx \frac{1}{T} \int x^2(t) dt = P_x\end{aligned}\quad (39)$$

Thus, the `bandpower` function was used throughout the program to estimate signal power.

9.2 Hilbert transform

The Hilbert transform is the LTI, norm-preserving magic sauce behind SSB. In layman's terms, this turns the negative part of the baseband signal into the quadrature part of the signal, which allows for packing the same signal as DSB AM into half of the bandwidth.

```
1 % hilbert transform of a signal; copied from earlier psets; for SSB AM
2 % params:
3 % sig_x = input signal
4 % returns:
5 % res    = hilbert transform of sig_x
6 function res = hilbert_transform(sig_x)
7     ft_x = fft(sig_x);
8     len_x = length(sig_x);
9
10    % get signum of frequency; +1 for 0 to pi, -1 for -pi to 0
11    sgn = [ones(1, floor(len_x/2)), -1*ones(1, ceil(len_x/2))];
12    res = ifft(-1j * sgn .* ft_x);
13 end
```

Listing 10: Hilbert transform function

9.3 Lowpass signal

This function is used to lowpass signals by multiplying by the frequency response of a first-order LPF. Used for filtering out noise above the audible range, but can be used with any cutoff frequency.

```

1 % lowpass: set tau to upper range of audible range
2 function sig_c = lowpass_audible(sig, f_s, tau)
3     wd = linspace(-pi, pi, length(sig));
4     f = wd * f_s / (2 * pi);
5     ft = fftshift(fft(sig));
6
7     % first-order LPF with cutoff frequency at tau
8     lpf = (1 + f/tau*1j).^-1;
9     ft = lpf .* ft;
10
11    sig_c = real(ifft(ifftshift(ft)));
12 end

```

Listing 11: Lowpass function

9.4 Fourier transform plotter

This function provides the frequency axis and Fourier transform of a signal.

```

1 % helper function to plot the fourier transform of a signal
2 % params:
3 % x      = signal
4 % f_s    = sampling rate
5 % returns:
6 % f      = frequency axis
7 % ft     = fourier transform
8 function [f, ft] = plot_ft(sig, f_s)
9     wd = linspace(-pi, pi, length(sig));
10    f = wd * f_s / (2 * pi);
11    ft = abs(fftshift(fft(sig)) / f_s);
12 end

```

Listing 12: Fourier transform plotting function

9.5 Saving figures to PDFs

This function saves figures to PDFs, with arbitrary page dimensions.

```

1 % helper function for printing to PDFs
2 function fig_save(fig_title, paper_size)
3     global pset_name;
4     pset_name = 'ece300_proj1';
5
6     set(gcf(), ...
7         'Units', 'centimeters', ...
8         'Position', [0 0 paper_size], ...
9         'InnerPosition', [0 0 paper_size], ...
10        'OuterPosition', [0 0 paper_size], ...
11        'PaperPositionMode', 'auto');
12    exportgraphics(gcf(), ...
13                  sprintf('%s_fig_%s.png', pset_name, fig_title), ...
14                  'ContentType', 'image');
15 end

```

Listing 13: Figure saver helper function

10 Appendix III: Calculating noise power in the digital domain

An AWGN random process $N(t)$ with variance σ^2 has the following autocorrelation function:

$$R_{N,dig}[n] = \sigma^2 \delta[n] \quad (40)$$

By the Weiner-Khinchin theorem, the PSD (in digital radians) of the AWGN noise is given by the DFT of the autocorrelation function:

$$S_{N,dig}(\omega) = \mathcal{F}\{R_N[n]\}(\omega) = \frac{\sigma^2}{2\pi} \quad (41)$$

The total power of the noise process before filtering is the integral of the PSD over its domain:

$$P_{N,dig} = \int_{-\pi}^{\pi} \frac{\sigma^2}{2\pi} d\omega = \sigma^2 \quad (42)$$

This can be confirmed empirically by finding the bandpower of an arbitrary AWGN noise signal. E.g., when $\sigma^2 = 4$ `bandpower(2 * randn(1, 1000000))` outputs 4. When we want to find the power of the noise in the baseband (demodulated) signal, then we assume a perfect filter with digital cutoff frequency ω_{cut} (i.e., this would be set to the bandwidth of the message signal). Then we have a rectangular pulse PSD centered at $\omega = 0$, with height $\sigma^2/(2\pi)$ and width $2\omega_{cut}$. The noise power at baseband is the integral (area) under this rectangle, i.e.:

$$P_{N,bb,dig} = \int_{-\pi}^{\pi} (\text{rect}_{\{-\omega_{cut}, \omega_{cut}\}})(S_N(\omega)) d\omega = \frac{\sigma^2}{2\pi} (2\omega_{cut}) \quad (43)$$

Making the conventional substitution for the noise power, $N_0 = 2\sigma^2$:

$$P_{N,bb,dig} = \frac{N_0}{2\pi} \omega_{cut} = \frac{N_0}{2\pi} \frac{2\pi f_{cut}}{f_s} = N_0 \frac{f_{cut}}{f_s} \quad (44)$$

where f_s is the sampling frequency. The final expression here is the most intuitive: it is the product of the noise power with the ratio of the cutoff frequency to the sampling frequency. This contrasts with the analog case, which was covered in class:

$$R_{N,ana}(t) = \sigma^2 \delta(n) \quad (45)$$

$$S_{N,ana}(f) = \sigma^2 \quad (46)$$

$$P_{N,ana} = \int_{-\infty}^{\infty} S_{N,ana}(f) df = \infty \quad (47)$$

$$P_{N,bb,ana} = \int_{-f_{cut}}^{f_{cut}} S_{N,ana}(f) df = N_0 f_{cut} \quad (48)$$

(If analog angular frequency was used instead of analog linear frequency, then there would also be the additional factor of $1/(2\pi)$.) Both cases involve clipping a flat noise PSD to that within the bandwidth specified by the cutoff frequency.

In summary: a noise power (i.e., in SNR equations) of $N_0 f_{cut}$ (or, equivalently, $N_0 W$) in the analog domain should be replaced with $N_0 f_{cut}/f_s = N_0 \omega_{cut}/(2\pi)$.

Note that most of the demodulation methods involve a LPF at the upsampled rate (i.e., the sampling rate used to sample the modulated signal), and the signal is passed through an additional LPF at the downsampled rate (i.e., the sampling rate used for the baseband audio) to filter out inaudible high-frequency noise. Note that downsampling changes the magnitude of the PSD.

11 Appendix IV: Source code

The core functions have already been defined. Here is the rest of the code used to produce the plots and results in this document.

```

1 close all; clear; clc;
2 set(0, 'defaultTextInterpreter', 'latex');
3
4 % Project details
5 global pset_name;
6 pset_name = 'ece300_proj1';
7
8 % audio details
9 audio_filename = 'welcome.wav';
10 audio_samples = 20000;
11
12 % Load audio
13
14 % loads y, Fs
15 [sig_m, f_s_m] = audioread(audio_filename);
16 sig_m = sig_m(1:audio_samples)';
17
18 % make maximum magnitude 1 to avoid problems with conventional AM
19 sig_m = sig_m / max(abs(sig_m));
20
21 % plot audio
22 figure('visible', 'off');
23 tiledlayout(1, 2, 'TileSpacing', 'Compact', 'Padding', 'Compact');
24
25 duration = length(sig_m) / f_s_m;
26 t_m = linspace(0, duration, length(sig_m));
27
28 nexttile();
29 plot(t_m, sig_m);
30 title('Original signal');
31 ylabel('$\mathbf{S}(t)$');
32 xlabel('$\mathbf{t}$');
33 xlim([0, duration]);
34 ylim([-1.2, 1.2]);
35
36 [f, ft] = plot_ft(sig_m, f_s_m);
37 nexttile();
38 semilogy(f, ft);
39 title('Magnitude of original signal Fourier transform');
40 ylabel('$|\mathbf{S}(f)|$');
41 xlabel('$\mathbf{f}$');
42 ylim([10e-8, 10e-2]);
43
44 fig_save('sig_m', [80 20]);
45
46 % get bandwidth
47 W = obw(sig_m, f_s_m);
48
49 % get signal power
50 P_m = bandpower(sig_m);
51
52 % write original signal to file
53 audiowrite('original.wav', sig_m, f_s_m);
54
55 % Modulation configuration
56
57 % conventional AM
58 a_conv = 0.5;
59 A_c_conv = 4;
60 f_c_conv = 500000;
61 f_s_c_conv = 2000000;
62
63 % SSB AM
64 f_c_ss = 500000;
65 A_c_ss = A_c_conv / sqrt(8);
66 f_s_c_ss = 2000000;
67 is_ussb = false;
68
69 % PM
70 f_c_pm = 500000;
71
72 A_c_pm = A_c_conv;
73 f_s_c_pm = 5000000;
74 k_p = 0.1;
75
76 % FM
77 f_c_fm = 1e6; %4000000;
78 A_c_fm = A_c_conv;
79 f_s_c_fm = 1e7; %5000000;
80 k_f = 10 * W;
81
82 % for when viewing the effect of varying noise std.
83 noise_stds = [0 0.1 0.5 1];
84
85 % for when viewing the effect of varying modulation index
86 noise_std_conv = 1;
87 noise_std_fm = 0.5;
88 noise_std_pm = 1;
89
90 % for storing the theoretical snrs
91 var_noise_theo = zeros(4,4);
92 var_noise_emp = zeros(4,4);
93 var_mod_ind_theo = zeros(3,3);
94 var_mod_ind_emp = zeros(3, 3);
95
96 % conventional AM modulation
97 sig_conv_modded = conv_mod(f_c_conv, A_c_conv, a_conv, ...
98 sig_m, f_s_m, f_s_c_conv);
99 sig_conv_demodded = conv_demod(sig_conv_modded, A_c_conv, ...
100 a_conv, f_s_m, f_s_c_conv, W);
101 plot_modded_demodded(sig_conv_modded, sig_conv_demodded, ...
102 f_s_m, f_s_c_conv, 'conv');
103 audiowrite('conv_demodded.wav', sig_conv_demodded, f_s_m);
104
105 % SSB AM modulation
106 sig_ss_modded = ssb_mod(f_c_ss, A_c_ss, sig_m, f_s_m, f_s_c_ss, ...
107 is_ussb);
108 sig_ss_demodded = ssb_demod(sig_ss_modded, f_c_ss, A_c_ss, ...
109 f_s_m, f_s_c_ss, W);
110 plot_modded_demodded(sig_ss_modded, sig_ss_demodded, f_s_m, ...
111 f_s_c_ss, 'ssb');
112 audiowrite('ssb_demodded.wav', sig_ss_demodded, f_s_m);
113
114 % PM Modulation
115 sig_pm_modded = pm_mod(f_c_pm, A_c_pm, sig_m, k_p, f_s_m, f_s_c_pm, ...
116 f_s_m, f_s_c_pm, W);
117 plot_modded_demodded(sig_pm_modded, sig_pm_demodded, f_s_m, ...
118 f_s_c_pm, 'pm');
119 audiowrite('sig_pm_demodded.wav', sig_pm_demodded, f_s_m);
120
121 % FM Modulation
122 sig_fm_modded = fm_mod(f_c_fm, A_c_fm, sig_m, k_f, f_s_m, f_s_c_fm);
123 sig_fm_demodded = fm_demod(sig_fm_modded, A_c_fm, f_s_m, f_s_c_fm, ...
124 k_f, W);
125 plot_modded_demodded(sig_fm_modded, sig_fm_demodded, f_s_m, ...
126 f_s_c_fm, 'fm_problematic');
127 audiowrite('sig_fm_demodded.wav', sig_fm_demodded, f_s_m);
128
129 % Conventional AM with noise
130 demod_fn = @(sig_mod) conv_demod(sig_mod, A_c_conv, a_conv, f_s_m, ...
131 f_s_c_conv, W);
132 [sig_conv_modded, sig_conv_demodded, snrs] = apply_noise(noise_stds, ...
133 demod_fn, sig_conv_modded, sig_conv_demodded, f_s_m, ...
134 f_s_c_conv, W, 'conv');
135 var_noise_emp(1,:) = snrs;
136
137 % calculate conventional AM theoretical SNR
138 var_noise_theo(1,:) = 10 * log10(a_conv^2 * A_c_conv^2 * ...
139 bandpower(sig_m) * (2 * (2 * noise_stds.^2) * W / f_s_c_conv).^(-1));
140

```

```

141 %% SSB AM with noise
142 demod_fn = @(sig_mod) ssb_demod(sig_mod, f_c_ssbb, A_c_ssbb, f_s_m, ...
143     f_s_c_ssbb, W);
144 [sigs_ssbb_modded, sigs_ssbb_demodded, snrs] = apply_noise(noise_stds, ...
145     demod_fn, sig_ssbb_modded, sig_ssbb_demodded, f_s_m, f_s_c_ssbb, ...
146     W, 'ssb');
147 var_noise_emp(2,:) = snrs;
148
149 %% calculate SSB AM theoretical SNR
150 var_noise_theo(2,:) = 10 * log10(A_c_ssbb^2 * bandpower(sig_m) * ...
151     ((2 * noise_stds.^2) * W / f_s_c_ssbb).^.-1);
152
153 %% PM with noise
154 demod_fn = @(sig_mod) pm_demod(f_c_pm, A_c_pm, sig_mod, k_p, f_s_m, ...
155     f_s_c_pm, W);
156 [sigs_pm_modded, sigs_pm_demodded, snrs] = apply_noise(noise_stds, ...
157     demod_fn, sig_pm_modded, sig_pm_demodded, f_s_m, f_s_c_pm, W, 'pm');
158 var_noise_emp(3,:) = snrs;
159
160 %% calculate PM theoretical SNR
161 var_noise_theo(3,:) = 10 * log10(k_p.^2 * A_c_pm.^2 * bandpower(sig_m) * ...
162     (2 * (2 * noise_stds.^2) * W / f_s_c_pm).^.-1);
163
164 %% FM with noise
165 demod_fn = @(sig_mod) fm_demod(sig_mod, A_c_fm, f_s_m, f_s_c_fm, ...
166     k_f, W);
167 [sigs_fm_modded, sigs_fm_demodded, snrs] = apply_noise(noise_stds, ...
168     demod_fn, sig_fm_modded, sig_fm_demodded, f_s_m, f_s_c_fm, W, 'fm');
169 var_noise_emp(4,:) = snrs;
170
171 %% calculate FM theoretical SNR
172 beta_f = k_f / W;
173 var_noise_theo(4,:) = 10 * log10(3 * beta_f.^2 * A_c_fm.^2 * ...
174     bandpower(sig_m) * (2 * (2 * noise_stds.^2) * W / f_s_c_fm).^.-1);
175
176 %% Changing modulation index of Conventional AM
177 mod_fn = @(sig_m, a) conv_mod(f_c_conv, A_c_conv, a, sig_m, f_s_m, ...
178     f_s_c_conv);
179 demod_fn = @(sig_mod, a) conv_demod(sig_mod, A_c_conv, a, f_s_m, ...
180     f_s_c_conv, W);
181 var_mod_ind_emp(1,:) = apply_modulation_factor(...
182     a.conv, noise_std_conv, mod_fn, demod_fn, sig_m, ...
183     sigs_conv_modded(4,:), sigs_conv_demodded(4,:), f_s_m, ...
184     f_s_c_conv, W, sig_conv_demodded, 'a', 'conv');
185
186 %% calculate conventional AM theoretical SNR
187 var_mod_ind_theo(1,:) = 10 * log10([a_conv/2 a_conv a_conv*2].^2 * ...
188     A_c_conv.^2 * bandpower(sig_m) * (2 * (2 * noise_std_conv) * W / ...
189     f_s_c_conv).^.-1);
190
191 %% Changing the modulation index of PM
192 mod_fn = @(sig_m, k_p) pm_mod(f_c_pm, A_c_pm, sig_m, k_p, f_s_m, f_s_c_pm);
193 demod_fn = @(sig_mod, k_p) pm_demod(f_c_pm, A_c_pm, sig_mod, k_p, ...
194     f_s_m, f_s_c_pm, W);
195 var_mod_ind_emp(2,:) = apply_modulation_factor(...
196     k_p, noise_std_pm, mod_fn, demod_fn, sig_m, sigs_pm_modded(4,:), ...
197     sigs_pm_demodded(4,:), f_s_m, f_s_c_pm, W, sig_pm_demodded, ...
198     'k_p', 'pm');
199
200 %% calculate PM theoretical SNR
201 var_mod_ind_theo(2,:) = 10 * log10([k_p/2 k_p k_p*2].^2 * A_c_pm.^2 * ...
202     bandpower(sig_m) * (2 * (2 * noise_std_pm.^2) * W / f_s_c_pm).^.-1);
203
204 %% Changing the modulation index of FM
205 mod_fn = @(sig_m, k_f) fm_mod(f_c_fm, A_c_fm, sig_m, k_f, ...
206     f_s_m, f_s_c_fm);
207 demod_fn = @(sig_mod, k_f) fm_demod(sig_mod, A_c_fm, f_s_m, ...
208     f_s_c_fm, k_f, W);
209 var_mod_ind_emp(3,:) = apply_modulation_factor(...
210     k_f, noise_std_fm, mod_fn, demod_fn, sig_m, sigs_fm_modded(3,:), ...
211     sigs_fm_demodded(3,:), f_s_m, f_s_c_fm, W, sig_fm_demodded, ...
212     'k_f', 'fm');
213
214 %% calculate FM theoretical SNR
215 beta_f = [k_f/2 k_f k_f*2] / W;
216 var_mod_ind_theo(3,:) = 10 * log10(3 * beta_f.^2 * A_c_fm.^2 * ...
217     bandpower(sig_m) * (2 * (2 * noise_std_fm.^2) * W / f_s_c_fm).^.-1);
218
219 %% Print out results
220 fprintf('Theoretical SNRs from varying noise variance');
221 var_noise_theo(:,2:4)
222 fprintf('Empirical SNRs from varying noise variance');
223 var_noise_emp(:,2:4)
224 fprintf('Theoretical SNRs from varying modulation indices');
225 var_mod_ind_theo
226 fprintf('Empirical SNRs from varying modulation indices');
227 var_mod_ind_emp
228
229 %% plot modulation modded and demodded signals and their FTs
230 %% params:
231 %% sig_modded = modulated signal
232 %% sig_demodded = demodulated signal
233
234 %% f_s_m = baseband sampling frequency
235 %% f_s_c = modulated signal sampling frequency
236 %% fname = name of figure
237 function plot_modded_demodded(sig_modded, sig_demodded, f_s_m, f_s_c, ...
238     fname)
239
240 figure('visible', 'off');
241 tiledlayout(2, 2, 'TileSpacing', 'Compact', 'Padding', 'Compact');
242
243 duration = length(sig_modded) / f_s_c;
244 t_c = linspace(0, duration, length(sig_modded));
245 t_m = linspace(0, duration, length(sig_demodded));
246
247 %% plot modded signal in time domain
248 nexttile();
249 plot(t_c, sig_modded);
250 title('Modulated signal');
251 ylabel('$\$u(t)\$');
252 xlabel('$t\$');
253 xlim([0 duration]);
254
255 %% plot modded signal in frequency domain
256 nexttile();
257 [f, ft] = plot_ft(sig_modded, f_s_c);
258 semilogy(f, ft);
259 title('Modulated signal Fourier transform magnitude');
260 ylabel('$|U(f)|$');
261 xlabel('$f\$');
262
263 %% plot demodded signal in time domain
264 nexttile();
265 plot(t_m, sig_demodded);
266 title('Demodulated signal');
267 ylabel('$\$r(t)\$');
268 xlabel('$t\$');
269 ylim([-1.2 1.2]);
270 xlim([0 duration]);
271
272 %% plot demodded signal in frequency domain
273 nexttile();
274 [f, ft] = plot_ft(sig_demodded, f_s_m);
275 semilogy(f, ft);
276 title('Demodulated signal Fourier transform magnitude');
277 ylabel('$|R(f)|$');
278 xlabel('$f\$');
279 ylim([10e-8, 10e-2]);
280
281 fig_save(fname, [40 20]);
282 end
283
284 %% apply noise to signals modulated with different modulation indices
285 %% params:
286 %% mod_index = modulation index values
287 %% noise_std = standard deviation of noise to add
288 %% mod_fn = lambda to modulate a signal
289 %% demod_fn = lambda to demod a noisy modulated signal
290 %% sig_m = message signal
291 %% sig_modded = reference modded signal
292 %% sig_demodded = reference demodded signal
293 %% f_s_m = baseband sampling frequency
294 %% f_s_c = modulated signal sampling frequency
295 %% tau = post-demodulation LPF cutoff frequency
296 %% sig_demodded_no_noise = reference no-noise demodulated signal
297 %% variant = name of modulation index
298 %% mod_type = modulation scheme
299 %% returns:
300 %% snrs = SNRs of noisy modulated signals
301 function snrs = apply_modulation_factor(...
302     mod_index, noise_std, mod_fn, demod_fn, sig_m, sig_modded, ...
303     sig_demodded, f_s_m, f_s_c, tau, sig_demodded_no_noise, variant, ...
304     mod_type)
305
306     sig_modded = zeros(3, length(sig_modded));
307     sigs_modded = zeros(3, length(sig_modded));
308
309 %% half the modulation index
310 sigs_modded(1,:) = mod_fn(sig_m, mod_index/2) + ...
311     noise_std * randn(1, length(sig_modded));
312 sigs_modded(1,:) = lowpass_audible(mod_fn(sigs_modded(1,:), ...
313     mod_index/2), f_s_m, tau);
314
315 %% original modulation index
316 sigs_modded(2,:) = sig_modded;
317 sigs_modded(2,:) = sig_demodded;
318
319 %% double the modulation index
320 sigs_modded(3,:) = mod_fn(sig_m, mod_index*2) + ...
321     noise_std * randn(1, length(sig_modded));
322 sigs_modded(3,:) = lowpass_audible(mod_fn(sigs_modded(3,:), ...
323     mod_index*2), f_s_m, tau);
324
325 %% plot

```

```

325 snrs = plot_with_noise([mod_index/2, mod_index, mod_index*2], ...      386 % demodded signal for a series of related signals with one varying
326     sigs_modded, sigs_demodded, f_s_m, f_s_c, ...                      387 % variable; also calculates SNRs and returns them
327     sig_demodded_no_noise, variant, mod_type);                           388 %
328 end                                                               389 %
329 % apply different noise to a modded signal                                390 %
330 % params:                                                               391 %
331 % noise_stds = list of standard deviations of the noise                392 %
332 % demod_fn = lambda to demod a noisy signal                            393 %
333 % sig_modded_no_noise = no-noise modded signal reference               394 %
334 % sig_demodded_no_noise = no-noise demodded signal reference            395 %
335 % f_s_m = baseband sampling frequency                                     396 %
336 % f_s_c = modulated signal sampling frequency                          397 %
337 % tau = cutoff frequency                                                 398 %
338 % mod_type = modulation scheme                                         399 %
339 % returns:                                                               400 %
340 % sigs_modded = modulated signals with applied noise                   401 %
341 % sigs_demodded = demodulated noisy modded signals                     402 %
342 % snrs = snrs for demodded signals                                       403 %
343 % snrs = zeros(length(noise_stds), ...                                     404 %
344 function [sigs_modded, sigs_demodded, snrs] = apply_noise(noise_stds, ... 405 %
345     demod_fn, sig_modded_no_noise, sig_demodded_no_noise, f_s_m, f_s_c, ... 406 %
346     tau, mod_type)                                                       407 %
347     sigs_modded = zeros(length(noise_stds), ...                           408 %
348         length(sig_modded_no_noise));                                      409 %
349     sigs_modded = zeros(length(noise_stds), ...                           410 %
350         length(sig_demodded_no_noise));                                     411 %
351 % first element will be the no-noise variant                           412 %
352     sigs_modded(1,:) = sig_modded_no_noise;                             413 %
353     sigs_demodded(1,:) = sig_demodded_no_noise;                         414 %
354 % loop through each noise process                                     415 %
355 for i = 2:length(noise_stds)                                           416 %
356     % modulate, add noise                                              417 %
357     % modulate, add noise                                              418 %
358     % modulate, add noise                                              419 %
359     % modulate, add noise                                              420 %
360     % modulate, add noise                                              421 %
361     % modulate, add noise                                              422 %
362     % demod                                              423 %
363     % demod                                              424 %
364     % demod                                              425 %
365     % demod                                              426 %
366 end                                                               427 %
367 % plot                                                               428 %
368 snrs = plot_with_noise(noise_stds, sigs_modded, sigs_demodded, ...       429 %
369     f_s_m, f_s_c, sig_demodded_no_noise, '\sigma_n', mod_type);           430 %
370 end                                                               431 %
371 % approximates SNR in dB, given the pristine demodded signal and a noisy 432 %
372 % variant; approximates noise as the difference between the two          433 %
373 % params:                                                               434 %
374 % sig_no_noise = demodded signal with no noise                          435 %
375 % sig_noisy = noisy signal to calculate the SNR of                      436 %
376 % returns:                                                               437 %
377 % res = SNR (in dB)                                                 438 %
378 function res = snr(sig_no_noise, sig_noisy)                            439 %
379     res = 10 * log10(bandpower(sig_no_noise, sig_noisy) / ...           440 %
380                     bandpower(sig_no_noise - sig_noisy));                  441 %
381 end                                                               442 %
382 % plots the demodded signal, spectrum of modded signal, and spectrum of 443 %
383 % plots the demodded signal, spectrum of modded signal, and spectrum of 444 %
384 % plots the demodded signal, spectrum of modded signal, and spectrum of 445 %
385 % plots the demodded signal, spectrum of modded signal, and spectrum of 446 | end

```

ECE300 – Project 2

Brian DeHority, Jonathan Lam, Danny Hong

November 16, 2020

1 Introduction

Digital modulation has become the predominant form of electronic communication, supplanting the ubiquity of non-digital (analog-only) modulation. In contrast to analog modulation, the data sent through digital modulation is sent and received in discretized symbols rather than a continuous stream of information. This project explores several digital modulation systems – binary antipodal pulse amplitude modulation (PAM), binary orthogonal PAM, quadrature amplitude modulation (QAM), phase shift keying (PSK), and differential phase shift keying (DPSK) – and evaluate their relative strengths.

The modulation, transmission (with channel-caused additive white gaussian noise (AWGN)), and demodulation are modeled in MATLAB. Normally, these symbols would be the modulation of a carrier signal with an alphabet of discrete signals, but we only deal with the (discrete) constellation representation of the symbols in the alphabet. These simulations are limited to two-dimensional constellations (defined by the in-phase and quadrature components of the modulated carrier signal). The receiver must use the received, AWGN-distorted signal to decide the most likely transmitted symbol. The decision rule for this project follows a least-squares (LS) decision rule, in which the decision minimizes the (Euclidean) distance error in the constellation space. With higher noise power or closer points on the constellation, the probability of error (intuitively) increases.

2 Review of Digital Modulation Schemes

2.1 Binary Antipodal PAM

The constellation of the binary antipodal scheme are two points equidistant and opposite one another from the origin. (Alternatively, it can be thought of as the special case of PSK, where $M = 2$.) This is a simple binary scheme and maximizes the distance between the two constellation points given the symbol energies. Like binary orthogonal and PSK schemes, the symbols are also equal-energy. The analog representation of this is that the two symbols would be negatives of one another, i.e.:

$$u_m(t) = \pm p(t), \quad m \in \{0, 1\}$$

where u_m denotes the analog symbol, and p denotes the unit-energy pulse. The bit error rate (BER) of this scheme is given by:

$$\text{BER} = Q\left(\sqrt{\frac{2\mathcal{E}_s}{N_0}}\right)$$

where \mathcal{E}_s is the energy for symbol (in non-equienergy schemes like generic QAM, this represents the average *average* per symbol) and N_0 is the noise power. We defer the derivation of this formula (and all of the other theoretical calculations of the BER) to the lecture notes or to the textbook.

2.2 Binary Orthogonal PAM

Binary orthogonal has a constellation where the two symbol points are equal energy and at a $\pi/2$ phase shift w.r.t. the origin. This can be thought of as half of a PSK scheme, where $M = 4$. While this is not as robust to noise as binary orthogonal PAM, it is also simple to implement (in-phase (e.g., cos) and quadrature (e.g., sin) components are orthogonal by definition), and it generalizes easily to higher-dimensional spaces. The analog symbols might look like:

$$u_m(t) = p(t) \cos(\omega_c t + \phi_m), \quad m \in \{0, 1\}, \quad \phi_m \in \{\phi_0, \phi_0 + \pi/2\}$$

The BER of this scheme is:

$$\text{BER} = Q\left(\sqrt{\frac{\mathcal{E}_s}{N_0}}\right)$$

Clearly, the difference between this and the binary antipodal scheme is a factor of 2 within the square root in the Q function. This can be seen intuitively with a geometric view: the effective noise power is increased by a factor of $\sqrt{2}$ due to the fact that the distance between the two points (assuming the same symbol energy for both schemes) is reduced by this factor.

2.3 (Phase-Coherent) PSK

PSK constellation points are placed in a circle around the origin. This has two main benefits: all symbols have equal energy, and only the angle (phase) of the received signal is required to make a decision (amplitude is irrelevant in making the decision). In addition, when using greycoding, the most likely error will only be one bit.

$$u_m(t) = p(t) \cos\left(\omega_c t + \frac{2\pi}{M}m\right), \quad m \in \{0, 1, 2, \dots, M-1\}$$

To maximize spacing between points (thus minimizing the probability of error), the symbols are equally distributed throughout the circle. The drawback of PSK is that it requires phase coherence; one method to resolve this, as we saw in analog modulation schemes, is the use of a pilot tone. DPSK is one solution to the phase coherence problem (without requiring a pilot tone).

When $M = 2$, the BER for this scheme is the same as that of binary antipodal. When $M > 2$, the BER for this scheme is given by:

$$\text{BER} = 2Q\left(\sqrt{\frac{2\mathcal{E}_s}{N_0}} \sin \frac{\pi}{M}\right)$$

2.4 DPSK

DPSK is similar to PSK, but rather than encoding the information within the angle of the symbol, ϕ_m , it is encoded in the difference in angles between subsequent transmissions ($\Delta\phi = \phi_m - \phi_{m-1}$). While removing the requirement for phase coherence, it transmits one fewer data point for every N symbol transmissions (a negligible loss when N is large). This scheme also roughly doubles the noise power (and thus making it less robust to noise than phase-coherent PSK), since there is variability in the phase of both of the symbols when taking the difference.

For $M = 2$, the BER for this scheme is given by:

$$\text{BER} = \frac{1}{2} \exp - \frac{\mathcal{E}_s}{N_0}$$

For $M > 2$, the BER for this scheme is given by:

$$\text{BER} = 2Q\left(\sqrt{\frac{\mathcal{E}_s}{N_0}} \sin \frac{\pi}{M}\right)$$

Similarly to binary antipodal vs. binary orthogonal schemes, the only difference between this and PSK is the factor of 2 in the sqrt in the Q-function, due to the above reasoning; for every piece of information (i.e., for every differential), there is variability in the phases of two points.

2.5 QAM

QAM encompasses all possible 2-D constellations, and thus all of the previous examples are specific instances of QAM. (However, binary orthogonal signals can easily be extended to higher-dimensional orthogonal constellations, but that is outside the scope of our 2-D constellation project.) In general, in a 2-D constellation space, two basis vectors \hat{i} and \hat{j} are required, and the constellation is of the form:

$$u_m(t) = a(t)\hat{i} + b(t)\hat{j}$$

Of course, since we love our trigonometric orthogonal basis functions, it usually comes in the form:

$$u_m(t) = A_m p(t) \cos(\omega_c t) - B_m p(t) \sin(\omega_c t)$$

The textbook provides equations for the BER when $M = 2^k$, and arranged either in a square (k is even) or a rectangle (k is odd), where the points are all equally-spaced in the quadrature and in-phase components. The benefit of this grid structure is that it requires lower average energy than schemes like PSK, and is simple to produce with the two basis signals.

For square constellations, the BER is (exactly):

$$\text{BER} = 1 - \left(1 - 2 \left(1 - \frac{1}{\sqrt{M}} \right) Q \left(\sqrt{\frac{3\mathcal{E}_s}{(M-1)N_0}} \right) \right)^2$$

When k is odd, the textbook provides a “tight upper bound” on the BER:

$$\text{BER} \leq 1 - \left(1 - 2Q \left(\sqrt{\frac{3\mathcal{E}_s}{(M-1)N_0}} \right) \right)^2$$

However, empirically we did not find that this was an upper bound, but rather lower than the empirical values in the $M = 32$ rectangular case. This is discussed further in the Discussion section.

3 Methods

3.1 Least squares decision rule

Our least squares decision rule finds the closest constellation point to a symbol (measured by Euclidean distance) using the `dsearchn` MATLAB function. (The `dsearchn` function uses a Delauney triangulation to efficiently find the nearest point.) This method is faster than using a `for` loop to iteratively find the closest point.

```
% l2_nearest: Finds nearest symbols in a constellation to a set of received
% (noisy) signals
%
% params:
% con    = complex constellation (M x 1)
% est    = complex received signals (N x 1)
%
% returns:
% N_hat = estimated signals
function nearest = l2_nearest(con, est)

    % transform complex numbers into a 2D real vectors
    con_2d = [real(con) imag(con)];
    est_2d = [real(est) imag(est)];

    % find nearest points (query points are the estimated vectors)
    nearest = con(dsearchn(con_2d, est_2d));
end
```

Figure 1: Least squares decision function

3.2 Transmission simulation

We simulate the transmission of a digital sequence by inputting the base constellation shape, a desired SNR per bit (in decibels), a desired noise power, and the sequence of symbols to transmit (taken from the base constellation). This function does not assume anything about the base constellation; there is no error checking. The base constellation and symbols to transmit are scaled so that the average SNR per bit is the desired value, given by the following equations:

$$\text{SNR per bit} = \frac{E_{avg,sym}}{N_0}$$
$$E_{avg,bit} = \frac{E_{avg,sym}}{\lceil \log_2 M \rceil}$$

The scaled transmitted symbols are now the true transmitted symbols (`true_sym`). Transmission through a noisy channel is approximated by adding 2-D AWGN with variance $\sigma^2 = N_0/2$; since the in-phase and quadrature components are independent additive gaussians, their variances sum;

and since we assume the noise to be isomorphic, each component has half of the variance of the overall AWGN ($\sigma_I^2 = \sigma_Q^2 = \sigma^2/2$). These noisy signals are the simulated received signal; they are then estimated to the nearest constellation points using the least squares decision rule to provide the estimated transmitted vectors (`est_sym`).

There a difference in procedure in simulating DPSK. Firstly, we do not assume phase coherence, so an arbitrary (constant) phase shift is applied to the transmitted vectors. (As shown in the results section, DPSK is not much affected by this, as expected.) This function assumes that the incoming constellation is a correct DPSK constellation (points lie evenly-spaced on a circle centered at the origin, with one of the constellation points lying on the ray $\theta = 0$). The true symbols and transmitted noisy symbols get transformed to their differentials by dividing their each point by the phase of the previous point (division by a complex number causes a phase shift). Only then do the transmitted noisy symbols get estimated to become the estimated transmitted vectors.

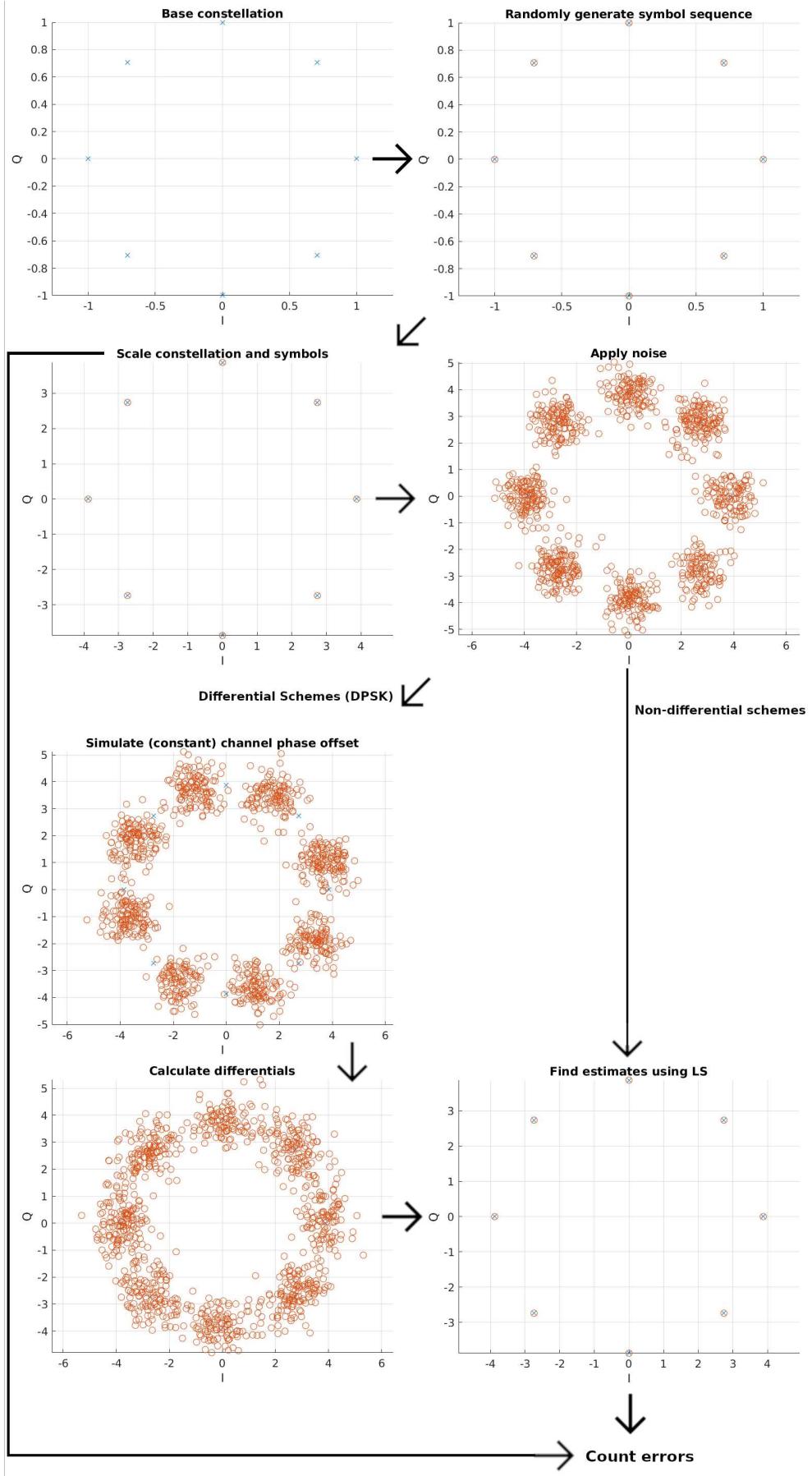


Figure 2: Illustration of the process of simulating and estimating the error for arbitrary constellations. Simulating the differential scheme takes additional work. $M = 8$, SNR=20dB, $N_0 = 1$.

```

% simulate_transmission: simulate transmitting symbols with noise, and
% returns the transmitted symbols and the estimates of the transmitted
% symbols after adding noise
%
% params:
% base_con = base constellation (M x 1)
% sym      = noise-free transmitted symbols (from base_cons, N x 1)
% NO       = noise power
% SNR      = desired SNR per bit (dB)
%
% returns:
% true_sym = transmitted (scaled) symbols
% est_sym   = estimated symbols
% scaling_factor = amount base constellation was scaled by
function [true_sym, est_sym, scaling_factor] ...
= simulate_transmission(base_con, sym, NO, SNR)

N = length(sym);
M = length(base_con);

% find desired average symbol energy
E_avg = mean(abs(base_con).^2);
E_bav = E_avg / ceil(log2(M));

% using SNR = E_av / NO
% Ebav = SNR * NO / log2(M)
E_bav_des = 10^(SNR/10) * NO;
scaling_factor = sqrt(E_bav_des/E_bav);

% scale base constellation and symbols to true constellation
scaled_con = base_con * scaling_factor;
true_sym = sym * scaling_factor;

% produce noise and add to transmitted vectors
variance = NO;
noise_proc = sqrt(variance/2) * (randn([N, 1]) + 1j*randn([N, 1]));
noisy_transmitted = true_sym + noise_proc;

% determine estimates of transmitted signal
est_sym = l2_nearest(scaled_con, noisy_transmitted);
end

```

Figure 3: Transmission simulation for phase-coherent (non-differential) schemes

```

% simulate_transmission_diff: simulate transmitting symbols with noise, and
% returns the transmitted symbols and the estimates of the transmitted
% symbols after adding noise; assumes a differential scheme constellation
% (i.e., evenly spaced throughout a circle) where one constellation point
% lies at theta=0
%
% params:
% base_con = base constellation (M x 1)
% sym      = noise-free transmitted symbols (from base_cons, N x 1)
% NO       = noise power
% SNR      = desired SNR per bit (dB)
%
% returns:
% true_sym = transmitted (scaled) symbols; first one will be zero for
%            convenience
% est_sym   = estimated symbols
% scaling_factor = amount base constellation was scaled by
function [true_sym, est_sym, scaling_factor] ...
    = simulate_transmission_diff(base_con, sym, NO, SNR)

N = length(sym);
M = length(base_con);

% find desired average symbol energy
E_avg = mean(abs(base_con).^2);
E_bav = E_avg / ceil(log2(M));
E_bav_des = 10^(SNR/10) * NO;
scaling_factor = sqrt(E_bav_des/E_bav);

% scale base constellation and symbols to true constellation
scaled_con = base_con * scaling_factor;
true_sym = sym * scaling_factor;

% produce noise and add to transmitted vectors
variance = NO;
noise_proc = sqrt(variance/2) * (randn([N, 1]) + 1j*randn([N, 1]));
noisy_transmitted = true_sym + noise_proc;

% randomly rotate symbols to constant channel phase shift (this doesn't
% really have any real effect, more for show than anything)
phase_shift = 2*pi*rand();
noisy_transmitted = noisy_transmitted * exp(1j*phase_shift);

% translate both noisy_transmitted and true_sym into their
% phase differentials (their "values"); do this by dividing by the
% phase component of the previous element
noisy_transmitted = noisy_transmitted(2:N) ./ ...
    (noisy_transmitted(1:N-1) ./ abs(noisy_transmitted(1:N-1)));
true_sym = true_sym(2:N) ./ ...
    (true_sym(1:N-1) ./ abs(true_sym(1:N-1)));

% determine estimates of transmitted signal
est_sym = l2_nearest(scaled_con, noisy_transmitted);
end

```

Figure 4: Transmission simulation for differential schemes

3.3 Counting number of errors

To determine the number of errors given a sequence of true (`true_sym`) and estimated transmitted vectors (`est_sym`), we use the MATLAB function `nnz`, which finds the number on nonzero elements that exists after taking the difference between the sequence of true symbols and sequence of estimated symbols. A small arbitrary positive threshold of 0.0001 is used to prevent false negatives. This function can be used for both the non-differential or differential schemes, because the estimates of the information (i.e., the differential in the DPSK scheme) was already calculated in the transmission simulation function.

3.4 Generating (base) constellation shapes

The base constellation for binary antipodal and binary orthogonal are known and fixed: $\{-1, 1\}$ and $\{1, j\}$, respectively.

The base constellation for (D)PSK with M symbols is $\{\exp j \frac{2\pi}{m}\}, m \in \{0, 1, 2, \dots, M-1\}$.

For QAM, we used square constellations when M is a square number ($M \in \{16, 64\}$), and a rectangular constellation when M is rectangular ($M = 32$). Our code is optimized only for powers of 2 (square means exponent is even, rectangular means exponent is odd), which is a fair assumption (digital systems, e.g., WiFi, like powers of 2) and lends to the concise constellation-generating code:

```

if mod(log2(M), 2) == 0
    x = (1:sqrt(M)) - (sqrt(M)+1)/2;
    qam_cons = x + x.*1j;
else
    x = (1:sqrt(M/2)) - (sqrt(M/2)+1)/2;
    y = (1:sqrt(M*2)) - (sqrt(M*2)+1)/2;
    qam_cons = x + y.*1j;
end
qam_cons = qam_cons(:);      % flatten

```

Figure 5: Generating the QAM constellations (assuming M is a power of 2)

(Additionally, the textbook only gives explicit formulas to estimate the bit error rate when M is a power of 2.)

The base constellations are shown in Figure 6.

4 Results and discussion

The figures on the following pages demonstrate our results: BER as a function of SNR/bit for the different schemes. These compare very closely to the corresponding images in the textbook. The plots are scaled to look like the plots in the textbook, so they do not show the same SNR range. (The textbook is *Communications Systems Engineering, Second Edition*.)

The results mostly match the theoretical when N is large. They confirm the (mostly intuitive) explanations of and the theoretical formulas for the BER provided in the Review of Digital Modulation Schemes section.

Visually, we can see that the binary antipodal and binary orthogonal schemes have a lower BER than the other schemes, but this is mostly because they have so few symbols (M is small). The BER for binary antipodal is lower than that for binary orthogonal, which makes sense because the constellation points are farther from each other. Similarly, the PSK has a lower SNR than the DPSK, which is expected because due to the encoding of information in symbol differentials (but it compromises inherent phase coherence). In general, it is clear that increasing M decreases the robustness to noise.

One error we noticed was that in the QAM $M = 32$ (rectangular) case (see Figure 10), the theoretical equation used was supposed to be a tight upper bound, but it was not always an upper bound. The theoretical estimates are close, but it looks as if the shape is slightly different.

4.1 Code efficiency

The exact techniques were described in the Methods section, and the elapsed time results of running the simulation code on $N = 100000$ symbols is shown in Figure 11. The times shown are not very fast (the visualizations take a few minutes to complete), but they are reasonable considering that running these on domain-specific hardware may allow much better parallelization and other optimizations.

Running the simulations with $N = 10000$ runs (as expected) much quicker and only makes the plots a little noisier. The choice to use $N = 100000$ as opposed to $N = 10000$ for the plots in this report was completely arbitrary.

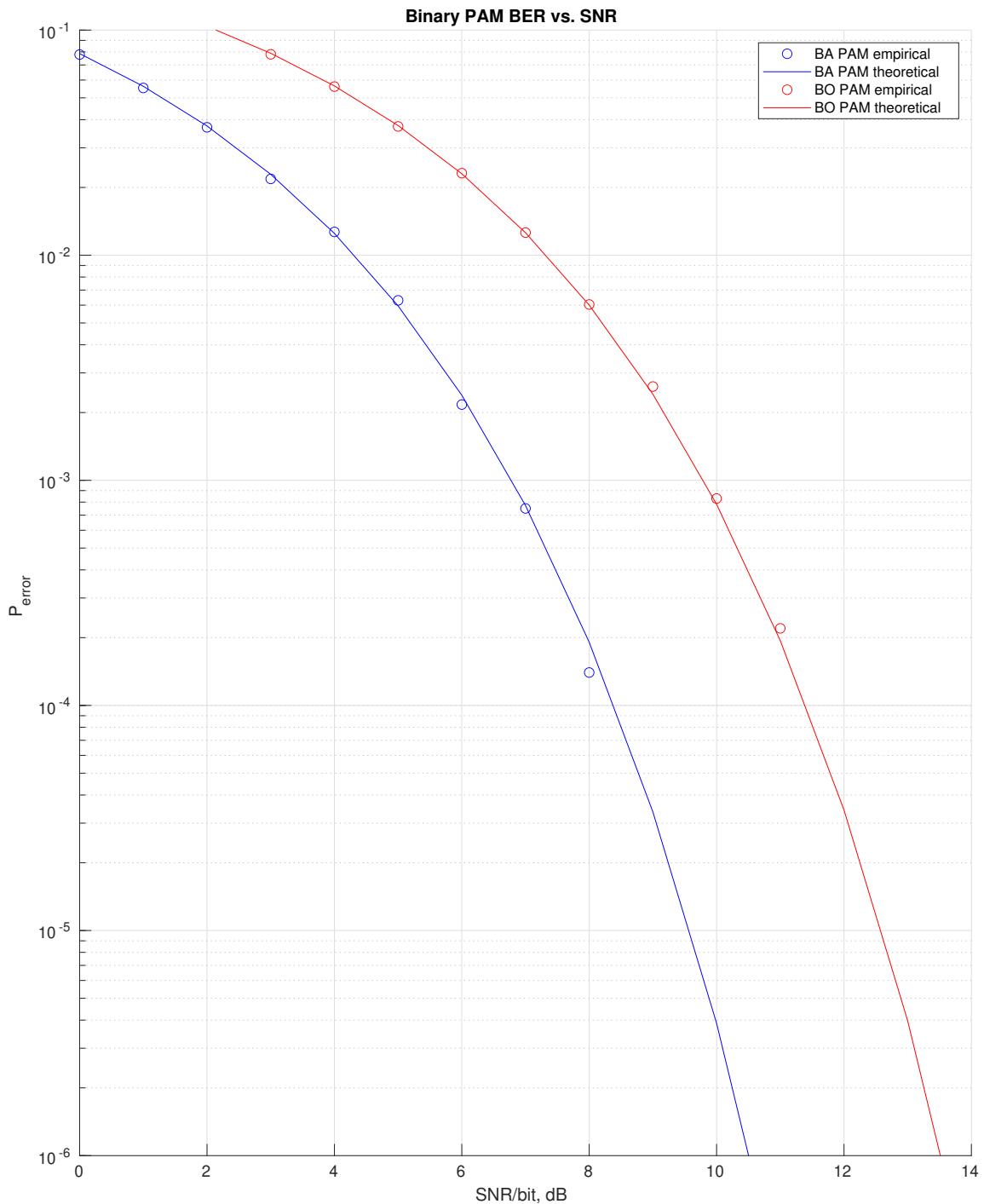


Figure 7: Bit error rate of binary PAM methods. Compare to Figure 7.53 in the textbook.

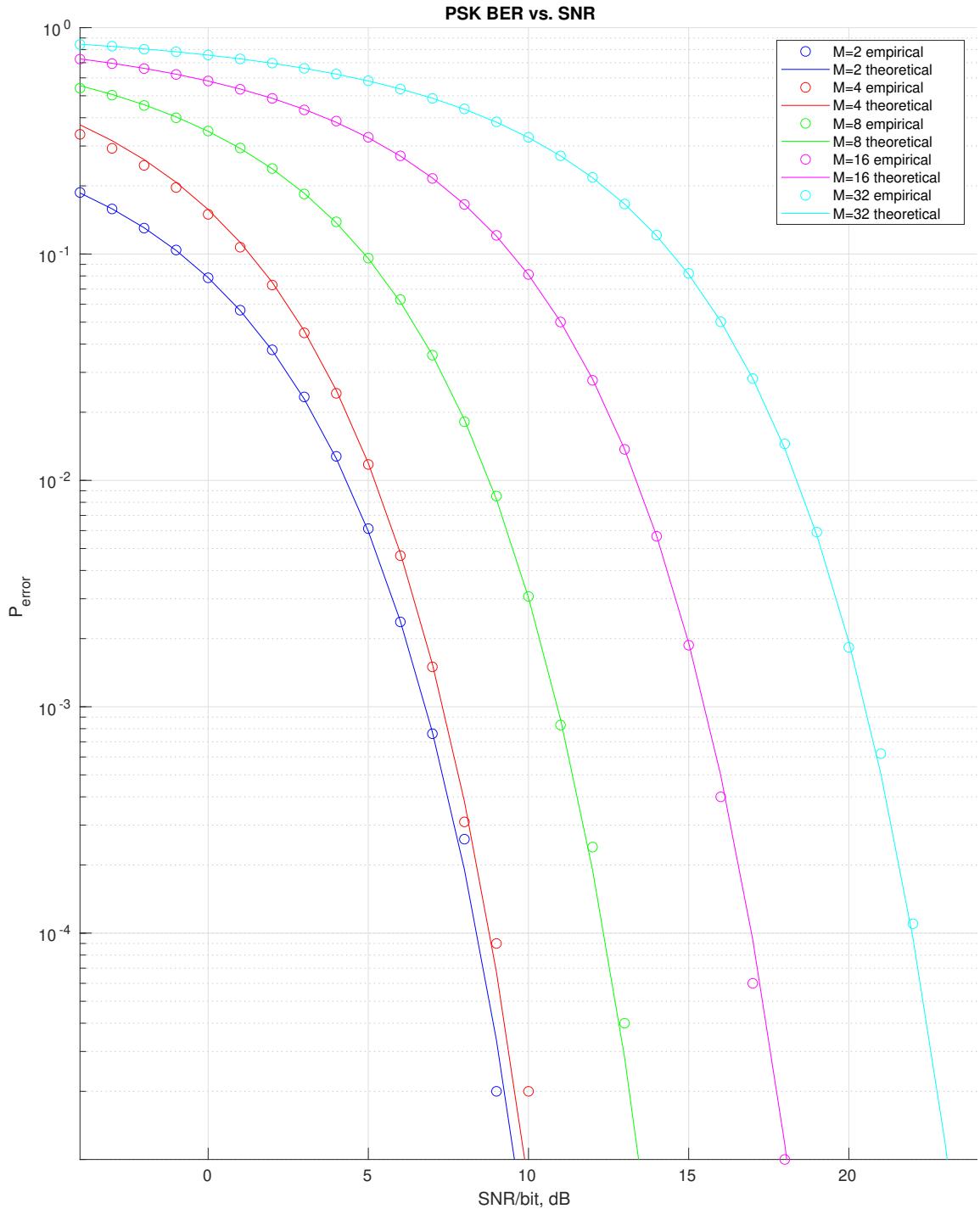


Figure 8: Bit error rate of PSK. Compare to Figure 7.57 in the textbook.

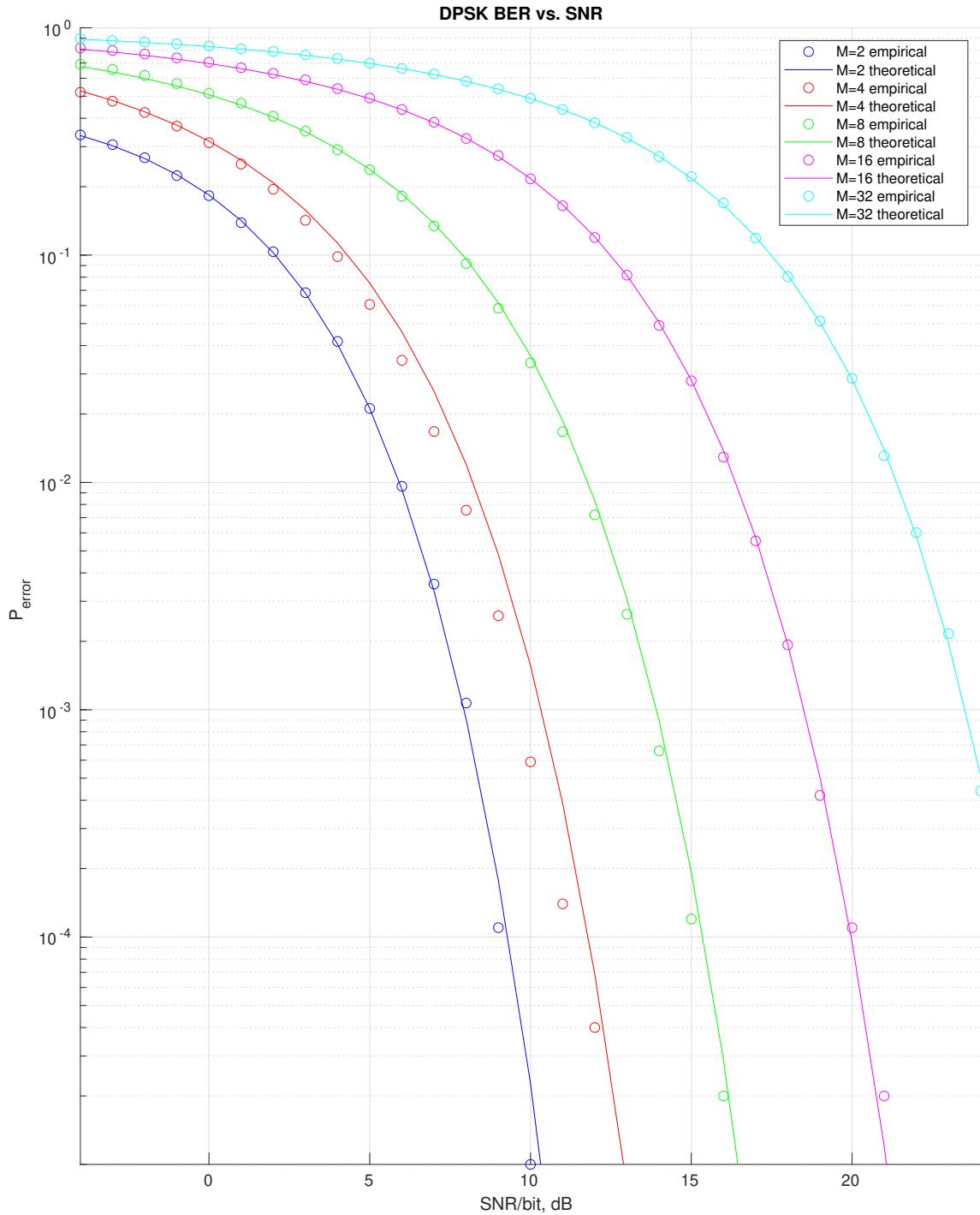


Figure 9: Bit error rate of DPSK. Compare to Figure 7.58 in the textbook.

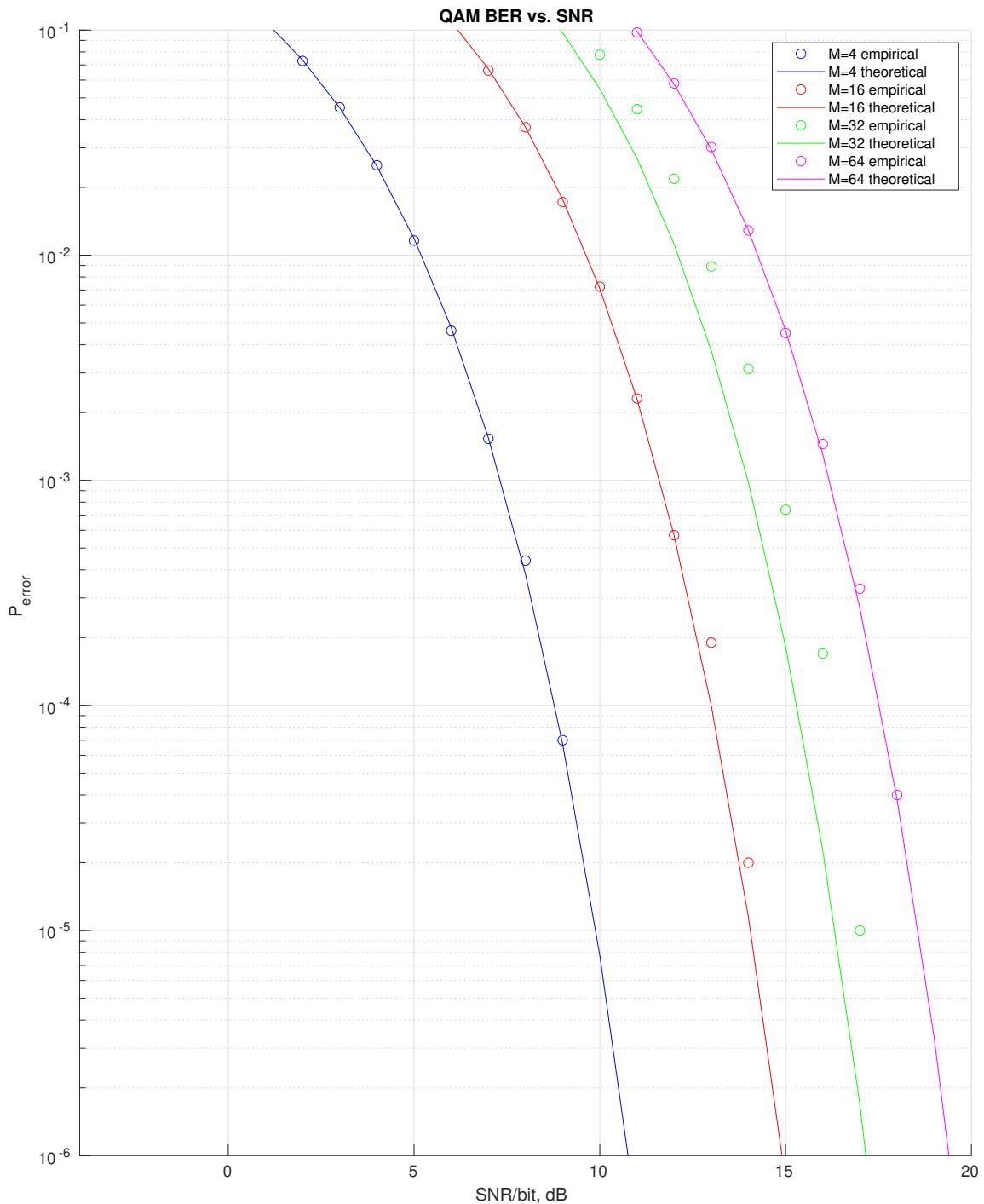


Figure 10: Bit error rate of QAM. Compare to Figure 7.62 in the textbook.

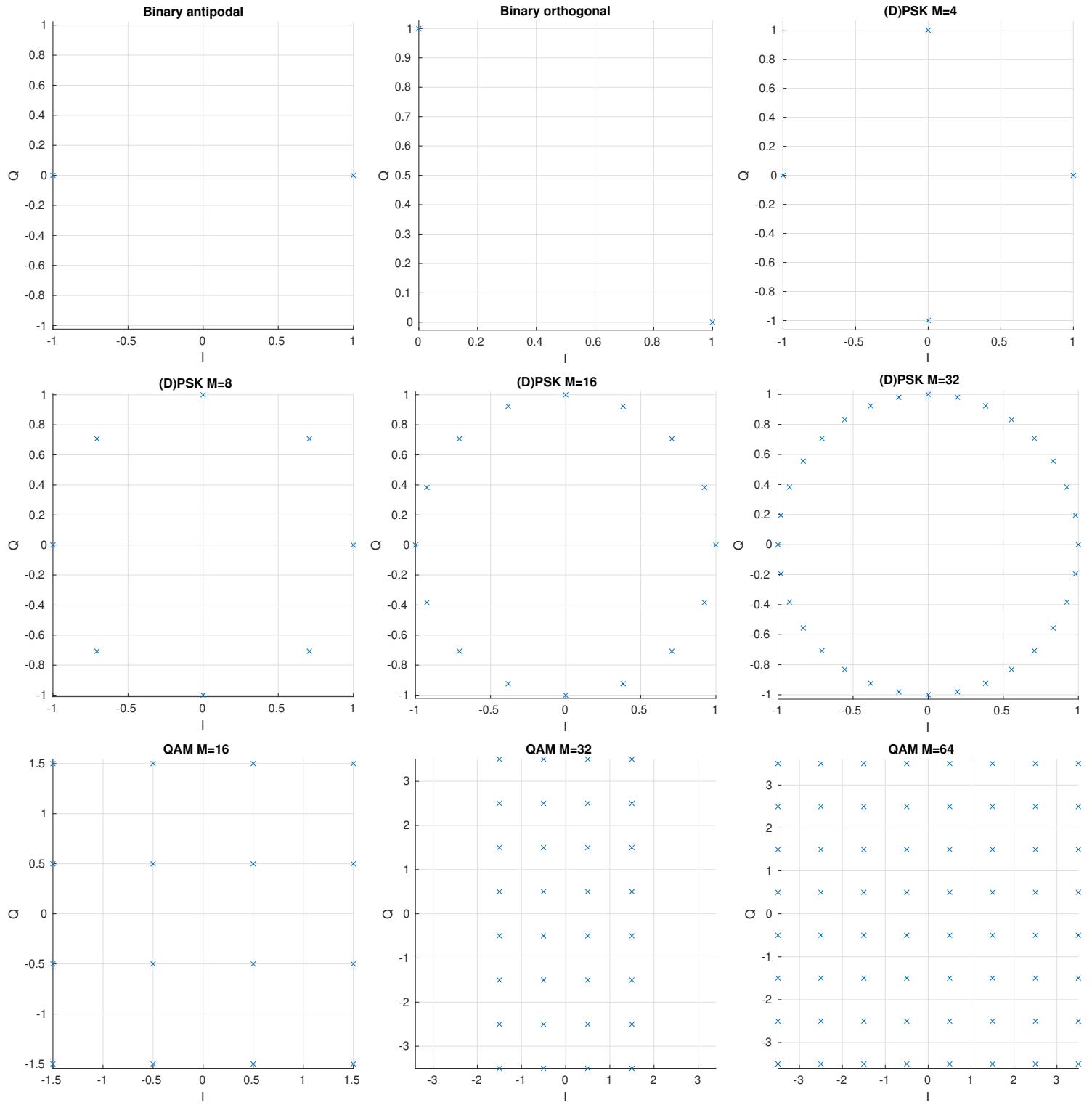


Figure 6: Base 16 constellation shapes

```
Elapsed for binary antipodal for N=100000: 10.33s
Elapsed for binary orthogonal for N=100000: 11.06s
Elapsed for PSK for N=100000: 62.05s
Elapsed for DPSK for N=100000: 62.71s
Elapsed for QAM for N=100000: 44.02s
```

Figure 11: Runtime for each section. Note that (D)PSK generates plots for four different M values, and QAM generates plots for four (larger) M values.

5 Conclusions and further inquiry

We were able to demonstrate the general relationships between SNR and bit error rate, as well as constellation shape and differential/non-differential coding, using a generic constellation framework. We designed MATLAB procedures that implement a least-squares decision given a generic set of symbols and 2-D constellation, as well as functions to simulate the transmission of given-SNR noisy digital symbols, for both differential and non-differential schemes. The results largely match the theoretical results for bit error rate.

We also did not resolve the error of the estimate for $M = 32$ for PAM (and presumably other rectangular constellations for odd powers of 2). This would be something else to explore in the future.

6 Appendix I: Simulation Code

```
% number of samples
N = 100000;

% noise power
NO = 1;

tic();

%% binary antipodal

% binary antipodal constellation and random signal sequence
bin_ap_const = [-1; 1];
bin_ap_sym = bin_ap_const(2 * ceil(rand(N, 1)));

% loop through SNRs from -4 to 20
Perrors_ap = zeros(1, 25);
Perrors_theo_ap = zeros(1, 25);
for i = 1:25
    SNR = i-5;

    [true_sym_ap, est_sym_ap, root_Eb] ...
        = simulate_transmission(bin_ap_const, bin_ap_sym, NO, SNR);
    Perror = num_errors(true_sym_ap, est_sym_ap) / N;
    Perrors_ap(i) = Perror;

    Perrors_theo_ap(i) = qfunc(root_Eb*sqrt(2/NO));

end

fprintf('Elapsed for binary antipodal for N=%d: %.2fs\n', N, toc());
tic();

figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
scatter(-4:20, Perrors_ap, 'b');
plot(-4:20, Perrors_theo_ap, 'b');

% binary orthogonal
bi_ortho_cons = [1; 1j];
bi_ortho_m = bi_ortho_cons(2 * ceil(rand(N, 1)));

Perrors_orth = zeros(1, 25);
Perrors_theo_orth = zeros(1, 25);
for i = 1:25
    SNR = i-5;
    [true_sym_ortho, est_sym_ortho, root_Eb] ...
        = simulate_transmission(bi_ortho_cons, bi_ortho_m, NO, SNR);
    Perror = num_errors(true_sym_ortho, est_sym_ortho)/N;
    Perrors_orth(i) = Perror;

    Perrors_theo_orth(i) = qfunc(root_Eb/sqrt(NO));
end

scatter(-4:20, Perrors_orth, 'r');
```

```

plot(-4:20, Perrors_theo_orth, 'r');
hold off;

title('Binary PAM BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
ylim([1e-6 1e-1]);
xlim([0 14]);
set(gca(), 'YScale', 'log');
grid on; % see note about fudge factor

legend([
    "BA PAM empirical", "BA PAM theoretical", ...
    "BO PAM empirical", "BO PAM theoretical" ...
]);
exportgraphics(gcf(), 'binary.eps');

fprintf('Elapsed for binary orthogonal for N=%d: %.2fs\n', N, toc());
tic();

%% PSK

Ms = [2 4 8 16 32];
colors = ["b", "r", "g", "m", "c"];
figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate basisResults
    theta = linspace(0, 2*pi, M+1);
    theta = theta(1:M).';
    PSK_cons = cos(theta) + ij*sin(theta);

    % generate signal sequence
    PSK_sym = PSK_cons(ceil(M * rand(N, 1)));

    Perrors_PSK = zeros(1,29);
    Perrors_theo_PSK = zeros(1,29);
    for j = 1:29
        SNR = j-5;
        [true_sym_psk, est_sym_psk, root_Eb] ...
            = simulate_transmission(PSK_cons, PSK_sym, NO, SNR);
        Perror = num_errors(true_sym_psk, est_sym_psk)/N;
        Perrors_PSK(j) = Perror;

        if M == 2
            Perrors_theo_PSK(j) = qfunc(root_Eb*sqrt(2/NO));
        else
            Perrors_theo_PSK(j) = 2*qfunc(root_Eb*sqrt(2/NO)*sin(pi/M));
        end
    end
    scatter(-4:24, Perrors_PSK, colors(i));
    plot(-4:24, Perrors_theo_PSK, colors(i));
end

```

```

hold off;
title('PSK BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=2 empirical", "M=2 theoretical", ...
    "M=4 empirical", "M=4 theoretical", ...
    "M=8 empirical", "M=8 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical"
]);
grid on;
ylim([1e-5 1e-0]);
xlim([-4 24]);
exportgraphics(gcf(), 'psk.eps');

fprintf('Elapsed for PSK for N=%d: %.2fs\n', N, toc());
tic();

%% DPSK

Ms = [2 4 8 16 32];
colors = ["b", "r", "g", "m", "c"];
figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate basis
    theta = linspace(0, 2*pi, M+1);
    theta = theta(1:M).';
    DPSK_cons = cos(theta) + 1j*sin(theta);

    % generate signal sequence
    DPSK_sym = DPSK_cons(ceil(M * rand(N, 1)));

    Perrors_DPSK = zeros(1,29);
    Perrors_theo_DPSK = zeros(1,29);
    for j = 1:29
        SNR = j-5;
        [true_sym_dpsk, est_sym_dpsk, root_Eb] ...
            = simulate_transmission_diff(DPSK_cons, DPSK_sym, NO, SNR);
        Perror = num_errors(true_sym_dpsk, est_sym_dpsk)/N;
        Perrors_DPSK(j) = Perror;

        if M == 2
            Perrors_theo_DPSK(j) = exp(-root_Eb^2/NO)/2;
        else
            Perrors_theo_DPSK(j) = 2*qfunc(root_Eb/sqrt(NO)*sin(pi/M));
        end
    end
    scatter(-4:24, Perrors_DPSK, colors(i));
    plot(-4:24, Perrors_theo_DPSK, colors(i));

```

```

end

hold off;
title('DPSK BER vs. SNR');
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=2 empirical", "M=2 theoretical", ...
    "M=4 empirical", "M=4 theoretical", ...
    "M=8 empirical", "M=8 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical"
]);
grid on;
ylim([1e-5 1e-0]);
xlim([-4 24]);
exportgraphics(gcf(), 'dpsk.eps');

fprintf('Elapsed for DPSK for N=%d: %.2fs\n', N, toc());
tic();

%% QAM
Ms = [4, 16, 32, 64];
colors = ["b", "r", "g", "m"];

figure('visible', 'off', 'position', [0 0 1000 1200]);
tiledlayout(1, 1, 'TileSpacing', 'Compact');
nexttile();
hold on;
for i = 1:length(Ms)
    M = Ms(i);

    % generate constellation (assume M is a power of 2)
    if mod(log2(M), 2) == 0
        x = (1:sqrt(M)) - (sqrt(M)+1)/2;
        % use broadcasting to generate sqrt(M)*sqrt(M) square
        qam_cons = x + x.*'1j';
    else
        x = (1:sqrt(M/2)) - (sqrt(M/2)+1)/2;
        y = (1:sqrt(M*2)) - (sqrt(M*2)+1)/2;
        % use broadcasting to generate sqrt(M/2)*sqrt(M*2) rectangular
        qam_cons = x + y.*'1j';
    end
    qam_cons = qam_cons(:);      % flatten

    % generate signal sequence
    qam_sym = qam_cons(ceil(M * rand(N, 1)));

    Perrors_QAM = zeros(1, 25);
    Perrors_theo_QAM = zeros(1, 25);
    for j = 1:25
        SNR = j-5;
        [qam_true_sym, qam_est_sym, scale_factor] ...
            = simulate_transmission(qam_cons, qam_sym, NO, SNR);
        Perrors_QAM(j) = num_errors(qam_true_sym, qam_est_sym) / N;

        E_avg = mean(abs(qam_true_sym).^2);
    end
end

```

```

if mod(log2(M), 2) == 0
    Prootm = 2*(1-1/sqrt(M))*qfunc(sqrt(3*E_avg/((M-1) * NO)));
    Perrors_theo_QAM(j) = 1 - (1 - Prootm)^2;
else
    Perrors_theo_QAM(j) = 1 - (1 - 2*qfunc(sqrt(3*E_avg/((M-1)*NO))))^2;
end
end

scatter(-4:20, Perrors_QAM, colors(i));
plot(-4:20, Perrors_theo_QAM, colors(i));
end

hold off;
title(sprintf('QAM BER vs. SNR'));
ylabel('P_{error}');
xlabel('SNR/bit, dB');
set(gca(), 'YScale', 'log');
legend([
    "M=4 empirical", "M=4 theoretical", ...
    "M=16 empirical", "M=16 theoretical", ...
    "M=32 empirical", "M=32 theoretical", ...
    "M=64 empirical", "M=64 theoretical" ...
]);
grid on;
ylim([1e-6 1e-1]);
xlim([-4 20]);
exportgraphics(gcf(), 'qam.eps');

fprintf('Elapsed for QAM for N=%d: %.2fs\n', N, toc());

```

ECE300 Project 3: LBCs and Coding Theory

Jonathan Lam, Sam Shersher, Arthur Skok, Hadassah Yanofsky

December 18, 2020

Contents

1	Introduction	2
1.1	On Modulation Schemes and Coding	2
1.2	Coding Methods	2
1.3	Context of the Project	3
2	Methods	4
2.1	Galois Field 2 Arithmetic	4
2.1.1	Alternative Methods for \mathbb{F}_2 Arithmetic	4
2.2	Simulating the Binary Symmetric Channel	4
2.3	Generating the Parity Check Matrix H	5
2.4	Correctable and Detectable Errors	5
2.5	Generating the Truncated Syndrome Array	6
2.6	Error Correction	7
3	Discussion	8
3.1	Bit Error Rate Comparison	8
3.2	The Truncated Syndrome Array	9
4	Appendix: Source code	10

1 Introduction

1.1 On Modulation Schemes and Coding

Practical communications theory has long since evolved from analog-dominated communication schemes to the binary coding schemes that are useful for digital systems. In the previous project, we explored several digital modulation schemes, such as PSK and QAM, and we compared differential and non-differential schemes.

We studied these methods in the context of one-shot communication (i.e., one signal corresponding to one message) and the effect of noise on each scheme. Each scheme is assumed to be transmitted over communication channels with additive white Gaussian noise (AWGN), and we analyzed their performances as measured by bit error rate (BER).

Having studied digital modulation schemes (which allow us to transmit digital signals), we then have to figure out how to encode our data into those digital signals. That brings us to information theory (in particular coding theory), which allows us to quantize information and develop codes that allow us to quantify and correct errors.

1.2 Coding Methods

Among the number of coding schemes to be explored in coding theory, this report will examine **Linear Block Codes**, through the manipulation and simulation of code-words in MATLAB.

A (n, k) linear block code is a coding scheme which contains a collection of $M = 2^k$ binary sequences where each are of length n . Each of these sequences are a **Codeword** belonging to this block's **Codebook**.

Hamming Codes are a special class of linear block codes that achieve the maximum possible rates for LBCs with a minimum Hamming distance between any two keywords of three and given their value of n . In a Hamming code, for some integer $m \geq 3$ then the code has $n = 2^m - 1$ and $k = 2^m - m - 1$.

For linear codes, one can create a **Generator Matrix**, such that all the codewords in the code are linear combinations of the rows in the generator matrix. The generator matrix for a (n, k) LBC has dimensions $k \times n$.

In special cases, one may have a **Systematic Code**, where the first k columns of a generator matrix are a $k \times k$ identity matrix, and thus each codeword would begin with an identifier for the respective word it represents, i.e., a block code that has $k = 2$ would have its first two columns be:

$$\begin{bmatrix} 1 & 0 & \dots \\ 0 & 1 & \dots \end{bmatrix}$$

1.3 Context of the Project

In this project, we deal with systematic codes. Two generator matrices with dimensions (4, 8) and (4, 12) are provided.

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$G_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Parity Check Matrices were created for each code. A parity check matrix is created directly from systematic generator matrix, i.e., a generator matrix of form:

$$G = [I_k \mid P]$$

The matrix P formed from the bits to the right of the identity matrix are also called the parity check bits.

Using these two components, the resulting parity check matrix H would be an $n - k$ identity matrix, appended to the negative of the transpose of the parity check bits:

$$H = [-P^T \mid I_{n-k}]$$

It should be noted that the correctness of a parity check matrix can be verified via matrix multiplication, if the following matrix equation holds true:

$$GH^T = \mathbf{0}_{k \times (n-k)}$$

The resulting zero product is attributed to the orthogonality between rows of the generator matrix, (each row being a codeword of the original code) to the rows of the parity check matrix.

The code being used to generate and utilize these structures are described in §Methods.

2 Methods

2.1 Galois Field 2 Arithmetic

We implemented Galois field 2 (henceforth denoted \mathbb{F}_2) addition and multiplication in MATLAB by defining separate functions for each. To implement this arithmetic, we used the `mod` function since the \mathbb{F}_2 addition and multiplication operations are equivalent to ordinary matrix operations over the field \mathbb{R} modulus 2. Note that these functions can be used for scalar values (i.e., degenerate matrices). Our implementations are shown in Figure 1.

```
function sum = galois2_add(A, B)
    sum = mod(A + B, 2);
end
```

(a) Addition

```
function prod = galois2_multiply(A, B)
    prod = mod(A * B, 2);
end
```

(b) Multiplication

Figure 1: Implementing \mathbb{F}_2 arithmetic on MATLAB arrays (or scalars).

2.1.1 Alternative Methods for \mathbb{F}_2 Arithmetic

MATLAB has a built-in \mathbb{F}_2 type that can be created with `gf2()`. It appears to be a special wrapper around `uint8` matrices that overloads several matrix operations, (most notably the addition and multiplication field operations) and only allows values of unity and zero. We were able to implement this for most of the basic operations, but failed when we had to deal with `NaN` values. We assume that this would perform better than our implementation, since it uses smaller datatypes and fixed-point (integral) calculations.

Using the understanding that operations modulo two (and only dealing with values of unity and zero) act similarly to bitwise operations, we also wondered if these would improve the performance of our simulation. These functions are shown in Figure 2. While these produced the same results as the implementation shown above, this actually performed roughly 10% slower. Our conjecture is that floating-point modulus is implemented as a native instruction (e.g., `fmod`), while bitwise operations are only defined on integral types (and thus require an extra FP-to-int and int-to-FP operation) for each \mathbb{F}_2 operation, which could be more expensive than the benefits that bitwise operations provide.

2.2 Simulating the Binary Symmetric Channel

The next step was to simulate corruption of a bitstring in order to simulate a binary symmetric channel. We used the `rand` function to create an array of random bit values Bernoulli-distributed between 0 and 1 with probability P_{err} , and used this to corrupt a provided bitstring (a \mathbb{F}_2 array). The function implementation is shown in Figure 3.

```

function sum = galois2_add_alt(A, B)
    sum = bitxor(A, B);
end

```

(a) Addition

```

function prod = galois2_multiply(A, B)
    prod = bitand(A * B, 1);
end

```

(b) Multiplication

Figure 2: Bitwise implementation of \mathbb{F}_2 arithmetic.

```

function corrupted = corrupt_bitstring(str, p_err)
    corrupted = galois2_add(str, rand(size(str)) < p_err);
end

```

Figure 3: Bit corruption with error probability p_{err}

2.3 Generating the Parity Check Matrix H

We wrote a function to find H for a given systematic code using the definition of H :

```

function H = parity_check(G)
    [k,n] = size(G);
    P = G(:, (k+1):end);
    I = eye(n - k);
    H = [P. ' I];
end

```

Figure 4: Generating the parity check matrix H for a LBC given G .

Then, we found the truncated syndrome array by multiplying the first word in each row - just the word of errors - of the truncated standard array by H in Galois field 2.

2.4 Correctable and Detectable Errors

To determine the maximum number of correctable errors, it was necessary to first determine the minimum Hamming distance between any two codewords in the code. This was simple to do given that $d_{\min} = w_{\min}$ in a LBC (proof deferred to class notes or the textbook), and the minimum hamming weight can be found by finding the codeword with the fewest number of set bits (ones).

Using this function we found that the d_{\min} are 3 and 5 for codes 1 and 2, respectively. Using the formula for the maximum correctable errors:

$$e_c = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$$

```

function min_dist = min_hamming_distance(code)
    min_dist = min(sum(code(2:end,:), 2));
end

```

Figure 5: Finding the minimum Hamming distance between two codewords of a code using the property that $d_{min} = w_{min}$.

and found the maximum correctable errors to be 1 and 2, respectively. If the number of errors exceeds this number for any code, the codeword is not correctable because there is no unique closest codeword. However, as long as the error does not transform the original codeword into another codeword, the error is detectable. In this case, the receiver will still know that there were at least as many bit errors as the minimum Hamming distance between the received bitstring and any other codeword.

2.5 Generating the Truncated Syndrome Array

Normally, the standard array is a $2^{n-k} \times 2^k$ matrix that contains all of the possible 2^k possible n -bit strings, with each row representing all possible errors resulting in a unique n -bit string that is not a codeword (except for the first row, in which the “error” is the n -bit zero string). We were only concerned with finding correctable errors, so we constructed a truncated standard array with only e_c bit errors for each code. In general, with a maximum of e_c correctable errors, this truncated standard array would have r rows (i.e., number of correctable errors, plus one if we include the no-error case), where r can be found by the following equation:

$$r = \sum_{i=0}^{e_c} \binom{n}{i}$$

In other words, we count up the number of ways zero errors can be performed (this is always the first row of the standard array), the number of ways one error can be performed, the (distinct) number of ways two errors can be performed, and so on until the number of correctable errors is reached. In our case, the first truncated standard array has $r = 1 + 8 = 9$ rows and the second truncated standard array has $r = 1 + 12 + 66 = 79$ rows. The full standard arrays of the first and second codes have 16 and 512 rows, respectively.

We can obtain the truncated syndrome array from the truncated standard array by applying the parity check matrix to any element in each row of the array. This holds because of a theorem discussed in class that states that every row of the standard array (and therefore the truncated standard array) is a coset, so that the parity check matrix H applied to any word in the row results in the coset leader, called the syndrome. This means that a matrix can be turned a single column, which leads to a large memory saving.

2.6 Error Correction

To correct errors efficiently for each code, we stored a hashtable (for $O(1)$ lookups) that maps each syndrome to its corresponding error. Since each syndrome and error are multiple-valued (matrices, which are not hashable by default), we first “hash” these by taking the decimal representation of both the syndromes and the errors. When the syndrome of a received word is found in this hashtable, then it is corrected by adding it to the received code (since addition is the same as subtraction in \mathbb{F}_2 arithmetic). Conversely, if it is not found, the error was not correctable since the truncated standard array contains all correctable errors. To signify this we set all the bits in the estimated received word to NaN. The implementation for the error correction is shown in Figure 6.

```
% correct_errors: returns corrected codeword if correctable,
% otherwise NaN
%
% params:
% SE_map= container.Map instance mapping syndromes to errors
% H      = parity check matrix for the code
% X      = received word ((# words) x N)
% returns:
% Xhat = estimated codeword if correctable, else NaN
function Xhat = correct_errors(SE_map, H, X)
    % hashmap can only be accessed linearly (not vectorizable),
    % so this has to be implemented in a loop
    Xhat = zeros(size(X));
    for i = 1:size(X, 1)
        x = X(i, :);
        try
            Xhat(i,:) = galois2_add(x, ...
                de2bi(SE_map(bi2de(...
                    galois2_multiply(x, H.'))), length(x)));
        catch
            % if not found in SE_map
            Xhat(i,:) = NaN * ones(size(x));
        end
    end
end
```

Figure 6: Correcting correctable errors in the codeword. Note that we allow arbitrary matrices (in which each row is a codeword) as the input X , for the convenience of the caller.

3 Discussion

3.1 Bit Error Rate Comparison

This project explored two different error correction schemes. The first code is a (8, 4) LBC and the second one is a (12, 4) LBC. Both codes are systematic codes. For each code the bit error rate was calculated over a range of transmission bit error probabilities (ranging from 0.001 to 0.1).

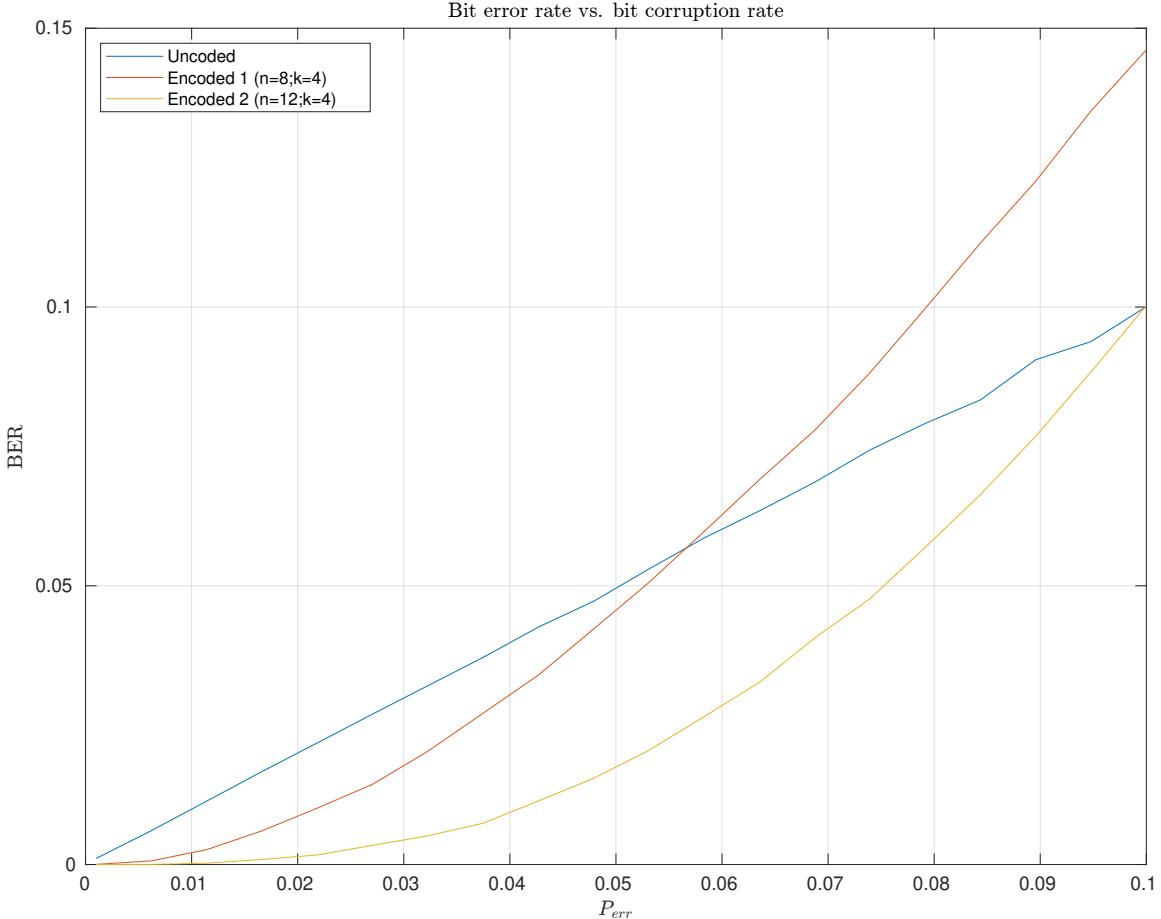


Figure 7: Error corrections schemes probability of error and corruption rates

We plot the different BERs over the range of transmission bit error probabilities for the two different schemes in Figure 7. Clearly the second scheme is better than the first scheme. The first scheme noticeably has larger bit rate error for the range of probabilities of error given, which makes sense given its ability to correct one more bit errors per codeword (the classic redundancy-error tradeoff). The probability of n errors occurring in the binary symmetric channel

is P_{err}^n , and thus even with the highest P_{err} tested, a non-correctable word is 10 times more likely to arise in the first code than in the second.

Note that the bit error rate (BER) using the first code actually overtakes the BER without an error correction scheme at around $P_{err} = 0.055$, meaning that in channels noisier than this, using no code seems be more reliable than using the first code. While this seems confusing, the reason for this is that our analysis between the encoded and uncoded schemes is different: when a detectable (but uncorrectable) error is found for the encoding scheme, then the entire word is thrown out (resulting in n errors); however, for the uncoded scheme, a single bit error is not regarded as the loss of an entire word; we only count the bits that are wrong.

The difference is that the encoded scheme (very likely) detects when a error has occurred in a codeword, and thus marks the entire word as wrong (and will most likely request re-transmission), while the uncoded scheme cannot tell when a code is incorrect without the use of some higher-level error checking scheme (e.g., checksumming). In other words, the “bit error rate” shown here for the encoded schemes is the number of words with a detected error divided by the total number of transmitted words, whereas the “bit error rate” shown here for the uncoded scheme is the total number of bit errors (which cannot be detected by the receiver) divided by the total number of transmitted bits. Clearly there is a discrepancy here in the interpretation (which comes from the phrasing of the question in the assignment). If we were to interpret the uncoded scheme in the former way, then no errors would ever be detected even when there are errors proportional to (roughly equal to) the transmission bit error; conversely, if we were to interpret the encoded scheme in the latter way, the bit error rate would be lower than the transmission bit error due to the fact that it has the ability to correct some errors. To summarize, the percentage of incorrect words should be lower for both encoded schemes than for the uncoded scheme.

3.2 The Truncated Syndrome Array

A truncated syndrome array was created in this project rather than the full syndrome array. This consists of the syndromes of all errors containing at most e_c bit errors. Errors with more than e_c map codewords to bitstrings that lie equidistant (in the Hamming sense) to at least two other codewords, or map codewords to other codewords; there is no way to choose the most reasonable correction, so we toss these words.

What we do with tossed words is application-dependent. For situations requiring high reliability, such as file transfer or any secure information transfer, it would make sense for a client (receiver) to re-request the corrupted words until it gets a word without corruption (and even then, it should check for non-detectable errors using other methods like checksums). For low-reliability situations, especially non-critical and low-latency (e.g., real-time in the consumer sense but not in the RTOS sense) applications like streaming, the tossed words may be discarded or re-requested depending on the communication channel bandwidth and rate, desired quality of communication, desired latency, etc.

4 Appendix: Source code

The code to generate the standard arrays and syndrome arrays for codes 1 and 2 is shown in Figure 8.

The code that drives the simulation and evaluation of the bit error rate after transmission for the uncoded and encoded schemes is shown in Figure 9. The standard output of this script, which comprises logging information about the probabilities of errors calculated and the elapsed time taken, is shown in Figure 10. The given output comes from running the simulation script on an i7-2600 CPU. It is evident from these slow output times (≈ 400 s for 100,000 codewords, a small number these days considering modern network speeds in the hundreds of megabits per second) that a CPU is not an optimal encoder and decoder for these schemes; it is probably much faster to use dedicated hardware with hardcoded logic and higher parallelization.

The source code for this project can also be found at <https://github.com/jlam5555/ece300-proj3>.

```

%% This file generates the syndrome arrays S1 and S2 (corresponding to
% codes c1, c2)
clc; clear; close all;

% sac = standard array correctable
load('code.mat');

%% code 1
corr1 = 1;
k1 = 4;
n1 = 8;
sac_1 = zeros(8+1, 2^k1, n1);

% first row of standard array is no errors
sac_1(1, :, :) = c1;

I = eye(n1);

for i = 1:n1
    sac_1(i+1, :, :) = galois2_add(c1, I(i, :));
end

% create
E1 = reshape(sac_1(:, 1, :), [], n1);
S1 = galois2_multiply(E1, parity_check(G1).');
SE_map1 = containers.Map(bi2de(S1), bi2de(E1));

%% code 2
corr2 = 2;
k2 = 4;
n2 = 12;
sac_2 = zeros(1+12+12*11/2, 2^k2, n2);

% 12
I = eye(n2);

% first row of standard array is no errors
sac_2(1, :, :) = c2;

counter = 2;

for i = 1:n2
    sac_2(counter, :, :) = galois2_add(c2, I(i, :));
    counter = counter + 1;
end

for i = 2:n2
    mat = I(i, :);
    for j = 1:i-1
        sac_2(counter, :, :) = galois2_add(c2, mat + I(j, :));
        counter = counter + 1;
    end
end

E2 = reshape(sac_2(:, 1, :), [], n2);
S2 = galois2_multiply(E2, parity_check(G2).');
SE_map2 = containers.Map(bi2de(S2), bi2de(E2));

%% save to .mat file
save('syndrome.mat', 'E1', 'S1', 'SE_map1', 'E2', 'S2', 'SE_map2');

```

Figure 8: Script to generate the standard and syndrome arrays

```

%% Q8: Evaluate the performance of the error correction
clc; clear; close all;
set(0, 'defaultTextInterpreter', 'latex');

load('code.mat'); % c1, c2, G1, G2
load('syndrome.mat'); % S1, S2, E1, E2, SE_map1, SE_map2

[K1, N1] = size(G1);
[K2, N2] = size(G2);

H1 = parity_check(G1);
H2 = parity_check(G2);

%% randomly generate 10^5 words for each scheme & encode
word_count = 1e5;

uncoded1 = de2bi(randi(2^K1-1, word_count, 1), K1);
uncoded2 = de2bi(randi(2^K2-1, word_count, 1), K2);

encoded1 = galois2_multiply(uncoded1, G1);
encoded2 = galois2_multiply(uncoded2, G2);

%% randomly corrupt bits with variable error probability
p_errs = linspace(1e-3, 1e-1, 20);

% results matrix
bers = zeros(3, length(p_errs));

tic();
for i = 1:length(p_errs)
    % note: this takes a while.....
    p_err = p_errs(i);
    fprintf('(%f) Simulating p_err=%f\n', toc(), p_err);
    bers(:, i) = [
        ber(corrupt_bitstring(uncoded1, p_err), uncoded1); ...
        ber(correct_errors(SE_map1, H1, ...
            corrupt_bitstring(encoded1, p_err)), encoded1); ...
        ber(correct_errors(SE_map2, H2, ...
            corrupt_bitstring(encoded2, p_err)), encoded2)
    ];
end

%% plot results

uncoded_bers = bers(1, :);
encoded1_bers = bers(2, :);
encoded2_bers = bers(3, :);

figure('visible', 'off', 'Position', [0 0 1000 750]);
plot(p_errs, uncoded_bers(:), ...
    p_errs, encoded1_bers(:), ...
    p_errs, encoded2_bers(:));
grid on;
title('Bit error rate vs. bit corruption rate');
xlabel('$P_{err}$');
ylabel('BER');
legend(['Uncoded', "Encoded 1 (n=8;k=4)", "Encoded 2 (n=12;k=4)"], ...
    'Location', 'northwest');
exportgraphics(gca(), 'ber_vs_bcr.pdf');

```

Figure 9: Driver script for evaluating and plotting the BER

```
(0.000116s) Simulating p_err=0.001000
(19.434827s) Simulating p_err=0.006211
(38.710674s) Simulating p_err=0.011421
(57.417325s) Simulating p_err=0.016632
(76.253500s) Simulating p_err=0.021842
(96.177417s) Simulating p_err=0.027053
(115.950588s) Simulating p_err=0.032263
(135.571747s) Simulating p_err=0.037474
(155.492180s) Simulating p_err=0.042684
(175.412776s) Simulating p_err=0.047895
(195.985129s) Simulating p_err=0.053105
(216.431429s) Simulating p_err=0.058316
(236.948146s) Simulating p_err=0.063526
(257.570973s) Simulating p_err=0.068737
(278.481355s) Simulating p_err=0.073947
(299.737082s) Simulating p_err=0.079158
(322.724341s) Simulating p_err=0.084368
(345.474427s) Simulating p_err=0.089579
(368.047150s) Simulating p_err=0.094789
(391.268665s) Simulating p_err=0.1000
```

Figure 10: Simulation script standard output

ECE300 - Pset 1

Jonathan Lam

September 7, 2020

1. A square wave has period T , amplitude A and duty cycle τ/T (the signal takes value A from time 0 to time τ , then 0 from time τ to T). Find the Fourier series representation of this signal.

Let the square wave be $x(t)$, and let c_n denote the Fourier coefficients.

$$\begin{aligned}
 c_n &= \frac{1}{T} \int_0^T x(t) \exp(-j\omega nt) dt \\
 &= \frac{1}{T} \left[\int_0^\tau A \exp(-j2\pi nt/T) dt + \int_\tau^T 0 \exp(-j2\pi nt/T) dt \right] \\
 &= \frac{1}{T} \left[A \int_0^\tau \exp(-j2\pi nt/T) dt + 0 \right] \\
 &= j \frac{A}{T} \frac{T}{2\pi n} \exp(-j2\pi nt/T) \Big|_{t=0}^{t=\tau} \\
 &= j \frac{A}{2\pi n} (\exp(-j2\pi n\tau/T) - 1)
 \end{aligned}$$

Note that this general formula doesn't work for the $n = 0$ case, so it needs to be done manually:

$$c_0 = \frac{1}{T} \int_0^\tau A \exp(0) dt = \frac{A\tau}{T}$$

Plugging into the Fourier series:

$$\begin{aligned}
 x(t) &= \sum_{n=-\infty}^{+\infty} c_n e^{j2\pi nt/T} \\
 &= c_0 e^0 + \frac{A}{2\pi} \sum_{\substack{n=-\infty \\ n \neq 0}}^{+\infty} \frac{1}{n} e^{j\pi/2} (e^{-j2\pi n\tau/T} - e^0) e^{j2\pi nt/T} \\
 &= \frac{A\tau}{T} + \frac{A}{2\pi} \sum_{\substack{n=-\infty \\ n \neq 0}}^{+\infty} \frac{1}{n} \left(e^{j(\pi/2+2\pi n/T(t-\tau))} - e^{j(\pi/2+2\pi n/T)} \right)
 \end{aligned}$$

Alternatively, to express this as a real signal, we know that $c_n = c_{-n}^*$, and therefore the summand $s_n := c_n e^{j2\pi nt/T}$ also has this conjugate symmetry:

$$s_n = c_n e^{j\pi nt/T} = \left((c_n^*) \left(e^{j2\pi nt/T} \right)^* \right)^* = \left(c_{-n} e^{j2\pi(-n)t/T} \right)^* = s_{-n}^*$$

Since $s_n + s_{-n} = s_n + s_n^* = 2 \operatorname{Re} s_n$, the value of the complex sum is the same as twice the real part of the summand for the nonzero terms. Thus, the Fourier series in terms of real functions only is:

$$x(t) = \frac{A\tau}{T} + (2) \frac{A}{2\pi} \sum_{n=1}^{+\infty} \frac{1}{n} \left(\cos\left(\frac{\pi}{2} + \frac{2\pi n}{T}(t - \tau)\right) - \cos\left(\frac{\pi}{2} + \frac{2\pi n}{T}t\right) \right)$$

2. Does the signal from the previous question have finite energy? Does it have finite power? If the answer to either question is yes, find the value of the energy or power.

The signal has infinite energy (since it is periodic, and thus has a nondiminishing average value even as $t \rightarrow \pm\infty$). It does have finite power. Since it is periodic, we only have to calculate the power over one period:

$$P = \frac{1}{T} \int_0^T |x(t)|^2 dt = \frac{1}{T} \left[\int_0^\tau A^2 dt + \int_\tau^T 0^2 dt \right] = \frac{A^2 \tau}{T}$$

3. What is $\operatorname{sinc}(t) * \operatorname{sinc}(t)$?

The Fourier transform of a (normalized) sinc function is the rect function. The (pointwise) product of the rect function with itself is itself, and the inverse Fourier transform of itself is the normalized sinc function. So, in this case the convolution of sinc with itself is itself.

$$\begin{aligned} \operatorname{sinc}(t) &:= \frac{\sin \pi t}{\pi t} \\ \operatorname{rect}(f) &:= \begin{cases} 1 & -1/2 \leq f < 1/2 \\ 0 & \text{else} \end{cases} \end{aligned}$$

$$\begin{aligned} \mathcal{F}\{\operatorname{sinc}(t)\}(f) &= \operatorname{rect}(\omega) \Leftrightarrow \mathcal{F}^{-1}\{\operatorname{rect}(f)\}(t) = \operatorname{sinc}(t) \\ \operatorname{sinc}(t) * \operatorname{sinc}(t) &= \mathcal{F}^{-1}\{\mathcal{F}\{\operatorname{sinc}(t) * \operatorname{sinc}(t)\}(f)\}(t) \\ &= \mathcal{F}^{-1}\{\mathcal{F}\{\operatorname{sinc}(t)\}(f) \cdot \mathcal{F}\{\operatorname{sinc}(t)\}(f)\}(t) \\ &= \mathcal{F}^{-1}\{\operatorname{rect}(f) * \operatorname{rect}(f)\}(t) = \mathcal{F}^{-1}\{\operatorname{rect}(f)\}(t) \\ &= \operatorname{sinc}(t) \end{aligned}$$

In the case of the unnormalized sinc function, its Fourier transform would be a different rect function, which leads to a similar answer (the only difference being that there would be a non-unity scaling factor introduced when multiplying the different rect function with itself).

4. Suppose a system acts on signal x and produces output y by the rule

$$y(t) = |x(t+3)|$$

Is the system linear? Is the system time invariant? Is the system causal?

The system is not causal because it depends on future values of t (e.g., $y(0)$ is a function of $x(3)$). The system is not linear and is time-invariant:

$$\begin{aligned} y\{cx_1 + x_2\}(t) &= |cx_1(t+3) + x_2(t+3)| \\ &\neq c|x_1(t+3)| + |x_2(t+3)| = cy\{x_1\}(t) + y\{x_2\}(t) \end{aligned}$$

$$y\{x(t)\}(t-t_0) = |x((t-t_0)+3)| = |x((t+3)-t_0)| = y\{x(t-t_0)\}(t)$$

5. Let $x(t)$ be the signal given by $\cos(2\pi f_0 t)$ for $0 \leq t \leq T$ and 0 otherwise, where $T = 1/f_0$. Find the Fourier transform, $X(\omega)$, of this signal. Demonstrate Parseval's theorem by comparing the norms of x and X .

$$\begin{aligned} \mathcal{F}\{\cos(2\pi f_0 t)\}(f) &= \frac{1}{2}(\delta(f+f_0) + \delta(f-f_0)) \\ \mathcal{F}\{\text{rect}_{(0,T)}(t)\}(f) &= \frac{1}{f_0} e^{-j\pi f/f_0} \text{sinc} \frac{f}{f_0} \\ X(f) &= \mathcal{F}\{x(t)\}(f) = \mathcal{F}\{\cos(2\pi f_0 t) \cdot \text{rect}_{(0,T)}(t)\}(f) \\ &= \mathcal{F}\{\cos(2\pi f_0 t)\}(f) * \mathcal{F}\{\text{rect}_{(0,T)}(t)\}(f) \\ &= \frac{1}{2f_0} \left(e^{-j\pi(f+f_0)/f_0} \text{sinc} \frac{f+f_0}{f_0} + e^{-j\pi(f-f_0)/f_0} \text{sinc} \frac{f-f_0}{f_0} \right) \end{aligned}$$

Parseval's theorem:

$$\int_{-\infty}^{\infty} |x(t)|^2 dt = \int_{-\infty}^{\infty} |X(f)|^2 df$$

Note that the support of x is $0 \leq t \leq T$ and the support of X is $-f_s/2 \leq f \leq f_s/2$, so we only have to integrate over these intervals.

$$\int_0^T |x(t)|^2 dt = \int_{-f_s/2}^{f_s/2} |X(f)|^2 df$$

I was able to calculate the energy of the signal analytically ($\int_0^T |x(t)|^2 dt = T/2$), but the integration of the latter was messy. Using MATLAB, I was able to check that the right hand side of the equation was also equal to $T/2$ by numerical integration (using `trapz`). With the parameters given in (6), the value of both integrals is equal to $T/2 = 0.25$. (Note: if we had integrated w.r.t. ω rather than f , there would be a scaling factor of $\frac{1}{2\pi}$.)

6. Suppose we pass $x(t)$ from the previous question through the system from question 4. Use MATLAB to find the amplitude and phase of the output signal's Fourier transform $Y(\omega)$. Plot $X(\omega)$ as well – how do the signals compare?

Parameters used:

- N (samples) = 1000

- T (sample window) = 10
- f_s (sampling frequency) = $N/T = 100$
- f_0 (cosine frequency) = 2
- T_0 (cosine period) = $f_0^{-1} = 0.5$

See (Figure 1) for the plots. Comparison of plots:

- As expected from the convolution of a sinc wave with two shifted deltas, $|X(f)|$ looks like it has two symmetric sinc-like waves centered at $\approx \pm f_0$. $|X(0)| = 0$, since there is no DC offset ($x(t)$ has an average value of 0).
- The plot of $|Y(f)|$ has a peak at $f = 0$ (DC), which makes sense now that the signal is purely positive. It has its two next-highest peaks at $\approx \pm 2f_0$, since $|\cos(\omega t)|$ looks somewhat like a sinusoid with double the frequency.
- The phase has changed from linearly increasing in $X(f)$ to linearly decreasing in $Y(f)$.

7. Let $z(t) = y(t)\cos(64\pi t + \theta)$. Write z as a sum of in-phase and quadrature components. Plot the Fourier transform $Z(\omega)$ for $\theta = \pi/3$, and comment on the effect of the modulation on amplitude and phase (compared to $Y(\omega)$).

$$\begin{aligned} z(t) &= y(t)\cos(64\pi t + \theta) \\ &= y(t)(\cos(\theta)\cos(64\pi t) - \sin(\theta)\sin(64\pi t)) \\ &= [y(t)\cos\theta]\cos(64\pi t) - [y(t)\sin\theta]\sin(64\pi t) \\ &= z_i(t)\cos(64\pi t) - z_q(t)\sin(64\pi t) \end{aligned}$$

(In phasor form, this is equivalent to $Z = [y(t)\cos\theta] + j[y(t)\sin\theta]$.)

(See (Figure 1) for the plots of $z(t)$ and $Z(f)$.) Notes on amplitude and phase:

- The magnitude plot of $|Z(f)|$ appears like the magnitude plot of $Y(f)$, duplicated with one centered at $\pm 32\text{Hz}$ (the carrier frequency, as expected).
 - The magnitude values in $|Y(f)|$ are about twice the values of the corresponding waveforms in $|Z(f)|$.
 - There is hardly any noticeable change in the phase plot.
8. Write a function that takes as an input a time-domain signal and outputs the Hilbert transform of that signal (also in the time-domain). Plot the Hilbert transforms of x , y and z in the frequency and time domains.

```
function res = hilbertTransform(x)
    X = fft(x);
    len = size(x, 2);
    % get signum of frequency;
    % +1 for (0, pi), -1 for (-pi, 0)
    sgn = [ones(1,floor(len/2)), -1*ones(1,ceil(len/2))];
    res = ifft(-1j * sgn .* X);
end
```

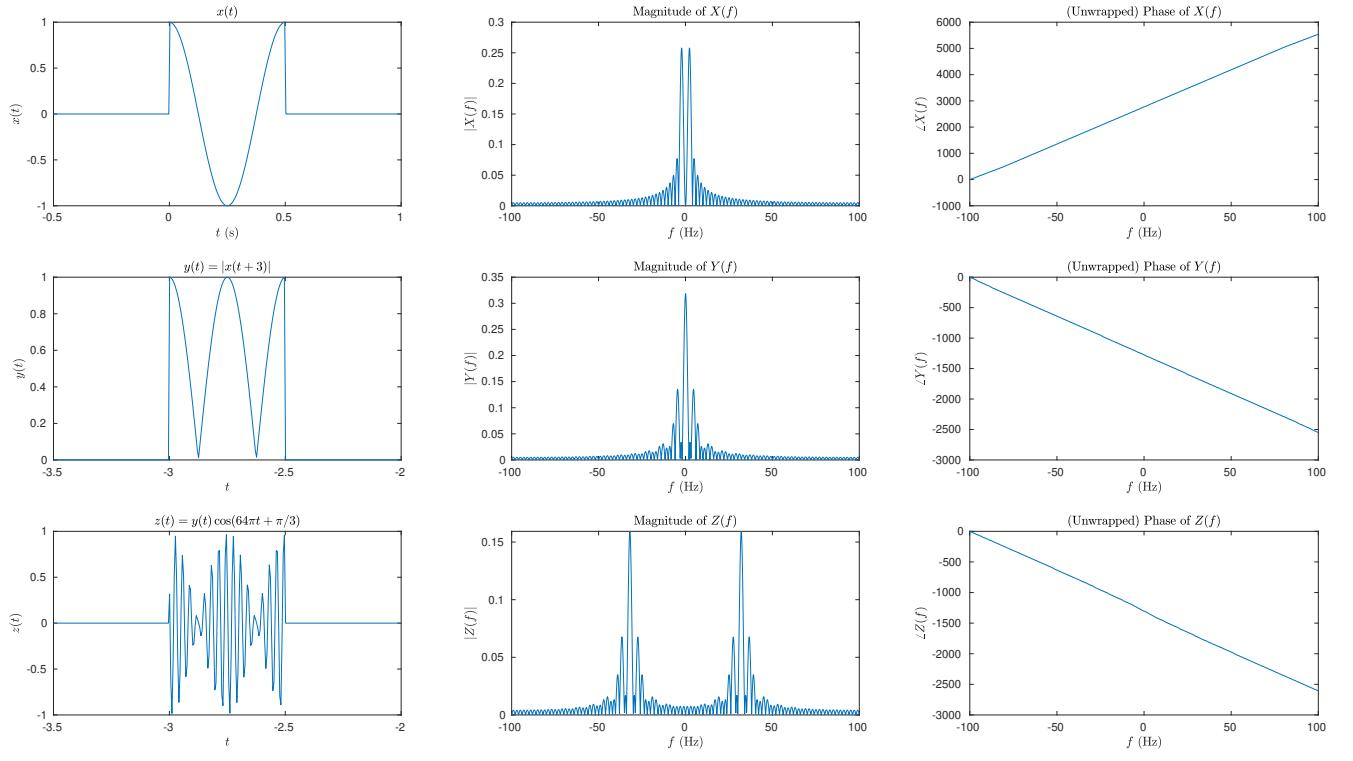


Figure 1: Plots of $x(t)$, $y(t)$, $z(t)$, and their Fourier transforms. See the parameters in (6).

See (Figure 2) for plots of the signals and their Hilbert transforms.

9. Using MATLAB, demonstrate the orthogonality of a signal and its Hilbert transform for all of x , y and z .

`trapz` was used to estimate the integral for the inner product:

$$\langle x_1, x_2 \rangle = \int_{-\infty}^{\infty} x_1(t)x_2^*(t) dt$$

The limits for t were $[-T, T]$, as it was for the previous parts of this problem set. As expected, the inner products were all close to zero:

$$\begin{aligned} |\langle x, \hat{x} \rangle| &\approx 6.255 \times 10^{-9} \\ |\langle y, \hat{y} \rangle| &\approx 1.014 \times 10^{-2} \\ |\langle z, \hat{z} \rangle| &\approx 1.624 \times 10^{-8} \end{aligned}$$

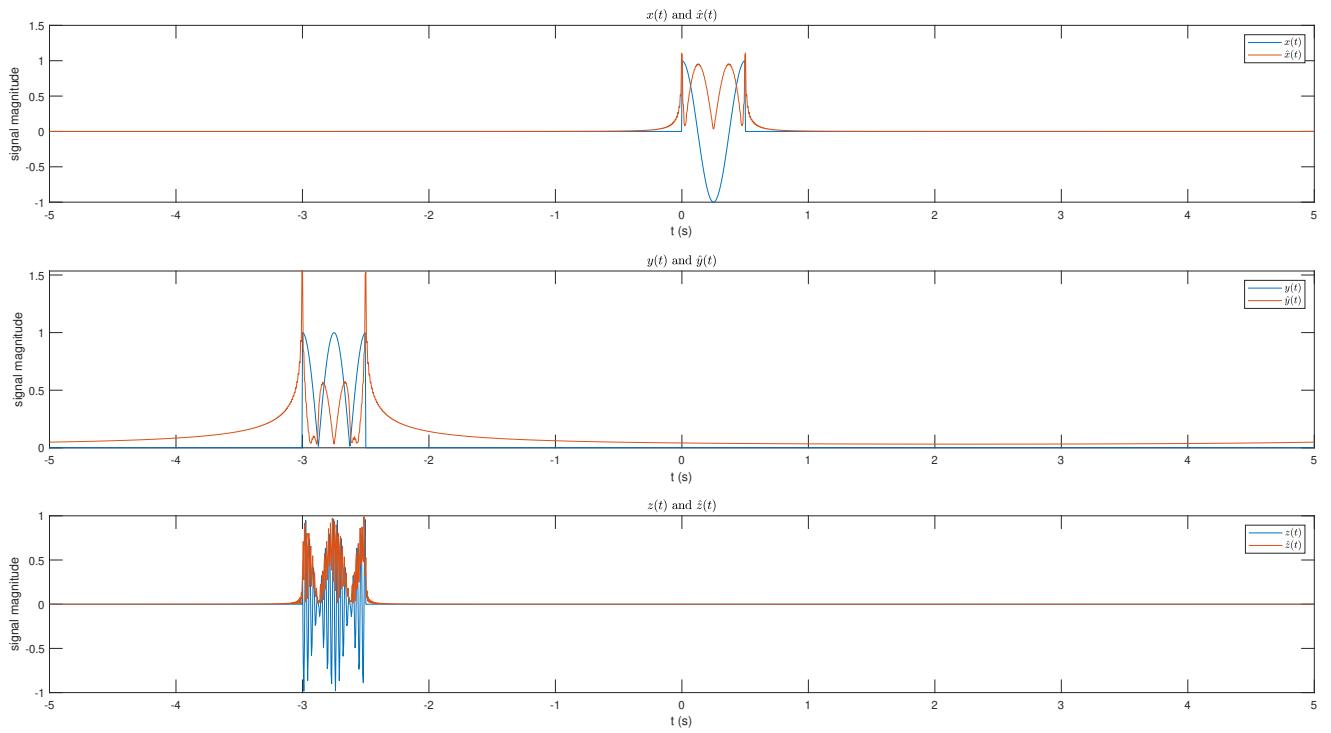


Figure 2: Plots of $x(t)$, $y(t)$, $z(t)$, and their Hilbert transforms.

Full MATLAB code:

```
% PSET1
% Jonathan Lam
% Prof. Frost
% ECE300
% Communications Theory
% 9/7/20

clear; close all; clc;
set(0, 'defaultTextInterpreter', 'latex');

% Q5

% sample details
N = 2000;
T = 10;
```

```

fs = N/T;

% sinusoid details
f0 = 2;
T0 = 1/f0;

% generate x(t)
t = linspace(-T/2, T/2, N);
x1 = cos(2*pi*f0*t);
x2 = rectangularPulse(0, T0, t);
x = x1 .* x2;

% plot x(t) from -T0 to 2*T0
figure();
subplot(3, 3, 1);
plot(t, x);
xlabel('$t$ (s)');
ylabel('$x(t)$');
title('$x(t)$');
xlim([-T0, 2*T0]);

% plot X(f) (magnitude and phase) for entire Nyquist bandwidth
X = fftshift(fft(x)/fs);
wd = linspace(-pi, pi, N);
f = wd * fs / (2 * pi);
subplot(3, 3, 2);
plot(f, abs(X));
xlabel('$f$ (Hz)');
ylabel('$|X(f)|$');
title('Magnitude of $X(f)$');
subplot(3, 3, 3);
plot(f, unwrap(angle(X)));
xlabel('$f$ (Hz)');
ylabel('$\angle X(f)$');
title('(Unwrapped) Phase of $X(f)$');

% compare x norm, X norm (they are equal)
% these both print out 0.25 (T/2) for the given parameters
fprintf('||x||=%d\n||X||=%d\n', trapz(t, abs(x).^2), trapz(f, abs(X).^2));

% Q6

% recompute x, shifted, and compute and plot y(t) from -3-T0 to -3+2*T0
x1Shifted = cos(2*pi*f0*(t+3));
x2Shifted = rectangularPulse(0, T0, (t+3));
xShifted = x1Shifted .* x2Shifted;

```

```

y = abs(xShifted);
subplot(3, 3, 4);
plot(t, y);
xlabel('$t$');
ylabel('$y(t)$');
title('$y(t)=|x(t+3)|$');
xlim([-3-T0, -3+2*T0]);

% plot Y(f) (magnitude and phase) for entire Nyquist bandwidth
Y = fftshift(fft(y)/fs);
subplot(3, 3, 5);
plot(f, abs(Y));
xlabel('$f$ (Hz)');
ylabel('$|Y(f)|$');
title('Magnitude of $Y(f)$');
subplot(3, 3, 6);
plot(f, unwrap(angle(Y)));
xlabel('$f$ (Hz)');
ylabel('$\angle Y(f)$');
title('(Unwrapped) Phase of $Y(f)$');

% q7

% compute z(t) and plot on same x-axis as y(t)
z = y .* cos(64*pi*t + pi/3);
subplot(3, 3, 7);
plot(t, z);
xlabel('$t$');
ylabel('$z(t)$');
title('$z(t)=y(t)\cos(64\pi t+\pi/3)$');
xlim([-3-T0, -3+2*T0]);

% compute and plot Z(f)
Z = fftshift(fft(z)/fs);
subplot(3, 3, 8);
plot(f, abs(Z));
xlabel('$f$ (Hz)');
ylabel('$|Z(f)|$');
title('Magnitude of $Z(f)$');
subplot(3, 3, 9);
plot(f, unwrap(angle(Z)));
xlabel('$f$ (Hz)');
ylabel('$\angle Z(f)$');
title('(Unwrapped) Phase of $Z(f)$');

% q8-9

```

```

innerProd = @(t, f, g) trapz(t, f .* conj(g));
figure();
signals = [x; y; z];
labels = ['x' 'y' 'z'];
for i = 1:3
    signal = signals(i, :);
    subplot(3, 1, i);
    plot(t, signal, t, abs(hilbertTransform(signal)));
    xlabel('t (s)');
    ylabel('signal magnitude');
    legend([string(sprintf('$%c(t)$', labels(i))), ...
        string(sprintf('$\hat{ %c}(t)$', labels(i)))), ...
        'interpreter', 'latex');
    title(sprintf('$%c(t)$ and $\hat{ %c}(t)$', labels(i), labels(i)), ...
        'interpreter', 'latex');

    % this should be fairly small (approximately 0) because the signals
    % and their hilbert transforms should be orthogonal
    fprintf('|<%c(t),\hat{ %c}(t)>|= %d\n', labels(i), labels(i), ...
        abs(innerProd(t, signal, hilbertTransform(signal))));
end

% Q8
function res = hilbertTransform(x)
X = fft(x);
len = size(x, 2);
% get signum of frequency; +1 for 0 to pi, -1 for -pi to 0
sgn = [ones(1, floor(len/2)), -1*ones(1, ceil(len/2))];
res = ifft(-1j * sgn .* X);
end

```

MATLAB text output:

```

||x||=2.502502e-01
||X||=2.502502e-01
|<x(t),\hat{x}(t)>|=6.255166e-09
|<y(t),\hat{y}(t)>|=1.014327e-02
|<z(t),\hat{z}(t)>|=1.624092e-08

```

ECE300 – Pset 2

Jonathan Lam

September 15, 2020

Figures are located at the end of the document.

Load a short audio clip (either one you record, one from online or one built in to MATLAB) into MATLAB and call this signal $m(t)$. You will explore noise-free AM strategies using $m(t)$ as your message signal. Scale the message so that the maximum, in absolute value, is 1.

```
clear; close all; clc;
set(0, 'defaultTextInterpreter', 'latex');

% Most variable names should be fairly clear, but the following
% abbreviations are made for commonly-used meanings:
% - sig:      signal (in time domain)
% - ft:       fourier transform (in freq. domain)
% - smp:      relating to the raw audio sample data
% - am:       relating to the AM-modulated wave
% - us/ds:    upsampled/downsampled
% - cv:        conventional AM
% - rcv:      rectified conventional AM
% - lrcv:     low-pass filtered rectified conventional AM

pset_name = 'ece300_pset2';           % for exporting figures as PDFs

% audio file parameters
file_smp = 'preamble.wav';           % filename of audio input file
                                      % audio source: [2]
file_out = 'preambleAM.wav';          % filename of audio output file
dur_smp = 5;                         % truncate audio file to this duration
                                      % in seconds to speed up program
W = 5000;                           % approximate bandwidth of audio;
                                      % 5kHz is typical in AM radio [1]

% AM-modulation parameters
fc_am = 600000;                     % AM carrier frequency; tend to range
                                      % from 550 to 1720kHz [1]
fs_am = 2000000;                    % sampling frequency (has to be >=2*Fc)

% loads audio signal into smp, and sets the variable Fs_smp to be the
% audio file's sampling rate
[sig_smp, fs_smp] = audioread(file_smp);
sig_smp = sig_smp(1:round(fs_smp * dur_smp));   % truncate audio file
N = length(sig_smp);                  % number of samples

% message signal should have |m(t)| <= 1
max_amplitude_smp = max(sig_smp);
```

```

sig_m = sig_smp / max_amplitude_smp;

% samplesT are the time values at which y were sampled
t_smp = linspace(0, dur_smp, N);

```

1. Plot $m(t)$ in time and its Fourier transform $M(\omega)$ (both amplitude and phase). What is the bandwidth of your message?

```

figure('visible', 'off');
subplot(3, 1, 1);
plot(t_smp, sig_m);
ylabel('$m(t)$');
xlabel('$t$ (s)');
title('Baseband $m(t)$ (not upsampled)');

wd = linspace(-pi, pi, N); % these three lines for generating
f_smp = wd * fs_smp / (2 * pi); % the FFT plots
ft_m = fftshift(fft(sig_m)) / fs_smp;

subplot(3, 1, 2);
plot(f_smp, abs(ft_m));
ylabel('$|\mathcal{F}\{m\}(f)|$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\{m\}(f)$');
subplot(3, 1, 3);
plot(f_smp, unwrap(angle(ft_m)));
ylabel('$\angle\{\mathcal{F}\{m\}(f)\}$');
xlabel('$f$ (Hz)');
title('Unwrapped Phase of $\mathcal{F}\{m\}(f)$');

```

Most of the information seems to have a frequency less than $\approx 7\text{kHz}$, so this is a reasonable value for the bandwidth (the valuable is also reasonable given that the input is a calm speaking voice). It seems that there's some information in the $7\text{kHz}-10\text{kHz}$ range, but its amplitude doesn't seem very significant. Past 10kHz the signal seems to have trivial frequency components.

(Despite this, I've approximated limited the bandwidth W to be 5kHz , since according to [1] this is a common value for the upper modulating frequency in AM. This is also the cutoff frequency used in the LPF for the conventional AM.)

2. Generate a carrier signal over the same timespan to be used for DSB-SC AM and DSB AM. There is no one signal that will work here, but try to make it realistic!

```

% generate carrier frequency at  $F_c$ , and sample it at  $F_{s\_am}$  (which should
% be  $\geq 2 \cdot F_c$ )
t_am = linspace(0, dur_smp, dur_smp * fs_am);
N_am = length(t_am);
sig_carrier = cos(2 * pi * fc_am * t_am);

% upsample sample to  $F_{s\_am}$  (same sampling frequency as for the carrier)
sig_m_us = interp1(t_smp, sig_m, t_am);

```

3. Generate the DSB-SC and DSB modulated signals with message $m(t)$ and plot them in the time and frequency domains. In one sentence, describe the difference.

```

sig_dsbsc = sig_carrier .* sig_m_us;

% plot DSB-SC signal in time domain
figure('visible', 'off');
subplot(3, 2, 1);
plot(t_am, sig_dsbsc);
ylabel('$dsbsc\{m\}(t)$');
xlabel('$t$ (s)');
title('$dsbsc\{m\}(t)$');

% plot DSB-SC signal in frequency domain
wd = linspace(-pi, pi, N_am);
f_am = wd * fs_am / (2 * pi);
ft_dsbsc = fftshift(fft(sig_dsbsc)) / fs_am;
subplot(3, 2, 3);
plot(f_am, abs(ft_dsbsc));
ylabel('$|\mathcal{F}\{dsbsc\{m\}\}(f)|$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\{dsbsc\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);
subplot(3, 2, 5);
plot(f_am, unwrap(angle(ft_dsbsc)));
ylabel('$\angle(\mathcal{F}\{dsbsc\{m\}\}(f))$');
xlabel('$f$ (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\{dsbsc\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);

% generate DSB signal, plot in time domain
% a small pilot amplitude Ap was used here so that the original frequency
% spectrum would still be visible in the fourier transform; if Ap = 1, for
% example, then it would look like the conventional AM case (just two
% deltas, which is less informative)
Ap_dsb = 0.05;
sig_dsb = sig_dsbsc + Ap_dsb * sig_carrier;
ft_dsb = fftshift(fft(sig_dsb)) / fs_am;
subplot(3, 2, 2);
plot(t_am, sig_dsb);
ylabel('$dsb\{m\}(t)$');
xlabel('$t$ (s)');
title('$dsb\{m\}(t)$');

% plot DSB signal in frequency domain
subplot(3, 2, 4);
plot(f_am, abs(ft_dsb));
ylabel('$|\mathcal{F}\{dsb\{m\}\}(f)|$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\{dsb\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);
subplot(3, 2, 6);
plot(f_am, unwrap(angle(ft_dsb)));
ylabel('$\angle(\mathcal{F}\{dsb\{m\}\}(f))$');
xlabel('$f$ (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\{dsb\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);

```

There is a much higher peak in the Fourier transform of the DSB signal than in that of the DSB-SC signal (which makes the former less power-efficient but phase-coherent.)

4. Using the same carrier frequency, generate lower SSB and upper SSB AM signals, and plot them in time and frequency.

```
% from previous problem set: take hilbert transform of a signal
function res = hilbertTransform(sig_x)
    ft_x = fft(sig_x);
    len_x = length(sig_x);
    % get signum of frequency; +1 for 0 to pi, -1 for -pi to 0
    sgn = [ones(1, floor(len_x/2)), -1*ones(1, ceil(len_x/2))];
    res = ifft(-1j * sgn .* ft_x);
end

% generate lssb, ussb AM-modulated signals
sig_mhat = hilbertTransform(sig_m_us);
sig_quad = sig_mhat .* sin(2 * pi * fc_am * t_am); % quadrature component
sig_lssb = (sig_dsb_sc + sig_quad) / 2;
sig_ussb = (sig_dsb_sc - sig_quad) / 2;

% plot lssb in time domain
figure('visible', 'off');
subplot(3, 2, 1);
plot(t_am, real(sig_lssb));
ylabel('$lssb\{m\}(t)$');
xlabel('$t$ (s)');
title('$lssb\{m\}(t)$');

% plot lssb in frequency domain
ft_lssb = fftshift(fft(sig_lssb)) / fs_am;
subplot(3, 2, 3);
plot(f_am, abs(ft_lssb));
ylabel('$|\mathcal{F}\{lssb\{m\}\}(f)|$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\{lssb\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);
subplot(3, 2, 5);
plot(f_am, unwrap(angle(ft_lssb)));
ylabel('$\angle(\mathcal{F}\{lssb\{m\}\}(f))$');
xlabel('$f$ (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\{lssb\{m\}\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);

% plot ussb in time domain
subplot(3, 2, 2);
plot(t_am, real(sig_ussb));
ylabel('$ussb\{m\}(t)$');
xlabel('$t$ (s)');
title('$ussb\{m\}(t)$');

% plot ussb in frequency domain
ft_ussb = fftshift(fft(sig_ussb)) / fs_am;
subplot(3, 2, 4);
plot(f_am, abs(ft_ussb));
ylabel('$|\mathcal{F}\{ussb\{m\}\}(f)|$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\{ussb\{m\}\}(f)$');
```

```

    xlim([-fc_am * 1.25, fc_am * 1.25]);
    subplot(3, 2, 6);
    plot(f_am, unwrap(angle(ft_ussb)));
    ylabel('$\angle(\mathcal{F}\{m\}(f))$');
    xlabel('f (Hz)');
    title('(Unwrapped) Phase of $\mathcal{F}\{m\}(f)$');
    xlim([-fc_am * 1.25, fc_am * 1.25]);

```

5. Using the same carrier frequency, create a conventional AM signal (ensure you choose the amplitude of the carrier correctly). Plot this signal in the time domain and frequency domain as well.

```

% this is pretty much the same as DSB AM, but with a different pilot
% amplitude
sig_cv = (1 + sig_m_us) .* cos(2 * pi * fc_am * t_am);

% plot conventional AM signal in time domain
figure('visible', 'off');
subplot(3, 3, 1);
plot(t_am, sig_cv);
ylabel('$cv(m)(t)$');
xlabel('t (s)');
title('Conventional AM Modulation');

% plot conventional AM signal in frequency domain
ft_cv = fftshift(fft(sig_cv)) / fs_am;
subplot(3, 3, 4);
plot(f_am, abs(ft_cv));
ylabel('$|\mathcal{F}\{cv(m)\}(f)|$');
xlabel('f (Hz)');
title('Magnitude of $\mathcal{F}\{cv(m)\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);
subplot(3, 3, 7);
plot(f_am, unwrap(angle(ft_cv)));
ylabel('$\angle(\mathcal{F}\{cv(m)\}(f))$');
xlabel('f (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\{cv(m)\}(f)$');
xlim([-fc_am * 1.25, fc_am * 1.25]);

```

6. For the conventional AM signal, do demodulate we rectify and then apply a low-pass filter. In MATLAB, you should be able to rectify the signal in one line. Plot the rectified signal in time and frequency. Then, design a first-order low-pass filter (you decide the cutoff frequency) and apply it to the signal. You can do this using MATLAB filter functions, or through convolution in time/multiplication in frequency directly. Plot the resultant output signal in time and frequency and play the signal as audio.

```

% rectify signal, subtract 1; estimate rectification with abs()
sig_rcv = abs(sig_cv);

% low pass in frequency domain:
% first-order LPF has transfer function H(f) = (1 + j*(f/fc))^{-1}
% choose cutoff frequency fc = W (original bandwidth)
% also, there seems to be some low-frequency periodic error when rectifying
% using abs() (i.e., the sample x-intercepts are not exactly correct), so

```

```

% I added an additional HPF here with fc = 20Hz
fc_lpf = W; % LPF cutoff freq. at W
fc_hpf = 20; % HPF cutoff freq. at 20Hz
lpf = (1 + f_am/fc_lpf*1j).^-1; % LPF
hpf = (1 - fc_hpf*f_am.^-1*1j).^-1; % HPF
ft_rcv = fftshift(fft(sig_rcv)) / fs_am; % rect. sig. in freq. domain
ft_lrcv = hpf .* lpf .* ft_rcv; % apply filters in freq. domain
sig_lrcv = ifft(ifftshift(ft_lrcv) * fs_am);

% plot rectified/shifted conventional signal in time domain
subplot(3, 3, 2);
plot(t_am, sig_rcv);
ylabel('$|cv\{m\}|(t)-1$');
xlabel('$t$ (s)');
title('Rectified, shifted conventional AM');

% plot rectified/shifted conventional signal in frequency domain
subplot(3, 3, 5);
plot(f_am, abs(ft_rcv));
ylabel('$\|\mathcal{F}\|_{cv\{m\}}^{-1}(f)$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\|_{cv\{m\}}^{-1}(f)$');
 xlim([-2*fc_am * 1.25, 2*fc_am * 1.25]);
 subplot(3, 3, 8);
plot(f_am, unwrap(angle(ft_rcv)));
ylabel('$\angle(\mathcal{F}\|_{cv\{m\}}^{-1}(f))$');
xlabel('$f$ (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\|_{cv\{m\}}^{-1}(f)$');
 xlim([-fc_am * 1.25, fc_am * 1.25]);

% plot rectified, shifted, filtered conventional AM signal in time domain
subplot(3, 3, 3);
plot(t_am, real(sig_lrcv));
ylabel('$lpf\{hpf\|_{cv\{m\}}^{-1}\}(t)$');
xlabel('$t$ (s)');
title('Rectified, shifted, filtered conventional AM');

% plot rectified, shifted, filtered conventional AM signal in freq. domain
subplot(3, 3, 6);
plot(f_am, abs(ft_lrcv));
ylabel('$\|\mathcal{F}\|_{lpf\{hpf\|_{cv\{m\}}^{-1}\}}(f)$');
xlabel('$f$ (Hz)');
title('Magnitude of $\mathcal{F}\|_{lpf\{hpf\|_{cv\{m\}}^{-1}\}}(f)$');
 xlim([-W * 1.25, W * 1.25]);
 subplot(3, 3, 9);
plot(f_am, unwrap(angle(ft_lrcv)));
ylabel('$\angle(\mathcal{F}\|_{lpf\{hpf\|_{cv\{m\}}^{-1}\}}(f))$');
xlabel('$f$ (Hz)');
title('(Unwrapped) Phase of $\mathcal{F}\|_{lpf\{hpf\|_{cv\{m\}}^{-1}\}}(f)$');
 xlim([-W * 1.25, W * 1.25]);

% reconstruct original audio signal, should approximately be sig_m:
% 1. downsample to original sampling frequency fs_smp (at times t_smp)
% (MATLAB cannot use such a high sample rate for audio anyways [3])
% 2. take the real part (filter produces complex wave)
% 3. re-scale to initial volume
sig_lrcv_ds = real(interp1(t_am, sig_lrcv, t_smp)) .* max_amplitude_smp;

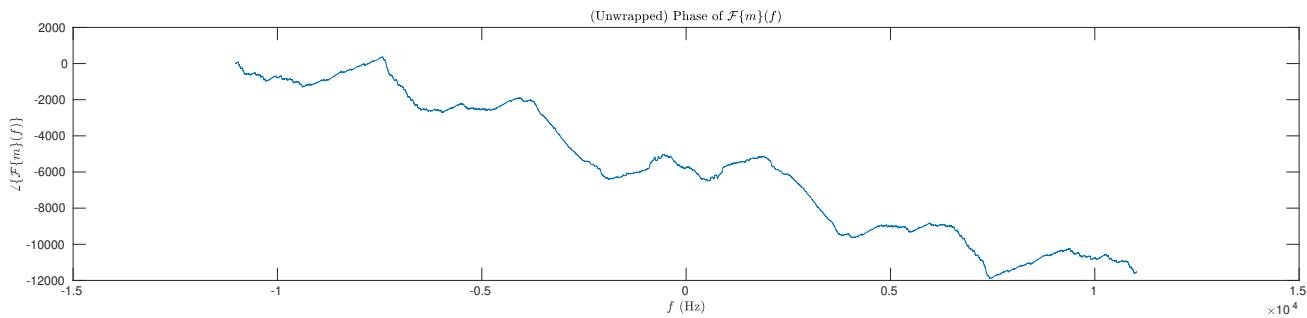
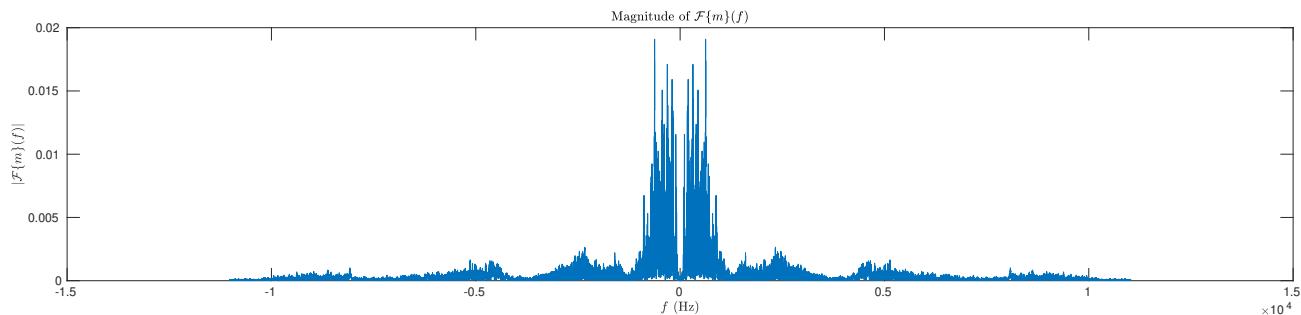
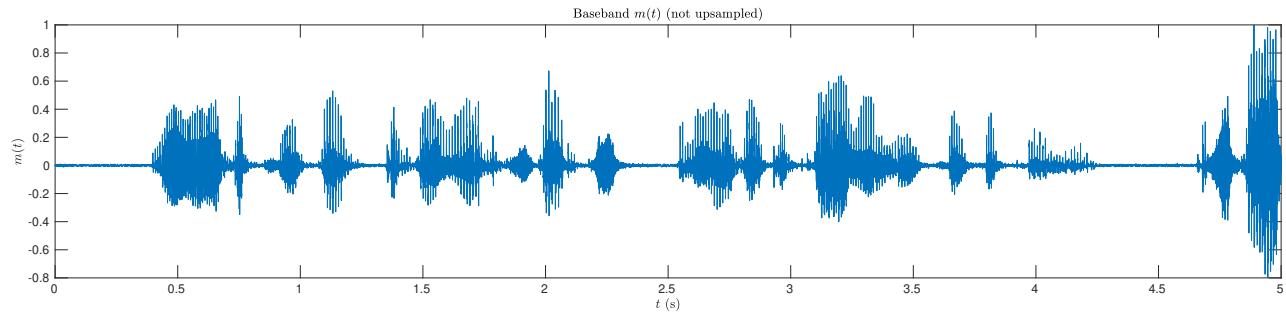
```

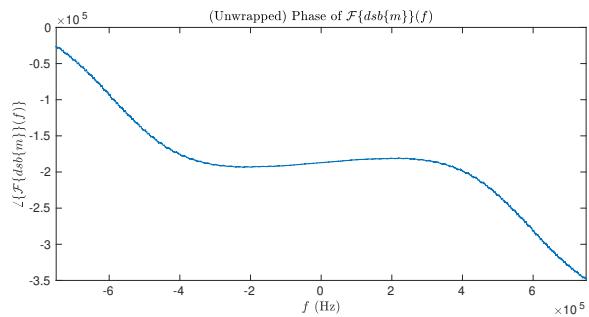
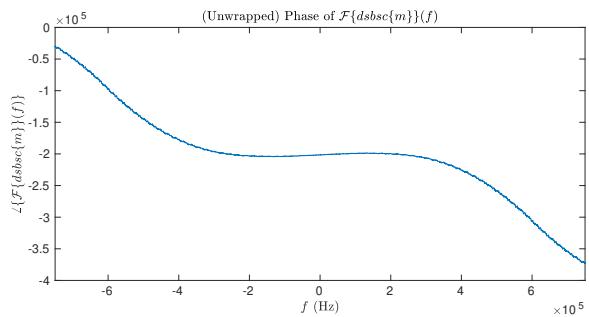
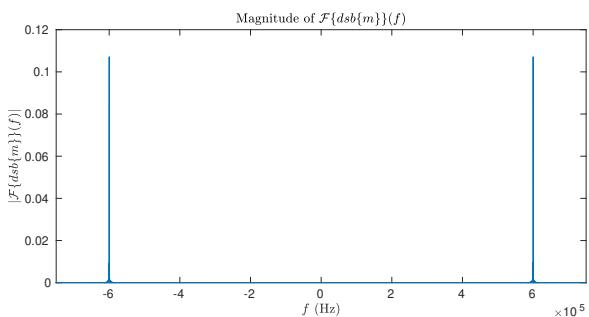
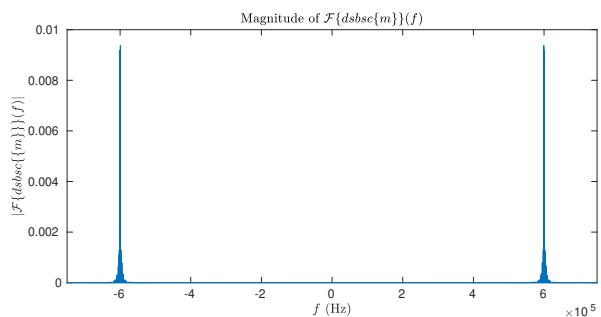
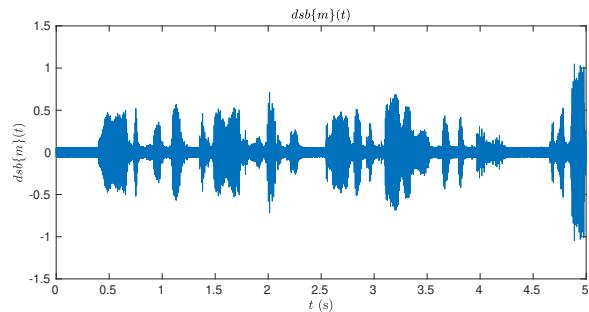
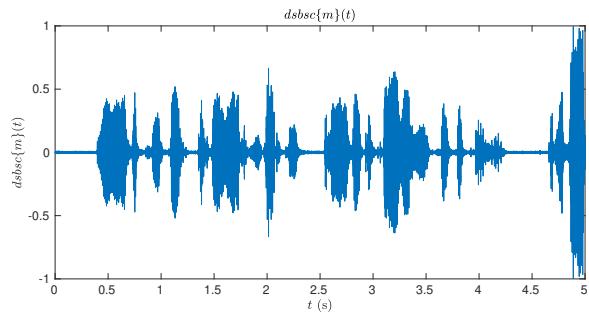
```
% play new audio signal and save to file_out
sound(sig_lrcv_ds, fs_smp);
audiowrite(file_out, sig_lrcv_ds, fs_smp);
```

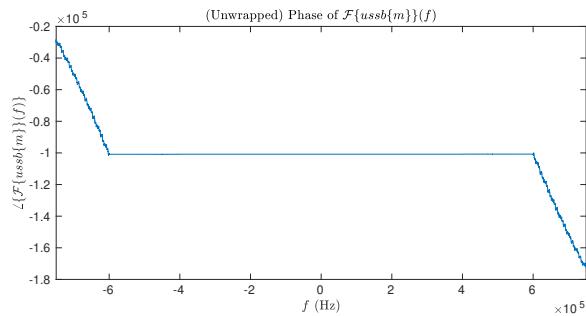
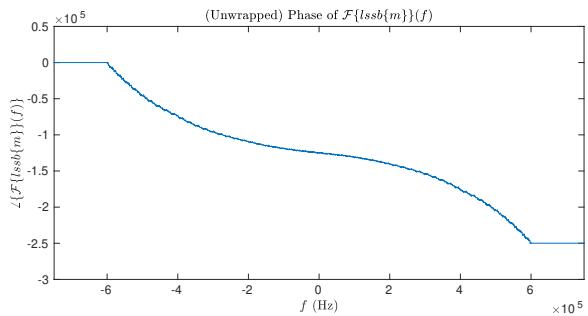
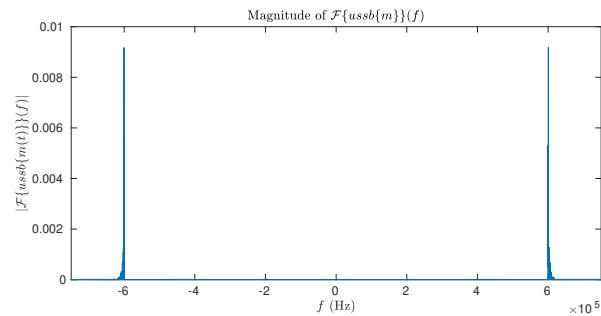
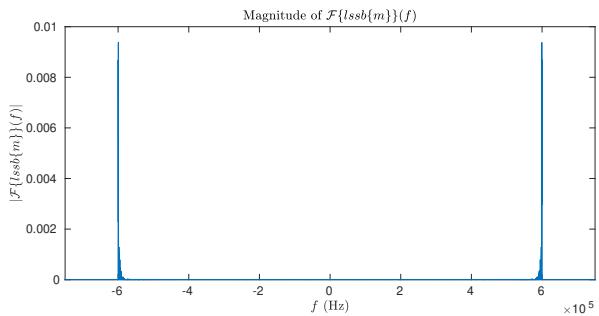
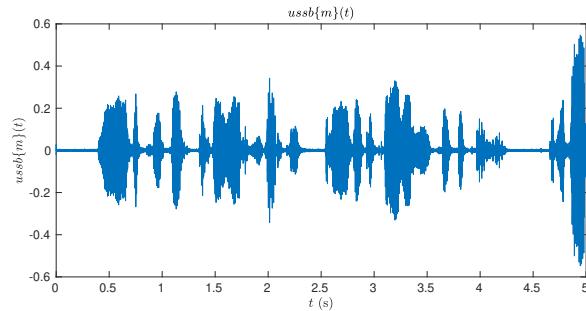
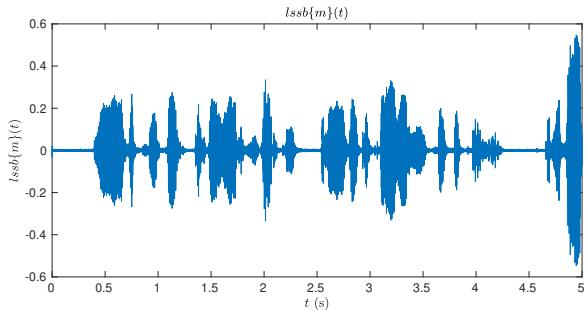
It is clear from the figure (“Rectified, shifted conventional AM”) that there is some distortion caused by rectification. If the rectification were perfect, the bottom line would be flat. However, due to limitations in machine accuracy and the periodic nature of the signal and sampling, there is a resulting low-frequency distortion that I had to filter out with a HPF. I chose 20Hz as the cutoff frequency for this filter, since that is approximately the lowest frequency we can hear, and it removes most but not all of the distortion; there is still a spike in the Fourier transform of the signal near $f = 0$ that was not present in the original signal. The resultant audio quality is not terrible.

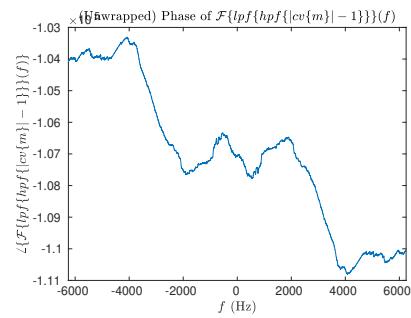
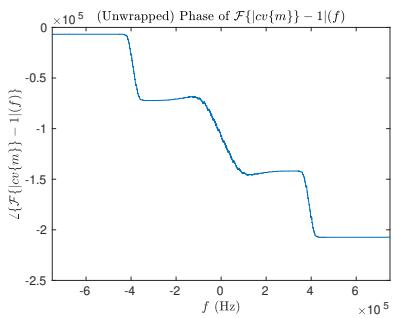
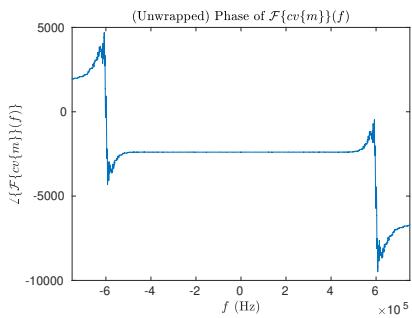
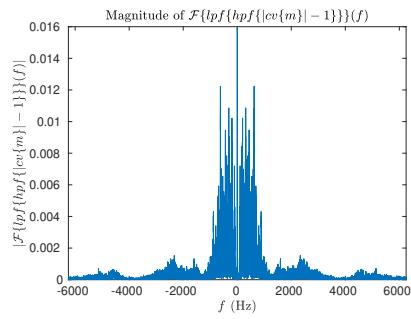
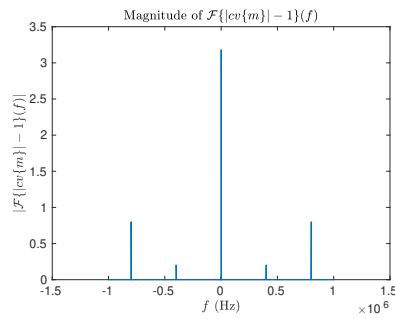
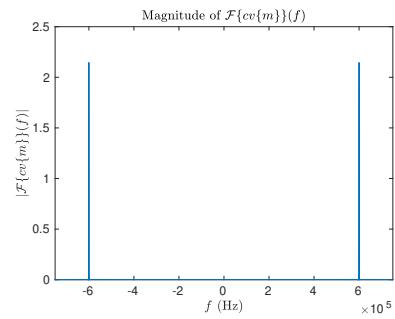
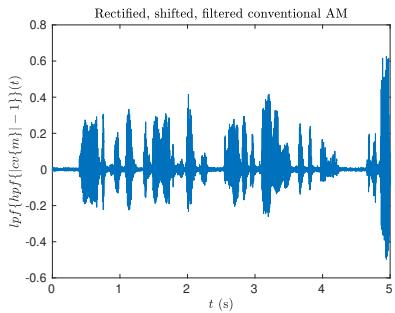
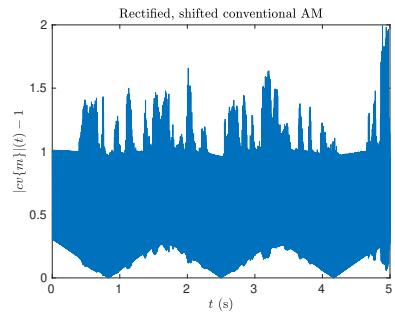
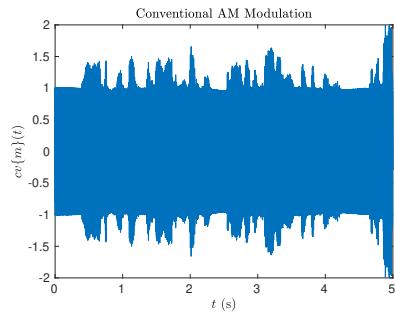
Cited sources

1. <https://fas.org/man/dod-101/navy/docs/es310/AM.htm>
2. <https://www2.cs.uic.edu/~i101/SoundFiles/>
3. <https://www.mathworks.com/matlabcentral/answers/195697-what-is-the-maximum-sample-rate-supported-by-sound>)









ECE300 – Pset 3

Jonathan Lam

September 26, 2020

1. A source generates 0s and 1s randomly according to a Bernoulli distribution, where the probability of a 0 is 0.3 and the probability of a 1 is 0.7. The value is then sent across a long wire, and corrupted by thermal noise. A system at the other end receives the corrupted signal and guesses if it received a 1 or a 0. It has an error probability (either guessing a 1 was a 0 or guessing a 0 was a 1) of 0.2. If the receiver guesses it received a 1, what is the probability a 1 was transmitted?

Let T represent the transmitted (true) value, and G represent the received (guessed) value.

$$\begin{aligned}\Pr(T = 0) &= 0.3 \\ \Pr(T = 1) &= 0.7 \\ \Pr(G = 1|T = 0) &= \Pr(G = 0|T = 1) = 0.2\end{aligned}$$

Use Bayes' Theorem (where $T = \{0, 1\}$ is the partition):

$$\begin{aligned}\Pr(T = 1|G = 1) &= \frac{\Pr(T = 1 \cap G = 1)}{\Pr(G = 1)} = \frac{\Pr(G = 1|T = 1)\Pr(T = 1)}{\sum_{i=0}^1 \Pr(G = 1|T = i)\Pr(T = i)} \\ &= \frac{(1 - 0.2)(0.7)}{(0.2)(0.3) + (1 - 0.2)(0.7)} = \frac{28}{31}\end{aligned}$$

2. Let θ be a random variable uniformly distributed from 0 to π . Let $X = \cos \theta$ and $Y = \sin \theta$. Are X and Y uncorrelated? Are they independent? Are they orthogonal?

$$\begin{aligned} f_\theta(\tau) &= \frac{1}{\pi} & (\tau \in [0, \pi]) \\ \mu_X &= E[X] = \int_0^\pi X(\theta = \tau) f_\theta(\tau) d\tau = \frac{1}{\pi} \int_0^\pi \cos \tau d\tau = 0 \\ \mu_Y &= E[Y] = \int_0^\pi Y(\theta = \tau) f_\theta(\tau) d\tau = \frac{1}{\pi} \int_0^\pi \sin \tau d\tau = \frac{2}{\pi} \\ r_{X,Y} &= E[XY] = \int_0^\pi X(\theta = \tau) Y(\theta = \tau) f_\theta(\tau) d\tau \\ &= \frac{1}{\pi} \int_0^\pi \cos \tau \sin \tau d\tau = 0 \end{aligned} \tag{1}$$

$$\text{cov}(X, Y) = E[XY] - \mu_X \mu_Y = 0 - (0) \left(\frac{2}{\pi} \right) = 0 \tag{2}$$

$$\begin{aligned} f_X(x) &= \frac{d}{dx} F_X(x) = \frac{d}{dx} \Pr(\cos \theta < x) = \frac{d}{dx} \Pr(\theta > \arccos x) = \\ &= \frac{d}{dx} \frac{1}{\pi} \int_{\arccos x}^\pi d\tau = \frac{1}{\pi} \frac{d}{dx} [\pi - \arccos x] = \frac{1}{\pi \sqrt{1-x^2}} \end{aligned} \tag{x \in [-1, 1]}$$

$$\begin{aligned} f_Y(y) &= \frac{d}{dy} F_Y(y) = \frac{d}{dy} \Pr(\sin \theta < y) = 2 \frac{d}{dy} \Pr(\theta < \arcsin y) = \\ &= \frac{d}{dy} \frac{2}{\pi} \int_0^{\arcsin y} d\tau = \frac{d}{dy} \frac{2}{\pi} [\arcsin y] = \frac{2}{\pi \sqrt{1-y^2}} \end{aligned} \tag{y \in [0, 1]}$$

$$f_{X,Y}(x, y) = \begin{cases} 1 & x^2 + y^2 = 1 \\ 0 & \text{else} \end{cases} \tag{x \in [-1, 1], y \in [0, 1]}$$

$$f_X(x)f_Y(y) \neq f_{X,Y}(x, y) \tag{3}$$

Uncorrelated (2), dependent (3), orthogonal (1). Orthogonality makes sense because \sin and \cos are orthogonal functions, dependence makes sense because X and Y rely on the same random variable (i.e., if you set X , then you fix the value of θ and thus also the value of Y), uncorrelatedness makes sense because an increase in X doesn't tend to cause an increase in Y .

3. Let $X(t)$ be a random process defined by $X(t) = A + Bt$ where A and B are independent random variables uniformly distributed from -1 to 1. Find $m_X(t)$ and $R_X(t_1, t_2)$. Is the process WSS? If not, is it cyclostationary? If the answer to either question is yes, find the PSD of X .

$$f_A(c) = f_B(c) = \frac{1}{2} \quad (c \in [-1, 1])$$

$$\mu_A = E[A] = \frac{1}{2} \int_{-1}^1 a da = 0 = \mu_B$$

$$m_X(t) = E[A + Bt] = E[A] + tE[B] = \mu_A + t\mu_B = 0 \quad (4)$$

$$E[A^2] = \frac{1}{2} \int_{-1}^1 a^2 da = \frac{1}{3} = E[B^2]$$

$$E[AB] = E[A]E[B] = 0 \quad (\text{by independence})$$

$$R_X(t_1, t_2) = E[X(t_1)X(t_2)] = E[(A + Bt_1)(A + Bt_2)] \quad (5)$$

$$= E[A^2 + AB(t_1 + t_2) + Bt_1t_2]$$

$$= E[A^2] + (t_1 + t_2)E[AB] + t_1t_2E[B^2]$$

$$= \frac{1}{3} + (0)(t_1 + t_2) + \frac{1}{3}t_1t_2 = \frac{1}{3}(1 + t_1t_2)$$

$$R'_X(t) = R_X(t + \tau, t) = \frac{1}{3}(1 + t^2 + \tau t) \quad (6)$$

The mean is not a function of time (4) but the autocorrelation is not a function of the delay $t_1 - t_2$ (5), so X is not WSS. If we express the autocorrelation as a function of the starting time $R'_X(t)$ (where $\tau = t_2 - t_1$ is the delay), we see that it is not periodic for any period, so it is not cyclostationary.

4. Let $X(t) = Y \cos(\omega_0 t) - Z \sin(\omega_0 t)$, where Y and Z are zero-mean independent gaussians with variance σ^2 . Find $m_X(t)$ and $R_X(t_1, t_2)$. Is the process WSS? If not, is it cyclostationary? If the answer to either question is yes, find the PSD of X .

$$\begin{aligned}
0 &= E[Y] = E[Z] \\
\sigma^2 &= E[(Y - \mu_Y)^2] = E[(Z - \mu_Z)^2] = E[Y^2] = E[Z^2] \\
m_X(t) &= E[X] = E[Y] \cos \omega_0 t + E[Z] \omega_0 t = 0 & (7) \\
E[YZ] &= E[Y]E[Z] = 0 & (\text{by independence}) \\
R_X(t_1, t_2) &= E[(Y \cos \omega_0 t_1 + Z \sin \omega_0 t_1)(Y \cos \omega_0 t_2 + Z \sin \omega_0 t_2)] & (8) \\
&= \cos(\omega_0 t_1) \cos(\omega_0 t_2) E[Y^2] + E[YZ] \cos(\omega_0 t_1) \sin(\omega_0 t_2) \\
&\quad + E[YZ] \sin(\omega_0 t_1) \cos(\omega_0 t_2) + E[Z^2] \sin(\omega_0 t_1) \sin(\omega_0 t_2) \\
&= \sigma^2 (\cos(\omega_0 t_1) \cos(\omega_0 t_2) + \sin(\omega_0 t_1) \sin(\omega_0 t_2)) + (0)(\dots) \\
&= \sigma^2 \cos(\omega_0(t_1 - t_2)) = R_X(t_2 - t_1) = R_X(\tau)
\end{aligned}$$

The mean is constant (7) and the autocorrelation is a function of time lag τ only (8), so it is WSS (and thus also cyclostationary).

$$S_X(\omega) = \mathcal{F}\{R_X(\tau)\}(\omega) = \frac{\sigma^2 \pi}{2} [\delta(\omega - \omega_0) + \delta(\omega + \omega_0)] \quad (\text{W-K Thm.})$$

5. If $X(t)$ has PSD $S_X(\omega)$, what is the PSD of $Y(t) = 4X'(t) + X(t-T)$? (Assume X is WSS.)

$$\begin{aligned}
S_Y(t) &= E[|Y(\omega)|^2] \\
&= E\left[\left|4j\omega X(\omega) + X(\omega)e^{-j\omega T}\right|^2\right] \\
&= E\left[|X(\omega)|^2 \left|4j\omega + e^{-j\omega T}\right|^2\right] \\
&= |4j\omega + e^{j\omega T}|^2 E[|X(\omega)|^2] \\
&= (4j\omega + e^{-j\omega T})(-4j\omega + e^{j\omega T}) S_X(t) \\
&= \left(16\omega^2 - 8\omega \left(\frac{e^{j\omega T} - e^{-j\omega T}}{2j}\right) + 1\right) S_X(t) \\
&= (16\omega^2 - 8\omega \sin \omega T + 1) S_X(t)
\end{aligned}$$

6. In the discrete-time case, suppose we have a random process defined by $\{X_n\}$ for $n \in \mathbb{Z}$. If we observe N samples of the random process, $\{x_n\}$ for $n = 1, 2, \dots, N$, we define the sample autocorrelation as

$$R_X(m) = \begin{cases} \frac{1}{N-m} \sum_{n=1}^{N-m} x_n x_{n+m}, & m = 0, 1, \dots \\ \frac{1}{N-|m|} \sum_{n=|m|+1}^N x_n x_{n+m}. & m = -1, -2, \dots \end{cases}$$

This approximates the autocorrelation. In practice, we don't let m take infinitely many values, but instead look at it over a finite range of values from $-M$ to M for some integer M . The Power Spectral Density is computed through the Wiener-Khinchin Theorem, using the DFT rather than the CTFT:

$$S_X(\omega) = \sum_{m=-M}^M R_X(m) \exp \frac{-j\omega m}{2M+1}$$

Using MATLAB, write a function that takes an input vector of N samples and an integer M as inputs, and returns the autocorrelation and PSD.

```
% X: 1xN
% m: scalar
% returns: scalar
function res = autocorr(X, m)
    m = abs(m);
    N = length(X);
    res = X(1:N-m) * X(m+1:N).'/ (N - m);
end

% X: 1xN
% M: scalar
% w: 1xW
% returns: 1xW
function res = psd(X, M, w)
    m = -M:M;
    res = arrayfun(@(m) autocorr(X, m), m) * exp(-1j * w .* m.' / (2 * M + 1));
end
```

Notes on implementation:

- Since R_X seems to be symmetric about $m = 0$, I wrote it without breaking up the cases.
- This `autocorr` implementation takes a signal and a scalar m , but the `psd` function can take an array of ω values over which it estimates the PSD. The reason for this difference is that while we can neatly broadcast the multiplication in the PSD function, the slices of X in `autocorr` are different lengths and cannot be broadcasted (as far as I know); thus they are mapped over with an `arrayfun` operation.

7. Use your function from above to plot the power spectral density white noise with some variance you choose so as to validate your function's operation. Make sure to use sufficiently large N and M .

```

N = 100000;                                % number of samples
M = 1000;                                   % autocorrelation range
n_variance = 32;                            % variance of white noise
w = -10000:10000;                          % frequency axis for PSD

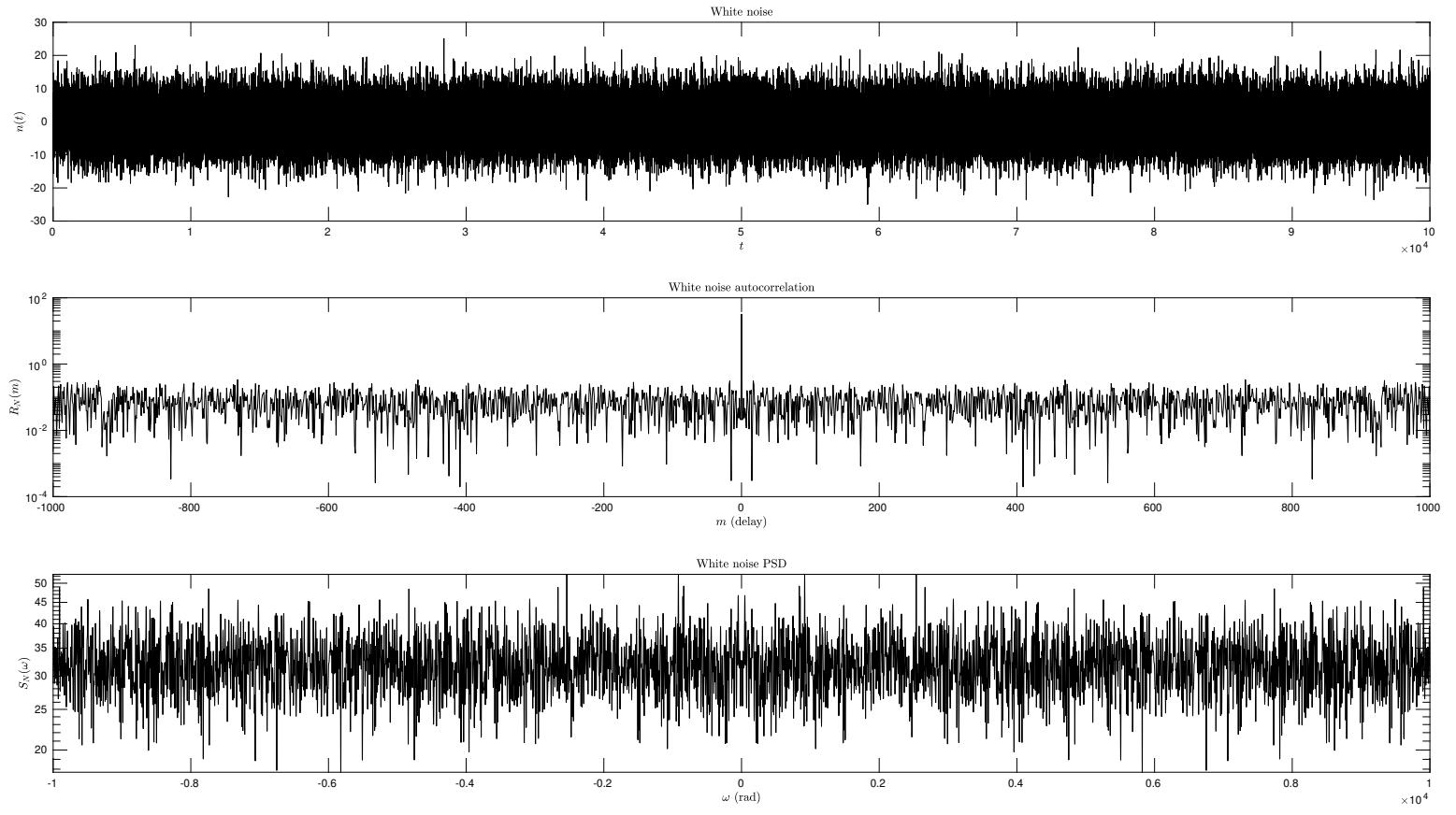
% generate white noise, plot it
n_sig = randn(1, N) * sqrt(n_variance);
subplot(3, 1, 1);
plot(n_sig);
title('White noise');
ylabel('$n(t)$');
xlabel('$t$');

% compute and plot autocorrelation of white noise
m = -M:M;
R_N = arrayfun(@(m) autocorr(n_sig, m), m);
subplot(3, 1, 2);
semilogy(m, abs(R_N));
title('White noise autocorrelation');
ylabel('$R_N(m)$');
xlabel('$m$ (delay)');

% compute and plot PSD of white noise, plot it
S_N = psd(n_sig, M, w);
subplot(3, 1, 3);
semilogy(w, abs(S_N));
title('White noise PSD');
ylabel('$S_N(\omega)$');
xlabel('$\omega$ (rad)');

```

(See figure on next page.)



- The white noise looks about correct.
- The autocorrelation of the white noise is mostly ≈ 0 (the plot axes are semilogy), with a sharp peak at $\omega = 0$, which jives with our understanding of white noise. The peak's magnitude is roughly that of the chosen variance σ^2 (arbitrarily chosen to be 32).
- The PSD is roughly centered about the variance and shows no pattern. While it looks very jagged here because the y-axis scale is narrow, it is arguably fairly “flat.”

8. Use your function to plot the PSD of the of $X(t)$ from problem 4 with $\omega = 10000$ rad/s, as well as the PSD of the integral of $X(t)$. Please don't perform an integral numerically (except to check your work if you want to) – instead, use the relationship of PSDs at the input and output of an LTI system.

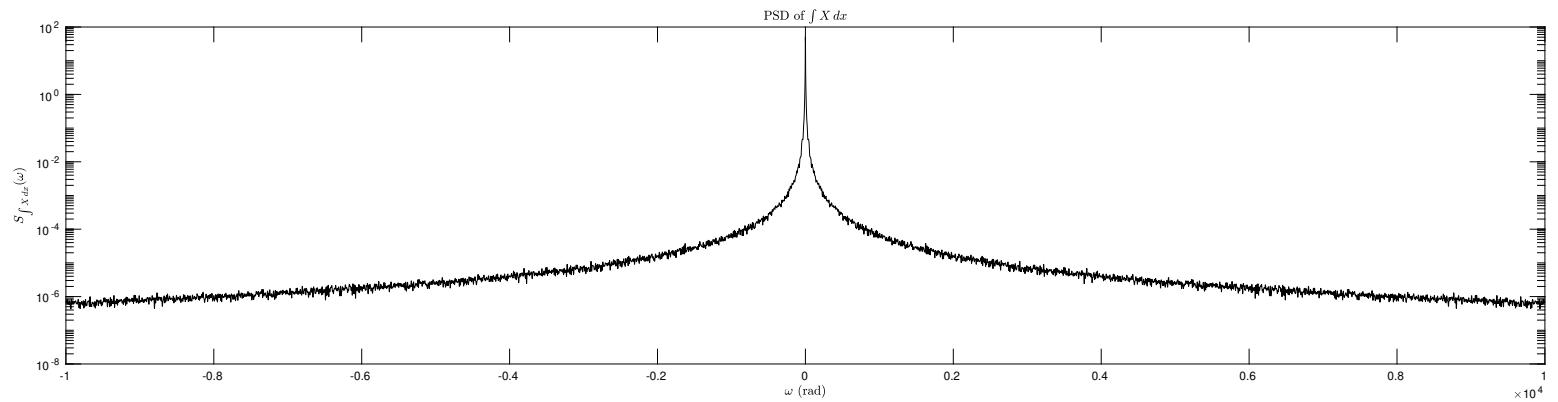
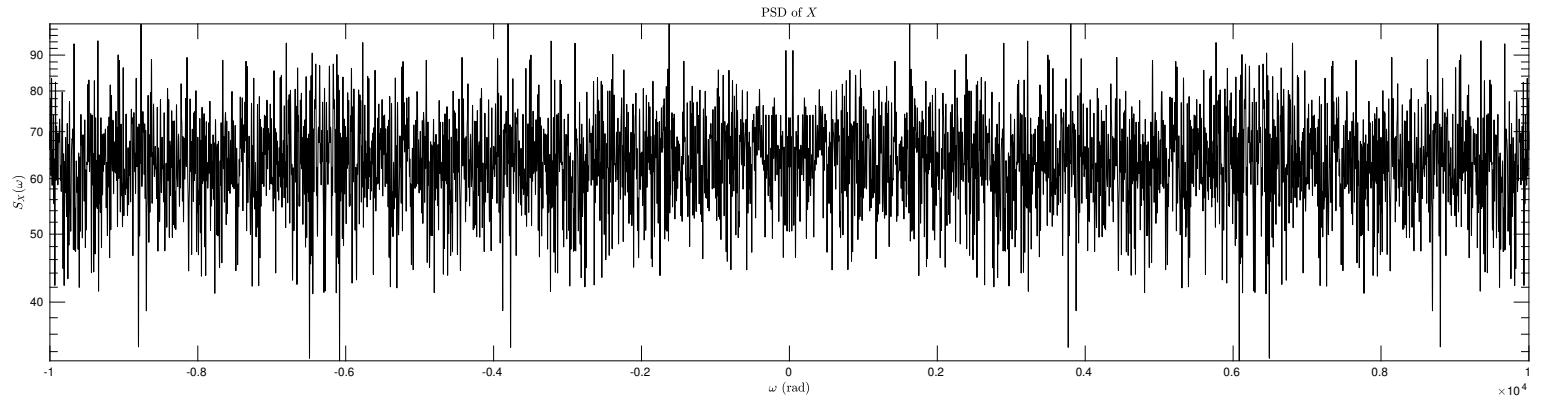
```
w_0 = 10000; % cosine frequency (Q8)
YZ_var = 64; % variance for Y and Z RVs (Q8)

% X(t) = Y*cos(w_0*t) - Z*sin(w_0*t), w_0 = 10000rad/s
t = 0:1/N:1;
Y = randn(size(t)) * sqrt(YZ_var);
Z = randn(size(t)) * sqrt(YZ_var);
X = Y .* cos(w_0 * t) - Z .* sin(w_0 * t);

% compute and plot PSD of X
S_X = psd(X, M, w);
figure();
subplot(2, 1, 1);
semilogy(w, abs(S_X));
title('PSD of $X$');
ylabel('$S_X(\omega)$');
xlabel('$\omega$ (rad)');

% compute and plot PSD of integral of X
S_Y = S_X ./ (w.^2);
subplot(2, 1, 2);
semilogy(w, abs(S_Y));
title('PSD of $\int X dx$');
ylabel('$S_{\int X dx}(\omega)$');
xlabel('$\omega$ (rad)');
```

(See figure on next page.)



- Like in the previous question, the PSD is relatively flat and centered around the variance (arbitrarily chosen to be 64 this time).
- The PSD of the integral of X was generated using the rule based on the Wiener-Khinchin Theorem, namely:

$$S_Y = |H|^2 S_X$$

In this case, the LTI system H has the magnitude response

$$|H|^2 = \left| \frac{1}{j\omega} \right|^2 = \frac{1}{\omega^2}$$

which, when multiplied with the relatively-flat PSD of X , gives the PSD in the second plot.

ECE300 – Pset 4

Jonathan Lam

October 27, 2020

1. I have a baseband analog signal with a 20kHz bandwidth. How fast must I sample to ensure a 4kHz guard band?

A 4kHz guard band means that the Nyquist bandwidth should be at $(20 + 4)\text{kHz} = 24\text{kHz}$, and the Nyquist rate should be $2 \times 24\text{kHz} = 48\text{kHz}$.

2. Suppose I have bandlimited noise with PSD 8 for frequency less than 200kHz in absolute value. Sampling this noise at Nyquist and applying a 16-level quantizer, what are the rate and distortion (mean squared error)? What is the SQNR?

The Nyquist rate is $2 \times 200\text{kHz} = 400\text{kHz}$. Sampling at this rate, with 16 levels (4 bits) per sample, yields a bit transmission rate of $4\text{b} \times 400\text{kHz} = 1.6\text{Mbps}$.

Denote the random process as X , and the quantization function Q . We're given the PSD of X :

$$S_X(f) = \begin{cases} 8, & |f| < 200000 \\ 0, & \text{else} \end{cases}$$

The mean squared error D is defined as follows:

$$D = E[(X - Q(X))^2] = \int_{-\infty}^{\infty} (x - Q(x))^2 f_X(x) dx$$

Now we need to find the PDF f_X . Assuming that the noise is white and gaussian, and knowing that the autocorrelation function of white gaussian noise is a delta with height σ_X^2 at $\tau = 0$, and knowing that the autocorrelation function is the inverse Fourier transform of the PSD:

$$\begin{aligned} \sigma^2 &= R_X(\tau) \Big|_{\tau=0} = \int_{-\infty}^{\infty} S_X(f) e^{j2\pi f \tau} df \Big|_{\tau=0} \\ &= \int_{-200000}^{200000} (8)(1) df = 3200000 \end{aligned}$$

Plugging this into the PDF of a gaussian:

$$f_X(x) = \frac{1}{\sqrt{2\pi \times 3200000}} \exp \frac{x^2}{2 \times 3200000}$$

Finally, substituting this into the mean squared error formula:

$$\begin{aligned} D &= \int_{-\infty}^{\infty} (x - Q(x)) f_X(x) dx \\ &= \sum_{i=1}^{16} \int_{R_i} (x - \hat{x}_i)^2 f_X(x) dx \end{aligned}$$

where R_i and x_i denote the quantization intervals and levels, respectively. Without knowing the quantization function, this is as far as we know.

Plugging into the definition for SQNR:

$$\text{SQNR} = \frac{E[X^2]}{E[(X - Q(X))^2]} = \frac{R_X(0)}{D} = \frac{3200000}{\sum_{i=1}^{16} \int_{R_i} (x - \hat{x}_i)^2 f_X(x) dx}$$

The next few problems will consider the following setup: Suppose I am doing a digital logic design project, and I represent 1 as a 2.5V pulse of duration A and 0 as a -2.5V pulse of duration A. My chips perform a least squares decision when they receive a signal. The noise over a wire is largely thermal noise caused by statistical variations in charge carriers, modeled as AWGN. This noise has variance $4k_B T R$ where $k_B \approx 1.38 \times 10^{-23} \text{ J/K}$ is the Boltzmann constant, T is the absolute temperature (in Kelvin) and R is the total resistance of the wire. A commonly used wire type (available in the lab at school) is 22 AWG solid copper wire, for which 1m of wire has $52.7 \text{ m}\Omega$ resistance. Recall that resistance is proportional to length.

3. *Describe the operation of a least squares decision system for this problem.*

If you had some way to measure the average voltage of the signal over the period $t \in [0, A]$, and see if that value is positive or negative (where positive indicates the positive pulse, and negative indicates the negative pulse), this would perform the least squares decision system. (This is closer to doing a matched filter (convolution with a rectangular pulse should give some measure of the average value over that pulse), but as stated below in (4), this is equivalent to a least-squares decision boundary. Clearly, the decision boundary is at zero average voltage.)

4. *Is the least squares decision rule the same as a matched filter decision rule in this case? Is it the same as maximum likelihood? Is it the same as maximum a posteriori? If there is not enough information to answer, give the conditions that would guarantee the equality.*

It is equivalent to MF, since the two pulses are equal-energy. The inner product is a measure of distance if the units are properly normalized.

It is equivalent to ML, since the PDF of the noise is symmetric (since it is gaussian). This is clear from a geometric perspective since the halfway point is in the middle due to symmetry.

It is equivalent to MAP iff both the positive and negative pulses are equiprobable. ML is MAP with the assumption that the priors are equal (events are equiprobable).

5. *What is the name of this modulation scheme? What is the basis? Draw the geometrical representation.*

This is the binary antipodal signaling scheme (a form of PAM).

The (single) basis vector is the positive signal (a rectangular pulse of height 2.5 and width A , denote this S_1) normalized to unit energy. (The choice of positive or negative symbol is arbitrary.) I.e.:

$$\psi(t) = \frac{S_1(t)}{\|S_1(t)\|} = \frac{S_1(t)}{\sqrt{\int_0^A 2.5^2 dt}} = \frac{1}{2.5\sqrt{A}} S_1(t)$$

Geometrical representation: along a 1-D line representing the ψ axis, the two signals are located at $\pm \|S_1(t)\| = \pm\sqrt{\varepsilon_1}$.

6. *For this scheme, suppose I transmit a 1 75% of the time – what are the MAP decision regions?*

MAP minimizes the probability of error, so the decision boundary should lie at the point where the probability of the transmitted signal being either 0 or 1 to be equal. Let y denote the signal in the ψ representation, and \hat{x} denote the transmitted signal. Let α denote the decision boundary.

$$\begin{aligned} P_{err} &= P[y > 0 | \hat{x} = 0]P[\hat{x} = 0] + P[y < 0 | \hat{x} = 1]P[\hat{x} = 1] \\ &= P[N(-2.5\sqrt{A}, 2\sqrt{k_BTR}) > 0] (0.25) + P[N(2.5\sqrt{A}, 2\sqrt{k_BTR}) < 0] (0.75) \\ &= \frac{1}{4} \frac{1}{2\pi(4k_BTR)} \int_{-\infty}^{\alpha} \exp \frac{-(x+2\sqrt{A})^2}{2(4k_BTR)} dx + \frac{3}{4} \frac{1}{2\pi(4k_BTR)} \int_{\alpha}^{\infty} \exp \frac{-(x-2\sqrt{A})^2}{2(4k_BTR)} dx \end{aligned} \quad (0.75)$$

This becomes an optimization problem to find $\text{argmin}_{\alpha} P_{err}$.

$$\begin{aligned} 0 &= \frac{\partial P_{err}}{\partial \alpha} \\ &= \frac{1}{\sqrt{2\pi(4k_BTR)}} \left[\frac{1}{4} \exp \frac{-(\alpha+2.5\sqrt{A})^2}{2(4k_BTR)} - \frac{3}{4} \exp \frac{-(\alpha-2.5\sqrt{A})^2}{2(4k_BTR)} \right] \\ \frac{3}{1} &= \frac{\exp \frac{-(\alpha+2.5\sqrt{A})^2}{2(4k_BTR)}}{\exp \frac{-(\alpha-2.5\sqrt{A})^2}{2(4k_BTR)}} = \exp \frac{-4\alpha(2.5\sqrt{A})}{2(4k_BTR)} \\ \alpha &= -\frac{4k_BTR}{5\sqrt{A}} \log(3) \end{aligned}$$

It makes sense that this value is negative, as the probability of transmitting a 1 is more likely.

7. Assume now that the symbols are equiprobable. In terms of A , T and wire length L , write a formula for the SNR and probability of error.

Since resistance is proportional to length, define the resistivity ρ to be the constant of proportion:

$$\begin{aligned} R &\propto L \\ R &= \rho L \\ \rho &= \frac{R}{L} = \frac{52.7 \text{ m}\Omega}{1 \text{ m}} = 52.7 \times 10^{-3} \Omega \text{ m}^{-1} \end{aligned}$$

SNR calculation (use the MF version here, since it is equivalent):

$$\text{SNR} = \frac{\frac{\varepsilon_y}{N_0/2}}{4k_BTR} = \frac{2.5^2 A}{4k_BTR}$$

Probability of error calculation:

$$\begin{aligned} P_{err} &= P[y > 0 | \hat{x} = 0]P[\hat{x} = 0] + P[y < 0 | \hat{x} = 1]P[\hat{x} = 1] \\ &= P[y > 0 | \hat{x} = 0] \\ y|(x = 0) &\sim N\left(-2.5\sqrt{A}, 2\sqrt{k_BTR}\right) \\ P_{err} &= Q\left(\frac{0 - (-2.5\sqrt{A})}{2\sqrt{k_BTR}}\right) \\ &= Q\left(\frac{5}{4}\sqrt{\frac{A}{k_BTR}}\right) \\ &= Q\left(\frac{5}{4}\sqrt{\frac{A}{k_B T \rho L}}\right) \end{aligned}$$

8. Assume we are operating the system at room temperature – how long does the wire have to be to yield an error probability of $1/10$? Write this number in lightyears. Compare this to the size of the Milky Way. This should give you an appreciation for how robust wired communication is to thermal noise

A value for A was never given, so arbitrarily let $A = 1 \text{ s}$. Also, let the

room temperature be 25°C . Plugging into the error formula:

$$\begin{aligned} \frac{1}{10} &= Q \left(\frac{5}{4} \sqrt{\frac{A}{k_B T \rho L}} \right) \\ L &= \left(\frac{4}{5} \sqrt{\frac{k_B T \rho}{A}} Q^{-1} \left(\frac{1}{10} \right) \right)^{-2} \\ &= \left(\frac{4}{5} \sqrt{\frac{(1.38 \times 10^{-23} \text{J/K})(298\text{K})(52.7 \times 10^{-3} \Omega\text{m}^{-1})}{(1\text{s})}} Q^{-1} \left(\frac{1}{10} \right) \right)^{-2} \\ &= 2.66 \times 10^{21} \text{m} = 4.39 \times 10^5 \text{ly} \end{aligned}$$

That is long. The Milky Way's diameter, in comparison, is roughly $1.06 \times 10^5 \text{ly}$, so they are on the same order of magnitude.

Granted, a one second pulse may be considered slow by today's standards. By the above formula, it is not hard to see that $L \propto A$. Even if our device was clocked in the microsecond range $A \approx 1\mu\text{s}$, the length would be $4.39 \times 10^{-1} \text{ly}$, still formidable.

ECE300 – Pset 5

Jonathan Lam

November 30, 2020

1. *Describe when and why we use raised cosine pulses, citing the necessary conditions, at least one theorem and drawing at least one figure.*

Raised cosine pulses are used to achieve the Nyquist zero-ISI criterion, which states that zero ISI is possible iff the sum of samples of the Fourier transform (with period $2\pi/T_s$, where T_s is the symbol period) is a constant, regardless of the starting point of the sampling. The assumption made is that the channel is a perfect low-pass filter (i.e., constant in the passband), which is a good-enough approximation for most cases. They are used because they are more closely realizable than sinc pulses (the tails in the time-domain representation dies off quicker).

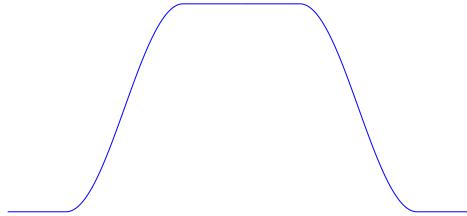


Figure 1: Sample raised cosine with $T_s = 1$ and $\alpha = 0.5$

Raised cosine:

$$X_{rc}(\omega) = \begin{cases} T, & 0 \leq |\omega| \leq \pi \frac{1-\alpha}{T} \\ \frac{T}{2} \left(1 + \cos\left(\frac{T}{2\alpha} (|\omega| - \pi \frac{1-\alpha}{T})\right)\right), & \pi \frac{1-\alpha}{T} \leq |\omega| \leq \pi \frac{1+\alpha}{T} \\ 0, & \pi \frac{1+\alpha}{T} \leq |\omega| \end{cases}$$

2. *True raised cosine pulses are not realizable – why not? What is done in practice to approximate the raised cosine?*

They are not realizable because they are not compact in frequency, and are thus infinite in time, and thus also noncausal. (The time-domain representation involves the sinc, which is infinite.) This can be estimated by truncating the signal (to make it compact in time) and delaying it (to make it causal).

3. Find the entropy of a geometrically distributed random variable (that is, a variable with probability mass function $f(m) = p(1-p)^{m-1}$ for positive integer m and some fixed probability p) as a function of p .

$$\begin{aligned}
H(X) &= \mathbb{E}[I(X)] = \mathbb{E}[-\log f(x)] \\
&= \sum_{x=1}^{\infty} -\log(p(1-p)^{x-1}) p(1-p)^{x-1} \\
&= \sum_{x=1}^{\infty} -[\log p + (x-1)\log(1-p)] p(1-p)^{x-1} \\
&= \sum_{x=1}^{\infty} [-p(\log p - \log(1-p))] (1-p)^{x-1} + \sum_{x=1}^{\infty} [-p \log(1-p)] x(1-p)^{x-1} \\
&= p \log\left(\frac{1-p}{p}\right) \sum_{x'=0}^{\infty} (1-p)^{x'} + p \log\left(\frac{1}{1-p}\right) \sum_{x=1}^{\infty} x(1-p)^{x-1}
\end{aligned}$$

Infinite sum formulas:

$$\begin{aligned}
\sum_{x=0}^{\infty} r^x &= \frac{1}{1-r} \quad (\text{for } |r| < 1) \\
\frac{d}{dr} \left[\sum_{x=0}^{\infty} r^x \right] &= \sum_{x=1}^{\infty} x r^{x-1} = \frac{1}{(1-r)^2} = \frac{d}{dr} \left[\frac{1}{1-r} \right] \quad (\text{for } |r| < 1)
\end{aligned}$$

Substituting:

$$\begin{aligned}
H(X) &= p \log\left(\frac{1-p}{p}\right) \frac{1}{1-(1-p)} + p \log\left(\frac{1}{1-p}\right) \frac{1}{(1-(1-p))^2} \\
&= \log\left(\frac{1-p}{p}\right) + \frac{1}{p} \log\left(\frac{1}{1-p}\right) \\
&= \log(1-p) - \log(p) - \frac{1}{p} \log(1-p) \\
&= \frac{p-1}{p} \log(1-p) - \log p \\
&= \log\left(\frac{(1-p)^{1-\frac{1}{p}}}{p}\right)
\end{aligned}$$

4. Suppose a source has alphabet $\{a_1, a_2, \dots, a_8\}$ with corresponding output probabilities $\{\frac{1}{16}, \frac{1}{16}, \frac{1}{4}, \frac{1}{32}, \frac{3}{32}, \frac{1}{8}, \frac{1}{16}, \frac{5}{16}\}$. Determine the entropy of the

source.

$$\begin{aligned}
H(X) &= \sum_{i=1}^8 -p_i \log p_i \\
&= \frac{4}{16} + \frac{4}{16} + \frac{2}{4} + \frac{5}{32} + \left(\frac{-3 \log 3}{32} + \frac{15}{32} \right) + \frac{3}{8} + \frac{4}{16} + \left(\frac{-5 \log 5}{16} + \frac{20}{16} \right) \\
&= \frac{7}{2} - \frac{3 \log 3 + 10 \log 5}{32} \\
&= \frac{7}{2} - \frac{\log 263671875}{32} \approx 2.626
\end{aligned}$$

5. Design a Huffman code for the above source. Validate that it satisfies the Kraft inequality, and that the average length satisfies the Huffman code entropy inequalities.

See Figure 2 for a visualization of the designed Huffman coding tree. Table 1 shows the equivalent mappings as a table.

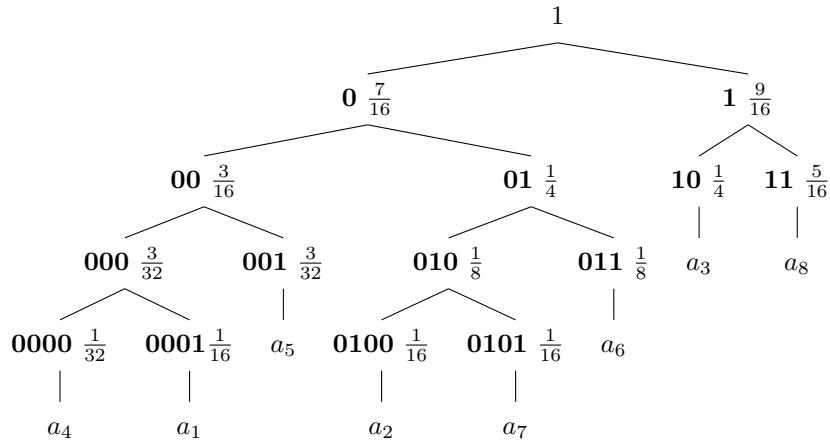


Figure 2: Huffman code tree. Bolded numbers are the codewords (codeword prefixes), and the numbers alongside them are their probabilities. Corresponding symbols are indicated at the leaves.

Kraft inequality (where l_i is the length of each word):

$$\sum_{i=1}^N 2^{-l_i} \leq 1$$

In this example:

$$\sum_{i=1}^8 2^{-l_i} = (4)2^{-4} + (2)2^{-3} + (2)2^{-2} = \frac{4}{16} + \frac{2}{8} + \frac{2}{4} = 1$$

Symbol	Code	Probability
a_1	0001	1/16
a_2	0100	1/16
a_3	10	1/4
a_4	0000	1/32
a_5	001	3/32
a_6	011	1/8
a_7	0101	1/16
a_8	11	5/16

Table 1: Huffman code symbol-to-code mapping

thus the Kraft inequality is satisfied. For a Huffman code, the average length satisfies:

$$H(X) \leq \bar{l} \leq H(X) + 1$$

In this example:

$$\begin{aligned}\bar{l} &= \sum_{i=1}^8 l_{a_i} p_i \\ &= \frac{4}{16} + \frac{4}{16} + \frac{2}{4} + \frac{4}{32} + \frac{9}{32} + \frac{3}{8} + \frac{4}{16} + \frac{10}{16} = \frac{85}{32} \approx 2.656\end{aligned}$$

This satisfies the above inequality, i.e., $2.626 \leq 2.656 \leq 3.626$.

6. Find the transmission power (in watts) necessary for an AWGN channel with $N_0 = 108\text{J}$ and transmission bandwidth $B = 1\text{MHz}$ to achieve a channel capacity of 1Mbps.

The channel capacity assuming AWGN (in bits/sec) is

$$C = B \log \left(1 + \frac{P}{P_N} \right)$$

Solving for P :

$$\begin{aligned}P_N &= N_0 B \\ P &= N_0 B \left(2^{C/W} - 1 \right) \\ &= (108\text{J})(10^6\text{Hz}) \left(2^{(10^6\text{Hz})/(10^6\text{Hz})} - 1 \right) \\ &= 1.08 \times 10^8 \text{W}\end{aligned}$$

7. The next several questions will refer to the (n, k) linear block code with generator matrix

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

What are n and k ?

$$n = 7, k = 3$$

8. Make a list of every data vector and associated codeword.

$$XG = C$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

There are 8 possible datawords and codewords. Each row of X and the corresponding row in C represent a dataword and codeword pair, respectively.

9. What is the minimum Hamming distance of the code, d_{min} ? How many errors can this code correct?

In a linear code, $d_{min} = w_{min}$ (minimum distance of a code is equal to the minimum weight of that code). It is not hard to see that the minimum weight of this code is 3. Thus this code can detect any error with no more than 2 bit errors, and correct any case with a single bit error.

10. There exists a systematic code with the same codewords as this code – how can you know that from looking at the list? It should be straightforward to create the generator matrix, G_S .

If we reduce G to reduced row-echelon form using elementary row operations, we preserve the row space (and thus the same outputted codewords). Since the RREF of G has an identity matrix for the first three columns (i.e., the first three columns are linearly independent), this becomes a systematic code with the same row space (codewords). (Visually, looking at the list of codewords, we see that we have a linear code where each of the datawords forms the first three bits of some codeword, so it should be possible to construct a systematic code.)

To find G_S , swap the first two rows, add (XOR) the third row to (with) the second row.

$$G_S = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array} \right) = (I_3 \mid P)$$

(Alternatively, the three row in G_S are the three codewords that begin with 001, 010, and 001, which can be thought of as a basis for the codewords/row space.)

11. Find the parity check matrix of G_S , H .

$$H = (P^T \mid I_{n-k}) = \left(\begin{array}{ccc|ccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array} \right)$$

12. Does H work as a parity check matrix for G ?

(The following matrix multiplications were performed in MATLAB.)

$$GH^T = G_S H^T = \mathbf{0}_{3 \times 4}$$

Thus H works as a parity check for both G and G_S . This is a nice property given that they have the same codewords.

ECE310 – Project 1

Jonathan Lam

September 29, 2020

Problem description

Design a sample rate converter that efficiently converts an input audio signal, sampled at 11025Hz, to an output audio signal sampled at 24000Hz. Record the number of floating-point operations (FLOs) performed.

The resulting system should approximately look like a low-pass filter, and it should meet the following specifications:

passband cutoff frequency	$\omega_p = 11025\pi/24000$
passband ripple	$R_p < \pm 0.1\text{dB}$
stopband frequency	$\omega_s = 1.2\omega_p$
stopband attenuation	$R_s > 70\text{dB}$
phase constraints (in passband)	$ \max\{\text{grpdelay}\} - \min\{\text{grpdelay}\} < 720$

Notes on implementation

- A three-stage system was used here. The ratio 24000/11025 can be reduced to the whole-number ratio 320/147, which can be expressed as the product $5/3 \times 8/7 \times 8/7$. These (small) integers were used to make reasonable upsample/filter/downsample blocks, where each system is implemented using a polyphase decomposition of the interpolation. (The polyphase implementation may also have been performed on the decimation, but here all of the interpolation ratios are larger than the decimation ratios, so this is more efficient.)
- Similar to the textbook, only FLOs relating to the polyphase implementation (convolution and additions) were counted.
- I decided to report the FLOs as floating point operations per input sample (how we should report efficiency was not specified in the problem description). If we were processing an input sample in real-time (which we are not), this can be translated into FLOPS by dividing by the input audio sample rate (11025Hz). To get the total number of FLOs, multiply the floating-point operations per sample by the number of samples.
- The LPFs were designed to be lowest-order Chebyshev filters using `cheb2ord` and `cheby2`.
- Greatly appreciated help was received from Dan Kim and Tony Belladonna.

Source code

Sample rate converter

```
% efficiently converts sampling rate from 11.025kHz to 24kHz
% (or any other resampling with a ratio of 320/147)
% returns the samplerate-converted signal and the number of FLOs per sample
function [x, flo] = srconvert(x)
    % break down into three stages
    [x, flo1] = efficientresample(x, 5, 3);
    [x, flo2] = efficientresample(x, 8, 7);
    [x, flo3] = efficientresample(x, 8, 7);

    % number of FLOs per sample
    flo = flo1 + flo2 + flo3;
end

% uses polyphase decomposition to efficiently up- and down-sample x
function [res, flo] = efficientresample(x, L, M)
    % design the LPF
    % source: https://www.mathworks.com/help/signal/ref/cheb2ord.html
    Wp = min(1/L, 1/M);
    Ws = Wp * 1.2;
    Rp = 0.1;
    Rs = 85;
    [n, Ws] = cheb2ord(Wp, Ws, Rp, Rs);
    [b, a] = cheby2(n, Rs, Ws, 'low');
    h = impz(b, a).';

    % decompose h into its polyphase components
    hc = poly1(h, L);

    % do all the convolutions
    res = zeros(1, L * length(x));
    for i = 1:L
        component = upsample(fftfilt(hc(i, :), x), L);
        res = res + [zeros(1, i-1) component(1:length(component)+1-i)];
    end

    % decimation and rescaling
    res = L * downsample(res, M);

    % counting calculations; same calculation as in textbook;
    % L * (N/L) = N multiplications/(sample period) in convolutions
    % L * (N/L - 1) = N - L adds/(sample period) in convolutions
    % L - 1 additions when adding polyphase components together
    % total = N + (N - L) + (L - 1) = 2 * N + 1 FLO per sample
    flo = 2 * length(h) + 1;
end
```

poly1.m (provided in assignment)

```
function E=poly1(h,M)
%
% Performs type I polyphase decomposition of h in M components.
% The ith row of E corresponds to the ith polyphase component.
% Assumes that the first point of h is index 0.
%
h = [h zeros(1, ceil(length(h)/M)*M-length(h))];
E = reshape(h, M, length(h)/M);
```

verify.m (provided in assignment)

```
% Verifies that h meets all the specifications given in the
% handout. Returns a 1 if all specifications are met.
function yes = verify(h)
    [R,G,A]=examlpf(h,147/320,147/320*1.2);

    % passband magnitude
    yes      = R <=0.1;
    subplot(3,1,1)

    if R <=0.1
        title(['Passband Response, Ripple = ' num2str(R) ' dB, OK']);
    else
        title(['Passband Response, Ripple = ' num2str(R) ' dB, too big']);
    end

    % Group delay
    subplot(3,1,2)
    if G <= 720
        title(['Group Delay in Passband, max-min = ' num2str(G) ', OK']);
    else
        title(['Group Delay in Passband, max-min = ' num2str(G) ', too big']);
    end
    yes      = yes*(G <= 720);

    % Stopband magnitude
    subplot(3,1,3)
    if A <=-70
        title(['Overall Response , Attenuation = ' num2str(A) ' dB, OK']);
    else
        title(['Overall Response , Attenuation = ' num2str(A) ' dB, too small']);
    end

    yes      = yes * (A <= -70);

    sprintf('Passband Ripple:      %5.3f dB \n',R)
    sprintf('Groupdelay Variation:  %5d samples \n',G)
    sprintf('Stopband Attenuation:  %5.3f dB \n',A)
return
```

Results

Testing sample rate converter on an audio file

```
% testing wagner; resample and write to new file
[x, fs] = audioread('Wagner.wav');
x = x.';
[y, flo] = srconvert(x);
fprintf('FLOs per sample: %d\n', flo);
audiowrite('resampledWagner.wav', y, 24000);

% plot x and y
subplot(2, 1, 1);
t = linspace(0, length(x)/fs, length(x));
plot(t, x);
title(sprintf('Original, sampled at 11025Hz (%d samples)', length(x)));
xlabel('t (s)');
ylim([-1 1]);
xlim([0 length(x)/fs]);
subplot(2, 1, 2);
t = linspace(0, length(x)/fs, length(y));
plot(t, y);
title(sprintf('Sample rate-converted to 24000Hz (%d samples)', length(y)));
xlabel('t (s)');
ylim([-1 1]);
xlim([0 length(x)/fs]);
figure();
```

Output:

```
FLOs per sample: 3477
```

See (Figure 1) for the generated plots.

Testing sample rate converter using verify()

```
ir = srconvert([1 zeros(1,3000)]);
verify(ir);
```

Output:

```
Passband Ripple:      0.001 dB
Groupdelay Variation: 1.585250e+01    samples
Stopband Attenuation: -73.208 dB
```

See (Figure 2) for the generated plots.

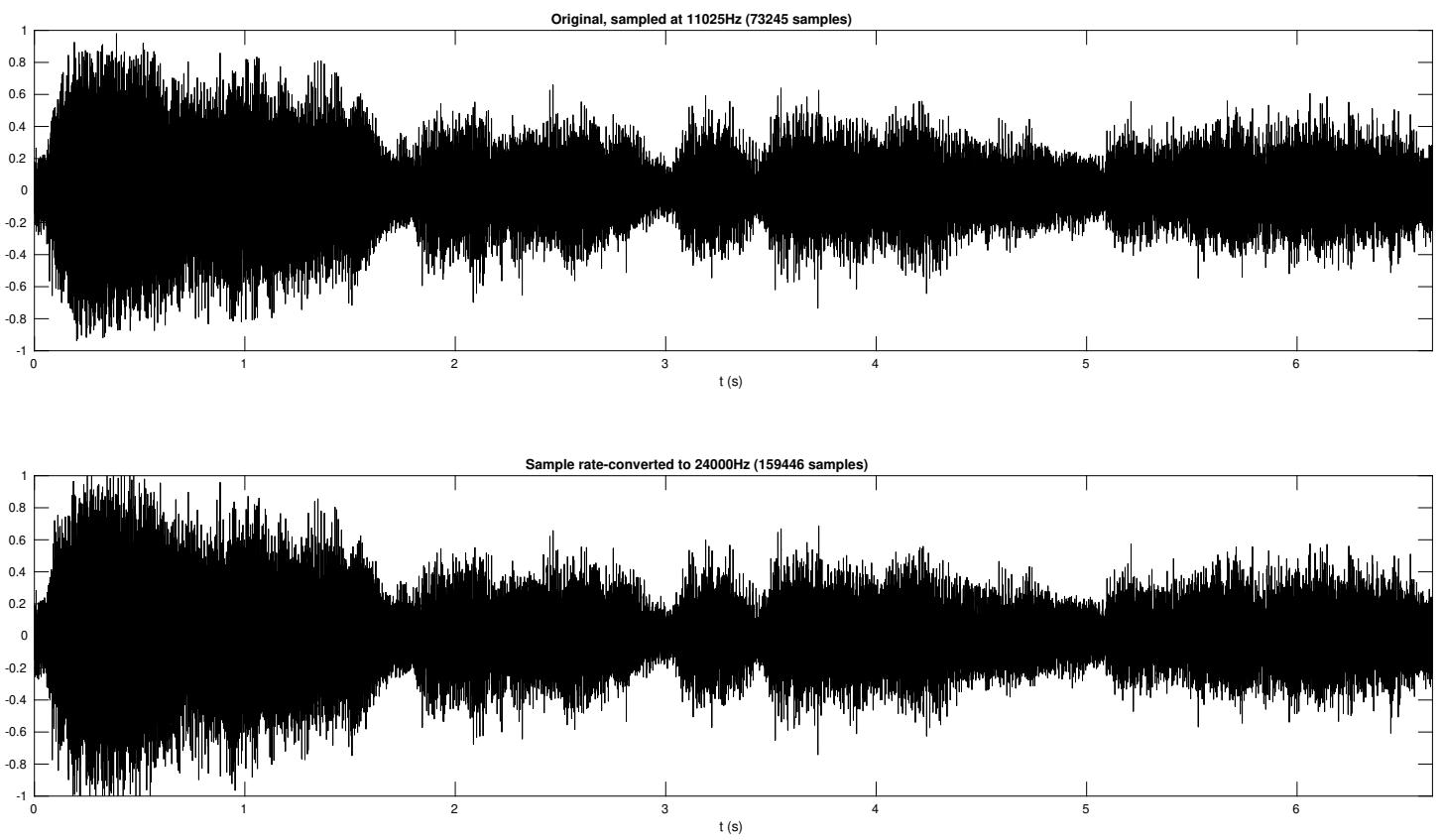


Figure 1: Comparison of original Wagner.wav with sample rate-converted signal. Both the visual quality (from these graphs) and the audio quality (from listening to the two audio files) are comparable.

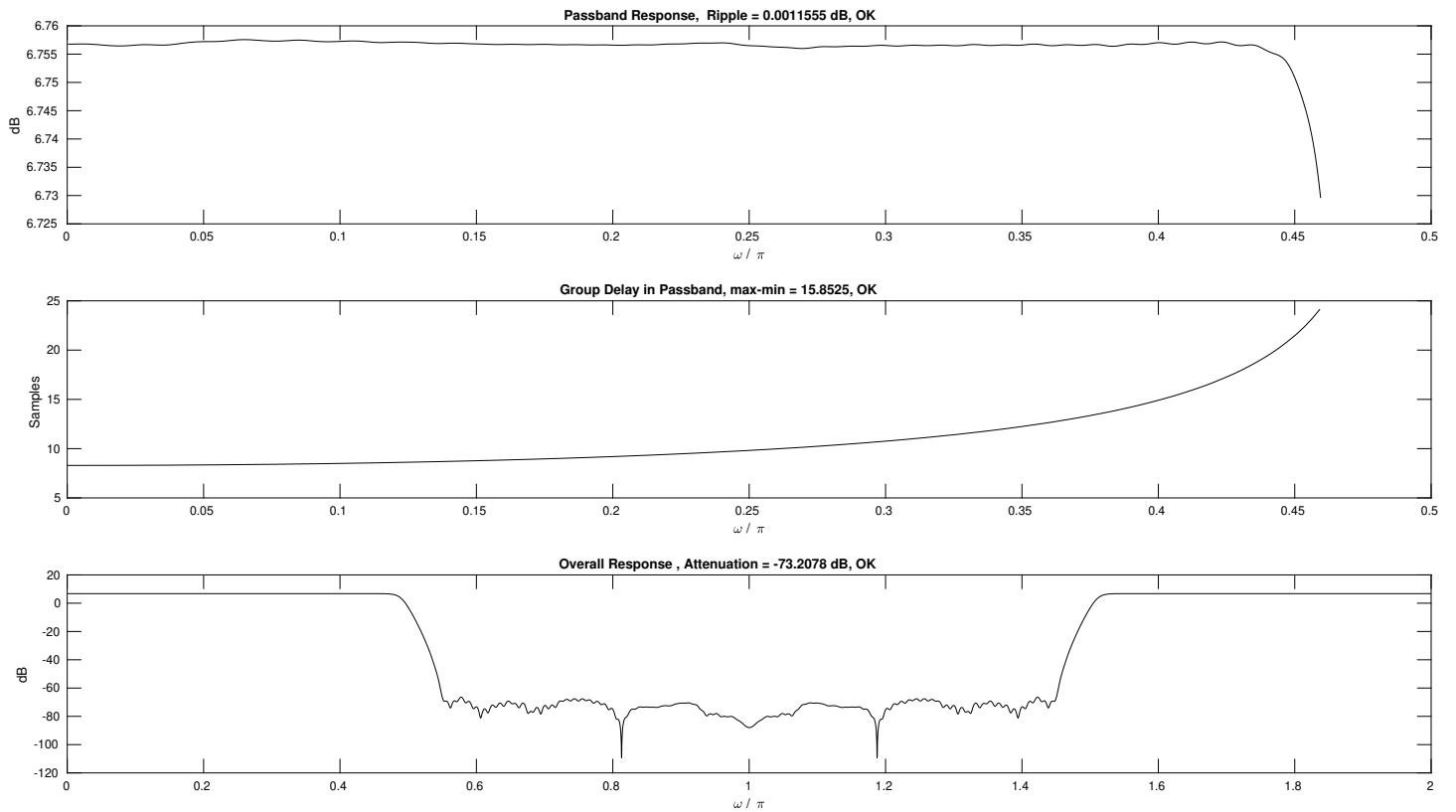


Figure 2: Verification that the system meets the specifications.

ECE310 – Project 2

Jonathan Lam

October 22, 2020

Project description

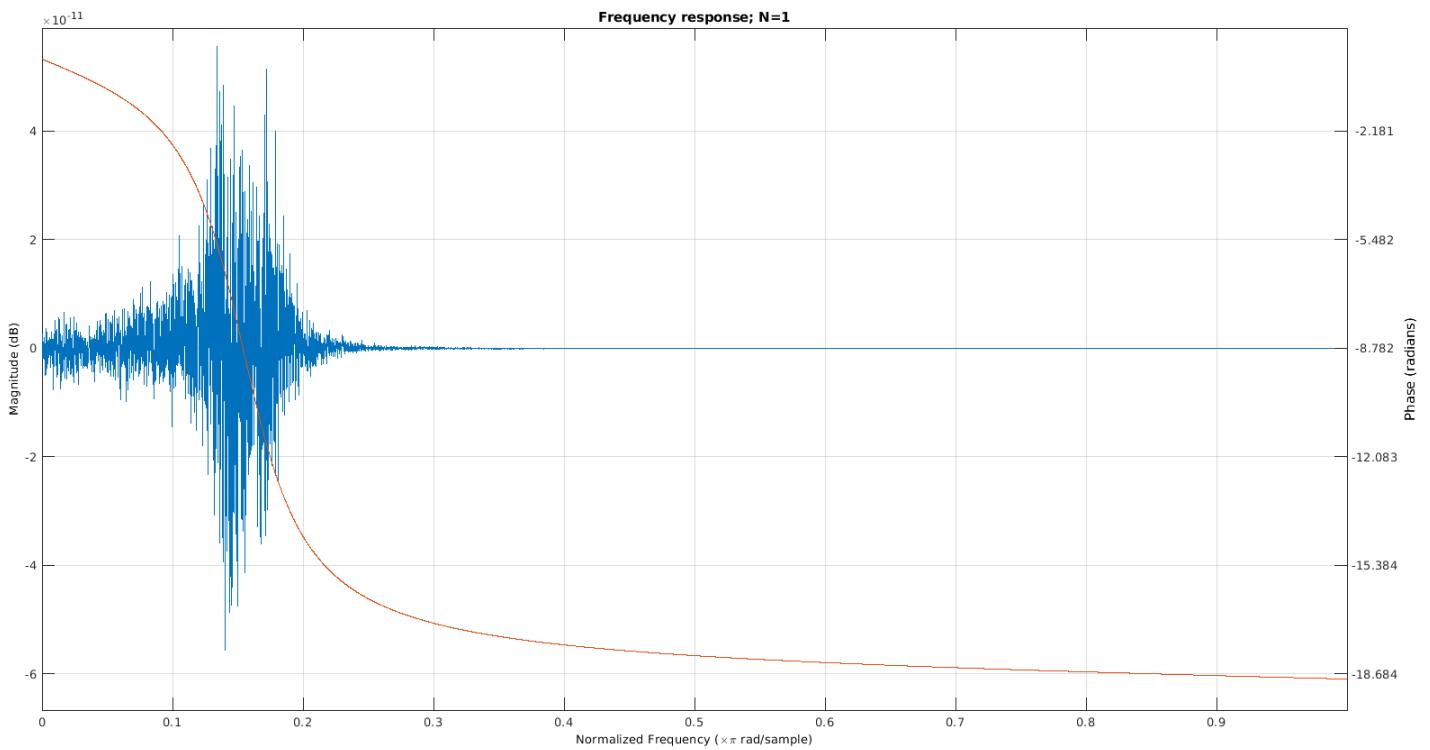
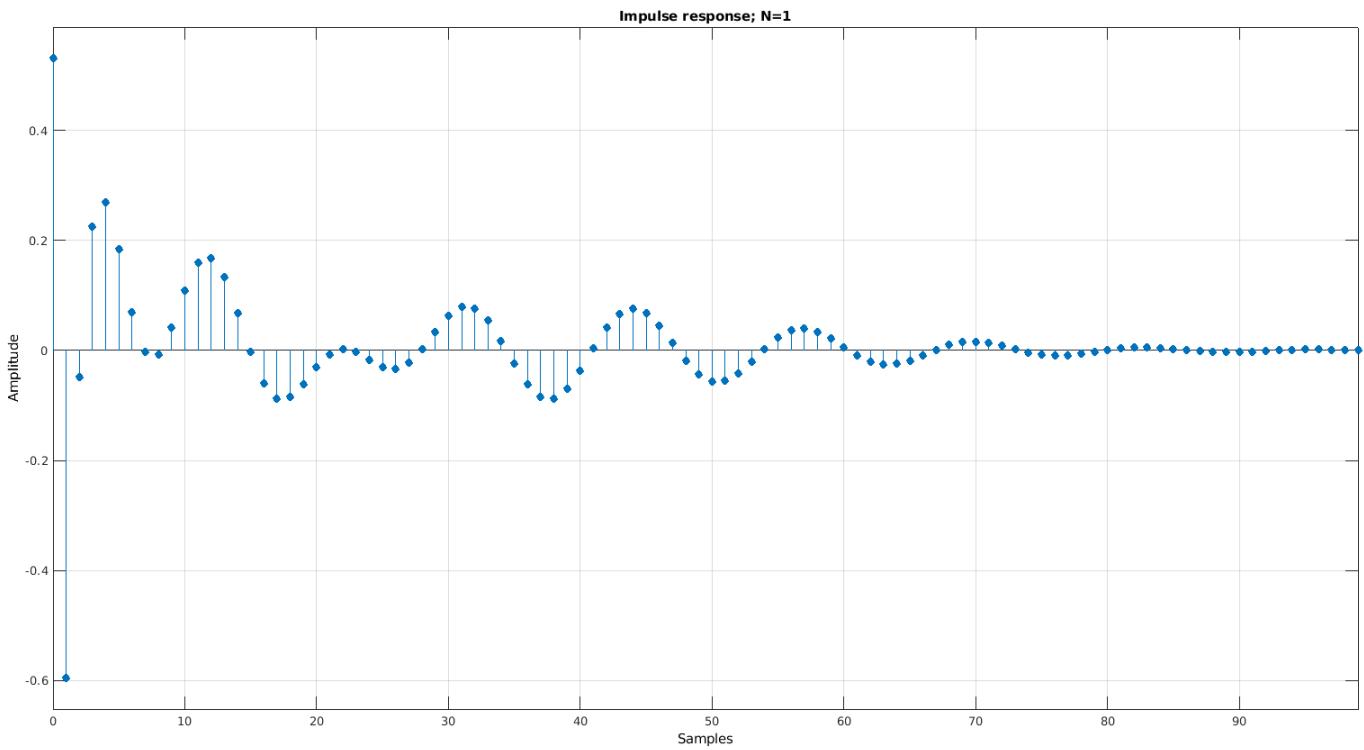
Explore MATLAB's `dfilt` package, cascading filters, different filter implementations (DF1, DF1SOS, DF2SOS, DF2TSOS).

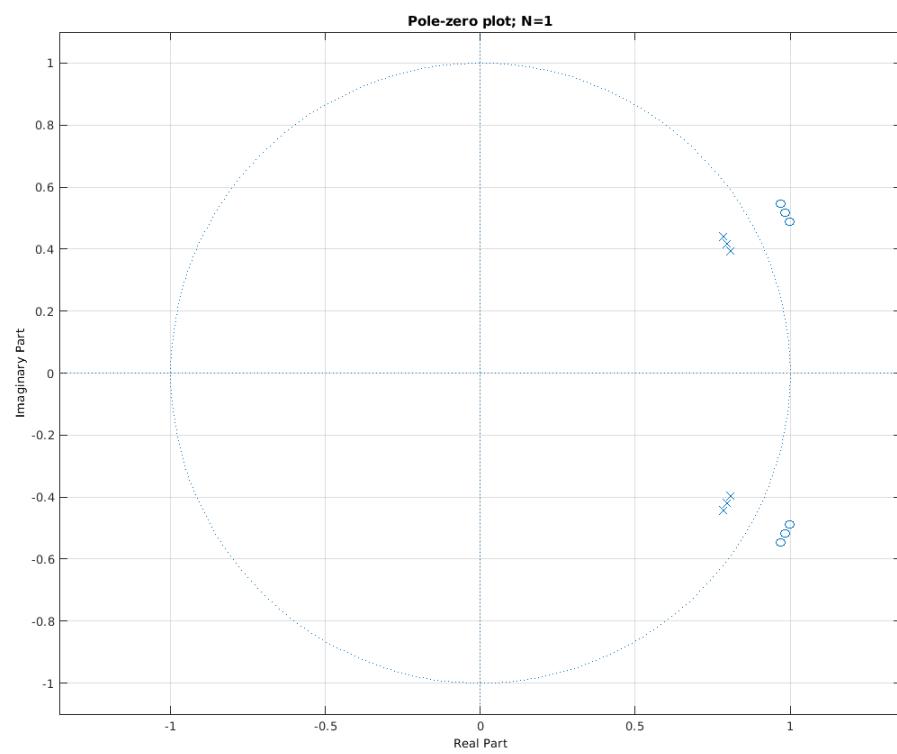
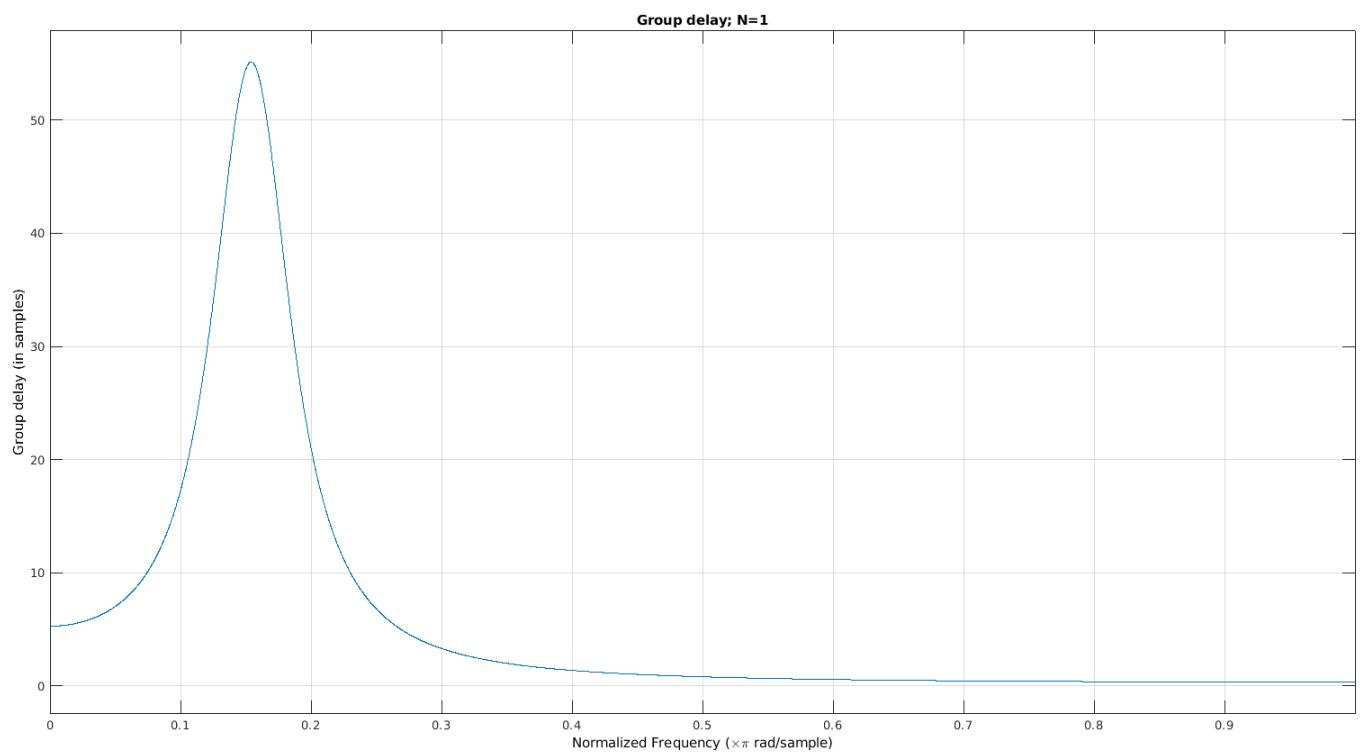
Notes/Answers to questions

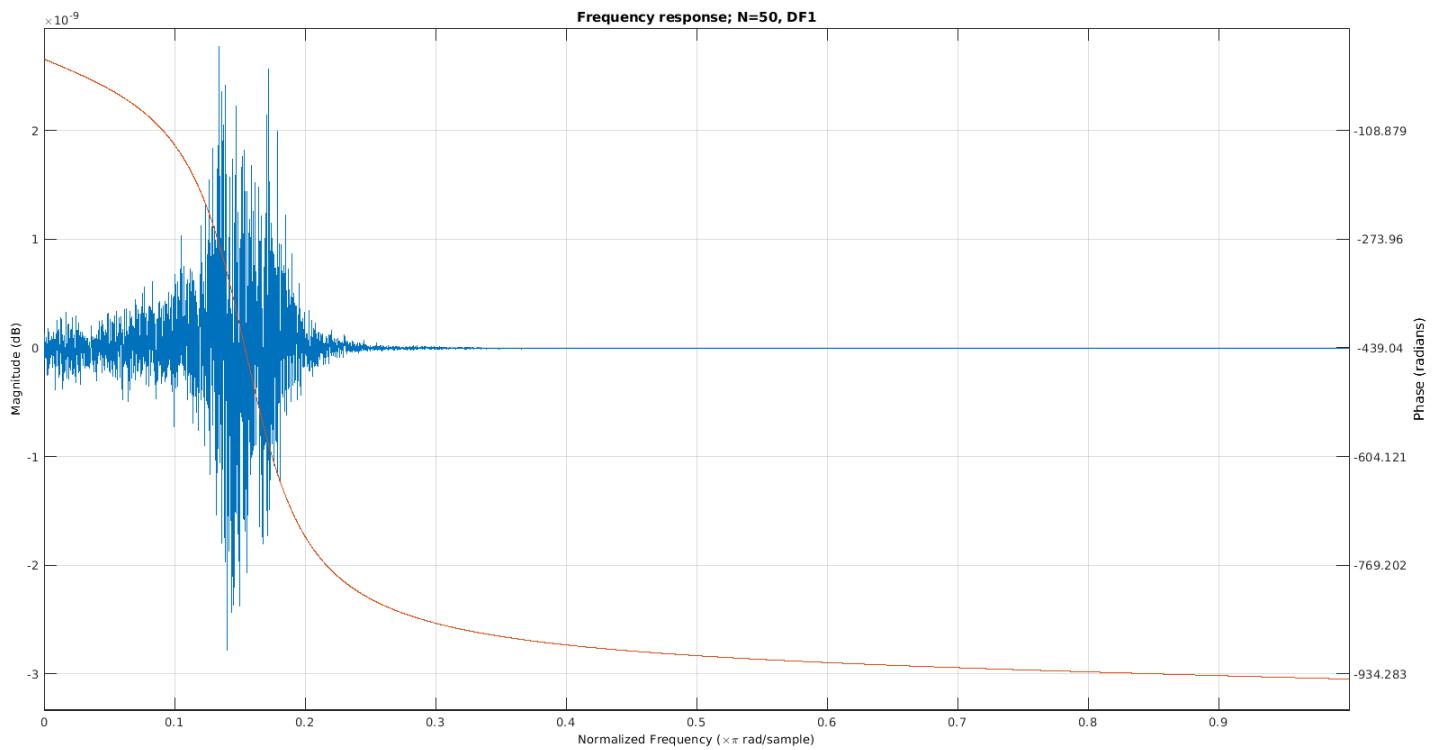
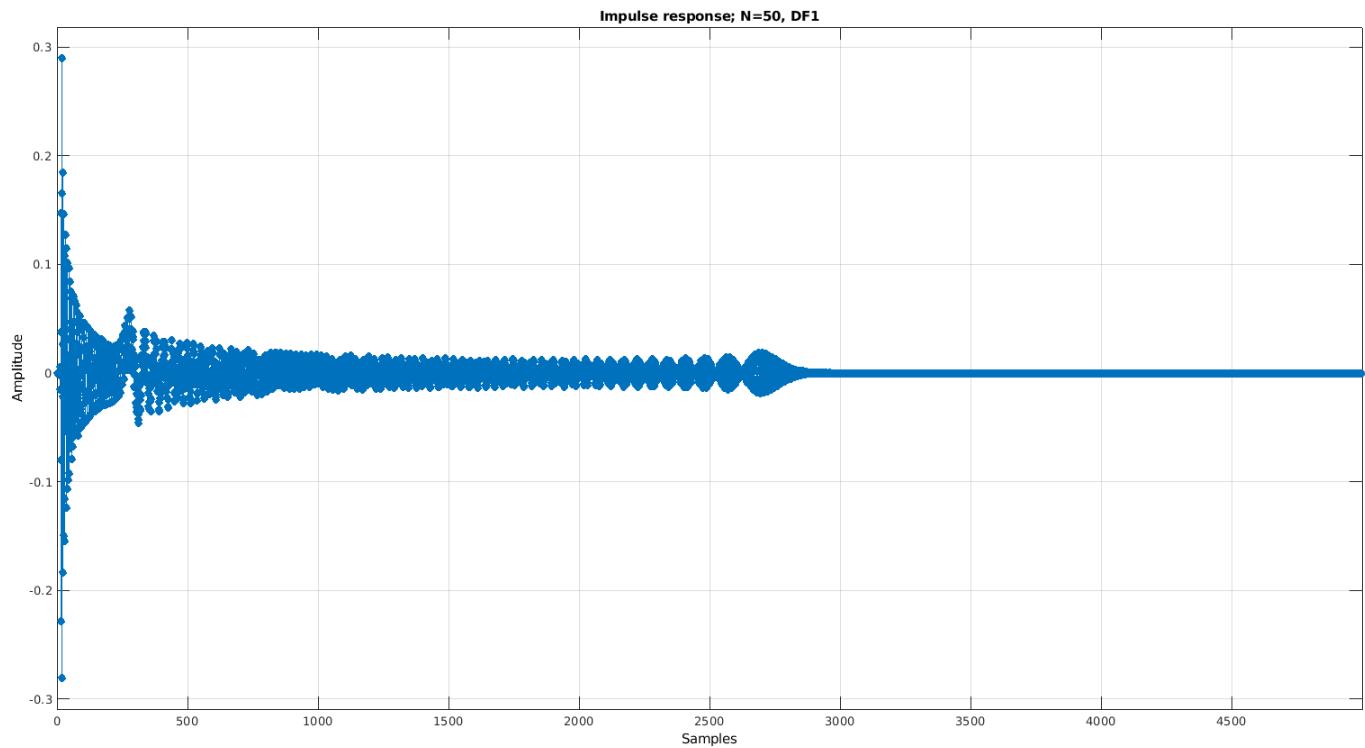
- With $N = 1$, the audio does sound roughly like the original, confirming that an all-pass filter with some phase distortion sounds like the original signal.
- When attempting generating the fiftieth-order transfer function polynomials using convolution of the first-order polynomials and plugging those into the `dfilt` objects, the resulting functions were very bad (as expected – bad numerical stability).
- The graphs and resulting audio produced by all of the implementations for $N = 50$ roughly all look and sound the same. The filtered audio sounds warbled/alienish, but roughly the same volume. Looking at the group delay plot, the audio at $\approx 0.15\text{rad}$ (263Hz) is now greatly delayed; since this falls within the range of human speech, it makes sense that the text sounds jumbled.

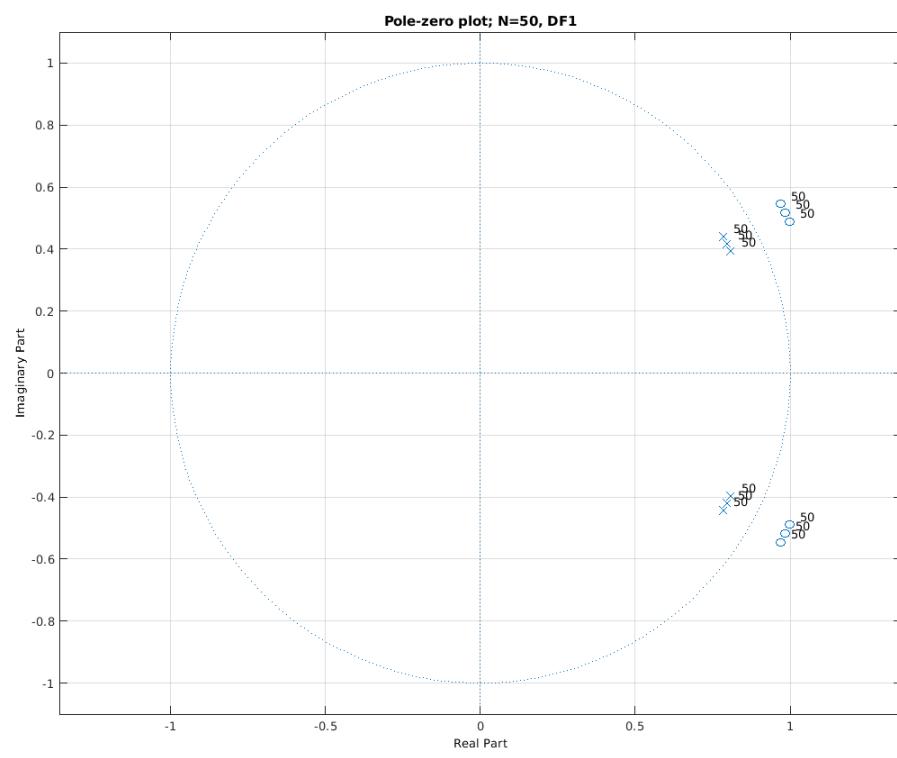
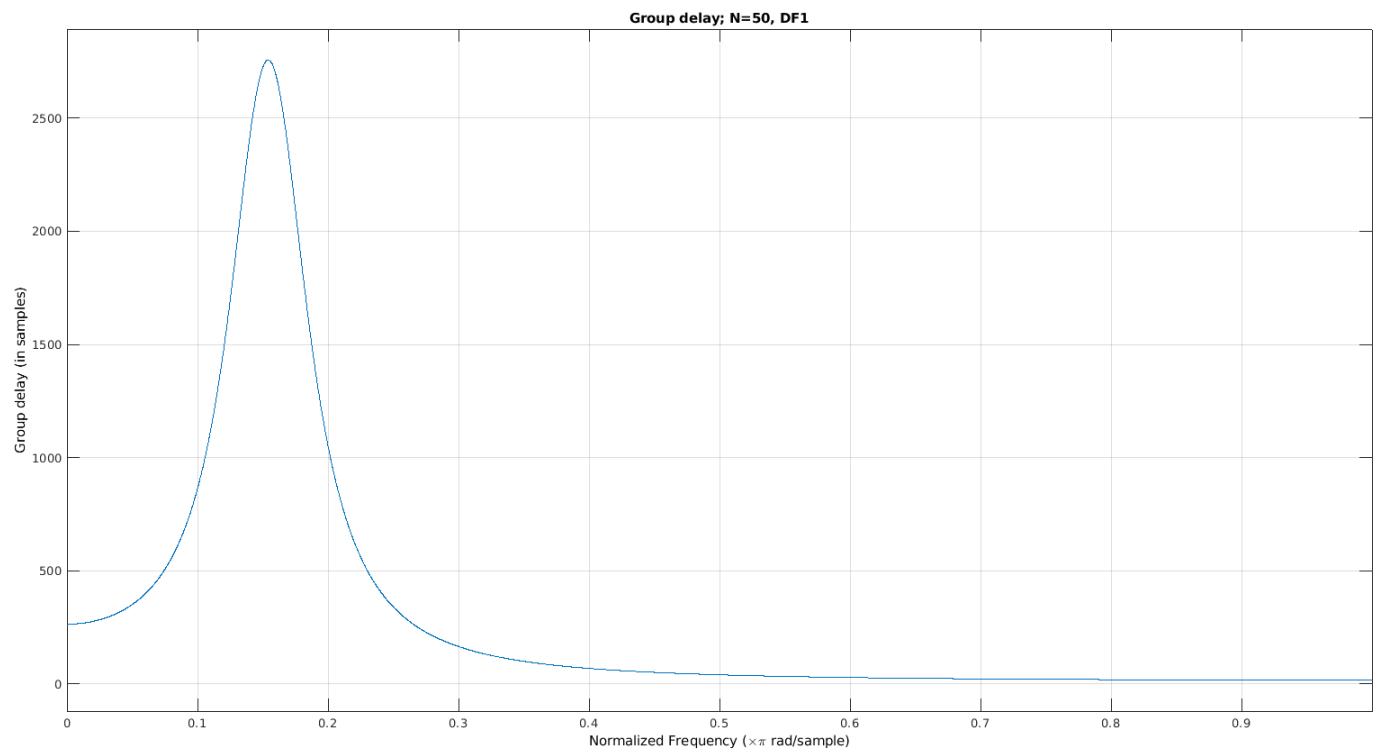
Figures

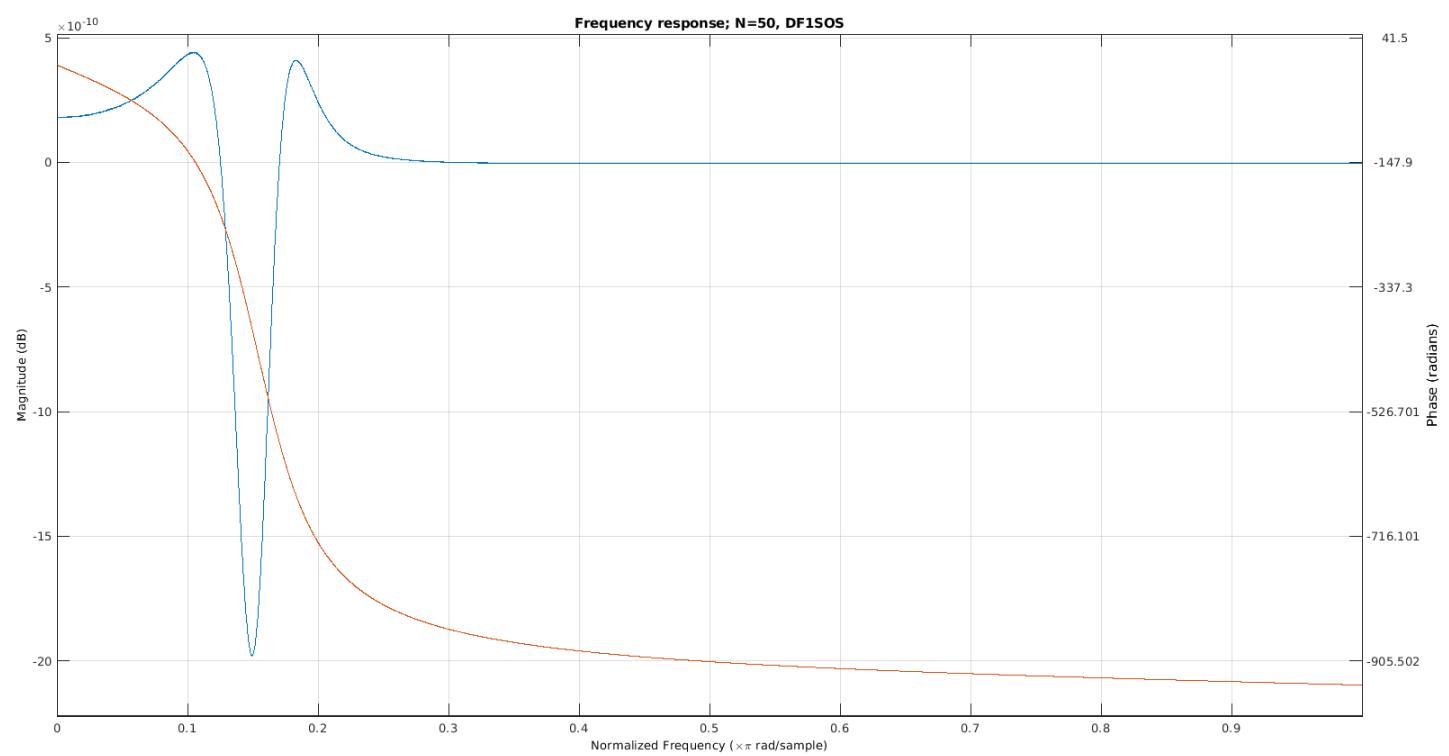
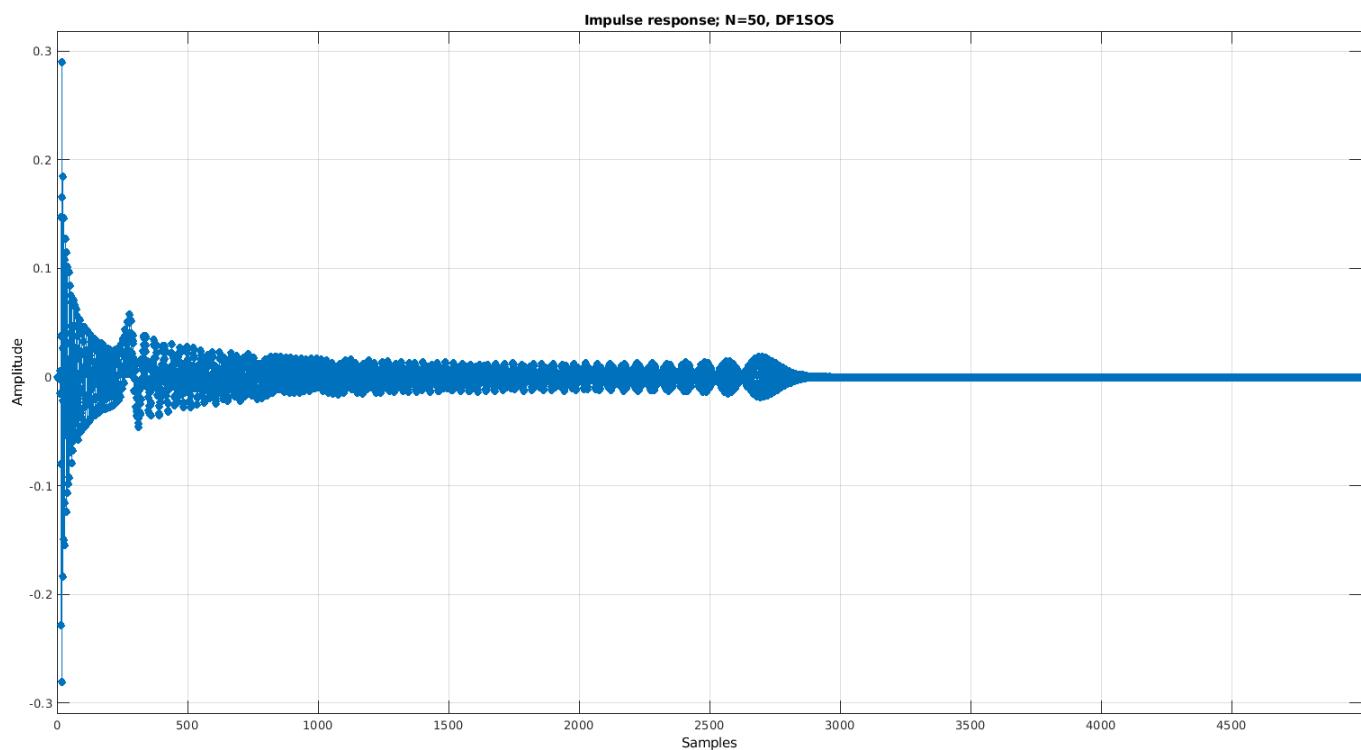
Figures are shown on the next page.

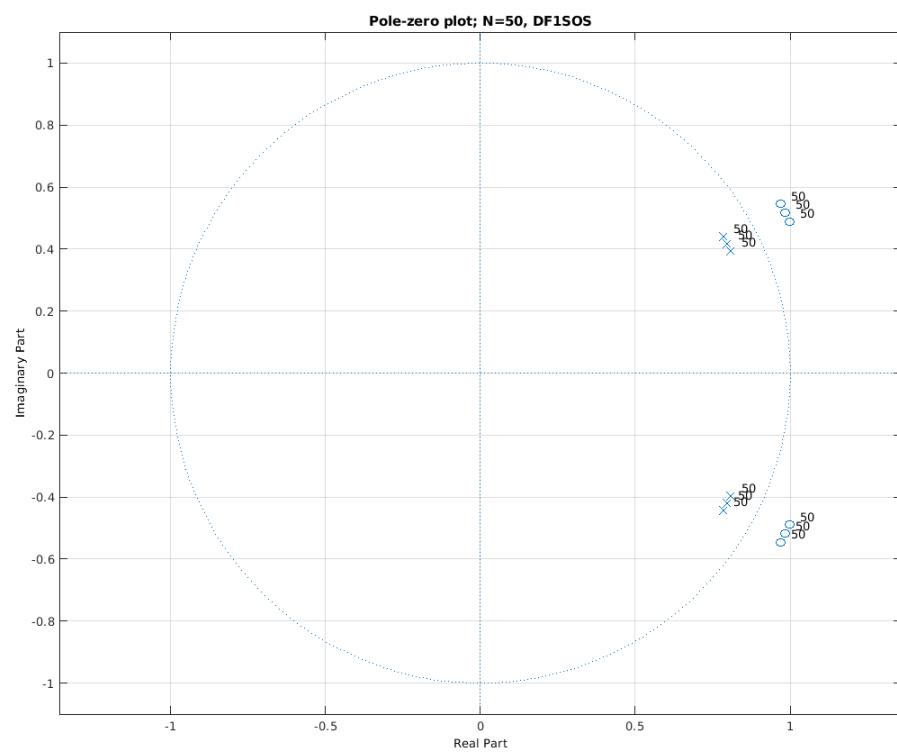
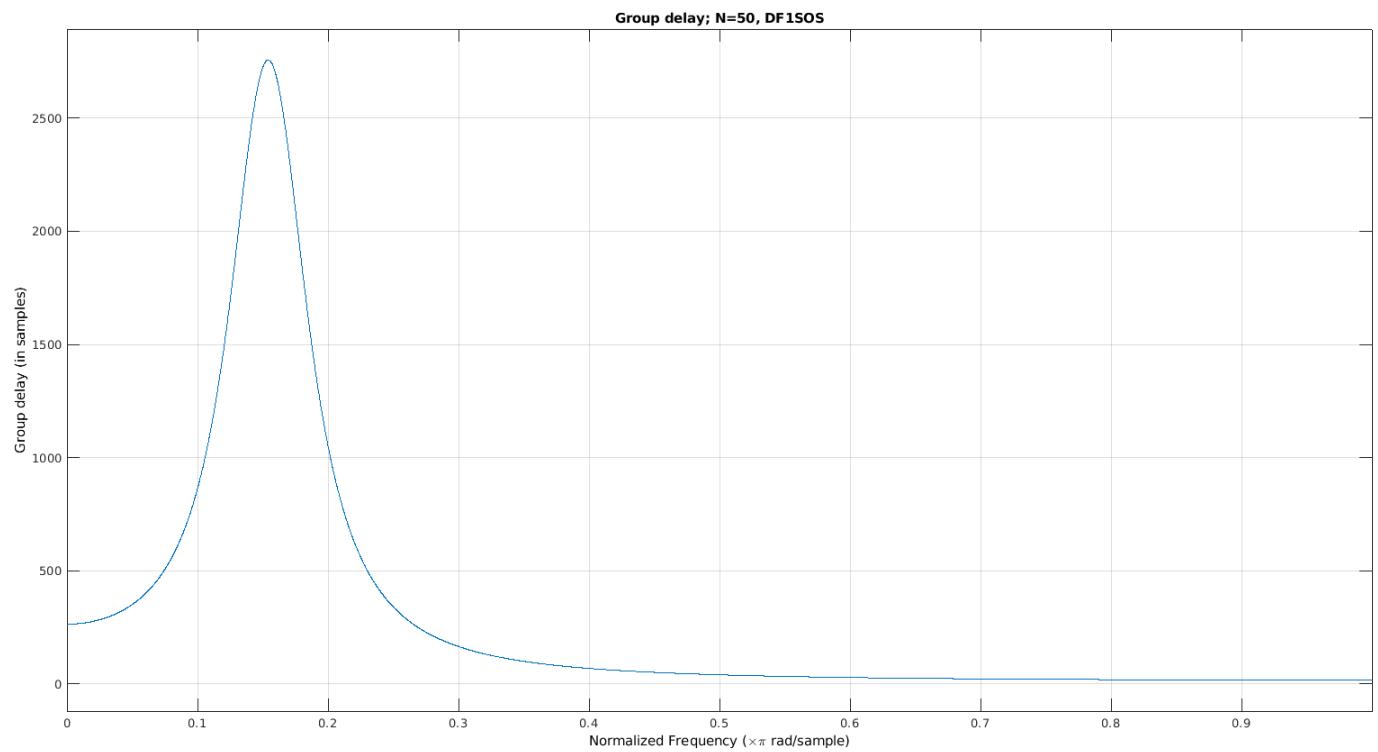


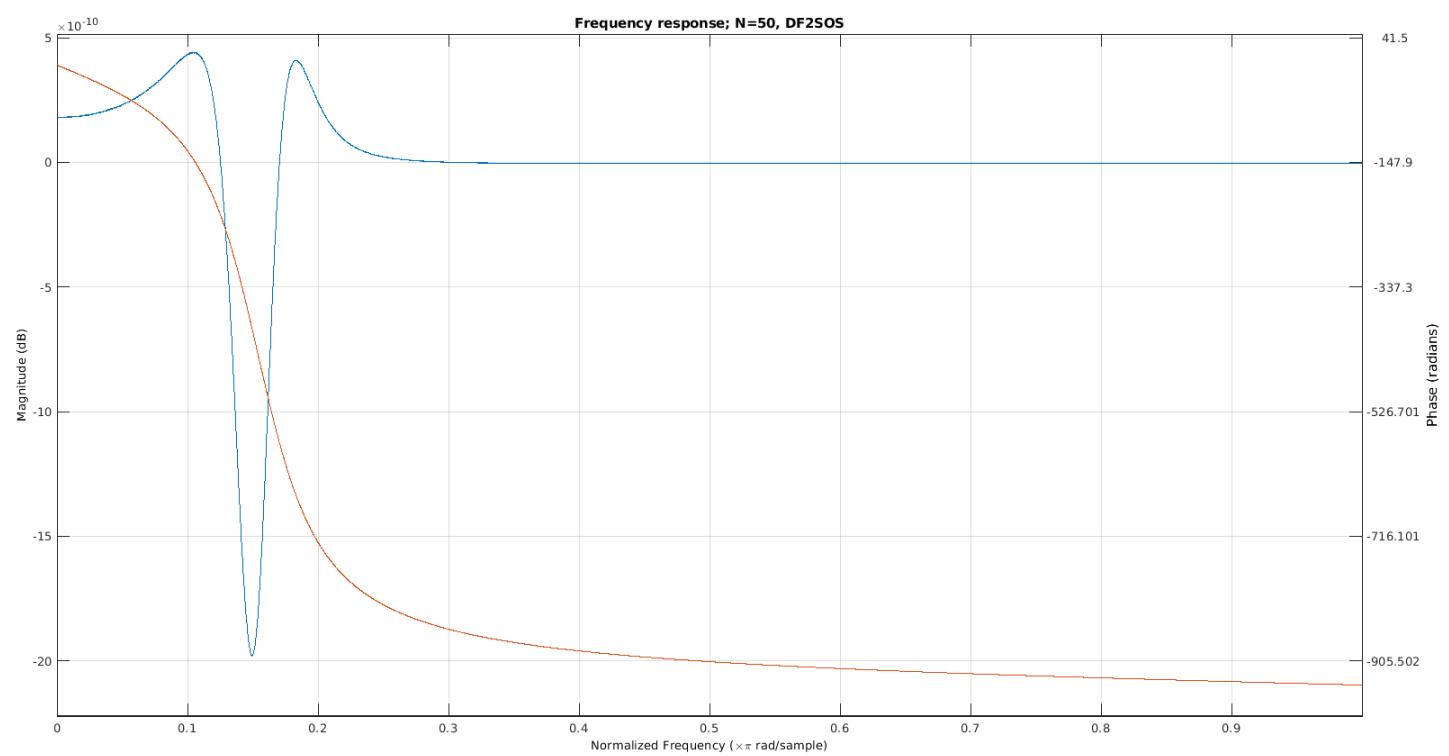
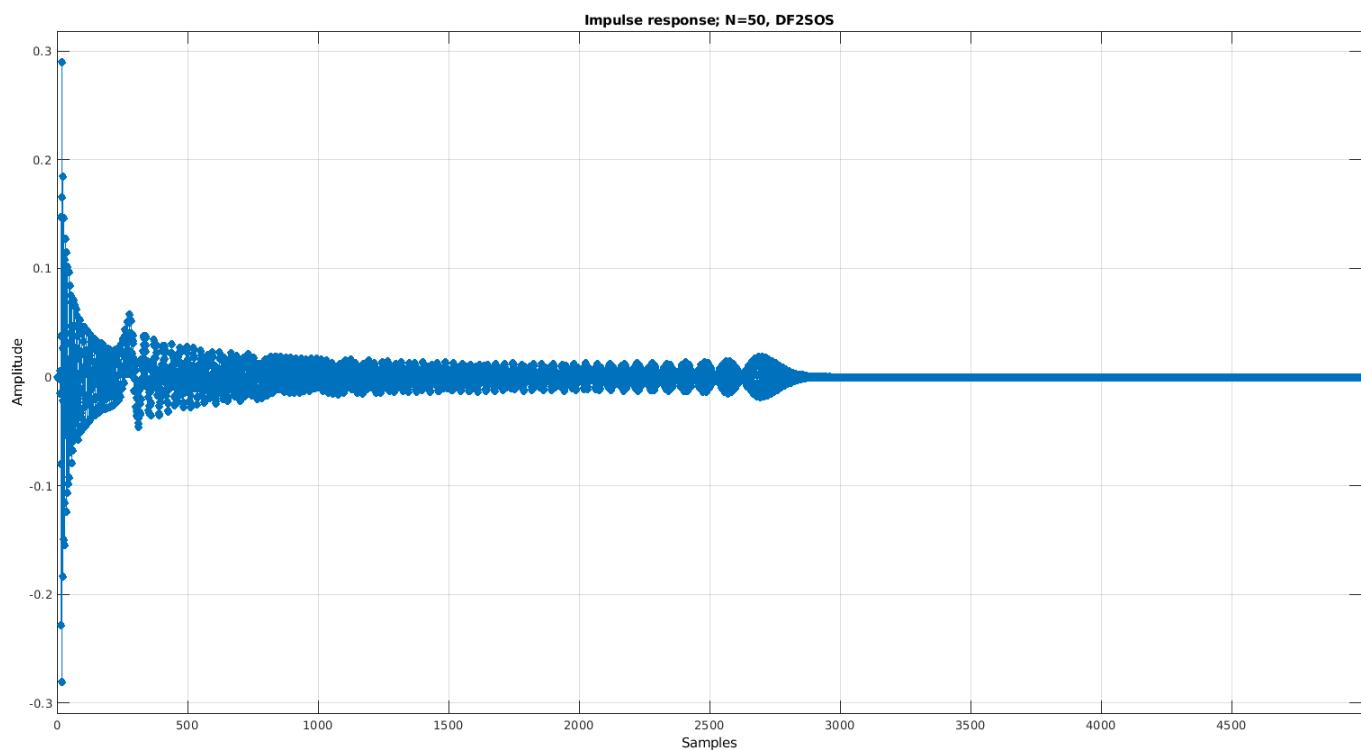


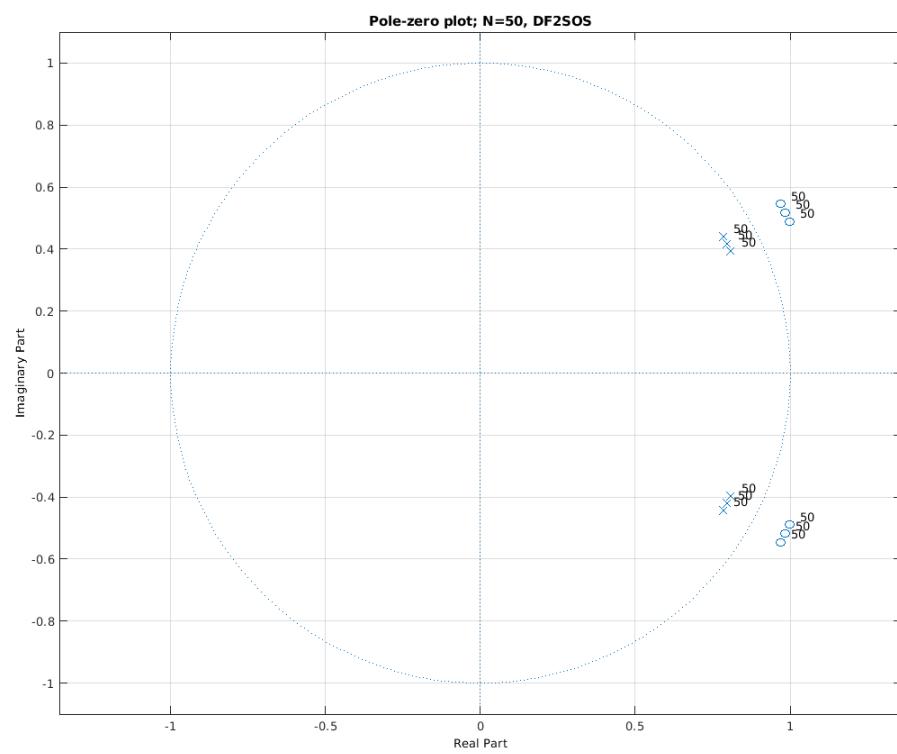
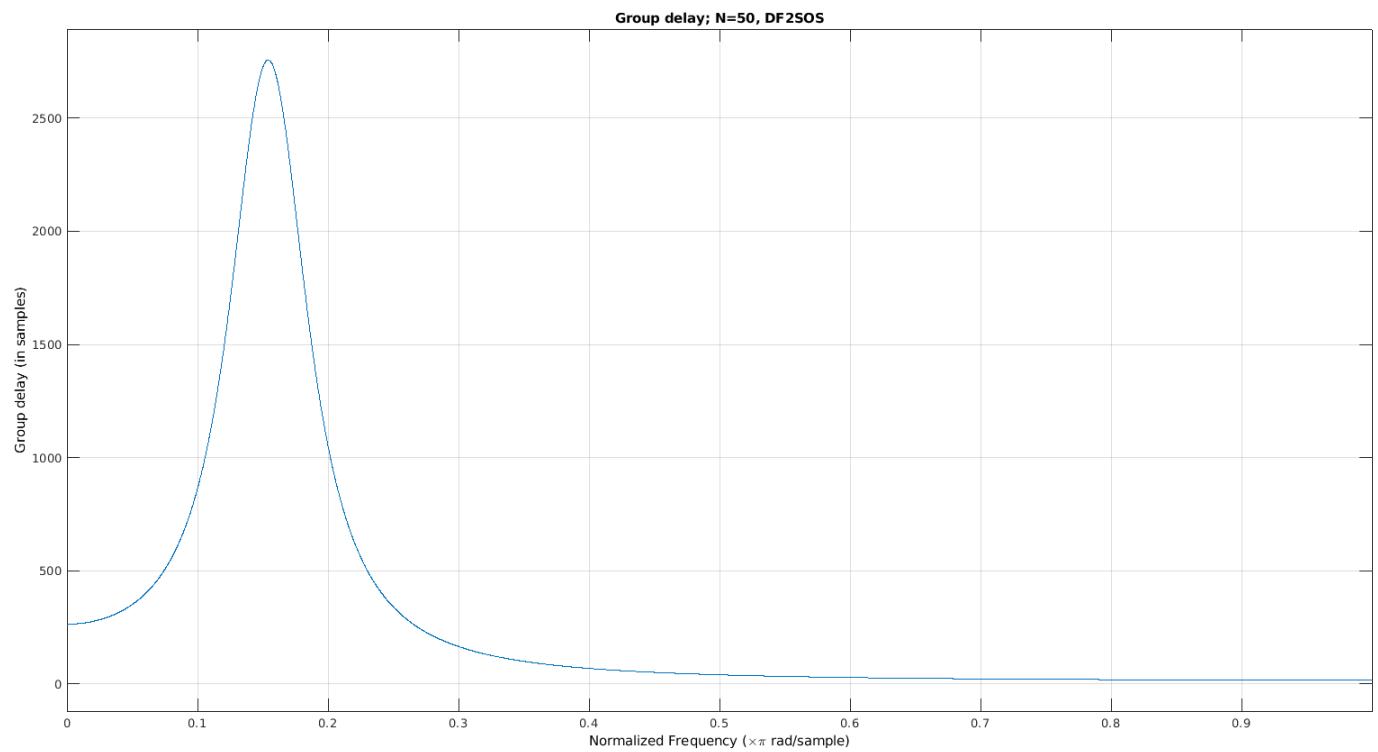


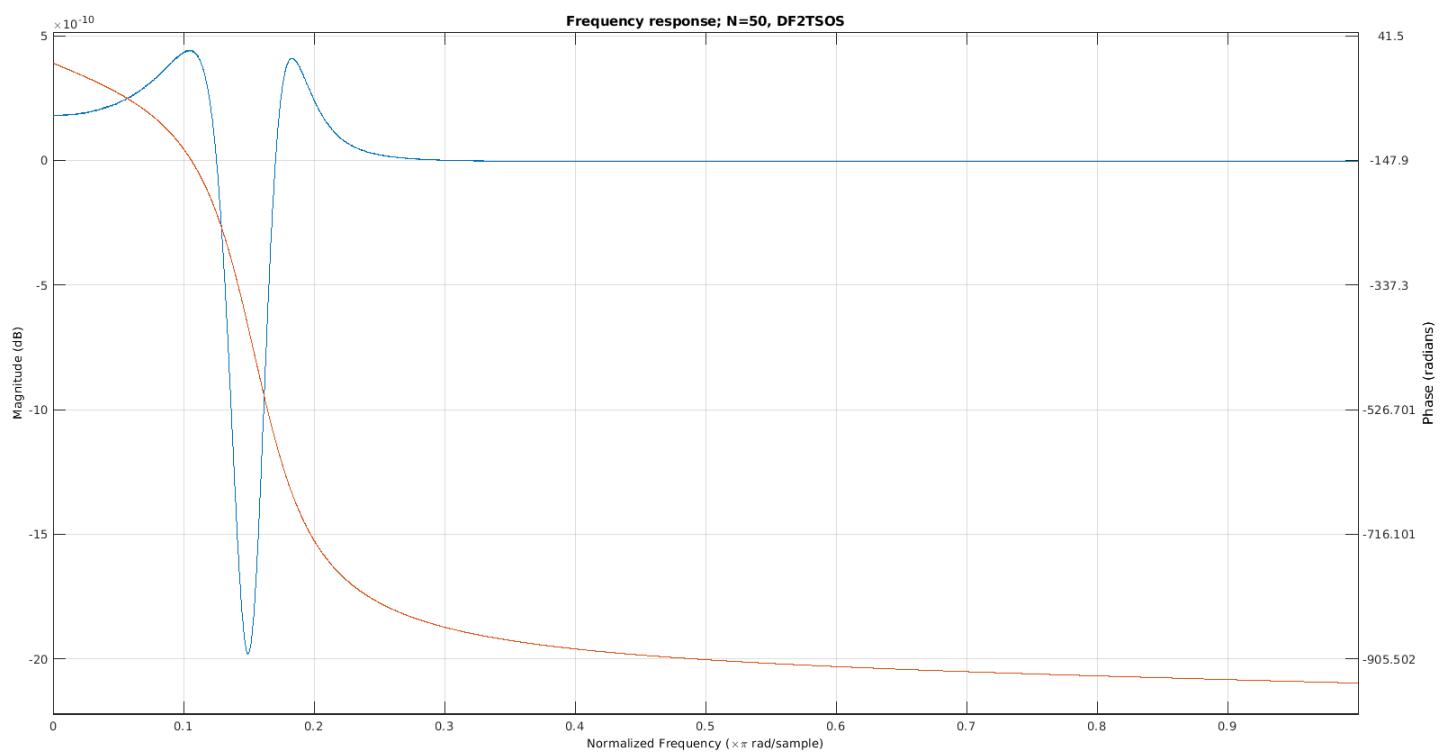
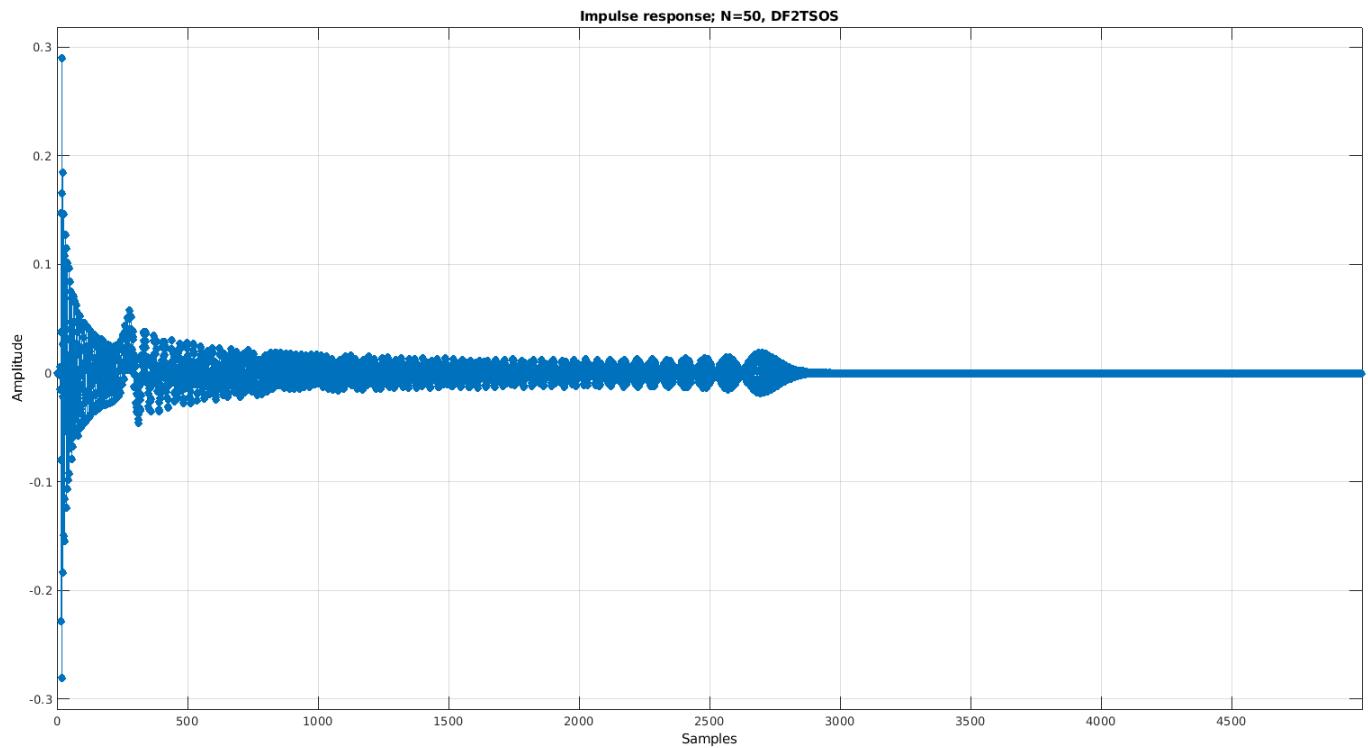


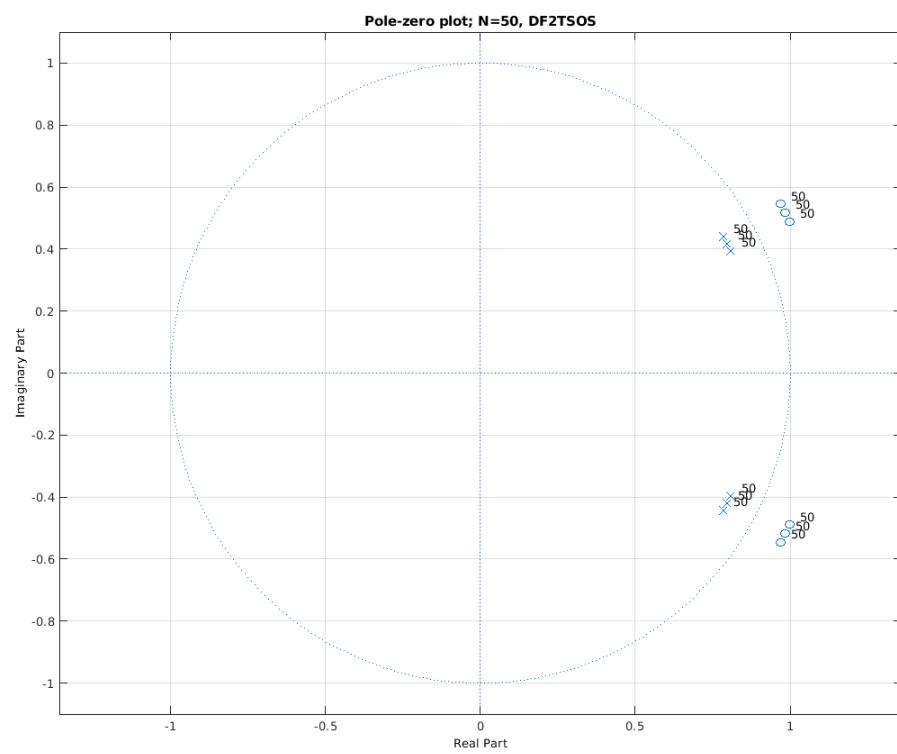
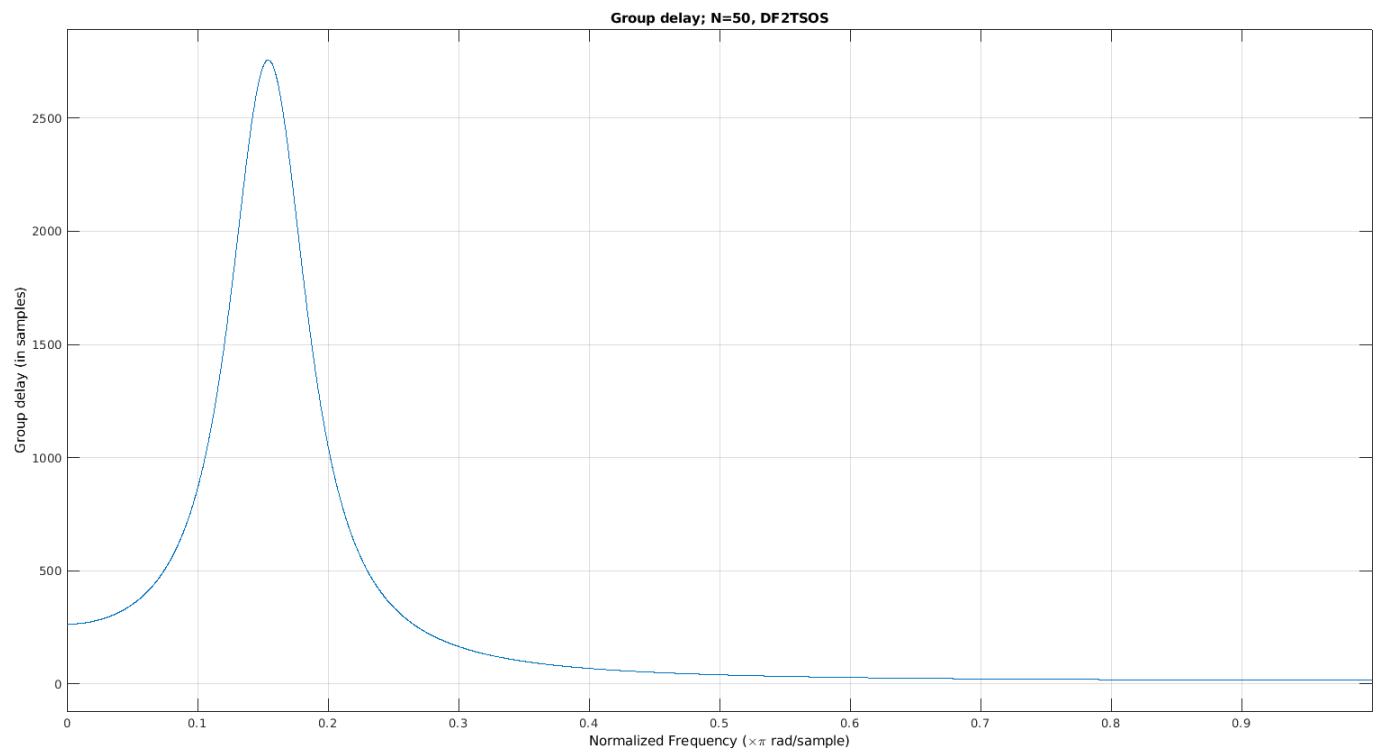












Source code

```
clc; close all; clear;
load('projIA.mat');

%% create direct form 1 filter
Hd = dfilt.df1(b, a);
plot_filter(Hd, 100, 'N=1');

%% save audio to file
audiowrite('n1.wav', filter(b, a, speech), fs);

%% attempted convolution
b50 = b;
a50 = a;
for i = 1:49
    b50 = conv(b, b50);
    a50 = conv(a, a50);
end % doesn't work; bad numerical stability

% used for second-order sections filters
[s, g] = tf2sos(b, a);

%% direct form 1
Hd_df1 = cascade_filter(dfilt.df1(b, a));
plot_filter(Hd_df1, 5000, 'N=50, DF1');
audiowrite('df1.wav', filter(Hd_df1, speech), fs);

%% direct form 1 (second-order sections)
Hd_df1sos = cascade_filter(dfilt.df1sos(s, g));
plot_filter(Hd_df1sos, 5000, 'N=50, DF1SOS');
audiowrite('df1sos.wav', filter(Hd_df1sos, speech), fs);

%% direct form 2 (second-order sections)
Hd_df2sos = cascade_filter(dfilt.df2sos(s, g));
plot_filter(Hd_df2sos, 5000, 'N=50, DF2SOS');
audiowrite('df2sos.wav', filter(Hd_df2sos, speech), fs);

%% direct form 2 transposed (second-order sections)
Hd_df2tsos = cascade_filter(dfilt.df2tsos(s, g));
plot_filter(Hd_df2tsos, 5000, 'N=50, DF2TSOS');
audiowrite('df2tsos.wav', filter(Hd_df2tsos, speech), fs);

%% cascade filter 50 times
function Hd = cascade_filter(Hd)
    Hd = dfilt.cascade(repmat(Hd, 1, 50));
end

%% plot impulse response, frequency response, group delay, and
% pole-zero plot of filter
function plot_filter(Hd, N, figname)
    impz(Hd, N);
    title(sprintf('Impulse response; %s', figname));
    freqz(Hd);
    title(sprintf('Frequency response; %s', figname));
    grpdelay(Hd);
    title(sprintf('Group delay; %s', figname));
```

```
zplane(Hd);
title(sprintf('Pole-zero plot; %s', figname));
end
```

ECE310 – Project 3

Jonathan Lam

November 8, 2020

1 Filter specs

Desired IIR digital filter specs

```
% filter specs
Wp = 2500 / (fs/2);
Ws = 4000 / (fs/2);
Rp = 3;
Rs = 95;
```

Desired FIR filter specs

```
% ripple needs to be in linear units for the FIR spec
% for passband, it's the difference between 1 and the linear value of the
% attenuation factor
Rp_linear_hat = 1-10^(-Rp/20);
% for stopband, it's the linear value of the attenuation factor
Rs_linear_hat = 10^(-Rs/20);

% using the equations from exercis 7.3 in the textbook to convert to the
% FIR spec (these are all in linear units):
% Rp = Rp_hat/(2-Rp_hat)
% Rs = 2Rs_hat/(2-Rp_hat)
Rp_linear = Rp_linear_hat/(2-Rp_linear_hat);
Rs_linear = 2*Rs_linear_hat/(2-Rp_linear_hat);

f = [Wp Ws];
a = [1 0];
```

2 Filter design and implementation

Butterworth filter. The multiplication by a factor of 100 is used to produce the desired 40dB gain in the passband. The filter is implemented using second-order-sections and a direct-form II transposed implementation.

```
%% butterworth
[n, Wn] = buttord(Wp, Ws, Rp, Rs);
[z, p, k] = butter(n, Wn);
k = k * 100;
```

```

%% implementation as df2t sos cascaded
butter_filter = dfilt.df2tsos(zp2sos(z, p, k));
butter_filtered = filter(butter_filter, noisy);

```

Chebyshev types I and II filters

```

%% cheby1
[n, cheby1_Wp] = cheblord(Wp, Ws, Rp, Rs);
[z, p, k] = cheby1(n, Rp, cheby1_Wp);
k = k * 100;

cheby1_filter = dfilt.df2sos(zp2sos(z, p, k));
cheby1_filtered = filter(cheby1_filter, noisy);

%% cheby2
[n, cheby2_Ws] = cheb2ord(Wp, Ws, Rp, Rs);
[z, p, k] = cheby2(n, Rs, cheby2_Ws);
k = k * 100;

cheby2_filter = dfilt.df2sos(zp2sos(z, p, k));
cheby2_filtered = filter(cheby2_filter, noisy);

```

Elliptic filter

```

%% elliptic
[n, ellip_Wp] = ellipord(Wp, Ws, Rp, Rs);
[z, p, k] = ellip(n, Rp, Rs, ellip_Wp);
k = k * 100;

ellip_filter = dfilt.df2sos(zp2sos(z, p, k));
ellip_filtered = filter(ellip_filter, noisy);

```

Parks-McClellan filter. Note that the +6 is a fudge factor; documentation says if the order calculated from firpmord doesn't match the spec, try increasing the order. (See: <https://www.mathworks.com/help/signal/ref/firpm.html>) The division by max(abs(H)) and subsequent multiplication by 100 is used to scale the maximum gain in the passband down to 0dB and then up to the desired 40dB.

```

%% parks mclellan
pm_filter = firpm(n+6, fo, ao, w);
H = freqz(pm_filter);
pm_filter = pm_filter / max(abs(H)) * 100;

pm_filtered = conv(pm_filter, noisy);

```

Kaiser filter

```

%% kaiser
[n, Wn, beta, ftype] = kaiserord(f, a, [Rp_linear Rs_linear]);
kaiser_filter = fir1(n, Wn, ftype, kaiser(n+1, beta), 'noscale');
H = freqz(kaiser_filter);
kaiser_filter = kaiser_filter / max(abs(H)) * 100;

kaiser_filtered = conv(kaiser_filter, noisy);

```

3 Plotting code

Plot filter given its zpk (for IIR filters). Note that the zpk form is used for zplane, since it is most stable. However, the impz, freqz, and grpdelay don't support the zpk form for its input arguments, so the second-order-systems matrix is used instead. (The SOS is also used to implement the IIR filters.)

```
function sos = plot_filter_zpk(z, p, k, name)
    sos = zp2sos(z, p, k);

    fig = figure('Visible', 'Off');
    tiledlayout(2, 2, 'TileSpacing', 'compact');

    nexttile();
    [h, t] = impz(sos, 100);
    stem(t, h);
    title('Impulse Response');
    ylabel('Amplitude');
    xlabel('n (samples)');

    nexttile();
    [H, w] = freqz(sos);
    plot(w, 20*log10(abs(H)));
    title('Frequency Response');
    ylabel('Magnitude (dB)');
    xlabel('Digital Frequency');

    nexttile();
    zplane(z, p, k);
    title('Pole-Zero Plot');

    nexttile();
    grpdelay(sos);
    title('Group Delay');

    set(fig, 'PaperUnits', 'centimeters');
    set(fig, 'PaperPosition', [0 0 30 20]);
    saveas(fig, sprintf('fig_%s.eps', name));
end
```

Plot filter given its impulse response (for FIR filters).

```
function plot_h(h, name)
    fig = figure('Visible', 'Off');
    tiledlayout(2, 2, 'TileSpacing', 'compact');

    nexttile();
    stem(h);
    title('Impulse Response');
    ylabel('Amplitude');
    xlabel('n (samples)');

    nexttile();
    [H, w] = freqz(h);
    plot(w, 20*log10(abs(H)));
    title('Frequency Response');
    ylabel('Magnitude (dB)');
```

```
xlabel('Digital Frequency');

nexttile();
zplane(h);
title('Pole-Zero Plot');

nexttile();
grpdelay(h);
title('Group Delay');

set(fig, 'PaperUnits', 'centimeters');
set(fig, 'PaperPosition', [0 0 30 20]);
saveas(fig, sprintf('fig_%s.eps', name));

end
```

4 Plots

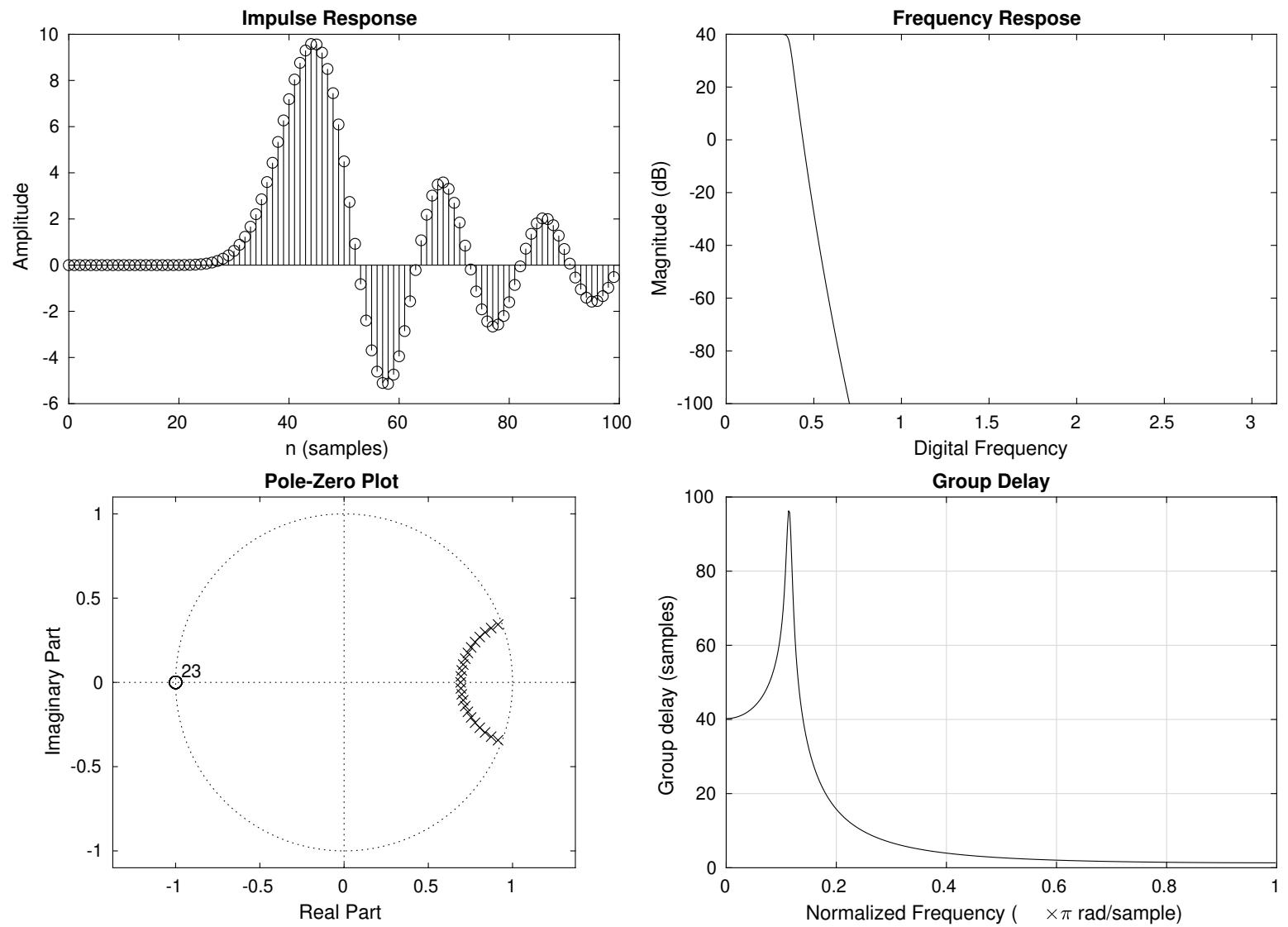


Figure 1: Butterworth filter

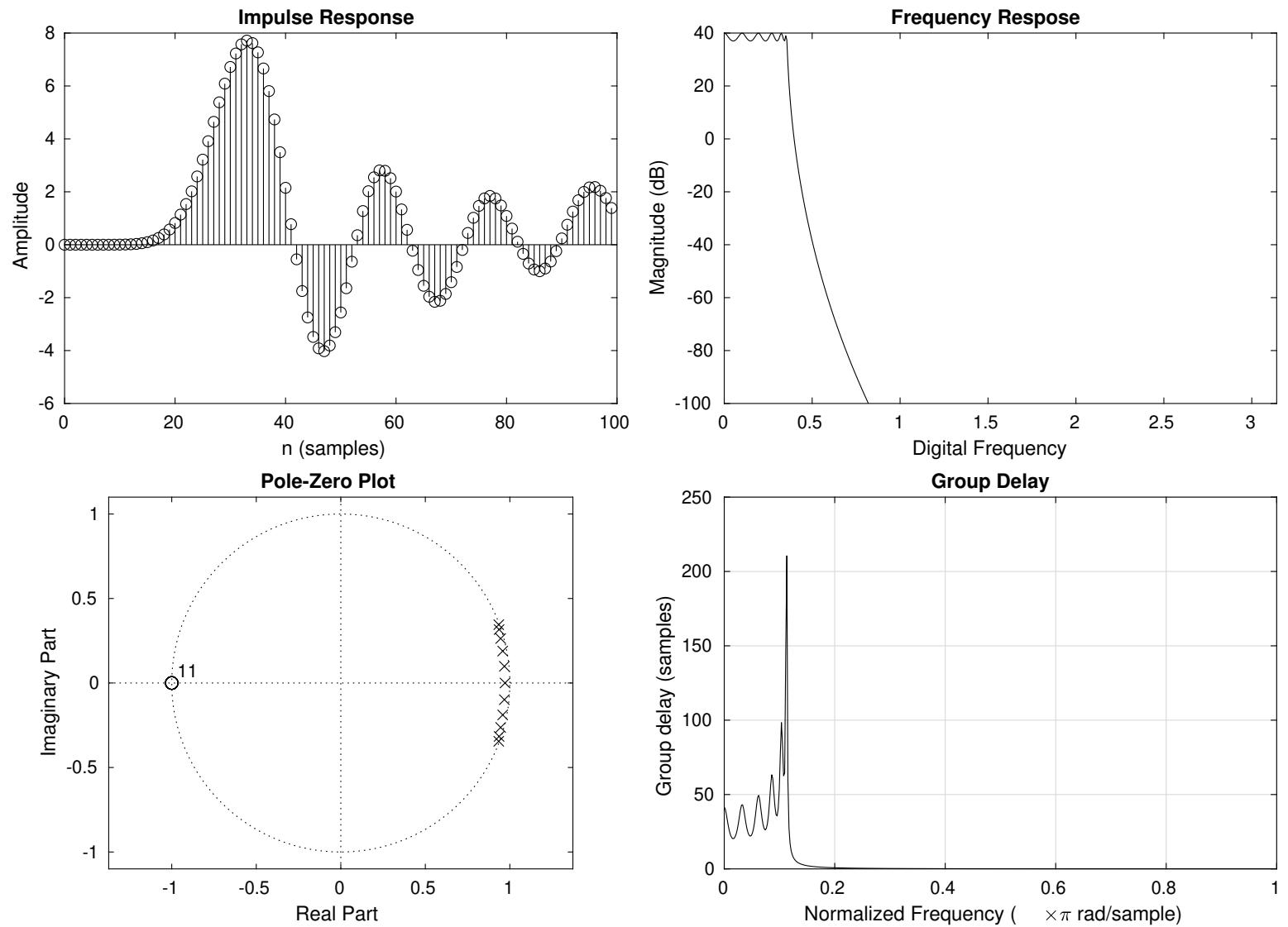


Figure 2: Chebyshev Type I Filter

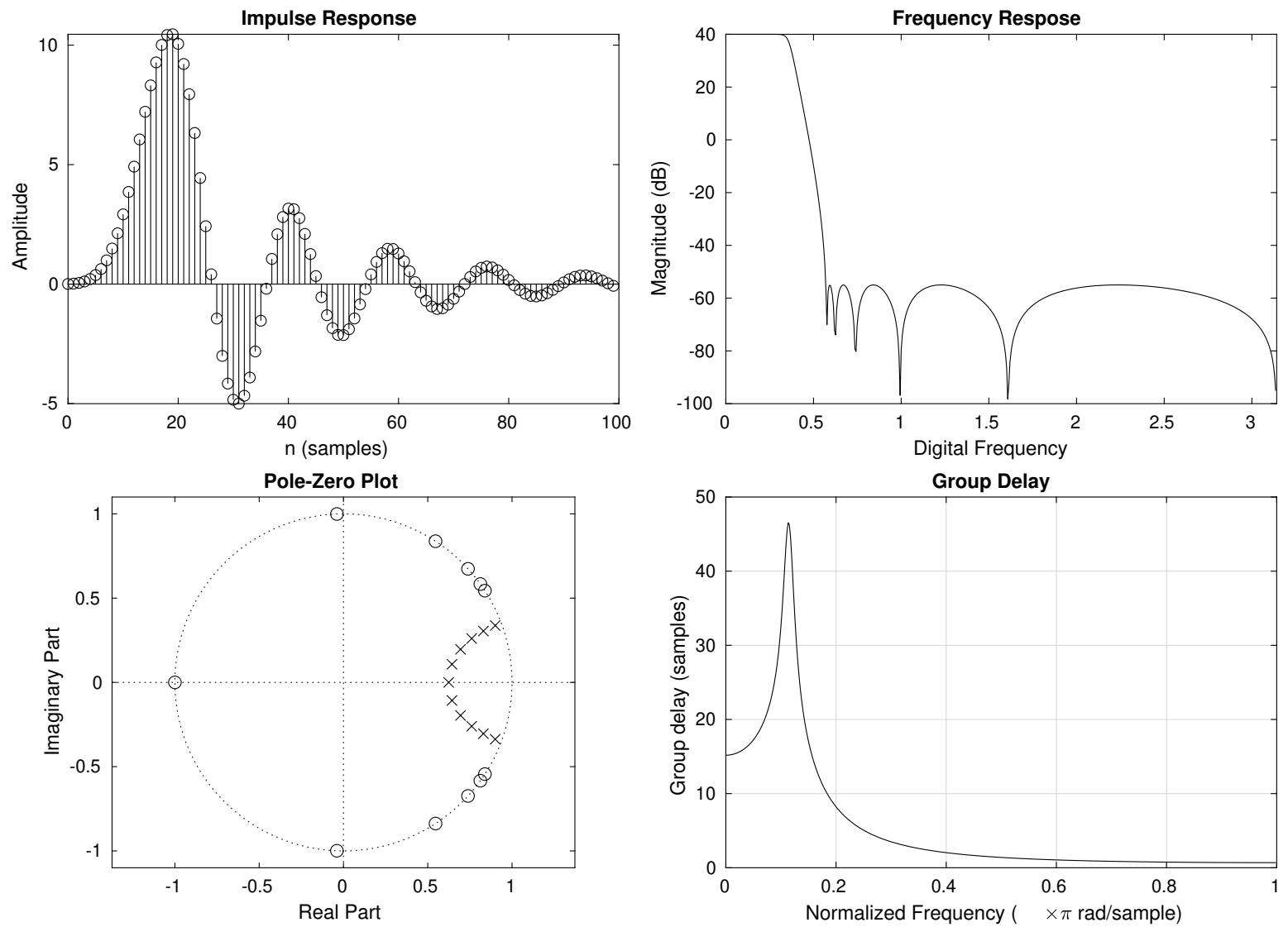


Figure 3: Chebyshev Type II Filter

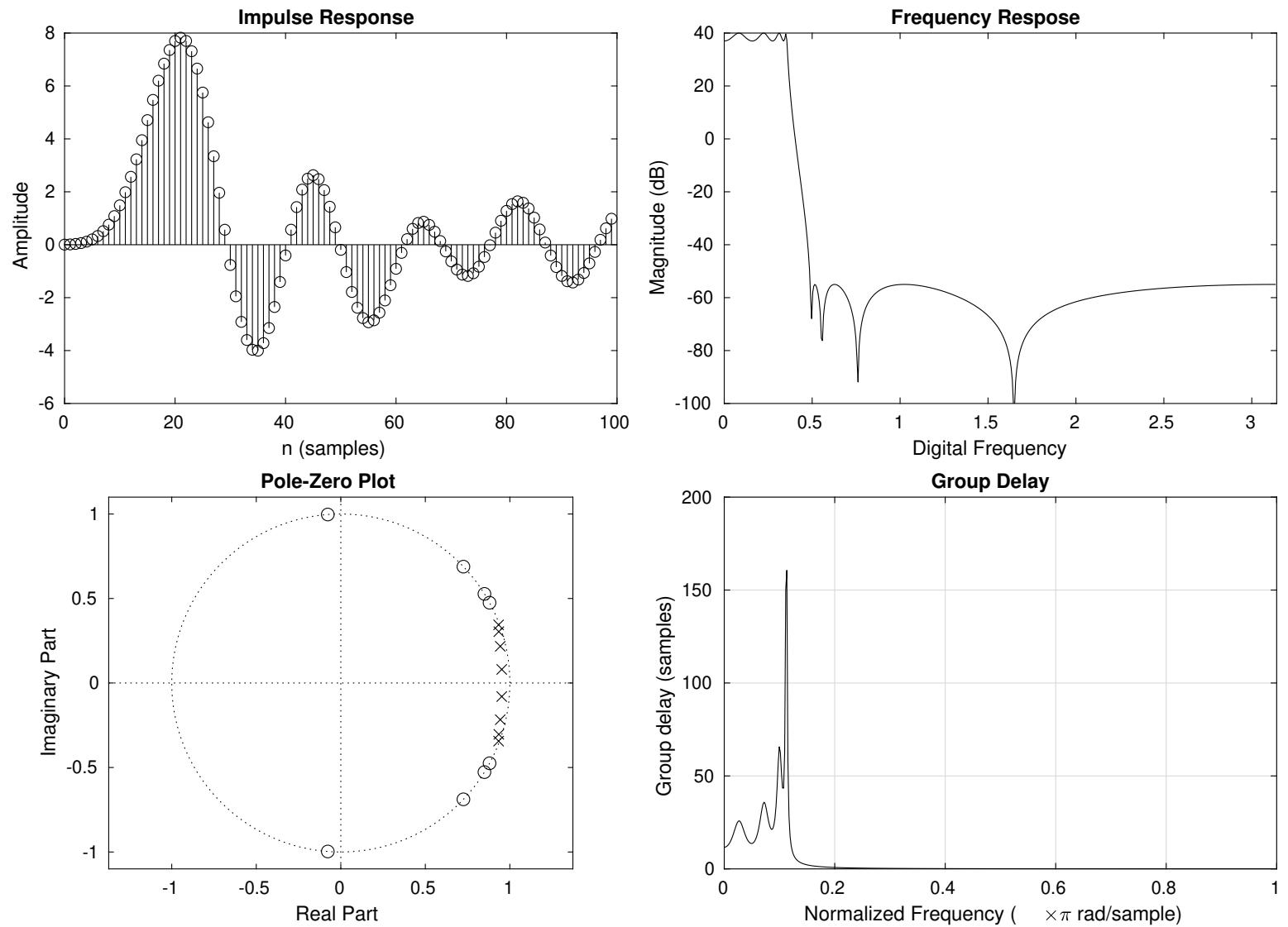


Figure 4: Elliptical Filter

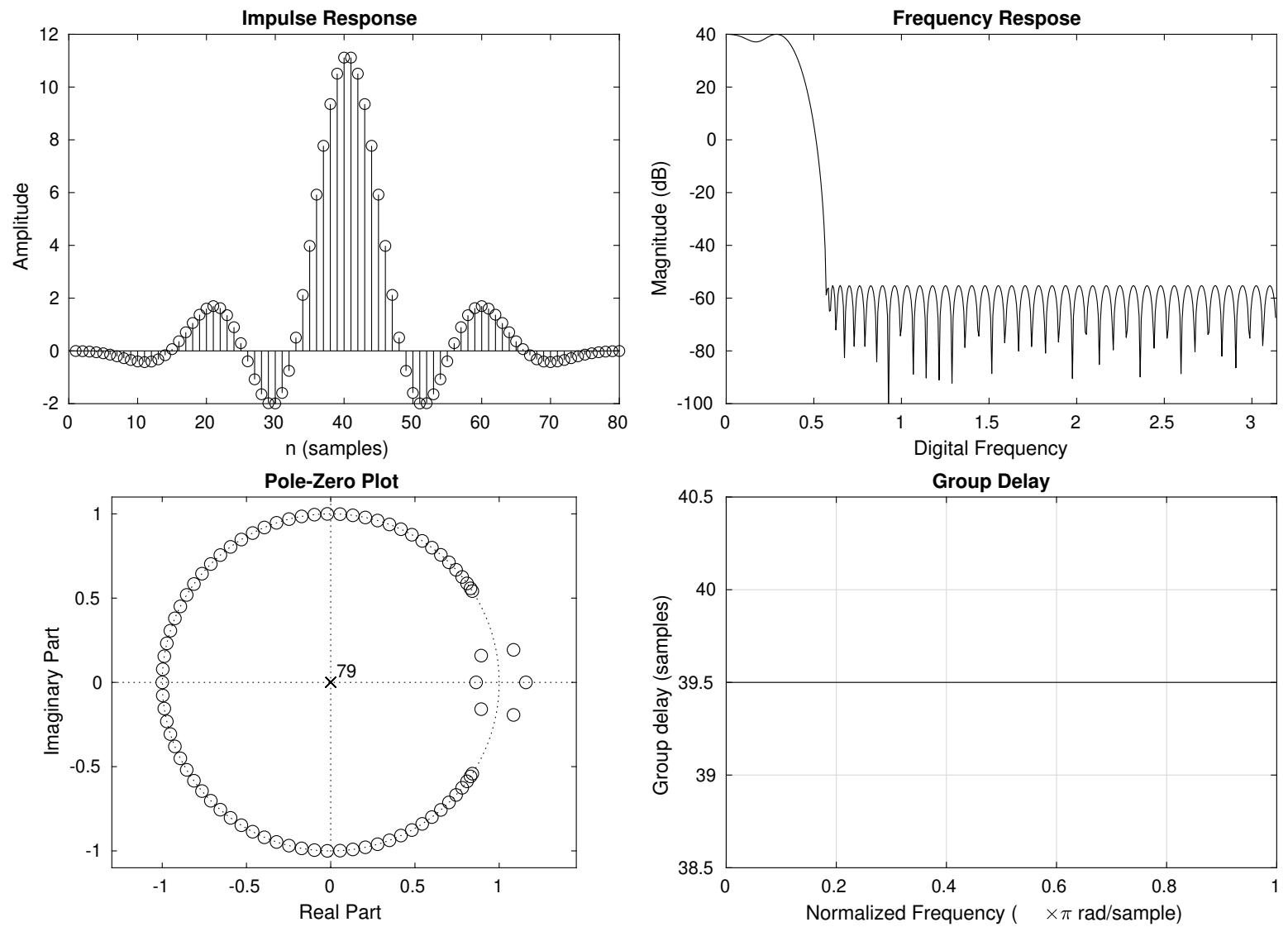


Figure 5: Parks-McClellan Filter

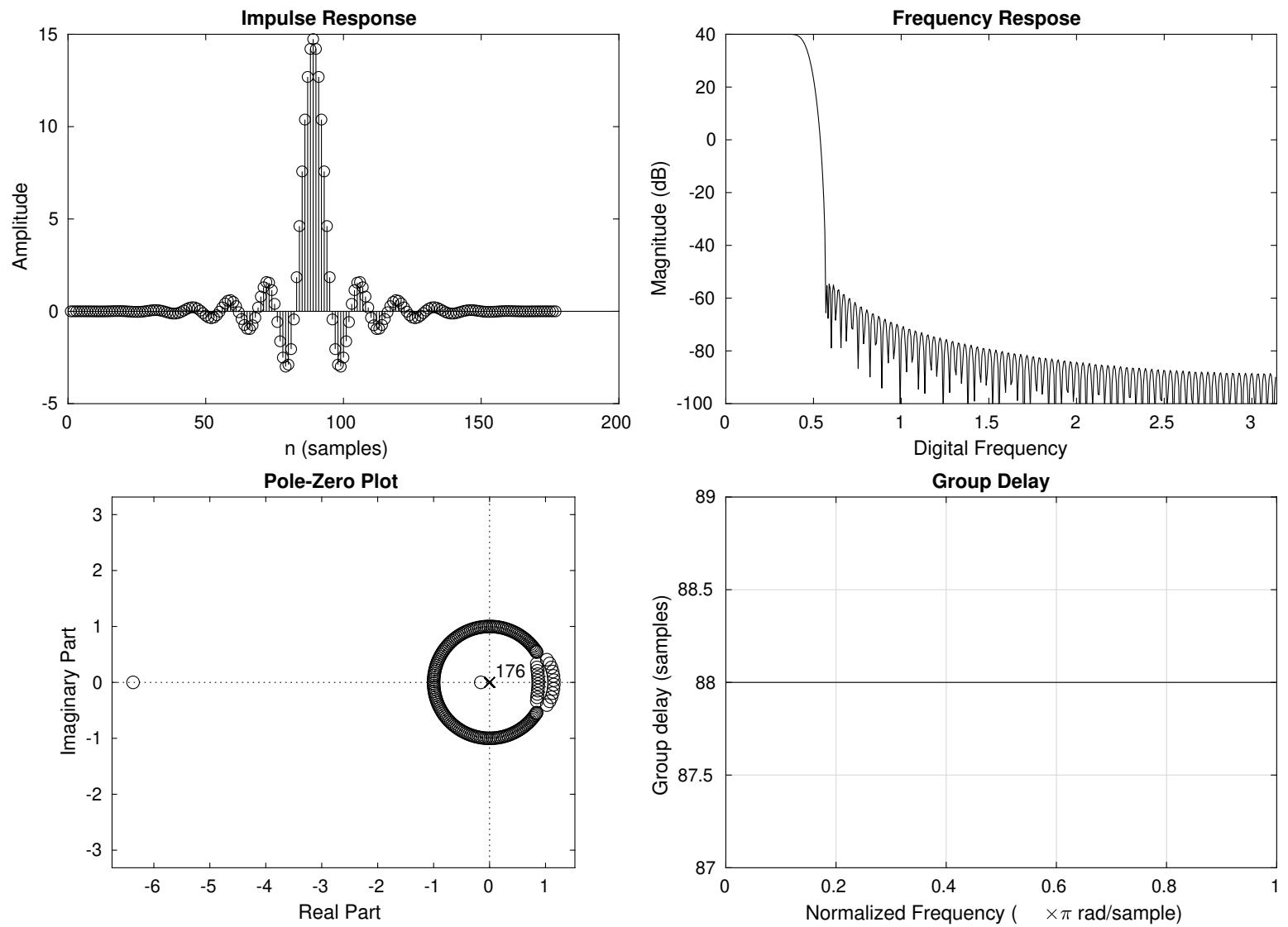


Figure 6: Kaiser Window Filter

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 $200\mu\text{s}$ with $f_s = 5\text{MHz}$:

```
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.

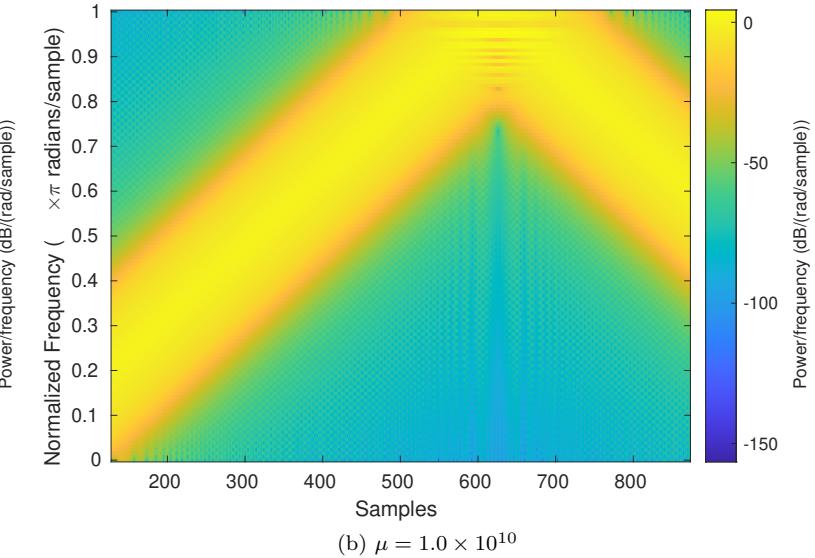
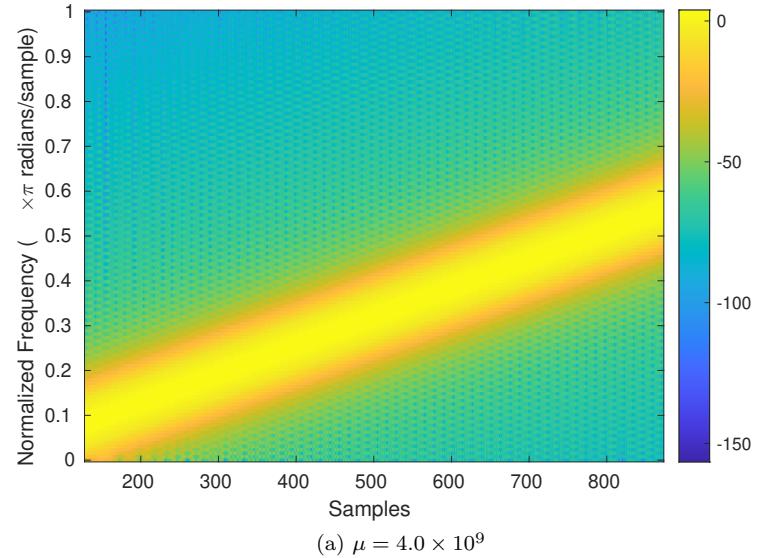


Figure 1: Chirp spectrograms

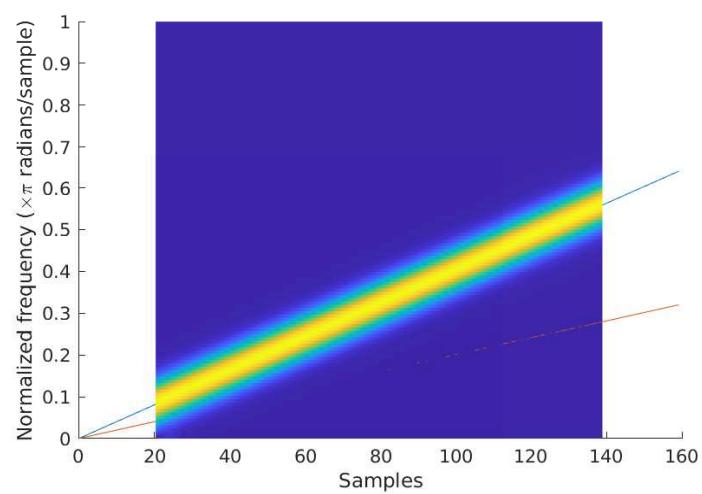


Figure 2: Chirp spectrogram slope

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');
```

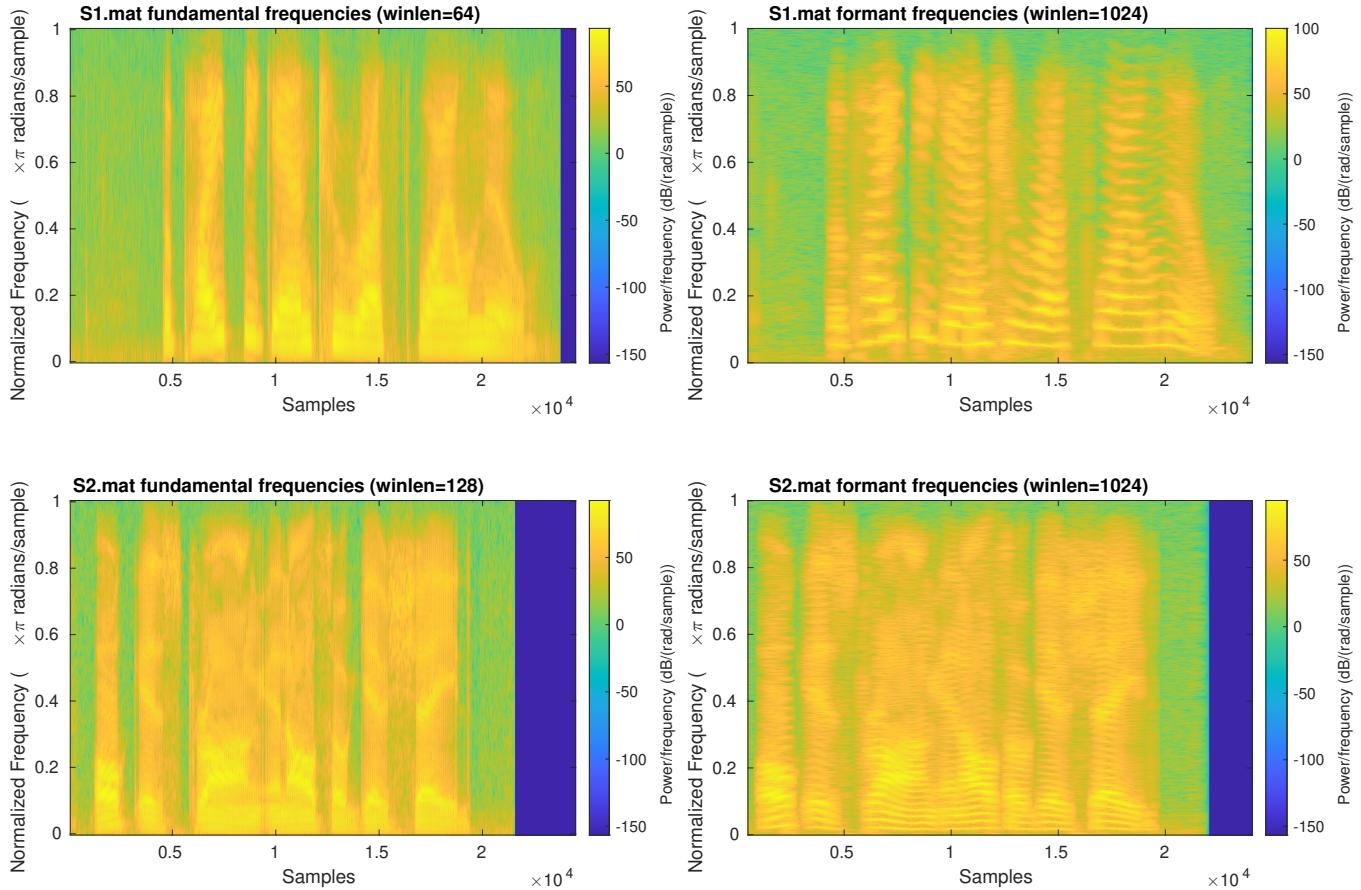


Figure 3: Speech analysis spectrograms

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);
```

```

% 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

ECE310 – Project 5

Jonathan Lam

December 20, 2020

1 Setup

(Denote the correlation of signal s by $\phi_s(\tau)$, and the PSD of a signal by $\Phi_s(\omega)$.)

We have a random Gaussian signal x s.t. $\Phi(x)$ has $\sigma^2 = 1$. We also have a channel with some frequency response $|H(e^{j\omega})|^2$. If the result of x passed through the filter h is y , then:

$$\Phi_y(e^{j\omega}) = |H(e^{j\omega})|^2 \Phi_x(e^{j\omega}) = |H(e^{j\omega})|^2$$

Thus estimating $\Phi_y(e^{j\omega})$ is equivalent to estimating $|H(e^{j\omega})|^2$. We are given a 512-point signal $y[n]$ and use this to estimate $|H(e^{j\omega})|^2$ by estimating its PSD. We are also given a 512-point sample of $|H(e^{j\omega})|^2$ to check our estimates by the error criterion (1). These signals are displayed in Figure 1.

$$\epsilon = \frac{1}{N} \sum_{k=0}^{N-1} \left| \left| \hat{H}(e^{j\omega}) \right|^2 - \left| H(e^{j\omega}) \right|^2 \right|^2 \quad (1)$$

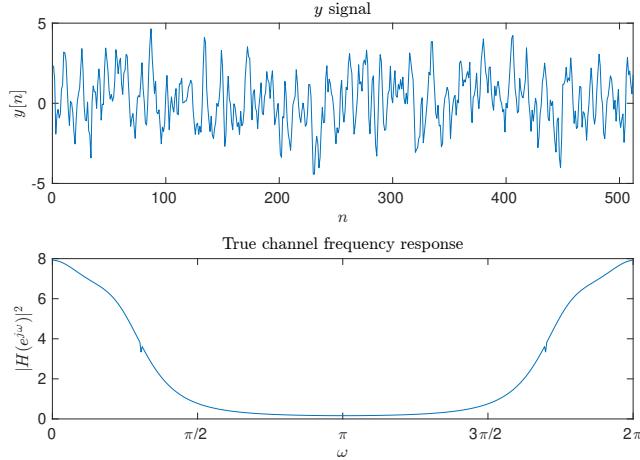


Figure 1: The provided signals $y[n]$ and $|H(e^{j\omega})|^2$

2 Autocorrelation and MATLAB (§A)

We can take the cross-correlation of x with itself (i.e., the autocorrelation) using the MATLAB function `xcorr`, which is similar to taking the convolution with the reverse of x (i.e., taking inner products at each shift).

```
y1 = y(1:32);
autocorr1 = xcorr(y1, y1, 'biased');
autocorr2 = conv(y1, flip(y1));
```

The difference is that the `xcorr` function normalizes to a biased estimate with $N = 32$, while the convolution does not.

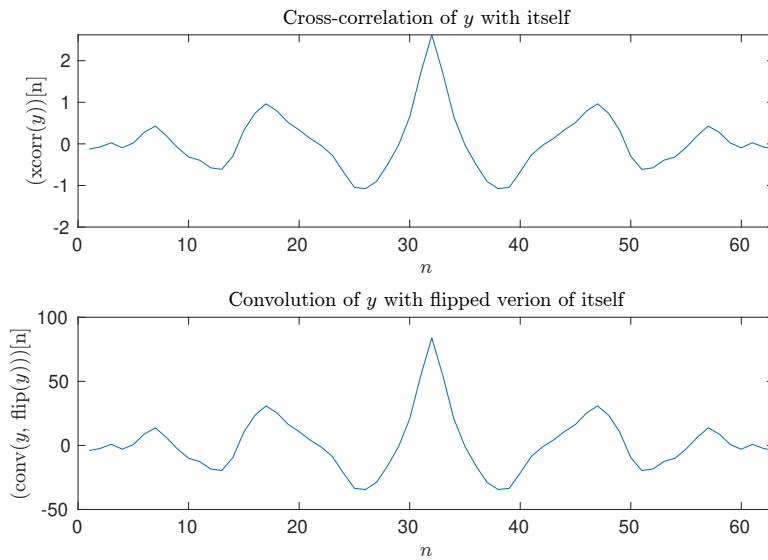


Figure 2: Comparison of `xcorr` vs `conv`

2.1 Biased vs. unbiased estimates (§A.1)

The term “biased” in this context means that the correlation is equal to the convolution scaled down by a factor of N , while an unbiased estimate is scaled down by a factor of $N - |n|$. The latter produces the expected value in the limit as $N \rightarrow \infty$ (hence “unbiased”), but suffers from large variance when n approaches N when a finite number of samples are used. The biased estimate does not approach the expected value in the limit, but it does not suffer from the large variance problem.

2.2 Deterministic autocorrelation (§A.2)

We can calculate the (deterministic) autocorrelation using:

$$\phi_s[m] = \frac{1}{N} \sum_{n=0}^{N-1} s[n+m]s[n]$$

A MATLAB implementation for this to calculate the autocorrelation of y_1 is:

```
phi_y1 = zeros(1, 32);
for i = 1:N_y1
    phi_y1(i) = sum(y1(1:N_y1-i+1) .* y1(i:N_y1)) / N_y1;
end
phi_y1 = [flip(phi_y1) phi_y1(2:end)];
```

This gives us exactly the same result as calculated by `xcorr` (which confirms that our result is correct).

The Fourier transform of this autocorrelation function is a positive, real function because it is the PSD (by Wiener-Khinchin), of which each point is positive and real (the power of a given frequency). However, when we plot the magnitude and phase, we do not get a positive real function – this is plotted in Figure 3. (This probably due to some symmetry of the problem.) (We do get linear phase, however – not sure if this is relevant.)

2.3 Plotting different PSD estimates (§A.3)

Three estimates of the PSD are plotted in Figure 4:

1. `abs(fft(xcorr(y1, 'biased'), 64))`
2. `abs(fft(y(1:32), 64)).^2`
3. `abs(fft(y(1:64), 64)).^2`

The first method uses Wiener-Khinchin to calculate the PSD by taking the FFT of the autocorrelation function. The second and third methods calculate the PSD directly by taking the power (magnitude squared) of the FFT. The third uses more samples of y to calculate the FFT.

It seems that the second method is less noisy than the first method. The third method, despite taking more samples of y , is more noisy than the second method; this is the strange property that is detailed in section 10.5.2 of the textbook, so taking fewer samples for the periodogram is actually better (less noisy).

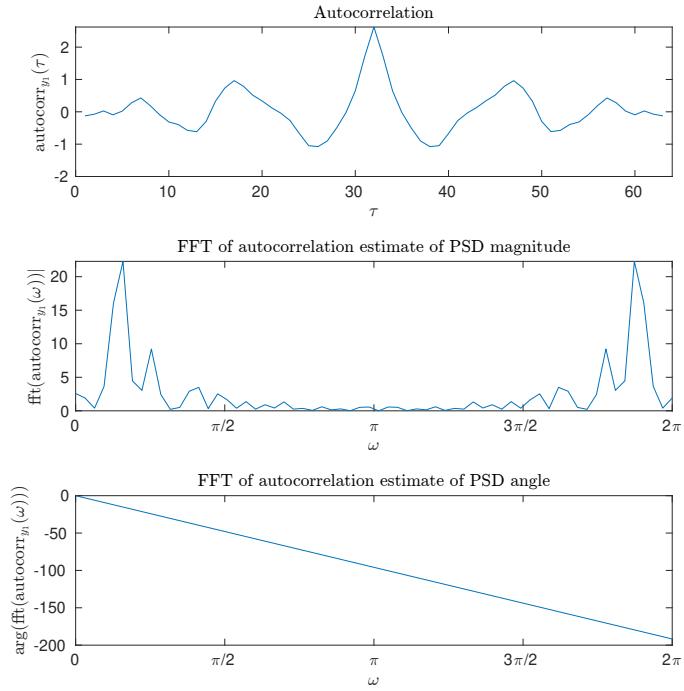


Figure 3: Autocorrelation and its FFT (estimate of PSD)

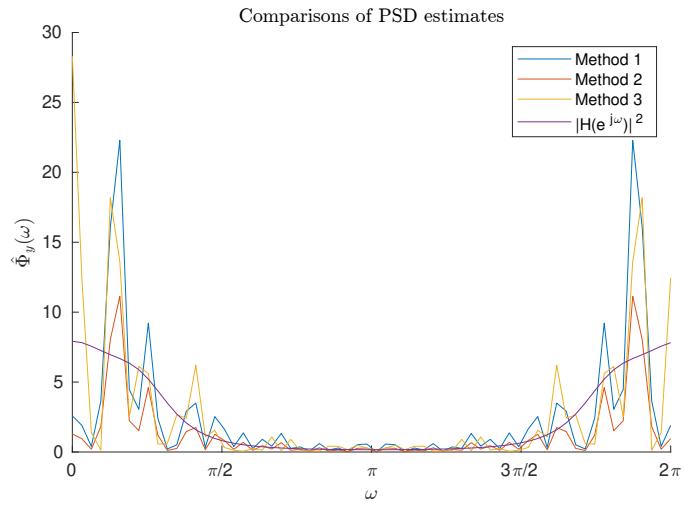


Figure 4: Comparison of the three PSD estimation methods described in §A.3

3 Nonparametric PSD estimation (§B)

3.1 PSD estimates with different sample and FFT lengths (§B.1-2)

The periodogram is calculated similar to the second method above (taking the FFT, and then finding its magnitude squared). We compare the result when 32 samples are taken (i.e., y_1) and a 64-point DFT is performed (same as method 2 in §A.3), and when all 512 samples are taken and a 1024-point DFT is performed. The results are shown in Figure 5. The same property is displayed as in Figure 4: more samples causes a noisier estimate of the PSD. These two methods have errors of 7.5039 and 7.5488 using the error measure (1).

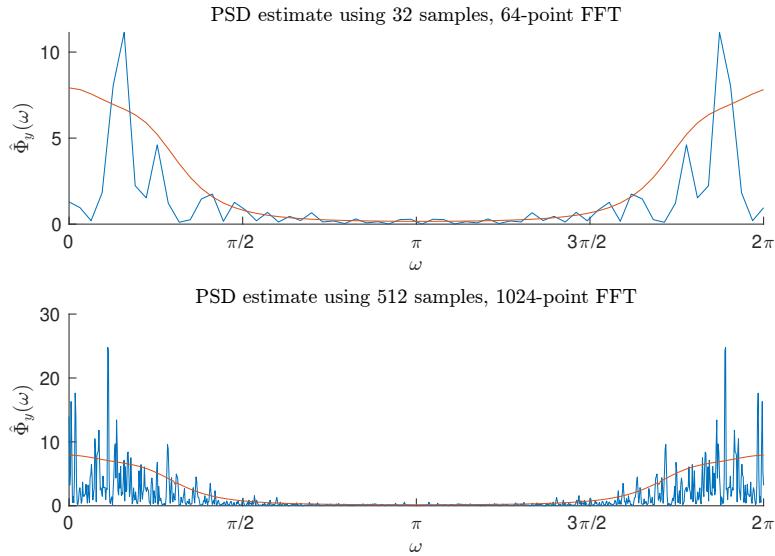


Figure 5: Comparison of PSD estimation methods with different numbers of samples and FFT sizes

3.2 Improving results by averaging (§B.3)

We can try to improve the results by taking multiple, shorter periodograms and averaging the results. Here, we break up the signal into 32-sample chunks, take the 64-point periodogram of each chunk, and average the results. This provides a smoother estimate and a lower error of 3.1403. The results are displayed in Figure 6.

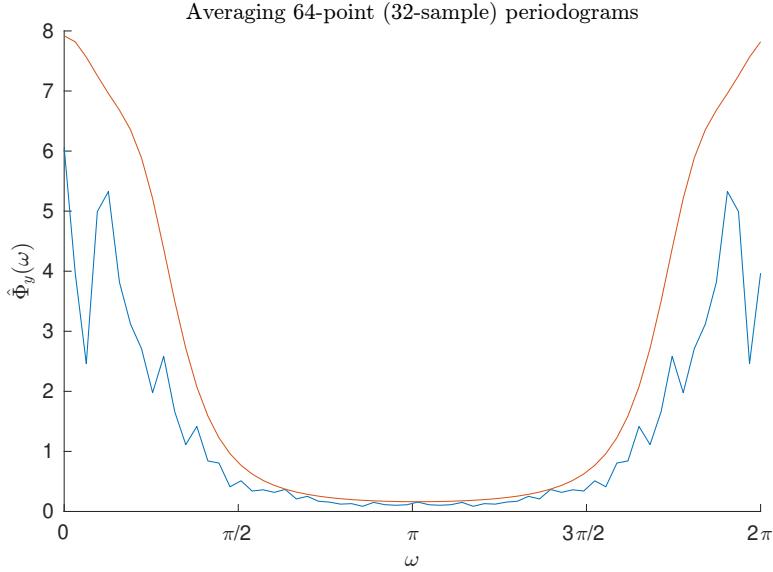


Figure 6: Averaging short periodograms produces a (relatively) smoother and more accurate result

3.3 Indirect Blackman-Tukey method (§B.4)

The Blackman-Tukey method is the first method discussed (of taking the FFT of the autocorrelation). Now, we take the autocorrelation of the entire signal ($N = 512$), then multiply this by a small rectangular centered at zero (all samples $|m| \leq 15$). Then we take the 64-point FFT. We see that this is somewhat noisy, with some “ripple” near $\omega = \pi$, but the tail behaviors are much closer to the actual function. This is most likely due to the fact that we used a biased estimator, so that the variance near the ends of the FFT is much lower. The result is plotted in Figure 7a, and the error is 1.1899.

We are also asked to do the same procedure, but windowing the autocorrelation with a 31-point triangular window (rather than a rectangular window) centered at $\tau = 0$. This generates the result shown in Figure 7b. This is much smoother and much closer on average to the true PSD than all of the other methods. Multiplying by a triangular window is similar to convolution by its Fourier transform, which has a smoothing effect; the tradeoff for losing variance is that we lose frequency resolution of the PSD, but the latter is not a concern in this case.

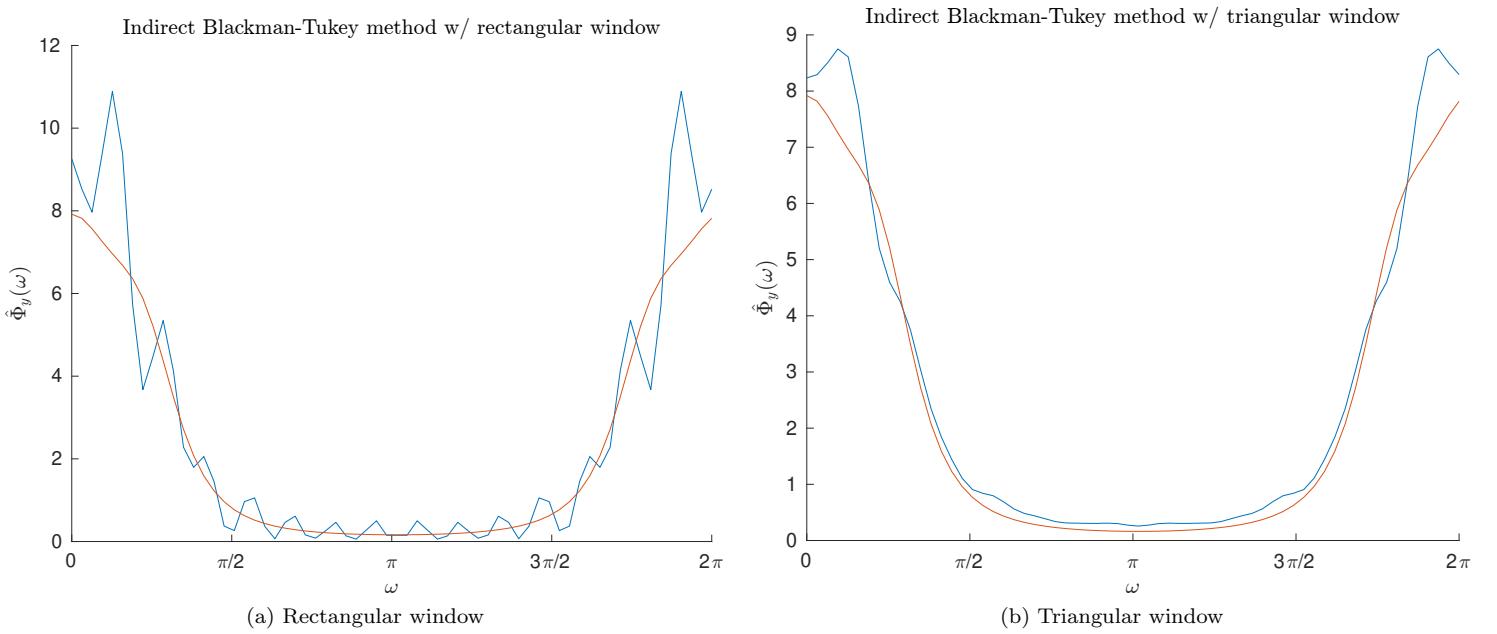


Figure 7: PSD estimate using Blackman-Tukey method with two different symmetric windows of length 31 (centered at $\tau = 0$)

3.4 Methods summary and B-T windowing (§B.5)

The summary of the methods is shown in Table 1.

Method	Error
64-point periodogram using first 32 samples (§B.1)	7.5039
1024-point periodogram using all 512 samples (§B.2)	7.5488
Averaged 64-point periodograms (32 samples each) (§B.3)	3.1403
Blackman-Tukey w/ length-31 rectangular window (§B.4)	1.1899
Blackman-Tukey w/ length-31 triangular window (§B.4)	0.2768

Table 1: Summary of methods and errors

The Blackman-Tukey method performed better than the periodogram method here – this is probably due to the fact that it is smoother than the periodograms. The Blackman-Tukey method using the triangular window performed by far the best – even though we lost frequency resolution, the expected shape was smooth so this wasn't a problem here.

The periodogram method that performed best was the averaging of small periodograms. The 64-point periodogram was not very accurate, and the 1024-point periodogram was very noisy; combining the better parts of both (less noise and smaller bias) in the average made it perform the best.

The shape of the averaged periodogram was pretty similar to that of the Blackman-Tukey with the rectangular window, but it (as well as the other periodogram estimates) all seem systematically lower than the real PSD. I'm not sure what the reasoning behind this is, but multiplying by a factor of roughly 1.7 seems to lower all of the error estimates; for example, multiplying the averaged periodogram by 1.7 (Figure 8) lowered the error to 0.9070, which is lower than for Blackman-Tukey with the rectangular window. This is for future analysis.

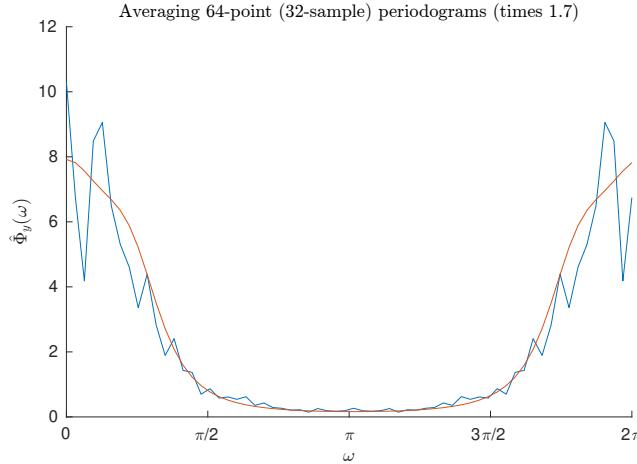


Figure 8: Multiplying averaged periodograms by a scalar factor lowers the error

ECE310 – Pset 3

Jonathan Lam

October 8, 2020

Problems: 5.11, 5.12, 5.28 (use MATLAB for part c), 5.34, 5.45

- 5.11** The system function of an LTI system has the pole-zero plot shown in Figure P5.11. Specify whether each of the following statements is true, is false, or cannot be determined from the information given.

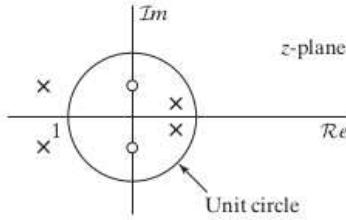


Figure 5.11

- a) The system is stable.

Cannot be determined from the information given. (3 possible ROCs, only one including unit circle is stable.)

- b) The system is causal.

Cannot be determined from the information given. (3 possible ROCs, only outermost one is causal.)

- c) If the system is causal, then it must be stable.

False. See reasoning above.

- d) If the system is stable, then it must have a two-sided impulse response.

True. The only stable ROC is bounded on both inside and outside by poles, which means that it has a two-sided impulse response.

- 5.12** A discrete-time causal LTI system has the system function

$$H(z) = \frac{(1 + 0.2z^{-1})(1 - 9z^{-2})}{1 + 0.81z^{-2}}$$

- a) Is the system stable?

Since this system is causal, and the only poles are at $\pm 0.9j$, the ROC is $|z| > 0.9$ includes the unit circle and is thus stable.

- b) Determine expressions for a minimum-phase system $H_1(z)$ and an all-pass system $H_{ap}(z)$ such that

$$H(z) = H_1(z)H_{ap}(z)$$

The only poles or zeros outside of the unit circle are the zeros at $z = \pm 3$, so we need to “flip” these into the unit circle to generate the minimum-phase component. I.e., we have to decompose the $(1 - 9z^{-2})$ term:

$$\begin{aligned} 1 - 9z^{-2} &= (1 + 3z^{-1})(1 - 3z^{-1}) \\ &= \left[\left(\frac{1 + 3z^{-1}}{3 + z^{-1}} \right) \left(3 \left(1 + \frac{1}{3}z^{-1} \right) \right) \right] \left[\left(\frac{1 - 3z^{-1}}{-3 + z^{-1}} \right) \left(-3 \left(1 - \frac{1}{3}z^{-1} \right) \right) \right] \\ &= \left[\frac{1 + 3z^{-1}}{3 + z^{-1}} \frac{1 - 3z^{-1}}{-3 + z^{-1}} \right] \left[-9 \left(1 + \frac{1}{3}z^{-1} \right) \left(1 - \frac{1}{3}z^{-1} \right) \right] \\ &= \frac{1 - 9z^{-2}}{-9 + z^{-2}} \left(-9 \left(1 - \frac{1}{9}z^{-2} \right) \right) \\ H(z) &= \frac{1 + 0.2z^{-1}}{1 + 0.81z^{-2}} \left(-9 \left(1 - \frac{1}{9}z^{-2} \right) \right) \left(\frac{1 - 9z^{-2}}{-9 + z^{-2}} \right) \\ &= \left[-9 \frac{(1 + 0.2z^{-1})(1 - \frac{1}{9}z^{-2})}{1 + 0.81z^{-2}} \right] \left[\frac{1 - 9z^{-2}}{-9 + z^{-2}} \right] \\ &= H_1(z)H_{ap}(z) \end{aligned}$$

5.28 A causal LTI system has the system function

$$H(z) = \frac{(1 - e^{j\pi/3}z^{-1})(1 - e^{-j\pi/3}z^{-1})(1 + 1.1765z^{-1})}{(1 - 0.9e^{j\pi/3}z^{-1})(1 - 0.9e^{-j\pi/3}z^{-1})(1 + 0.85z^{-1})}$$

- a) Write the difference equation that is satisfied by the input $x[n]$ and output $y[n]$ of this system.

$$\begin{aligned} (1 - e^{j\pi/3}z^{-1})(1 - e^{-j\pi/3}z^{-1})(1 + 1.1765z^{-1}) &= 1 + 0.1765z^{-1} - 0.1765z^{-2} + 1.1765z^{-3} \\ (1 - 0.9e^{j\pi/3}z^{-1})(1 - 0.9e^{-j\pi/3}z^{-1})(1 + 0.85z^{-1}) &= 1 - 0.05z^{-1} + 0.045z^{-2} + 0.6885z^{-3} \\ y[n] - 0.05y[n-1] + 0.045y[n-2] + 0.6885y[n-3] &= x[n] + 0.1765x[n-1] - 0.1765x[n-2] + 1.1765x[n-3] \end{aligned}$$

- b) Plot the pole-zero diagram and indicate the ROC for the system function.

See Figure 1; Since the system is causal, the ROC is the outermost region, i.e., $|z| > 0.9$.

- c) Make a carefully labeled sketch of $|H(e^{j\omega})|$. Use the pole-zero locations to explain why the frequency response looks as it does.

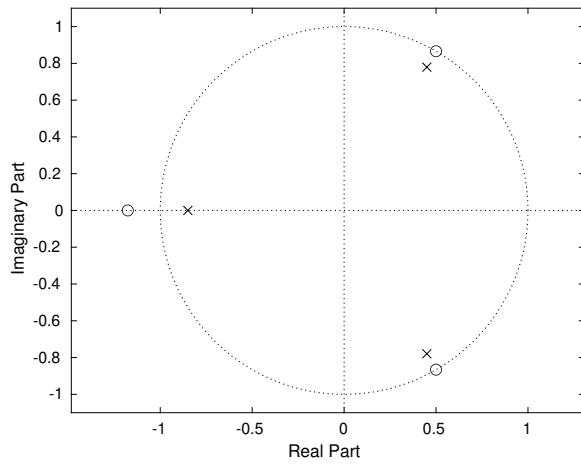


Figure 1: Pole-zero plot of $H(z)$.

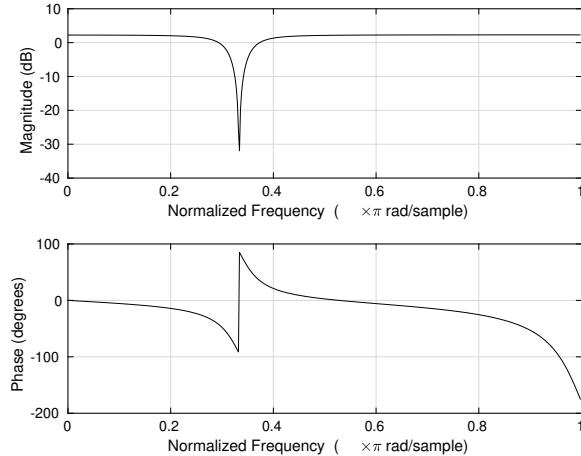


Figure 2: Frequency response of $H(z)$.

See Figure 2. (Using MATLAB's freqz generates both magnitude and phase response, and only for positive normalized frequency (it should be symmetric for negative frequencies); the top chart is the answer to this question.)

- d) State whether the following are true or false about the system:
- The system is stable.*
True. The ROC includes the unit circle (all poles lie inside the unit circle).
 - The impulse response approaches a nonzero constant for large n .*
False. Since the system is stable, the impulse response must approach zero when n is large.
 - Because the system function has a pole at angle $\pi/3$, the magnitude of the frequency response has a peak at approximately $\omega = \pi/3$.*

False. The system also has a zero at $\omega = \pi/3$ (and actually on the unit circle), whereas the pole is slightly off the unit circle), so the magnitude of the frequency response is actually zero at $\omega = \pi/3$.

- iv) *The system is a minimum-phase system.*

False. There is a zero at $z = -1.1765$ (outside of the unit circle).

- v) *The system has a causal and stable inverse.*

False. Since it is not minimum phase, its inverse will be either non-causal or unstable.

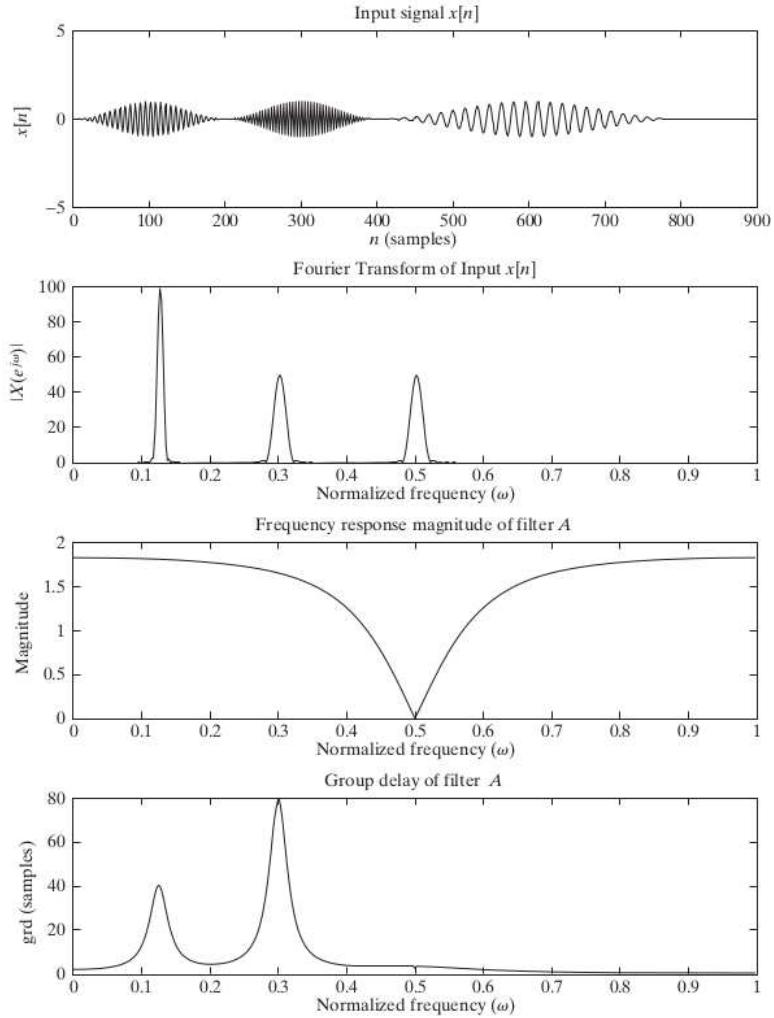


Figure 5.34-1: The input signal and the filter frequency response.

5.34 A discrete-time LTI system with input $x[n]$ and output $y[n]$ has the frequency response magnitude and group delay functions shown in Figure P5.34-1. The signal $x[n]$, also shown in

Figure P5.34-1, is the sum of three narrowband pulses. In particular, Figure P5.34-1 contains the following plots:

- $x[n]$
- $|X(e^{j\omega})|$, the Fourier transform magnitude of a particular input $x[n]$
- Frequency response magnitude plot for the system
- Group delay plot for the system

In Figure P5.34-2 you are given four possible output signals, $y_i[n] i = 1, 2, \dots, 4$. Determine which one of the possible output signals is the output of the system when the input is $x[n]$. Provide a justification for your choice.

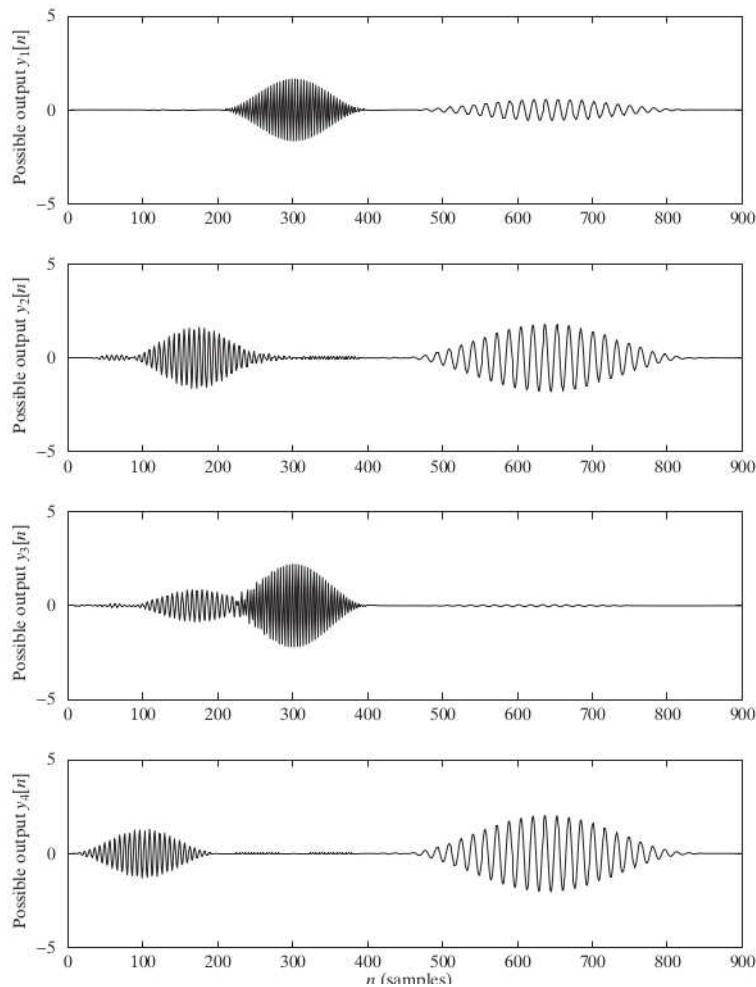


Figure 5.34-2: Possible output signals.

The input signal consists of three pulses with different frequencies. The first pulse is at a medium frequency (0.3 on a normalized scale), the second is at a higher frequency (0.5), and the last is at the lowest frequency (≈ 0.12).

The filter is a bandstop filter with the stopband frequency at 0.5, so the high frequency pulse should not be seen in the output. Both of the other pulses should not be significantly attenuated; the 0.3 frequency pulse will be scaled by less than the 0.1 frequency pulse.

The filter also has a high group delay (80 samples) for the 0.3 frequency pulse, and a lower group delay for the 0.1 frequency pulse (40 samples), which means that the distance between the two pulses will be smaller in the output.

The one that matches this best is $y_2[n]$, which has the first pulse shifted right by approximately 80 samples, the second pulse obliterated by the band-stop filter, and the last pulse shifted right by approximately 40 samples and slightly scaled up more than the first pulse.

5.45 The pole-zero plots in Figure P5.45 describe six different causal LTI systems.

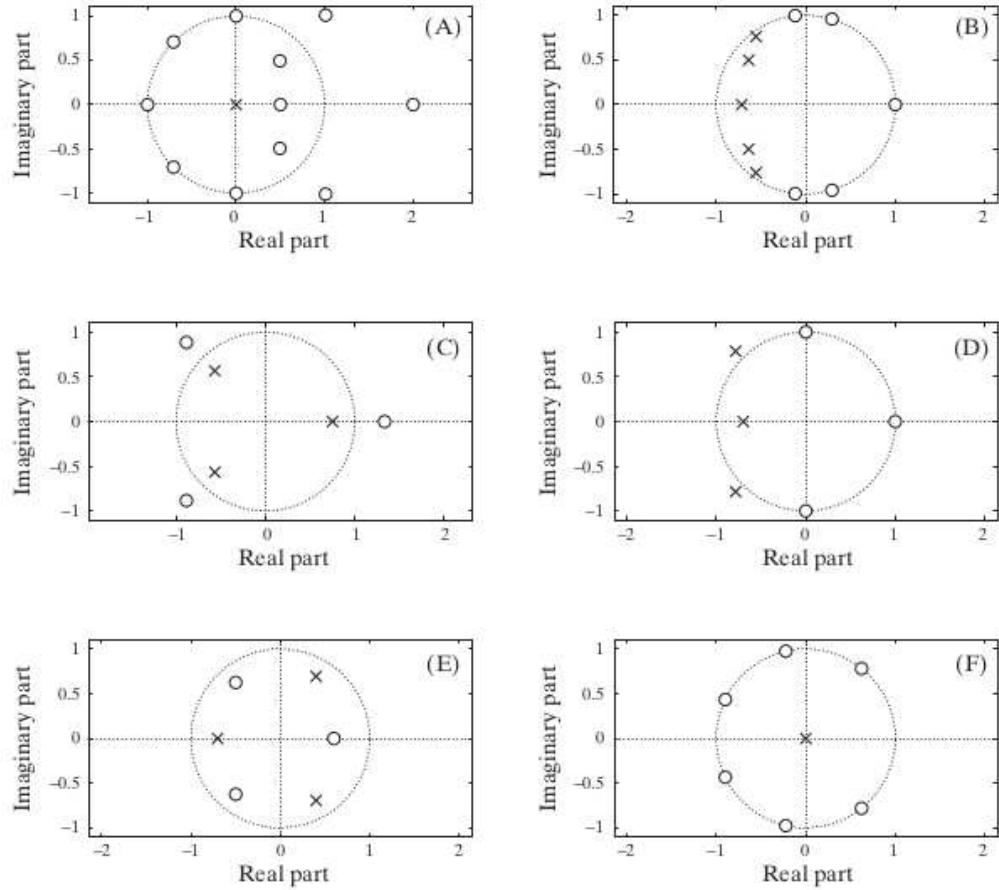


Figure 5.45

Answer the following questions about the systems having the above pole-zero plots. In each case, an acceptable answer could be none or all.

- a) *Which systems are IIR systems?*
(B), (C), (D), (E). IIR systems have poles at places other than the origin and infinity.
- b) *Which systems are FIR systems?*
(A), (F). An FIR system only has poles at the origin or infinity.
- c) *Which systems are stable systems?*
(A), (B), (C), (E), (F). A causal system is stable if all its poles lie inside the unit circle.
- d) *Which systems are minimum-phase systems?*
(E). A causal system is minimum phase if its poles and zeros all lie inside the unit circle.
- e) *Which systems are generalized linear-phase systems?*
(A), (F). Minimum phase systems have to be FIR and have to be symmetric, and have zeros only on the unit circle or in reciprocal pairs.
- f) *Which systems have $|H(e^{j\omega})| = \text{constant}$ for all ω ?*
(C). A system is all-pass if it has poles and zeros at conjugate reciprocal pairs. (We can't really tell if the points in C are truly at conjugate reciprocal distances away from the unit circle but it appears to be so.)
- g) *Which systems have corresponding stable and causal inverse systems?*
(E). This is equivalent to being minimum phase.
- h) *Which system has the shortest (least number of nonzero samples) impulse response?*
(F). System F only has seven zeros or poles, corresponding to seven nonzero samples in the impulse response. (System A has more, and the rest are IIR).
- i) *Which systems have lowpass frequency responses?*
(A), (F). A lowpass frequency response will occur when a system has a high magnitude response near zero frequency (i.e., no zeros too close to $z = 1$) and low magnitude response everywhere else (i.e., zeros along the rest of the circle are fine).
- j) *Which systems have minimum group delay?*
(E). Minimum-phase systems minimize phase distortion (phase response is roughly constant), and thus also minimize group delay.

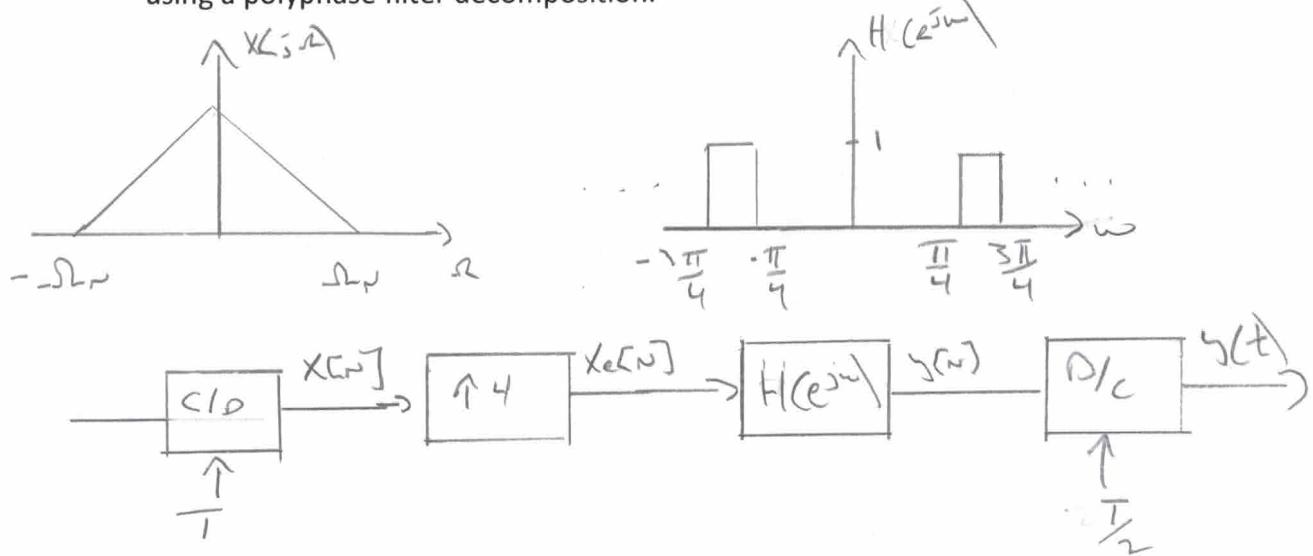
DSP Fall 2020

Quiz #1

Name:

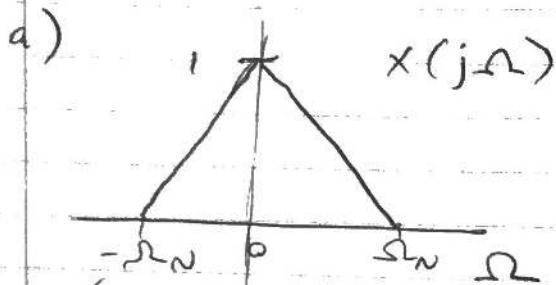
Consider the following system below, with input $X_c(j\Omega)$, and the frequency response of $H(e^{j\omega})$ are as shown. $X_c(t)$ is sampled at exactly Nyquist ($2\pi/T = 2\Omega_n$) and that the D/C converter is running at rate $\frac{1}{T}$.

- (8 points) Sketch $X(e^{j\omega})$, $X_e(e^{j\omega})$, $Y(e^{j\omega})$, and $Y_c(j\Omega)$. Label all frequencies and amplitudes.
- (1 point) Can you find an equivalent continuous time LTI system? If so, specify the system. If not, explain why not.
- (1 point) Can you use a polyphase implementation to reduce the amount of computation used to perform the upsampling and filtering operations? If no, explain why not. If yes, sketch a block diagram showing an equivalent system implemented using a polyphase filter decomposition.



DSP Quiz 1

Jonathan Lam
9/23/2020

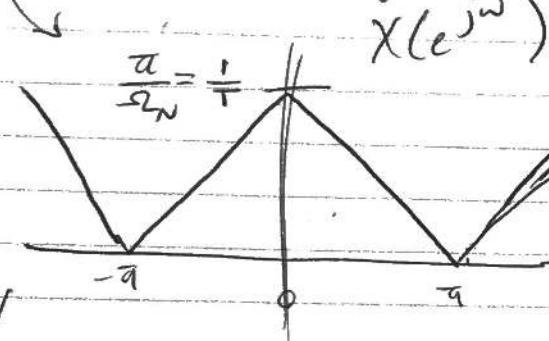


sampled @

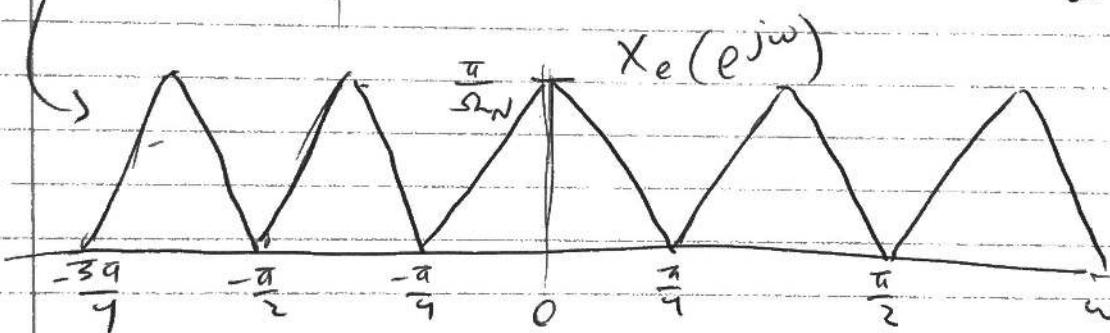
$$\Omega_s = \frac{2\pi}{T} = 2\Omega_N$$

$$\Rightarrow T = \frac{\pi}{\Omega_N}$$

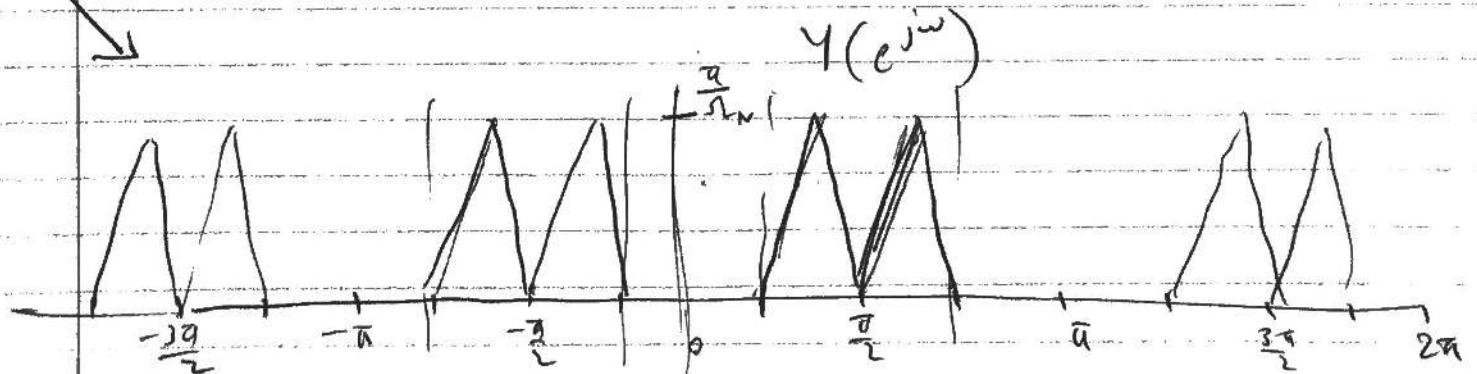
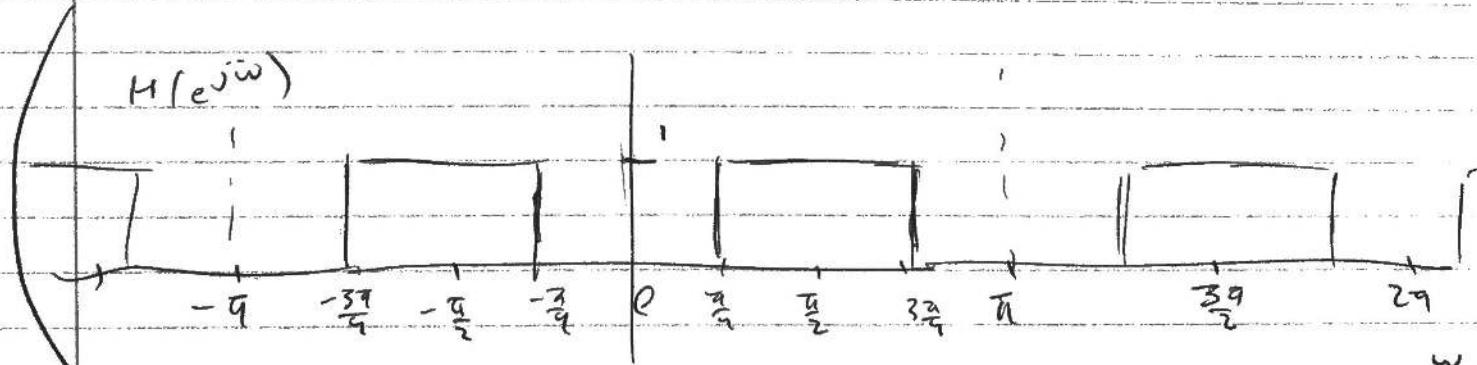
(don't need anti-aliasing filter, no aliasing is going to happen)



(x-axis scaled to digital frequency, magnitude scaled by $\frac{1}{T}$ (eq. 4.6))



(x-axis compressed by factor of 4, no scaling of magnitude (eq. 4.85))



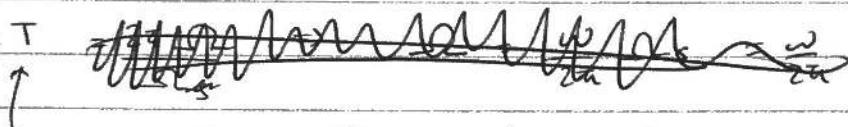
DSP Quiz 1

JONATHAN
LAM

a, cont'd.) D/C converter involves rescaling x-axis to analog frequency, rescaling magnitude by $\frac{1}{T}$, and passing through an anti-imaging LPF

D/C converter @ sampling rate $\frac{2}{T}$

$$\omega = \Omega T$$



here $T = \frac{T_0}{2} = \frac{\pi}{2\Omega_N}$ where T_0 was the C/D sampling period

$$\Omega = \omega \left(\frac{2\Omega_N}{\pi} \right)$$



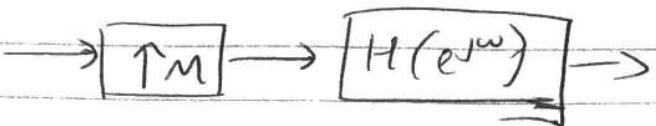
b) There is no LTI equivalent to this system, because of sampling at different frequencies (i.e., the C/D and D/C operate at different rates) and is thus not ~~not~~ TI.

Another way to tell that it is ~~not~~ LTI is that the support of $Y(j\Omega)$ includes frequencies that were not present in the support of $X(j\Omega)$, which is impossible if the system is linear.

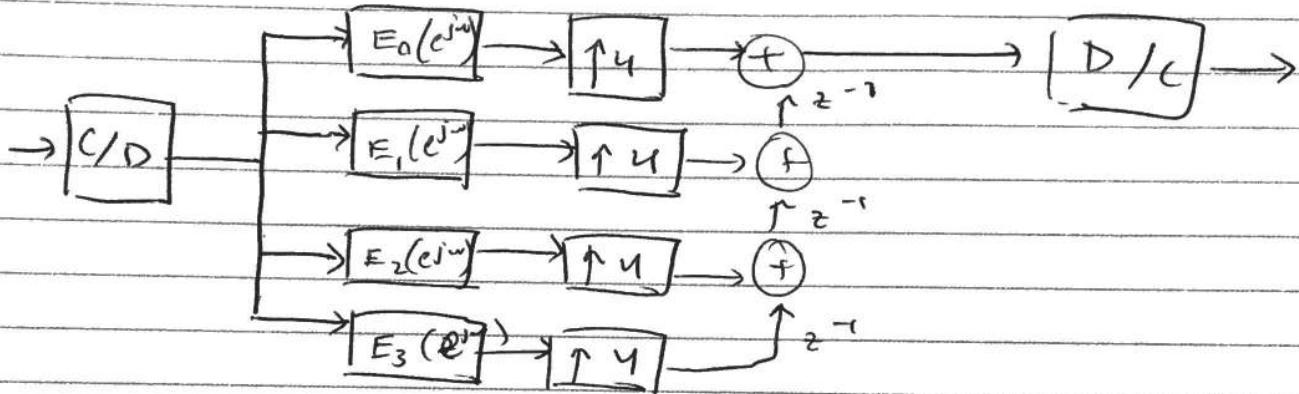


c). a

Yes, a polyphase implementation can be useful here.
In particular we have the pattern:



which means that the system H is operating at the higher (upsampled) frequency, and can be benefitted by ~~writing this on an~~ ~~polyphase~~ operating at the original frequency, i.e.,:



where $\{E_i\}$, $0 \leq i < 4$ are the polyphase components of $H(e^{j\omega})$.



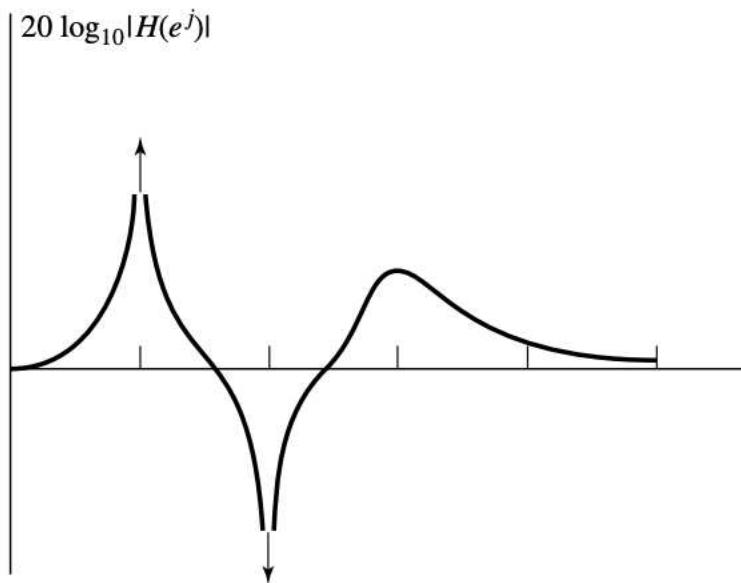
I apologize for my handwriting

DSP Quiz #2

10/7/20

Name:

Question 1: Consider a causal linear time-invariant system with system function $H(z)$ and real impulse response. $H(z)$ is evaluated for $z = e^{j\omega}$ and is shown in the figure below:



1 point each:

- Carefully sketch a pole-zero plot for $H(z)$ showing all information about the pole and zero locations that can be inferred from the figure.
- What can be said about the length of the impulse response? Justify your answer
- Is this a linear phase system? How can you tell?
- Is this system stable? How can you tell?
- Is this an all-pass system? How can you tell?
- Is this a minimum-phase system? How can you tell?

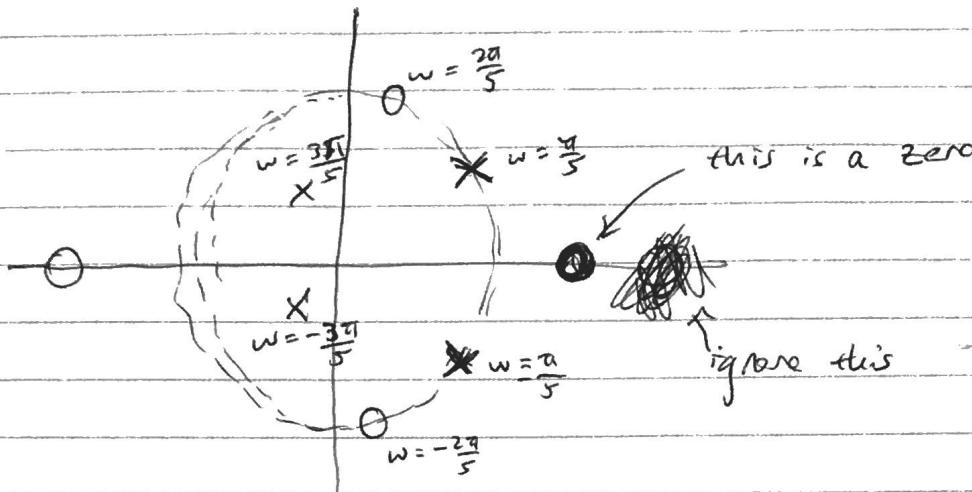
Question 2:

Consider the following system function:

$$H(z) = \frac{(1 - .5z^{-1})(1 + 4z^{-2})}{1 - .64z^{-2}}$$

- (1 point) Sketch the pole-zero plot for $H(z)$
- (1 point) Write the difference equation that relates the input and the output of this system
- (2 points) Find expressions for a minimum-phase system such that $H_1(z)$ and an all-pass system $H_{ap}(z)$ such that $H(z) = H_1(z)H_{ap}(z)$

1) a)



~~█~~ We can tell from this magnitude response plot is that there is a pole @ $w = \pm \frac{\pi}{5}$, & zero @ $w = \pm \frac{2\pi}{5}$. I'm assuming from the graph that these points on the magnitude response graphs are asymptotes, i.e., the pole and the zero lie exactly on the unit circle.

Since the magnitude response at $w = 0$ is $\approx 0 \text{ dB}$, this means there must be some other zero on the plot to balance ~~out~~ the poles near 0. There might also be some poles $\approx w = \pm \frac{3\pi}{5}$ to cause the magnitude response to rise again after reaching the zero, and again there might be a zero near $w = \pi$ to bring the magnitude response down to near 0 dB again (i.e., the right end of the magnitude response plot).

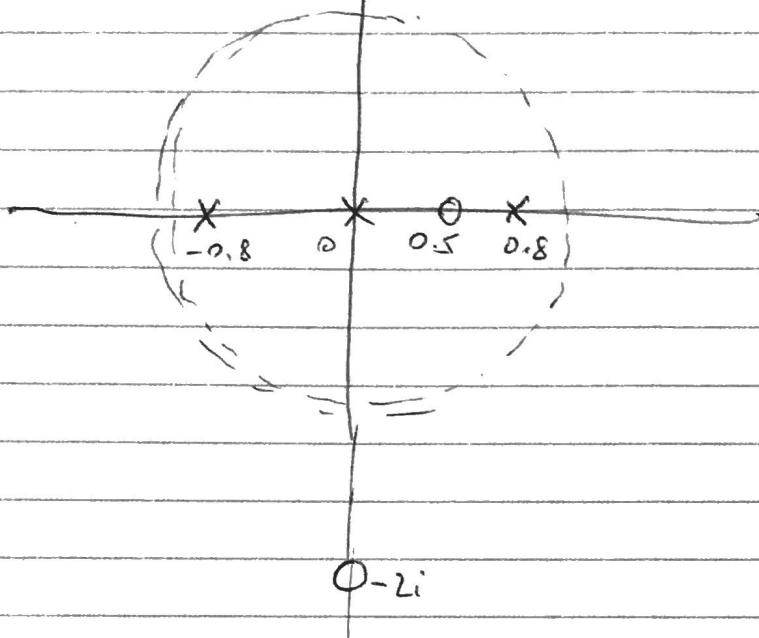
- b) IIR: has poles that are ~~not~~ at the origin.
- c) not ~~linear~~ phase because it is IIR.
- d) not stable, poles on the unit circle.
- e) not all-pass, magnitude response is not constant
- f) not minimum phase, not ~~all poles and zeroes are~~ in the unit circle.

$$2.) H(z) = \frac{(1 - 0.5z^{-1})(1 + 4z^{-2})}{1 - 0.64z^{-2}}$$

zeros @ $z = 0.5, \pm 2i$.

poles @ $z = \pm 0.8, \cancel{0}$
 $\oplus 2i$

a)



$$b) H(z) = \frac{Y(z)}{X(z)} = \frac{1 + 4z^{-2} - 0.5z^{-1} - 2z^{-3}}{1 - 0.64z^{-2}}$$

$$Y(z)(1 - 0.64z^{-2}) = X(z)(1 + 4z^{-2} - 0.5z^{-1} - 2z^{-3})$$

// inverse z-transform

$$y[n] - 0.64y[n-2] = x[n] - 0.5x[n-1] + 4x[n-2] - 2x[n-3]$$

c) Only the $\pm 2i$ zeros are outside the unit circle, have to "flip" them inside the circle to $\mp \frac{1}{2}i$. (conjugate \Leftrightarrow reciprocal).

$$(1 + 4z^{-2}) = (1 + 2iz^{-1})(1 - 2iz^{-1})$$

$$= \left(\frac{1 + 2iz^{-1}}{1 - 2iz^{-1}} \right) (1 - 2iz^{-1}) \left(\frac{1 - 2iz^{-1}}{1 + 2iz^{-1}} \right) (1 + 2iz^{-1})$$

$$= \left(\frac{1 + 2iz^{-1}}{1 - 2iz^{-1}} \right) \frac{1 - 2iz^{-1}}{1 + 2iz^{-1}} (1 - 2iz^{-1})(1 + 2iz^{-1})$$

$$= \left(\frac{1 + 4z^{-2}}{1 + 4z^2} \right) (1 + 4z^2) = \underbrace{\left(\frac{1 + 4z^{-2}}{1 + \frac{1}{4}z^{-2}} \right)}_{\text{all-pass}} \underbrace{(1 + \frac{1}{4}z^{-2})}_{\text{minimum phase-}}$$

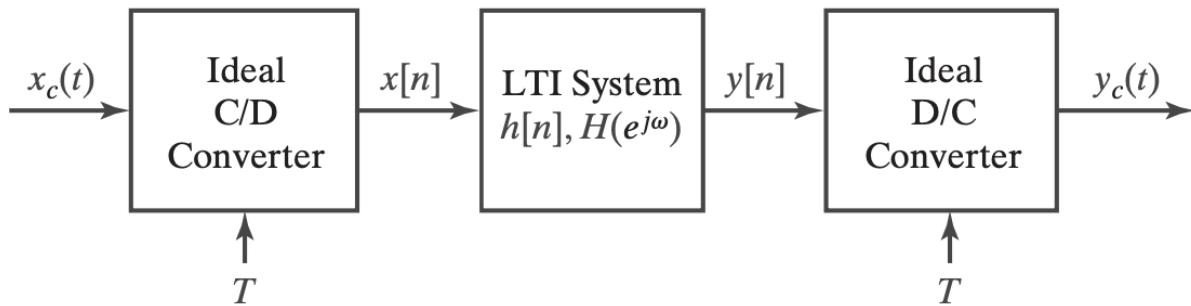
plugging back into the original equation:

10/8/20

$$H(z) = \underbrace{\left(\frac{1+4z^{-2}}{1+\frac{1}{4}z^{-2}} \right)}_{\text{all-pass-}} \underbrace{\left(\frac{(1-0.5z^{-1})(1+\frac{1}{4}z^{-2})}{1-0.64z^{-2}} \right)}_{\text{minimum phase}}$$

Question 1:

Consider the system below, where $H(e^{j\omega})$ is a bandpass filter, with passband ripple $\delta_1 = .001$ and stop band ripple $\delta_2 = .001$ and band edges $\omega_{s1} = .2\pi$, $\omega_{p1} = .4\pi$, $\omega_{p2} = .6\pi$, $\omega_{s2} = .8\pi$. The sampling rate for the ideal C/D and D/C is $1/T = 10,000$ samples per second. Furthermore, it is known that the filter $H(e^{j\omega})$ has a maximum group delay of 34 samples.



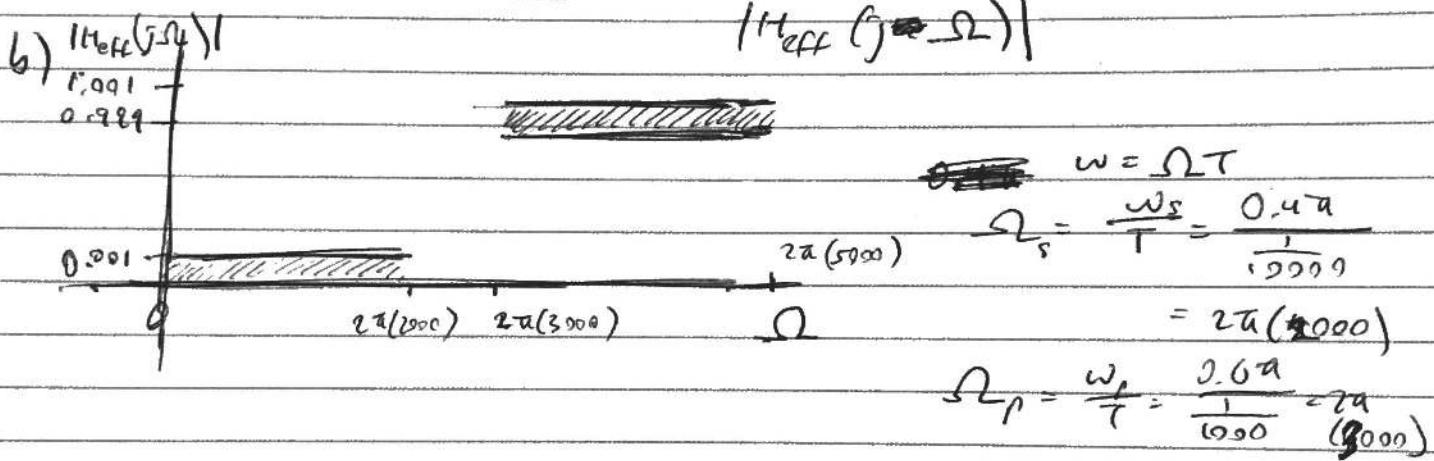
- a) (1 point) What property should the input signal have so that the overall system behaves as an LTI system with $Y_c(j\Omega) = H_{\text{eff}}(j\Omega) X_c(j\Omega)$?
- b) (1 point) For the conditions found in part a), sketch $|H_{\text{eff}}(j\Omega)|$, include as much detail as possible.
- c) (2 points) Based on the information given, could the filter $H(e^{j\omega})$ have been designed using the impulse invariance method? If so, specify the analog prototype filter $H_{ap}(j\Omega)$. If not, explain why not.
- d) (2 points) Based on the information given, could the filter $H(e^{j\omega})$ have been designed using the bilinear transformation method? If so, specify the analog prototype filter $H_{ap}(j\Omega)$. If not, explain why not.
- e) (2 points) Based on the information given, could the filter $H(e^{j\omega})$ have been designed using the Kaiser window method? If so, specify the ideal impulse response used in the design. If not, explain why not.
- f) (2 points) Based on the information given, could the filter $H(e^{j\omega})$ have been designed using the Park-McLellan algorithm? If so, specify possible values for the group delay. If not, explain why not.

$H(e^{j\omega})$ is a highpass filter, $d_p = 0.001$, $d_s = 0.001$, $\omega_s = 0.4\pi$, $\omega_p = 1.6\pi$.

$$T_s = \frac{1}{10000} \text{ s} \Rightarrow \Omega_s = 2\pi(10000) \frac{\text{rad}}{\text{s}}$$

$H(e^{j\omega})$ has maximum group delay of 14 samples

a) - If there is any aliasing, then the system will not be LTI. Thus ~~the~~ the bandwidth of $X_e(t)$ should be \leq Nyquist bandwidth, i.e., should be less than $2\pi(5000) \frac{\text{rad}}{\text{s}}$.



c) Impulse invariance cannot be used for a HPF due to aliasing (i.e., higher frequencies will alias into lower frequencies). However, bilinear transform would work b/c there's no aliasing there.

d) bilinear transformation can be used because it doesn't ~~introduce~~ introduce aliasing.

analog prototype filter:

$$\Omega = \frac{2}{T_d} \tan\left(\frac{\omega}{2}\right)$$

$$\left\{ \begin{array}{l} 0.999 \leq |H(e^{j\omega})| \leq 1.001 \quad \text{when } 0 \leq \omega \leq \pi. \end{array} \right.$$

$$\left. \begin{array}{l} |H(e^{j\omega})| \leq 0.001 \quad \text{when } 0 < \omega \leq 0.4\pi \end{array} \right.$$

↓

$$\left\{ \begin{array}{l} 0.999 \leq \left| H_{ap}(j\Omega) \right| \leq 1.001 \quad \text{when } 2(10000)\tan\left(\frac{0.6\pi}{2}\right) \leq \Omega \\ \# \quad \left| H_{ap}(j\Omega) \right| \leq 0.001 \quad \text{when } 0 \leq \Omega \leq 2(10000)\tan\left(\frac{0.4\pi}{2}\right) \end{array} \right.$$

No

e) Yes, Kaiser windows can generally represent HPFs.

$\Delta\omega = 0.2\pi$ by subtracting one LPF window from another

$$A = -20 \log_{10} \delta = -20 \log_{10} (0.001) = 60$$

$$\beta = 0.1102 (60 - 8.7)$$

$$M = \frac{A - 8}{2.285 \Delta\omega} = \frac{60 - 8}{(2.285)(0.2\pi)} \approx 36.2 \rightarrow 37$$

$$w[n] = \frac{\sin\left(\pi\left(n - \frac{36}{2}\right)\right)}{\pi\left(n - \frac{36}{2}\right)} - \frac{\sin\left(\pi\left(n - \frac{36}{2}\right)\right)^{0.5\pi}}{\pi\left(n - \frac{36}{2}\right)}$$

The group delay of this Kaiser window is $\frac{M}{2} = 18 > 14$

So this cannot be the filter.

No
Yes, we can compute this like for the Kaiser window.
f). using equation 7.117:

$$M = \frac{-10 \log_{10} (\delta_1 \delta_2)}{2.324 \Delta w} - 13$$

$$= \frac{-10 \log_{10} ((0.001)(0.001))}{2.324 (0.2\pi)} - 13$$

$$\approx 32.2 \rightarrow 33$$

round
up

$$\text{group delay} = \frac{33}{2} \approx 17.$$

but group delay is too high again ($17 > 14$).

DSP Quiz #4

Discrete Fourier Transform

11/23/2020

Consider the following finite length sequence:

$$x[n] = 2\delta[n] + \delta[n - 1] + \delta[n - 3]$$

- a) Compute the 5-point DFT $X[k]$
- b) Compute the 5-point DFT $Y[k] = X[k]^2$
- c) Compute the inverse 5-point DFT of $Y[k]$ to find the sequence $y[n]$ for $n = 0, 1, 2, 3, 4$
- d) If N-point DFTs are used in the two-step procedure, how should we choose N so that $y[n] = x[n]*x[n]$ for $0 \leq n \leq N - 1$?
- e) Repeat steps a-c with the value you found in part D.

DSP Quiz 4

JONATHAN LAM

$$x[n] = 2f[n] + f[n-1] + f[n-3], \quad N=5$$

a) DFT is linear, and DFT of $f[n-d]$ is w_N^{dk} .

$$\Rightarrow X[k] = \underbrace{2w_5^0}_2 + w_5^k + w_5^{3k}$$

b) Compute 5pt DFT $y[k] = (X[k])^2$

$$Y[k] = (X[k])^2 = (2 + w_5^k + w_5^{3k})(2 + w_5^k + w_5^{3k})$$

$$= 4 + 2w_5^{2k} + 2w_5^{3k} + 2w_5^{4k} + w_5^{2k} + w_5^{4k} \\ + 2w_5^{3k} + w_5^{4k} + \underbrace{w_5^{6k}}_{= w_5^{2k}} \leftarrow \text{wrap around due to aliasing}$$

$$= 4 + 5w_5^{2k} + w_5^{4k} + 4w_5^{3k} + 2w_5^{4k}$$

c) $g[n] = 4f[n] + 5f[n-1] + f[n-2] + 4f[n-3] + 2f[n-4]$

d) To avoid aliasing, choose N s.t. N is the length of the convolution, i.e., $N = \cancel{5+5-1} = 7$

(Note: I did $4+4-1$ rather than $5+5-1$ because the support of the original signal is length 4, not 5).

$$e) \quad N = 7.$$

$$X[k] = \underbrace{2w_7^0}_{=2} + w_7^k + w_7^{5k}.$$

$$y[k] = 4 + 4w_7^k + w_7^{2k} + 4w_7^{3k} + 2w_7^{4k} + w_7^{6k}$$

(almost the same as previous, but the w_7^{6k} doesn't
"wrap around")

$$y[n] = 4\delta[n] + 4\delta[n-1] + \delta[n-2] + 4\delta[n-3] + 2\delta[n-4] + \delta[n-6]$$

$$= x[n] * x[n]$$

\uparrow (I checked the convolution of x with itself on MATLAB)

DSP Quiz 5

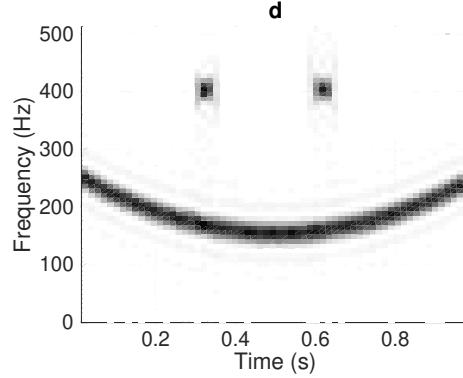
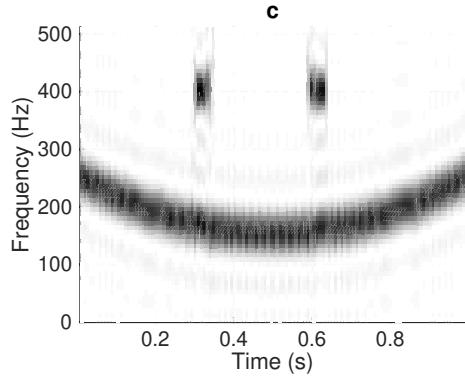
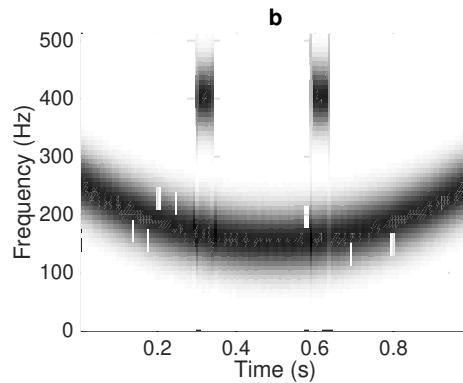
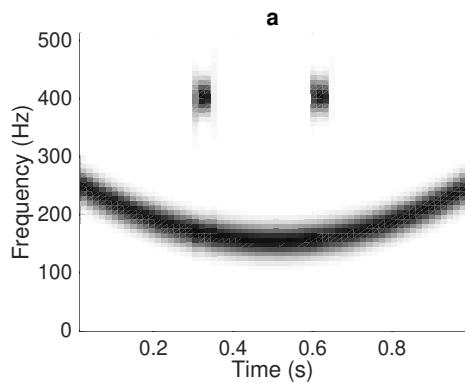
11/16/2020

Name:

Question 1

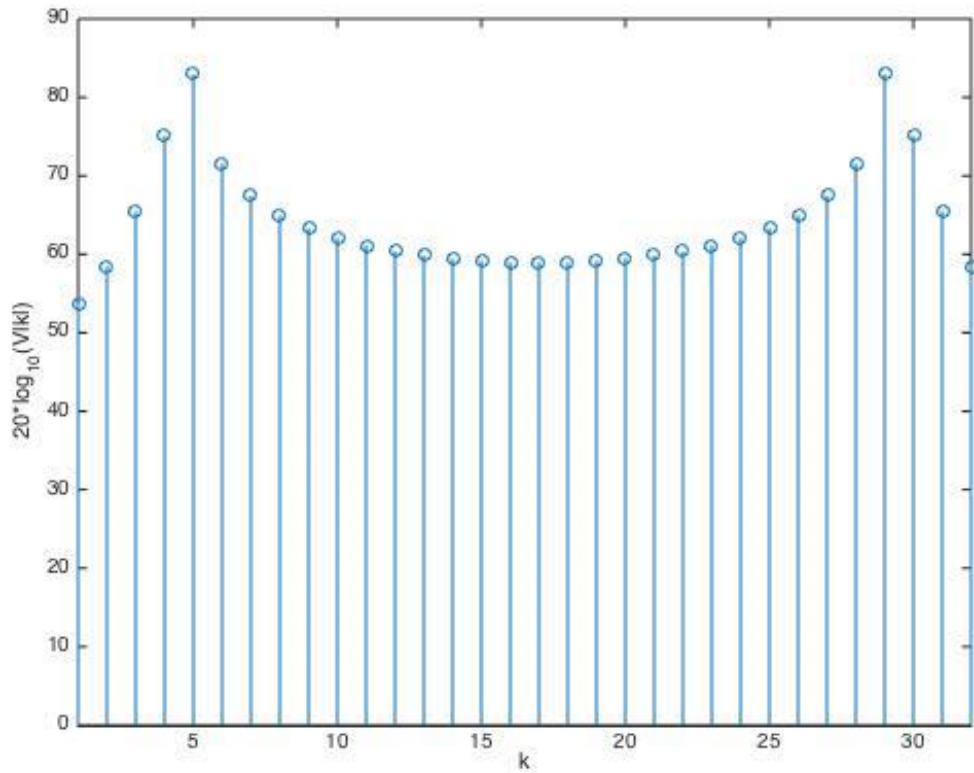
The following spectrograms were computed using either rectangular or Hamming windows, on the same signal. Answer the following questions, justify your answer completely.

- a) Which spectrograms were computed with a rectangular window?
- b) Which spectrograms have approximately the same frequency resolution?
- c) What is the approximate time window of spectrogram a? Mark the plot if it helps indicate your answer
- d) Write as detailed an equation as you can for the 'eye's, assuming that only pure sinusoidal tones were used to create them.



Question 2

Consider the following plot of the magnitude, in dB, of the DFT of a continuous time signal sampled at $T = 10^{-3}$. A 32 point DFT was taken using a rectangular window.



Listed below are 10 signals, one or more of which could have been the continuous time signal that produced the above plot. Indicate which signals could have been the input signal $x_c(t)$. Justify your answer completely

$$\begin{array}{ll}
 x_1(t) = 1000 \cos(230\pi t) & x_6(t) = 1000 \exp(j \cdot 250\pi t) \\
 x_2(t) = 1000 \cos(115\pi t) & x_7(t) = 10 \cos(250\pi t) \\
 x_3(t) = 10 \exp(j \cdot 460\pi t) & x_8(t) = 1000(\cos(218.75\pi t)) \\
 x_4(t) = 1000 \exp(j \cdot 230\pi t) & x_9(t) = 10 \exp(j \cdot 200\pi t) \\
 x_5(t) = 10 \exp(j \cdot 230\pi t) & x_{10}(t) = 1000 \exp(j \cdot 187.5\pi t)
 \end{array}$$

ECE310 – Quiz 5

Jonathan Lam

December 16, 2020

1. Smiley face spectrograms

- (a) Spectrograms (c) and (d) use the rectangular window (they have prominent sidelobe ripples). (a) and (b) use the Hamming window.
- (b) The spectrograms pairs ((a) and (d)) and ((b) and (c)) have similar frequency resolutions. Frequency resolution is the blurriness along the y-axis (frequency axis).
- (c) By a really rough estimate, blurring starts at $t = 0.29\text{s}$ and ends at $t = 0.31\text{s}$, which would indicate a time window length of 0.02s . To be safe, I think the length of the time window lies somewhere in the range $[0.01, 0.05]\text{s}$.
- (d) By another rough estimate, it looks like the eyes start at 0.3s and 0.6s and last for 0.05s each.

$$y(t) = \begin{cases} \cos(2\pi(400)t), & |t - 0.325| < 0.025 \text{ or } |t - 0.625| < 0.025 \\ 0, & \text{else} \end{cases}$$

2. DFT of a sinusoid

The peaks occur at $k = 4$ and $k = 28$ ($k = -4$ due to the wraparound). This means that the cosine lies somewhere between $k = 3$ and $k = 5$ on the DFT; looking at the way the peak is skewed it looks like it's slightly lower than $k = 4$. Use the formula:

$$\Omega_4 = 2\pi \frac{k}{NT} = 2\pi \frac{4}{(32)(0.001\text{s})} = 250\pi \text{ rad/s}$$

We also note that $\Omega_3 = 187.5\text{rad/s}$, and halfway between Ω_3 and Ω_4 is 218.75rad/s . Using the intuition above, we know that the true frequency lies in the non-inclusive range $(218.75, 250)$. ($\Omega = 218.75$ wouldn't be valid, because then the two frequencies at $k = 3$ and $k = 4$ would be equal height due to the symmetry of the DFT of the cosine, and similarly $\Omega = 250$ wouldn't be valid either). That eliminates all but x_1 , x_4 , and x_5 . Due to the asymmetry of the DFT of a complex signal (i.e., look at Euler's formula, the sin has a phase shift that makes it asymmetric), we can also eliminate x_4 and x_5 . Thus, x_1 is the only signal that could have this DFT.

ECE342 – Simulation Project

Jonathan Lam

December 9, 2020

Consider the following circuit:

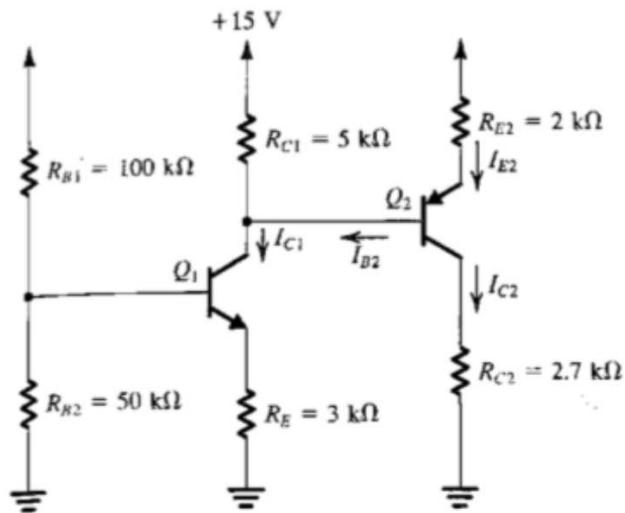


Figure 1: The circuit of consideration.

The transistor values are not given, so two common transistors were chosen: 2N2222 (NPN, Q_1) and 2N2907 (PNP, Q_2), which have corresponding LTSpice models. The values for these models are:

$$\begin{aligned}I_{S1} &= I_{S2} = 10^{-14}\text{ A} \\ \beta_1 &= 200 \\ \beta_2 &= 250\end{aligned}$$

1 Manual calculations

1.1 Solving for quiescent points

1.1.1 Solving for quiescent point of Q_1

Solve for V_{BE_1} , I_{C_1} numerically by iteration. Assume no early effect ($R_0 = \infty$), and assume that the base currents are negligible relative to the bias currents ($I_{B_1} \approx I_{B_2} \approx 0$). Alternate between the relations:

$$V_{BE_1} = V_T \log \frac{I_{C_1}}{I_{S_1}}$$

$$I_{C_1} = \frac{\beta_1}{\beta_1 + 1} \frac{V_{B_1} - V_{BE}}{R_{E_1}}$$

Using Python to perform the calculations, with initial values $V_{BE_{10}} = 0.7V$ and $I_{C_{10}} = 1mA$, we get the quiescent point for Q_1 (after 2 iterations):

$$I_{C_1} = 1.44mA$$

$$V_{BE_1} = 0.668V$$

With this, we can calculate the base current I_{B_1} to affirm our assumption that it is negligible relative to the bias current I_{bias_1} (the current through the biasing resistors):

$$I_{B_1} = \frac{I_{C_1}}{\beta_1} = \frac{1.44mA}{200} = 7.18 \times 10^{-6}A \ll 1.00 \times 10^{-4}A = \frac{15V}{100k\Omega + 50k\Omega} = I_{bias_1}$$

1.1.2 Solving for quiescent point of Q_2

We can use I_{C_1} to estimate the bias voltage V_{B_2} of the base of Q_2 :

$$V_{B_2} = V_{C_1} = V_{CC} - I_{C_1} R_{C_1} = 15V - (1.44mA)(5k\Omega) = 7.82V$$

Now we can calculate the quiescent point for Q_2 :

$$V_{EB_2} = V_T \log \frac{I_{C_2}}{I_{S_2}}$$

$$I_{C_2} = \frac{\beta_2}{\beta_2 + 1} \frac{V_{CC} - (V_{B_2} + V_{EB_2})}{R_{E_2}}$$

Python is used to calculate the results again, using the same initial values ($V_{EB_{20}} = 0.7V$, $I_{C_{20}} = 1mA$). This gives:

$$I_{C_2} = 3.23mA$$

$$V_{EB_2} = 0.689V$$

Again, we can check that the base current I_{B_2} is indeed small relative to the bias current I_{C_1} (the current going through the first transistor):

$$I_{B_2} = \frac{I_{C_2}}{\beta_2} = \frac{3.23mA}{250} = 1.29 \times 10^{-5}A \ll 1.44 \times 10^{-3}A = I_{C_1}$$

1.2 Small signal model

The small signal model is simply two common-emitter topologies in series.

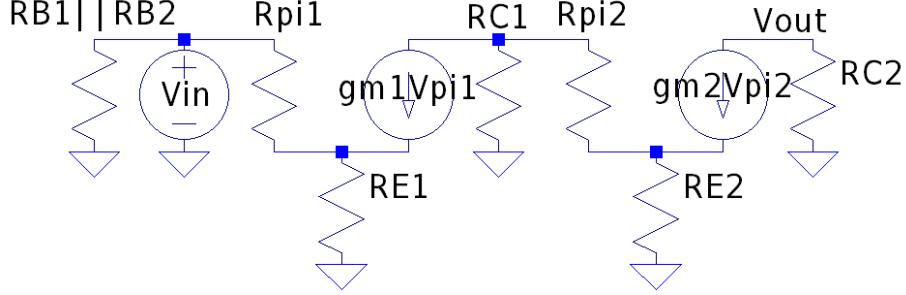


Figure 2: Small signal model

$$g_{m_1} = \frac{I_{C_1}}{V_T} = \frac{1.44\text{mA}}{26\text{mV}} = 5.53 \times 10^{-2}\Omega^{-1}$$

$$R_{\pi_1} = \frac{\beta_1}{g_{m_1}} = \frac{200}{5.53 \times 10^{-2}\Omega^{-1}} = 3.62 \times 10^3\Omega$$

$$g_{m_2} = \frac{I_{C_2}}{V_T} = \frac{3.23\text{mA}}{26\text{mV}} = 1.24 \times 10^{-1}\Omega^{-1}$$

$$R_{\pi_2} = \frac{\beta_2}{g_{m_2}} = \frac{250}{1.24 \times 10^{-1}\Omega^{-1}} = 2.01 \times 10^3\Omega$$

1.3 Voltage gain

Rederiving the voltage gain of a common-emitter with emitter degeneration (and assuming no early effect):

$$\begin{aligned} \frac{V_\pi}{R_\pi} + g_m V_\pi &= \frac{V_{in} - V_\pi}{R_E} \\ V_{out} &= -g_m V_\pi R_C \\ V_\pi \left(\frac{1}{R_\pi} + g_m + \frac{1}{R_E} \right) &= V_{in} \left(\frac{1}{R_E} \right) \\ \frac{V_{out}}{V_{in}} &= -\frac{g_m R_C}{R_E \left(\frac{1}{R_E} + \frac{1}{R_\pi} + g_m \right)} = -\frac{-g_m R_C}{1 + \frac{R_E}{R_\pi}(\beta + 1)} = -\frac{R_C}{\frac{1}{g_m} + R_E \frac{(\beta+1)}{\beta}} \\ A_V &\approx -\frac{R_C}{\frac{1}{g_m} + R_E} \end{aligned}$$

Intuitively, the input impedance for the common-emitter with emitter degeneration is $R_{in} = R_\pi + (\beta + 1)R_E$. In this example:

$$\begin{aligned}
A_V &= A_{V_1} A_{V_2} \\
&= \left[-\frac{R_{C_1} || R_{in_2}}{\frac{1}{g_{m_1}} + R_{E_1}} \right] \left[-\frac{R_{C_2}}{\frac{1}{g_{m_2}} + R_{E_2}} \right] \\
&= \frac{R_{C_1} || (R_{\pi_2} + (\beta_2 + 1)R_{E_2})}{\frac{1}{g_{m_1}} + R_{E_1}} \frac{R_{C_2}}{\frac{1}{g_{m_2}} + R_{E_2}} \\
&= \frac{5k\Omega || (2.01 \times 10^3 \Omega + (250 + 1)2k\Omega)}{\frac{1}{5.53 \times 10^{-2} \Omega^{-1}} + 3k\Omega} \frac{2.7k\Omega}{\frac{1}{1.24 \times 10^{-1} \Omega^{-1}} + 2k\Omega} \\
&= 2.21
\end{aligned}$$

2 LTSpice Simulation

2.1 Verifying the analytical results

See Figure 4 for the schematic used to check the quiescent points of Q_1 and Q_2 , and 3 for the measured values.

See Figure 6 for the schematic used to check the small-signal voltage gain of the circuit, and 5 for the result. To measure the zero-centered voltage gain, a coupling capacitor and a large pull-down resistor $R_L = 1M\Omega$ were used (i.e., not a heavy load). The maximum amplitude of V_{in} and V_{out} were compared to calculate A_V .

I did not explicitly try to estimate the R_π and g_m values experimentally in LTSpice; the proximity of the calculated A_V to the true value is a good indicator that the calculated values are correct.

Overall, the experimental values were fairly close to the calculated ones. This reassures us that the assumptions made in the calculations were relatively safe to make ($I_{B_1} \approx I_{B_2} \approx 0$, $R_0 = \infty$).

	Calculated	Experimental
V_{BE_1}	0.668V	0.663V
I_{C_1}	1.44mA	1.35mA
V_{EB_2}	0.689V	0.685V
I_{C_2}	3.23mA	3.03mA
A_V	2.21	2.17

Table 1: Calculated vs. experimental metrics

2.2 Adding a load

When the load impedance R_L is made very small, we see that the voltage gain is catastrophically attenuated. It is not hard to see that the relationship between

R_L (Ω)	A_V (w/o EF)	A_V (w/ EF)
1	0.000804	0.122
2	0.00161	0.231
3	0.00241	0.329
4	0.00321	0.418
5	0.00401	0.497
6	0.00483	0.570
7	0.00563	0.637
8	0.00642	0.698
9	0.00722	0.755
10	0.00803	0.807

Table 2: Voltage gain after adding load with and without emitter-follower stage

A_V and R_L is roughly linear, which is expected from the voltage gain equation. ($A_V \propto R'_{C_2}$, where $R'_{C_2} = R_{C_2} || R_L \approx R_L$ (when $R_L \ll R_{C_2}$) when a load is attached.) A rough calculation shows that these values are expected: when $R'_{C_2} \approx 10\Omega$, $A'_V = 8.03 \times 10^{-3} \approx 8.19 \times 10^{-3} = \frac{10\Omega}{2.7k\Omega}(2.21) = \frac{R'_{C_2}}{R_{C_2}} A_V$.

The results are shown in Table 2.

2.3 Mitigating the effect of the load

To mitigate the effect of the load, we can use an additional emitter-follower stage before the load. While emitter-followers don't have a high gain, they have a much lower output impedance, so the gain will be attenuated less. Emitter-followers have the following output impedance:

$$R_{out} = \left(\frac{R_S}{\beta + 1} + \frac{1}{g_m} \right) \parallel R_E$$

(derivation deferred to the textbook, where R_S is the resistance in series with the base). The first term greatly decreases the output impedance. A common-emitter with emitter degeneration has output impedance $R_{out} = R_C$, which is much higher.

See the schematic in Figure 7. The results are shown in Table 2. It is clear that the gain is still significantly degraded, but is over a factor of 100 greater than the gain without the EF stage.

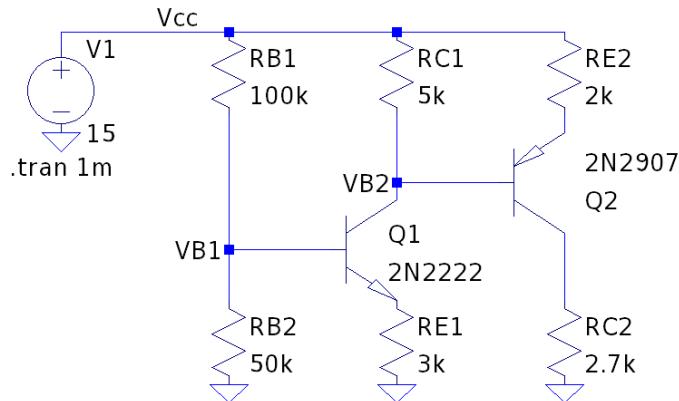


Figure 3: Biasing circuit in LTSpice, for checking the quiescent points of the transistors.

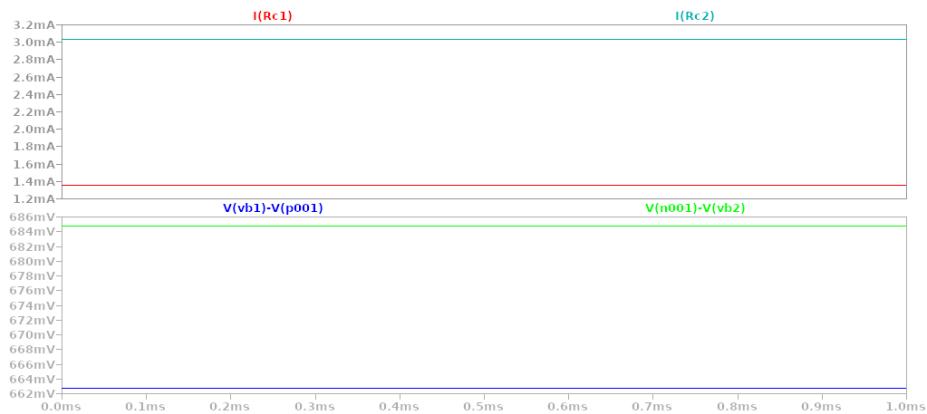


Figure 4: Quiescent points measured in LTSpice. Measured values: $I_{C_1} = 1.35\text{mA}$, $V_{BE_1} = 0.663\text{V}$, $I_{C_2} = 3.03\text{mA}$, $V_{EB_2} = 0.685\text{V}$.

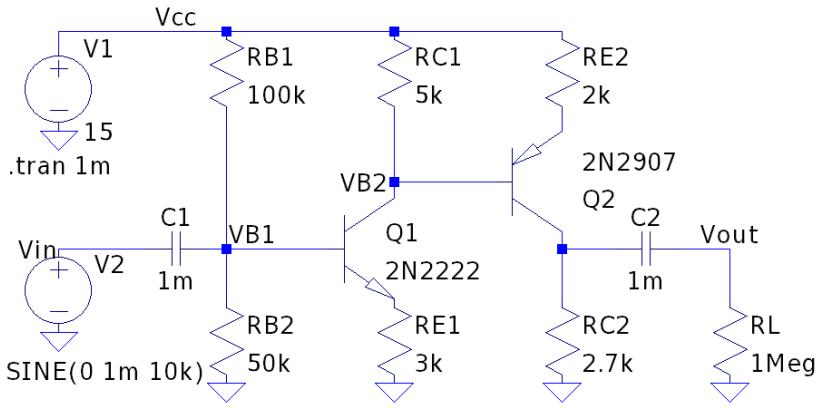


Figure 5: Circuit schematic in LTSpice. An input and output circuit with coupling capacitors are added to input and output a small, zero-DC-biased signal. Large coupling capacitors are used so that the small signals are not attenuated by much. The load resistor is large ($1M\Omega$) when testing A_V so as to not load the circuit heavily.

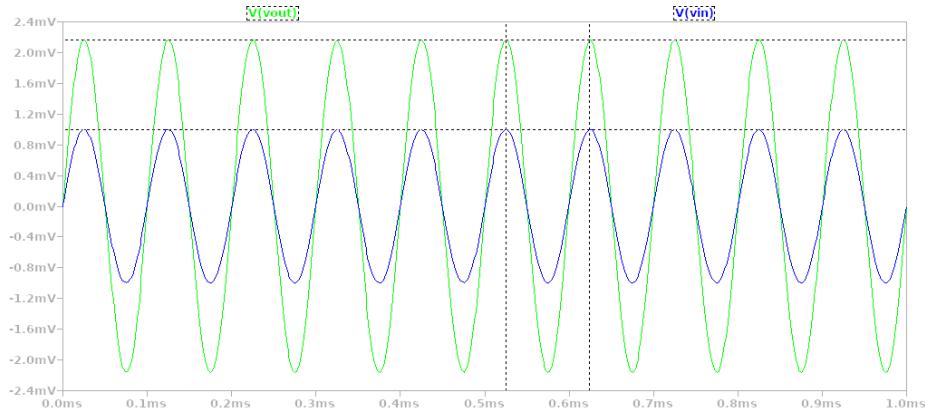


Figure 6: Two cursors placed at the maximum amplitudes of V_{in} and V_{out} when a small signal is applied. $V_{in} = 0.996\text{mV}$, $V_{out} = 2.16\text{mV}$; $A_V = 2.17$. (I also tried plotting V_{in}/V_{out} , but this causes numerical instability as $V_{out} \rightarrow 0$).

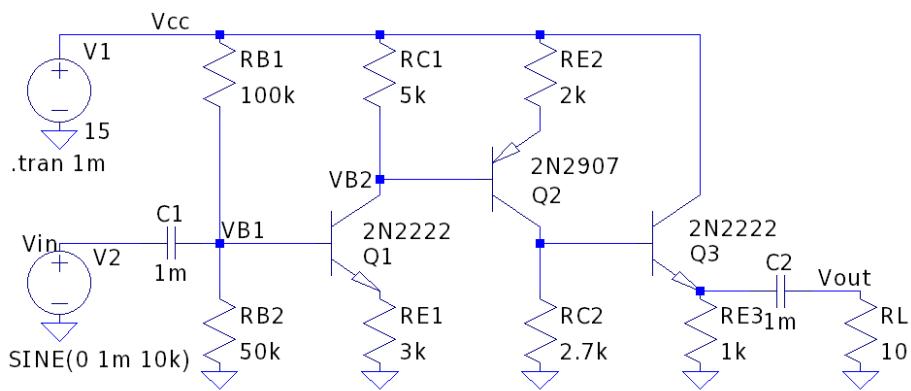
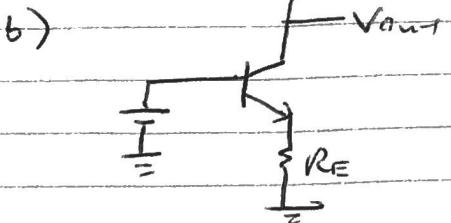
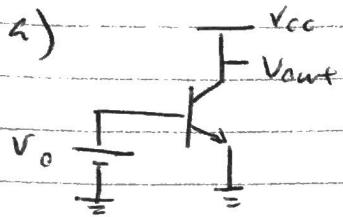


Figure 7: Schematic with emitter-follower stage to reduce output impedance, and thus be able to drive heavier loads (small R_L).



i) which one is a better current source?

In general a good current source should not be too sensitive to fluctuations in voltage, temperature, etc.

large signal model :

$$I_c = I_s \exp\left(\frac{V_{BE}}{V_T}\right) \left(1 + \frac{V_{CE}}{V_A}\right)$$

↑
non-ideal: changes in V_{CE}
cause I_c to change.
(early effect)

For (a), $V_{BE} = V_0$

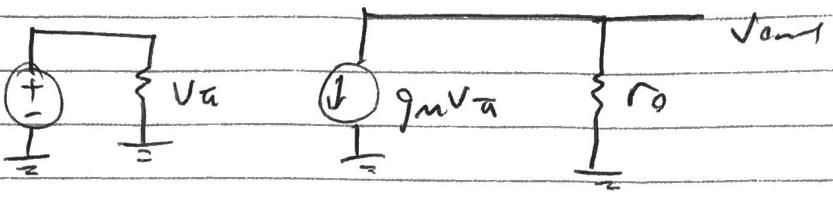
For (b), $V_{BE} = V_0 - I_E R_E$

Due to the early effect, when V_{CE} increases, I_c also increases. However, for (b), V_{BE} also ~~decreases~~ if $I_E \approx I_c$ increases, which helps to lower I_c and thus mitigates the early effect nonideality. Thus (b) acts as a better current source because it is more resistant to changes in V_{CE} .

(Here, the assumption is made that $V_A \neq \infty$. (i.e., has a nontrivial effect on the circuit). If $V_A = \infty$, then the early effect would be gone and the two circuits would behave similarly as a current source).

(ii) Find output impedances of the circuits.

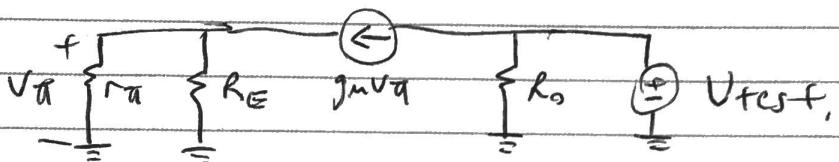
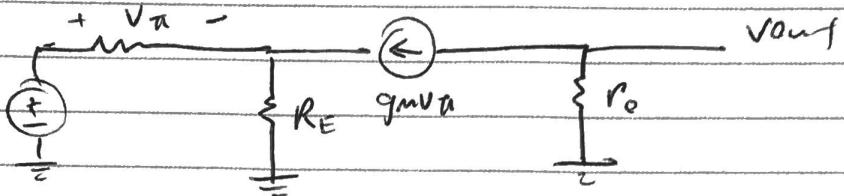
SSM (a):



$$\frac{V_A}{R_A} = 0 \quad \text{and} \quad \frac{g_m V_A}{R_o} = 0 \quad \text{so} \quad V_{test} = V_{out}$$

Clearly output impedance = R_o .

SSM (b):



$$\text{By KCL on left node: } \frac{V_A}{R_A} + \frac{V_A}{R_E} = g_m V_A$$

$$\text{assuming } V_A \neq 0: \frac{1}{R_A} \left(\frac{V_A}{R_A} + \frac{V_A}{R_E} \right) = \frac{1}{R_A} (g_m V_A)$$

$$\frac{1}{R_A} + \frac{1}{R_E} = g_m.$$

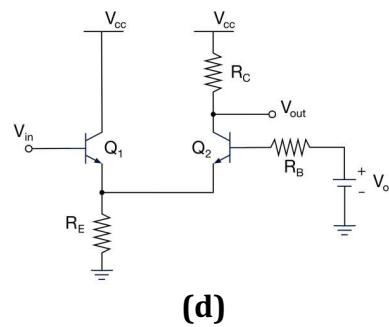
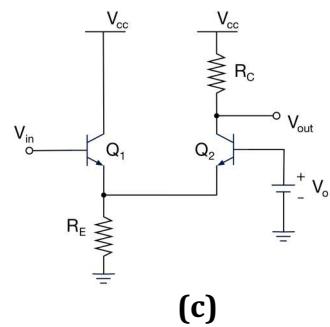
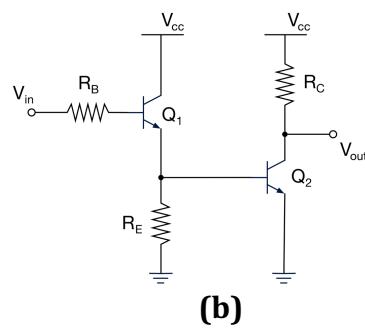
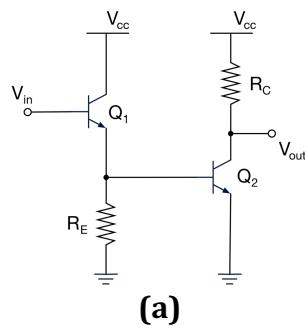
This equation makes no sense, so $V_A = 0$.

Thus $g_m V_A = 0$, so no current flows through the VCCS
so output impedance = R_o again.

The output impedances are the same, so it doesn't really support ~~the claim in part (i)~~ the claim in part (i). It just shows that both circuits are affected by the early effect (R_o is related to V_A). This is because the resulting change in V_{BE} (that helps to mitigate the early effect) is not a small-signal property.

Problem 1

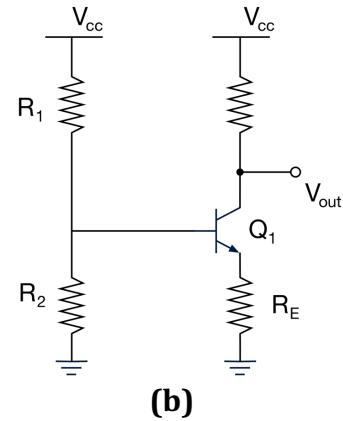
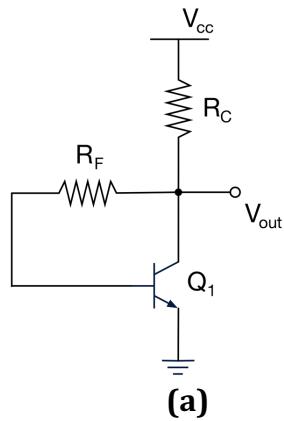
Find the gain for the following cascaded stage topologies and draw the small signal models. (Assume $V_A = \infty$)



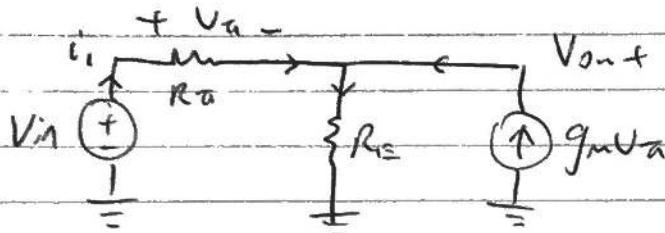
Problem 2

Consider the two C-E stages below, where $V_{cc} = 2.5$ V, $\beta = 100$, and $I_s = 5 \times 10^{-16}$ A. Perform the following:

- Bias the circuits such that $I_c = 1$ mA
- How much will V_{BE} change if V_{CC} is increased by 5%? Which circuit is less sensitive to variations in V_{CC} for the specific values that you chose for biasing the circuits.
- Explain the mechanism by which each circuit inhibits changes in V_{BE} due to variation in V_{CC} .



EF



Finding A_v for
common collector follower

1

$$i_1 + g_m V_a = \frac{V_{out}}{R_E}, \quad V_a = V_{in} - V_{out}, \quad i_1 = \frac{V_a}{R_a}$$

$$\frac{V_a}{R_a} + g_m V_a = \frac{V_{out}}{R_E}$$

$$\frac{V_{in} - V_{out}}{R_a} + g_m (V_{in} - V_{out}) = \frac{V_{out}}{R_E}$$

$$V_{in} \left(\frac{1}{R_a} \right) + V_{in} (g_m) = \frac{V_{out}}{R_E} + V_{out} \left(\frac{1}{R_a} + g_m \right) + V_{out} (g_m)$$

$$V_{in} \left(\frac{1}{R_a} + g_m \right) = V_{out} \left(\frac{1}{R_E} + \frac{1}{R_a} + g_m \right)$$

$$g_m = \frac{\beta}{R_a}$$

$$V_{in} \left(\frac{\beta + 1}{R_a} \right) = V_{out} \left(\frac{\beta + 1}{R_E} + \frac{1}{R_a} \right).$$

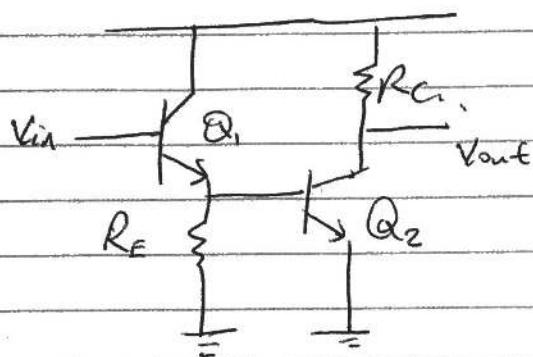
$$\frac{V_{out}}{V_{in}} = \frac{\frac{\beta + 1}{R_a}}{\frac{\beta + 1}{R_a} + \frac{1}{R_E}} = \frac{R_E}{R_E + \frac{R_a}{\beta + 1}}$$

$$\frac{R_a}{\beta + 1} \approx \frac{1}{g_m} = \frac{R_E}{R_E + \frac{1}{g_m}}$$

When there is an ~~base~~ resistor R_B at the base, it is in series with R_a , it would make it like:

$$A_v = \frac{R_E}{R_E + \frac{R_a + R_B}{\beta + 1}} \approx \frac{R_E}{R_E + \frac{1}{g_m} + \frac{R_B}{\beta + 1}}$$

1.) Find the gain:



First stage

$$A_{V1} \text{ of emitter follower stage} = \frac{R_E'}{R_E' + \frac{1}{g_m}}$$

$$\text{where } R_E' = R_E \parallel R_{in2}.$$

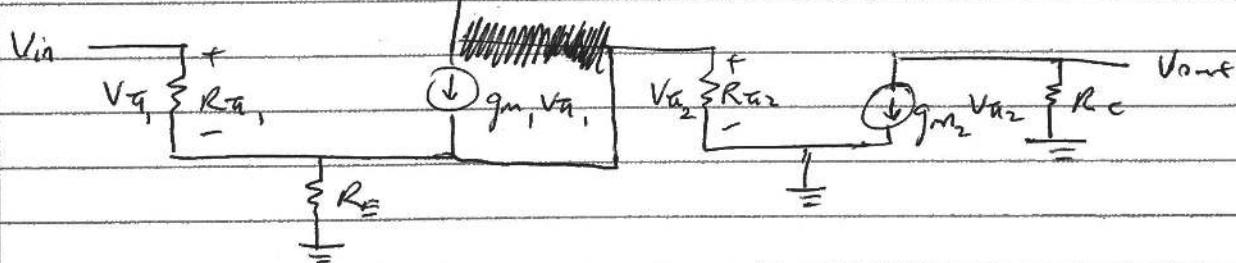
$$R_{in2} = R_{A2} \text{ (input impedance of a CE stage).}$$

Second stage

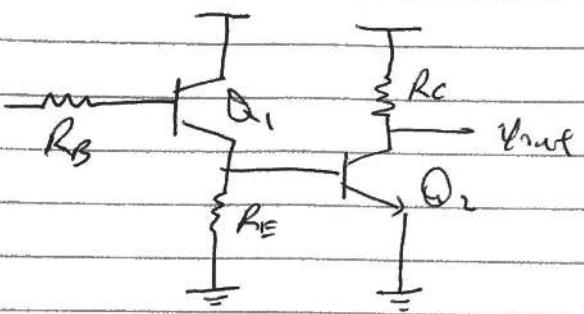
$$A_{V2} = -g_m R_{C2}$$

$$\text{total gain} = A_{V1} \cdot A_{V2} = \frac{R_E \parallel R_{A2}}{(R_E \parallel R_{A2}) + \frac{1}{g_m}} \cdot (-g_{m2} R_{C2}).$$

Small signal model:



b)

First stage

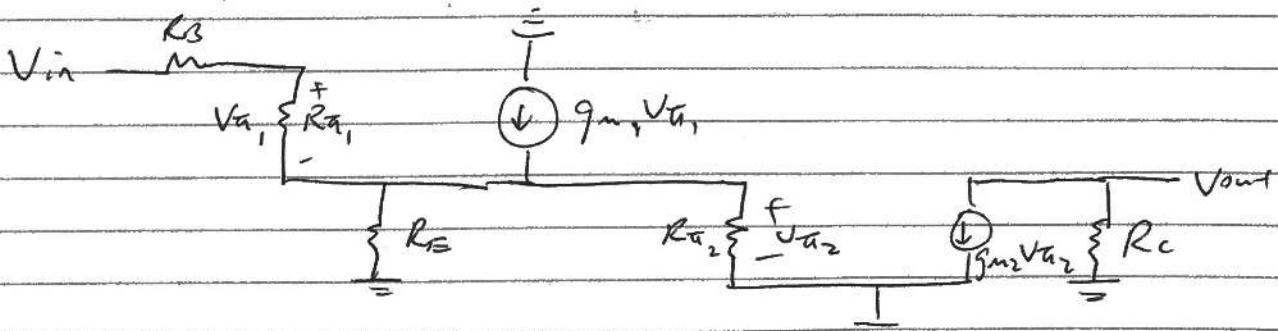
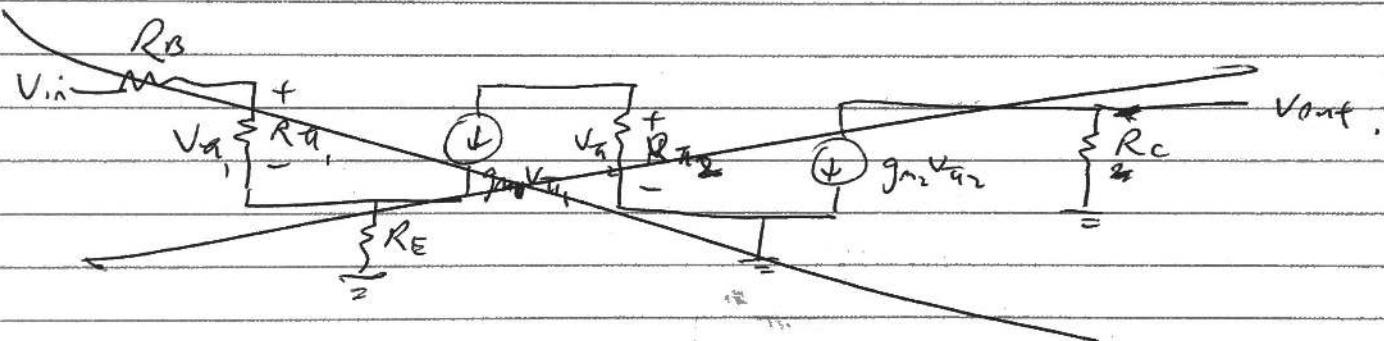
$$A_{V1} = \frac{R_C}{R_E' + \frac{1}{g_m1} + \frac{R_B}{\beta_1 + 1}}$$

where $R_E' = R_E \parallel R_{in2}$.

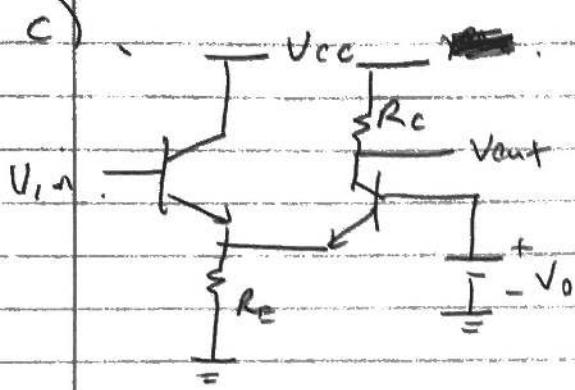
$$R_{in2} = R_{q2}$$

$$A_{V2} = -g_{m2}R_C$$

total gain: $A_v = A_{V1} \cdot A_{V2} = \left(\frac{R_E \parallel R_C}{(R_E \parallel R_C) + \frac{1}{g_{m1}} + \frac{R_B}{\beta_1 + 1}} \right) \left(-g_{m2}R_C \right)$

Small-Signal model:

(4)



First stage

$$A_{V1} = \frac{R_E}{R_E'}$$

$$R_E' = R_E \parallel R_{in_2}$$

where R_{in_2} is looking into the emitter,

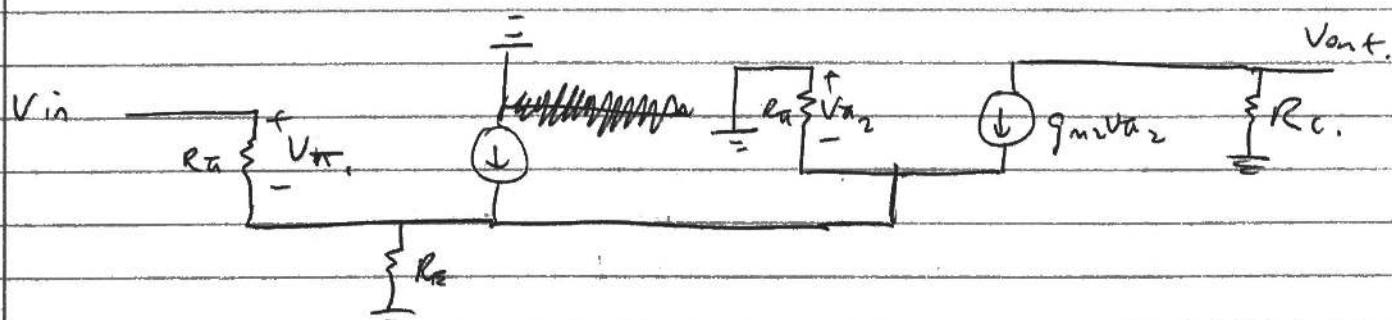
$$= \frac{R_{in_2}}{\beta_2 + 1} \approx \frac{1}{g_m 2}$$

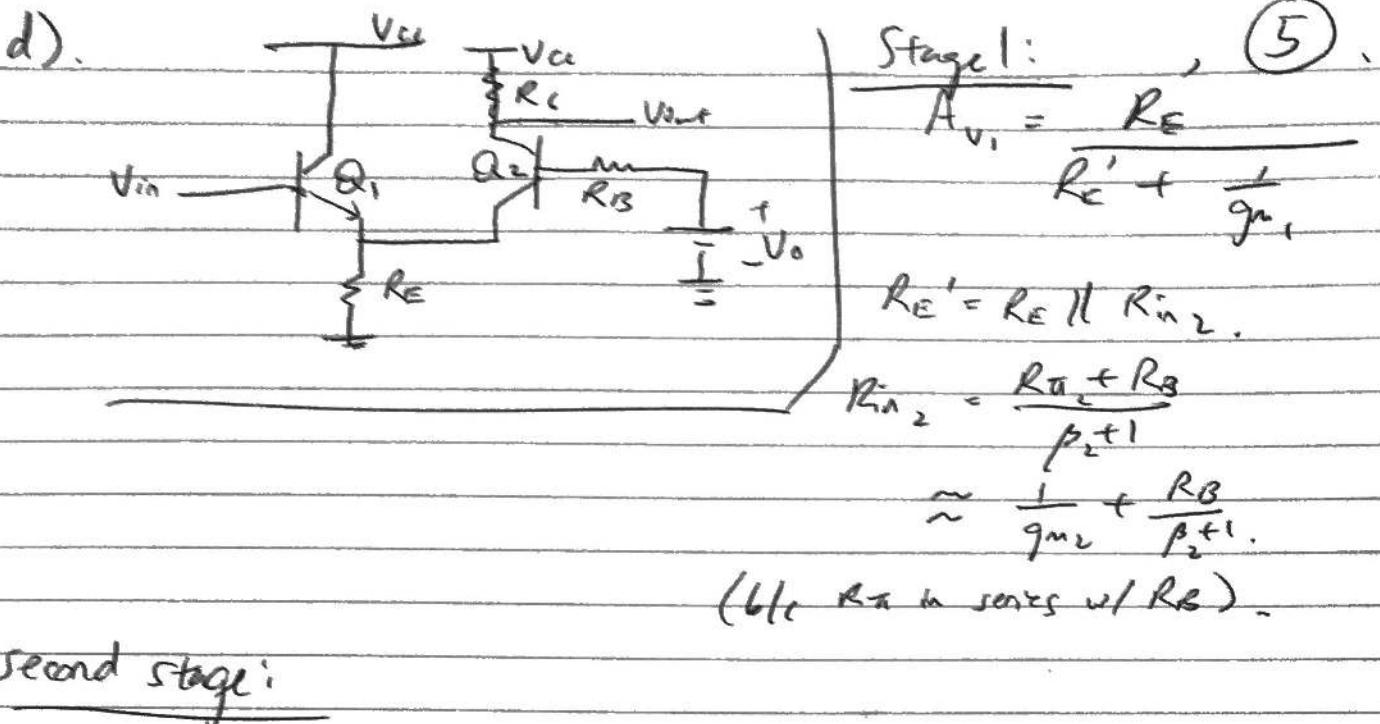
Second stage

$$A_{V2} = g_m 2 R_C \quad (\text{common base})$$

Total gain: $A_V = A_{V1} \cdot A_{V2} = \left(\frac{R_E \parallel \frac{1}{g_m 2}}{(R_E \parallel \frac{1}{g_m 2}) + \frac{1}{g_m 1}} \right) \left(g_m 2 R_C \right)$

Small-signal model:





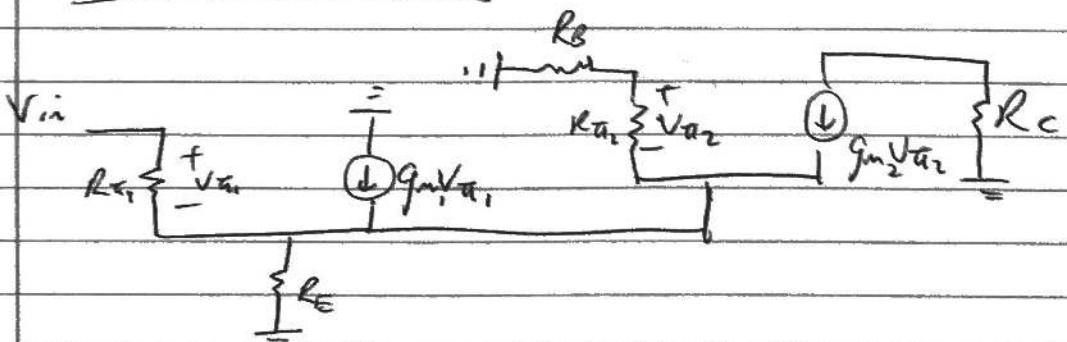
Second stage:

$$A_{v2} = g_{m2} R_C$$

Total gain:

$$A_v = A_{v1} \cdot A_{v2} = \left(\frac{R_E \parallel \left(\frac{1}{g_{m2}} + \frac{R_B}{\beta_2 + 1} \right)}{R_E \parallel \left(\frac{1}{g_{m2}} + \frac{R_B}{\beta_2 + 1} \right) + \frac{1}{g_{m1}}} \right) (g_{m2} R_C)$$

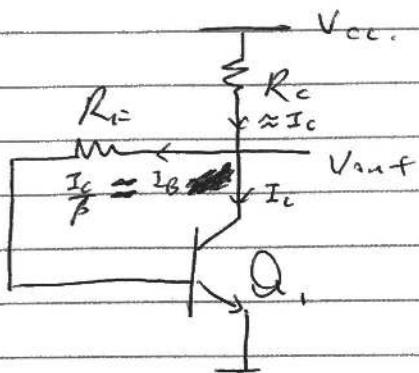
Small signal model:



(6)

Problem 2.

$$V_{CC} = 2.5V, \beta = 100, I_S = 5 \times 10^{-16} A.$$

a) Bias the circuit so that $I_C = 1mA$.

Use KVL:

$$V_{BE} + I_B R_F + I_C R_C = V_{CC}$$

$$I_B = \frac{1}{\beta} I_C$$

$$V_{BE} + \frac{1}{\beta} I_C R_F + I_C R_C = V_{CC}$$

We want to make the voltage V_{CB} small so that we can approximate $\frac{V_{CC} - V_{out}}{R_C} = I_C$. Thus: $\frac{1}{\beta} I_C R_F \ll I_C R_C$

$$\text{or } \frac{R_F}{\beta} \ll R_C$$

$$\text{So let } R_C = 10 \left(\frac{R_F}{\beta} \right)$$

$$V_{BE} + \frac{1}{\beta} I_C R_F + I_C \left(\frac{10}{\beta} R_F \right) = V_{CC}$$

$$V_{BE} + I_C R_F \left(\frac{11}{\beta} \right) = V_{CC}$$

$$\text{To find } V_{BE}: I_C = I_S \exp \left(\frac{V_{BE}}{V_T} \right) \Rightarrow V_{BE} = V_T \ln \left(\frac{I_C}{I_S} \right)$$

$$V_T \ln \left(\frac{I_C}{I_S} \right) + I_C R_F \left(\frac{11}{\beta} \right) = V_{CC}$$

Now we can find R_C, R_F .

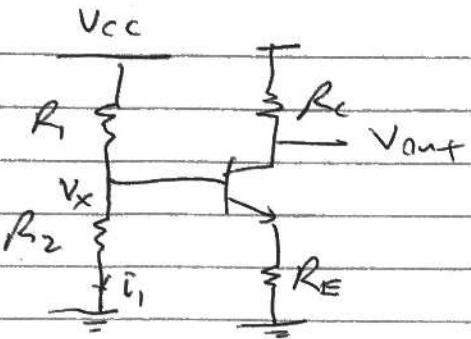
$$R_F = \frac{V_{CC} - V_T \ln \left(\frac{I_C}{I_S} \right)}{I_C \cdot \frac{11}{\beta}} = \frac{(2.5V) - (26mV) \ln \left(\frac{1mA}{5 \times 10^{-16} A} \right)}{(1mA) \cdot \frac{11}{100}}$$

$$= 16.0 k\Omega$$

$$R_C = \frac{10 R_F}{\beta} = \frac{R_F}{10} = 1.6 k\Omega$$

(7)

a, c or d)



Want to make $I_B \ll I_c$ (current through R_1 and R_2)

so that we can approximate $V_x \approx V_{ce} - \frac{I^2 R_2}{R_1 + R_2}$.

$$\text{Thus, let } 10I_B = I_c = \frac{V_{ce}}{R_1 + R_2}.$$

$$\text{Also know that } I_B = \frac{1}{\beta} I_c = \frac{1 \text{ mA}}{100}.$$

$$\Rightarrow R_1 + R_2 = \frac{V_{ce}}{10 I_B} = \frac{V_{ce}}{\frac{10}{100} \cdot 1 \text{ mA}} = \frac{2.5 \text{ V}}{\frac{1 \text{ mA}}{10}} = 25 \text{ k}\Omega.$$

$$\text{Also know that } V_{BE} = V_T \ln\left(\frac{I_c}{I_S}\right), \quad V_x = I_c R_E + V_{BE} \\ \approx I_c R_E + V_{BE}$$

$$\text{and that } I_c R_E + I_c R_C + V_{BE} = V_{cc}.$$

$$\text{and that } \frac{V_{cc} - V_{out}}{R_C} = I_c = 1 \text{ mA.}$$

Assume $V_{BE} = V_{ce}$ (on the edge of saturation), then

$$V_x = V_{out}, \Rightarrow \frac{V_{cc} - V_x}{R_C} = I_c \Rightarrow \frac{V_{cc} - (I_c R_E + V_{BE})}{R_C} = I_c,$$

$$\text{then } V_{cc} = I_c R_C + I_c R_E + V_{BE}.$$

Using the guideline: $R_E I_c > 4V_T$,

$$\text{let } R_E = \frac{4V_T}{I_c} = \frac{4(26 \text{ mV})}{1 \text{ mA}} = 104 \Omega$$

(8)

Solving for R_C :

$$V_{ce} = I_c R_C + V_{BE} + V_{CE}$$

$$\Rightarrow R_C = \frac{V_{ce} - I_c R_E - V_{BE}}{I_c}$$

$$= 2.5V - (1mA)(104\Omega) - (26mV) \ln \left(\frac{1mA}{5 \times 10^{-6}A} \right)$$

(mA)

$$\approx 1.66k\Omega$$

$$V_x \approx I_c R_E + V_{BE} = (1mA)(104\Omega) + (26mV) \ln \left(\frac{1mA}{5 \times 10^{-6}A} \right)$$

$$= 1.05V.$$

$$V_x = V_{ce} \cdot \frac{R_2}{R_1 + R_2}$$

$$\Rightarrow \frac{R_2}{R_1 + R_2} = \frac{V_x}{V_{ce}} = \frac{1.05V}{2.5V}$$

$$\Rightarrow R_2 = \left(\frac{1.05}{2.5} \right) (R_1 + R_2) = \left(\frac{1.05}{2.5} \right) (25k\Omega)$$

$$= 10.5k\Omega$$

$$R_1 = 25k\Omega - 10.5k\Omega = 14.5k\Omega.$$

$$R_1 = 14.5k\Omega$$

$$R_2 = 10.5k\Omega$$

$$R_E = 1.66k\Omega$$

$$R_E = 104\Omega$$

b) How much will V_{BE} change if V_{CC} is increased by 5%. Which circuit is less sensitive to changes?

Circuit (a) $V_{CC} \uparrow 5\% \Rightarrow V_{out} \uparrow 5\%$.

Since $\frac{1}{\beta} I_c$, current goes through the base,

$$V_{BE} \uparrow \frac{1}{\beta} I_c \cdot 5\%$$

Sorry, ran out of time here.

Circuit (b) : $V_{CC} \uparrow 5\% \Rightarrow V_x \uparrow 5\%$.

$$V_x \approx (0.41) V_{CC}$$

$$\text{so } V_x \text{ increases by roughly } (0.05)(0.41)(25V) = 0.05125V$$

If we assume all of this is initially absorbed by V_{BE} , then:

$$I_c = I_s \exp\left(\frac{V_{BE}}{V_T}\right) \Rightarrow I'_c = I_s \exp\left(\frac{V_{BE} + \Delta V_{BE}}{V_T}\right) = I_s \exp\left(\frac{V_{BE}}{V_T}\right) \times \exp\left(\frac{\Delta V_{BE}}{V_T}\right)$$
$$\exp\left(\frac{0.05125V}{26mV}\right) = 7.18.$$

So I_c increases by a large factor.

$$\text{Thus } I_c R_E = (7.18)(1mA)(10k\Omega) = 0.75V.$$

Thus the voltage V_x at the base is decreased again

Sorry, didn't finish these calculations.

c) Explain the mechanism by which each circuit inhibits changes in V_{BE} due to variation in V_{CC} .

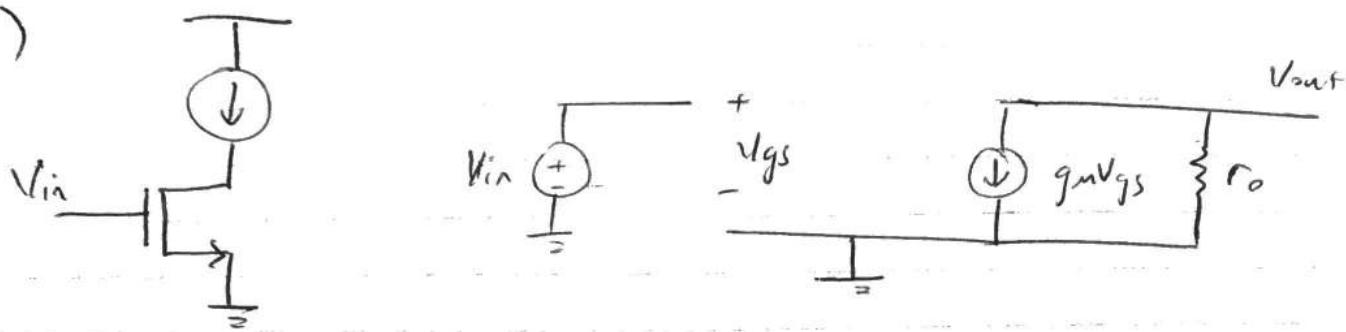
(10)

Circuit (a) (self-biasing circuit) uses a negative feedback loop to regulate its input. Increasing V_{BE} causes I_C to increase, which causes less current to go through the feedback loop to the base, which causes the voltage at the base to drop again.

Circuit (b) uses emitter degeneration to inhibit changes in V_{BE} . If V_{CC} increases, then the voltage at the base (call this V_x) also increases, but $V_x = V_{BE} + I_E R_E$. So the current I_E would also increase, "absorbing" much of the change in V_x so that V_{BE} doesn't change much.

Common-Source stage w/ current source load

a)



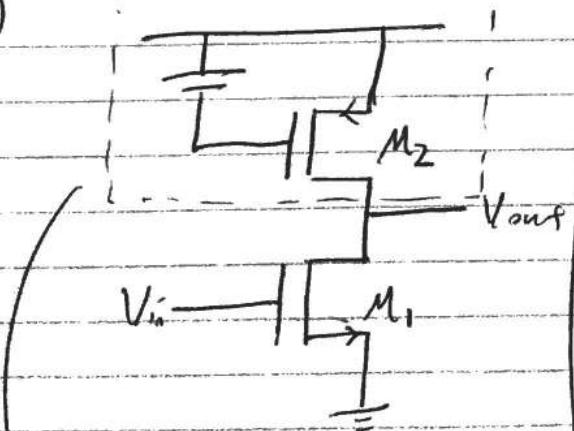
$$V_{out} = -g_m V_{gs} r_o = -g_m V_{in} r_o$$

$$\Rightarrow \frac{V_{out}}{V_{in}} = -g_m r_o$$

$R_{out} = r_o$ (since $V_{gs} = 0$, no current flows through the VCCS)

$A_v = -g_m r_o$
$R_{out} = r_o$
$R_{in} = \infty$

b.)



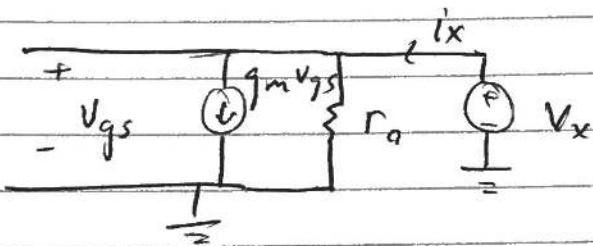
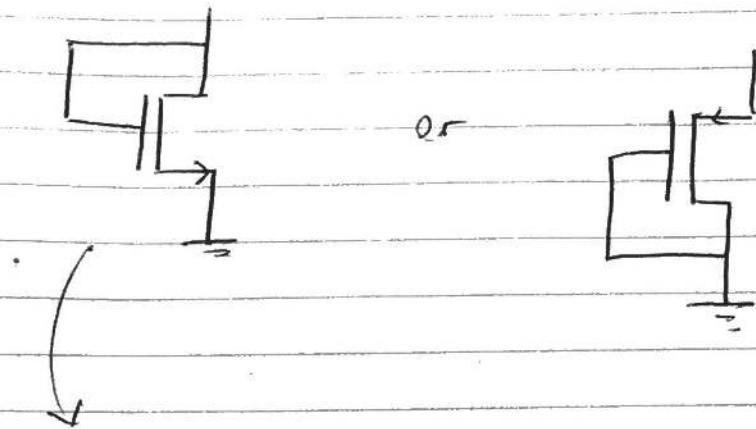
$A_v = -g_m (r_{out1} \parallel r_{out2})$
$= -g_m (r_o \parallel r_{o2})$

$R_{out} = r_{out1} \parallel r_{out2} = r_o \parallel r_{o2}$
$R_{in} = \infty$

→ find impedance looking into the drain (r_{out2})

by the symmetry of NMOS and PMOS, impedance looking into the drain $r_{out2} = r_{o2}$

A diode-connected device.



~~g_m = \frac{1}{r_o}~~

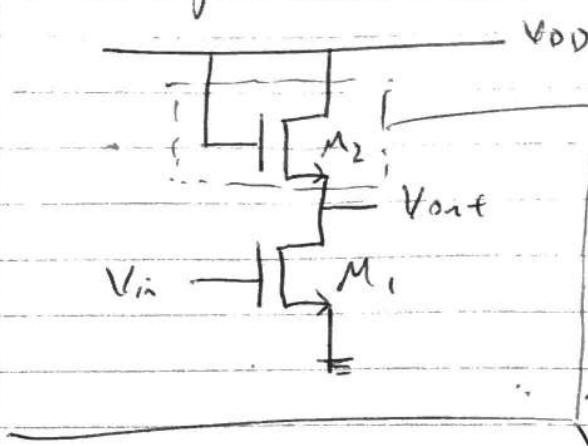
$$i_D = g_m V_x + \frac{V_x}{r_o}$$

$$i_D' = V_x \left(g_m + \frac{1}{r_o} \right)$$

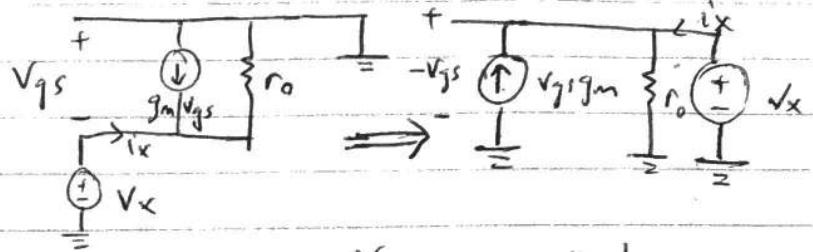
$$R_{out} = \frac{V_x}{i_D} = \frac{1}{g_m + \frac{1}{r_o}} \cdot \frac{\frac{1}{g_m}}{\frac{1}{g_m}} = \frac{r_o \cdot \frac{1}{g_m}}{r_o + \frac{1}{g_m}} = \left(r_o \parallel \frac{1}{g_m} \right)$$

$$R_{out} = r_o \parallel \frac{1}{g_m}$$

CS stage w/ Diode-connected load,



impedance looking into the source:



b) assume $\lambda \neq 0$:

$$\frac{V_x}{i_x} = r_o \parallel \frac{1}{g_m} \text{ again}$$

$R_{out} = R_{out}, \parallel \cancel{\text{impedance looking into source of } M_2}$

$$= r_o \parallel r_o \parallel \frac{1}{g_{m2}}$$

$$A_v = -g_m, R_{out} = -g_m, (r_o \parallel r_o \parallel \frac{1}{g_{m2}})$$

$$R_{in} = \infty.$$

for $\lambda = 0$. case, assume $r_o_1 = r_o_2 = \infty$.

a) $\lambda = 0$

$$A_v = -\frac{g_m}{g_{m2}}$$

$$R_{in} = \infty$$

$$R_{out} = \frac{1}{g_{m2}}$$

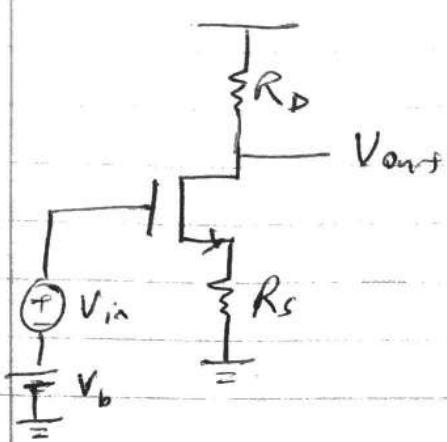
b) $\lambda \neq 0$

$$A_v = -g_m, (r_o \parallel r_o \parallel \frac{1}{g_{m2}})$$

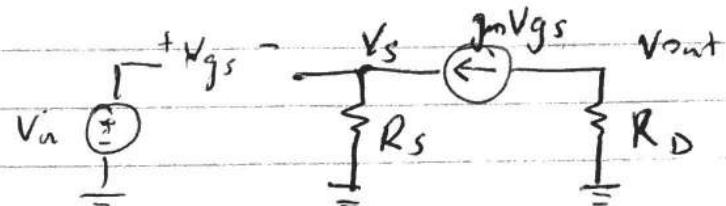
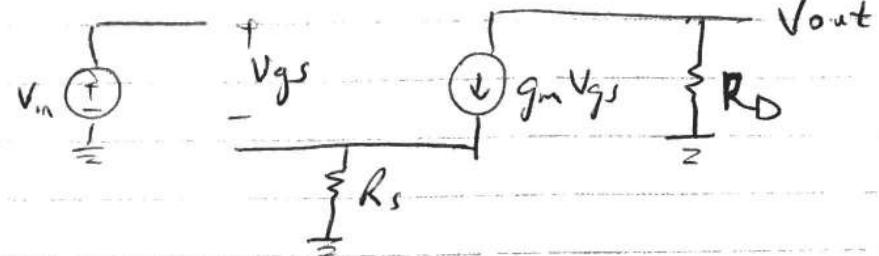
$$R_{in} = \infty$$

$$R_{out} = r_o \parallel r_o \parallel \frac{1}{g_{m2}}$$

Common-Source w/ Degeneration.



assume $\lambda = 0$



$$V_{gs} = V_{in} - V_s$$

$$g_m (V_{in} - V_s) = \frac{V_s}{R_s} = -\frac{V_{out}}{R_D}$$

$$V_s = \frac{R_s}{R_D} V_{out}$$

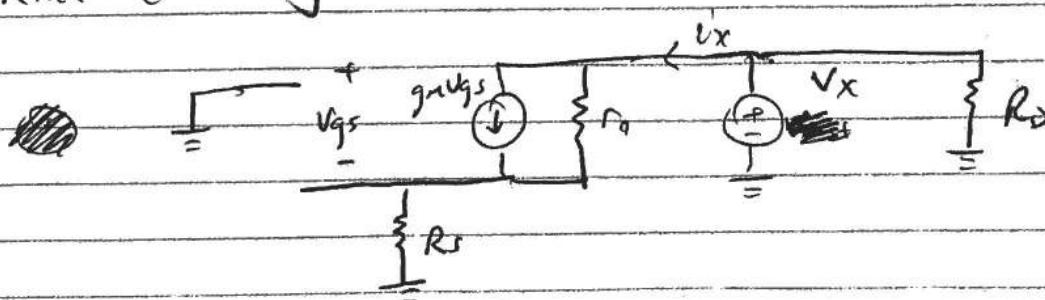
$$g_m \left(V_{in} + \frac{R_s}{R_D} V_{out} \right) = -\frac{V_{out}}{R_D}$$

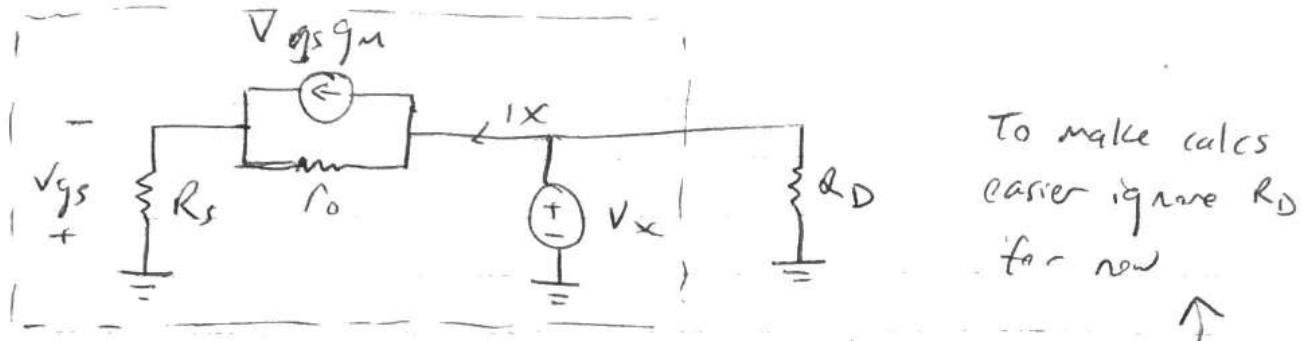
$$V_{in} (g_m R_D) = -V_{out} (1 + g_m R_s)$$

$$\frac{V_{out}}{V_{in}} = -\frac{g_m R_D}{1 + g_m R_s} = \boxed{-\frac{R_D}{\frac{1}{g_m} + R_s} = A_v}$$

(assuming $\lambda = 0$)

R_{out} (assuming $\lambda \neq 0$).





$$ix = -\frac{V_{gs}}{R_s} = gm V_{gs} + \frac{V_x + V_{gs}}{r_o}$$

$$\Rightarrow V_{gs} = -ix R_s$$

$$ix = gm(-ix R_s) + \frac{V_x}{r_o} + \frac{-ix R_s}{r_o}$$

$$ix \left(1 + gm R_s + \frac{R_s}{r_o} \right) = V_x \left(\frac{1}{r_o} \right)$$

$$R_{out} = \frac{V_x}{ix} = r_o \left(1 + gm R_s + \frac{R_s}{r_o} \right) = r_o (1 + gm R_s) + R_s$$

~~REMOVED~~

Note that this is in parallel with R_D

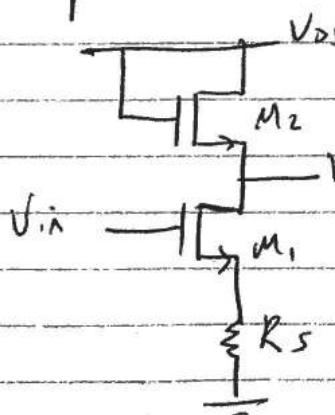
So: $R_{in} = \infty$

$$R_{out} = (r_o (1 + gm R_s) + R_s) \parallel R_D$$

Example.

$$R_D = r_{o_2} \parallel \frac{1}{gm_2}$$

$$R_{out} = (r_{o_1} (1 + gm_1 R_s) + R_s) \parallel (r_{o_2} \parallel \frac{1}{gm_2})$$



$$\text{assume } \lambda = 0 \Rightarrow r_{o_1} = r_{o_2} = \infty$$

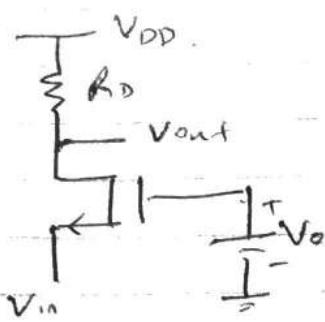
$$R_D = \frac{1}{gm_2}$$

$$R_{out} = \infty \parallel \infty \parallel \frac{1}{gm_2} = \frac{1}{gm_2}$$

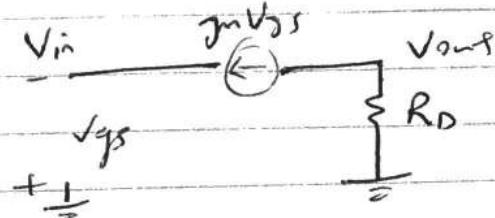
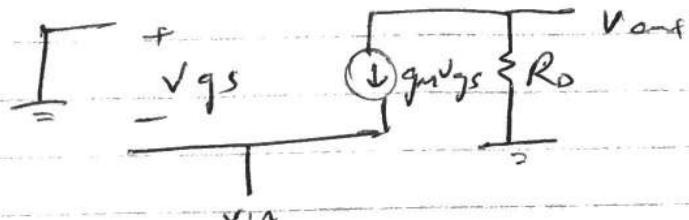
$$A_V = -\frac{R_D}{\frac{1}{gm_1} + R_s} = -\frac{\frac{1}{gm_2}}{\frac{1}{gm_1} + R_s}$$

assuming $\lambda = 0$, $R_{out} = \frac{1}{gm_2}$, $A_V = -\frac{\frac{1}{gm_2}}{\frac{1}{gm_1} + R_s}$

Common-Gate Topology

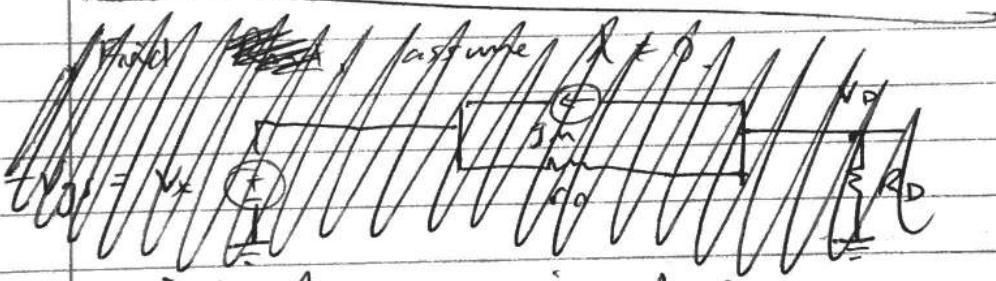


Find A_v , assume $\lambda = 0$.

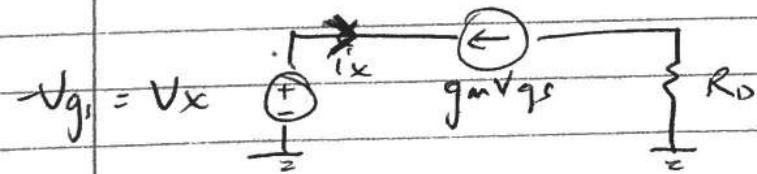


$$V_{out} = -gmV_{gs}R_D = gmV_{in}R_D$$

$$\Rightarrow \frac{V_{out}}{V_{in}} = gmR_D = A_v.$$



Find R_{in} , assume $\lambda = 0$

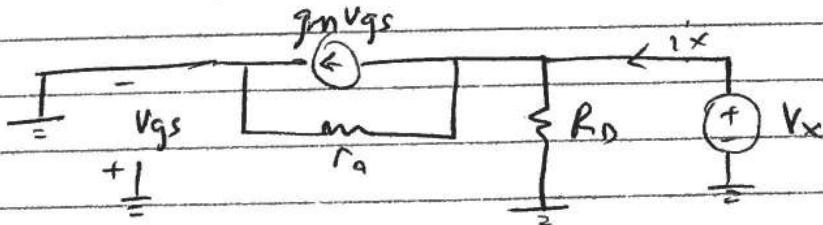


$$i_x = -gmV_{gs}$$

$$= gmV_x$$

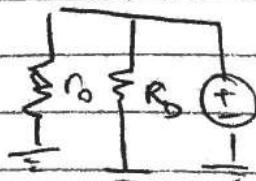
$$\Rightarrow \frac{V_x}{i_x} = \frac{1}{gm} = R_{in},$$

Find R_{out} , assume $\lambda \neq 0$.



$$V_g = V_s = 0 \Rightarrow V_{gs} = 0 \Rightarrow gmV_{gs} = 0 \Rightarrow$$

$$\Rightarrow R_{out} = R_D \parallel r_o.$$



$$\text{if } \lambda = 0, r_o = \infty, R_{out} = R_D.$$

Common Gate Topology

$$\lambda = 0$$

$$\lambda \neq 0$$

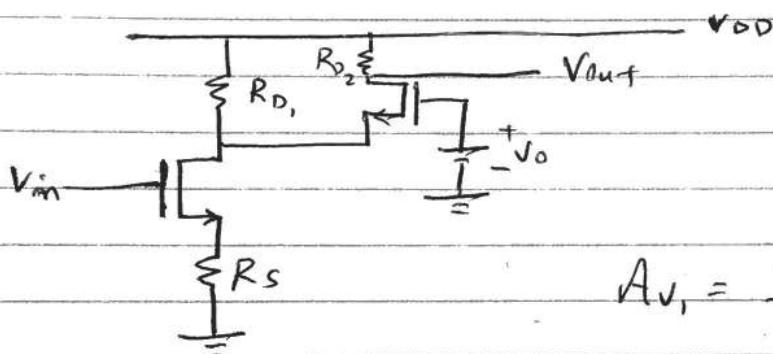
$$A_V = g_m R_D$$

$$R_{out} = R_D$$

$$R_{in} = \frac{1}{g_m}$$

$$R_{out} = r_0 \parallel R_D$$

Cascade Example



$$\lambda = 0$$

$$A_V = A_{V1}, A_{V2}$$

$$A_{V1} = -\frac{R_D \parallel R_{in2}}{\frac{1}{g_{m1}} + R_S}$$

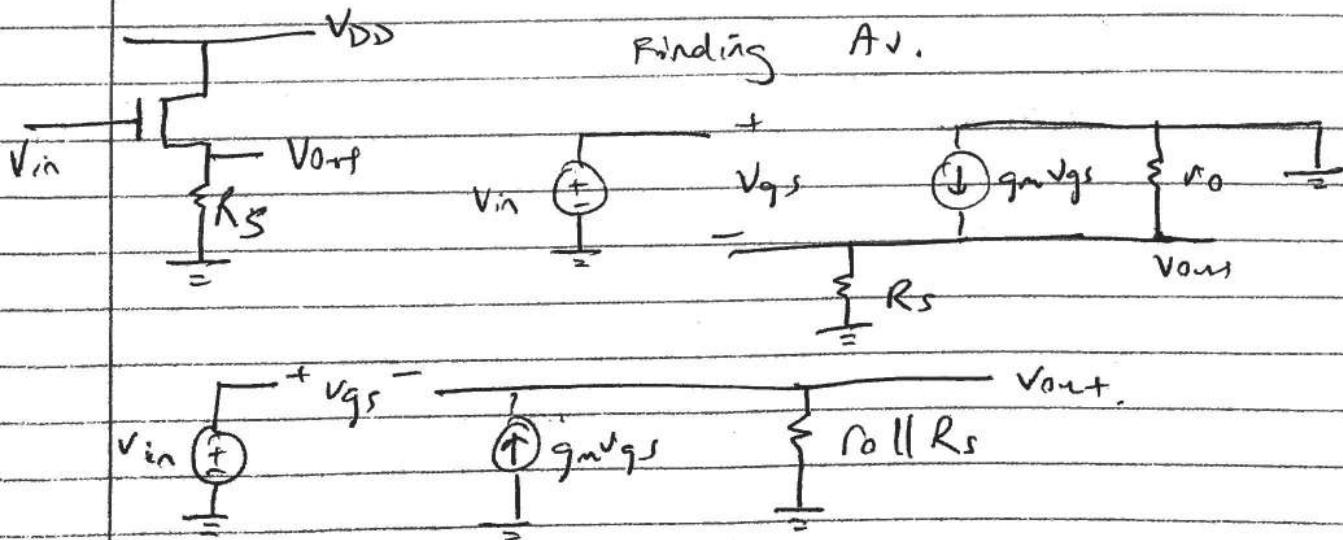
$$A_{V2} = g_{m2} R_{D2}$$

$$R_{in2} = \frac{1}{g_{m2}}$$

$$A_V = -\left(\frac{R_D \parallel \frac{1}{g_{m2}}}{\frac{1}{g_{m1}} + R_S}\right) \left(g_{m2} R_{D2}\right).$$

Source Follower.

$$\lambda \neq 0.$$



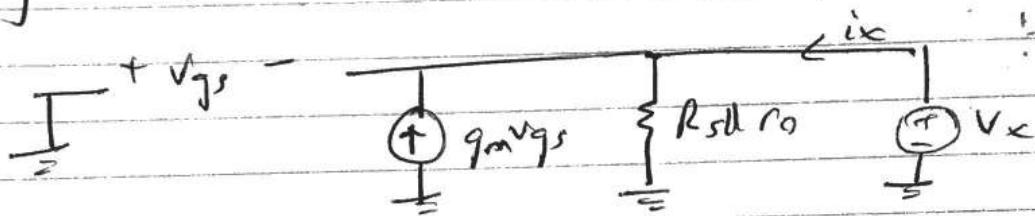
$$V_{out} = g_m V_{gs} (R_s \parallel r_o) \\ = g_m (V_{in} - V_{out}) (R_s \parallel r_o)$$

$$V_{out} (1 + g_m (R_s \parallel r_o)) = V_{in} (g_m (R_s \parallel r_o))$$

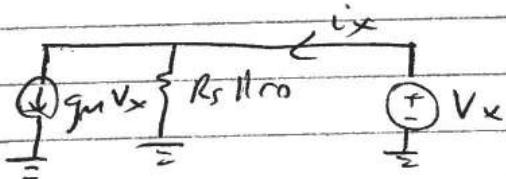
$$A_v = \frac{V_{out}}{V_{in}} = \frac{g_m (R_s \parallel r_o)}{1 + g_m (R_s \parallel r_o)} = \frac{R_s \parallel r_o}{\frac{1}{g_m} + (R_s \parallel r_o)}$$

Finding R_{in} , $\lambda \neq 0$. $R_{in} = \infty$.

Finding R_{out} , $\lambda \neq 0$.



$$V_{gs} = -V_x$$



impedance of current source is parallel w/ resistance

$$R_{out} = \frac{1}{g_m} \parallel (R_s \parallel r_o)$$

Source Follower

a) $\lambda = 0$

$$A_v = \frac{R_s}{\frac{1}{g_m} + R_s}$$

$$R_{in} = \infty$$

$$R_{out} = \frac{1}{g_m} \parallel R_s$$

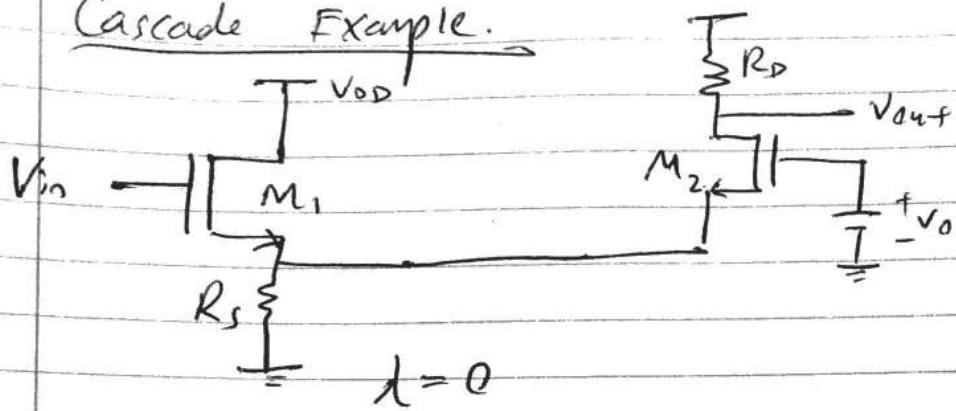
b) $\lambda \neq 0$.

$$A_v = \frac{R_s \parallel r_o}{\frac{1}{g_m} + R_s \parallel r_o}$$

$$R_{in} = \infty$$

$$R_{out} = \frac{1}{g_m} \parallel R_s \parallel r_o$$

Cascade Example.



$$A_V = A_{V1} A_{V2}$$

$$A_{V1} = \frac{R_s \parallel R_{in2}}{\frac{1}{g_m1} + R_s \parallel R_{in2}}$$

$$R_{in2} = \frac{1}{g_m2}$$

$$A_{V2} = g_m2 R_D$$

$$A_V = \left(\frac{R_s \parallel \frac{1}{g_m2}}{\frac{1}{g_m1} + R_s \parallel \frac{1}{g_m2}} \right) (g_m2 R_D)$$

ECE342 – Quiz 4

Jonathan Lam

December 19, 2020

1 MOS differential amplifier circuit

The circuit in Figure 1 uses transistors for which $\mu_n C_{ox} = 200 \mu\text{A/V}^2$, $V_{TH} = 2\text{V}$ and $\lambda = 0.02\text{V}^{-1}$. The supplies are $V_{DD} = 12\text{V}$ and $V_{SS} = -12\text{V}$.

Resistor	Value (kΩ)
R_1	55
R_D	40
$R_{D,2}$	4
R_5	6

Table 1: Resistor values for question 1

1.1 DC analysis

The first thing to do is to choose a value of the aspect ratio W/L . Since this is not provided in the problem, I chose an arbitrary value that I believe is reasonable (similar to some of the values provided in the textbook problems):

$$\frac{W}{L} = 15 \quad (1)$$

Assume all transistors are in saturation mode – although it's not shown here, all of the transistors can be shown to be in saturation mode by showing that $V_{DS} > V_{GS} - V_{TH}$ and $V_{GS} > V_{TH}$. We first have to examine the biasing of the circuit, and thus the current mirror at M_4 and M_5 . This can be solved for with iteration using the following two equations (see Figure 6 for the iteration source code):

$$V_{GS,4} = (V_{DD} - I_{D,4}R_1) - V_{SS} \quad (2)$$

$$I_{D,4} = \frac{1}{2}\mu_n C_{ox} \frac{W}{L} (V_{GS,4} - V_{TH})^2 (1 + \lambda V_{DS,4}) \quad (3)$$

I tried these in Python but it wouldn't converge, so I tried rearranging them:

$$I_{D,4} = \frac{V_{DD} - V_{GS,4} - V_{SS}}{R_1} \quad (4)$$

$$V_{GS,4} = V_{TH} + \sqrt{\frac{2I_{D,4}L}{\mu_n C_{ox} W (1 + \lambda V_{DS,4})}} \quad (5)$$

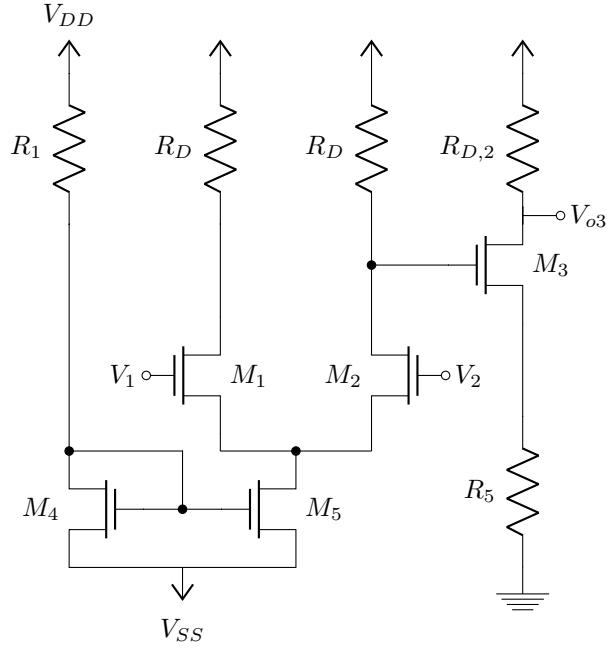


Figure 1: Schematic for question 1

Using the initial values $I_{D,1} = 1\text{mA}$ and $V_{GS,1} = 2.1\text{V}$ results in the values:

$$I_{D,4} = 0.391\text{mA} \quad (6)$$

$$V_{GS,4} = 2.50\text{V} \quad (7)$$

This current should be equal to the drain current through M_5 , and equal to double the drain currents through the differential pair M_1 and M_2 .

$$I_{D,4} = I_{D,5} = 2I_{D,1} = 2I_{D,2} \quad (8)$$

$$V_{D,1} = V_{D,2} = V_{DD} - I_{D,1}R_D = 12\text{V} - (0.196\text{mA})(40\text{k}\Omega) = 4.18\text{V} \quad (9)$$

$$V_{GS,1} = V_{GS,2} = V_{TH} + \sqrt{\frac{2I_{D,1}L}{\mu_n C_{ox} W(1 + \lambda V_{DS,1})}} \quad (10)$$

Assuming that initially the inputs are centered at DC 0V, then $V_{DS,1} = V_{D,1} - (0 - V_{GS,1}) = V_{D,1} + V_{GS,1}$. We can solve this by iterating this one equation until convergence (in Python):

$$V_{DS,1} = V_{DS,2} = 6.52\text{V} \quad (11)$$

$$V_{GS,1} = V_{GS,2} = 2.34\text{V} \quad (12)$$

Looking at the last transistor M_3 , we know the gate voltage, but we do not know the source voltage, drain voltage, or drain current. However, these values can all be expressed as functions of $I_{D,3}$ and

$V_{GS,3}$, so we only need to solve for those two variables:

$$V_{G,3} = V_{D,2} \quad (13)$$

$$V_{D,3} = V_{DD} - I_{D,3}R_{D,2} \quad (14)$$

$$V_{S,3} = V_{G,3} - V_{GS,3} \quad (15)$$

We can solve for the two missing variables $I_{D,3}$ and $V_{GS,3}$ by iterating the two equations:

$$I_{D,3} = \frac{(V_{G,3} - V_{GS,3}) - V_{SS}}{R_5} \quad (16)$$

$$V_{GS,3} = V_{TH} + \sqrt{\frac{2I_{D,3}L}{\mu_n C_{ox} W (1 + \lambda((V_{DD} - I_{D,1}R_{D,2}) - (V_{G,3} - V_{GS,3})))}} \quad (17)$$

where the long last term in the denominator is equal to $V_{DS,3}$. By iterating in Python with initial values $I_{D,3} = 1\text{mA}$ and $V_{GS,3} = 2.1\text{V}$, we get:

$$I_{D,3} = 0.295\text{mA} \quad (18)$$

$$V_{GS,3} = 2.41\text{V} \quad (19)$$

and from this we are able to calculate the rest of the voltages:

$$V_{D,3} = V_{DD} - I_{D,3}R_{D,2} = 12\text{V} - (0.295\text{mA})(4\text{k}\Omega) = 10.8\text{V} \quad (20)$$

$$V_{S,3} = V_{G,3} - V_{GS,3} = 4.18\text{V} - 2.41\text{V} = 1.78\text{V} \quad (21)$$

1.2 Differential mode voltage gain

Decompose the problem into two stages: the first stage is the differential pair with M_1 and M_2 with gain $A_{V,1}$ and the second stage is the CS stage M_3 with voltage gain $A_{V,2}$. The total gain is the product of these gains.

For the differential pair, the voltage gain is equal to half of the voltage gain of one of the CS transistors. It is only half because only one of the differential pair outputs is used.

$$A_{V,1} = \frac{1}{2}(-g_{m,1}(R_D || R_{O,1})) \quad (22)$$

$$R_{O,1} \approx \frac{1}{\lambda I_{D,1}} = \frac{1}{(0.02\text{V}^{-1})(0.391/2\text{mA})} = 256\text{k}\Omega \quad (23)$$

$$g_{m,1} = \mu_n C_{ox} \frac{W}{L} (V_{GS,1} - V_{TH}) = (200\mu\text{A/V}^2)(15)(2.34\text{V} - 2\text{V}) = 0.00102\Omega^{-1} \quad (24)$$

$$A_{V,1} = -\frac{(0.00102\Omega^{-1})(40\text{k}\Omega || 256\text{k}\Omega)}{2} = -17.6 \quad (25)$$

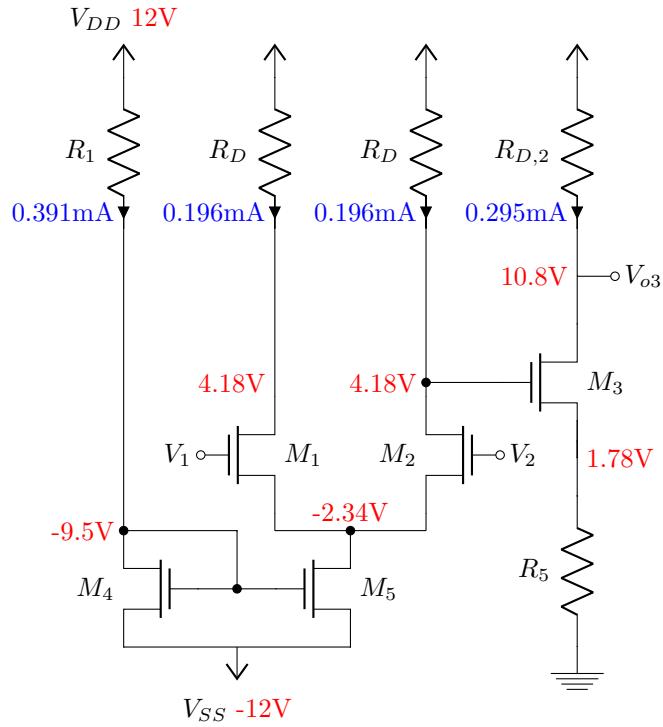


Figure 2: Schematic for question 1 labeled with DC voltages and currents

The second stage is a CS stage with emitter degeneration:

$$A_{V,2} = \frac{-R_{D,2}||R_{O,3}}{\frac{1}{g_{m,3}} + R_{S,3}} \quad (26)$$

$$R_{O,3} \approx \frac{1}{\lambda I_{D,3}} = \frac{1}{(0.02V^{-1})(0.295mA)} = 169k\Omega \quad (27)$$

$$g_{m,3} = \mu_n C_{ox} (V_{GS,3} - V_{TH}) = (200\mu A/V^2)(15)(2.41V - 2V) = 0.00123\Omega^{-1} \quad (28)$$

$$A_{V,2} = -\frac{4k\Omega||169k\Omega}{\frac{1}{0.00123\Omega^{-1}} + 6k\Omega} = -0.574 \quad (29)$$

The total voltage gain is thus:

$$A_V = (-17.6)(-0.574) = 10.1 \quad (30)$$

1.3 Common mode voltage gain

The common mode voltage gain of a differential pair is zero due to its CM rejection property. Thus, the first stage would have a CM voltage gain of zero, leading to the entire circuit having a CM voltage gain of zero.

2 Bipolar differential amplifier circuit

Consider the bipolar op-amp circuit in Figure 3. For simplicity, you may assume $|V_{BE}| = 0.7V$, $\beta = 100$, and $V_A = \infty$. The supplies are $V_{DD} = 15V$ and $V_{SS} = -15V$.

Note: I rename the resistors for sake of my sanity. Each resistor is labeled with C or E and a number to indicate what transistor it is associated with, and whether it is connected to the collector or the emitter of that transistor.

Resistor	Value ($k\Omega$)
$R_{C,1}$	20
$R_{C,2}$	20
$R_{C,5}$	3
$R_{E,7}$	2.3
$R_{C,7}$	15.7
$R_{E,8}$	3
$R_{C,9}$	28.6

Table 2: Resistor values for question 2

2.1 DC analysis

Like in the previous question, assume all BJTs are in FA mode. This is not checked explicitly, but can be checked by making sure that $V_{CE} > V_{BE}$. We start by examining the current mirror at Q_9 :

$$V_{B,9} = V_{EE} + V_{BE} = -15V + 0.7V = -14.3V = V_{C,9} \quad (31)$$

$$I_{C,9} = \frac{GND - V_{C,9}}{R_{C,9}} = \frac{0 - (-14.3V)}{28.6k\Omega} = 0.5mA \quad (32)$$

Since this current mirror's current is only being replicated $n = 5$ times and β is fairly large, we can approximate the copied current to be equal to the reference one. I.e.:

$$I_{C,3} = I_{C,9} = \frac{1}{4}I_{C,6} \quad (33)$$

Now we have the emitter current biases for the two differential pairs. Since all of the V_{BE} s are equal, we would expect the currents $I_{C,1}$ and $I_{C,2}$ to be equal and half of $I_{C,3}$. Neglecting any base currents:

$$I_{C,1} \approx I_{C,2} \approx I_{E,1} = I_{E,2} = \frac{1}{2}I_{C,3} = 0.25mA \quad (34)$$

$$V_{C,1} \approx V_{C,2} \approx V_{CC} - I_{C,1}R_{C,1} = 15V - (0.25mA)(20k\Omega) = 10V \quad (35)$$

We also assume the inputs would be centered at DC 0V, thus:

$$V_{B,1} = V_{B,2} = 0V \quad (36)$$

$$V_{E,1} = V_{E,2} = V_{C,3} = V_{B,1} - V_{BE} = 0V - 0.7V = -0.7V \quad (37)$$

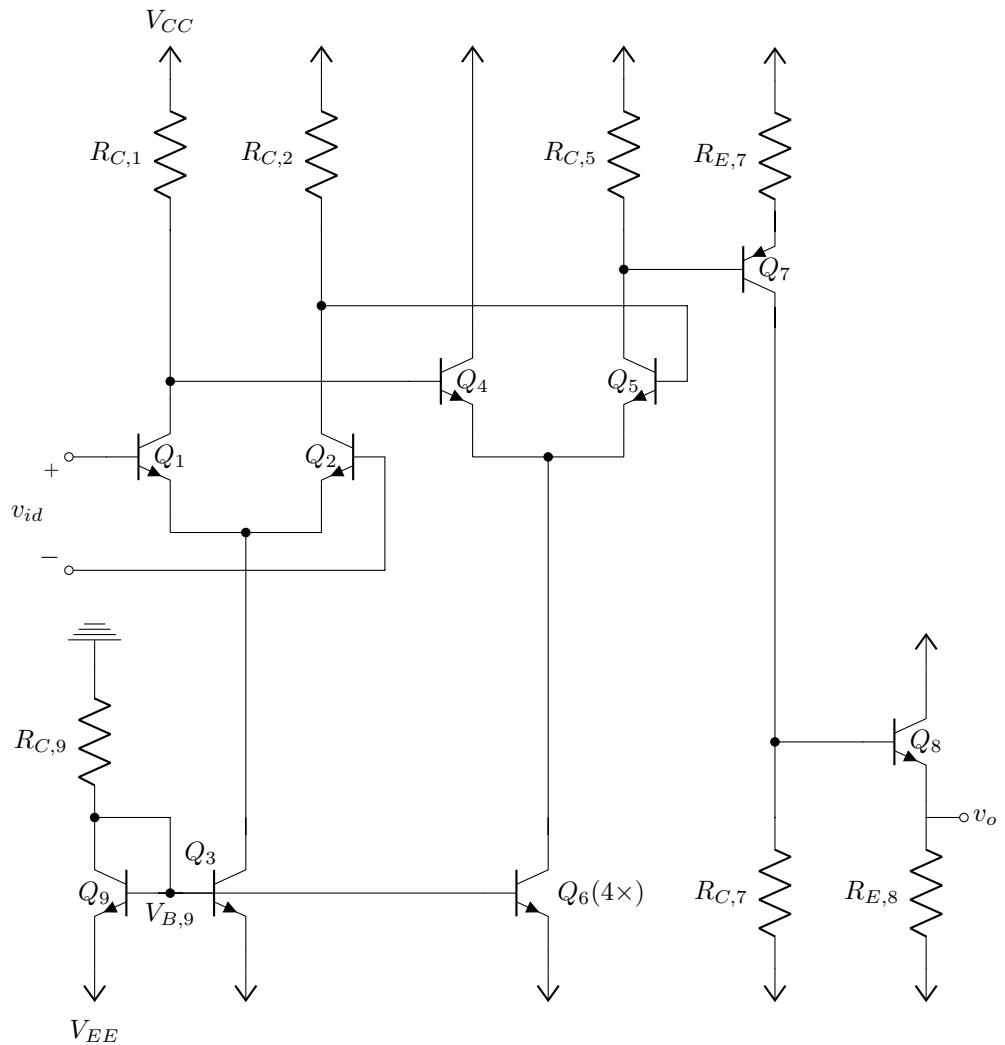


Figure 3: Schematic for question 2

Moving onto the second differential pair, again assume that the currents are split evenly between Q_4 and Q_5 :

$$I_{C,4} \approx I_{C,5} \approx I_{E,4} = I_{E,5} = \frac{1}{2}I_{Q,6} = \frac{1}{2}(4)(0.5\text{mA}) = 1\text{mA} \quad (38)$$

$$V_{C,5} = V_{CC} - I_{C,5}R_{C,5} = 15\text{V} - (1\text{mA})(3\text{k}\Omega) = 12\text{V} \quad (39)$$

$$V_{B,4} = V_{B,5} = V_{C,1} = V_{C,2} = 10\text{V} \quad (40)$$

$$V_{E,4} = V_{E,5} = V_{C,6} = V_{B,4} - V_{BE} = 10\text{V} - 0.7\text{V} = 9.3\text{V} \quad (41)$$

Looking at Q_7 and Q_8 :

$$V_{B,7} = V_{C,5} = 12\text{V} \quad (42)$$

$$V_{E,7} = V_{B,7} + V_{EB} = 12\text{V} + 0.7\text{V} = 12.7\text{V} \quad (43)$$

$$I_{E,7} = \frac{V_{CC} - V_{E,7}}{R_{C,7}} = \frac{15\text{V} - 12.7\text{V}}{2.3\text{k}\Omega} = 1\text{mA} \approx I_{C,7} \quad (44)$$

$$V_{C,7} = V_{EE} + I_{C,7}R_{C,7} = -15\text{V} + (1\text{mA})(15.7\text{k}\Omega) = 0.7\text{V} = V_{B,8} \quad (45)$$

$$V_{E,8} = V_{B,8} - V_{BE} = 0.7\text{V} - 0.7\text{V} = 0\text{V} = v_o \quad (46)$$

$$I_{E,8} = \frac{V_{E,8} - V_{EE}}{R_{E,8}} = \frac{0\text{V} - (-15\text{V})}{3\text{k}\Omega} = 5\text{mA} \approx I_{C,8} \quad (47)$$

2.2 Quiescent power dissipation

To find this, we look at the sum of the DC currents at the voltage sources multiplied by the source voltages. We then take the absolute value since power is always positive.

$$\begin{aligned} P &= \sum |IV| \\ &= |((2)(0.25\text{mA}) + (2)(1\text{mA}) + 1\text{mA} + 5\text{mA})(15\text{V})| \\ &\quad + |(0.5\text{mA} + 0.5\text{mA} + 2\text{mA} + 1\text{mA} + 5\text{mA})(-15\text{V})| \\ &= 262.5\text{mW} \end{aligned} \quad (48)$$

2.3 Voltage gain

We break this up into stages. Call the first differential pair (Q_1, Q_2) the first stage with gain $A_{V,1}$; the second differential pair (Q_4, Q_5) the second stage with gain $A_{V,2}$; the PNP common-emitter Q_7 the third stage with gain $A_{V,3}$, and the emitter follower Q_8 the final stage with gain $A_{V,4}$. (Note that this deviates from the earlier convention where the index denotes the associated transistor; here the index denotes the stage for which we're examining the voltage gain) The total gain is the product of these values, i.e.:

$$A_V = \prod_{i=1}^4 A_{V,i} \quad (49)$$

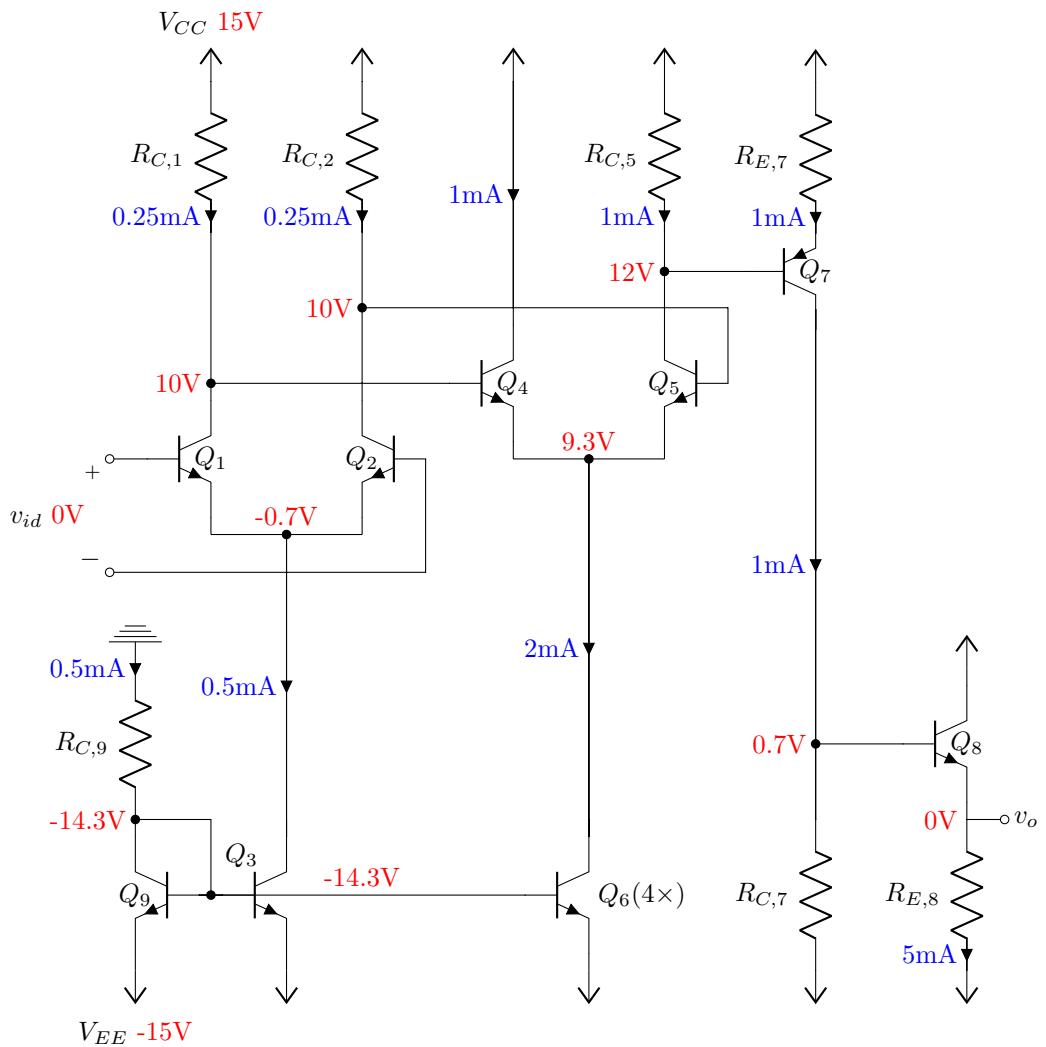


Figure 4: Schematic for question 2 with labeled DC currents and voltages

Looking at the first differential pair, Q_1 and Q_2 are symmetric and the differential voltage gain is equal to the voltage gain of either one.

$$A_{V,1} = -g_{m_1} R_{out,1} \quad (50)$$

$$g_{m_1} = \frac{I_C}{V_T} = \frac{0.25\text{mA}}{26\text{mV}} = 0.00962\Omega^{-1} \quad (51)$$

$$R_{out,1} = R_{C,1} \parallel R_{in,4} \quad (52)$$

Due to the “virtual ground” nature of the emitter in a differential pair, it is easy to find the input resistance of each common emitter in a differential pair:

$$R_{in,4} = R_{\pi,4} = \frac{\beta}{g_m} = \frac{100}{\frac{1\text{mA}}{26\text{mV}}} = 2600\Omega \quad (53)$$

$$A_{V,1} = -g_{m_1} R_{out,1} = -(0.00962\Omega^{-1})(20\text{k}\Omega \parallel 2.6\text{k}\Omega) = -22.1 \quad (54)$$

Looking at the second differential pair, the output only uses one end of the differential pair, so the voltage gain is half of the voltage gain of the Q_5 CE:

$$A_{V,2} = \frac{1}{2} g_{m,5} R_{out,5} \quad (55)$$

$$g_{m,5} = \frac{I_{C,5}}{V_T} = \frac{1\text{mA}}{26\text{mV}} = 0.0385\Omega^{-1} \quad (56)$$

$$R_{out,5} = R_{C,5} \parallel R_{in,7} \quad (57)$$

Q_7 is just a single CE transistor (with emitter degeneration), so its input impedance is given by:

$$R_{in,7} = R_{\pi,7} + R_{E,7}(\beta + 1) = \frac{\beta}{g_{m,7}} + R_{E,7}(\beta + 1) = \frac{100}{\frac{1\text{mA}}{26\text{mV}}} + (2.3\text{k}\Omega)(101) = 235\text{k}\Omega \quad (58)$$

$$A_{V,2} = -\frac{1}{2} g_{m,5} (R_{C,5} \parallel R_{in,7}) = -\frac{1}{2} (0.00385\Omega^{-1})(3\text{k}\Omega \parallel 235\text{k}\Omega) = -57.0 \quad (59)$$

Now, looking at the third stage, Q_7 , which is a CE stage with degeneration:

$$A_{V,3} = -\frac{g_{m,7} R_{out,7}}{1 + g_{m,7} R_{E,7}} \quad (60)$$

$$g_{m,7} = \frac{I_{C,7}}{V_T} = \frac{1\text{mA}}{26\text{mV}} = 0.0385\Omega^{-1} \quad (61)$$

$$R_{out,7} = R_{C,7} \parallel R_{in,8} \quad (62)$$

$$R_{in,8} = R_{\pi,8} + R_{E,8}(\beta + 1) = \frac{\beta}{g_{m,8}} + R_{E,8}(\beta + 1) = \frac{100}{\frac{5\text{mA}}{26\text{mV}}} + 3.0\text{k}\Omega(101) = 304\text{k}\Omega \quad (63)$$

$$A_{V,3} = -\frac{g_{m,7} (R_{C,7} \parallel R_{in,8})}{1 + g_{m,7} R_{E,7}} = -\frac{(0.0385\Omega^{-1})(15.7\text{k}\Omega \parallel 304\text{k}\Omega)}{1 + (0.0385\Omega^{-1})(2.3\text{k}\Omega)} = -6.42 \quad (64)$$

And finally, the last stage, Q_8 , which is an emitter follower:

$$A_{V,4} = \frac{R_{E,8}}{R_{E,8} + \frac{1}{g_{m,8}}} = \frac{3\text{k}\Omega}{3\text{k}\Omega + \frac{26\text{mV}}{5\text{mA}}} = 0.998 \quad (65)$$

Thus the total gain is:

$$A_V = \prod_{i=1}^4 A_{V,i} = (-22.1)(-57.0)(-6.42)(0.998) = -8070 \quad (66)$$

(There may be a lot of roundoff error here, due to rounding in between steps.)

2.4 Input and output impedances

2.4.1 Input impedance

Since we have a differential input (between two input terminals, not between an input terminal and ground), the setup is a little different, as shown in Figure 5.

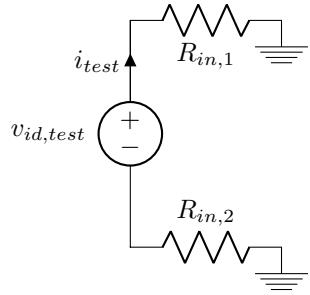


Figure 5: Input impedance setup for a two-terminal (differential) circuit

Due to the symmetry of Q_1 and Q_2 , we expect $R_{in,1} = R_{in,2}$, so we expect that:

$$R_{in} = \frac{1}{2}(R_{in,2} + R_{in,1}) = R_{in,1} \quad (67)$$

Here, $R_{in,1}$ is a CE stage with degeneration, where the emitter impedance is the impedance looking into the emitter of Q_2 in parallel with the impedance looking into the collector of Q_3 :

$$R_{in} = R_{in,1} = R_{\pi,1} + R_{E,1}(\beta + 1) \quad (68)$$

$$R_{E,1} = R_{\text{into emitter of } Q_2} \parallel R_{\text{into collector of } Q_3} \quad (69)$$

$$R_{\text{into emitter of } Q_2} = \frac{R_{\pi,2}}{\beta + 1} \quad (70)$$

$$R_{\text{into collector of } Q_3} \approx \infty \quad (71)$$

$$R_{E,1} = \frac{R_{\pi,2}}{\beta + 1} \parallel \infty = \frac{R_{\pi,2}}{\beta + 1} \quad (72)$$

$$R_{in} = R_{\pi,1} + \frac{\beta + 1}{\beta + 1} R_{\pi,2} = 2R_{\pi,1} \quad (73)$$

$$R_{\pi,1} = \frac{\beta}{I_{C,1}} = \frac{100}{\frac{0.25\text{mA}}{26\text{mV}}} = 10.4\text{k}\Omega \quad (74)$$

$$R_{in} = (2)(10.4\text{k}\Omega) = 20.8\text{k}\Omega \quad (75)$$

2.4.2 Output impedance

The output impedance is that of a emitter-follower with a resistance in series with the base (R_S in series with $R_{\pi,8}$). (This output impedance is Eq. 5.329 in the textbook.)

$$R_{in,8} = R_{E,8} \parallel \left(\frac{R_{out,7}}{\beta + 1} + \frac{1}{g_{m,8}} \right) \quad (76)$$

$$R_{out,7} = R_{C,7} \parallel R_{\text{into collector of } Q_7} \quad (77)$$

$$R_{\text{into collector of } Q_7} \approx \infty \quad (78)$$

$$R_{out,7} = R_{C,7} \parallel \infty = R_{C,7} \quad (79)$$

$$R_{in,8} = R_{E,8} \parallel \left(\frac{R_{C,7}}{\beta + 1} + \frac{1}{g_{m,8}} \right) = (3k\Omega) \parallel \left(\frac{15.7k\Omega}{101} + \frac{26mV}{5mA} \right) = 152\Omega \quad (80)$$

```

import numpy as np
from math import sqrt

mu_c = 200e-6    # mu_n * C_{ox}
wol = 15          # W/L
v_th = 2           # V_{TH}
lam = 0.02         # lambda

r_1 = 55e3
r_d = 40e3
r_d2 = 4e3
r_5 = 6e3

v_dd = 12
v_ss = -12

# finding i_d4, v_gs4
i_d4 = 1e-3      # I_{D,4}
v_gs4 = 2.1       # V_{GS,4} = V_{DS,4}

def iterate():
    global i_d4, v_gs4
    i_d4 = (v_dd - v_gs4 - v_ss) / r_1
    v_gs4 = v_th + sqrt(2*i_d4/(wol * mu_c * (1 + lam * v_gs4)))
    print(f'i_d4 {i_d4} v_gs4 {v_gs4}')

print(f'i_d4 {i_d4} v_gs4 {v_gs4}')
for _ in range(5):
    iterate()

# finding v_gs1, v_gs2
i_d1 = i_d4/2
v_d1 = 12-r_d*i_d1
v_gs1 = 2.1

def iterate():
    global v_gs1
    v_ds1 = v_d1 + v_gs1
    v_gs1 = v_th + sqrt(2 * i_d1 / (mu_c * wol * (1 + lam * v_ds1)))
    print(f'v_ds1 {v_ds1} v_gs1 {v_gs1}')

print(f'v_gs1 {v_gs1}')
for _ in range(5):
    iterate()

# finding i_d3, v_gs3
v_g3 = v_d1
i_d3 = 1e-3
v_gs3 = 2.1

def iterate():
    global i_d3, v_gs3
    i_d3 = (v_g3 - v_gs3) / r_5
    v_d3 = v_dd - i_d3 * r_d2
    v_s3 = v_g3 - v_gs3
    v_ds3 = v_d3 - v_s3
    v_gs3 = v_th + sqrt(2 * i_d3 / (mu_c * wol * (1 + lam * v_ds3)))
    print(f'i_d3 {i_d3} v_gs3 {v_gs3}')

print(f'i_d3 {i_d3} v_gs3 {v_gs3}')
for _ in range(5):
    iterate()

```

Figure 6: Source code for iteration steps in question 1

ECE393 – Lab 1

Jonathan Lam

November 12, 2020

Labs 1-1 through 1-6 from *Student Manual for the Art of Electronics*.

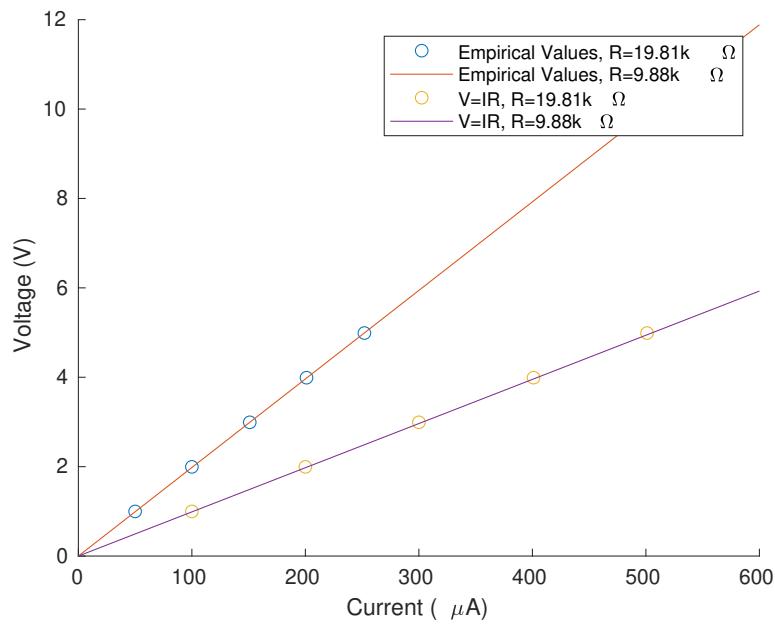
1 Ohm's law

V (V)	I (μ A)	V/I (k Ω)
4.989	252	19.8
3.990	201	19.9
2.992	151	19.8
1.995	100	20.0
0.998	50	20.0

(a) $R = 19.81\text{k}\Omega$

V (V)	I (μ A)	V/I (k Ω)
4.989	501	9.96
3.991	401	9.95
2.992	300	9.97
1.995	200	9.98
0.998	100	9.98

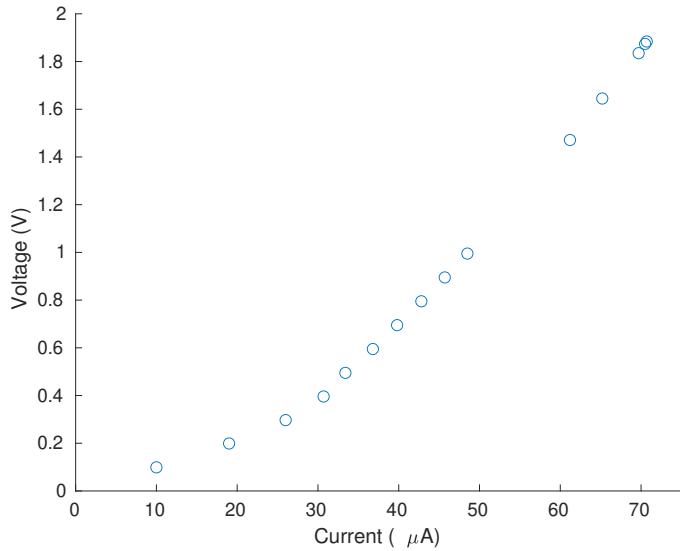
(b) $R = 9.88\text{k}\Omega$



2 An incandescent lamp

The ADALM2000 power supply has a maximum current rating of $50\mu\text{A}$. Thus, even if we attempt to set the voltage (V_{att}) to some high values, the actual applied voltage (V_{app}) will drop because the device can't source enough current. The maximum current observed from the device was roughly $70\mu\text{A}$.

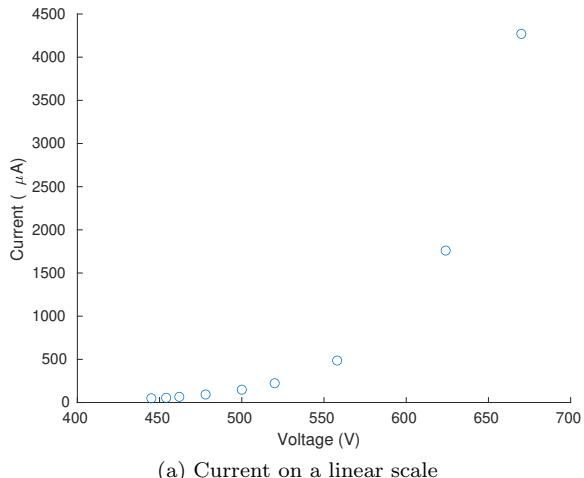
V_{att} (V)	V_{app} (V)	I (μA)	V_{app}/I ($\text{k}\Omega$)
5.0	1.884	70.7	26.6
4.0	1.874	70.5	26.6
3.0	1.835	69.7	27.1
2.0	1.645	65.2	25.2
1.5	1.471	61.2	24.0
1.0	0.995	48.5	20.5
0.9	0.895	45.7	19.6
0.8	0.795	42.8	18.6
0.7	0.695	39.8	17.5
0.6	0.595	36.8	16.2
0.5	0.495	33.4	14.8
0.4	0.396	30.7	12.9
0.3	0.297	26.0	11.4
0.2	0.199	19.0	10.5
0.1	0.099	10.0	9.90



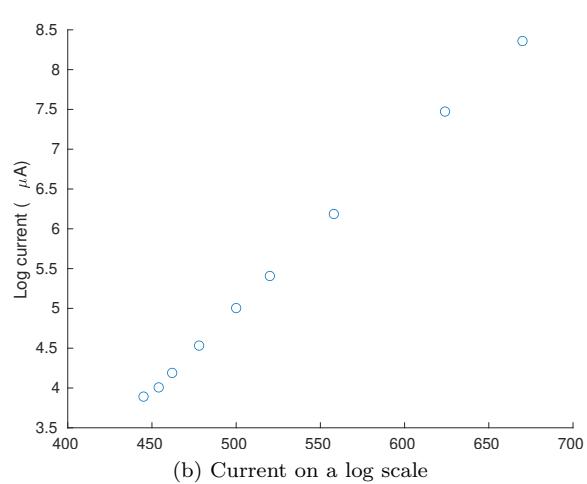
We see a slight nonlinearity (almost linear, not an exponential nonlinearity) in the voltage vs. current curve. For higher currents (and higher voltages), the slope of the chart (the effective resistance) seems to increase.

3 The diode

V (mV)	I (μ A)
445	49
454	55
462	66
478	93
500	149
520	223
558	486
624	1760
670	4270



(a) Current on a linear scale



(b) Current on a log scale

It is clear that the I/V relationship of a diode is exponential. If the diode is reversed, then the voltage and current through the diode are both zero.

4 DC Voltage divider

$V_{in} = 5V$ rather than 15V because this is the maximum voltage of the ADALM2000 power supply.

	Voltage divider	Thevenin equivalent
S.C. Voltage (Thevenin Voltage) (V)	2.48	2.47
S.C. Current (Thevenin Current) (μA)	493	492
Voltage over 10k Ω load (V)	1.65	1.65
Current through 10k Ω load (μA)	167	166

The Thevenin-equivalent circuit uses a voltage source with the S.C. voltage ($V_{th} = 2.48V$) and $R_{th} = V_{th}/I_{th} = 4.98k\Omega$.

5 AC voltage divider

Using the same circuit setup as the previous section, except that V_{in} is now an AC source with $V_{in,pp} = 10V$, gives almost exactly the same voltage values as the previous example for peak-to-peak voltages. The difference is that it is not straightforward to measure (AC) current values, so we cannot use this to calculate the Thevenin impedance.

ECE393 – Lab 3

Jonathan Lam

December 23, 2020

1 Project summary

This lab follows Lab 2-4: “Low-pass filter” in the *Student Manual for the Art of Electronics*. This lab involves the analysis of a simple first-order RC LPF, such as its 3dB point and asymptotic behavior. This involves a theoretical derivation of the gain (magnitude and phase) characteristics, followed by a LTSpice simulation and a physical implementation to empirically demonstrate the physical results. The theory closely matches the experimental results.

2 Setup

We have the circuit in Figure 1, with $R = 15\text{k}\Omega$ and $C = 0.01\mu\text{F}$. Using

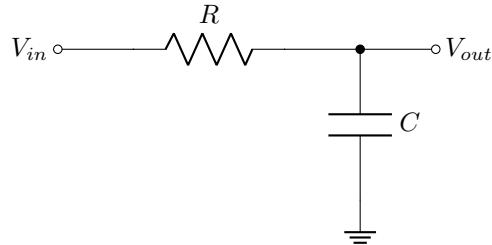


Figure 1: Generic RC LPF circuit

the frequency-domain representation of the circuit, we can make the capacitor a linear resistor with complex impedance $\frac{1}{j\omega C}$. Then, the circuit becomes a voltage divider, as shown in Figure 2. The gain of the voltage divider is the

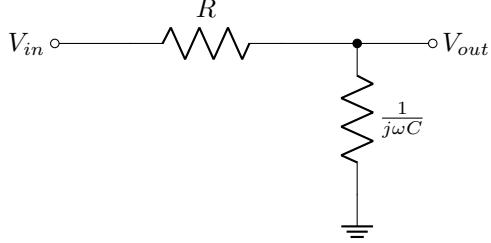


Figure 2: RC LPF becomes voltage divider with complex impedances

same as any voltage divider:

$$A_V = \frac{\frac{1}{j\omega C}}{R + \frac{1}{j\omega C}} = \frac{1}{1 + j\omega RC}$$

$$|A_V| = \left| \frac{1}{1 + j\omega RC} \right| = \frac{|1|}{|1 + j\omega RC|} = \frac{1}{\sqrt{1^2 - j^2\omega^2 R^2 C^2}} = \frac{1}{\sqrt{1 + \omega^2 R^2 C^2}}$$

$$\angle A_V = \angle(1) - \angle(1 + j\omega RC) = 0 - \arctan\left(\frac{\omega RC}{1}\right) = -\arctan\omega RC$$

From these derivations, it is clear that the magnitude of the gain is upper limited at one (and this is its value as $\omega = 0$). We also note that the phase shift is between 0 (at $\omega = 0$) and $-\pi/2$ (as $\omega \rightarrow \infty$), which confirms our understanding that resistors do not alter phase and capacitors perform a negative 90-degree phase shift.

3 3dB point

The 3dB point is when the power gain is $\frac{1}{2}$, or the voltage gain is $\frac{1}{\sqrt{2}}$. Thus:

$$|A_V| = \frac{1}{\sqrt{2}} = \frac{1}{\sqrt{1 + \omega_{3dB}^2 R^2 C^2}}$$

$$\Rightarrow 2 = 1 + \omega_{3dB}^2 R^2 C^2$$

$$\Rightarrow \omega_{3dB} = \frac{1}{RC}$$

In this case, $\omega_{3dB} = ((15k\Omega)(0.01\mu F))^{-1} = 6.67 \times 10^3 \text{ rad/s}$, or $f_{3dB} = 1.06 \times 10^3 \text{ kHz}$. At this frequency, the phase shift is:

$$\angle A_V = -\arctan\left(\frac{1}{RC}\right) = -\arctan 1 = -\frac{\pi}{4}$$

4 6dB per decade

From the above derivation, we see that $\omega^2 R^2 C^2 = 1$ at the 3dB point. At frequencies (much) smaller than the 3dB point, then $\omega^2 R^2 C^2 \ll 1$ and $|A_V| \approx$

1; at frequencies (much) larger than the 3dB point, then $\omega^2 R^2 C^2 \gg 1$, and $|A_V| \approx (\omega RC)^{-1}$. In this case, doubling ω halves the gain (a 6dB attenuation). E.g., if we look at ten and twenty times the the 3dB point, we should see half the gain in the latter.

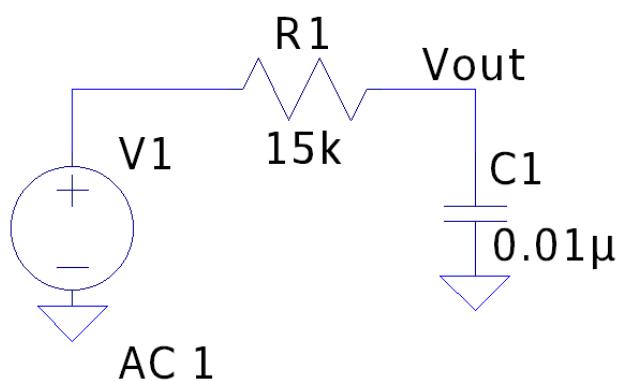
5 Integrator vs. LPF

Is the integrator a special case of a low-pass filter, or is the LPF a special case of the integrator?

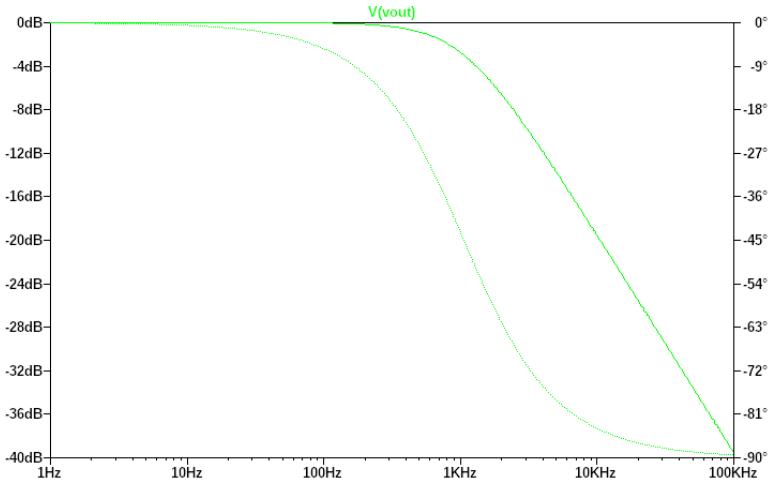
The LPF is a specific case (i.e., specific application) of integrators. Using the geometric interpretation of integration as the (signed) area under a curve, a high-frequency signal has a very small amplitude under the wave because it quickly alternates, while a low-frequency signal fluctuates less slowly and can have a larger area under the curve before the signal changes sign again. Thus the integrator is functionally a LPF – it is not an ideal LPF (i.e., it has that logarithmic “roll-off” of 6dB per decade), but it attenuates higher frequencies much more than lower frequencies.

6 LTSpice Simulations

We can simulate this circuit in LTSpice (Figure 3a) and perform an AC sweep on the gain (Figure 3b). The upper line in Figure 3b is the magnitude of the gain, and the lower line is the phase. As expected, the phase goes from 0 (when $\omega = 0$) to $-\pi/2$ radians (as ω becomes large).



(a) LPF schematic in LTSpice



(b) Voltage gain magnitude and phase plotted as a function of a (logarithmic) frequency sweep

Figure 3: LTSpice simulation

If we zoom in on the 3dB (more close to 3.01dB) point, we can see that it occurs at 1.06kHz and $-\pi/4$ phase shift, as previously calculated.

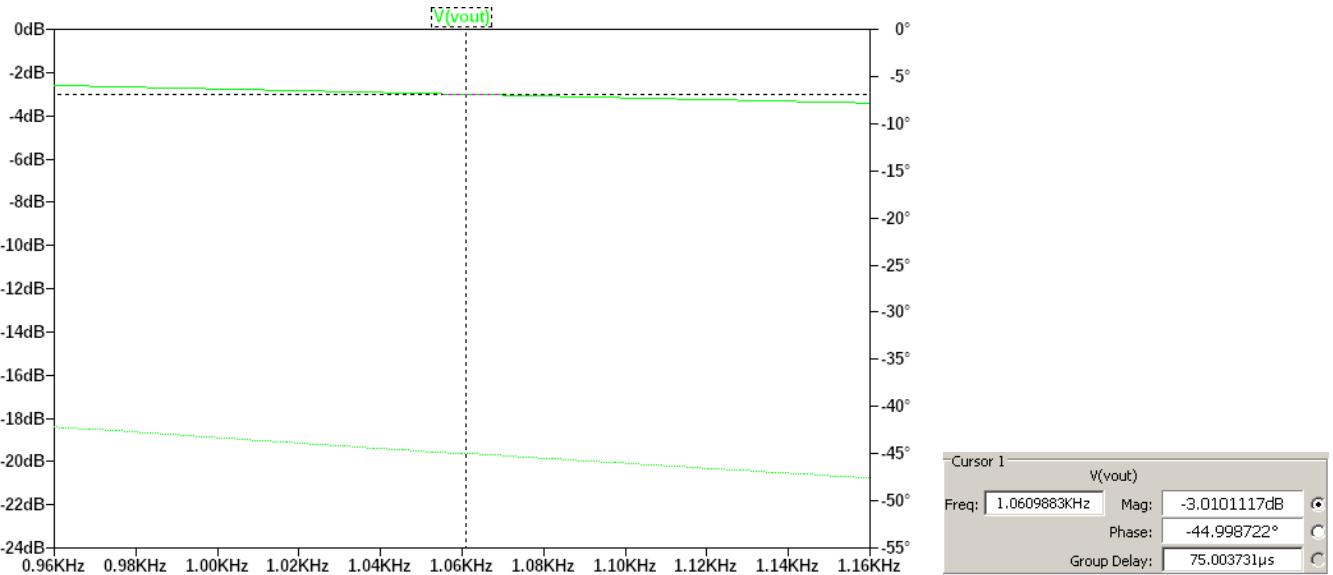
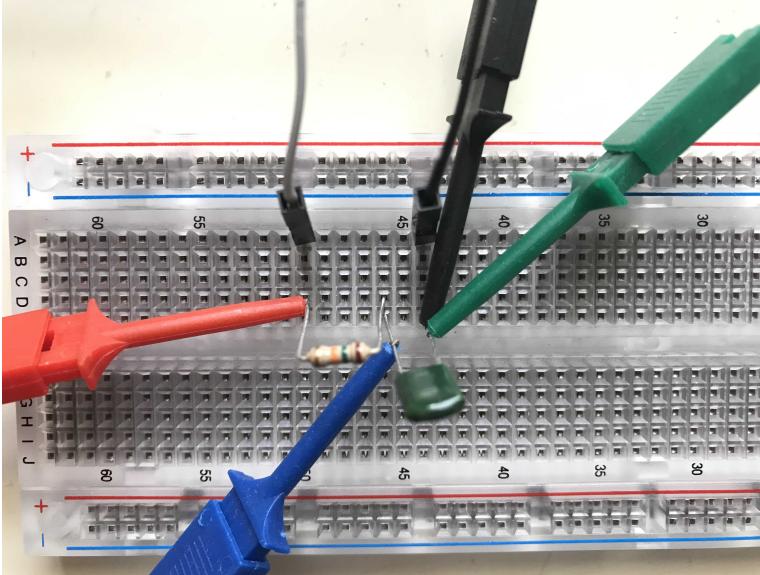


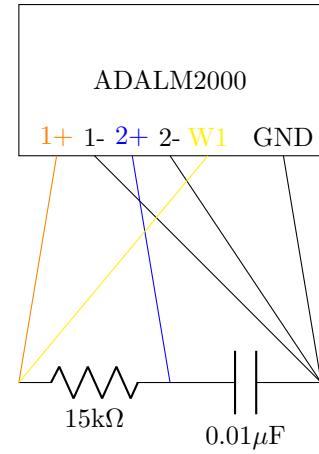
Figure 4: Close-up of 3dB point

7 Experimental results

The circuit was implemented on a breadboard and the signal generator and oscilloscope functions on the ADALM2000. The Scopy program was used to view the results on a computer.



(a) Photo of realization



(b) Block diagram showing connections with ADALM2000

Figure 5: Realization of circuit. The gray and black jumpers are connected to the waveform generator (ADALM W1 and GND). The red and black probes (across the signal generator, V_{in}) are for scope 1 (ADALM 1+, 1- pins); the blue and green probes (across the capacitor, V_{out}) are for scope 2 (ADALM 2+, 2- pins).

The 3dB point was found experimentally by driving a 1V amplitude sinusoid at a frequency such that the amplitude of the voltage over the capacitor is roughly 0.707V (3dB). For this setup, this was roughly 1.00kHz, which is not far from the theoretical 1.06kHz – this is reasonable given the tolerance of the resistor, capacitor, and ADALM2000. The phase shift can be calculated by $2\pi\Delta t/T = 2\pi(-1.22 \times 10^{-4}s)/(1.00 \times 10^{-3}s) = -0.244\pi$ (a negative phase shift indicates a delay, as we see in the figure), which is close to the theoretical $-\pi/4$.

When we plot this against a larger range of frequencies, as in Figure 7, the results match the theoretical results very closely. (The experimental and theoretical trends are largely indistinguishable for lower frequencies and diverge a little at the higher frequencies, but this divergence is likely due to instrumental error due to the nature of the high frequencies and the small amplitude of the



Figure 6: A plot of the experimentally-determined 3dB point on the Scopy oscilloscope. This is being driven by a 1.00kHz sine wave. The plot has $100\mu\text{s}/\text{div}$. on the x-axis, $250\text{mV}/\text{div}$. on the y-axis. The horizontal cursors are placed at $t = 0\mu\text{s}$ and $t = -122\mu\text{s}$ for the zero-crossing points of V_{out} and V_{in} , respectively ; the vertical cursors are placed at $V = 0.704\text{V}$ and $V = 1.00\text{V}$, for the amplitudes of V_{out} and V_{in} , respectively.

attenuated signal.) We also see the asymptotic behavior as $\omega \rightarrow \infty$; that is, the 6dB/octave loss from 10.0kHz to 20.0kHz (or between 100.0kHz and 200.0kHz). This is equivalent to saying that doubling the frequency halves the gain; this is clear on the plot given the slope of negative one (on a log-log scale) for frequencies much higher than the 3dB frequency.

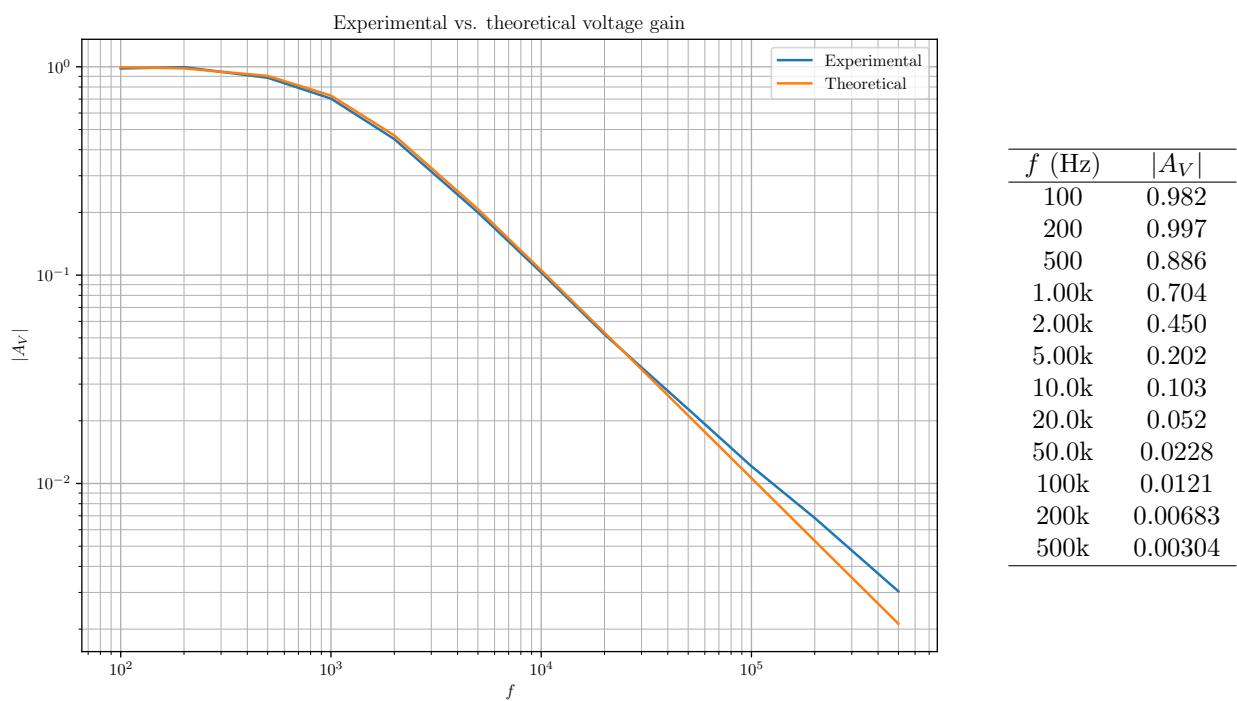


Figure 7: Experimental vs. theoretical gain results plotted on a log scale

ECE393 – Quiz 1

Jonathan Lam

October 8, 2020

Goal

Make a DTMF decoder in LTSpice. It should take a .wav file as an input, and you should be able to be able to clearly differentiate between different tones based on some visible (or audible) output.

See Figures 1 and 2, on the following pages, for the schematic and the output plots.

Implementation details

- The circuit is 8 bandpass filters in parallel, one for each of the DTMF frequencies.
- Each bandpass filter is a LC tank (I tried using RC filters, but this was much more unreliable, and LC tanks work pretty well because they resonate at a single tone, which is what we need here). For simplicity, I set all of the capacitor values to the same value (1 mF), and calculated the required inductor values using the formula

$$2\pi f = \frac{1}{\sqrt{LC}}$$

where f is the set of DTMF tones (i.e., 697Hz, 770Hz, 852Hz, 941Hz for the low group, and 1209Hz, 1336Hz, 1477Hz, and 1633Hz for the high group).

- To read the DTMF values, I overlaid the low group values (LG1-4) on one plot plane, and the high group values (HG1-4) on another plane and read the DTMF values by looking at which LC tank resonated for each tone.

In the case of the .wav file whose plot is shown in Figure 2, the determined DTMF sequence is:

Low group tone	High group tone	Decoded symbol
4	1	*
4	3	#
3	2	8
2	1	4
1	3	3

V_{in}
 V_1
 wavefile=C:\Users\jon\Downloads\DTMFgnep.wav
 $.tran 4s$

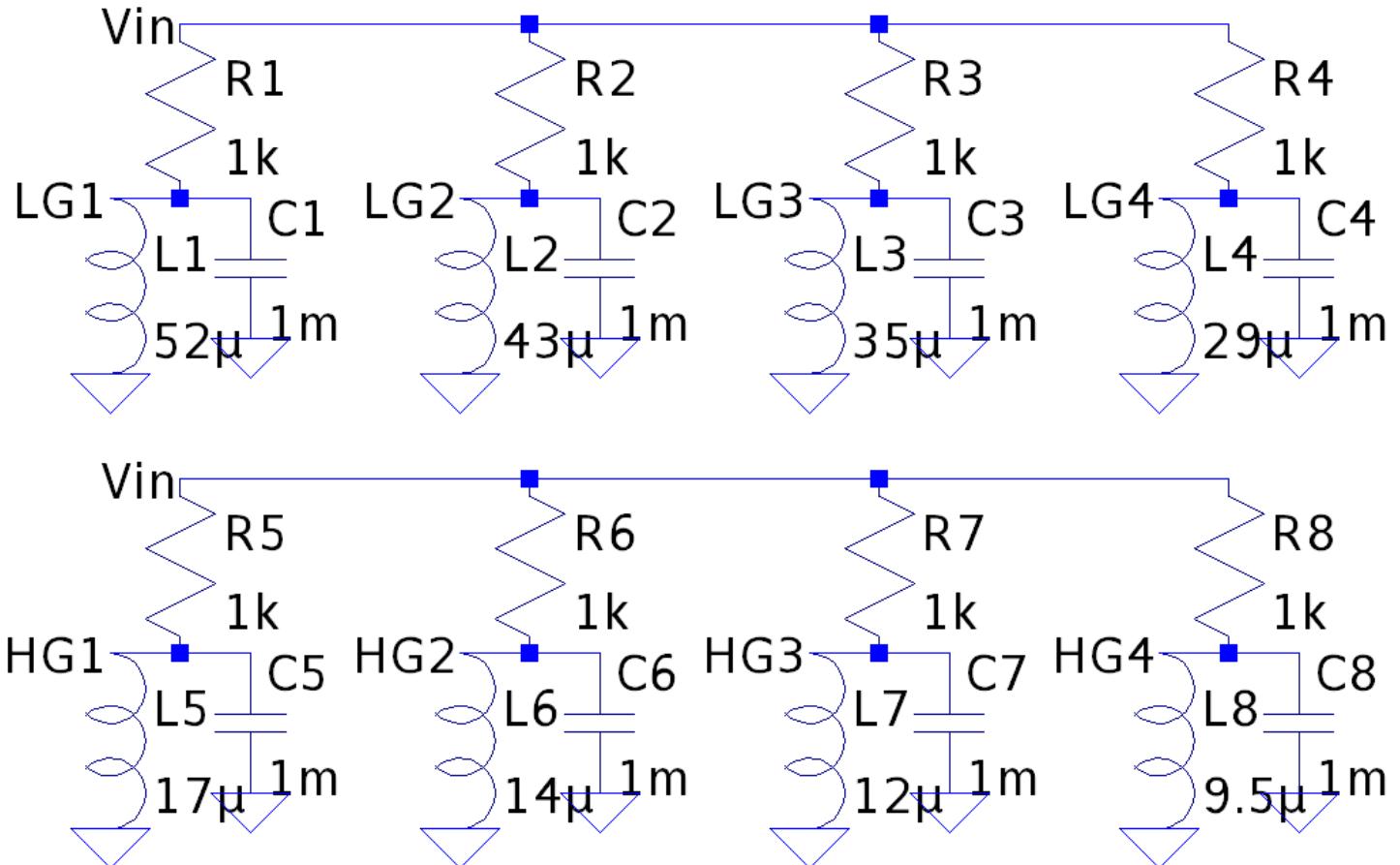


Figure 1: Schematic

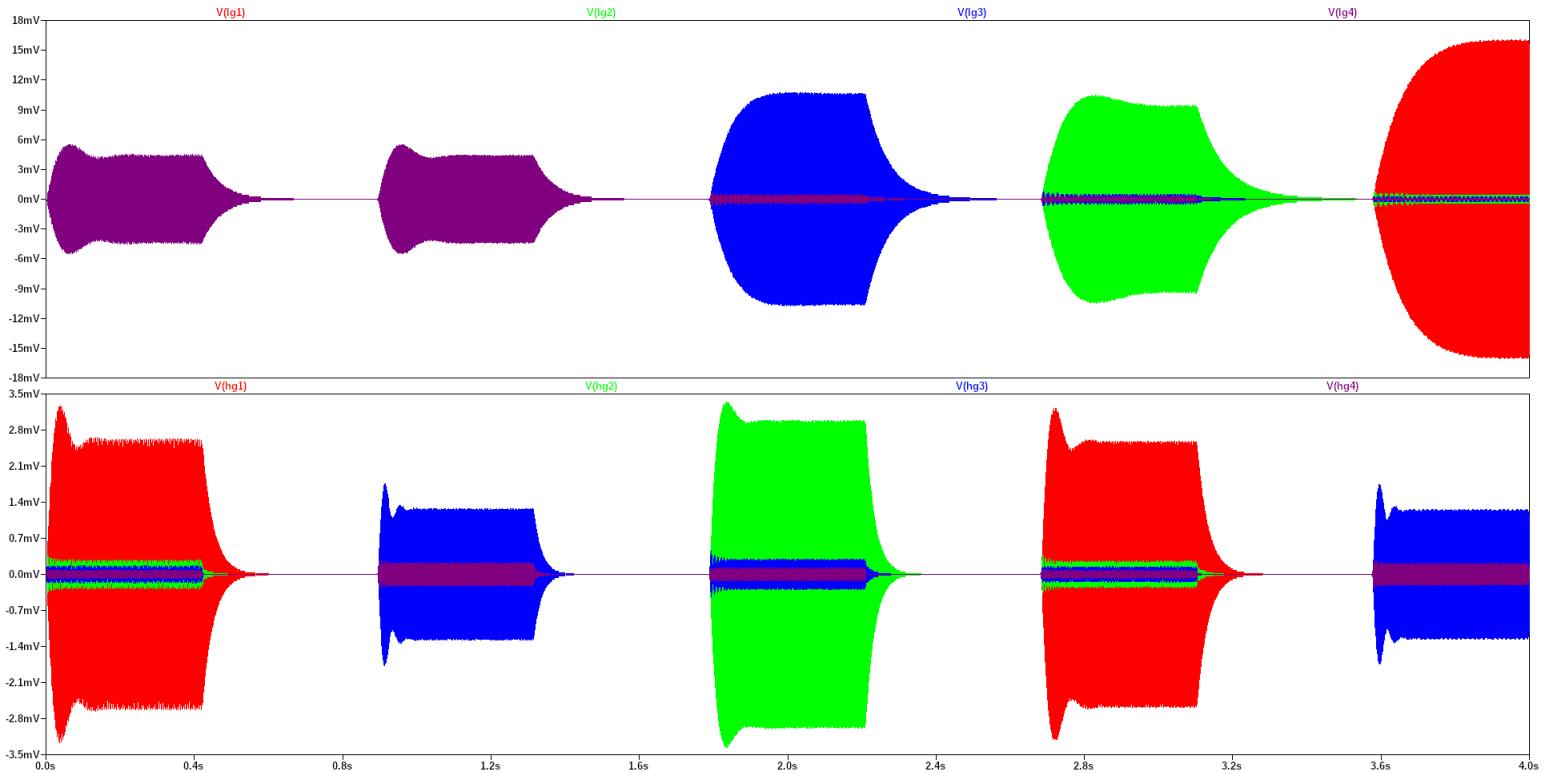


Figure 2: Top plot shows low-group tones; bottom plot shows high-group tones. In order from lowest to highest frequency: red, green, blue, purple. Thus the low group tones are [4, 4, 3, 2, 1], and the high group tones are [1, 3, 2, 1, 3].

Artificial Intelligence
Fall 2020
Project #1
Grading Sheet

Name: Jonathan Lam

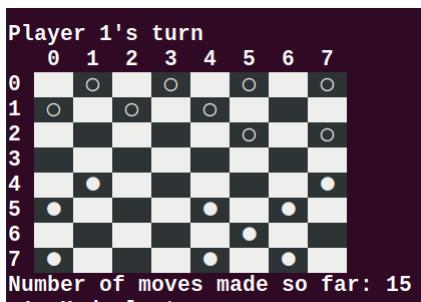
Email: lam12@cooper.edu

Grade: 97

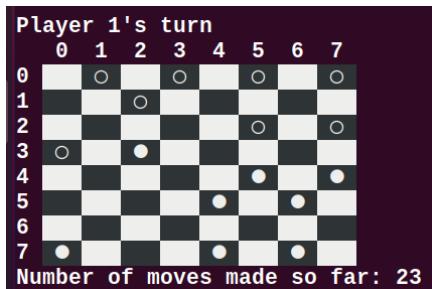
Notes:

You have implemented Checkers. You used C++ first, then ported it to Scheme. As discussed in an email chain, I will at least start off grading the C++ version, because it tends to search a bit deeper, but I'll try out the Scheme version at the end. I am using an Ubuntu 20.04 virtual machine with 2 GB of RAM on my home desktop.

I am playing my first game moving first, as black, giving the program (which moves second) 5 seconds per move. The interface is black and white but looks quite good. The early moves by the program seem fine. It is reporting searches to depth 10 early. There were some early trades. When the program has a forced move, it makes it right away. The program did have to abandon one square of its back row to avoid losing material, which may put me at a slight positional advantage. Here is the early situation on one of my turns:



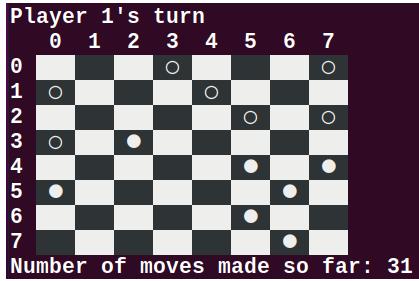
A bit later, I forced a trade that may put me in a position where I will eventually gain material; here is the updated board on my move:



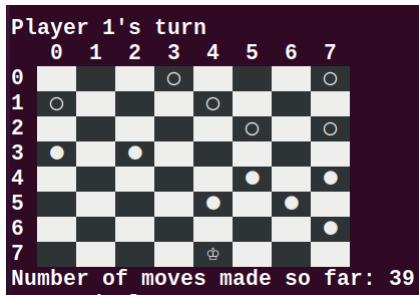
That piece at 3,2 is possibly too far forward; I will start to attack it (starting with 0,5->1,4) and possibly pull ahead. When the trading started, the computer was searching to depth 9, which was

not far enough to see that I will eventually gain material (if I am correct that I will, which is far from certain); so, even if I manage to pull ahead, this is not due to a bug.

Nope. The program manages to defend, and we are still even. Here is the situation later:

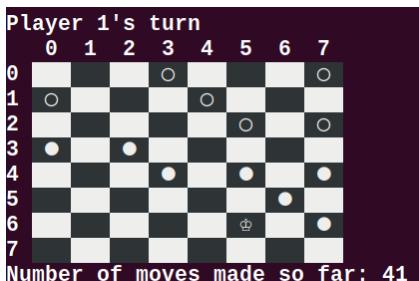


I think about 0,3->1,2 to attach more, but after 6,5->5,4, I don't think I have a path to gain material, and my back row would be exposed. So instead, I will play 3,0 -> 4,1 and work toward getting a king. (Another option was 2,5->3,6 to force a two-for-two trade.) I go on to get the first king. (Cool little character icon that looks like a crown!) Here is the updated board:



The program just searched to depth 11; very good for the middle game! We'll see if my king is enough of an advantage for me to win, as I start chasing its pieces.

I don't know if I like its next move:



Now, after 2,5->3,6, it will have to jump me and I will get a double jump in return, plus have a path to another king. However, looking at its options, none were good. I should go on to win here if I don't make a mistake.

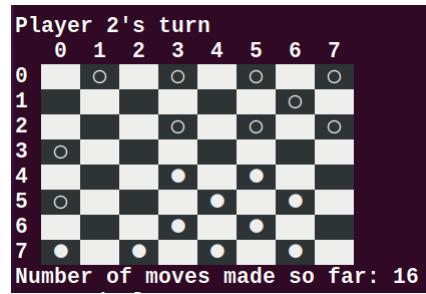
Soon, I am ahead five pieces to three; I have three kings, the program has two. I'll win unless I make a major blunder, but let's see how hard the program makes it for me!

The program is playing fine, putting off the inevitable. I have been careful, and I managed to force a couple of trades. Now I am ahead three pieces to one, and I have two kings (and can get a third easily). The program moved its king to a double corner, so I will have to force it out.

Near the end game, the program starts to make its move very fast. Interestingly, the depth search is 7, which I don't think is to the end of the tree, but perhaps the program sees that it will definitely lose somehow (maybe with my extra piece, there is a way to win quicker than I see). In any case, the end of the game goes quickly. The program puts off its loss as long as it can. The program exits gracefully after the loss.

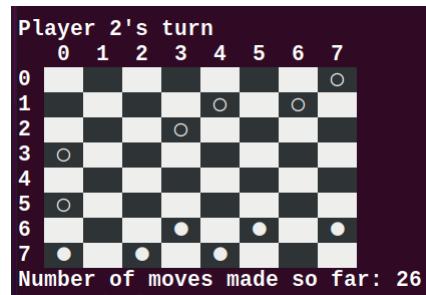
I start a second game as "red" (white on the board), with the computer as black, again giving it 5 seconds per move. It's playing fine. It is searching to depths around 10.

I may notice one minor flaw (not bug, but something non-ideal) with its heuristic. Look at this situation:



It just moved from 1,4 to 2,3. Now, I will move from 4,2 to 3,4; it will jump, I will double jump, it will jump. In the trade, it will abandon one of its back-row squares. I looked back at your writeup, and you do say it values keeping the back row protected; but maybe that weight should be a bit higher. Last game, it also moved one of those pieces early. That may be why I was eventually able to get a king, and ultimately won that game.

Here is the situation a bit later:



As you can see, my back row is much better protected than the programs. Whether that will be enough for me to win (or even avoid losing), I don't know.

Here is the situation a bit later:

Player 2's turn							
0	1	2	3	4	5	6	7
0							○
1							
2			●				
3	○			○		○	
4			○				
5	○					●	
6			●				●
7	●			●			

Number of moves made so far: 38

I have the clearest path to a king, of course, but this is confusing. It may still be a theoretical draw. It can go from 3,6 to 4,5; then 4,3 to 5,4. Anyway, I'll just walk ahead for now and get my king. Hopefully (for me) getting it one or two moves ahead gives me some leeway.

Things play out basically like I expected. Here is the updated board:

Player 2's turn							
0	1	2	3	4	5	6	7
0							○
1							
2							
3	○			○			
4				○			
5	○		●			●	
6				○			●
7	●			●			

Number of moves made so far: 44

I will start utilizing my king. After it gets its king at 7,6, I can move from 5,6 to 4,7 so the program's king will be trapped.

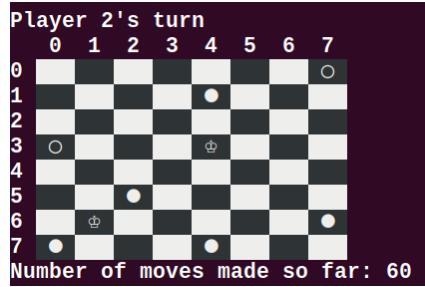
Just as I felt I might be headed to another win, I really like the program's latest move:

Player 2's turn							
0	1	2	3	4	5	6	7
0							○
1							
2							
3	○						
4			●		○		
5			●				●
6				●			●
7	●			●			●

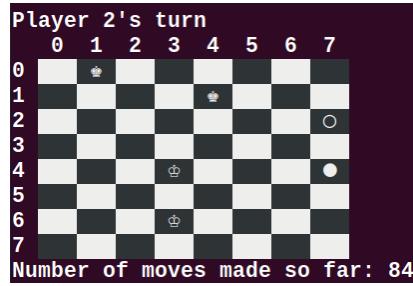
Number of moves made so far: 52

Note that I was temporarily ahead a piece. However, it just moved 4,5 to 5,4; this forces a trade, but it will get my king for a regular piece. Although I will still be temporarily ahead one piece, it will have two kings, I will have none, and its kings will no longer be trapped. If anything, I think a loss for me is more likely than a win now. We will see!

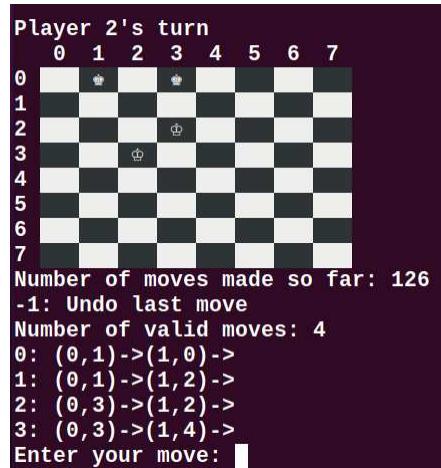
Here's the board a bit later:



I could just get the king at 3,0, and then after the program jumps me, move my king to 1,2 to it won't become trapped. That would probably be an even game, where we each have four pieces, although I would be down one king without much flexibility. I think better for me is to move 5,2 to 4,3 forcing a jump; then 7,4 to 6,3 forcing another jump (both of regular pieces); then I capture its king at 6,1. I think this gives me clear paths to get two kings (if I am not missing something). This has some risk (I cannot visualize future boards), but if I am processing it correctly, it will be a more clearly even game. This is what I will try. I do, and things play out as I hope. This is a clearly even end game position (but I'll play it out a bit to see what happens):



Eventually, I force another trade, and now we each have two kings. One of my kings is in a double corner. I am just moving that king back and forth; the program is also moving one of its kings back and forth. I declare this game a draw! Here is the final situation:

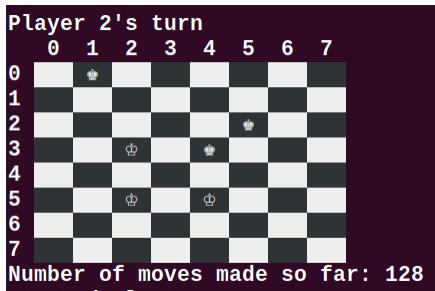


Since I haven't tried your undo move feature yet, I will test it now, just to see that it works. I type -1 a bunch of times (I didn't realize you can do it more than once). It works great! I see the game unplay a bunch of moves, then try moving forward again and it works. Nice feature!

Next, I try loading one of my boards with many, crazy multiple jumps available. (I see you already have them available for me in your "states" folder.) It loads fine, and all the legal moves are found. (I try this in person vs. person mode, which works fine, I can make moves for both players.)

Next, I load a board where I have one king in a double corner, and the program has two kings that start far away. The program marches its kings toward me, forces me out of the double corner (probably as quickly as it could), and goes on to win the game. The program handles the situation perfectly.

Next, I watch the program play a game against itself, giving it three seconds per move. The feature works fine. Both sides seem to be playing well. I can't really follow all the logic at this speed, but I notice no bad moves. Both sides get the first king at close to the same time. At first, I think the player that got the king a few moves ahead may be able to use it to pull ahead material, but the other side found a tactic to keep the material even. Now, both sides have three pieces, all kings. This should be a draw, but I'll let it play a bit longer. I let it play a bit longer, and I think there is some repetition now. I declare this game of computer vs. computer a draw. Here is the final situation where I end it:



I will try the Scheme version now! I install chezscheme. I don't think I need to change the permissions (as in your report); the play-game.ss file is already an executable. I try this command, as indicated in the report: ./play -game .ss

I get an error message from your program: "Exception: incorrect number of arguments to #<procedure char-ready?>"

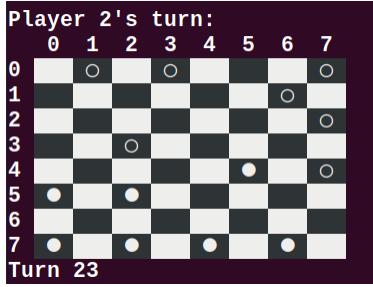
I try it in interactive mode, and the same thing happens.

Ah, I see you addressed this in your email. I applied the fix that you sent me (one line changes in main.ss). Now, the game runs fine. I am starting a game moving second (as red/white), giving the program five seconds per move. The program is searching to depth 9 early (one less than when I started games in C++). I see you added two extra choices (quit, or use iterative deepening to find the best move), as mentioned in the writeup.

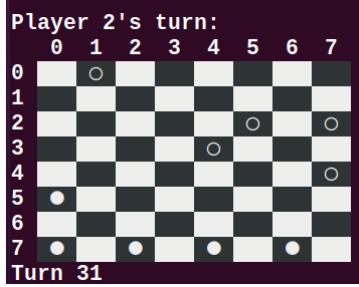
One thing I like about the Scheme version is that the program indicates its chosen move (which didn't happen in the C++ version); for example: "Best move: -1: (1,6)->(2,7)". I don't know why it shows -1, but the move is clearly indicated. (With the C++ version, at times I would scroll up

to look at the previous board to see what move was made. Of course, this is not a big deal, but just one nice difference I noticed in the interface.)

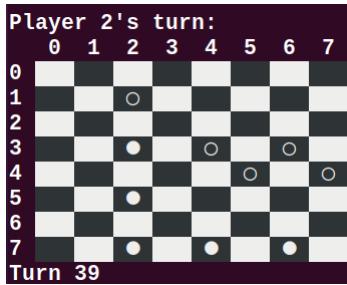
As we enter the middle game, the program is searching to depth 8. Again, this is less than the C++ version, but still clearly enough to play well. We've done a bunch of trades, and we are even in the middle game:



Here is the situation a bit later:

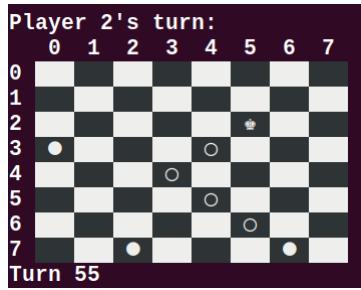


Note that my back row is still perfectly protected (although I will be forced to abandon one of those squares soon); the program's is mostly exposed. Although I have no clear path to a king (when I start matching 5,0 forward, it will likely use the piece at 0,1 to block), I think I have enough of an advantage here that I should get a king first. I don't know if I will win, but we'll see! Here is the situation a bit later:



After I move from 3,2 to 2,3, I will get the first king and have a clear advantage (I think!).

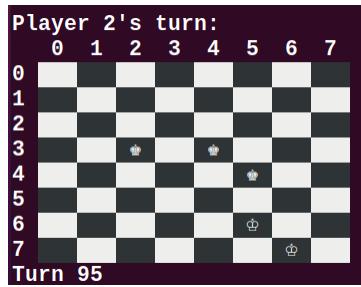
The program hangs on for a while without losing a piece, but now we are here:



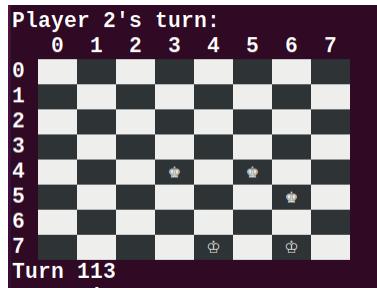
Now, I will move from 7,2 to 6,3, forcing a jump, and getting a double jump in return.

The program recently searched to depth 11, by the way; about as good as the C++ version!

Soon, I am ahead three pieces to two. The program has two kings, and I only have one, but I have clear paths to convert the other pieces. Now, I am ahead three kings to two kings. One of the programs' kings is in a double corner. (If both of the program's kings were in the two opposite double corners, that is a theoretical win for me, but tough.) The program now maneuvers its second king to be by the same double corner as the first game. This is not as tough as when they are in opposite double corners, but still kind of tough! Here is the current situation on my move:



Of course, I am trying to force a trade (since two kings vs. one is an easy win for me); the program is doing a good job resisting. Here is the updated board:



Finally, I can sneak into the double corner. This still may not be easy though.

A bit later, the program actually forces a trade. I will win easily now. My guess is that within its depth search, it saw that a trade was inevitable (although I had not yet seen how to force it)!

The program marches its king into the opposite double corner. It stays there until I force it out. I inevitably win, but the program pushes it off as long as possible.

I have enough to grade. This is an excellent project. All features have been implemented, plus some additional features (such as undo). The interface is nice. The program makes no clearly bad moves, and it finds good moves when available. It recognizes complex, multiple jumps. You also impressively implemented the game in both C++ and Scheme! The Scheme version searches almost as far with a time limit as the C++ version (I'd say typically, one depth less). Now, I played three games, and I won two with one draw. One of the wins was against the Scheme version. I think the main reason ai won the two games is that the program does not value protecting the back row strongly enough. In both games that I won, I made no mistakes (playing better than I usually play, I think), and got the first king; then I was able to convert that to a gain of a piece. Still, the program plays well. It also knows how to win when it is ahead two kings to one with the opponent's king in a double corner, and it does so quickly. Despite winning two games out of three, I think this program deserves an A+, or at least a borderline A/A+. It would be a clear A+ if it played a bit stronger, to avoid losing. I think with a tweak in the heuristic, it would be very hard to beat. I will count this as a 97. It's a great project.

ECE469 – Checkers AI

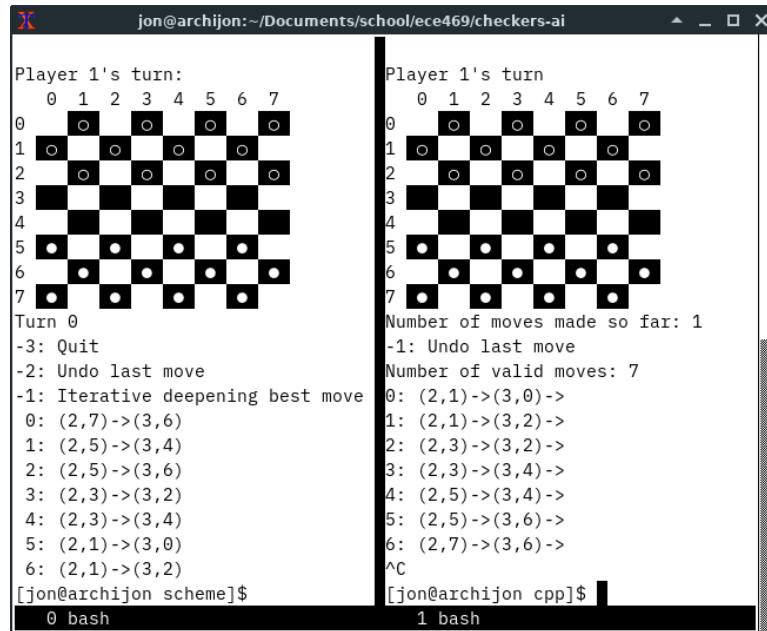
Jonathan Lam

November 5, 2020

Since the assignment is fairly well-defined, this “report” will mostly be commentary on non-normative details. The relevant source code and toy states are located at <https://github.com/jlam5555/checkers-ai>.

1 Ruleset

- Ruleset from the Standard Laws of Checkers [1]. Jumps must be taken. Black moves first, white second. (In the C++ code I call the second player RED, but this naming difference is inconsequential.)



1 bash

2 Dependencies, compilation, and execution

The development environment was Arch Linux with g++ 10.2.0, Chez Scheme version 9.5.4. The game is played in a standard ANSI-compliant terminal with a monospace font.

2.1 Dependencies

The C++ version requires any modern version of g++. (clang/LLVM was not tested.)

The Scheme version requires that Chez Scheme is installed and the program `scheme` is in your path. Depending on the installation of Scheme, you may have to symlink the default scheme executable; e.g., if `scheme` is installed at `/usr/bin/chezscheme9.5`, then you will have to symlink it to the scheme executable (`ln -sf /usr/bin/chezscheme9.5 /usr/bin/scheme`) or change the interpreter on the shebang of `play-game.ss`.

2.2 Compiling and running the program

There's not really a need for a Makefile in either case. Navigate to the respective source directories and run:

2.2.1 C++

```
$ g++ -o checkers checkers.cpp
$ ./checkers
```

2.2.2 Scheme

```
$ chmod +x play-game.ss
$ ./play-game.ss
```

2.2.3 Scheme (REPL)

The Scheme version may also be played interactively through the REPL. Example commands are:

```
$ scheme                      # enter REPL
> (load "main.ss")           ; load game files

> (play-game)                ; start interactive game

> (define state              ; manually load state
  (car
   (load-state-file
    "../states/s1.txt")))

> (print-board state)        ; print board of current state

> (valid-moves state)       ; get valid-moves for state
```

```

> (list-ref
  (valid-moves state) 3) ; print board after
                           ; applying fourth valid move

> (print-move "First\u2022move"
  (car (valid-moves state))) ; print the first valid move
                             ; (car gets first move)

> (define best-move
  (iterative-deepening-search
    state 2)) ; get best move from iterative
               ; deepening search with a
               ; two-second timeout

> (print-move "Best\u2022move"
  best-move) ; print best move

> (define state
  (move-state best-move)) ; apply best move to state

> (define state
  (move-state
    (list-ref
      (valid-moves state)
      3))) ; apply the fourth valid move
             ; to state

```

3 Notes on the TUI

- The board comprises an 8×8 black-and-white board. The checkers can only fall on the black squares. Black (player 1) starts on the top and its men move downward; white (player 2) starts on the bottom and its men move upward.
- Black (player 1) pieces are represented by non-solid circles and kings, and white (player 2) pieces are represented by solid circles and kings.
- The possible moves are listed and numbered. The coordinates are in (row, column) order; both row and column are zero-indexed.
- There is not very strict user input validation and error checking, but most inputs should behave as expected (e.g., throw an error if the input state file is not found). When prompting for a move, the program will keep re-prompting until a valid move number is entered.
- Ctrl+C should be sufficient to exit the game at any point. (If running in the Scheme REPL, you may need to press Ctrl+C multiple times, and this will enter an interrupt handler; exit this by entering “q”.)

4 Main algorithm implementation details

Representation of states and moves Since there are 32 playable squares on a checkers board, we can represent the positions of the black checkers and of the white checkers (regardless of

their type) as two 32-bit bitmasks. The player's pieces are also stored in two arrays of length 12 (maximum of twelve checkers per player), where each element includes the checker type and coordinates (these values are packed into one byte). The former representation allows for constant-time lookup of a board position, and the latter allows for efficient iteration over the players' checkers. Each state object contains these two fields (boards, 8 bytes; pieces, 24 bytes) as well as which player has the next move.

Generating new moves The valid-moves function first checks for capture moves, and if there are none, then checks for non-capture moves. Capture moves are implemented by using DFS with a stack of intermediate moves.

Minimax search with alpha-beta pruning This was taken almost directly from the textbook. The min-value function only returns the best value, while the max-value returns both the best value and the best move; otherwise they are the same. (I didn't implement negamax.)

Time-based iterative deepening At the top of the max-value and min-value functions, there is a check for whether the time limit will be reached within 10 milliseconds. In the C++ version, a special value is passed up the stack; in the Scheme version, a continuation directly passes control back to the iterative-deepening driver function.

Heuristic function The score is calculated as follows (in order of importance):

- king and men count** number of kings and men, kings are worth more than men
- men distance to king** closeness of men to being promoted to kings
- back row** how many men are in the back row; higher is generally better as it doesn't allow enemies to get promoted as easily
- trade affinity** if winning, try to maximize the ratio of the number of checkers remaining
- sum of distances to corners** only used in endgame position, used to chase pieces out of double corners
- board center** how close the pieces are to the center, where there is more freedom and less of a chance to get trapped
- randomization** used to easily break ties and make the game nondeterministic

This is a simple heuristic that does fairly well, but it may not see far enough in certain endgame positions to make a decisive win when it might be clear to the human eye. For example, in some cases where a player has a three-to-two piece advantage, it may not efficiently dispatch the pieces to capture the losing side's pieces one by one.

Early stopping There are two conditions when the iterative deepening algorithm stops before the time limit is reached. Namely, this is if there is only one valid move, or if the entire game tree is searched (i.e., a definite win or definite loss is determined). From a practical view, it may be better to stall even if a definite loss is determined so that the match lasts longer, but this implementation does not prolong the inevitable.

5 Extra features

- The way I implemented the game, it was very easy to implement an undo function. In short, a list of the past game states is stored as the game progresses, and it is easy to revert to a prior state by setting the current state to the stored past state. This was implemented in both versions of the game.
- In the Scheme version, I also added the option to let iterative deepening choose the next move instead of manually choosing a move, as well as an option to quit the game.

6 General comments

- The C++ version was written first, and the logic was translated directly into the Lisp version. It took roughly two days to plan how to generate the moves (and the relevant data structures), two days to implement the main algorithm in C++, three days to research and design a simple but satisfactory heuristic, and another three days to rewrite the main program in Scheme. Then I spent about a week on general touch-ups.
- I realized I could make some additional optimizations when doing the Scheme version (e.g., a 32-bit bitmask suffices to store the board state, 64-bits is not necessary; and only one bitmask is needed for each player, I used two in the C++ implementation).
- In the C++ version, I was worried about saving space at first (originally I had the idea that we were to save the entire tree in memory, which I realized was false after starting to implement the project), so there are a lot of 8-bit integral data types thrown around. In Lisp, everything is of the fixnum type (60-bit integral type).
- I tried to avoid dynamic memory allocation completely in C++; everything is allocated on the stack (i.e., in small fixed-size buffers). The buffer lengths were somewhat arbitrarily chosen so that they weren't too large but were large enough for any reasonable gameplay.
- In Scheme, there is a lot less control over memory allocation, so I used the defaults for memory allocation. I tried to stick to immutable values to follow the functional paradigm of Lisp, but I broke this when modifying the bytevector buffers (board bitmasks) in place when necessary. I also used linked lists for storing the list of available moves rather than an array buffer. I'm not sure how costly Lisp's memory allocation is; I believe it uses slab allocation to improve performance, but it is likely a major reason that the Scheme version is slower than the C++ version.
- In general, the C++ code felt a lot more terse and easy to write. However, one thing that was really nice about the Lisp implementation was the use of (first-class) continuations; this allows for a really clean and easy way to return control to the iterative deepening function from anywhere in the minimax search when time runs out. In the C++ version, you either have to traverse all the way up the call stack or do some unfavorable signaling or cross-function jumps (e.g., setjump/longjump).
- The colors were set using ANSI escape sequences, and the printed board pieces are unicode. Make sure to use a terminal emulator that supports these ANSI escapes (most *nix terminals do) and uses a monospace font.

- From my very-informal tests, the C++ version usually gets to the same depth or outperforms the Scheme version by fewer than three extra depths. I also loosely benchmarked the valid-moves function on some sample states, and found that the C++ version generates nodes roughly four times as quickly, which could correspond to no or one extra search depths.
- On all of the toy states (the provided states 1 through 10), both versions play well with only one second timeouts. In the case of an ordinary game of the AI playing against itself, it is not uncommon to see wins or losses on either side, nor is it uncommon to see draws.

References

[1] <http://www.chesslab.com/rules/CheckerComments3.html>

ECE469 – Project 2

Jonathan Lam

December 2, 2020

1 Problem description

1. Implement a fully-connected (FC) neural network architecture containing one hidden layer (with an arbitrary number of inputs, hidden nodes, and output nodes) for multiclass binary classification, where each node uses a sigmoid activation. Each output should be a binary classification using a mean squared-error (MSE) loss.
2. Allow the user to input the initial weights for the network, input a training dataset, and train a neural network initialized with those weights on the given dataset for a given number of epochs and a given learning rate. Allow the user to output the trained network weights to a file.
3. Allow the user to evaluate the dataset on a test dataset, and output the results (statistics) to a file.
4. Create a custom dataset and run steps 2. and 3. on it. I.e., find or create a binary multiclass classification dataset, generate the train and test dataset files for this dataset, decide on a network architecture (i.e., number of hidden nodes), generate the weights to initiate this architecture, train and evaluate the model, and output the statistics of the model on the test dataset.

2 File formats

These are delineated in the assignment file, but included here for completeness. For sake of reproducibility, all weights and feature values should be rounded to three decimal places (zero-padded if necessary), all other numbers (numbers of features, numbers of nodes, binary labels, etc.) should be integral values, all lines should not contain leading or trailing spaces, all values on a line should be delimited by a single space, and newlines should follow the Unix format. There are example files in the GitHub repository.

2.1 Train and test datasets

The first line should contain the following three values:

1. Number of samples
2. Feature count
3. Output count

The rest of the file will include sample information, with one sample per line. Each line should contain a list of all the features followed by a list of all the output labels (0 or 1), all delimited by spaces.

2.2 Network weights

This file assumes a fully-connected architecture. The first line should be a list of the widths (number of nodes) of each layer, including the “input layer” (whose width is the number of features). Thus, in a neural network with one hidden layer, this would be the number of input features, the number of hidden nodes, and the number of output nodes.

The rest of the file are the weights of the network. Each line contains the weights of one node, and the number of weights on each node’s line should be one more than the number of inputs to that node’s layer (the extra weight is the bias node). The first weight should be the node’s threshold weight (which gets multiplied by -1, i.e., the negative bias), and the rest of the weights should be ordered corresponding to the order of the inputs. The nodes should be grouped together by layer, and the layers should be ordered with the more shallow layer (i.e., closer to the input layer) on top of the deeper layer.

2.3 Evaluation statistics

	Expected = 1	Expected = 0
Predicted = 1	A	B
Predicted = 0	C	D

Table 1: Contingency table

Each line except for the last two lines are the statistics for a single binary class, with the classes ordered in the same way as they are in the dataset and weights files. Each line contains:

1. A (from contingency table)
2. B (from contingency table)
3. C (from contingency table)
4. D (from contingency table)
5. overall accuracy: $\frac{A+D}{A+B+C+D}$
6. precision: $\frac{A}{A+B}$
7. recall: $\frac{A}{A+C}$
8. F1 metric: $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

The final two lines contain the micro- and macro-averaged overall accuracy, precision, recall, and F1 metrics, respectively.

3 Description of implementation

3.1 Tech stack and performance considerations

The language of choice is Scheme, because Lisp is fun. Chez Scheme is chosen for the implementation. Performance was clearly not the goal here – everything is built using linked lists and intended to be as much declarative-style as possible (all elements are immutable, records are rebuilt on updates), there is no batch training, SGD is not utilized, sigmoid is used rather than ReLU, nothing is vectorized nor parallelized. This is fine for the toy datasets in use here and for understanding neural network architectures, and it also makes for a fun exercise in applying a lot of list transformations on a 3-D “tensor” (e.g., using `(apply map list M)` can be thought of as transposing a matrix `M`).

(Using Chez Scheme’s `--optimize-level 3` flag decreases runtime by about 10% by skipping runtime checks, for a tiny performance gain.)

3.2 Data structures

There are two major data structures: `layer` and `model`. These are defined as record-types (i.e., like C’s `struct`).

Each `layer` has three fields: `train`, `infer`, and `weights`. `train` and `infer` are procedures to train the network and to use the trained layer with its current weights (if any), respectively. `weights` stores the layer’s weights, if any. For this project, three different layer types are defined: `sigmoid-layer`, `dense-layer`, and `loss-layer`. Only the `dense-layer` has weights associated with it; the other two layers are essentially hardcoded functions (the sigmoid function and the MSE loss) that pass along gradients during backpropagation.

Each `model` has two fields: `layers` and `shape`. `layers` is a list of layers, in which the final layer is always a `loss-layer`. `shape` is a list of the widths of each layer, including the “input layer” – i.e., if the first line of the network weights file were “30 20 10” then `shape` would be (30 20 10). Note that this generalizes past networks with a single hidden layer; an arbitrary number of layers (with arbitrary layer types) can be supported.

(Also note that there is no “input layer,” which is only a useful abstraction for describing the architecture shape and redundant once we have already built the network.)

3.3 Training procedure

The training and backpropagation are handled recursively. Each layer’s `train` method should take as arguments the layer inputs (equal to the features for the first layer), the label for the current sample (only to be used in the loss layer), the learning rate (for updating weights), and the following layers. For all of the layers except the final layer (`loss-layer`) the following steps are performed:

1. (Forward pass) Call the `infer` method on this layer’s inputs to get this layer’s outputs.

2. Recursively call the next layer's `train` method using this layer's outputs as the next layer's inputs, and passing along the rest of the arguments. This will return the gradients of this layer's nodes, as well as the updated deeper layers.
3. (Backpropagation) Using the gradients of this layer's nodes (w.r.t. the loss) and this layer's weights, calculate the gradients of each of the input nodes w.r.t. the loss using chain rule.
4. (Backpropagation) Using the gradients of this layer's nodes (w.r.t. the loss) and the inputs to this layer, update the weights for this layer using the derivative of each weight w.r.t. this node and the chain rule in order to minimize the loss. (To practice immutability, a new layer is created with the new weights and prepended to the list of updated deeper layers.)
5. (Backpropagation) Return the gradients of the input layer's nodes and the updated network (this layer and all deeper layers).

The procedure for the final layer is a different. Using the estimated outputs and true output labels, simply computes the loss and the gradient of the loss w.r.t. the outputs of the network. (This can be thought of as the base case of the recursive procedure.)

To train a model on an example, all you need to do is call the `train` procedure of the first layer, which will initiate the recursive training of the entire network. This will return a network with all of the weights updated. The `model-train` procedure performs this on every sample of the training dataset, reporting the loss, and repeats this for the number of epochs.

3.4 Inference procedure

Evaluation of a model simply involves a folding operation over the `infer` methods of the layers of the model, where the initial value are the sample features. This is implemented with the `model-predict` procedure.

Variants are written to map this procedure over a set (i.e., a test dataset rather than a single test sample) (`model-predict-set`), and binary variants are written to output the rounded model estimates for single and sets of samples (`binary-model-predict`, `binary-model-predict-set`).

3.5 Evaluation procedure

Evaluation of a trained model occurs by performing inference on a test dataset and calculating the statistics as described in §2.3. The `model-evaluate` procedure takes as input a trained model, test feature set, and test labels, and returns a list of:

1. per-class contingency tables (i.e., A, B, C, D)
2. per-class statistics (i.e., overall accuracy, precision, recall, F1 metric)

3. micro-averaged statistics
4. macro-averaged statistics

3.6 File structure

`data/` Sample data files
`data/gen_spam_ds.js` The file used to preprocess/generate the files for the spam dataset
`stats/` Sample statistics files; filenames are in the form `dataset_lr_epochs.stats`
`weights/` Sample weights files; filenames are in the form `dataset_lr_epochs.init` or `dataset_lr_epochs.trained`
`arch-defs.ss` Neural network record type definitions
`autorun.ss` Script to easily initiate the training and test procedures (see §4.1)
`dense-layer.ss` Dense layer implementation
`loss-layer.ss` Loss layer implementation
`main.ss` Main starting point; imports dependencies and defines prompted train/test procedures
`model-io.ss` Defines utilities for loading/exporting model weights/datasets/stats
`model.ss` Defines model train, test (predict), and evaluation procedures
`sigmoid-layer.ss` Sigmoid layer implementation

3.7 Source code

The source code, datasets, weight files, and statistics files are located on GitHub at [@jlam5555/nm-scheme](https://github.com/jlam5555/nm-scheme).

3.8 Note on floating-point precision

Once when running on a peer’s dataset that there was a difference of 0.001 in one number of the macro-averaged statistics, when all other outputs (statistics and trained weights) matched exactly for all other datasets, including both the provided datasets and both of our custom datasets.

Chez Scheme uses arbitrary-precision floating-point by default (analogous to Java’s `BigDecimal`), which differs from the more standard IEEE 32-bit or 64-bit floating point. I believe this may be due to the rounding difference rather than some algorithmic error.

4 Instructions

Make sure Chez Scheme is installed. The following are tested with Chez Scheme 9.5 on Debian 10 (kernel 4.19.0).

4.1 Without entering the REPL

```
$ scheme --script autorun.ss
```

For sake of example, this is what I used to train and time the spam example (`--optimize-level 3` reduces execution time by roughly 10%):

```
$ time scheme --optimize-level 3 --script autorun.ss 1>/dev/null <<EOF
weights/spam.init
data/spam.train
0.1
25
weights/spam_0.1_25.trained
data/spam.test
stats/spam_0.1_25.stats
EOF
```

4.2 Using the REPL

```
$ scheme --optimize-level 3
> (load "main.ss")
> (prompt-train-test-model)
```

These two commands run the same commands as `autorun.ss`. The procedure `prompt-train-test-model` is a convenience function to facilitate the training and testing of a FC model from start-to-finish (as described for this assignment), prompting the user for the relevant model parameters and input/output files. This method calls the procedures `prompt-train-model` and `prompt-test-model`, also defined in `main.ss`, which can also be used separately. It should be easy to see from the source code that this calls the underlying model loading, training, testing, and exporting procedures described in more detail in the previous section.

5 Custom dataset

The dataset I chose was the Spambase UCI dataset¹. This attempts to classify emails as spam or not spam based on a number of custom features. The dataset comes from a set of work emails.

The dataset comprises 4601 samples. There are 57 continuous inputs (email features), and one boolean output (whether the email is classified as spam). A more in-depth description of the features can be found at the UCI website.

- Features 1-48 indicate the percentage frequencies of 48 common words found in spam emails.
- Features 49-54 indicate the percentage frequencies of 6 common character sequences (mostly emoticons) that are commonly found in spam emails.
- Feature 55 indicates the average length of capital letter runs (consecutive capital letters).
- Feature 56 indicates the maximum length of capital letter runs.
- Feature 57 indicates the sum of the lengths of capital letter runs.

The dataset files (`data/spam.train`, `data/spam.test`, and `weights/spam.init`) are generated using the script `data/gen_spam_ds.js`. The number of outputs (arbitrarily chosen to be 64 for this problem) and hidden nodes (1 output in this case) must be explicitly specified. The preprocessing steps are:

1. Download the dataset file from the UCI repository.
2. Read in the data as a 2-D matrix, where each row is one sample.
3. Shuffle the dataset (Fisher-Yates).
4. Separate features from labels.
5. Scale each feature to the range [0, 1] (min-max scaling).
6. Create an 80/20 train/test split.
7. Generate weight matrices using number of input nodes, number of hidden nodes, and number output nodes. (Weights are randomly generated from a standard normal distribution using the Box-Muller transform).
8. Export the train/test datasets and weights to their respective files.

I arbitrarily chose to use a network with 64 hidden nodes, and trained with a 0.1 learning rate for 25 epochs. This gives decent overall accuracy (> 90%, which is fair for the non-critical task of spam detection). Given that this dataset is much larger than the provided ones, and the network is larger, the training takes much longer; 25 epochs takes roughly 26 seconds on my system (i7-2600 CPU) with `--optimize-level 3`, or 30 seconds without it.

¹<https://archive.ics.uci.edu/ml/datasets/Spambase>

ECE469 – Pset 1

Jonathan Lam

October 5, 2020

1. For each task environment, indicate what categories the task environment belongs to. Also state whether the rational agent would base its decisions on only the current percept or the entire percept history.

- (a) A program that plays Go against the user.

fully observable	entire Go board is observable
strategic	the current state is based on your past moves and your opponent's move (which you cannot control)
sequential	previous moves affect current available moves
semi-dynamic	there may be time limits on a move, but otherwise nothing changes while you're thinking about the next move to make
discrete	finite, non-continuous possible next moves
multi-agent	a second player is playing against you

All of the possible moves are governed by the current board state, so the program need only look at the current board state to determine a next move. (This is as far as I can tell by briefly skimming over the rules, but I am not sure if there are any moves like a castle that do indeed depend on previous state and are not knowable from the current board state.)

- (b) A program controlling a robotic arm that detects and stacks Jenga pieces (one of the senior projects this year!). The goal is not to have the robotic arm play the game, but rather it will detect pieces on a surface and build the initial tower.

The assumption here is that this robot is in an generic uncontrolled setting, where other factors may interfere and the Jenga blocks may not be entirely uniform. We can consider any unexpected behavior (e.g., bug landing on block, wind gust, or person moving block) to be part of a stochastic environment, rather than an adversarial agent.

partially observable	sensors may not capture every minute aspect of the Jenga tower and other environmental conditions (e.g., temperature, humidity)
stochastic	other unspecified environmental factors (e.g., a person or another robot nearby) may affect the environment
sequential	previous moves affect the shape of the tower (and thus the next moves)
dynamic	the tower can fall over or be otherwise altered while the program is deciding which move to make
continuous	infinitely many different actions (even if there are finitely-many pieces, any piece can be moved in infinitely many different ways)
single agent	no other specified agents

What information the robot uses depends on how it calculates the next move. If it creates an internal physical model of the world which it updates periodically with its sensors (which is common in robots; credit to Derek Lee for suggesting this idea to me, since I don't know much about robotics), then it relies on this model (and thus its percept history). However, it is also possible that a simpler robot simply takes a snapshot of the current situation with its sensors and tries to choose the next move based only on its current percept.

- (c) *A face recognition system that detects faces in an image and looks them up in a database of known faces to find matches (assume the system has already been trained).*

fully observable	entire image is observable
deterministic	there is no random actions happening in an episode; the algorithm proceeds deterministically given an image
episodic	current image recognition doesn't depend on previous images (this would be different if it were training on this dataset)
static	image and database doesn't change while the system is processing an image
continuous	using the given assumption that an image is continuous
single agent	no other agents

Since the system is already trained, it only depends on the current percept (the current image) to determine whether the face matches something in the dataset. (I.e., to say that it uses its percept history means that it uses other images from the evaluation dataset to inform its opinions, which is false.)

2. *Assume a program is playing a deterministic (really strategic), turn-taking, two-player, zero-sum game of perfect information.*

- (a) *Assuming that no pruning occurs, fill in the minimax values of all the internal nodes (including the root) in the game tree. Also circle the move that is selected according to the search.*

See Figure 1.

- (b) *Now assuming that alpha-beta pruning occurs, draw lines through the edges in the game tree that separate the nodes at which the pruning decision occurs from the nodes which are never considered.*

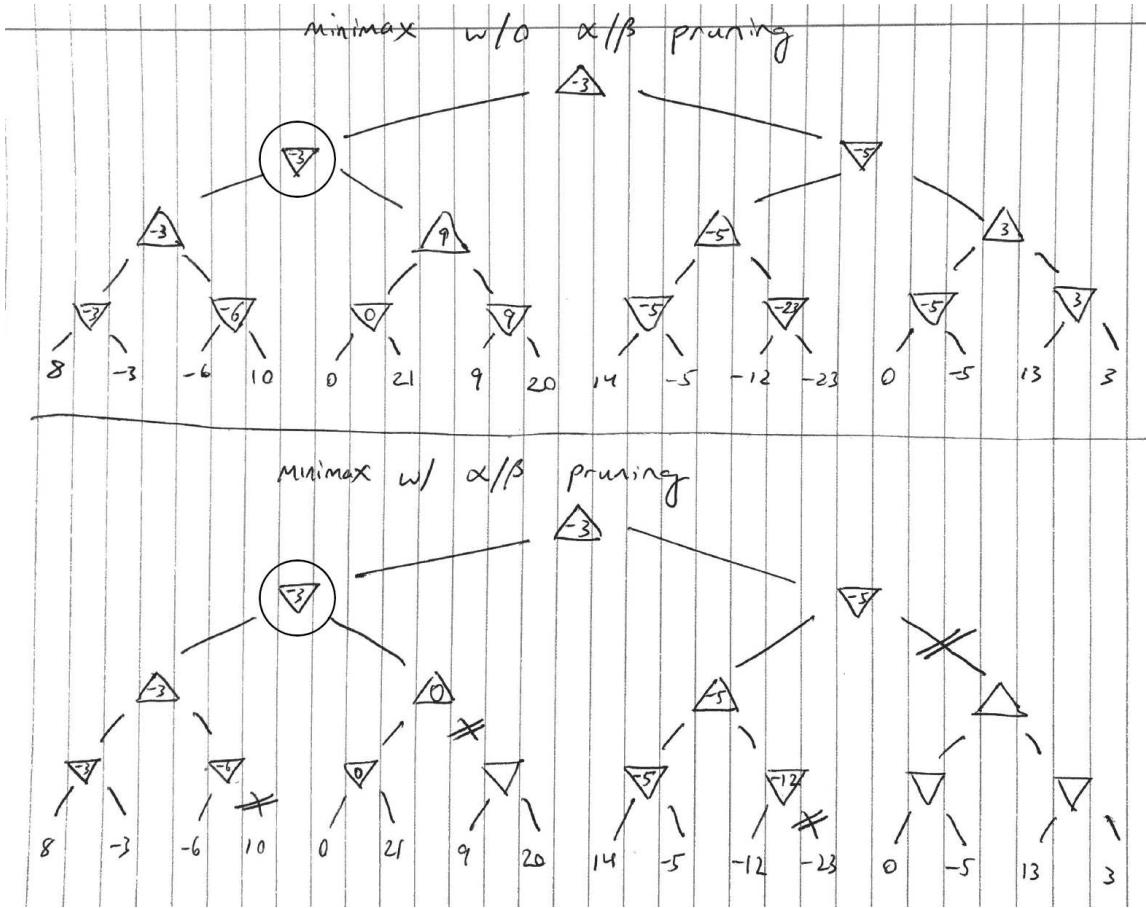


Figure 1: Minimax search, with and without α - β pruning. The circled node is the selected next action for MAX. It is clear that both give the same minimax values.

I did the pruning intuitively, so the α and β values are not shown. It is clear that the minimax values at each of the nodes where it is generated is equal to the unpruned tree. The intuition for each prune, from left to right:

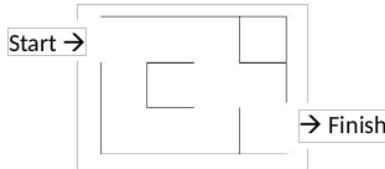
- -6 at the MIN node was lower than the -3 value at its parent MAX node, so it wouldn't be chosen by MAX at the parent node
- 0 at the MAX node is higher than the -3 value at its parent MIN node, so it wouldn't be chosen by MIN at the parent node
- -12 at the MIN node is lower than the -3 value at the root MAX node, so it wouldn't be chosen by MAX at the root
- -5 at the MIN node is lower than the -3 value at the root (its parent) MAX node, so it wouldn't be chosen by MAX at the root

(c) If both players play perfectly, what will be the outcome of the game (i.e., will MAX or

MIN win or will it be a draw)? Briefly explain your answer.

MIN will win with a score of (-)3, as evident by the value of the root node. In other words, this is the minimax value, which is the end result given considering the worst case scenario for both players (if both players played optimally).

3. Consider an AI software agent (a program) that processes a maze contained in an $N \times M$ grid, where N (the number of rows) and M (the number of columns) are guaranteed to be at least 2. The entire maze is available to the agent at once, not as an image, but as an array indicating where the walls are. The agent is tasked with finding a path from the top left square to the bottom right square. The path cost is the number of steps (i.e., every move from one square to an adjacent square adds one to the cost). Only horizontal and vertical steps are allowed. Walls will only occur between squares. It is possible that there may be loops, multiple paths to the solution, or perhaps no path to the solution. An example of such a maze, where N is 3 and M is 4:



- (a) First consider applying a tree search version of depth-first search (DFS) to this problem. For this question, consider a version that does not remember all reached nodes, but that does remember nodes on the current path to avoid cycles. Is this strategy complete? Is this strategy optimal?

This strategy is complete (remembering nodes on the current path will avoid infinite loops, and DFS will eventually expand all nodes in the (finite) space just like BFS will), but not optimal (the first solution found will not necessarily be the shortest path to the bottom right, it will just be a path to the bottom right).

- (b) Now consider applying a graph search version of breadth-first search (BFS) to this problem, as discussed in class. Is such a strategy complete? Is such a strategy optimal?

Since the search space is a finite board (and we can easily keep track of repeated nodes since this is a graph-search algorithm), this is complete. BFS is also optimal since we are looking for the shortest path to the finish, which BFS will find first.

- (c) Now consider applying the graph search version of A* search, as discussed in class, to the problem. You will consider four heuristic, where x is the current row and y is the current column. Assume rows are labeled from 1 to N and columns are labeled from 1 to M ; the top-left cell (the starting position) is $(1,1)$, and the bottom-right cell (the destination) is (N,M) . The four heuristics we are considering are:

$$\begin{aligned} H_1(n) &= 0 \\ H_2(n) &= N - x + M - y \\ H_3(n) &= \sqrt{(N - x)^2 + (M - y)^2} \\ H_4(n) &= (N - x)(M - y) \end{aligned}$$

Which of the above four heuristics would guarantee that the graph search version of A search is optimal? Specify all that apply.*

Any admissible heuristic will lead to an optimal A* search. In particular, the smallest number of moves between two points when only horizontal or vertical moves are considered is the Manhattan distance. Thus, any of the heuristics which is guaranteed to be no greater than the Manhattan distance will guarantee optimality with A*. In particular:

- $H_2 \geq H_1 = 0$ since $N > x$ and $M > y$ (i.e., Manhattan distance is positive).
- $H_2 \geq H_3$ because of the triangle inequality.
- H_2 is not always greater than H_4 (take for instance when $N - x = 2$ and $M - y = 3$).

H_4 is thus not admissible, and it can be shown to be not optimal by a counterexample (in which $H_4 >$ Manhattan distance for some tile on the optimal path, and thus the optimal path is not chosen by A*).

- (d) *Which of the above heuristics is the best one to use with the graph search version of A* search? Why?*

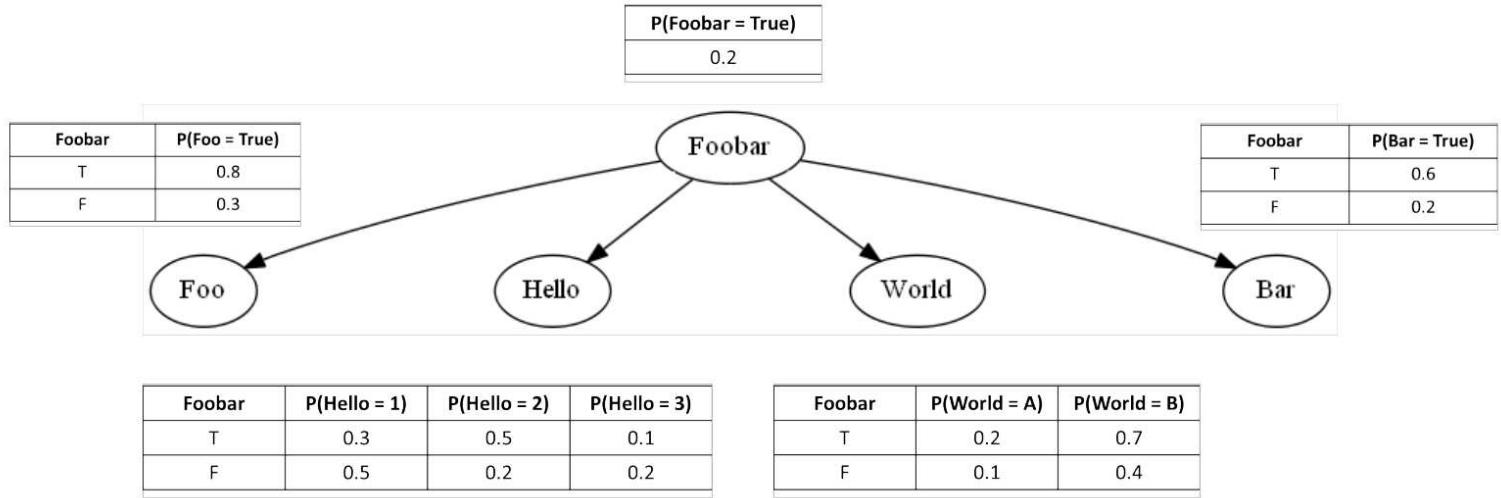
In general, the largest admissible heuristic is the most efficient, since it gets us closer to the actual total path costs. H_2 (the Manhattan distance) dominates the other admissible heuristics by the arguments in the previous answer (in fact, it is the largest admissible heuristic for this problem), so it is the best one to use with A*.

ECE469 – Pset 2

Jonathan Lam

November 23, 2020

1. Bayesian networks, maximum likelihood estimates, and naïve Bayes



Assume the Bayesian network the one shown above, where the (conditional) probabilities are already determined using maximum likelihood estimates.

Define $\mathbf{E} := \{\text{Foo}, \text{Hello} = 2, \text{World} = \text{C}, \overline{\text{Bar}}\}$.

- (a) *Compute $P(\mathbf{E} | \text{Foobar})$.*

With the naïve Bayes assumption we get the property that a node is conditionally independent of all its non-descendants given its parents. Thus, Foo, Hello, World, and Bar are all conditionally independent of each other given Foobar (and Foobar is not conditionally independent of any of the other nodes given any other node). If nodes A and B are conditionally independent of each other given C, then $P(A \wedge B | C) = P(A | C)P(B | C)$. This result extends to the case of multiple elements being mutually conditionally independent (via induction, not shown here). Thus:

$$\begin{aligned} P(\mathbf{E} | \text{Foobar}) &= P(\text{Foo} \wedge (\text{Hello} = 2) \wedge (\text{World} = \text{C}) \wedge \overline{\text{Bar}} | \text{Foobar}) \\ &= P(\text{Foo} | \text{Foobar}) P(\text{Hello} = 2 | \text{Foobar}) P(\text{World} = \text{C} | \text{Foobar}) P(\overline{\text{Bar}} | \text{Foobar}) \\ &= (0.8)(0.5)(0.1)(0.4) = 0.016 \end{aligned}$$

- (b) *Compute $P(\mathbf{E} | \overline{\text{Foobar}})$.*

This is the same as the previous question, except that Foobar is replaced with $\overline{\text{Foobar}}$.

$$P(\mathbf{E} | \overline{\text{Foobar}}) = (0.3)(0.2)(0.5)(0.8) = 0.024$$

- (c) *Compute $P(\text{Foobar} | \mathbf{E})$ and $P(\overline{\text{Foobar}} | \mathbf{E})$.*

This is finding a posterior probability using Bayes' rule.

$$\begin{aligned} P(\text{Foobar} | \mathbf{E}) &= \frac{P(\text{Foobar} \wedge \mathbf{E})}{P(\mathbf{E})} \\ &= \frac{P(\mathbf{E} | \text{Foobar}) P(\text{Foobar})}{P(\mathbf{E} \wedge \text{Foobar}) + P(\mathbf{E} \wedge \overline{\text{Foobar}})} \\ &= \frac{P(\mathbf{E} | \text{Foobar}) P(\text{Foobar})}{P(\mathbf{E} | \text{Foobar}) P(\text{Foobar}) + P(\mathbf{E} | \overline{\text{Foobar}}) P(\overline{\text{Foobar}})} \\ &= \frac{(0.016)(0.2)}{(0.016)(0.2) + (0.024)(0.8)} = \frac{0.0032}{0.0224} = \frac{1}{7} \\ P(\overline{\text{Foobar}} | \mathbf{E}) &= [P(\text{Foobar} | \mathbf{E})]^C = 1 - \frac{1}{7} = \frac{6}{7} \end{aligned}$$

2. Machine learning concepts

- (a) Related to machine learning, explain the concept of feature selection.

Feature selection involves reducing the dimensionality of the input space as a form of simplifying the model (whether to improve generalization and/or to reduce computation count), and features are usually chosen by evaluating against a validation set.

- (b) Related to decision trees, would it ever make sense for the same feature / attribute to be tested twice along a single path from a root to a leaf?

Yes – splits (using the algorithm discussed in class) attempt to maximize the amount of information (in the information-theoretic sense) in the split. It is plausible that splitting on one variable provides the most information at one point, and then after conditioning (splitting) on one or more other variable(s), splitting on this variable again gives the most information again.

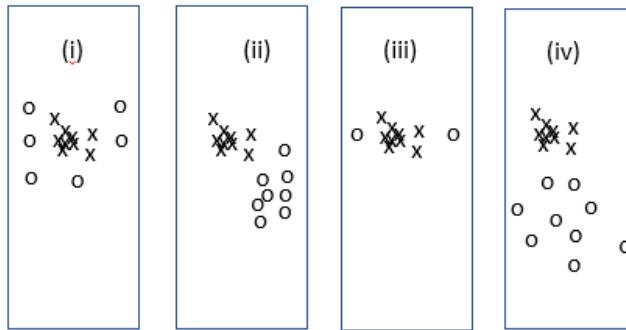
- (c) Briefly explain the conclusion of the No Free Lunch theorem.

No machine learning representation (e.g., decision trees, neural networks, Bayesian networks, etc.) can efficiently model all possible values of the hypothesis space (all possible mappings from inputs to outputs). In other words, we create efficient models by making assumptions about the form of the learning problem, and if trying to model all possible learning problems (hypotheses), and if all hypotheses were equally likely, random guessing is as good as you can get.

- (d) Briefly explain why it typically takes longer to apply a k-nearest neighbor system than to train it. Is this a good thing or a bad thing?

Training a KNN is essentially remembering all of the points (trivial); inference using a KNN involves finding the K nearest neighbors (less trivial, have to comb through the points in some way). This isn't great; usually we want a fast inference, because this is how the model will be used (most other models have complex models that take a lot of computation to train but are straightforward to apply).

- (e) Consider figures (i) through (iv) below. In each of them, the x's represent positive examples of some class, and the o's represent negative examples of the same class. Each example is fully represented as a two-dimensional numeric feature vector. Which of these classification tasks can theoretically be represented by a perceptron? Which of them can theoretically be represented by a neural network with one or more hidden layers? (Specify all that apply.)



A (single-level) perceptron is limited in that it can only represent linearly-separable classes, while a neural net with at least one hidden layer (or multi-level perceptron) can represent non-linearly separable classes. Thus, (ii) and (iv) are representable with a perceptron, all of them are representable with a neural network with hidden layers.

ECE469 – Pset 3

Jonathan Lam

December 14, 2020

1. Consider the famous sentence, “The quick brown fox jumps over the lazy dog.” Draw a reasonable parse tree for the sentence (assuming the existence of reasonable grammar rules). The root of the tree should be S , representing a sentence, and the leaves should be the words of the sentence. Also express the CFG rules, including the lexical rules, that are implied by the tree.

The sentence can be represented with the following parse tree:

```
[S
  [NP
    [Article The]
    [Adjs
      [Adjective quick]
      [Adjs [Adjective brown]]]
    [Noun fox]]
  [VP
    [VP [Verb jumps]]
    [PP
      [Prep over]
      [NP
        [Article the]
        [Adjs [Adjective lazy]]]
      [Noun dog]]]]]
```

(NB: this parse tree is valid Lisp (Chez Scheme) syntax (except perhaps that the word literals should be quoted)! Shows how well Lisp works with grammars.)

This uses the CFG given in the class example, with a different lexicon. The subset of the rules used in this example is summarized in the table below.

S	$\rightarrow \text{NP VP} \mid \dots$
NP	$\rightarrow \text{Article Adjs Noun} \mid \dots$
VP	$\rightarrow \text{Verb} \mid \text{VP PP} \mid \dots$
Adjs	$\rightarrow \text{Adjective} \mid \text{Adjective Adjs}$
PP	$\rightarrow \text{Prep NP}$
Article	$\rightarrow \text{the} \mid \dots$
Noun	$\rightarrow \text{fox} \mid \text{dog} \mid \dots$
Adjective	$\rightarrow \text{quick} \mid \text{brown} \mid \text{lazy} \mid \dots$
Verb	$\rightarrow \text{jumps} \mid \dots$
Prep	$\rightarrow \text{over} \mid \dots$

2. *Naïve Bayes systems work well for some text categorization tasks, even though the “naïve” assumption is clearly false. Explain what it means for the assumption to be false for this task, and give a specific example that demonstrates it is false.*

The naïve Bayes formulation for text categorization is as follows:

$$P(C|w_{1:N}) = \alpha P(C) \prod_{i=1}^N P(w_i|C)$$

The naïve assumption is that each word is conditionally independent of all of the other words given the category; this is false if two words are often seen together. E.g., the words “Volkswagen” and “beetle” should not be conditionally independent of the category “vehicle”: $P(\text{beetle}|\text{vehicle})$ is probably much lower than $P(\text{beetle}|\text{vehicle, Volkswagen})$, which will make the product of probabilities lower than it should be.

3. *Consider a conventional, feedforward neural network applied to the task of text categorization, and one sentence is being classified at a time. Assume it has been trained on a corpus with D labeled sentences, and the total size of the vocabulary is V . It is now being used to classify a document with T total tokens and U unique, or distinct, tokens. If a conventional feedforward neural network is being used for the task, what would typically be the number of input nodes? What would be represented by each input node?*

Without word embeddings, there would likely be V input nodes, where each input node represents the presence or frequency of a single word in the vocabulary. (D is irrelevant because one sentence is being classified at a time, and limiting the number of input nodes to T and U would not allow the network to use the whole vocabulary.)

4. Now consider text categorization involving d -dimensional word embeddings and a recurrent neural network (either a simple RNN, or a variation such as an LSTM). We have learned that it shouldn't be necessary to pad sentences to ensure they have equal length. When using other types of deep neural networks with word embeddings (such as a feedforward neural networks or a CNN), it typically is necessary to pad the input sentence. Why isn't it generally necessary to pad sentences when using an RNN for text categorization?

A FF NN makes its inference based only on its inputs (it has no sense of persistent state or context), so it requires sentence context from the surrounding words (the rest of a sentence); since they usually have a fixed number of inputs, they require a padded input. A RNN does have an internal state that persists over a series of inputs, so it can handle dynamic-length sentences, and only requires one input at a time.

5. Now consider a hidden Markov model being used for part-of-speech (POS) tagging. If the tagger is trained using a treebank (a corpus containing labeled examples of POS), what parameters need to be learned?

In a HMM using POS tagging, the current (hidden) state represents the part of speech. Thus, we need to learn two parameters: the transitions (“transition probabilities”) between different states (the next likely part of speech), as well as the most likely word given the current POS (“emission probability”).

6. Now consider a simple RNN being used as a POS tagger (in practice, a variation such as an LSTM would more likely be used). If the tagger is trained using a treebank, what parameters need to be learned?

The RNN's internal state is a function of the current input word's embedding and the previous state. It has to learn the function (weights) to transform the current state to the next state (similar to the transition probability), as well as the function (weights) to estimate the POS given the current state (similar to the emission probability).

SEARCHING FOR A MORE MINIMAL INTRINSIC DIMENSION OF OBJECTIVE LANDSCAPES

Jonathan Lam & Richard Lee

Department of Electrical Engineering

The Cooper Union for the Advancement of Science and Art

New York, NY 10003, USA

{lam12, lee66}@cooper.edu

ABSTRACT

We present a series of experiments aimed towards finding a more minimal “intrinsic dimension” of an objective landscape, as first described in Li et al. (2018). The intrinsic dimension is the smallest number of free variables needed to solve a given learning problem in a given neural architecture (which together form the objective landscape). It can be roughly thought of as the minimal parameterization of the objective landscape. The original paper presented a method to estimate intrinsic dimension using a random linear projection from the set of intrinsic weights to the set of full network weights; we extend this procedure by first applying a non-linear mapping on the intrinsic weights before applying the linear projection. We find that applying a power-term nonlinearity does not perform better or worse than the linear projection, and applying a random Fourier feature nonlinearity offers a small but notable gain in accuracy for a given intrinsic dimension.

1 INTRODUCTION

Neural networks have been at the forefront of machine learning applications, and their expressivity causes us to wonder how complex of a problem a given network architecture can solve; or, conversely, the minimum size network needed to describe some learning problem. While the former is mostly a theoretical question of the capacity of the largest networks, such as GPT-3 (Brown et al. (2020)), the latter is of practical importance in model deployment on embedded systems with limited resources. Many models have been suggested for reducing the memory and computational requirements of neural networks to meet these requirements without overcompromising accuracy; Cheng et al. (2017) presents a survey of such methods.

A related question lies not in the capacity of a model, but rather the size constraint of a given learning problem. This may be useful when reducing the size of a network may be harmful its capacity (e.g., parameter pruning or otherwise reducing the depth, width, units in a dense layer, and/or filters or kernel size in a convolutional layer), and thus we focus on the problem size rather than the network size; or when the the weights in all possible solution sets are not totally independent of one another (and thus can be reparameterized by some smaller problem). Toward the former point, Urban et al. (2017) discovers that the deep and convolutional structure of common image classification problems is very efficient, such that changing or reducing the network structure such as in knowledge distillation may have counterproductive effects. Thus we focus on the question: given a neural network architecture and a specific learning problem, what is the minimum description length (MDL) of a given objective landscape (which depends both on the architecture and the learning problem), without changing the overall network architecture? Li et al. (2018) attempt to tackle this problem by measuring what they call the “intrinsic dimension” of a problem. The “intrinsic weights” are a set of d weights, which are then projected onto the network’s set of D weights, where (presumably) $d < D$.

Li’s formulation follows the form (following the notation in Li’s paper):

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)} \quad (1)$$

where $\theta^{(D)} \in \mathbb{R}^D$ is the set of all weights in the network, $\theta_0^{(D)} \in M_{D \times 1}(\mathbb{R})$ is a non-trainable randomly-initiated set of initial biases for $\theta^{(D)}$, $P \in M_{d \times D}(\mathbb{R})$ is a non-trainable randomly-initiated linear projection matrix, and $\theta^{(d)} \in \mathbb{R}^d$ is the set of trainable intrinsic weights. Each set of weights θ can be each thought of as a column vector in this matrix equation, although the resulting $\theta^{(D)}$ has to be reshaped into the form necessary for its operation (e.g., reshaped into a convolutional kernel for a convolutional layer). We denote $\theta_0^{(D)}$ the “initial weights” to distinguish it from “bias weights,” which more typically refers to the offset weights (e.g., the set of weights b in the dense layer calculation $W\vec{x} + \vec{b}$).

Another rough interpretation of this formulation is that the model’s weights $\theta^{(D)}$ contain some large amount of information. However, it may be possible to find a function that manages to encode much of those redundancies of that information into a much smaller number of weights, $\theta^{(d)}$.

Li presents a toy problem to describe the intuition behind this solution, in which only ten simple linear constraints on the weights need to be met to constitute a correct (zero-loss) solution in a network with 1000 weights. Even if we randomly fix 990 of the weights, we can still solve the solution in a ten-dimensional space by ordinary gradient descent; in this contrived example, the intrinsic dimension of the problem is 10, and there are 990 degrees of freedom. If we randomly fix $\theta_0^{(D)}$, generate a random projection matrix P that will map the ten intrinsic dimensions to the full 1000 dimensions, and only train the smaller-dimensional vector $\theta^{(d)}$, we should be able to reach a solution.

This also gives a straightforward method to compress a model, namely by assuming a certain network architecture, and storing the seeds for the random initialization of $\theta_0^{(D)}$ and P , as well as the intrinsic weights. This has a memory savings of d/D for transmission or storage of the model. It is important to note that this method does not cause inference-time savings in either memory or computation, but the insight offered by the estimated intrinsic dimensionality may be used to engineer a more efficient network.

Our goal is to take the form of (1) and experiment with different forms of projection. While Li attempted to train in a random subspace \mathbb{R}^d of \mathbb{R}^D , which is defined by a linear projection matrix P , we experiment with different techniques for projection. Most notably, we aim to look at nonlinear mappings, with a focus on random Fourier features and power function mappings. This involves a projection of the more general form:

$$\theta^{(D)} = \theta_0^{(D)} + f_{proj}(\theta^{(d)}) \quad (2)$$

In our paper, our projection function consists of “augmenting” the intrinsic weights with a set of nonlinear mappings performed on it, and then performing a projection from this augmented space onto the final output space. This is akin to adding nonlinear features or interaction terms before applying a machine learning model in order to capture more interesting relations between the input features (in this case, the “features” are $\theta^{(d)}$). This can be represented in the following form:

$$\theta^{(D)} = \theta_0^{(D)} + P \begin{bmatrix} \theta^{(d)} \\ f_1(\theta^{(d)}) \\ f_2(\theta^{(d)}) \\ \vdots \end{bmatrix} \quad (3)$$

The goal of these experiments is to try to reduce the intrinsic dimension even further; there may be some redundancies that may not be best captured in linear mappings, and thus allow us to create an even smaller MDL, or gain more insights about the weights of a trained network. Moreover, since Li’s paper is the seminal work on this idea of “intrinsic dimension” and has not spawned any derivative papers to date, we experiment with various facets of the architecture that were unexplored in the original paper.¹

¹Relevant code is located at github.com/jlam55555/intrinsic-dimension-projections.

2 RELATED WORK

While model compression is not the ultimate goal of our research, it is perhaps the most straightforward application of attempting to find the smallest intrinsic dimension, as this leads to a smaller parameterization of the model. Cheng et al. (2017) present a survey of methods related to model compression and acceleration; four main techniques appear: parameter pruning and/or quantization, low-rank factorization, transferred or compact convolutional filters, and knowledge distillation. The method we use is most similar to the low-rank factorization method, as we attempt to factor every layer’s weights into the product of a projection matrix and the same set of weights. The major difference is that one matrix in every factorization pair is fixed, and the other matrix is the same for all factorization pairs; i.e., there is only one set of intrinsic weights for the entire network, not for each trainable weight vector in the model. This allows our method to achieve a result closer to the MDL than low-rank factorization, albeit at the cost of accuracy.

Li et al. (2018) perform a number of experiments on the implications of intrinsic dimension, such as examining effect of model shape (depth and width), comparing the weight efficiency of convolutional versus fully-connected architectures for different objective landscapes (learning problems of different natures), and finding the intrinsic dimension of various common problems such as MNIST and CIFAR-10. However, there is still a largely-unexplored space in the formulation of the intrinsic dimension that this paper does not consider; we attempt to explore some of those orthogonal directions.

3 METHODS AND INTUITION

3.1 DESIGNING THE BASELINE ARCHITECTURE

There is a high-dimensional search space of hyperparameters for even simple problems such as the spiral classification problem and MNIST. We use the simple fully-connected model for MNIST as presented in Li’s paper. This is a 784-200-200 FC network with a softmax final layer ($D \approx 2 \times 10^5$). We choose this simple model mainly for its simplicity and low training time. Using our custom projection layers significantly increases training time, so this was useful given our limited time and resources.

We conjecture that these results will extend to larger models. Li shows empirical evidence in the seminal work on intrinsic dimension that the concepts of intrinsic dimension were extensible to larger networks and were valid for both convolutional and fully-connected layers.

3.2 LINEAR PROJECTION

The linear projection matrix P may be simply generated by random initialization: this results in a (very large) dense matrix. Since the output dimensionality D is large, the resulting vectors in the output space (the columns of P) can be approximated as roughly orthogonal (<https://math.stackexchange.com/users/6179/did>). If these vectors are normalized (forming an orthonormal basis), then distance in the intrinsic and output dimensions is preserved.

In Li et al. (2018), the columns are normalized but not explicitly orthogonalized. Besides the simple random dense projection, two other random linear projection methods (a sparse method and the Fastfood method) are used. These methods decrease computational and memory cost but do not benefit compression, and we do not experiment with them in our research. We use the basic dense linear projection in our base model. We do not explicitly orthogonalize the vectors in the output space. We also experiment with normalizing these vectors.

We implement the formula (1) on a per-layer basis; i.e., while there is only one set of intrinsic dimension weights, there is a separate $\theta_0^{(D)}$ and P initialized for every layer. This offers some flexibility in case different initialization methods are desired; if all initialization methods are set to the same value, then this method is equivalent to agglomerating all of the weights of the entire model into a model-wide $\theta^{(D)}$.

This raises the question of initialization of P , and regularization. Intuitively, the initialization of the non-trainable variables P and $\theta_0^{(D)}$ may seem more critical to performance than in the case

of trainable parameters. Following Li’s example, we use a random normal initializer for P , the `he_normal` initializer initially proposed in He et al. (2015) for $\theta_0^{(D)}$, and perform regularization on $\theta^{(D)}$ rather than directly on the trainable weights $\theta^{(d)}$. Attempting different initialization and regularization schemes may be a direction for future research.

3.3 POWER TERMS AUGMENTATION

The use of nonlinear mappings is based on the simple intuition that nonlinear functions can capture more complex relationships (and thus more information) than linear functions given the same number of parameters. A quadratic relationship that can be expressed in three parameters can take multiple linear approximations to estimate well. While the toy problem involved a linear constraint on the weights in the solution set, we conjecture that this is likely not always the case. Thus we might be able to design a better $f_{proj}(\theta^{(d)})$ than a simple linear projection (i.e., a matrix multiplication).

The simplest nonlinear mapping we can perform is a quadratic mapping. The most general quadratic form would be:

$$\theta_k^{(D)} = \sum_{1 \leq i \leq d} a_{ik} \theta_i^{(d)} + \sum_{1 \leq i \leq j \leq d} b_{ijk} \theta_i^{(d)} \theta_j^{(d)} \quad (4)$$

This can be represented as the concatenation of $\theta^{(d)}$ with the quadratic terms $\theta_i^{(d)} \theta_j^{(d)}$ as one long column vector. This would form a $\mathbb{R}^{d+d^2/2}$ column vector, which could then be projected by an augmented projection matrix. (Consider this a form of augmenting the intrinsic weights with quadratic terms, and then performing a linear projection of these augmented weights onto the full network weights.)

With this method, however, the number of cross terms grows quadratically, which causes a blowup in memory usage. If we ignore cross terms, we can still have squared terms with double the number of weights. Similarly, we can have cubed terms, fourth-power terms, etc. with a linear memory requirement:

$$\theta_k^{(D)} = \sum_{i=1}^d a_{ik} \theta_i^{(d)} + b_{ik} (\theta_i^{(d)})^2 + c_{ik} (\theta_i^{(d)})^3 + \dots \quad (5)$$

We conjecture that this might be able to have some of the benefit of nonlinear (polynomial) functions without an unreasonable number of parameters. In other words:

$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ (\theta^{(d)})^2 \\ (\theta^{(d)})^3 \\ \vdots \end{bmatrix} \quad (6)$$

where the exponents denote Hadamard (element-wise) exponentiation. In our experimentation we (arbitrarily) limit ourselves to squared and cubed terms.

Adding new function types also introduces many new hyperparameters, especially for the initialization of the random vectors. That is, if the weights of the projection matrix P were drawn from the same distribution, then for $\theta_i^{(d)} \gg 1$, squaring or cubing could result in a few dominant intrinsic weights, and for $\theta_i^{(d)} \ll 1$, squaring or cubing would result in a few intrinsic weights of negligible impact.

In other words, each $\theta_j^{(D)}$ is formed by a linear combination of terms from the augmented intrinsic weights matrix, which can be thought of as basis functions. Since the linear, squared, and cubed terms are not independent, we are forcing $\theta_j^{(D)}$ to be expressed as a sum of “basis functions” that are third-degree polynomials of a single variable ($\theta_i^{(d)}$) with random, fixed coefficients. Intuitively, polynomials with larger high-order (second- and third-order) coefficients are more nonlinear and more unstable. We were not exactly sure what the effect of this would be: would larger high-order coefficients increase performance by improving expressivity, or would it hurt convergence because of the extra constraints on each intrinsic weight? The results in the following sections show that

forcing the basis functions to be more linear by attenuating the squared and cubed coefficients had surprisingly little effect on accuracy.

In order to check our intuition, we experiment with a “pre-training” session in which the projection matrix is trainable. More details follow in the experiments section. Following the recurring theme of this paper, there is a large unexplored space in initialization and regularization of the intrinsic weights and projection matrix.

3.4 RANDOM FOURIER FEATURES AUGMENTATION

The concept of Fourier features has been recognized for its success with scaling up the “kernel trick” and have been analyzed in great mathematical detail (see Li et al. (2019); Rahimi & Recht (2007)). However, we will focus on the more intuitive explanation given by Tancik et al. (2020), which focuses on the inverse rendering problem. In the inverse rendering problem, a network is trained on an image to output the pixel value given a set of coordinates (x, y) . Historically, this problem learns only low-frequency functions (general patches of an image), and is not well-suited for high-frequency features (details and textures). Tancik et al. (2020) employs the mapping γ defined as follows to the input vector $\vec{v} = (x, y)$.

$$\gamma(\vec{v}) = \begin{bmatrix} a_1 \cos(2\pi \vec{b}_1^T \vec{v}) \\ a_1 \sin(2\pi \vec{b}_1^T \vec{v}) \\ a_2 \cos(2\pi \vec{b}_2^T \vec{v}) \\ a_2 \sin(2\pi \vec{b}_2^T \vec{v}) \\ \vdots \\ a_m \cos(2\pi \vec{b}_M^T \vec{v}) \\ a_m \sin(2\pi \vec{b}_M^T \vec{v}) \end{bmatrix} \quad (7)$$

The vectors \vec{b}_m , $m = 1, 2, \dots, M$ are random vectors; they can be interpreted as a random frequency term in M -dimensions. In the inverse rendering case, this can be interpreted as “frequency-sampling” the image for patterns centered at \vec{v} with M random frequencies (random in both magnitude and direction). This mapping is periodic and shift-invariant, and more amenable to high-frequency patterns. The range of frequencies that will be captured by this pattern are dependent on the distribution from which the elements of \vec{b}_m are drawn; various distributions are discussed in Tancik’s paper.

While we don’t have the geometric interpretation of frequencies as in the inverse rendering problem, the random Fourier feature mapping can be thought of as a general way to extract more interesting interactions between features. Using the same framework as in the power terms, we augment the (linear) intrinsic weight terms with a set of random Fourier features.

$$\theta_j^{(D)} = \sum_i a_{ij} \theta_i^{(d)} + \sum_m \cos(b_m \theta^{(d)}) + \sin(b_m \theta^{(d)}) \quad (8)$$

The equivalent operation may be written as a matrix operation. Note that this still requires a projection from the Fourier feature-augmented intrinsic weights to the output dimension D .

$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ \cos(B\theta^{(d)}) \\ \sin(B\theta^{(d)}) \end{bmatrix} \quad (9)$$

Since the (real-valued) sine and cosine functions are bounded, we don’t have to worry as much about a blowup of these terms as we did with the power terms.

3.5 DATASETS AND PREPROCESSING

Note that, despite the ability of this method to compress a network for storage or transmission, this method increases computational and memory cost during training. Inference is not affected. Given that P is a $D \times d$ matrix, this has a much higher memory consumption than the original network containing only D weights. Using the sparse or Fastfood methods should partially alleviate

this problem. Since we implement linear projections in multiple places, we elected to only use the simpler (but more expensive) random dense linear projection.

As mentioned previously, we perform most of our experiments on MNIST due to the increased cost. Some initial tests were performed on a simple generated spiral binary classification dataset. We did not possess the ability to train on larger datasets such as CIFAR-10 nor ImageNet.

As we are not looking to achieve the best performance, but rather compare the performance of various projection types against the non-projection models, we do not perform pre-processing (e.g., data augmentation) on any of the datasets. This results in our model having poorer performance than reported in Li’s paper using the same architecture, dataset (MNIST), intrinsic dimension, and other hyperparameters (e.g., initialization and regularization parameters); as a result, we create and present our own baselines using the proposed architecture.

4 EXPERIMENTS

4.1 DIRECT VS. LINEAR PROJECTION MODELS

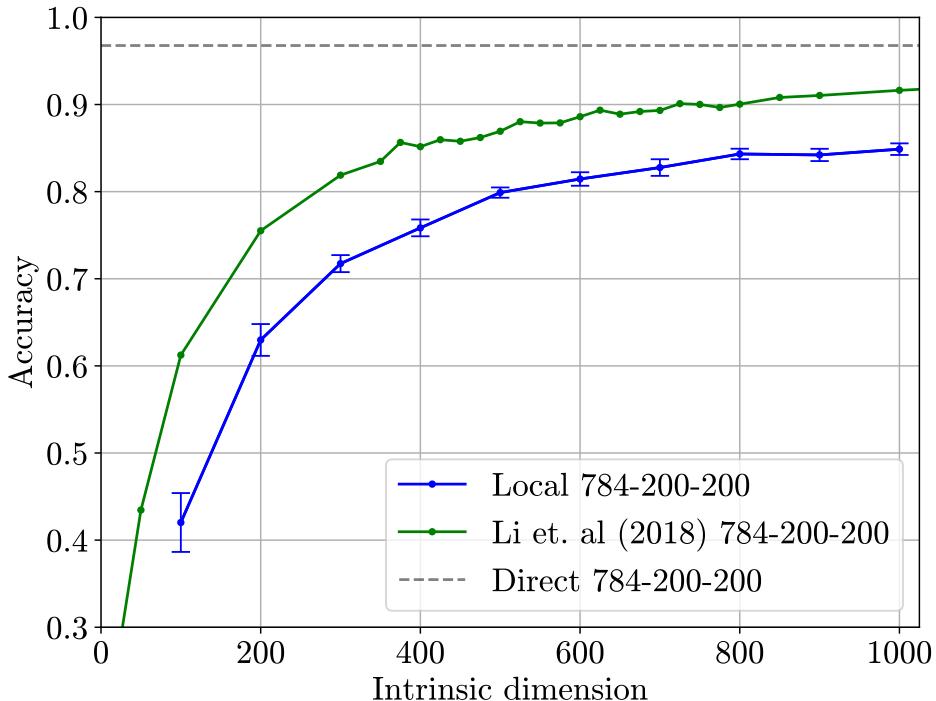


Figure 1: Comparison of our local MNIST implementation to Li et al. (2018). Both models are 784-200-200 FC models with default L2 regularization parameter $\lambda = 0.001$. Each dot for our local model represents the mean of eight trained models, and the error bars represent one standard deviation from the mean. We do not perform input data augmentation, which we believe is the most significant cause of the discrepancies in the accuracy. Li et al. data from https://github.com/uber-research/intrinsic-dimension/blob/master/intrinsic_dim/plots/main_plots.ipynb.

The 784-200-200 model was recreated directly using the standard Keras layers, and through projection using our own custom Keras layers. While we were able to achieve a similarly high result using the direct model (with test accuracy of 96.77%, which is not much lower than the reported test accuracy of 98.47% using the direct model in Li et al. (2018)), we were not able to recreate the 90% accuracy intrinsic dimension estimate of $d_{int,90} \approx 800$ for MNIST. For most of the intrinsic value

dimensions we tested, we achieve roughly 10% lower accuracy for the same number of intrinsic dimensions. We speculate that this mostly is due to a lack of input image augmentation, but warrants further experimentation.

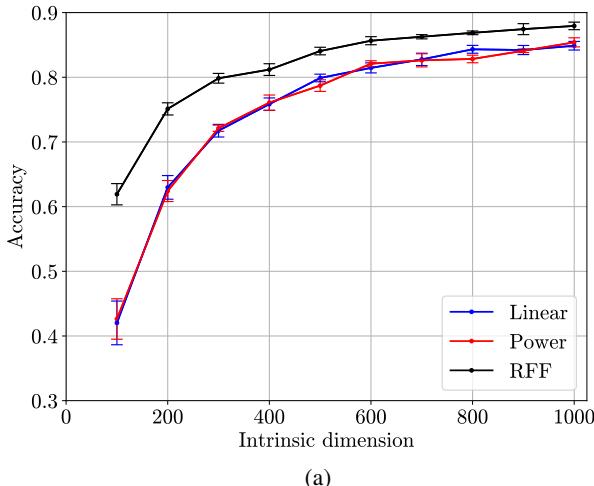
Our goal in these experiments is to achieve a (relatively) higher accuracy with a fixed intrinsic dimensionality, not to achieve the highest (absolute) accuracy or to recreate the results in Li et al. (2018) exactly. As a result, we use our results as-is and predict that our general results still hold in a relative sense. Given that accuracy as a function of intrinsic dimension is monotonically increasing, our result is equivalent to finding a lower intrinsic dimension given a fixed accuracy goal: i.e., attempting to minimize $d_{int,90}$.

For most of our models, we train with intrinsic dimensions $d = 100, 200, \dots, 1000$, a default learning rate of $\alpha = 0.001$, and L2 regularization for both the FC layers' kernels and biases with $\lambda = 0.001$. The latter two parameters appeared to have a large impact: setting a higher learning rate (e.g., $\alpha = 0.01$) quickly causes a catastrophic drop in accuracy due failure to converge. In the case of augmenting the intrinsic weights with power terms, even $\alpha = 0.001$ sometimes does not converge for the larger d values – this is shown in some of the later results. We use the same regularization parameter value as Li; removing regularization empirically causes a slight decrease in test accuracy.

We follow Li's example and use `he_normal` to initialize $\theta_0^{(D)}$ and `random_normal` to initialize P . We initialize the intrinsic weights using `random_normal` rather than `zeros`. These choices are somewhat arbitrary and can be refined using a hyperparameter search.

4.2 AUGMENTING LINEAR PROJECTION WITH POWER AND RFF TERMS

Figure 2 displays the accuracy of the various projection models, as described in the Methods section, for intrinsic dimensions 100 through 1000. We observe a considerable difference between the accuracy of the power-terms-augmented model and the ordinary linear projection model.



(a)

d	Accuracy		
	Linear	Power	RFF
100	0.42 ± 0.03	0.43 ± 0.02	0.62 ± 0.02
200	0.63 ± 0.02	0.624 ± 0.009	0.751 ± 0.009
300	0.72 ± 0.01	0.721 ± 0.008	0.798 ± 0.008
400	0.758 ± 0.009	0.761 ± 0.009	0.812 ± 0.009
500	0.799 ± 0.006	0.787 ± 0.006	0.841 ± 0.006
600	0.815 ± 0.008	0.821 ± 0.006	0.857 ± 0.006
700	0.83 ± 0.01	0.826 ± 0.003	0.863 ± 0.003
800	0.843 ± 0.006	0.828 ± 0.003	0.869 ± 0.003
900	0.842 ± 0.007	0.841 ± 0.008	0.874 ± 0.008
1000	0.849 ± 0.007	0.854 ± 0.006	0.880 ± 0.006

(b)

Figure 2: Comparison between projection types and accuracy vs. intrinsic dimensions. The linear and power-terms-augmented models perform roughly equally for all intrinsic dimensionalities, and the RFF-augmented model clearly outperform the other two methods for all intrinsic dimensionalities. Each dot in 2a represents the mean of eight trained models, and the error bars represent one standard deviation from the mean. 2b represents the same data in tabular form.

It is clear from the figure that augmentation with power terms has no meaningful effect on test accuracy when compared to the linear projection model, and the RFF-augmented model has a small but notable gain in accuracy for every value of intrinsic dimension.

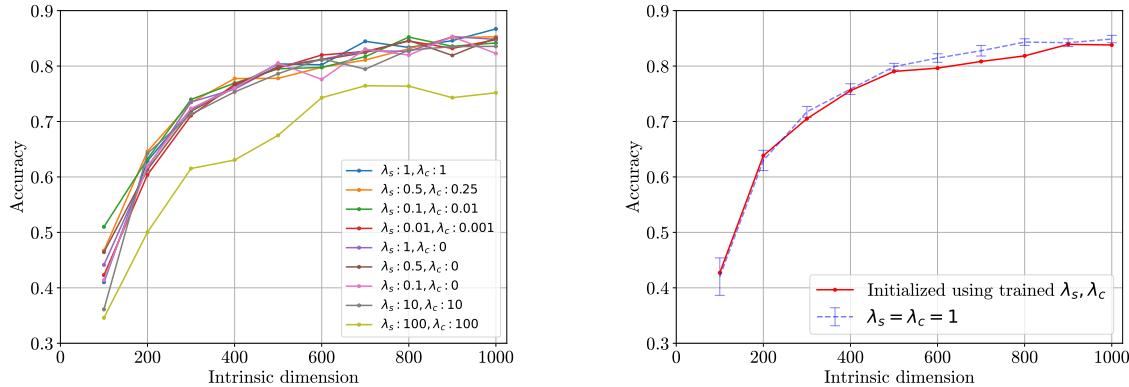
As stated in the methods section, we imagined that adding power terms might improve performance by being able to capture more complex relationships between intrinsic weights, and we were not sure what effect random Fourier feature mappings would have, having lost the intuitive geometric interpretation that can be found in the inverse rendering problem. In hindsight, after completing experiments that determine the distribution of the trained weights, we have a different hypothesis to explain our results. The results of those experiments are shown in the following section.

We found that the distribution of the intrinsic weights tend to be very narrow $\sigma_{\theta(d)} \ll 1$; i.e. most of the trained intrinsic weights tend to converge towards very small numbers. As a result, the linear terms greatly dominate the squared and cubed terms when augmenting using power terms, and they provide almost no contribution. On the other hand, since the cosine function non-linearly maps small arguments (small intrinsic weights) to values near unity, there is a change in variance from the intrinsic weights matrix to its augmented form, which may allow for more expressivity in the resulting $\theta^{(D)}$. It would be interesting to follow up this research with a mathematical examination of this theory, which is something outside the knowledge scope of the current paper's authors.

4.3 DIFFERENT COEFFICIENTS FOR LINEAR, SQUARED, AND CUBED TERMS

After training an initial set of models as shown in the previous section, we attempt to improve the unremarkable performance of the models augmented by power terms. The results of this section give rise to the proposed explanation of the empirical results given in the previous section.

In our formulation of augmenting with power terms (5), we first began by choosing the projection weights $\{a_{ik}\}$, $\{b_{ik}\}$, and $\{c_{ik}\}$ from the same Normal distribution. Our intuition was that the squared and cubed terms might greatly dominate the linear terms or end up having negligible effect, so we experimented with setting the standard deviation of the weights in the projection matrix for the squared and cubed terms to be much smaller than those for the linear terms. We define λ_s and λ_c to be scaling factors for the standard deviation of the random Normal initializer used for the squared and cubed weights in the projection matrix (these are normalized to the standard deviation of the projection matrix weights for the linear, i.e., $\lambda_l = 1$). The results are shown in Figure 3a. There is no noticeable difference between any of these training runs, except for very large coefficients, which only harm accuracy.



(a) First attempt: Arbitrarily choosing coefficients for projection matrix weights. The accuracy is mostly roughly equal between the models, except when making the coefficients λ_s and λ_c very large.

(b) Second attempt: Using the parameters of the projection matrix weights from the networks trained with a trainable projection variable. This provides no accuracy gain over the models in 3a.

Figure 3: Attempts at improving the performance of power-terms-augmented models via different initialization of projection weights for linear, squared, and cubed terms.

However, we felt that this method of choosing the coefficients λ_s and λ_c was very arbitrary and not robust at all. Our second attempt involves allowing the matrix P to be trainable, and thus estimating the values λ_s and λ_c from the trained distributions in P . Sample distributions of the

projection matrix weights are shown in Figure 4 for $d \in \{100, 500, 1000\}$. These generated λ_s and λ_c values were then used to initialize the P matrices, and all of the models were retrained. The result of this experiment is shown in Figure 3b. There are a number of noteworthy details. First,

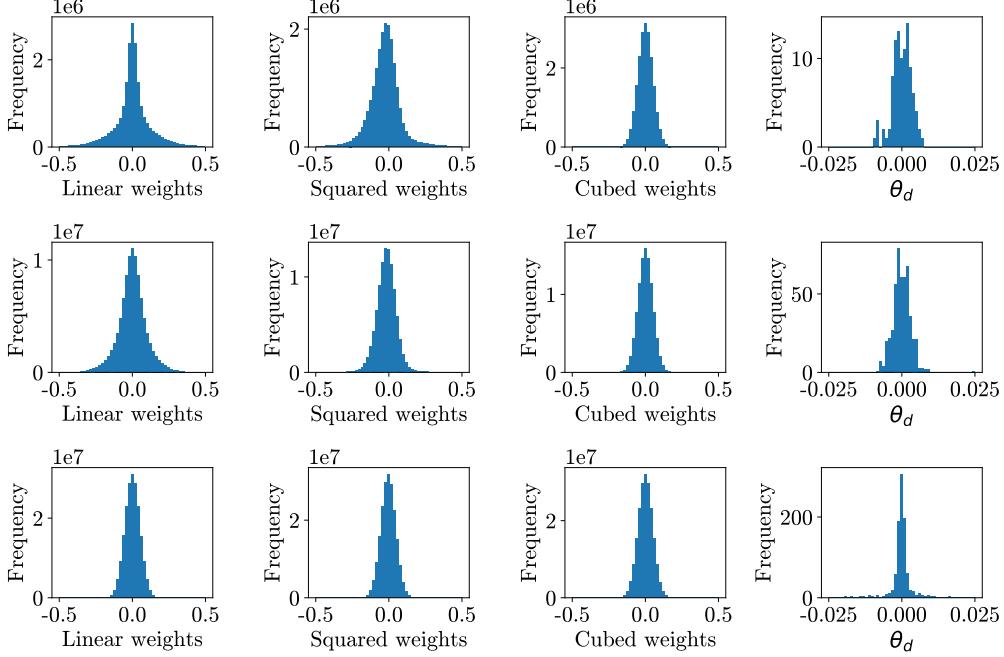


Figure 4: The distributions of weights in the projection matrix for linear, squared, and cubed weights, generated from training with a trainable projection matrix, as well as the distribution of intrinsic weights. The first row is for $d = 100$, the second for $d = 500$, and the last for $d = 1000$.

we notice that the distributions of P are indeed different between the linear, quadratic, and cubic terms; as expected, the coefficients of the cubic terms are generally more closely centered at zero ($\lambda_c < \lambda_l, \lambda_s$). However, what we failed to realize is that the distribution of trained intrinsic weights is also very close to zero, which makes the squared and cubic terms even less significant – this results in the contributions to $\theta^{(D)}$ coming largely from the linear terms in the linear combination. As a result of these two facts, the test accuracies resulting from the trained λ_s and λ_c initializers in Figure 3b show no notable difference from the other power-term-augmented models or the linear projection model.

Using this intuition, we tried increasing the initializer coefficients for the squared and cubed terms rather than decreasing them. With $\lambda_s = \lambda_c = 10$, the results are not considerably different from the other power-term-augmented models. With $\lambda_s = \lambda_c = 100$, then we start to see a breakdown of the accuracy.

This also has an implication for the RFF results. Looking at the intrinsic weight distributions in Figure 4, we note that the variance of the intrinsic weights (the right-most column) is a decreasing function of d . If the variance to the (smooth) nonlinear mappings is small, then the nonlinear sine and cosine functions will be approximately linear. In other words, the smaller variance at high d will reduce the nonlinearity of the mapping; this may explain why the RFF was more advantageous for small d .

4.4 LEARNING RATE PROBLEM

One of the general concerns with adding more operations to a network is that the added complexity may affect convergence. In particular, multiplication by a very large matrix P that multiplies the total weights in the network by a factor of d (during training) adds a large, undesirable complexity to the model; additionally, we are calculating the more unstable gradients of nonlinear functions. We

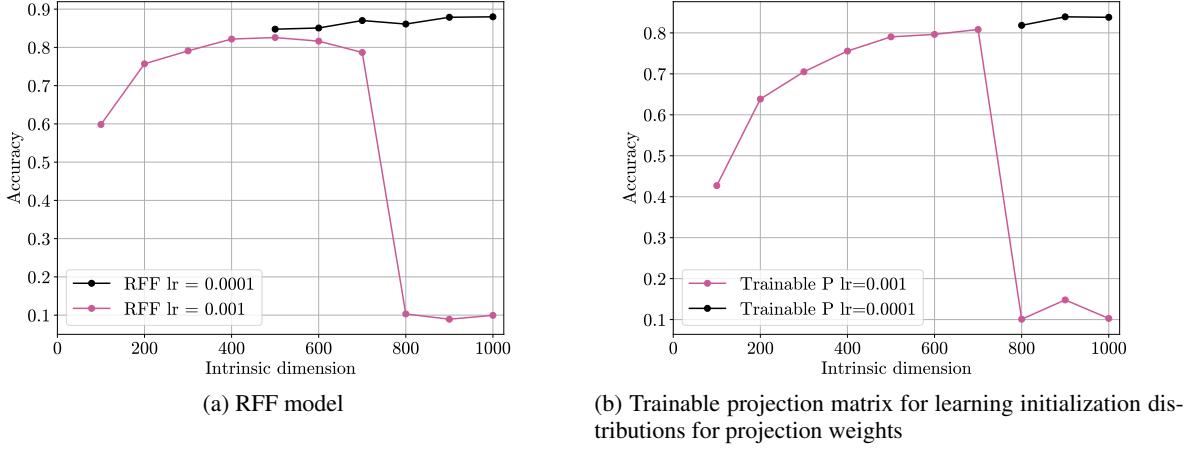


Figure 5: Training accuracy of the two models with a higher ($\alpha = 0.001$) and lower learning rate ($\alpha = 0.0001$). Failure to converge at higher complexities (higher intrinsic dimensions) requires a smaller learning rate.

encountered convergence issues several times during training when the model for higher intrinsic dimensions, such as in the case of the random Fourier features (Figure 5a) and in the case of a trainable projection matrix (Figure 5b). In both cases, decreasing the default learning rate by a factor of 10 ($\alpha \leftarrow 0.0001$) fixed the convergence issue. This learning rate convergence issue does not occur for the direct model for similar learning rates.

A different interpretation for this effect is that by augmenting the intrinsic weights matrix with nonlinear mappings of itself, the gradient updates for each intrinsic weight $\theta_i^{(d)}$ become more complicated as it has more dependencies in the computational graph, which can be thought of loosely as “constraints” during backpropagation. Trying to meet three times as many constraints for either augmented model may be a cause for slower convergence and failure to converge at higher intrinsic dimensions.

Given that MNIST is a small dataset, this is a concerning issue that should be experimented on with larger datasets. We are unsure if this problem is still present if sparse or Fastfood linear projections are used in the place of dense linear projections.

4.5 NORMALIZING PROJECTION OUTPUT BASIS VECTORS

The projection matrix P can be interpreted as the orthogonal basis of an output space. Li et al. describe normalizing the columns of P to form an orthonormal basis to make the linear projection isometric (i.e., distance-preserving).

We make the case that this distance preservation, and thus the normalization, will have little effect on accuracy. The reasoning for this is that a set of random basis vectors should have roughly equal length, and thus this normalization is roughly equivalent to scaling each column of P down by some constant average length. This will then only have the effect of scaling P ’s variance down, which is inconsequential to a linear operation such as projection (it can be counteracted by learning the intrinsic weights to be larger by an equal factor).

In a departure from the procedure described in Li et al. (2018), we began by not normalizing the columns of the projection matrix, opting to run the linear, power, and RFF projections with the randomly generated data. We then tried normalizing the columns as suggested in the paper, leading to the results shown in (Figure 6). There is no noticeable difference between these results and the results when the projection matrix is not normalized, confirming our hypothesis that the normalization step is inconsequential.

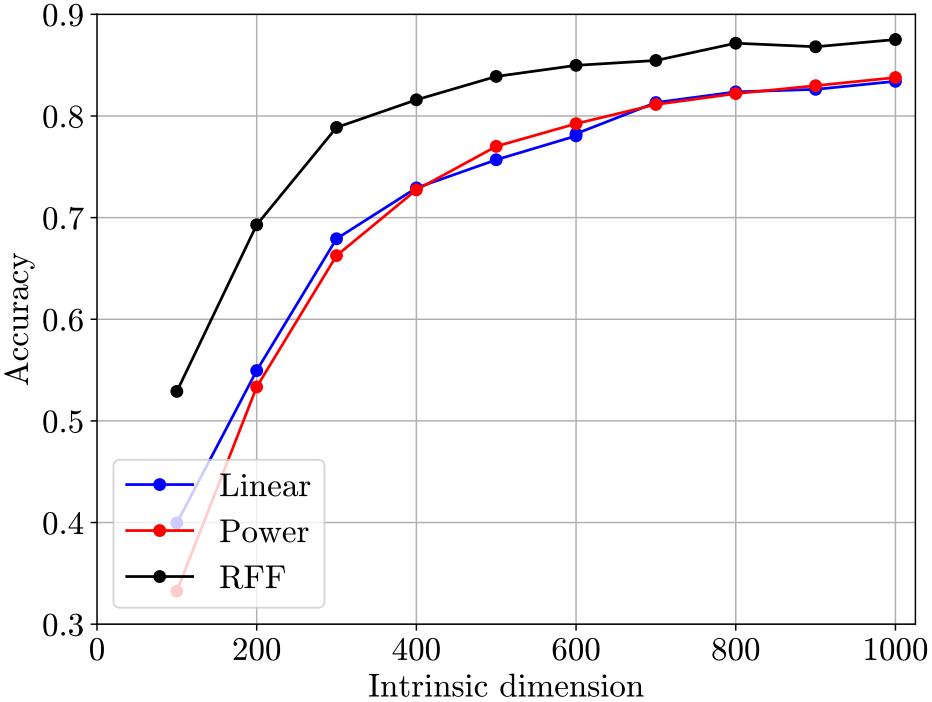


Figure 6: Accuracy vs. intrinsic dimensions for the projection types when the projection matrices are normalized. This is no appreciable difference from the results with a non-normalized projection matrix.

5 CONCLUSIONS AND FURTHER RESEARCH

We experimented with augmenting the method of intrinsic weights proposed in Li et al. (2018) by concatenating nonlinear-mapped features to the set of intrinsic weights. We use power terms (square and cubed terms) and a random Fourier feature mapping for the nonlinear mappings. Despite various attempts at improving the initialization of the projection matrix for the power terms, they did not considerably improve (or worsen) the accuracy as compared to the random linear projection. The RFF-augmented model does provide a noticeable improvement of 3-10% accuracy for all tested intrinsic dimensions. We attribute the improvement gain of the RFF to the fact that the sinusoids (especially the cosine function) increase variance in the small intrinsic weights, and the insignificant effect of the power terms to the fact that squaring or cubing the small weights results in inconsequential terms.

While most of Li et al.’s paper and this paper focus on the theoretical aspects of intrinsic dimension, some practical concerns should be raised. First of all, this method increases computational and memory complexity, especially in the case of dense projection matrices (as is the case of our experiments); while this may be mitigated using the Fastfood transform or sparse projection matrices, there is still a significant overhead over the underlying network architecture. Additionally, there is the issue of bad convergence on higher intrinsic dimensions. These factors make it a poor choice of a methodology for any practical deployment on resource-limited systems, and more of a theoretical toy aimed at comparing the difficulties of problems (e.g., comparing CIFAR-10 to MNIST) and architecture types (e.g., FC versus convolutional networks for image classification). However, it may be interesting to attempt to combine this with some existing compression technique, i.e., to find a way in which the intrinsic dimension can directly influence some aspect of model design.

There are many additional directions in which one could explore. For example, we limited ourselves to a single dataset and network architecture for the sake of simplicity, reproducibility, and practi-

cally completing the research in the given timeframe. It may be informative to test if these results extend to similar problems (e.g., CIFAR-10), different problems (e.g., generative models), much larger architectures, convolutional architectures, recurrent networks, residual networks (i.e., with skip connections), etc. It may also be interesting to use this method to “probe” network and layer architectures: one might intuit that the “factored convolution” used in the MobileNets architecture from Howard et al. (2017) might contain much less redundant layer than a corresponding ordinary convolution, and thus only able to represent problems with a lower intrinsic dimension; using this intrinsic dimension method may be a way to confirm this suspicion.

REFERENCES

- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
- Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017. URL <http://arxiv.org/abs/1710.09282>.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL <http://arxiv.org/abs/1502.01852>.
- Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.
- Did (<https://math.stackexchange.com/users/6179/did>). Why are randomly drawn vectors nearly perpendicular in high dimensions. Mathematics Stack Exchange, 2018. URL <https://math.stackexchange.com/q/995678>. URL: <https://math.stackexchange.com/q/995678> (version: 2018-05-15).
- Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.
- Zhu Li, Jean-Francois Ton, Dino Oglic, and Dino Sejdinovic. Towards a unified analysis of random fourier features, 2019.
- Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS’07, pp. 1177–1184, Red Hook, NY, USA, 2007. Curran Associates Inc. ISBN 9781605603520.
- Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.
- Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. Do deep convolutional nets really need to be deep and convolutional?, 2017.

A APPENDIX: NOTES ON IMPLEMENTATION

A.1 LIBRARIES, HARDWARE, AND SOURCE CODE

The Tensorflow Keras library is used for general deep learning utilities. For each model, we create an instance of the `IntrinsicWeights` class, which is a semantic wrapper around a trainable `tf.Variable` instance. We then implement a `WeightCreator` class, which has utilities to return the calculated weights $\theta^{(D)}$ given a set of intrinsic weights and a projection type. This in turn is used in custom Keras layers, which are implemented very similarly to the ordinary non-projection (“direct”) layers in the `tf.keras.layers` package, with the exception that the `add_weight` method is overloaded to use generate weights using a `WeightCreator` instance.

Experiments were performed on the Cooper Union Kahan deep learning server (primarily using GTX Titan X’s). Roughly 200 GPU-hours of training were used for the experiments in this paper. The source code for our experiments can be found at <https://github.com/jlam55555/intrinsic-dimension-projections>, which was in turn heavily based on the source code from Li’s paper, which can be found at <https://github.com/uber-research/intrinsic-dimension>.

A.2 DIFFICULTIES WITH THE TENSORFLOW LIBRARY

The first step in experimentation was updating the software implementation used in Li et al. (2018) to a more modern stack. (Their implementation uses Tensorflow 1 and Python 2.7; we attempt to upgrade the stack to Tensorflow 2 and Python 3.7.) The main reason for reimplementing the entire research was to gain a better understanding of the methods, and to try to independently reproduce the original results; a secondary reason is that Python 2.7 is soon to be deprecated, and Tensorflow 2 is more modern and offers a very different paradigm to similar tasks than Tensorflow 1.

There were a number of issues regarding the upgrade. In particular, TF2 uses eager evaluation by default, which is not the case with TF1. Since the weights $\theta^{(D)}$ is based on a calculation (and thus $\theta^{(D)}$ is a computational graph node rather than a leaf `tf.Variable` instance), this is actually easier to perform in the older version of Tensorflow. We had to explicitly declare the delayed computation of $\theta^{(D)}$ using `tf.Variable` instances, which is bulkier than passing around the computation tensor itself.

Another issue we encountered was that it appears that the TF2 Keras API requires that all trainable weights (all trainable `tf.Variable` instances) be instance members of the layer class; otherwise, even if they are part of the computational graph for the loss, they do not receive the gradient update rule. This results in some messier code.

We encountered an out-of-memory (OOM) error when running any model in a training loop, which appears to be an unfixed error in the Tensorflow library due to model VRAM not being garbage collected. A workaround that worked for us² was to run models in separate processes; the process clean-up sequence appears to successfully handle the memory freeing.

A.3 SIMILARITIES TO AND DIFFERENCES FROM THE ORIGINAL RESEARCH

Following the conventions of the original paper (and of linear algebra), we denoted the projection matrix as a $D \times d$ matrix, and the intrinsic weights $\theta^{(d)}$ and full network weights $\theta^{(D)}$ as length D and length d column vectors, respectively. However, following the implementation in their codebase, the projection is a $d \times D$ matrix, the weights are row vectors, and the order of the matrix multiplication is swapped. There is no functional difference here, but it may be confusing at first sight.

We had difficulty implementing the sparse and Fastfood implementations that were used in the original paper, and thus chose to omit them from any of our final results. With more time, we would hope to be able to include these in our analysis.

²<https://github.com/tensorflow/tensorflow/issues/36465#issuecomment-582749350>

Searching for a more minimal intrinsic dimension of objective landscapes¹

Jonathan Lam & Richard Lee

December 17, 2020

¹<https://github.com/jlam5555/intrinsic-dimension-projections>

“Measuring the intrinsic dimension of objective landscapes”²

- ▶ Objective landscape (combination of learning problem + network architecture)
- ▶ Defines concept of “intrinsic weights”
- ▶ Proposed method of finding intrinsic weight of objective landscape by method of random linearly-projected weights
- ▶ Method to approximate minimum description length (MDL); can be used for model compression
- ▶ **Our goal:** to find a method to describe an objective landscape with even fewer weights

²Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. *Measuring the intrinsic dimension of objective landscapes*. In International Conference on Learning Representations, 2018.

Method of random linearly-projected weights

$$\theta^{(D)} = \theta_0^{(D)} + P \times \theta^{(d)}$$

The diagram illustrates the decomposition of a weight matrix $\theta^{(D)}$ into three components. On the left, a vertical rectangle labeled "Store Seed" contains seven horizontal lines. In the middle, another vertical rectangle labeled "Store Seed" also contains seven horizontal lines. To the right of these two rectangles is a large square grid labeled "P" with four columns and seven rows. To the right of the grid is a vertical rectangle labeled "Augmentation?" containing four horizontal lines. The entire equation is separated by plus and multiplication signs.

Notation

- ▶ $\theta^{(D)}$: ordinary network weights; not stored as a `tf.Variable`, but rather the result of this calculation
- ▶ $\theta_0^{(D)}$: “base initialization weights” – like an initial bias; randomly initialized and non-trainable
- ▶ P : projection matrix; randomly initialized and non-trainable
- ▶ $\theta^{(d)}$: intrinsic weights; randomly initialized and trainable

Augmenting $\theta^{(d)}$ with squared terms

$$\theta^{(D)} = \theta_0^{(D)} + P \times \left(\theta^{(d)} \oplus (\theta^{(d)})^2 \right)$$

The diagram illustrates the augmentation of a vector $\theta^{(d)}$ with squared terms. It shows the addition of a constant term $\theta_0^{(D)}$ to the product of a matrix P and a vector containing $\theta^{(d)}$ and its square $(\theta^{(d)})^2$. The result is $\theta^{(D)}$.

Below the diagram, the equation is expanded:

$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ (\theta^{(d)})^2 \end{bmatrix}$$

What are random Fourier features (RFFs)?

RFFs are a nonlinear many-to-many mapping that can be used to help capture different frequency components.

$$\gamma(\vec{v}) = \begin{bmatrix} a_1 \cos(2\pi \vec{b}_1^T \vec{v}) \\ a_1 \sin(2\pi \vec{b}_1^T \vec{v}) \\ a_2 \cos(2\pi \vec{b}_2^T \vec{v}) \\ a_2 \sin(2\pi \vec{b}_2^T \vec{v}) \\ \vdots \\ a_m \cos(2\pi \vec{b}_M^T \vec{v}) \\ a_m \sin(2\pi \vec{b}_M^T \vec{v}) \end{bmatrix}$$

We can append these to our intrinsic weights again:

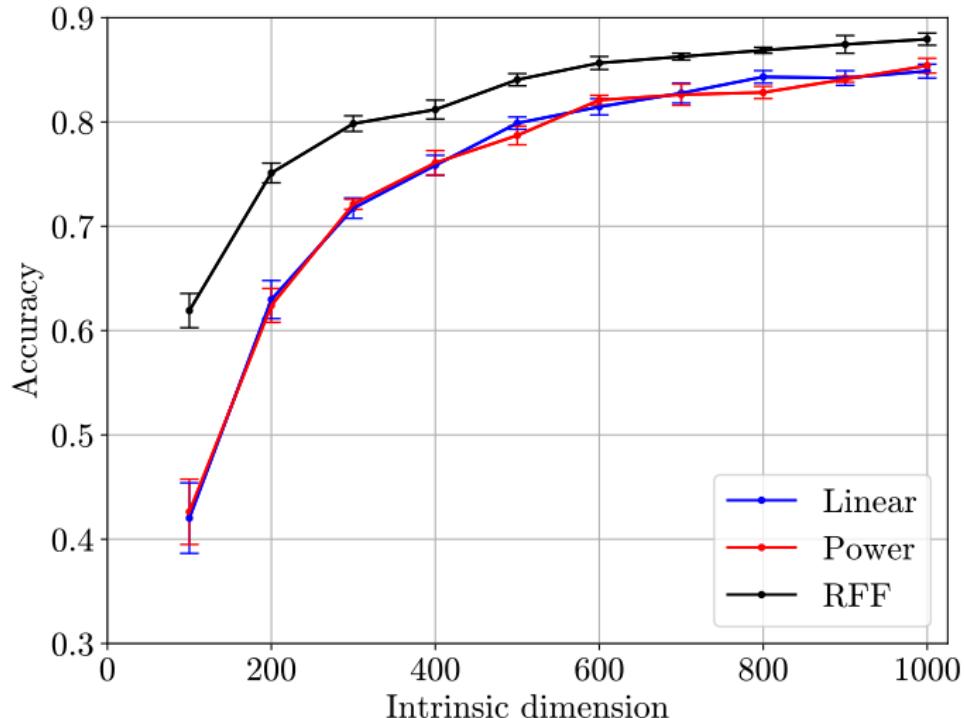
$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ \cos(B\theta^{(d)}) \\ \sin(B\theta^{(d)}) \end{bmatrix}$$

Example: Augmenting $\theta^{(d)}$ with RFF terms

$$\theta^{(D)} = \theta_0^{(D)} + P \begin{bmatrix} \theta^{(d)} \\ \cos(B\theta^{(d)}) \end{bmatrix}$$

The diagram illustrates the decomposition of a vector $\theta^{(D)}$ into a sum of two components. On the left, $\theta^{(D)}$ is shown as a vertical vector of height D . To its right is an equals sign. Next is $\theta_0^{(D)}$, also a vertical vector of height D . To its right is a plus sign. Following the plus sign is a large grid labeled P , which is D rows by M columns. To the right of P is a multiplication sign (\times). Below the multiplication sign is a plus sign (\oplus). To the right of the plus sign is a vertical vector labeled $\theta^{(d)}$. A curved arrow points from this vector to the bottom row of the grid P . Below the grid P is another vertical vector labeled $\cos(B\theta^{(d)})$. A curved arrow points from this vector to the second row of the grid P .

Augmenting $\theta^{(d)}$ with power, RFF terms

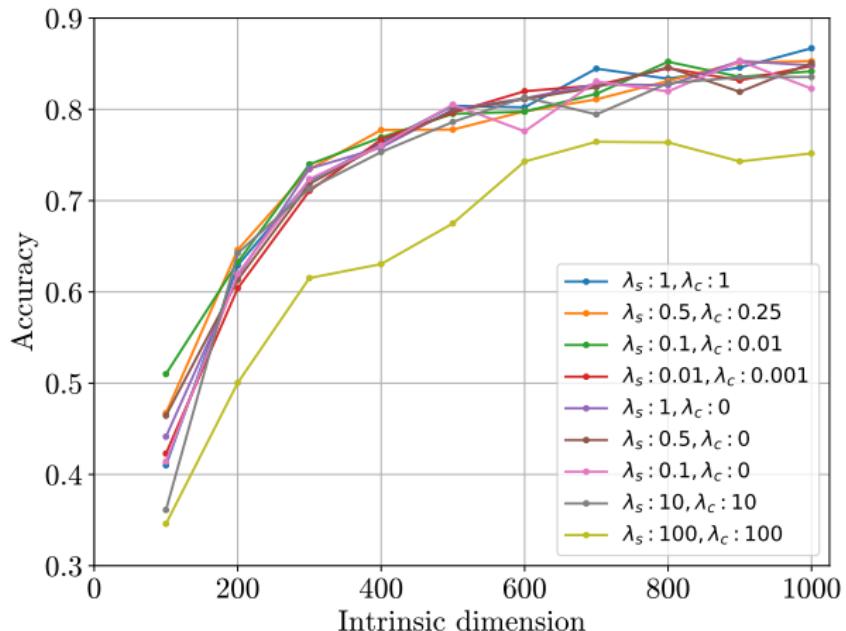


Varying initialization of P : motivation

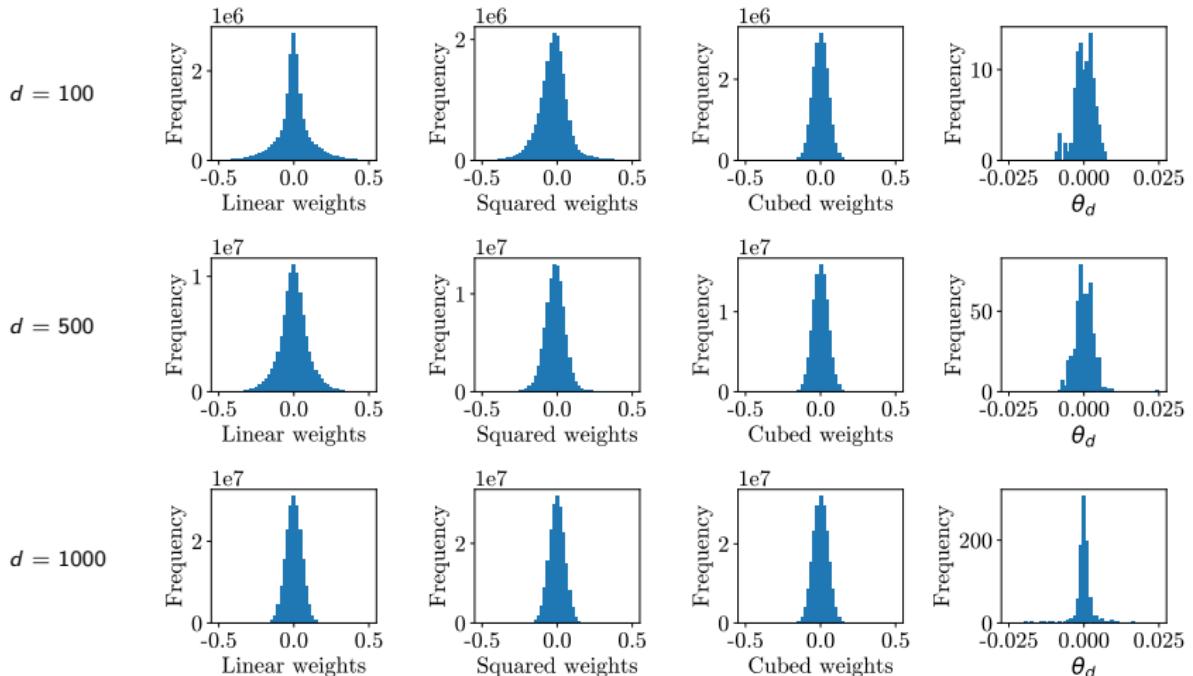
$$\theta^{(D)} = \theta_0^{(D)} + P \begin{bmatrix} \theta^{(d)} \\ \lambda_s (\theta^{(d)})^2 \end{bmatrix}$$

The diagram illustrates the decomposition of the final parameter vector $\theta^{(D)}$. It is shown as the sum of an initial value $\theta_0^{(D)}$ and a product of a matrix P and a vector. The matrix P is partitioned into two parts: "weights for $\theta^{(d)}$ " (blue) and "weights for $(\theta^{(d)})^2$ " (orange). The vector being multiplied by P is labeled $\begin{bmatrix} \theta^{(d)} \\ \lambda_s (\theta^{(d)})^2 \end{bmatrix}$. This vector is generated by a process involving a multiplication by x^2 , an addition (\oplus), and a scaling by λ_s .

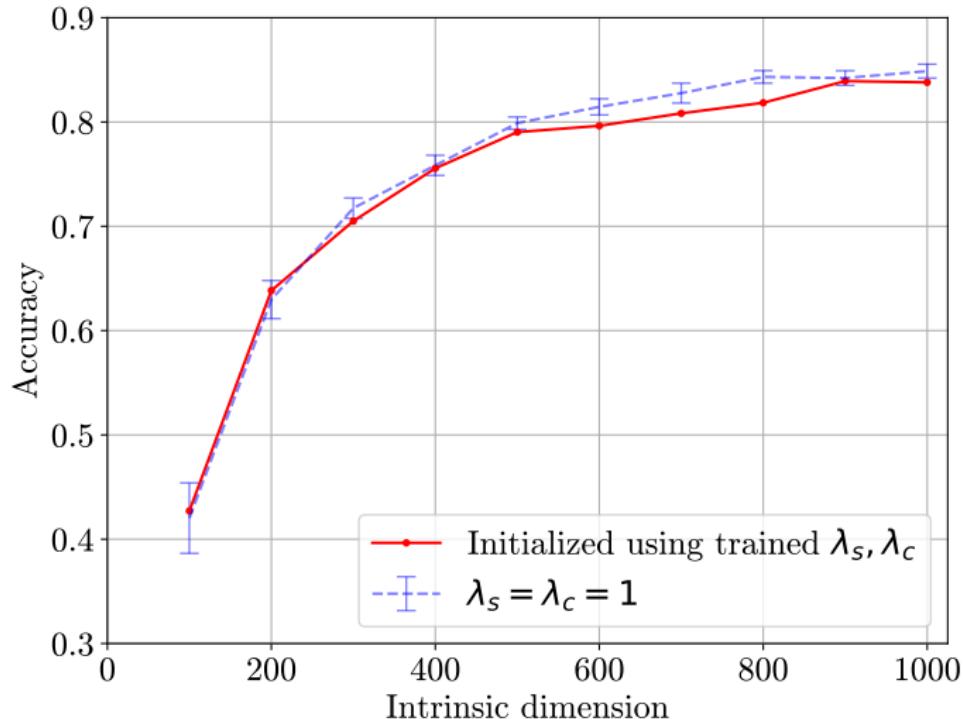
Varying the initialization of P



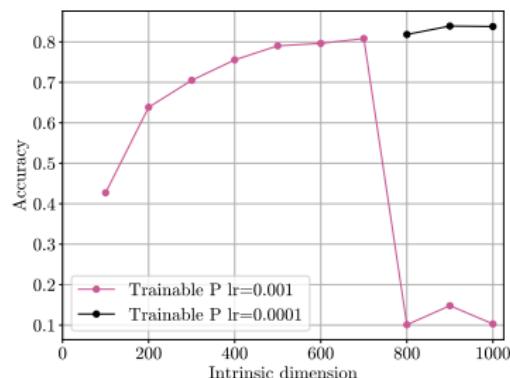
Trained P and $\theta^{(d)}$ weights



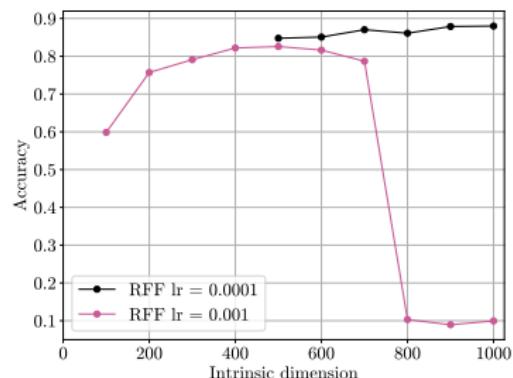
Initializing P with trained distributions



Bad convergence → decreased learning rates

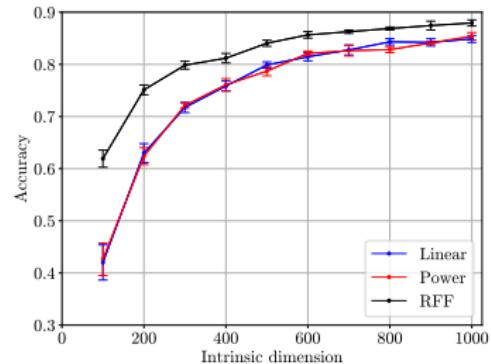


(a) Trainable Projection Matrix

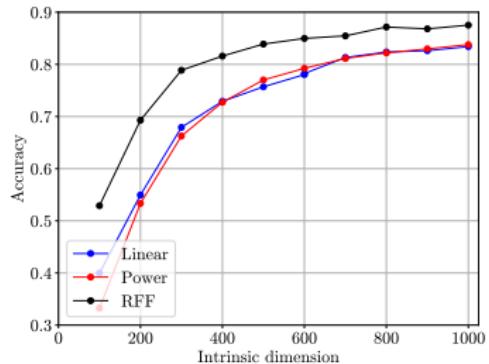


(b) Random Fourier Augmentation

Normalize P ?



(c) Non-normalized Projection Matrix



(d) Normalized Projection Matrix

Conclusions and future research

- ▶ RFF > linear \approx power terms
- ▶ Still have a lot to try: different data, larger models, different layer types, etc.
- ▶ Compression? Practicality?

Reproducing MobileNets on CIFAR-10

ECE472 Midterm Project
Richard Lee & Jonathan Lam

October 29, 2020

1 Introduction

1.1 Project Goal

We aim to independently reproduce some of the results from the original MobileNets paper [1]. The assignment states to recreate the results from the paper alone, without using guidance from an existing implementation of MobileNets. The goal is not to reproduce the exact results, or to achieve the highest possible performance, but to realize similar patterns of results as the Mobilenets authors in their original research.

Much of our time spent on this project was in adapting the architecture from the original ImageNet-centric architecture to one designed for CIFAR-10, and those design challenges are discussed in this report. Following the original setup, we were able to reproduce most of the desired results (Figures 4 and 5 from [1]).

1.2 Reproduced work

In our project, we:

1. Used the principles of MobileNets to produce a more feasible smaller-scale version to test on CIFAR-10
2. Explained the equations for parameter count and computational cost presented in the MobileNets paper
3. Reproduced Figures 4 and 5 from the original MobileNets paper

Most of the results and architecture in the original MobileNets paper was based on ImageNet, which was outside the limits of our hardware (see Hardware and time constraints).

2 Overview of MobileNets

MobileNets is a deep neural network architecture that is aimed toward reducing computational cost and parameter count, which is highly desirable in the world of IoT and embedded devices. It does this by using “depthwise separable convolutions” (first introduced in [2]), which combines a

“depthwise convolution” layer and an ordinary 1×1 convolutional layer; this mimicks an ordinary convolutional layer but requires many fewer computations without compromising much cost.

The MobileNets authors also created two high-level hyperparameters to generally tune the network, which can be useful for matching hardware requirements. The first hyperparameter, α , is the width multiplier, which is a multiplicative factor of the number of input/output channels of the layers. The second hyperparameter, ρ , is the resolution multiplier; by changing the resolution of the input images, the number of computations can also be reduced. (This hyperparameter is already implicitly set by the input dimensions, relative to some set baseline resolution; i.e., changing the input resolution of the network naturally affects the number of computations.)

The authors show that α is proportional to the square of the number of parameters and computational cost, and ρ is proportional to the square of the computational cost only. They state the general relationship between these two hyperparameters and the number of weights, as well as the relationship between these hyperparameters and the computational cost, and they provide a high-level intuition as justification. In Appendix I, we delve deeper into how the number of weights and computational cost are quantified, calculated theoretically, and calculated empirically.

3 Methodology

3.1 Hardware and time constraints

Our results were run on a combination of Google Colab with GPU accelerators as well as a local machine with a single GPU. The original MobileNets paper focused on large models such as ImageNet, but given the time constraints for this midterm project, we chose to focus on CIFAR-10 for our testing, as it is another commonly used baseline and can train in a reasonable amount of time on our limited computational power. (For example, even with CIFAR-10 we ran out of free GPU usage on Colab, so training on CIFAR-100 or larger problems would likely be infeasible.)

3.2 Designing the baseline model

The MobileNets authors listed the structure of their MobileNets architecture for ImageNet in the paper. However, ImageNet images have a resolution of 224×224 , while CIFAR-10 has a resolution of 32×32 : there is a huge difference in the scale of the problem. ImageNet also has 1000 output classes; CIFAR-10 only has 10.

We cannot use this architecture as-is for CIFAR; if we were to keep the depth of the network (and $\rho = 1$), we would have to modify the strides because of the difference in dimensions. The MobileNets architecture for ImageNet, through the use of strides, halves the image dimensions multiple times ($224 \times 224 \rightarrow 112 \times 112 \rightarrow 56 \times 56 \rightarrow 28 \times 28 \rightarrow 14 \times 14 \rightarrow 7 \times 7$) before doing a final 7×7 average pooling layer. Each time it halves the image dimensions in the



“Jonathan Lam, circa 3AM some days before project deadline, 2020, colorized.” Richard Lee, 2020.

depthwise convolution layer by using stride 2, it also doubles the number of feature channels in the corresponding 1×1 convolution.

We implemented this model and called it `full_mobilenet`. It is still based around 224×224 resolution images. The only major change is that dropout was added to every MobileNet block (detailed in the regularization section) and before the final dense layer, and that the final dense layer was set to have 10 outputs (because CIFAR-10 has 10 classes). Since the final dense layer has 1024 input features, this reduces the number of weights from the original model greatly (1024×1000 to 1024×10 means a reduction of over one million weights). With 1000 classes, our model has 4,179,920 trainable weights, which matches the reported value of 4.2 million weights. Reducing this to 10 classes reduces the number of trainable parameters to 3,188,930 trainable weights.

Using the vanilla MobileNet model for ImageNet, there are two straightforward ways to adapt this to our model. The first way is to more-or-less keep the same network, but only perform strides in the latter part of the network. This means that the image feature map stays at 32×32 until the ImageNet-centric architecture reduces the image dimensions to a smaller value. While viable, this method seems to contradict the design principles of creating an “efficient and streamlined” architecture for images classification. Because of this, and due to time constraints, we did not perform testing with this modification to the architecture.

An alternative way to use the vanilla model is to set the resolution multiplier ρ to $1/7$ (i.e., $32 = 224/7$). This reduces the input image resolution as well as the internal layer resolutions by 7, which aligns the ImageNet-centric architecture to 32×32 input images. We tested this model briefly, and present the results in Section 4.3.

However, it is probably clear that both of these methods, which only slightly modify a model designed for ImageNet, are probably overkill for CIFAR-10. Thus, we tried to design a simpler model from the using the building blocks and starting the ground up. The MobileNets authors do not specify how they decided on the details of their architecture (e.g., number of layers, widths, when to perform stride 2, etc.) so we attempt to mimic the overall design principles, namely utilizing the depthwise convolutions to significantly reduce required computational power and time.

We began by implementing the basic MobileNet block, which is shown in Figure 1. This block

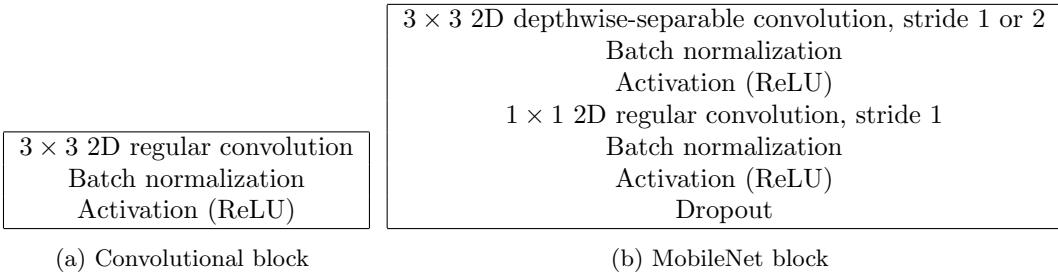


Figure 1: Comparison between our implementations of regular convolutional and MobileNet blocks

follows the structure presented by the authors of the MobileNet paper, with the exception of an additional dropout layer that we added to prevent overfitting (see 3.2.2). Using this basic MobileNet block, we created two models, CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2.

Both models have a regular convolution as the first layer, followed by 5 MobileNet blocks, followed by an average pooling, a dropout, and a final softmax layer for classification. What differs between V1 and V2 is the number filters (width) of each MobileNet block. Specifically, V1 uses, in

order, 64, 128, 256, 512, 1024 filters for the 5 MobileNet Blocks, while V2 uses, 32, 32, 64, 128, 256 filters. V1 is slightly closer to using the widths of the original MobileNets architecture, while V2 is much smaller (number of parameters and computational cost) while not performing significantly worse in accuracy. V2 also follows the original design a little closer, in which the number of output channels is only increased iff the depthwise convolution has stride 2, which keeps the number of parameters low. Note that the strides in a MobileNet block refer to the depthwise convolution layer (the 1×1 convolutions always have stride 1). See Appendix II for the source code implementations.

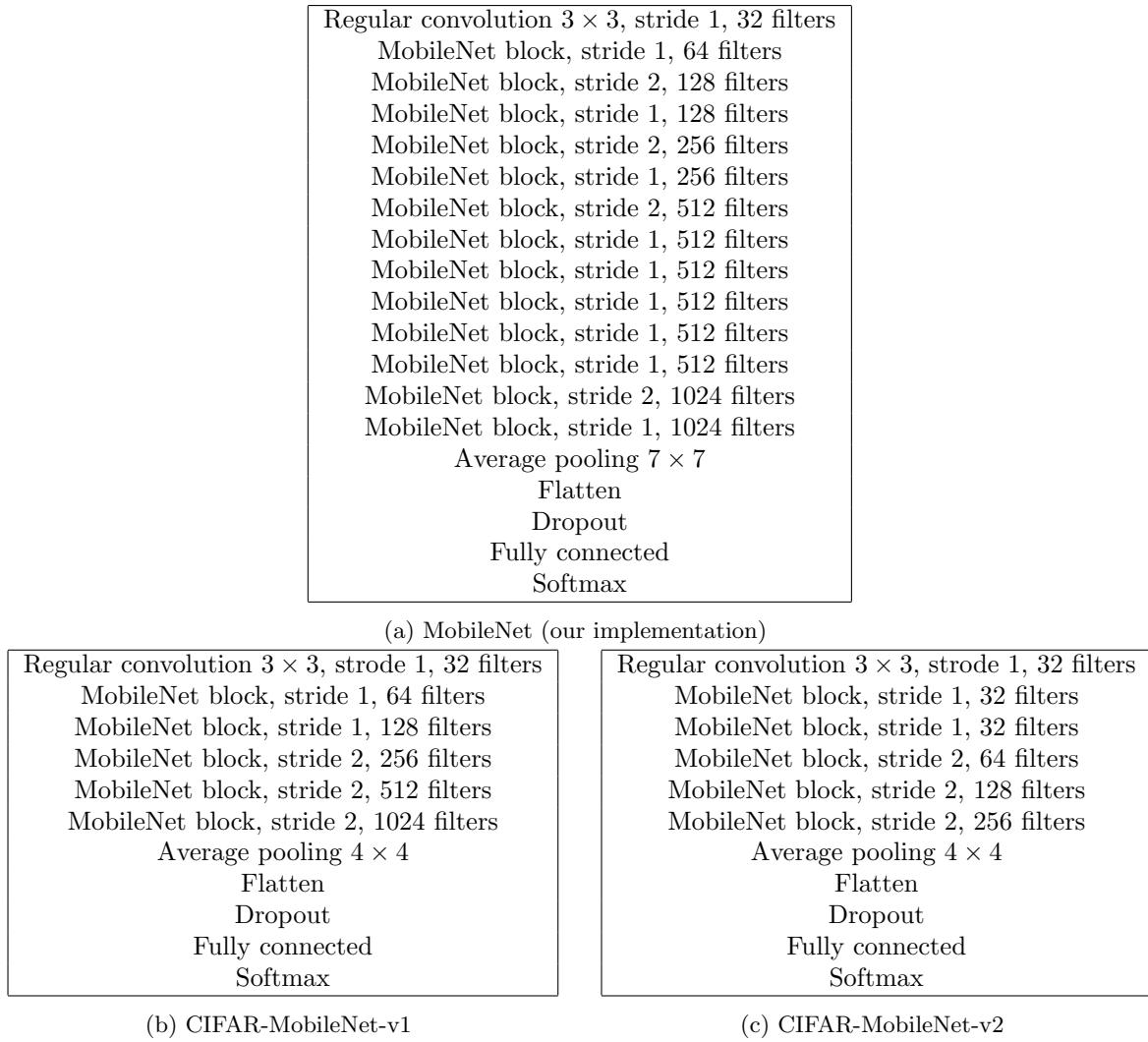


Figure 2: Comparison between the different architectures we implemented

3.2.1 Validation

We randomly split a portion of the training set (20%) for a validation set to provide early stopping, but we did not do any hyperparameter tuning through validation.

The goal of this project is not to produce the highest-scoring results on CIFAR-10, but rather to more-or-less recreate the general patterns that the authors of MobileNets observed. Our baseline model just needs to produce sufficiently decent results on CIFAR-10 so that we know it works. Finding optimal hyperparameters (e.g., number of MobileNets blocks, best width multipliers, best depth multipliers, best layers to perform stride 2 convolutions, etc.) for a MobileNets-based architecture on CIFAR-10 was beyond the scope of this project and open for further research.

3.2.2 Regularization

The MobileNets authors mention that explicit regularization (e.g., through dropout and L1/L2-regularization) is not necessary because a simpler model tends not to overfit. While MobileNets may be considered a relatively simple model for ImageNet, this is not the case for CIFAR-10, and we observed serious overfitting. Thus, our MobileNet implementations include several dropout layers.

While our professor notes that it is usually strange to include both batch normalization and dropout layers, we included batch normalization to keep in line with the original model, and dropout to enforce stronger regularization.

3.2.3 Initializers and generic model hyperparameters

We let the dense and convolutional layers (both regular and depthwise separable) use the default initializers. At the time of writing, the defaults for all of these layers is the Glorot uniform initialization method.

For all of the models, we used the Adam optimizer (the original paper uses RMSprop) and mostly used the default learning rate 0.001 (except for CIFAR-MobileNet-v2, in which we saw a lot of instability in the training, so the learning rate was reduced to 0.0001).

The dropout values were not chosen by any strict validation scheme; the dropout rate of 0.2 was somewhat-arbitrarily chosen, and it seemed to work well regularizing most of our models.

3.2.4 Image preprocessing

As with regularization, the authors of MobileNets stated that image augmentation was unnecessary because the smaller model architecture did not have trouble with overfitting. To simplify the process of training many models, we did not perform image augmentation. This provided decent accuracy, and it did not seem to introduce problems with overfitting (as omitting regularization might).

The only preprocessing we performed was a min-max scaling to [0, 1] (i.e., dividing pixel values by 255).

3.3 Implementing the MobileNet hyperparameters

The MobileNet authors use the following notation to indicate a given model: “ α MobileNet-resolution”. (The default implementation, with $\alpha = 1$ and $\rho = 1$ (full 224×224 pixel resolution is denoted 1.0 MobileNet-224). We follow this convention as well. To be overly explicit, we will use the name “CIFAR-MobileNet” when referring to our reduced MobileNet structure, although it should be clear from the smaller resolution.

To implement α , we had to multiply the 1.0 CIFAR-MobileNet-32 layer widths by α (i.e. the number filters/output channels). This was relatively straightforward, so long as we were careful that the number of filters rounded to an integral number. We tested α values 0.5, 0.75, 0.8, 0.9, 1, and 2.

To implement ρ , the changes that have to be made are:

- The images have to be rescaled during preprocessing (we used `tensorflow.image.resize` to accomplish this).
- The input layer of the network has to expect this new image size.
- The final average pooling layer pool size has to be scaled down by ρ . This value is rounded to an integer, and for any meaningful output, must be at least one.

With that in mind, we chose ρ values of 16/32, 20/32, 24/32, 28/32, 30/32, and 1. The common denominator of 32 was chosen to align with the 32 pixel image size of CIFAR-10, so the numerators (16, 20, 24, 28, 30, 32) describe the input image sizes.

4 Results and discussion

4.1 Verifying the paper’s results for the full-size model

Since our goal is to reproduce some of the results of the original MobileNets paper, we attempted to provide some verification that our model’s properties matched those stated in the paper. In particular, Table 4 in the original paper give a summary of the mult-adds and millions of parameters (weights) for the 1.0-MobileNet-224 model and the equivalent model with regular convolutional blocks. To have a fair comparison, we ran the profiler to get the number of million mult-adds and trainable weights from `full_mobilenet` with a 1000-width final dense layer (which was intended to be exactly the same model as what the authors described); that same network with regular convolutional blocks rather than MobileNet blocks; and the provided implementation of MobileNets in Keras. `tf.keras.applications.MobileNet`. The results are displayed in Table 1. Explanation of the calculations of mult-adds and parameters, as well as the profiling code is given in Appendix I. We note that our value of “mult-adds” is given by the approximation that the number of mult-adds is half the number of FLOPs (Appendix I), which means that we are slightly overcounting number of mult-adds in our empirical profiling calculations. We get complete agreement with the number

	Million Mult-Adds Reported	Million Mult-Adds Empirical	Million Parameters Reported	Million Parameters Empirical
Conv MobileNet	4866	4898	22.9	22.9
MobileNet (Keras) MobileNet	569	576	4.2	4.2

Table 1: Comparison of MobileNet empirically-calculated mult-adds and parameters to the values reported in the paper. Conv MobileNet is `full_mobilenet` with `use_mobilenet_blocks=False, alpha=1, rho=1`; MobileNet is `full_mobilenet` with `use_mobilenet_blocks=True, alpha=1, rho=1`; and Keras MobileNet is the `tf.keras.applications.MobileNet` implementation. Empirical values were computed using the `tf.compat.v1.profiler` profilers.

of reported and empirical million parameters (to the number of significant figures reported by the authors).

4.1.1 Fidelity of our model to the original model

We also compared our results against the Keras MobileNet implementation. While our assignment explicitly instructs us not to look at existing implementations of the work published in the research paper for guidance, we thought that using this would be an interesting way to gauge our model’s fidelity to what the authors described. This is the only place we used an existing MobileNet implementation. We only used it to look at profiling and layer summary details and did not look at the source code nor use it to influence our model.

When examining the model summaries of our model versus the builtin implementation’s model summary (i.e., by examining `tf.keras.Model::summary()`), we noticed a few differences. These were only an after-the-fact realization and we didn’t incorporate these changes into our model.

- The initial convolutional layer in the Keras implementation also includes batch normalization and an activation. We misinterpreted the original paper and only have a convolutional layer there, since it is not described like a regular MobileNet block in the MobileNet architecture described in the original paper.
- The keras implementation’s final fully-connected (FC) layer uses a 1×1 convolution operation, while ours uses a dense layer. We don’t believe that there’s any real difference in the operation here, but it’s interesting to see that a fully-connected layer can be done in multiple ways.
- As stated before, we use dropout in every convolutional block. The keras implementation only has a single dropout layer before the final classification step.
- The convolutional layers (both depthwise and regular) in the keras implementation do not have bias terms (`use_bias=False`), while ours do (using the default `use_bias=True`). This slightly reduces the number of weights and computations. We believe that the bias is omitted because the convolutional layers are immediately followed by batch normalization layers, which would introduce a learned bias already. We didn’t think of this when concocting our models.

Despite these (relatively minor) differences, we believe that our implementation of the full MobileNet, and thus the ideas transmitted to the reduced CIFAR MobileNets, are sound.

4.2 Reproducing the effect of hyperparameters

In the original MobileNets paper, the authors had two primary results: comparing MobileNets to existing deep convolutional layers models with regular convolutional layers (e.g., VGG and AlexNet); and varying α and ρ and viewing the relationships between those hyperparameters, their test accuracies, number of parameters, and computational cost. We do not explore the first goal; our primary focus is on the latter.

We tested 72 total models formed from the Cartesian product of $model \in \{V1, V2\}$, $\alpha \in \{0.5, 0.75, 0.8, 0.9, 1, 2\}$, and $\rho \in \{16/32, 20/32, 24/32, 28/32, 30/32, 1\}$. (In contrast, the MobileNets paper uses $\alpha \in \{0.25, 0.5, 0.75, 1\}$ and $\rho \in \{128/228, 160/228, 192/228, 1\}$.) These parameters values were arbitrarily chosen. We chose to test $\alpha = 2$ to see if the additional filters in each MobileNet block would provide any improvement in test accuracy; the original paper only went up to $\alpha = 1$.

(On the other hand, increasing ρ past 1 is not meaningful, as you cannot simply increase resolution to gain new information.)

The following plots and tables describe the relationships between hyperparameters, accuracy, number of trainable weights (i.e., size of the model in memory), and computational cost (multiplied in mult-adds).

4.2.1 Accuracy, number of weights, computational cost vs. α , ρ

The direct effect of hyperparameters on the other three metrics (accuracy, number of parameters, and computational cost) are shown in Tables 2, 3, and 4, respectively. We can make a number of observations from these tables:

- The relationship between α and accuracy, or between ρ and accuracy (Table 2), is a positive direct relationship. This data is noisy relative to what the MobileNets authors observe, as neither relationship is monotonically increasing, but the general trends are still apparent. The monotonic relationship is more noticeable for CIFAR-MobileNet-v2 than for CIFAR-MobileNet-v1.
- The number of parameters is only affected by α (and not ρ). Intuitively, this makes sense as kernel size is independent of feature map resolution. The relationship between number of parameters and α is roughly a squared relationship.
- The computational cost is roughly proportional to the square of both α and ρ , as expected.

4.2.2 Accuracy vs. Number of Trainable Weights

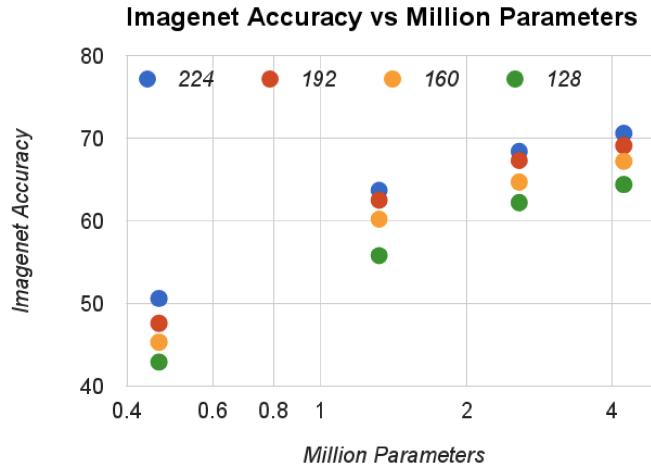


Figure 3: Figure 5 from the original MobileNets paper

Figure 5 (in our report) is a visual representation of Table 2, and plots test accuracies against number of trainable weights. These plots are intended to mirror Figure 5 in the original MobileNets

paper (included here as Figure 3). For both CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2, we notice a general increase in accuracy as the number of weights increased. As noted before, these results are “noisier” than those of the original paper. We attribute this to the inherent randomness in model training, as well as the fact that CIFAR-10 is a much smaller dataset with fewer classes, so we would expect there to be less variation than in that of ImageNet.

It is clear from Figure 5 that changing ρ while keeping α constant doesn’t change the number of parameters, but still changes the accuracy. We also connected the points in the same ρ -series (i.e., points with the same ρ) in order to more easily see how changing α affects ρ -series. We observe that the slopes of the ρ -series in the v2 model are much more consistent and steep than the slopes in the v1 model.

We can also observe that for any fixed value of α , increasing ρ generally tends to increase the accuracy, except for higher α in v1. To address this anomaly, we posit that the v1 models are expressive enough in their vanilla forms ($\alpha \leq 1$ and $\rho \leq 1$) for the CIFAR-10 dataset (i.e. little additional information is added by the extra filters). This shows that for a MobileNet with a given depth, it is important to sweep over α to test what width works best without overfitting.

4.2.3 Accuracy vs. Number of Mult-Adds (Computational Cost)

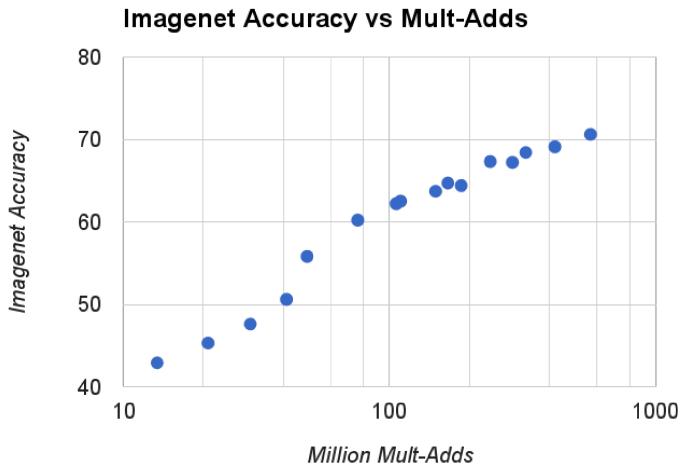


Figure 4: Figure 4 from the original MobileNets paper

In Figure 4 of the original MobileNets paper (included here as Figure 4), they describe the number of mult-adds that each of the models used to achieve the various test accuracies to measure the trade off between computational cost and test accuracy. They found a log-linear relationship between the number of mult-adds and test accuracy.

We created similar plots for CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2 in Figure 6. Once again, the results are noisier than that of the original paper, but a log-linear relationship can still be drawn, especially for v2 - as the computational cost increases, a general increase in test accuracy can be expected. However, with v1, we only noticed a positive log-linear relationship for a majority

of the models. We draw a second log-linear regression line, as the general trend shows that there are negative returns despite increasing the computational cost of the models.

Comparing against Table 4, we can observe that primarily models with the highest α of 2 and 1 also contain the highest computational cost (intuitively, more filters results in more computations). This result supports our previous conclusion that the models in v1 are expressive enough for CIFAR-10 and additional parameters do not provide additional useful information.

We also provide Figure 7, which shows the same data plotted on the same axes as in Figure 6, but with the ρ -series distinguished like in Figure 5. The $\alpha = 1$ and $\alpha = 2$ values in this representation are the two rightmost points on each ρ -series, which we can tell do not have higher accuracies than lower α values within the same ρ -series in the v1 model. Another interesting conclusion that can be drawn from this modified chart is which value of ρ to use to maximize accuracy efficiency (i.e., highest ratio of accuracy to number of mult-adds); this graphically would be indicated by which ρ -series is highest for any given number of mult-adds. However, it is unclear by the plots which ρ -series is best, because the lines overlap often. More research could be done on quantifying this effect.

4.3 Comparing Convolution Types

We noted in Section 3.2 that it would be possible to use the vanilla MobileNets model (designed for a 224×224 resolution image) by using a $\rho = 1/7$, and we present the results in Table ?? below. We did not test out how the hyperparameters affected this model, since it is larger than our CIFAR-specific models, but we ran it with regular convolution blocks and with MobileNet blocks to compare the two.

	MobileNet with regular convolution	MobileNet
Test dataset accuracies (%)	76.32	73.51
Test dataset evaluation time (s)	3.076	1.378
Millions of parameters	56.6	6.43
Millions of mult-adds	28.3	3.23

We note that the model using regular convolutions attains a slightly higher test accuracy than when using depthwise convolutions, but the depthwise convolutional model is able to evaluate the entire test set in less than half the time of the plain convolution model. This supports the authors' claim that depthwise convolutions (and thereby MobileNets) is more computationally efficient for image classification than other conventional models, while still achieving similar accuracy.

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	79.59	82.34	78.59	82.05	79.3	78.47
	1	84.24	79.13	81.83	81.81	78.53	77.54
	0.9	83.54	81.04	82.56	79.57	78.85	78.21
	0.8	84.6	80.05	80.01	80.75	79.05	76.15
	0.75	82.67	81.74	79.81	80.17	78.74	76.52
	0.5	80.42	79.58	79.65	78.56	76.56	72.35

(a) CIFAR-MobileNet-v1

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	76.87	73.39	72.58	76.06	76.69	71.68
	1	72.51	67.35	68.53	68.66	69.87	68.22
	0.9	68.68	71.16	68.28	68.72	66.26	65.96
	0.8	65.61	69.31	68.78	64.64	65.28	64.13
	0.75	66.89	66.24	66.81	61.7	63.93	65.75
	0.5	63.45	64.12	60.87	61.24	61.76	59.73

(b) CIFAR-MobileNet-v2

Table 2: Test accuracy (%) w.r.t. ρ and α .

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2		2851466				
	1		727370				
	0.9		590014				
	0.8		468839				
	0.75		414586				
	0.5		189098				

(a) CIFAR-MobileNet-v1

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2			197130			
	1			53514			
	0.9			43709			
	0.8			35242			
	0.75			31690			
	0.5			15498			

(b) CIFAR-MobileNet-v2

Table 3: Number of model parameters w.r.t. ρ and α . (ρ is independent of model parameter count.) Values were empirically determined using the `tf.compat.v1.profiler` profiler.

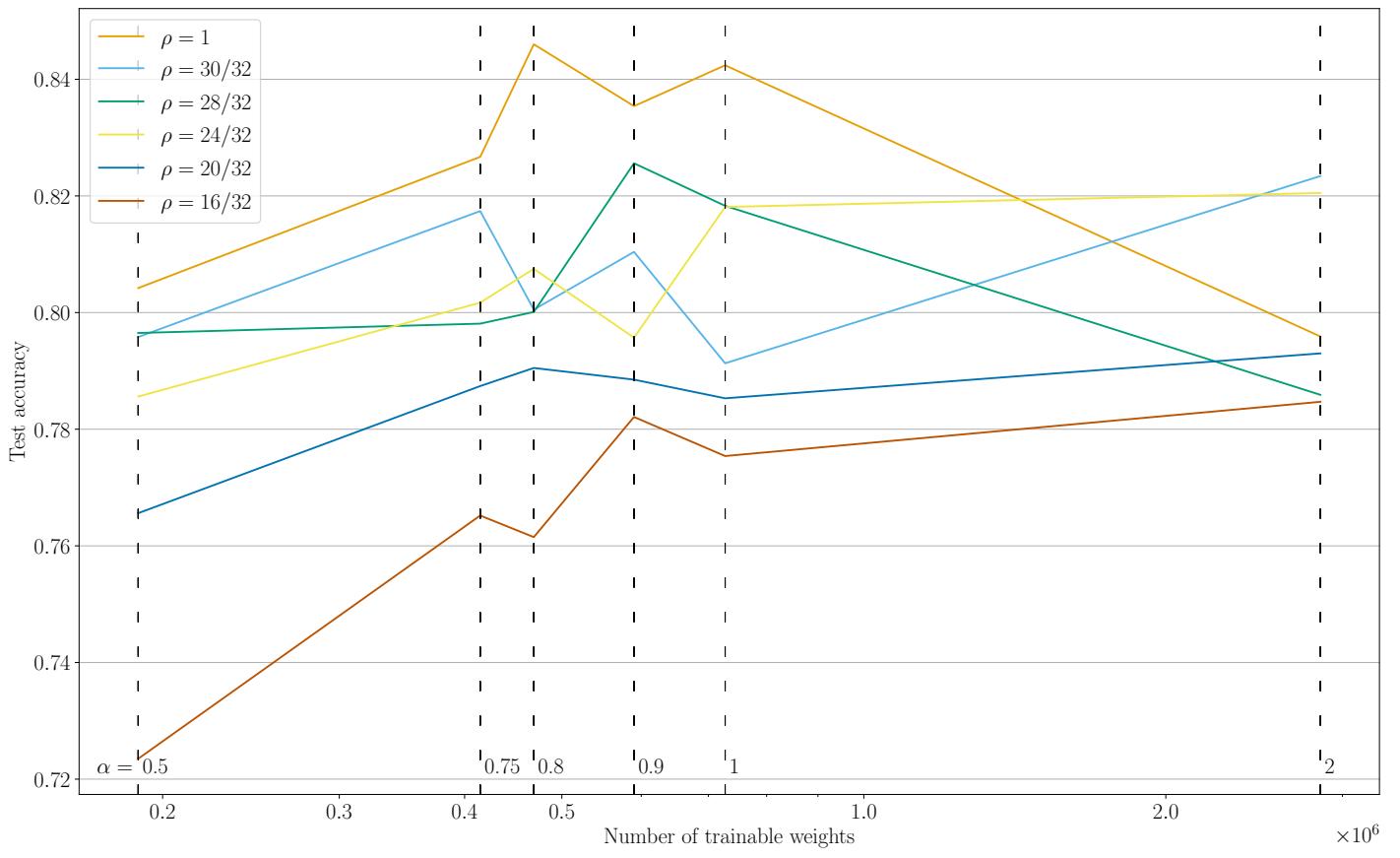
		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	154.64	144.93	127.92	90.02	70.43	43.86
	1	40.98	38.41	34.01	24.27	19.16	12.33
	0.9	33.36	31.28	27.72	19.83	15.7	10.17
	0.8	26.78	25.12	22.27	15.99	12.68	8.28
	0.75	23.92	22.43	19.9	14.31	11.36	7.45
	0.5	11.41	10.7	9.53	6.96	5.57	3.77

(a) CIFAR-MobileNet-v1

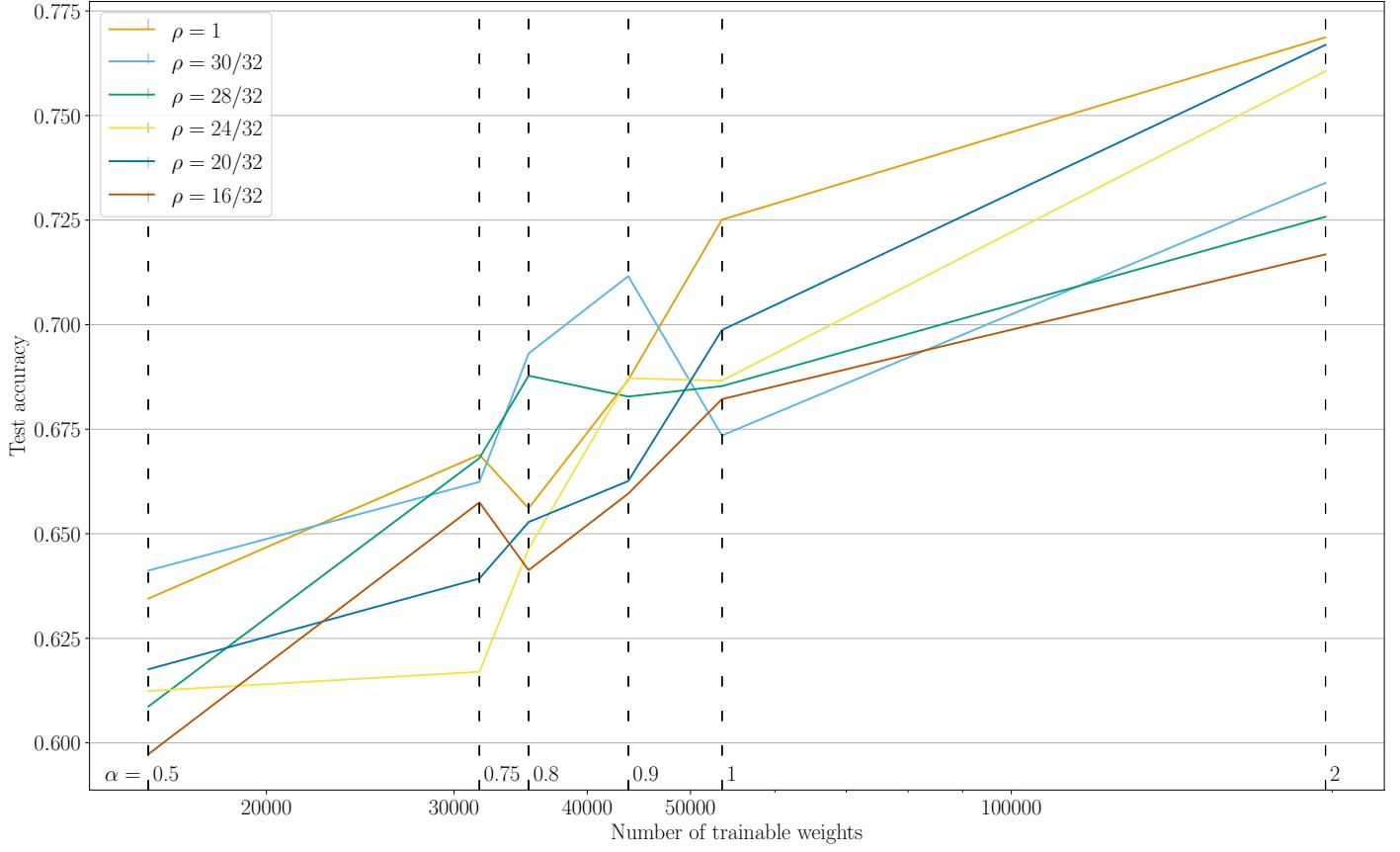
		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	18.3	16.63	14.56	10.38	7.64	4.73
	1	5.43	4.92	4.3	3.08	2.25	1.4
	0.9	4.4	3.98	3.48	2.49	1.82	1.13
	0.8	3.65	3.29	2.88	2.07	1.51	0.94
	0.75	3.38	3.05	2.67	1.92	1.39	0.87
	0.5	1.79	1.61	1.41	1.01	0.73	0.46

(b) CIFAR-MobileNet-v2

Table 4: Millions of mult-add operations w.r.t. ρ and α . Values were empirically determined using the `tf.compat.v1.profiler` profiler.

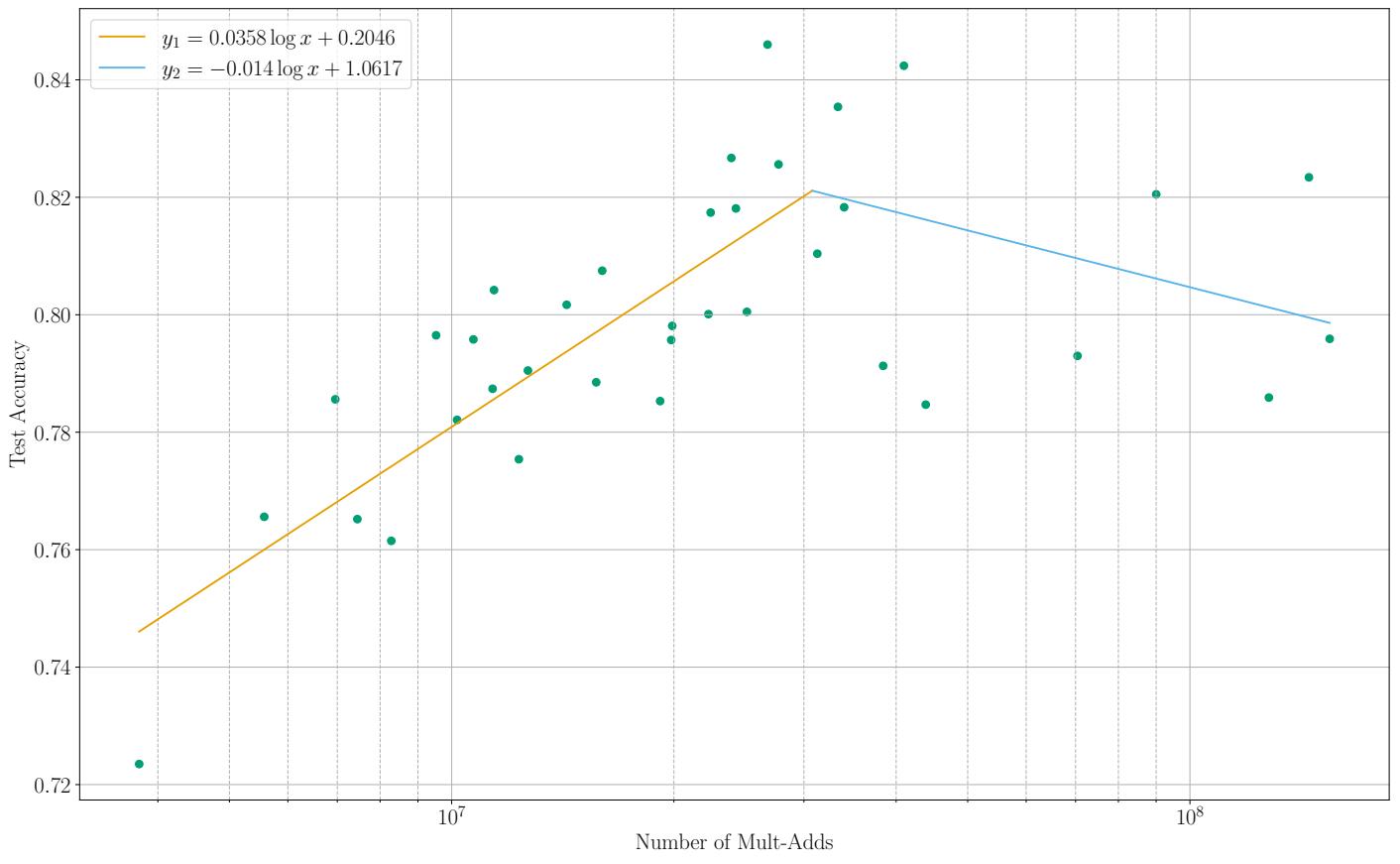


(a) CIFAR-MobileNet-v1

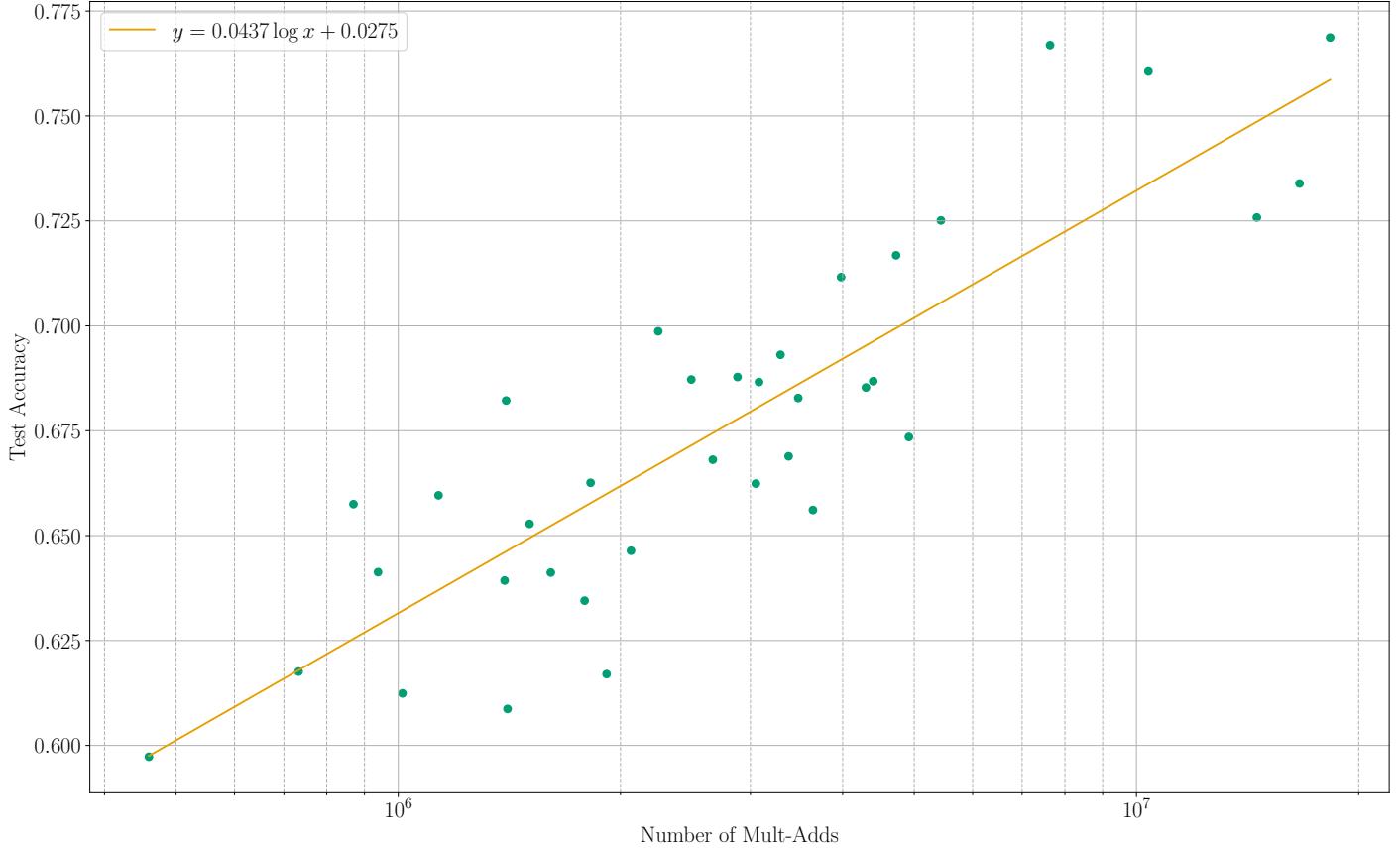


(b) CIFAR-MobileNet-v2

Figure 5: Accuracy vs. number of trainable weights for CIFAR-MobileNet models. ρ -series (with varying α) are shown as lines. Models with the same α share the same number of weights and lie on the same vertical dotted line.

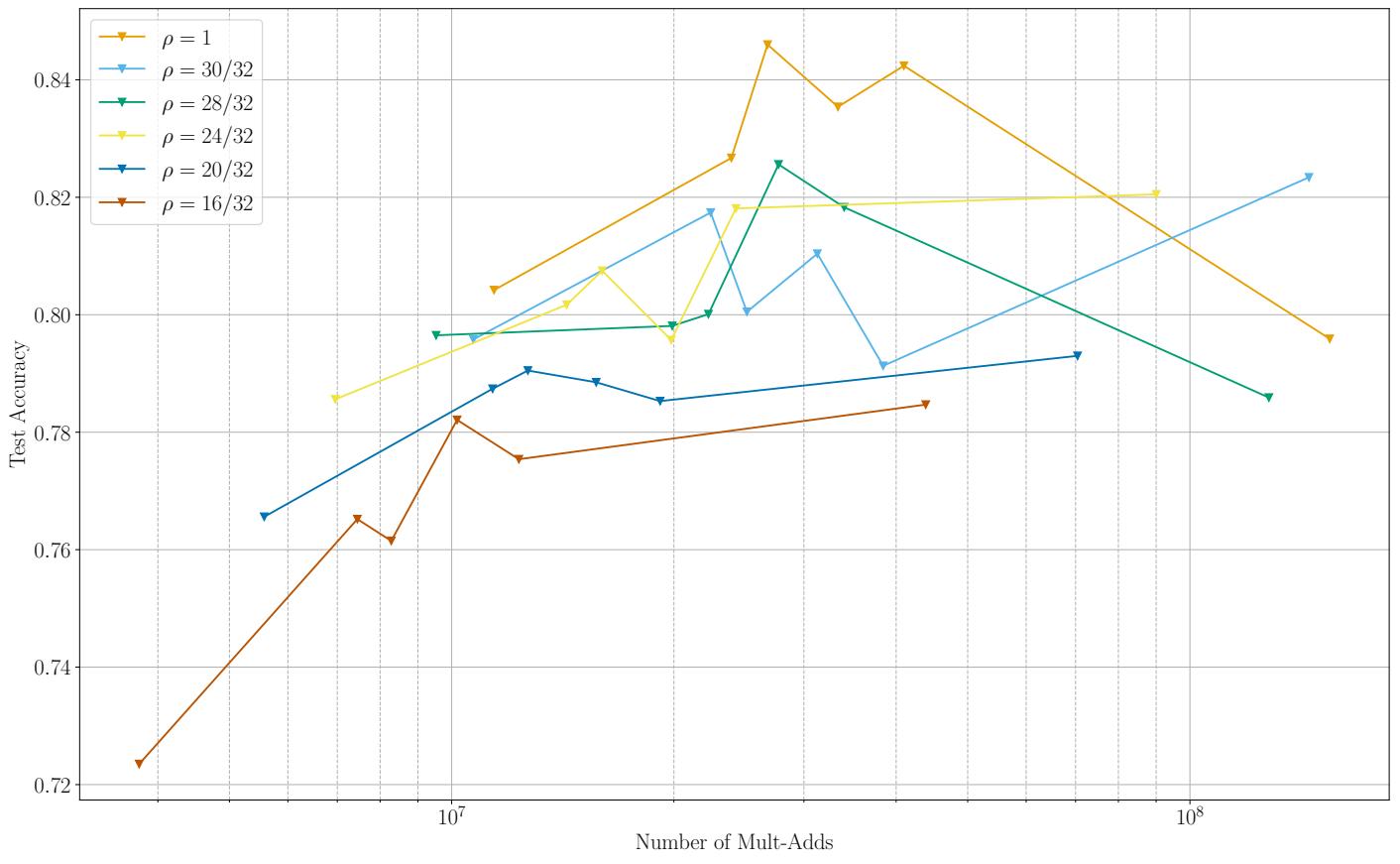


(a) CIFAR-MobileNet-v1

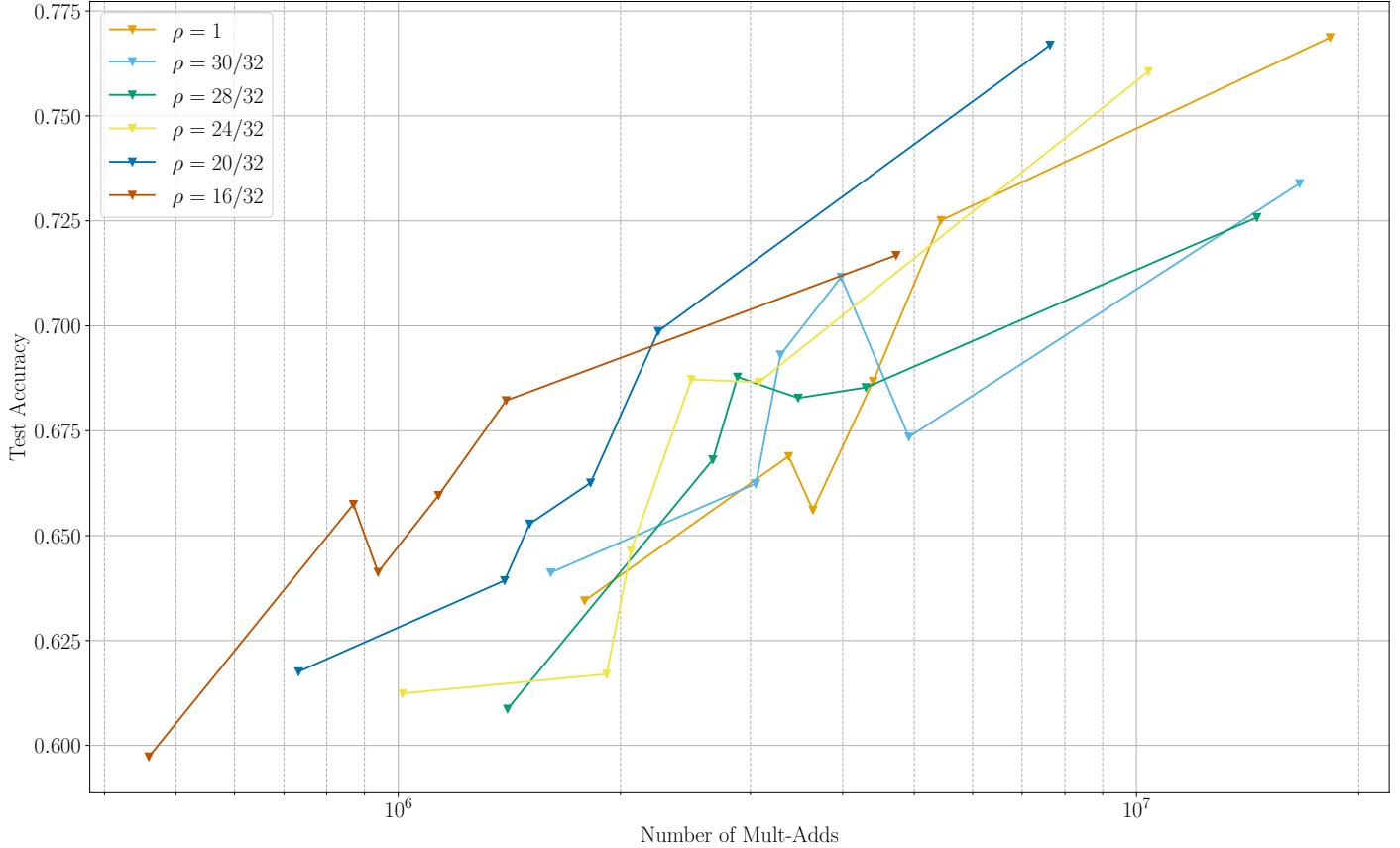


(b) CIFAR-MobileNet-v2

Figure 6: Accuracy vs. number of mult-adds for CIFAR-MobileNet models. Logarithmic regressions (logarithmic w.r.t. the number of mult-adds) are shown. (For v1, the logarithmic regressions were performed on the data left and right of the highest-accuracy point and were plotted to intersect.)



(a) CIFAR-MobileNet-v1



(b) CIFAR-MobileNet-v2

Figure 7: Accuracy vs. number of mult-adds for CIFAR-MobileNet models. ρ -series (with varying α) are shown as lines.

5 Conclusions and further inquiry

We were able to design a MobileNet implementation for CIFAR-10 incorporating the design features present in the original model designed for ImageNet. Our work is mainly concerned with comparing different versions (i.e., by varying the MobileNet parameters) of the same base MobileNet implementation, rather than being concerned with comparing MobileNets to other existing deep convolutional networks. We rederived the equations that the authors proposed for number of parameters and computational cost (as a function of the hyperparameters α and ρ) and verified their correctness against profiling data. We were able to train 72 different models (36 of CIFAR-MobileNet-v1 and 36 of CIFAR-MobileNet-v2), and used this to reproduce two figures from the original paper. While our plots are noisier than the results reported in original paper, the reproductions show the same general patterns.

Our original plan for this project also included running our MobileNet implementation on embedded hardware (as this would demonstrate MobileNet’s intended purpose), but we were not able to complete this on top of the other goals we had. Doing so would be a good continuation of this model.

Further research into time to train and time to infer using this model could be very fruitful. Neither we nor the authors of MobileNet provide statistics on either of these. Even better, examining how efficient depthwise convolution may be (as opposed to regular convolutions) based on how they utilize low-level linear algebra frameworks (e.g., BLAS) would be an interesting exploration.

References

- [1] Howard, Andrew G., et al. “MobileNets: Efficient convolutional neural networks for mobile vision applications.” *arXiv preprint arXiv:1704.04861* (2017).
- [2] Sifre, Laurent, and Stéphane Mallat. “Rigid-motion scattering for image classification.” *Ph. D. thesis* (2014).
- [3] Malcolm. “Answer to how to calculate a Mobilenet FLOPs in Keras.” *Stack Overflow*, Web, 15 Aug 2018. <https://stackoverflow.com/a/51866343/>.

6 Appendix I: Convolutional Layer Analyses

Because of the relevance of the fundamental depthwise-separable convolution to this paper, and for our own academic purposes, we decide to go more in-depth with an analysis of the convolutional layer types. The MobileNets paper does not go into detail about how they calculate number of weights and number of mult-adds, and since we attempt to reproduce their results, we describe how we understand and calculate these metrics.

We use the same notation as the book, i.e., for any given layer: $D_f :=$ input feature map size (length); $D_k :=$ kernel filter size (length); $M :=$ the number of input channels; $N :=$ the number of output channels. (The filters and feature maps are assumed to be square to make the calculations a little simpler, but this can be extended to any rectangular map.) I.e., if the input to a layer were a $32 \times 32 \times 256$ feature map, and number of output channels is 512, and the kernel filter size is 3×3 , then $D_f = 32$, $D_k = 3$, $M = 256$, and $N = 512$. In the case of depthwise convolution, $M = N$. In the case of a 1×1 convolution, $D_k = 1$.

A “mult-add” is not defined in the original paper, but we assume that it is some computational-efficient multiplication-addition combination. The authors of the original paper mention that it is important to not only be able to measure the number of FLOPS or similar measure, but also take into consideration how efficiently those operations can be carried out in hardware, which lends itself to the idea of general matrix multiplies (GEMMs) and mult-adds.

6.1 Regular convolution

```

1 pixels ← input feature map, dimensions (Df, Df, M)
2 filters ← array of convolutional kernel filters, dimensions (N, M, Dk, Dk)
3 biases ← array of biases for each output channel, dimensions (N)
4 out ← zero-initialized output feature map, dimensions (Df, Df, N)
5 for i, j ← pixels
6   for n ← 1..N
7     for m ← 1..M
8       out[i, j, n] += conv2d(filters[N, M, :, :], pixels[i:i+Dk, j:j+Dk, m])
9     endfor
10    out[i, j, n] += biases[n]
11  endfor
12 endfor
13 return out

```

Figure 8: Pseudocode for a regular convolution

From Figure 8, we can see that there are $NMD_k^2 + N$ learnable weights in the kernel and biases. Each regular convolution takes D_k^2 multiplications and $D_k^2 - 1$ additions. It is added to $\text{out}[i, j, n]$ so this adds one more addition; this is $2D_k^2$ FLOPs or D_k^2 mult-adds. This gets multiplied by the number of input channels, number of output channels, and the number of pixels for a total of $D_f^2NMD_k^2$ mult-adds. If the convolutional layer uses a bias term (i.e., toggled with `tf.keras.layers.Conv2D`'s `use_bias` parameter), then there are an additional D_f^2N additions. This latter term is dominated by the former if M or D_k is large (in our case, M gets very large), so we can safely ignore it from the overall mult-add calculations. (Also, while the default value for `use_bias` is `True`, and this is the value we use, in the MobileNets keras implementation there is no bias added. This is likely due to the fact that the layer is followed immediately by a batchnorm layer, which will provide a learned bias. We did not consider this when originally constructing our model design.)

6.2 Depthwise convolution

From Figure 9, there are $MD_k^2 + M$ learnable weights in the kernel and biases. As with regular convolution, a single convolution takes D_k^2 mult-adds. However, they are not combined in a sum to make an output channel; instead, each output channel is an input channel with one convolutional filter applied to it. Thus this makes $D_f^2MD_k^2$ mult-adds. Again, the number of operations from addition of a bias term are not important (and the keras implementation doesn't use the bias in depthwise convolution).

```

1 pixels ← input feature map, dimensions (Df, Df, M)
2 filters ← array of convolutional kernel filters, dimensions (M, Dk, Dk)
3 biases ← array of biases for each channel, dimensions (M)
4 out ← output feature map, dimensions (Df, Df, M)
5 for i, j ← pixels
6   for m ← 1..M
7     out[i, j, m] ← conv2d(filters[M, :, :], pixels[i:i+Dk, j:j+Dk, m])
8     out[i, j, m] += biases[m]
9   endfor
10 endfor
11 return out

```

Figure 9: Psuedocode for a depthwise-separable convolution

6.3 Including hyperparameters in calculations

This is fairly self-explanatory and is explained in the original paper. The width multiplier α scales the input and output feature sizes. The resolution multiplier ρ scales the feature map size. Since the feature map does not affect the number of weights, ρ is independent of the number of weights in the model (i.e., the memory requirement of the model). The computational cost of the regular convolutional layers (which account for most of the calculations in the MobileNet model) is jointly proportional to α^2 and ρ^2 .

6.4 Summary of equations and example calculation

Layer type	Input shape	Weight count
2D depthwise convolution	$112 \times 112 \times 64$	640
Batch Normalization	$56 \times 56 \times 64$	256
ReLU	$56 \times 56 \times 64$	0
2D convolution	$56 \times 56 \times 64$	8320
Batch Normalization	$56 \times 56 \times 128$	512
ReLU	$56 \times 56 \times 128$	0

Table 5: Sample MobileNet block. This block has a 3×3 2D depthwise convolution with stride 2, and a 1×1 convolution that doubles the number of output channels, a typical pattern in the MobileNet blocks. (This is the second MobileNet block in `full_mobilenet` with $\alpha = \rho = 1$.) Note that the majority of the calculations and parameters lie in the 1×1 convolutions.

The calculations for number of weights and computational cost are summarized in the table below. (S is the stride length.)

$$\begin{aligned}
|W_{\text{regular conv.}}| &= D_k^2 \times \alpha M \times \alpha N + \alpha N \\
C_{\text{regular conv.}} &\approx \rho^2 D_f^2 (\div S^2) \times \alpha M \times \alpha N \times D_k^2 \\
|W_{\text{depthwise conv.}}| &= D_k^2 \times \alpha M + \alpha M \\
C_{\text{depthwise conv.}} &\approx \rho^2 D_f^2 (\div S^2) \times \alpha M \times D_k^2
\end{aligned}$$

Using these equations, we can estimate that the cost of the MobileNet block in Table 5 is:

$$C \approx (1)^2(112)^2 \times (1)(64) \times (3)^2 \div (2)^2 + (1)^2(56)^2 \times (1)(64) \times (1)(128) \times (1)^2 = 27496448$$

The value given by the profiler is 27806660 mult-adds (55613319 FLOPs $\div 2$). There is only a 1% error between theory and practice (even with some estimates in the theoretical calculation), which is not bad. It is also not hard to check that the number of weights match the values given by the equations. Now that we have shown by example that these equations are correct, we defer the rest of the calculations to the profiler.

6.5 Empirical calculations

Rather than calculating all of the models' weight counts and complexities by hand, we used the profiler from Tensorflow's v1 (compatibility module). This not only saved us a lot of trouble, but gets closer to the actual cost complexities of the models; our calculations only accounted for the convolutional layers, which form the bulk of both the weights and the computational cost, but there are still many calculations and weights in the batchnorm and dense layers.

The code used for profiling came from an answer on Stack Overflow [3]. (This source also gave us the estimate for mult-adds based on FLOPs.) It uses Tensorflow v1 code, so we have to use the compatibility module.

```

1 session = tf.compat.v1.Session()
2 graph = tf.compat.v1.get_default_graph()
3
4 with graph.as_default():
5     with session.as_default():
6         model = model_class(**model_parms,
7             input_tensor=tf.compat.v1.placeholder('float32',
8                 shape=(1, int(model_parms['rho']*32), int(model_parms['rho']*32), 3)))
9
10        run_meta = tf.compat.v1.RunMetadata()
11        opts = tf.compat.v1.profiler.ProfileOptionBuilder.float_operation()
12        flops = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta,
13            cmd='op', options=opts)
14
15        opts = tf.compat.v1.profiler.ProfileOptionBuilder.trainable_variables_parameter()
16        params = tf.compat.v1.profiler.profile(graph=graph,
17            run_meta=run_meta, cmd='op', options=opts)
18
19        flops, num_params = flops.total_float_ops, params.total_parameters

```

where `model_class` is a factory function for the model we want to analyze. To be consistent with the original authors, we report our results in mult-adds. Since the profiler outputs FLOPs, we make the approximation that one mult-add is approximately two FLOPs; this is justified by the fact that the vast majority of calculations occur in the regular convolution layers and dense layer, and in those layers most operations can be modeled as a multiplication followed by an addition. This gives us results almost exactly equal to the reported values in the original paper, so this is likely a sound assumption.

7 Appendix II: Source Code

```
1 # define a mobilenet block
2 depthwise_conv = tf.keras.layers.DepthwiseConv2D
3 regular_conv = tf.keras.layers.Conv2D
4 batchnorm = tf.keras.layers.BatchNormalization
5 activation = tf.keras.layers.ReLU
6 average_pooling = tf.keras.layers.AveragePooling2D
7 flatten = tf.keras.layers.Flatten
8 dropout = tf.keras.layers.Dropout
9 dense = tf.keras.layers.Dense
10
11 # original convolutional block, can be used when calculating the
12 # difference in the number of parameters or comparing accuracy;
13 # returns a list that can be dropped into a keras Model.Sequential
14 def regular_conv_block(stride_2, out_channels, with_dropout=True):
15     return [
16         regular_conv(int(out_channels), (3, 3), padding='same',
17                     strides=((2, 2) if stride_2 else (1, 1))),
18         batchnorm(),
19         activation()
20     ]
21
22 # mobilenet block, as described in the original paper;
23 # also returns a list that can be used in a keras Model.Sequential
24 def mobilenet_block(stride_2, out_channels, with_dropout=True):
25     return [
26         depthwise_conv((3, 3), padding='same',
27                         strides=((2, 2) if stride_2 else (1, 1))),
28         batchnorm(),
29         activation(),
30         regular_conv(int(out_channels), (1, 1)),
31         batchnorm(),
32         activation(),
33
34         # not part of the original model
35         dropout(0.2)
36     ]
```

Figure 10: Convolutional and MobileNet blocks

```

1  # full MobileNet model; run this with rho=1/7 to work with CIFAR-10 without
2  # further modification (designed for 224x224 image);
3  # input_tensor necessary when profiling model
4  def full_mobilenet(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
5      block = mobilenet_block if use_mobilenet_block else regular_conv_block
6
7      model = tf.keras.Sequential([
8          *[tf.keras.Input((int(rho * 224), int(rho * 224), 3))] \
9              if input_tensor is None else [tf.keras.Input(tensor=input_tensor)],
10
11         regular_conv(int(alpha * 32), (3, 3), padding='same', strides=(2, 2)),
12
13         *block(False, int(alpha * 64)),
14         *block(True, int(alpha * 128)),
15         *block(False, int(alpha * 128)),
16         *block(True, int(alpha * 256)),
17         *block(False, int(alpha * 256)),
18         *block(True, int(alpha * 512)),
19
20         *block(False, int(alpha * 512)),
21         *block(False, int(alpha * 512)),
22         *block(False, int(alpha * 512)),
23         *block(False, int(alpha * 512)),
24         *block(False, int(alpha * 512)),
25
26         *block(True, int(alpha * 1024)),
27         *block(False, int(alpha * 1024)),
28
29         average_pooling((int(rho * 7), int(rho * 7))),
30         flatten(),
31         dropout(0.2),
32
33         # this differs from ImageNet because of number of classes;
34         # change to 1000 if want to measure params/cost on 224x224 image
35         # (i.e., to compare results with original paper)
36         dense(10)
37     ])
38
39     model.compile(
40         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
41         optimizer=tf.keras.optimizers.Adam(),
42         metrics=['accuracy']
43     )
44
45     return model

```

Figure 11: Full MobileNet factory function (designed for $224 \times 224 \times 3$ images)

```

1  # CIFAR-MobileNet-v1: designed for 32x32 images
2  # input_tensor necessary when profiling model
3  def cifar_mobilenet_v1(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
4      block = mobilenet_block if use_mobilenet_block else regular_conv_block
5
6      model = tf.keras.Sequential([
7          *([tf.keras.Input((int(rho * 32), int(rho * 32), 3))] \ 
8              if input_tensor is None else [tf.keras.Input(tensor=input_tensor)]),
9
10         regular_conv(int(alpha * 32), (3, 3), padding='same'),
11
12         *block(False, alpha * 64),
13         *block(False, alpha * 128),
14         *block(True, alpha * 256),
15         *block(True, alpha * 512),
16         *block(True, alpha * 1024),
17
18         average_pooling((int(rho * 4), int(rho * 4))),
19         flatten(),
20         dropout(0.2),
21         dense(1000),
22         dense(10)
23     ])
24
25     model.compile(
26         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
27         optimizer=tf.keras.optimizers.Adam(),
28         metrics=['accuracy']
29     )
30
31     return model

```

Figure 12: CIFAR-MobileNet-v1 factory function (designed for $32 \times 32 \times 3$ images)

```

1  # CIFAR-MobileNet-v2: designed for 32x32 images
2  # input_tensor necessary when profiling model
3  def cifar_mobilenet_v2(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
4      block = mobilenet_block if use_mobilenet_block else regular_conv_block
5
6      model = tf.keras.Sequential([
7          *([tf.keras.Input((int(rho * 32), int(rho * 32), 3))] \
8              if input_tensor is None else [tf.keras.Input(tensor=input_tensor)]),
9
10         regular_conv(int(alpha * 32), (3, 3), padding='same'),
11
12         *block(False, alpha * 32),
13         *block(False, alpha * 32),
14         *block(True, alpha * 64),
15         *block(True, alpha * 128),
16         *block(True, alpha * 256),
17
18         average_pooling((int(rho * 4), int(rho * 4))),
19         flatten(),
20         dropout(0.2),
21         dense(10)
22     ])
23
24     model.compile(
25         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
26         optimizer=tf.keras.optimizers.Adam(1e-4),
27         metrics=['accuracy']
28     )
29
30     return model

```

Figure 13: CIFAR-MobileNet-v2 factory function (designed for $32 \times 32 \times 3$ images). Note that the learning rate is smaller (due to training instabilities we noticed). This is essentially a thinner CIFAR-MobileNet-v1.

```

1  # session flops and parm count are cumulative, so we need to subtract old cumulative value
2  # in order to get current model's flops and parm count
3  cum_flops, cum_parms = 0, 0
4
5  # run model, collect summary and history
6  # - model_class: model factory function
7  # - num_epochs: number of epochs to train for
8  # - run_model: if true, will actually train the model; if false, will still generate
9  #   model metadata and profiling details without training
10 # - model_parms: parameters to forward to the model factory function
11 def run_model(model_class, num_epochs, run_model=True, model_parms={}):
12     global cum_flops, cum_parms
13
14     # counting flops and number of trainable parms/weights in model
15     # see: https://stackoverflow.com/a/59862883/2397327
16     session = tf.compat.v1.Session()
17     graph = tf.compat.v1.get_default_graph()
18
19     with graph.as_default():
20         with session.as_default():
21             # create and compile model
22             model = model_class(**model_parms,
23                                 input_tensor=tf.compat.v1.placeholder('float32',
24                                                       shape=(1, int(model_parms['rho']*32), int(model_parms['rho']*32), 3)))
25
26             # profiling
27             run_meta = tf.compat.v1.RunMetadata()
28             opts = tf.compat.v1.profiler.ProfileOptionBuilder.float_operation()
29             flops = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta, cmd='op', options=opts)
30
31             opts = tf.compat.v1.profiler.ProfileOptionBuilder.trainable_variables_parameter()
32             params = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta, cmd='op', options=opts)
33
34             flops, num_params = flops.total_float_ops - cum_flops, params.total_parameters - cum_parms
35             cum_flops += flops
36             cum_parms += num_params
37
38             metadata = {
39                 'model': 'v1' if model_class == cifar_mobilenet_v1 else 'v2',
40                 'epochs': num_epochs,
41                 'params': model_parms,
42                 'summary': '',
43                 'flops': flops,
44                 'num_params': num_params
45             }
46
47             # save model summary to a string (not the default)
48             def save_model_summary_to_string(summary_line):
49                 nonlocal metadata
50                 metadata['summary'] = metadata['summary'] + summary_line + '\n'
51             model.summary(print_fn=save_model_summary_to_string)
52
53             # run model
54             if run_model:
55                 # don't resize feature sets
56                 resized_train_images = train_images
57                 resized_test_images = test_images

```

```

58     # resize images based on rho; note this also depends on the model's
59     # default image size (32 for our model, 224 for original model)
60     rho = model_parms['rho'] if 'rho' in model_parms else 1
61     if rho != 1:
62         default_size = 224 if model_class == full_mobilenet else 32
63         resized_train_images = tf.image.resize(
64             resized_train_images,
65             (int(rho * default_size), int(rho * default_size)))
66         )
67         resized_test_images = tf.image.resize(
68             resized_test_images,
69             (int(rho * default_size), int(rho * default_size)))
70         )
71
72     # train
73     history = model.fit(
74         resized_train_images, train_labels,
75         epochs=num_epochs, validation_split=0.2,
76         callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',
77             patience=10, min_delta=0.005)]).history
78
79     # final test accuracy
80     test_accuracy = model.evaluate(resized_test_images, test_labels,
81         verbose=1, return_dict=True)
82 else:
83     history = None
84     test_accuracy = None
85
86     return {
87         'metadata': metadata,
88         'history': history,
89         'test_accuracy': test_accuracy
90     }
91
92 models = [cifar_mobilenet_v1, cifar_mobilenet_v2]
93 epochs = [100]
94 alphas = [2, 1, 0.9, 0.8, 0.75, 0.5]
95 rhos = [1, 30/32, 28/32, 24/32, 20/32, 16/32]
96
97 results = []
98 grid_count = len(models) * len(epochs) * len(alphas) * len(rhos)
99 for model in models:
100     for epoch in epochs:
101         for alpha in alphas:
102             for rho in rhos:
103                 results.append(run_model(model, epoch, run_model=True, model_parms={
104                     'alpha': alpha,
105                     'rho': rho,
106                     # 'use_mobilenet_block': False,
107                 }))

```

Figure 14: Test rig and profiling code. (Note that this code does not provide any of the analysis or plotting of the data. It only generates the data.)

ECE472 – Project 1

Jonathan Lam

September 9, 2020

Notes on implementation

- TensorFlow and GradientTape were used to perform automatic differentiation.
- A version of stochastic gradient descent was implemented, where a small random sample (adjustable via *batchSize*) was chosen from the input samples for every step. I didn't really find any performance gain from this, however (I guess even the full-size matrices are still considered relatively small), nor did it appear to noticeably affect the manifold. I set a default *batchSize* of 10.
- For initial values, I chose to set all the coefficients w and bias b to 0, uniform-randomly spaced the means μ throughout the sample support, and gave a constant positive value for σ .
- The learning hyper-parameters of $stepSize = 0.01$ and $stepCount = 1000$ seemed to work well enough to closely approximate the sinusoid (with the given initial values).
- I experimented with M between 2 and 1000, and all values in this range (even as few as 2) appeared to give good approximations of the single period of the sine wave (with the initial given values). For this report, I've plotted $M = \{1, 2, 4, 8, 16, 32\}$ on some different functions just to see what kind of effect they would have.
- To improve performance and also get a little more familiar with Python's libraries, I also used the `multiprocessing` package to speed up some of the calculations. (A rough estimate of the running time for one basis fitting was approximately 2s when $stepSize = 1000$, and about 30s when $stepSize = 10000$.)

Notes on figures

I tried a few functions other than the sinusoid.

- $y = \sin(2\pi x)$, $x \in [0, 1]$

This was fit pretty fast by gradient descent and Gaussians, which makes sense given the similarity in their shapes and its smoothness. Even with $M = 2$ a decent fit is made. Clearly, when M is large (e.g., $M = 16$ or $M = 32$), there seems to be some overfitting.

- $y = \sin(6\pi x)$, $x \in [0, 1]$

Unsurprisingly, this also fit well given enough basis functions ($M = 16, M = 32$ worked well). The basis functions spread out really evenly for these.

- $y = \text{sinc}(x)$, $x \in [-6, 6]$

The sinc function was harder to fit. Even with $M = 32$, I had to up the *stepCount* to 10000 or increase *stepSize* to 0.1 to get it to approximate the given curve well, and even then the fit is not as good as the earlier sine functions.

- $y = \text{rect}_{(0,2)}(x)$, $x \in [-2, 5]$

As expected, this also took longer to fit, so I increased *stepSize* to 10000. It has Gibbs phenomenon-like overshoots, which is not unexpected. With $M = 16$ or $M = 32$, this provided a surprisingly okay manifold.

Source code

```
# Jonathan Lam
# Prof. Curro
# ECE472 -- Deep Learning
# 9 / 8 / 20
# Project 1

import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import multiprocessing as mp

# definition of a gaussian; may be used with scalars, vectors, and/or matrices
# (as long as vectors and matrices are of the same size)
def gaussian(x, mu, sig):
    return tf.math.exp(-((x - mu) / sig)**2)

# fits a set of sample (x, y) pairs to a linear combination of M Gaussian
# basis vectors
class GaussianFit:

    # x, y sample (x, y) pairs
    # M number of basis functions
    # stepSize learning rate when updating vars with gradients
    # stepCount number of learning steps to perform during fit
    # batchSize batch size for stochastic gradient descent step
    def __init__(self, x, y, M=5, stepSize=0.01, stepCount=1000, batchSize=10):
        self.M = M
```

```

    self.N = tf.size(x)
    self.stepSize = stepSize
    self.stepCount = stepCount
    self.batchSize = batchSize

    # for use in step when randomizing indices; don't need to regenerate
    # this every time
    self._indices = np.arange(len(x))

    # learning variables (will be optimized with in the fit() function)
    # initial values: w, b set to 0, mu equally spaced throughout the sample
    # interval, and sig set to a uniform value
    xMin = np.min(x)
    xMax = np.max(x)
    self.vars = {
        'w': tf.Variable([0] * self.M, dtype=tf.float32),
        'b': tf.Variable(0, dtype=tf.float32),
        'sig': tf.Variable([(xMax - xMin) / 10] * self.M, dtype=tf.float32),
        'mu': tf.Variable(tf.linspace(xMin, xMax, self.M), dtype=tf.float32)
    }
    self.x = x
    self.y = y

# calculate y using current vars
def f(self, x):
    MU, X = tf.meshgrid(self.vars['mu'], x)
    SIGMA, _ = tf.meshgrid(self.vars['sig'], x)
    return self.vars['b'] + \
        gaussian(X, MU, SIGMA) @ tf.reshape(self.vars['w'], (self.M, 1))

# perform a single step of stochastic gradient descent
# single step can be used for animating (if I have time to do that)
def step(self):
    # generate random mini-batch for sgd
    sampleIndices = np.random.choice(self._indices, self.batchSize)
    xSample = tf.gather(self.x, sampleIndices)
    ySample = tf.gather(self.y, sampleIndices)

    # calculate estimate, loss (and perform autodiff)
    with tf.GradientTape() as tape:
        yhat = tf.reshape(self.f(xSample), (self.batchSize,))
        loss = tf.reduce_mean((ySample - yhat) ** 2)

    # subtract gradients

```

```

        for var, grad in tape.gradient(loss, self.vars).items():
            self.vars[var].assign_sub(self.stepSize * grad)

# perform stochastic gradient descent
# returns self for convenience later
def fit(self):
    for i in range(self.stepCount):
        self.step()
    return self

# helper function to plot a given gaussian fit model, the sample points it was
# meant to approximate, and the function that generated the sample points
#
# sampleX, sampleY (x, y) pairs of (noisy) sample data
# f original function approximated by sample data
# gf gaussian fit model object
# axes pair of axes to plot on
# name name of function
def plotFit(sampleX, sampleY, f, gf, axes, name):
    ax1, ax2 = axes
    t = tf.linspace(np.min(sampleX), np.max(sampleX), 100)

    # plot noisy sample, true wave, estimate wave
    ax1.set_title(f'{name} estimate manifold (M={gf.M})')
    ax1.plot(sampleX, sampleY, 'x', label='Noisy sample')
    ax1.plot(t, f(t), label='Clean signal', color='#dddddd')
    ax1.plot(t, gf.f(t), '--', label='Generated manifold')
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.legend(loc='upper right')

    # plot bases
    ax2.set_title(f'{name} estimate basis functions (M={gf.M})')
    for i in range(gf.M):
        ax2.plot(t, gaussian(t, gf.vars['mu'][i], gf.vars['sig'][i]))
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')

# sweepMs: helper function to plot multiple charts for different M values;
# threadFn is called in separate processes to speed up execution
#
# xMin, xMax range of X values for sample
# stdNoise noise standard deviation

```

```

# N number of points
# f function
# name name of function
def threadFn(x, y, M, kwargs, name):
    print(f'Running fit on {name} with M={M}')
    return GaussianFit(x, y, M=M, **kwargs).fit()
def sweepMs(xMin, xMax, stdNoise, N, f, name, uniformX=False, **kwargs):
    # generate x (either uniformly or uniformly randomly distributed)
    x = tf.linspace(xMin, xMax, N) if uniformX \
        else tf.random.uniform([N], xMin, xMax)

    # generate noisy signal
    y = f(x) + tf.random.normal([N], 0, stdNoise)

    # sweep M's, run in mp pool to speed up
    Ms = [1, 2, 4, 8, 16, 32]
    with mp.Pool(mp.cpu_count()) as pool:
        gfThreads = [pool.apply_async(threadFn, (x, y, M, kwargs, name)) \
            for M in Ms]
        gfs = [res.get() for res in gfThreads]

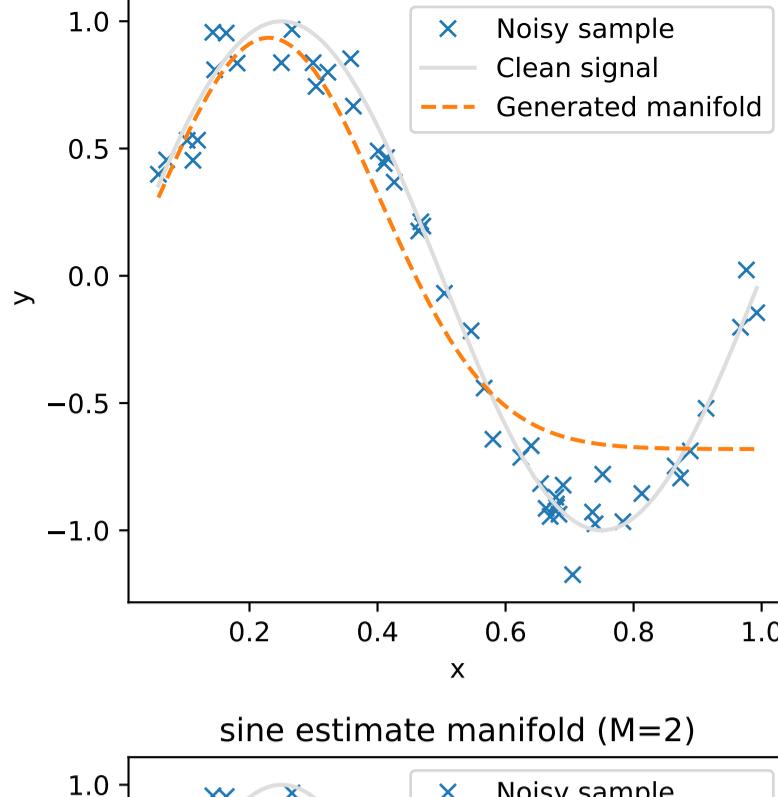
    fig, axes = plt.subplots(len(Ms), 2, figsize=(8.5, 4*len(Ms)))
    for gf, currAxes in zip(gfs, axes):
        plotFit(x, y, f, gf, currAxes, name)

    plt.tight_layout()
    plt.savefig(f'out/fn_{name}.pdf')

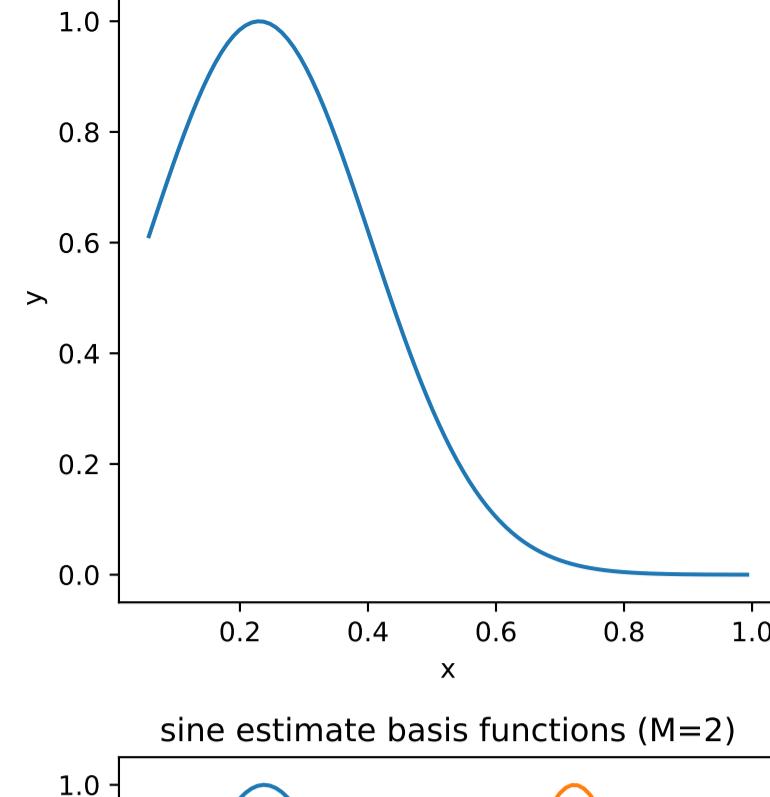
# try the same on some other functions
sweepMs(0., 1., 0.1, 50, lambda x: tf.math.sin(2 * np.pi * x), 'sine')
sweepMs(0., 1., 0.1, 50, lambda x: tf.math.sin(6 * np.pi * x), 'multicyclesine')
sweepMs(-6., 6., 0., 50, lambda x: np.sinc(x), 'sinc', \
    uniformX=True, stepSize=0.1)
sweepMs(-2., 5., 0., 50, lambda x: np.where(np.abs(x - 1) < 1, 1, 0), \
    'rectwin', uniformX=True, stepSize=0.1, stepCount=10000)

```

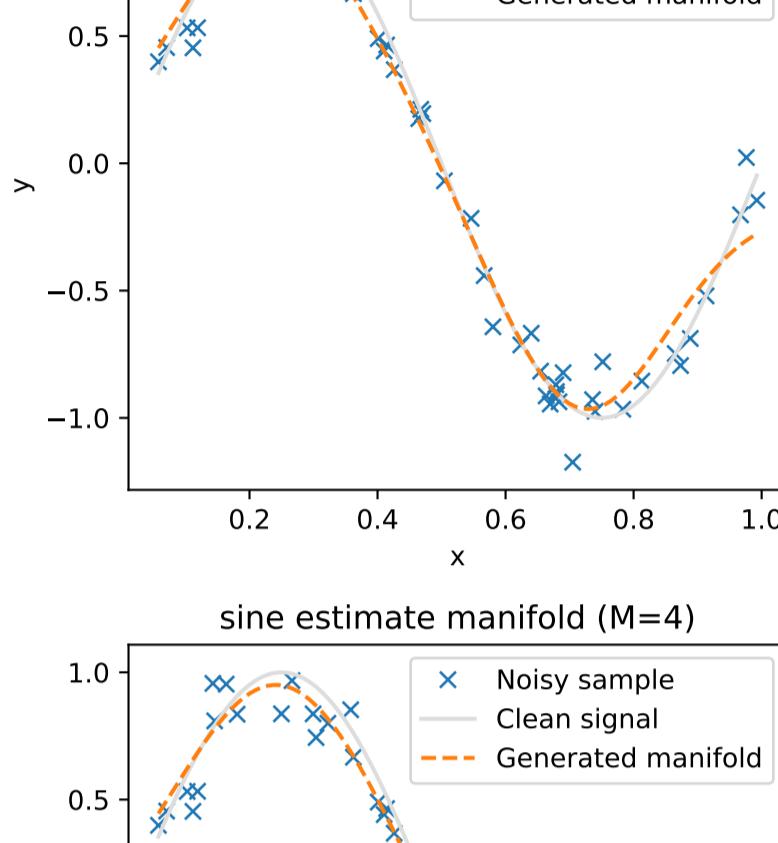
sine estimate manifold (M=1)



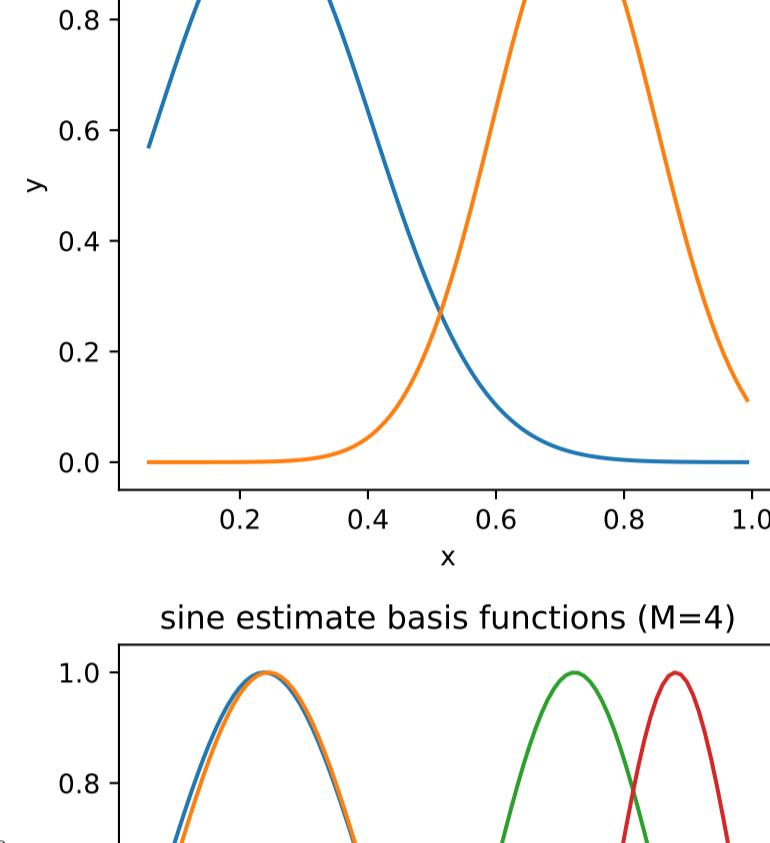
sine estimate basis functions (M=1)



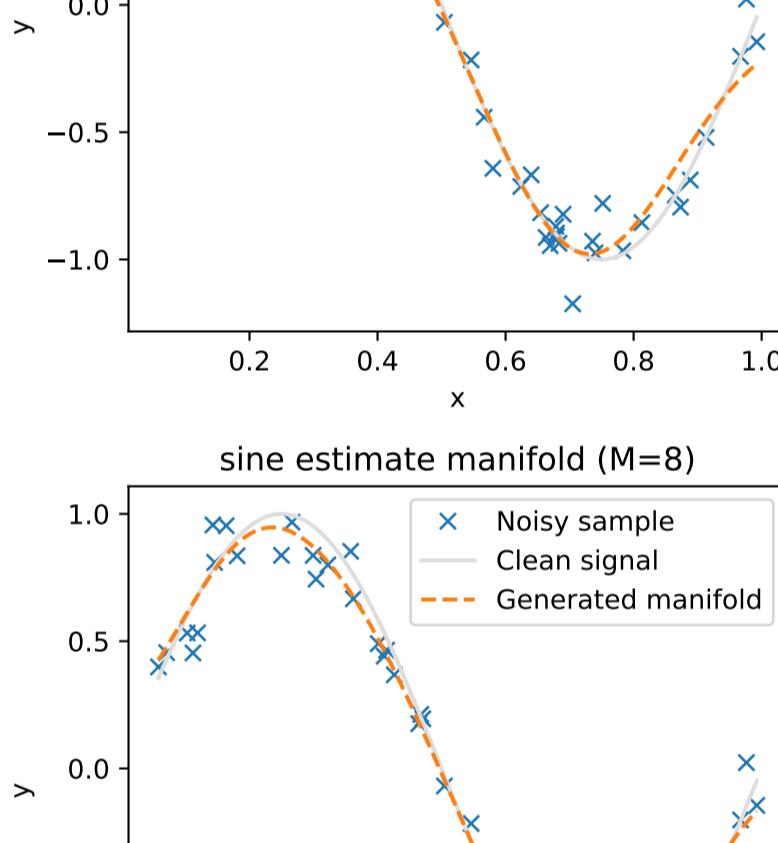
sine estimate manifold (M=2)



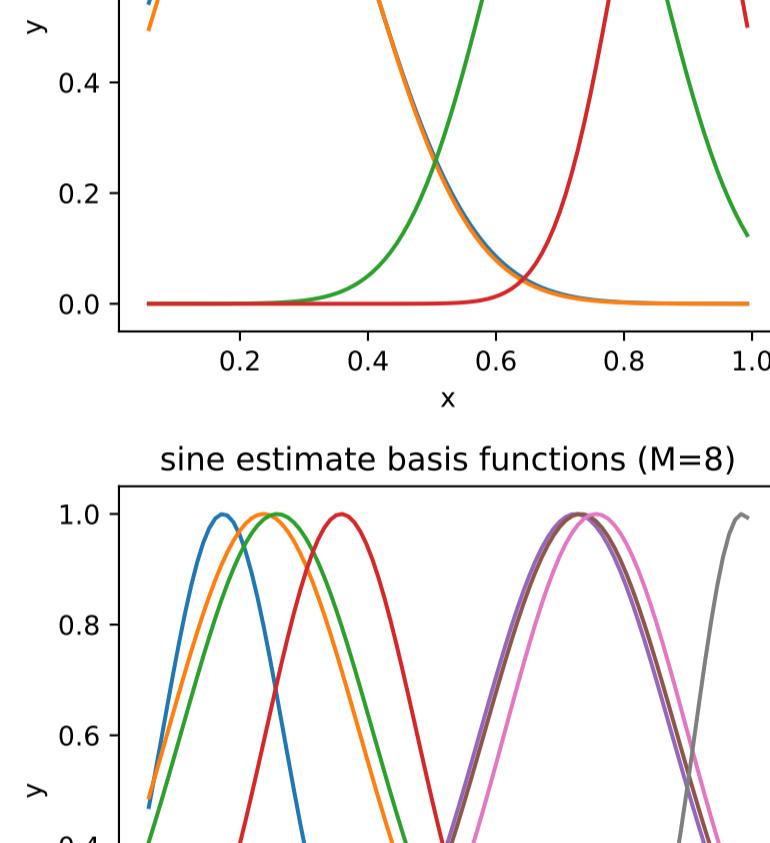
sine estimate basis functions (M=2)



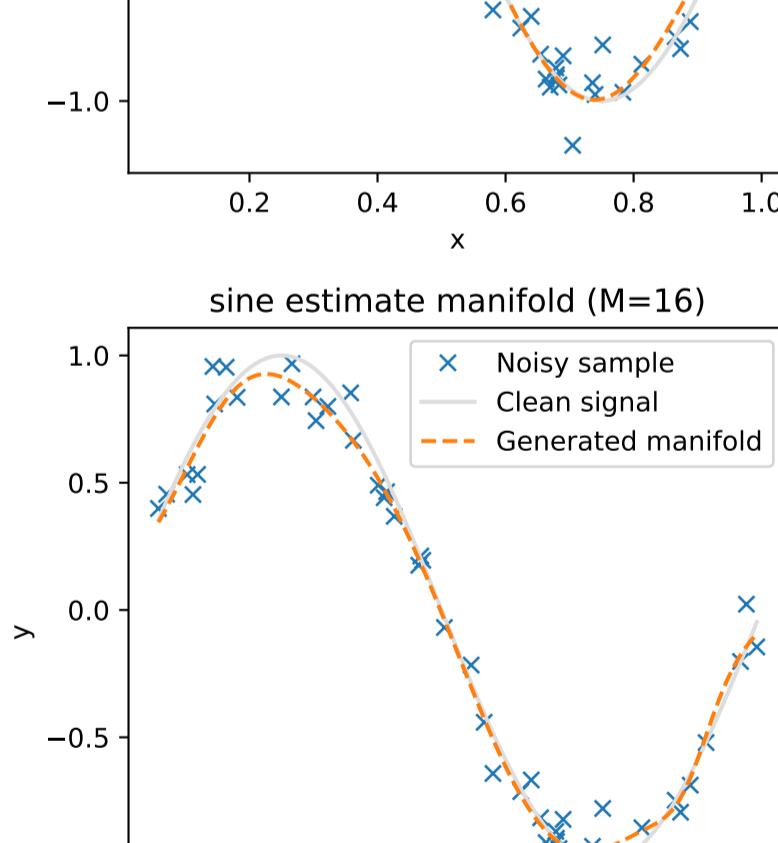
sine estimate manifold (M=4)



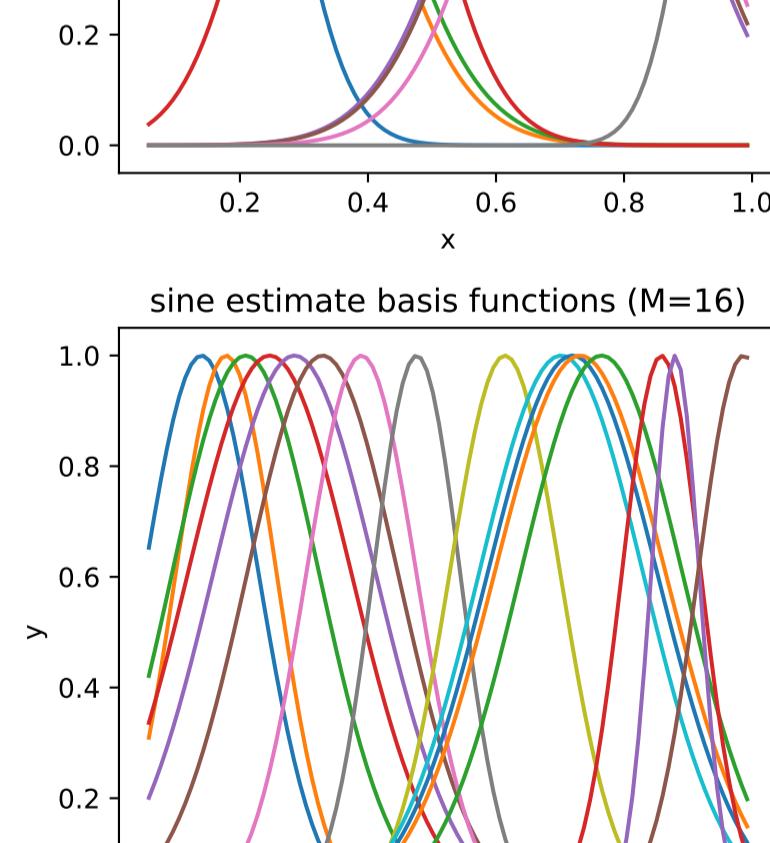
sine estimate basis functions (M=4)



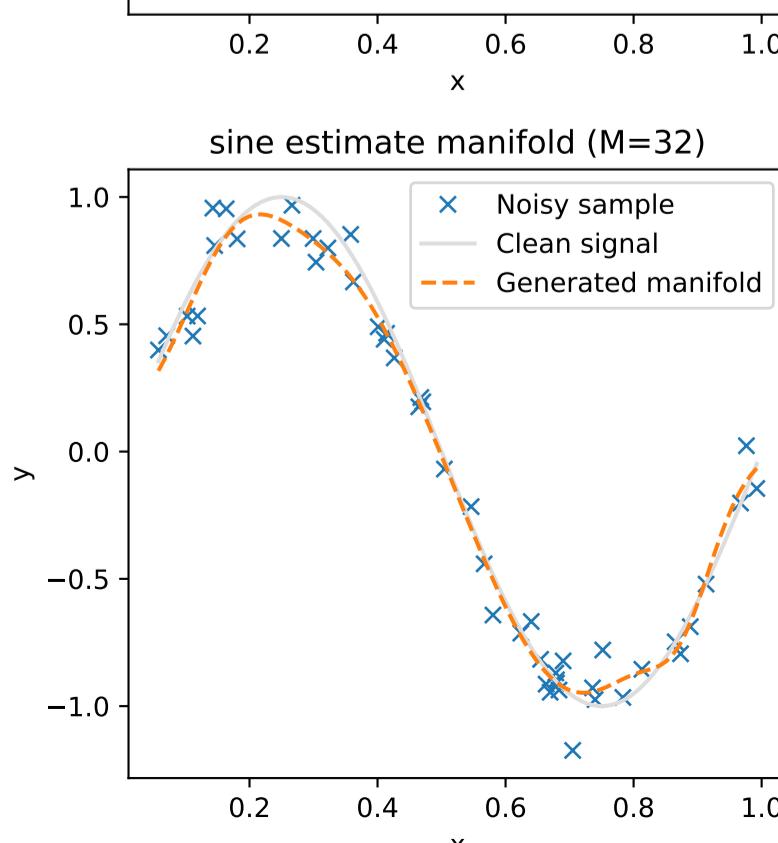
sine estimate manifold (M=8)



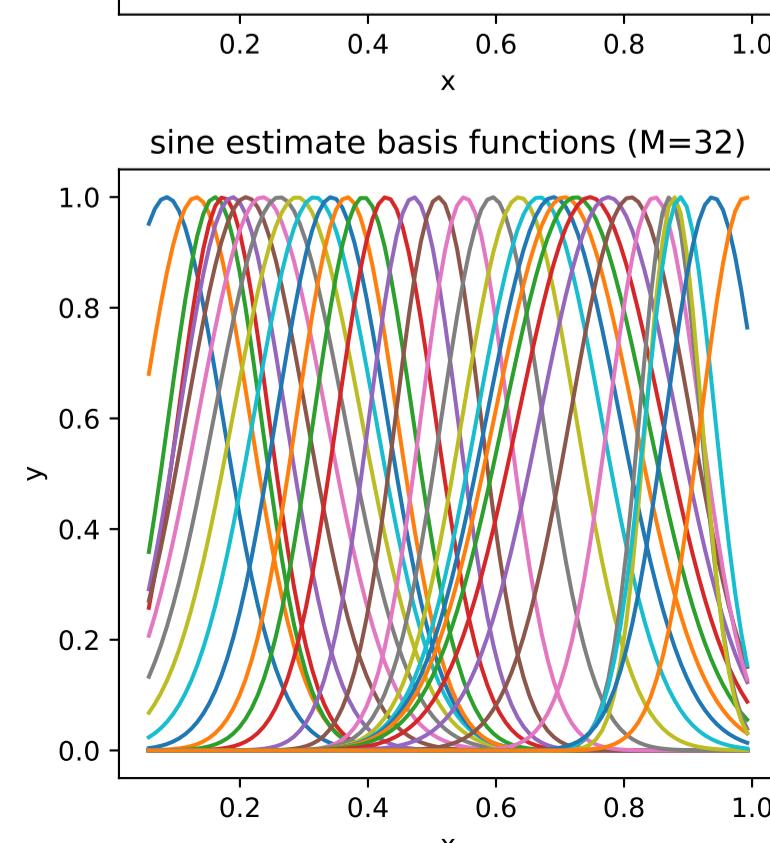
sine estimate basis functions (M=8)



sine estimate manifold (M=16)



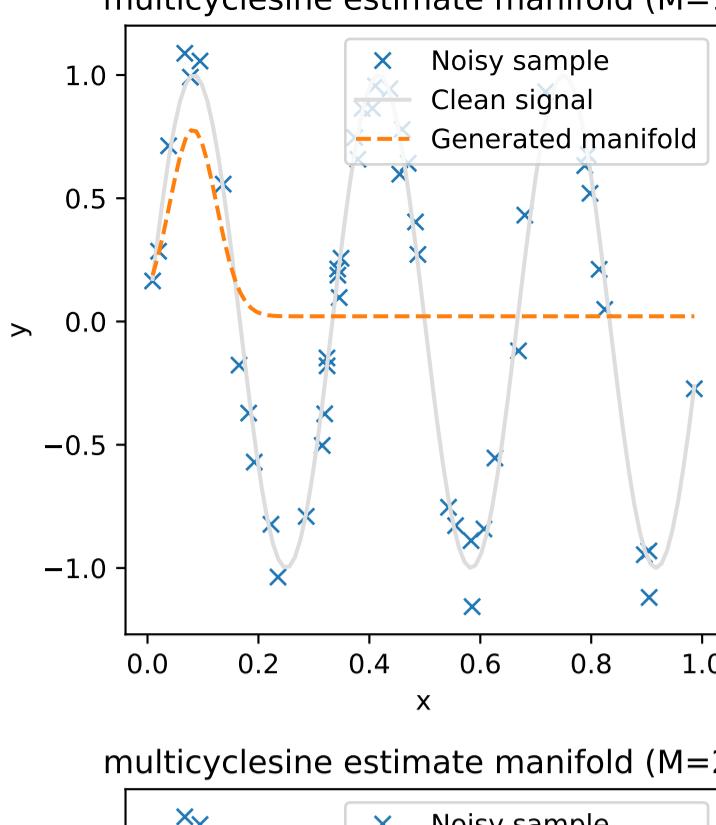
sine estimate basis functions (M=16)



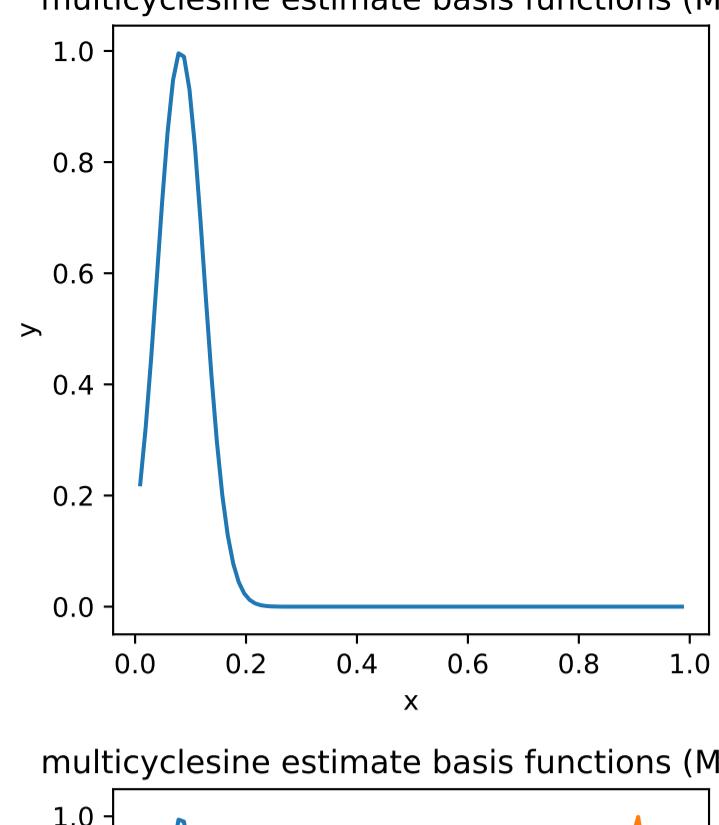
sine estimate manifold (M=32)

sine estimate basis functions (M=32)

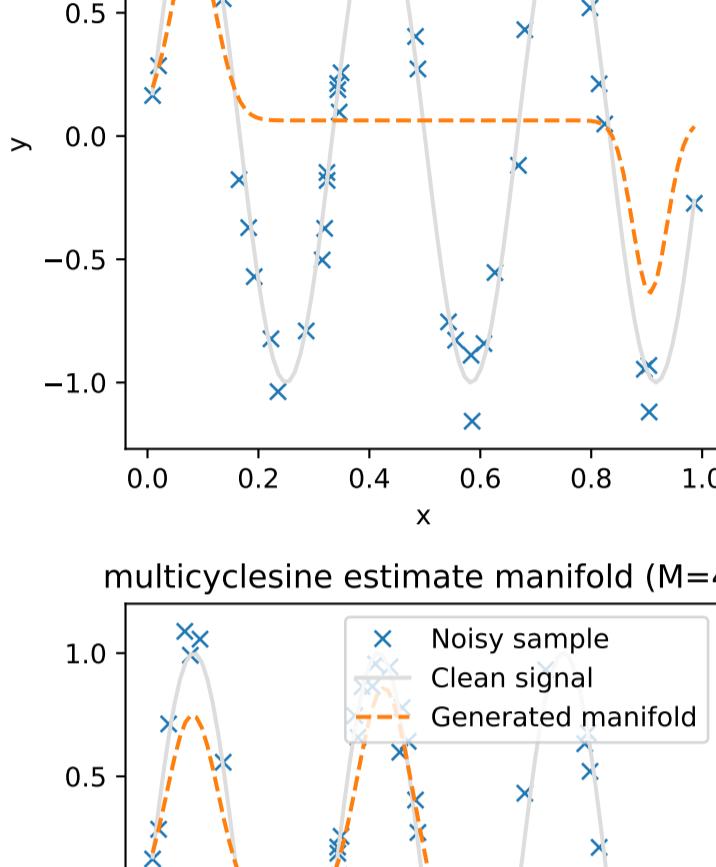
multicyclesine estimate manifold (M=1)



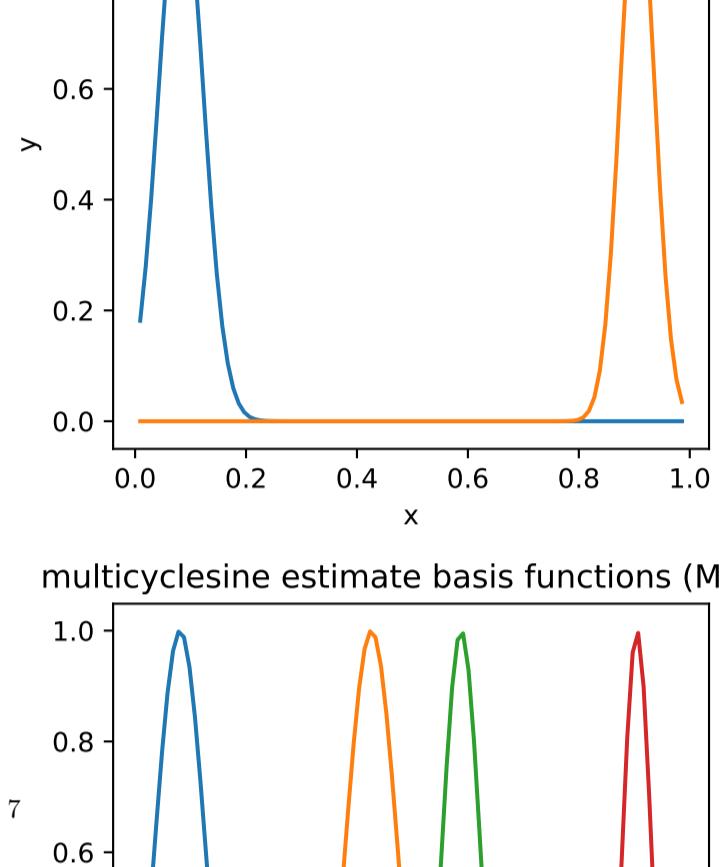
multicyclesine estimate basis functions (M=1)



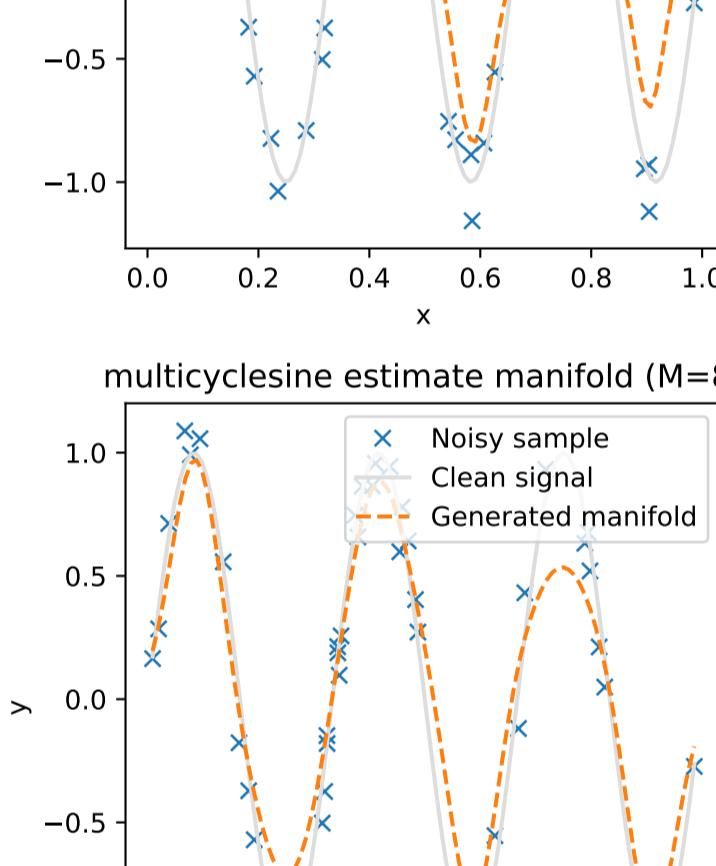
multicyclesine estimate manifold (M=2)



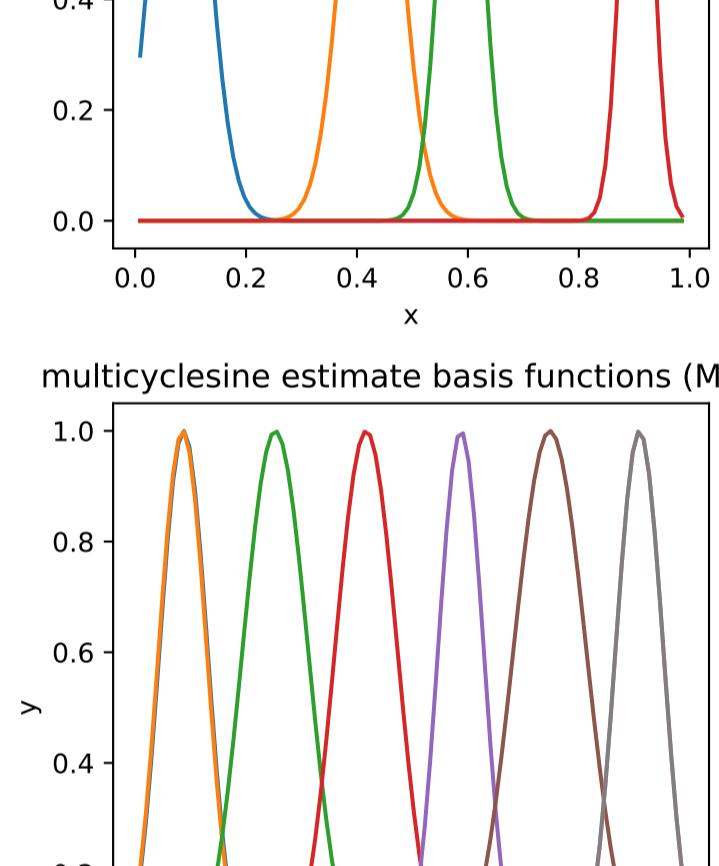
multicyclesine estimate basis functions (M=2)



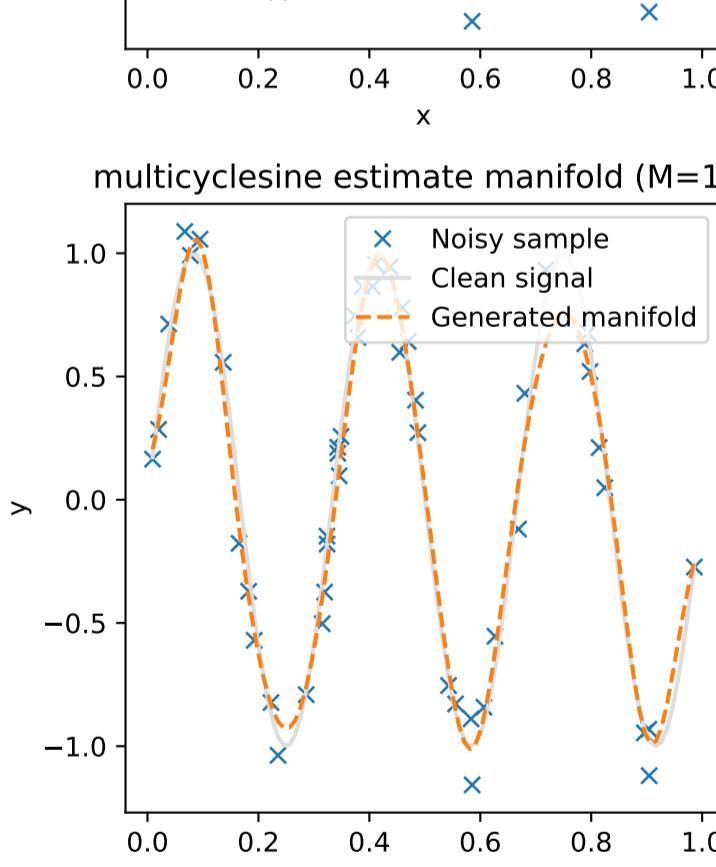
multicyclesine estimate manifold (M=4)



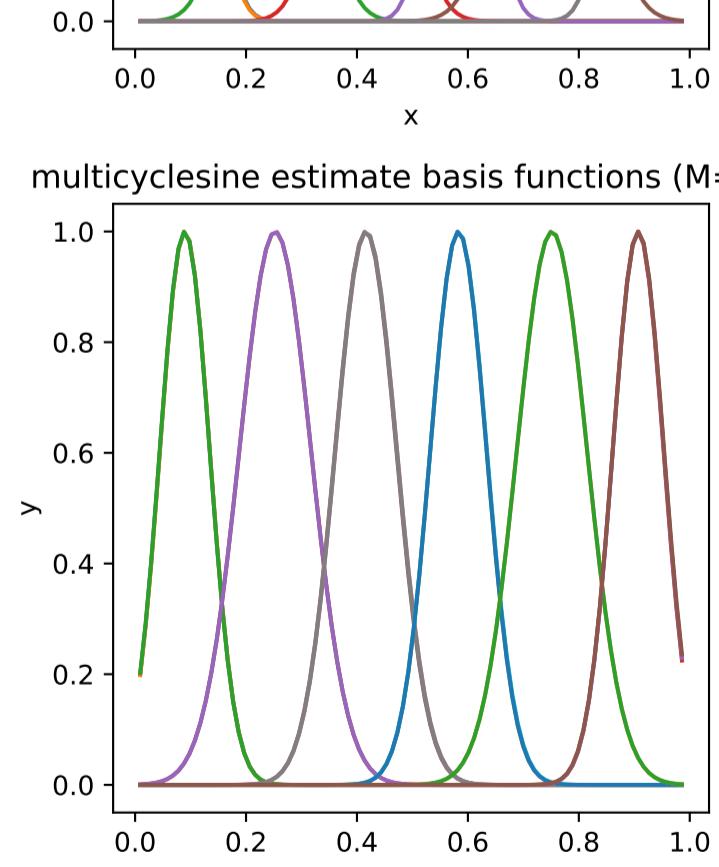
multicyclesine estimate basis functions (M=4)



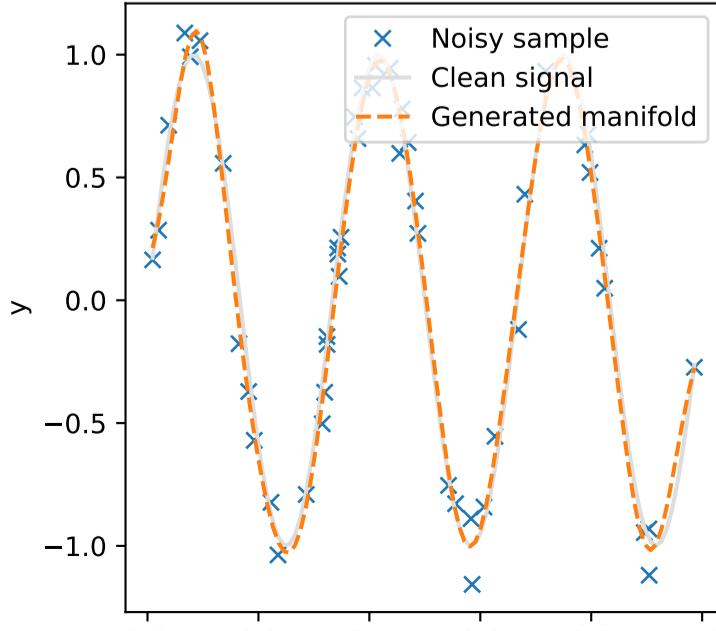
multicyclesine estimate manifold (M=8)



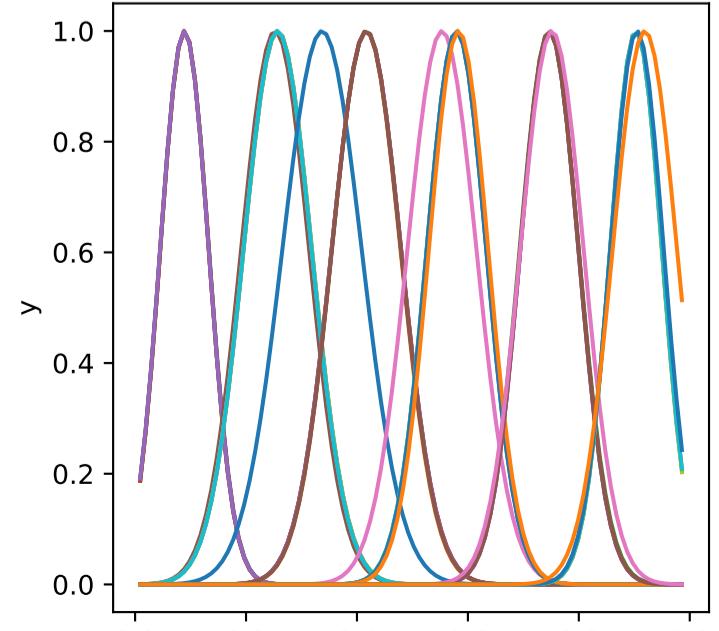
multicyclesine estimate basis functions (M=8)



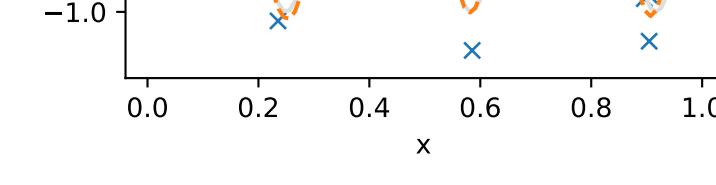
multicyclesine estimate manifold (M=16)



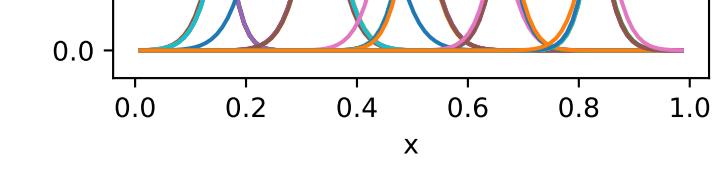
multicyclesine estimate basis functions (M=16)



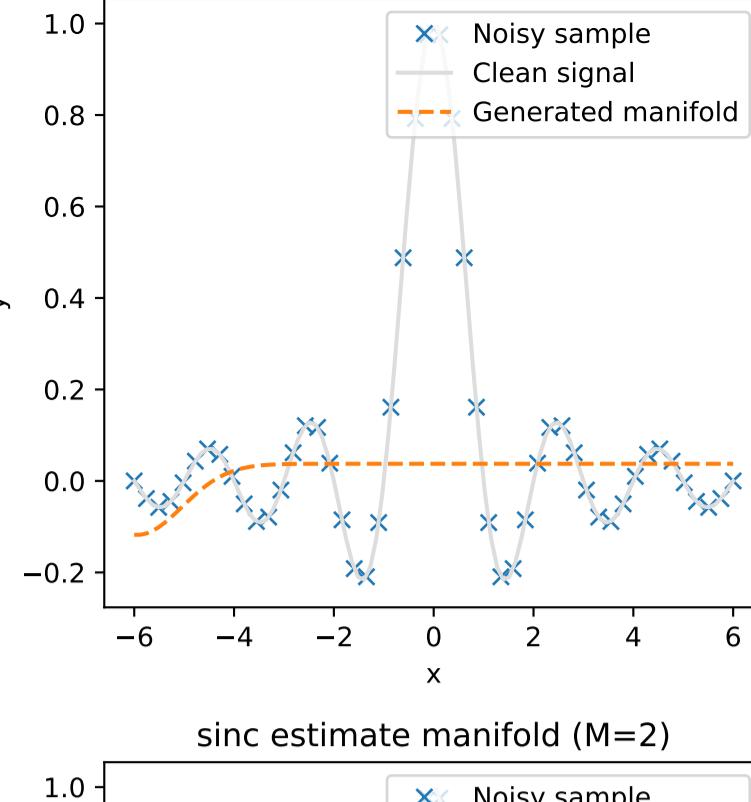
multicyclesine estimate manifold (M=32)



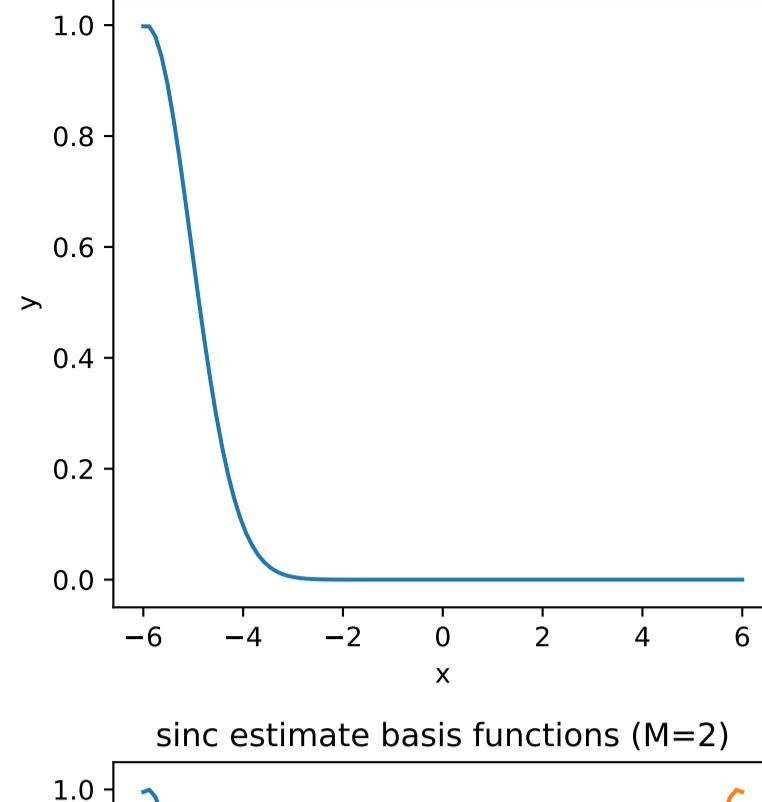
multicyclesine estimate basis functions (M=32)



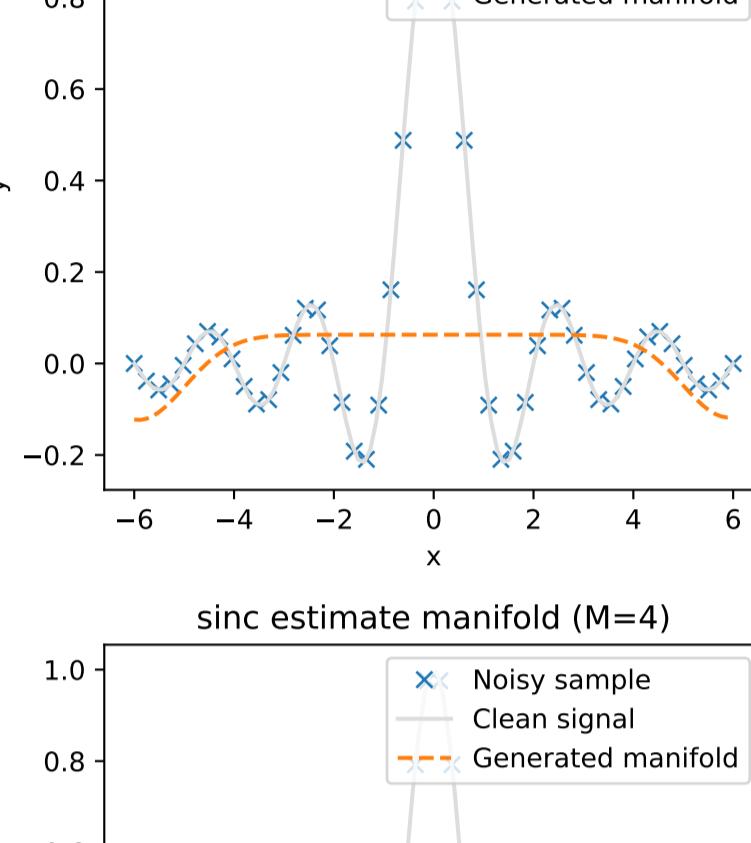
sinc estimate manifold (M=1)



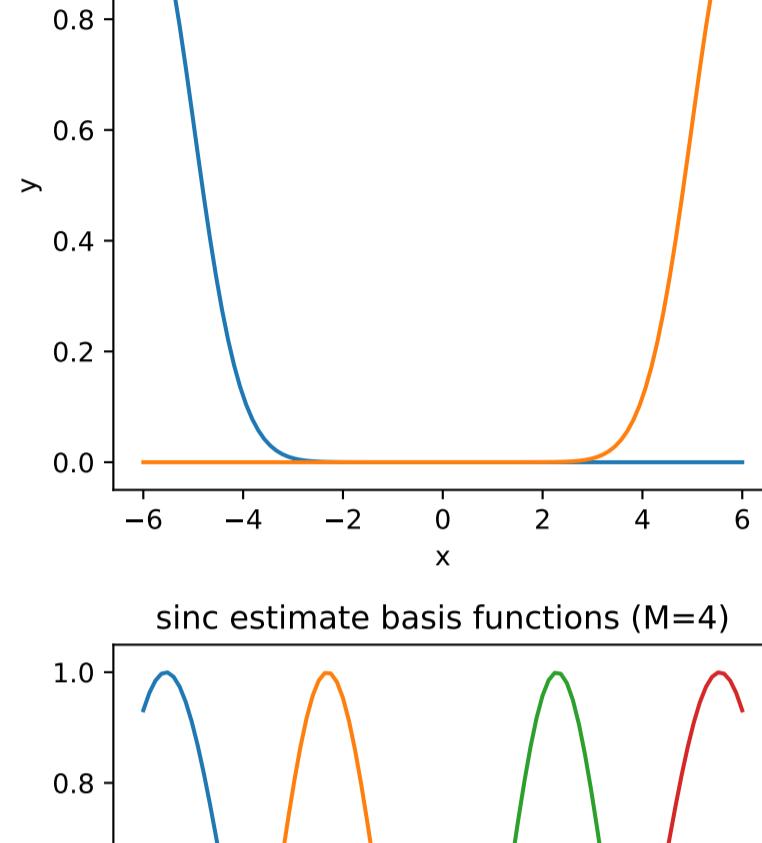
sinc estimate basis functions (M=1)



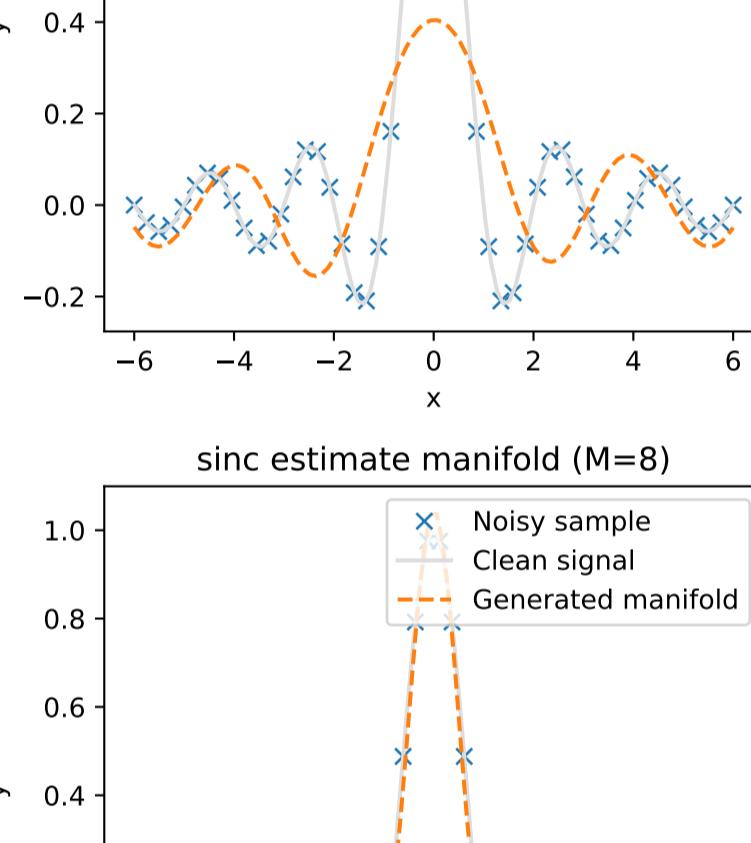
sinc estimate manifold (M=2)



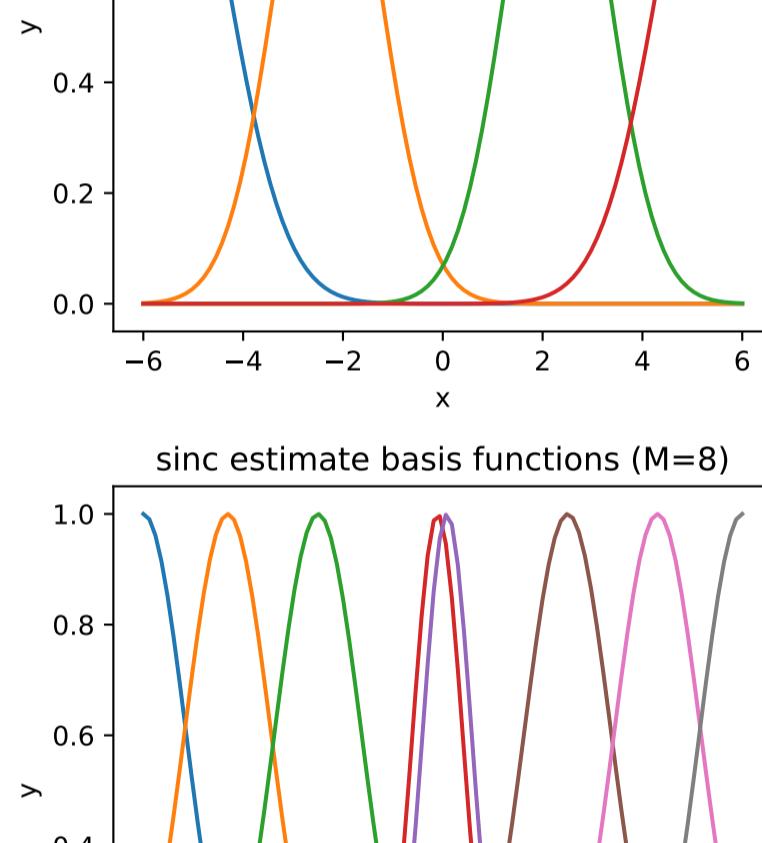
sinc estimate basis functions (M=2)



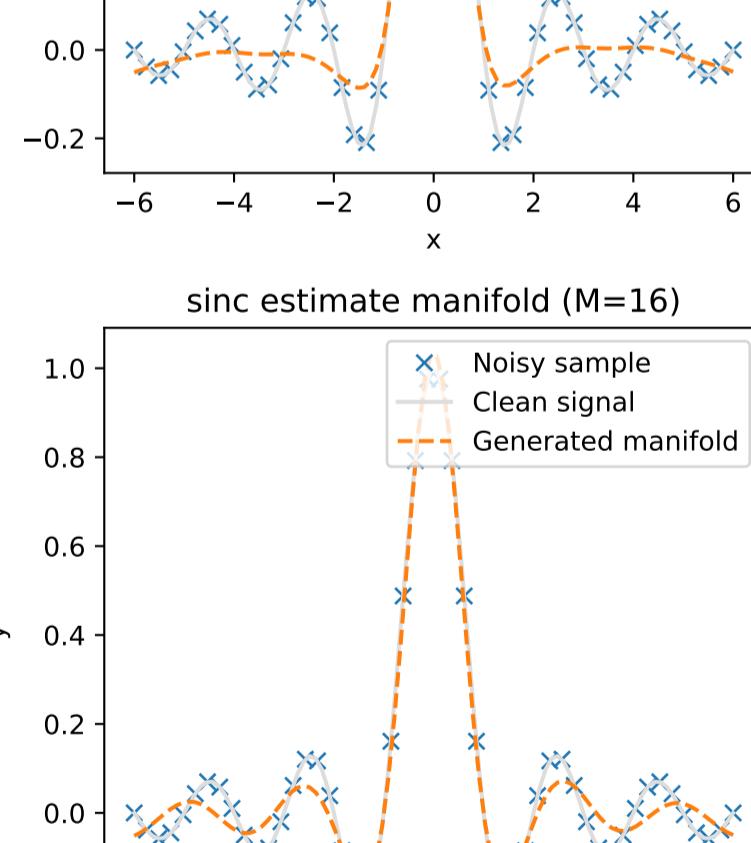
sinc estimate manifold (M=4)



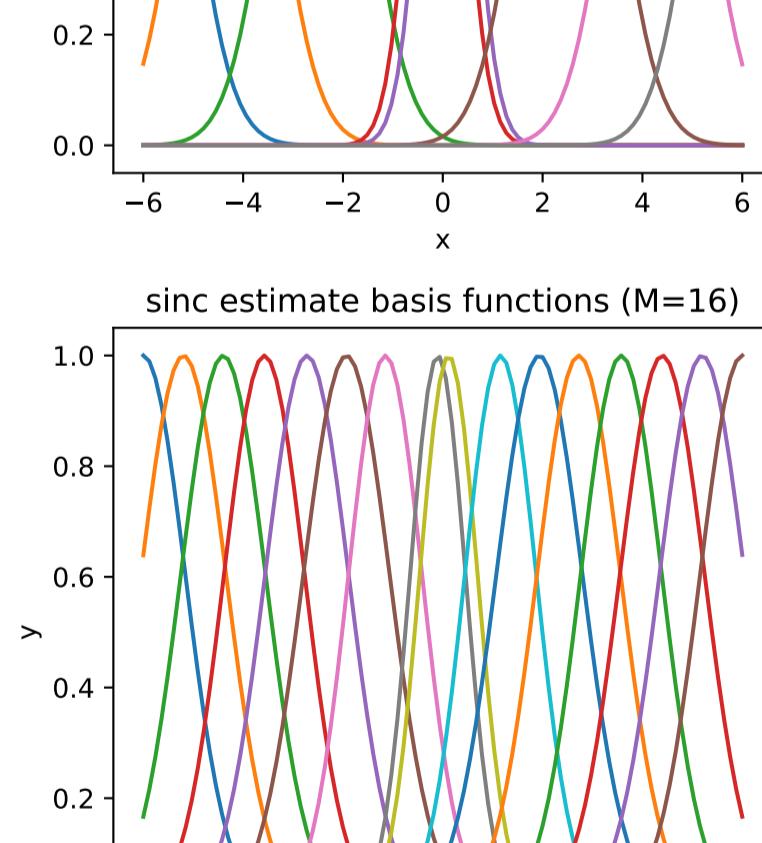
sinc estimate basis functions (M=4)



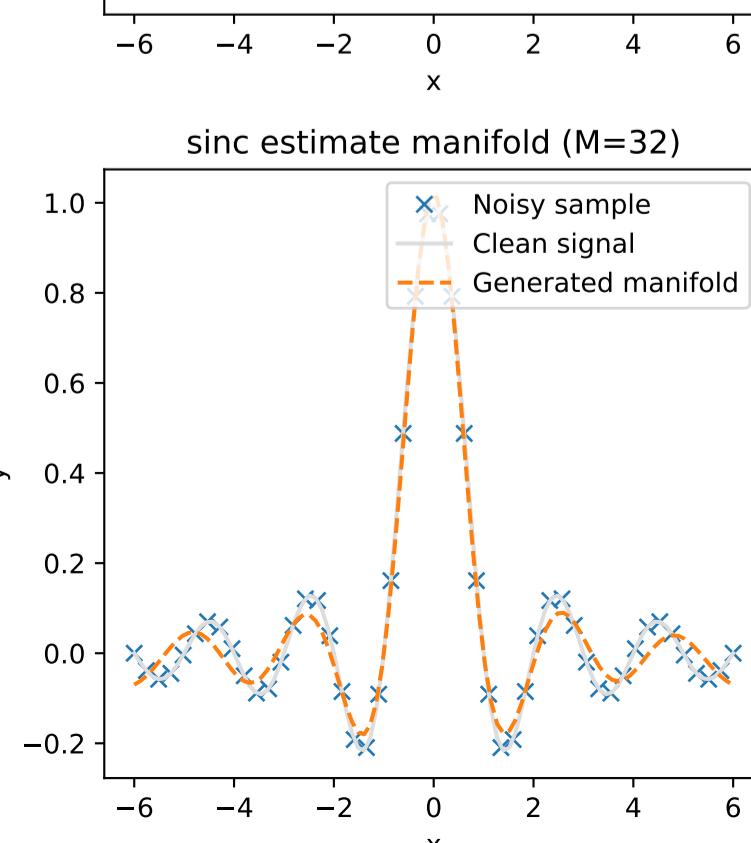
sinc estimate manifold (M=8)



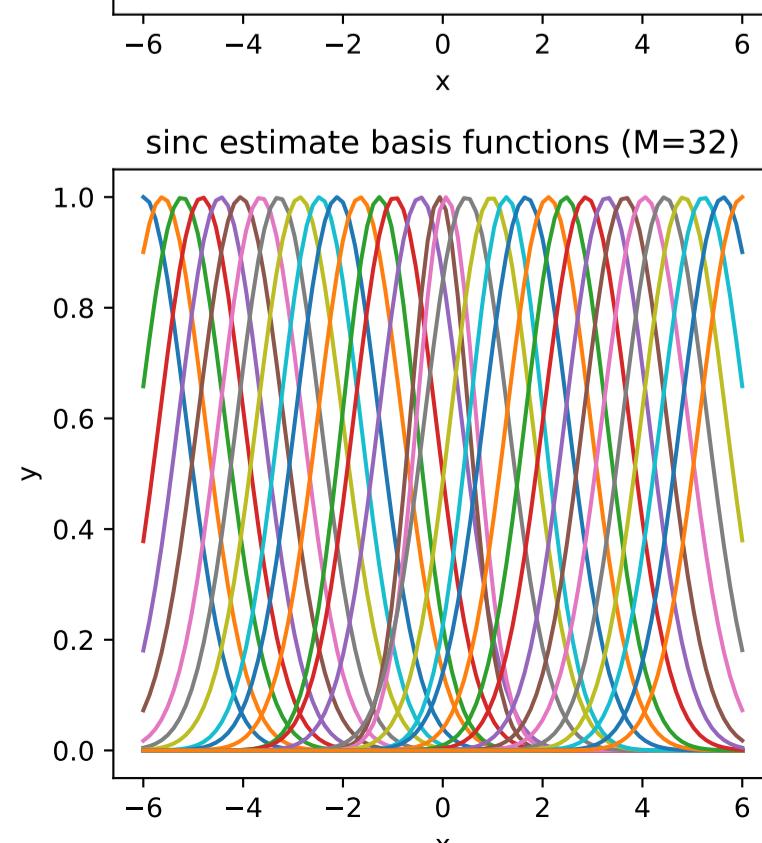
sinc estimate basis functions (M=8)



sinc estimate manifold (M=16)



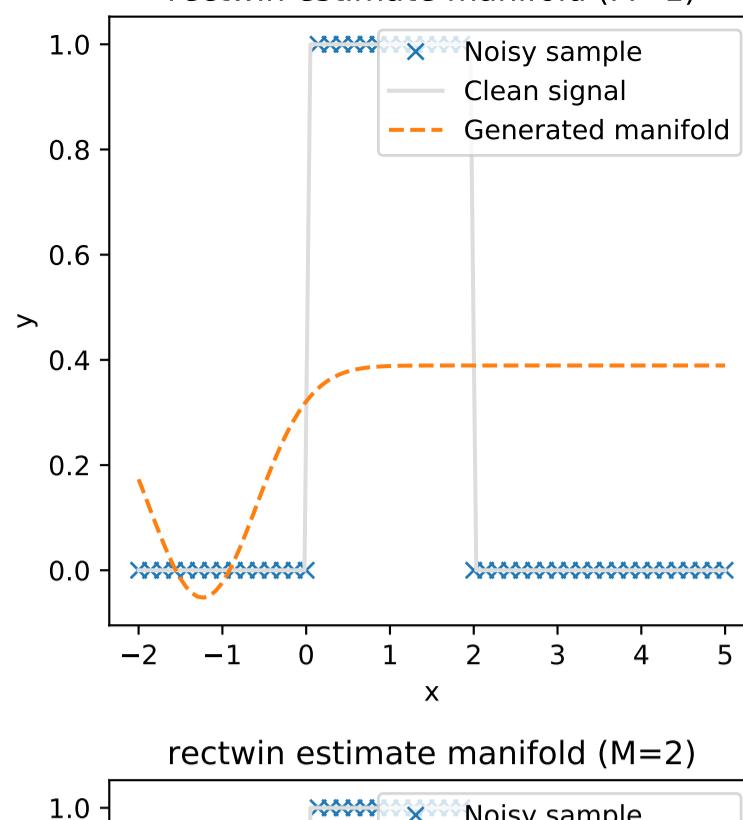
sinc estimate basis functions (M=16)



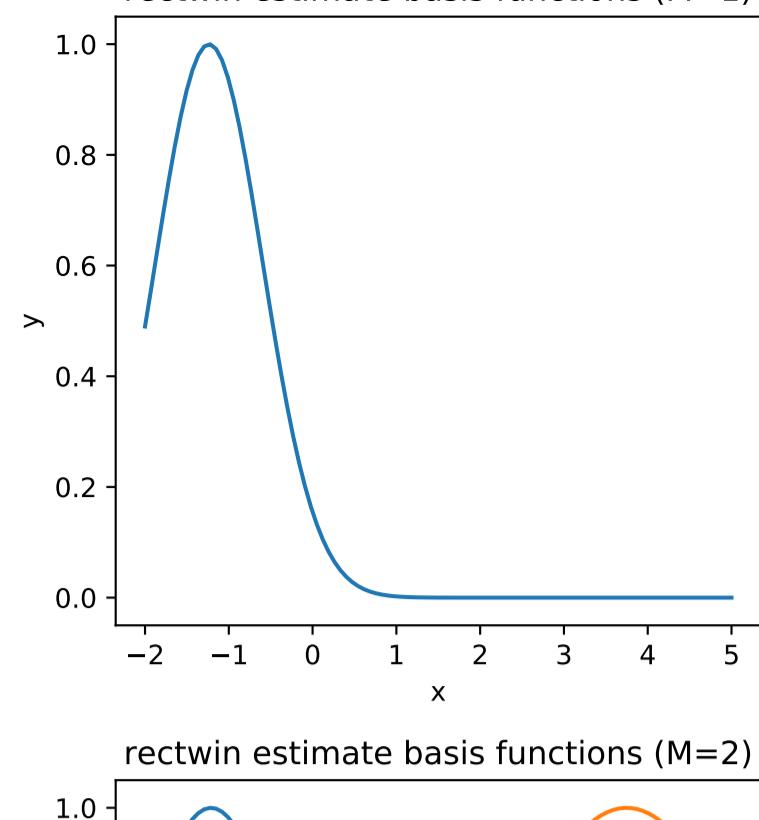
sinc estimate manifold (M=32)

sinc estimate basis functions (M=32)

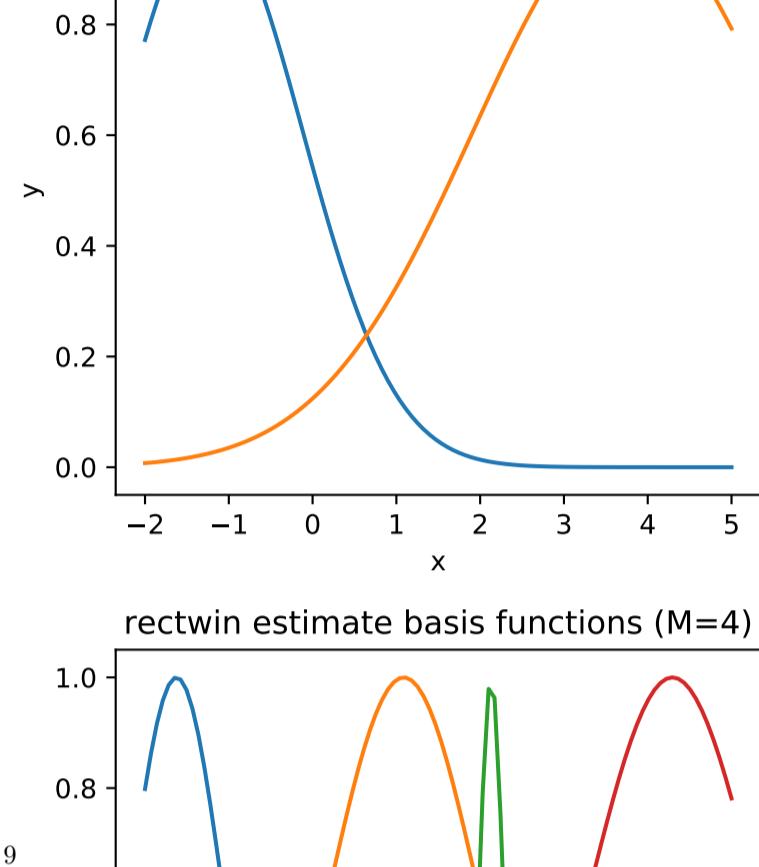
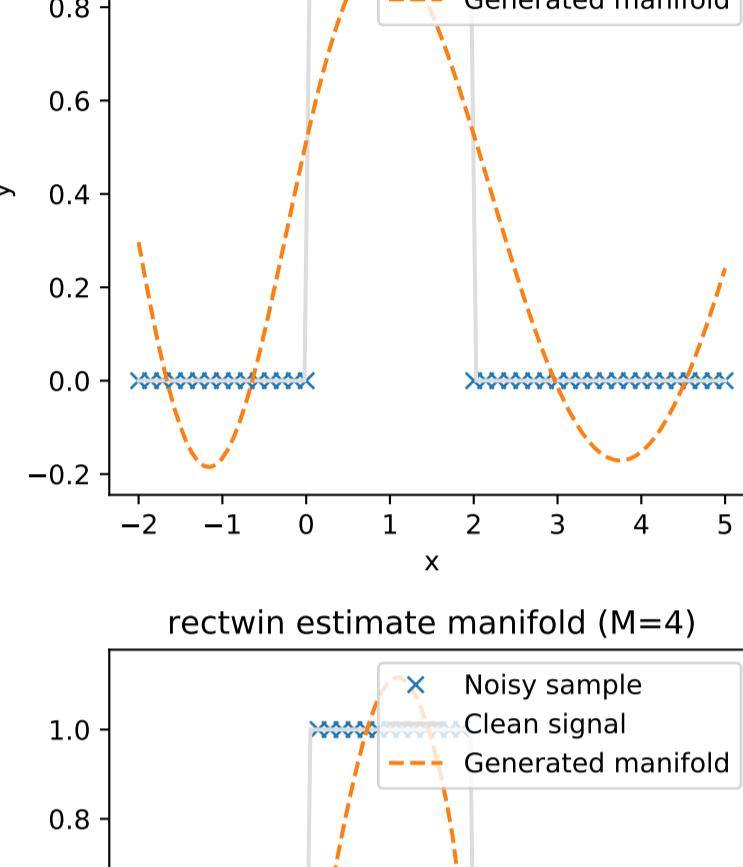
rectwin estimate manifold (M=1)



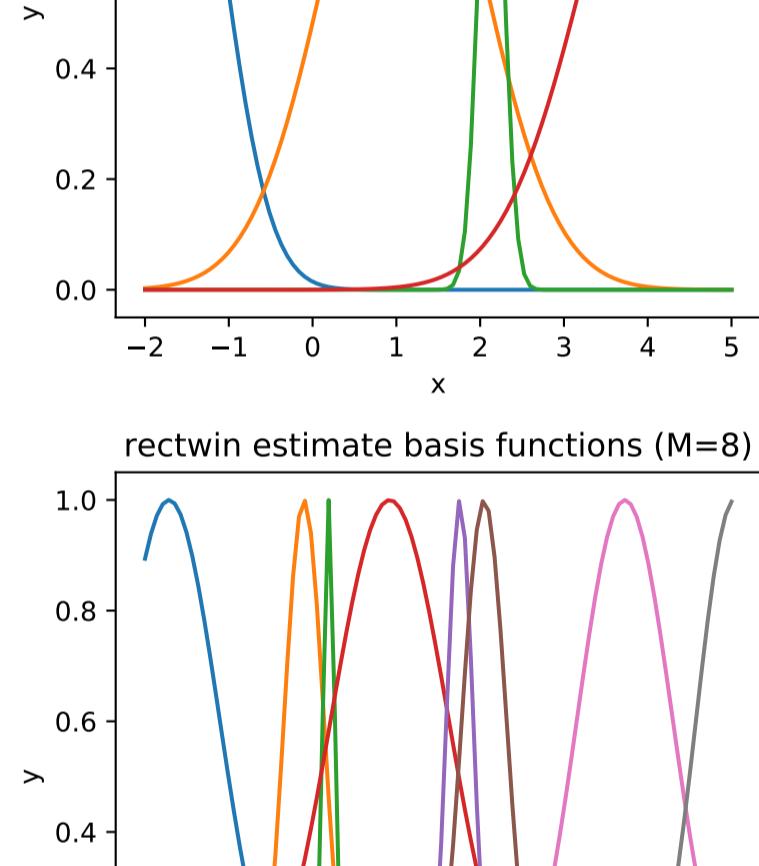
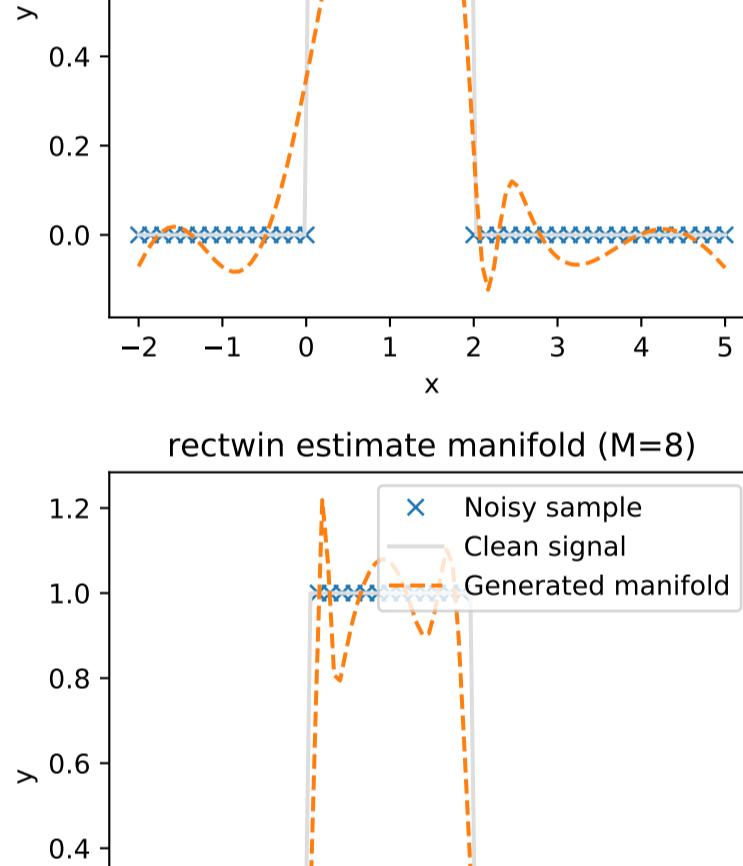
rectwin estimate basis functions (M=1)



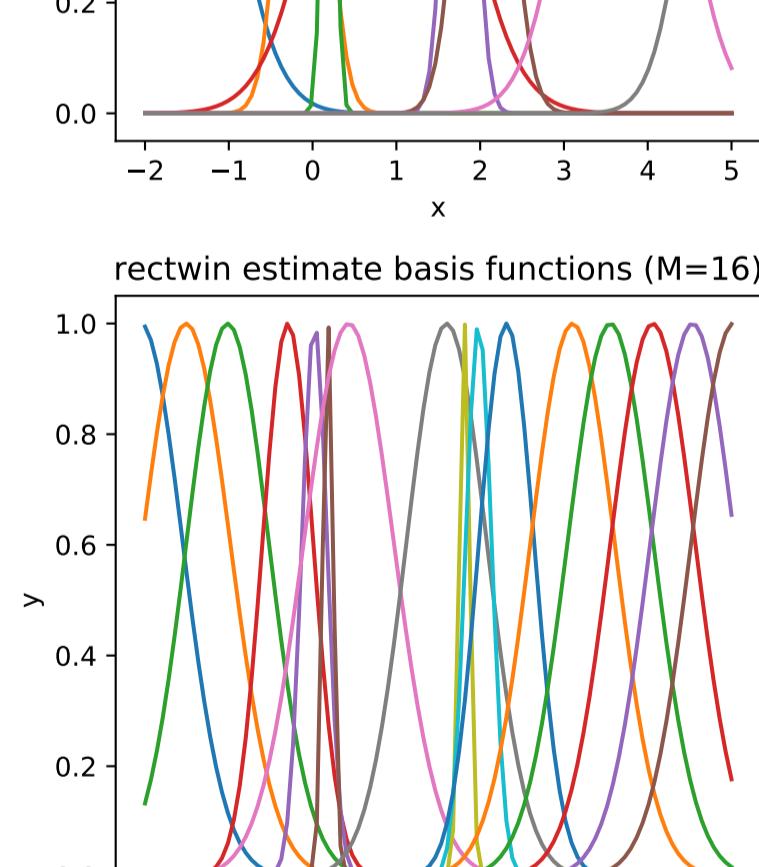
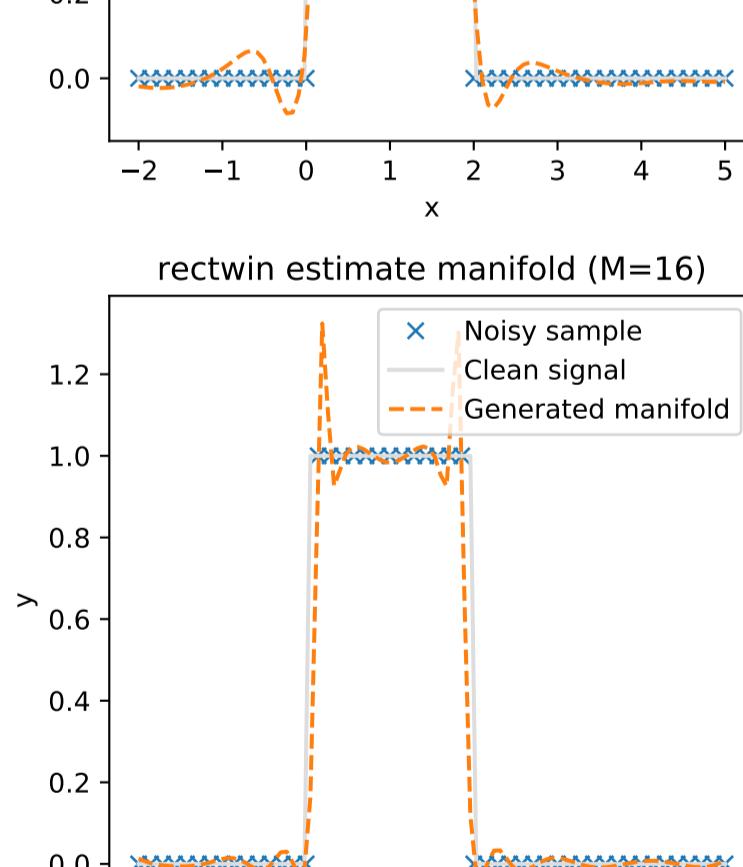
rectwin estimate manifold (M=2)



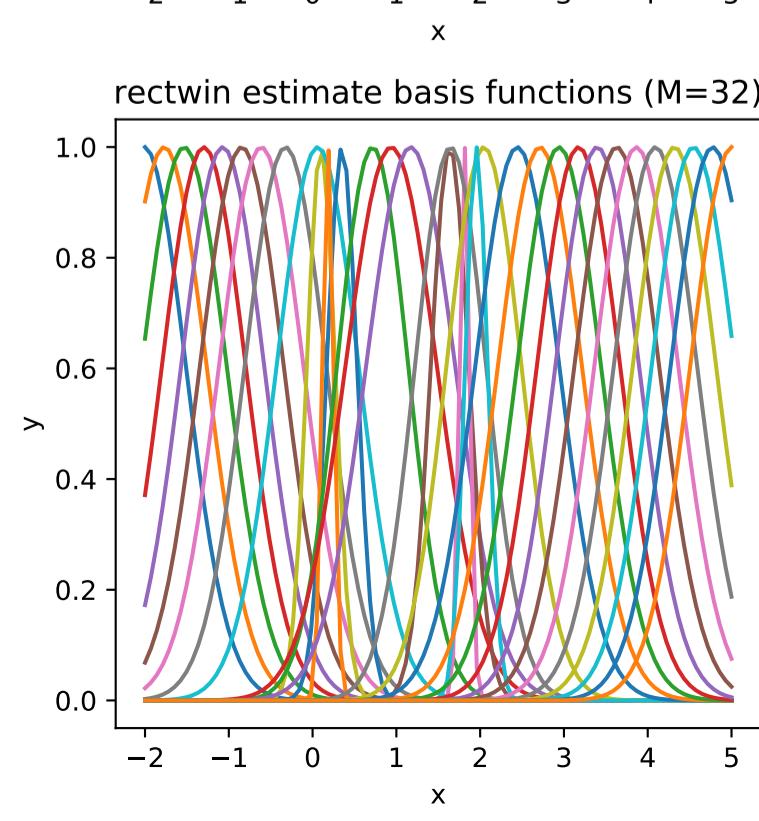
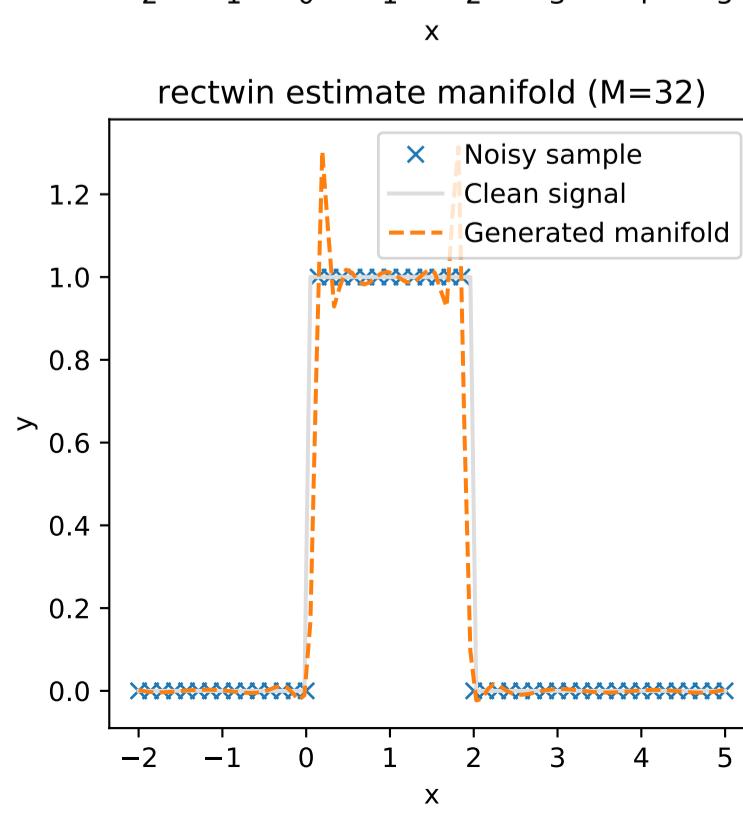
rectwin estimate manifold (M=4)



rectwin estimate manifold (M=8)



rectwin estimate manifold (M=16)



rectwin estimate manifold (M=32)

ECE472 – Project 2

Jonathan Lam

September 16, 2020

Notes on model development

Initialization Like in the last project, initialization was done with Gaussians. The only problems I had were that if I initialized the ReLUs with zero-centered bias matrices, then the convergence appeared to be slower (most likely due to many “dead” ReLUs), so I initialized them with a largely-positive bias (`layers[i]['bias']`). I also realized that as the number of layers increased, I had to downscale the initial weights for the activation functions so that there weren’t convergence issues (`layers[i]['coef']`).

Layer widths At first, I had relatively low layer widths. Following the example of the textbook, which used a two-layer perceptron to model XOR, where the hidden layer had a width of 2, I tried widths of 2, 4, and 8. This didn’t classify all of the points correctly (it seemed to converge, but not near zero), so at this point I added another layer, which only helped a little bit. Only when I had five layers (four with width 10, the last one with width 1) was I able to make the model converge.

After speaking with some classmates (Derek Lee), I learnt that even this was considered a low number, and I experimented with larger sizes, which in general tended to converge faster. The final values I chose (32, 64, 32, 1) are somewhat arbitrary but seem to converge quickly (< 1000 iterations with Adam). Using smaller values (e.g., 16, 16, 16, 1) I can still classify all of the points correctly, but occasionally there are classification errors with only 1000 iterations and would benefit with more iterations, and the boundary is not as smooth.

Layer count Initially, I began with three ReLU layers and one sigmoid layer (this is the same as my final). I only tried up to five layers (adding one additional ReLU), which helped a little when the widths of the layers were small.

Update rule, iteration count, and dynamic step sizes Most of my experimentation worked with a basic gradient descent algorithm with a step size that was manually attenuated with higher iterations, e.g., $\alpha \leftarrow \alpha/2$ every 1000 iterations. This eventually converged (most of the time) without problems, but there were generally many irregular spikes on the loss vs. iterations plot. Typical convergence took 5000-10000 iterations using this scheme.

After reading the paper on Adam, I wanted to see how it would affect my model. I wasn’t expecting much, but this did wonders right away on the convergence (both the smoothness and convergence rate): the same model only required 500-1000 iterations to converge. The loss vs. iteration count plot immediately became smooth.

The end result is a four-layer perceptron using batch gradient descent with the Adam update rule, with three ReLU layers with widths 32, 64, and 32, followed by a sigmoid layer to map onto the probability space. The loss function is a binary cross-entropy function with L2 penalties for the weight matrices W_i (but not the bias matrices b_i). 1000 iterations are sufficient to train this model, which runs in approximately 6s on an i7-7500U processor with integrated graphics.

Source code

```

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

### DEFINE MODEL

# activation function definitions
def relu(X, W, b):
    return tf.nn.relu(tf.transpose(W) @ X + b)

def sigmoid(X, W, b):
    return tf.math.sigmoid(tf.transpose(W) @ X + b)

# multi-layer perceptron classifier model definition
class MultiLayerPerceptronClassifier:

    def __init__(self,
                 sampleDim,           # dimensionality of input
                 layers,              # layer definitions
                 lmbda=0.01,           # l2 penalty coefficient
                 betal=0.9,            # decay factor for first moment (adam)
                 beta2=0.999,          # decay factor for second moment (adam)
                 alpha=0.001):         # upper bound on learning rate (adam)

        self._lmbda = lmbda
        self._betal = betal
        self._beta2 = beta2
        self._alpha = alpha
        self._layers = layers

    # model variables to optimize; initialize with normal distribution
    self._modelVars = [
        {
            'W': tf.Variable(tf.random.normal(
                (layers[i-1]['width'] if i > 0 else sampleDim,
                 layer['width']), dtype=tf.float64) * layer['coef']),
            'b': tf.Variable(tf.random.normal((layer['width'], 1),
                                              dtype=tf.float64) * layer['coef'] + layer['bias'])
        } for i, layer in enumerate(layers)
    ]

    # adam variables: weighted first and second moments of gradients
    self._adamVars = [
        {
            'zW': tf.zeros_like(layerVars['W']),
            'zb': tf.zeros_like(layerVars['b']),
        }
    ]

```

```

        'zW2': tf.zeros_like(layerVars['W']),
        'zb2': tf.zeros_like(layerVars['b']),
    } for layerVars in self._modelVars
]

def f(self, X):
    for layer, layerVars in zip(self._layers, self._modelVars):
        X = layer['actFn'](X, layerVars['W'], layerVars['b'])
    return X

# binary cross-entropy loss with L2 penalty
def bceLossL2Penalty(self, y, yhat):
    loss = -y * tf.math.log(yhat) - (1 - y) * tf.math.log(1 - yhat)
    for layerVars in self._modelVars:
        loss += self._lmbda * tf.nn.l2_loss(layerVars['W'])
    return loss

def step(self, X, y):
    # don't do random batches, just use entire input on every step
    with tf.GradientTape() as tape:
        loss = self.bceLossL2Penalty(y, self.f(X))

    # calculate gradient, store loss
    grad = tape.gradient(loss, self._modelVars)
    self._losses.append(tf.math.reduce_mean(loss))

    # update model and adam variables
    for adam, layerVars, layerGrad \
        in zip(self._adamVars, self._modelVars, grad):

        # update adam moments
        adam['zW'] = self._beta1 * adam['zW'] \
            + (1 - self._beta1) * layerGrad['W']
        adam['zb'] = self._beta1 * adam['zb'] \
            + (1 - self._beta1) * layerGrad['b']
        adam['zW2'] = self._beta2 * adam['zW2'] \
            + (1 - self._beta2) * layerGrad['W']**2
        adam['zb2'] = self._beta2 * adam['zb2'] \
            + (1 - self._beta2) * layerGrad['b']**2

        # adam update rule
        ep = 0.0001 # epsilon to prevent division by zero
        layerVars['W'].assign_sub(self._alpha * adam['zW'] \
            / (tf.math.sqrt(adam['zW2']) + ep))
        layerVars['b'].assign_sub(self._alpha * adam['zb'] \
            / (tf.math.sqrt(adam['zb2']) + ep))

def train(self, X, y, iterations):
    self._losses = []
    for i in range(iterations):
        self.step(X, y)
    return self._losses

### GENERATE SAMPLE DATA

# spiral definition
offset = 2 # offset of spirals from center (radially)

```

```

N = 200                                     # sample count
noiseStd = 0.25                               # sample noise (radially)
spiralEnd = 3.5 * np.pi                        # spiral end (radians)

# generate spirals
t = tf.random.uniform((N,), 0., spiralEnd, dtype=tf.float64)
noise = tf.random.normal((2*N,), 0, noiseStd, dtype=tf.float64)
x1 = (t + noise[:N] + offset) * tf.math.cos(-t)
y1 = (t + noise[:N] + offset) * tf.math.sin(-t)
x2 = (t + noise[N:] + offset) * tf.math.cos(-t + np.pi)
y2 = (t + noise[N:] + offset) * tf.math.sin(-t + np.pi)

# samples of zeroes and ones; sample matrices include both inputs and labels
S0 = tf.concat((x1[tf.newaxis], y1[tf.newaxis],
                 tf.zeros((1, N), dtype=tf.float64)), axis=0)
S1 = tf.concat((x2[tf.newaxis], y2[tf.newaxis],
                 tf.ones((1, N), dtype=tf.float64)), axis=0)
S = tf.concat((S0, S1), axis=1)

# samples predictor matrices, label matrices
X = S[0:2, :]
y = S[2, :][tf.newaxis]

### CREATE AND RUN MODEL

# configure layers, widths, functions; width is the number of outputs for a fn,
# number of inputs inferred from last layer's width (or sample's dimensions)
layers = [
    {'actFn': relu, 'width': 32, 'coef': 0.25, 'bias': 0.5},
    {'actFn': relu, 'width': 64, 'coef': 0.25, 'bias': 0.5},
    {'actFn': relu, 'width': 32, 'coef': 0.25, 'bias': 0.5},
    {'actFn': sigmoid, 'width': 1, 'coef': 0.25, 'bias': 0.},
]
classifier = MultiLayerPerceptronClassifier(2, layers)
losses = classifier.train(X, y, 1000)

### PLOT RESULTS

# plot spirals
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

res = classifier.f(X)[0]
x1 = X[0][res<0.5]
y1 = X[1][res<0.5]
x2 = X[0][res>=0.5]
y2 = X[1][res>=0.5]

# plot original sample points
ax1.plot(x1, y1, 'x', x2, y2, 'x')

# plot manifold
x1, x2 = np.meshgrid(np.linspace(-15, 15, 100), np.linspace(-15, 15, 100))
yhat = np.reshape(classifier.f(np.vstack((x1.flatten(), x2.flatten()))),
                  (100, 100))
ax1.contourf(x1, x2, yhat, 1, vmin=0, vmax=1)
ax1.set_ylim([-16, 16])

```

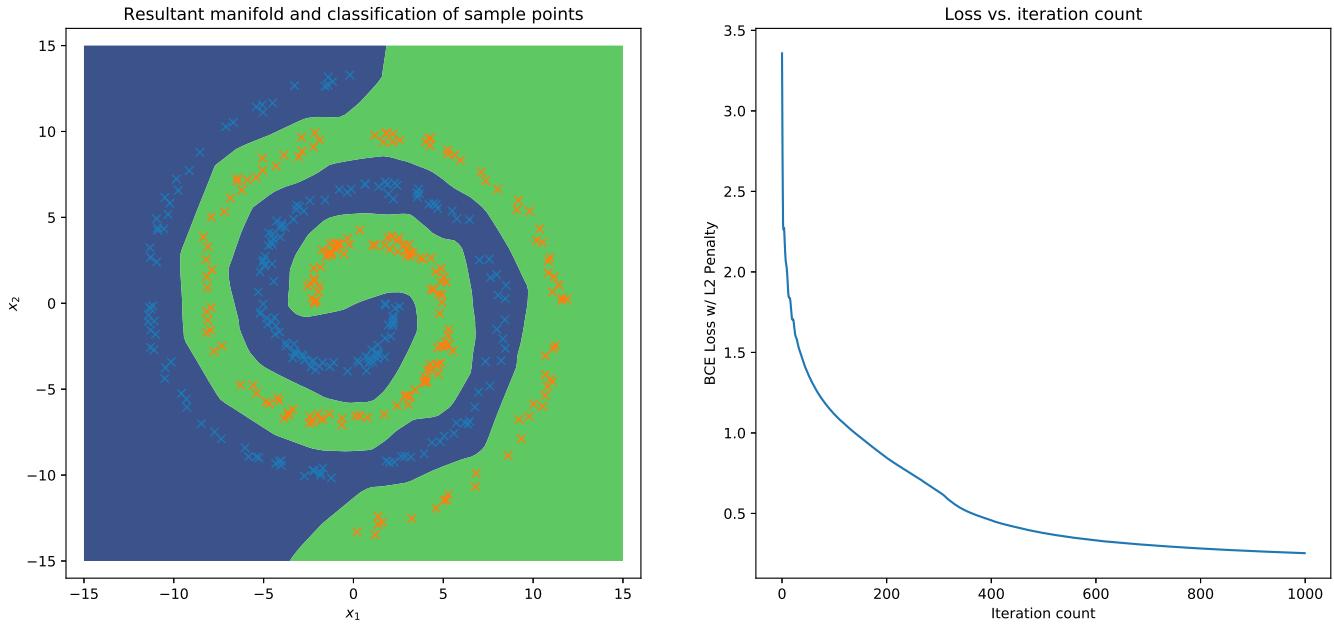
```

ax1.set_xlim([-16, 16])
ax1.set_ylabel('$x_2$')
ax1.set_xlabel('$x_1$')
ax1.set_title('Resultant manifold and classification of sample points')

# plot losses vs. iteration count
ax2.plot(losses)
ax2.set_ylabel('BCE Loss w/ L2 Penalty')
ax2.set_xlabel('Iteration count')
ax2.set_title('Loss vs. iteration count')
plt.show()

```

Plots



The blue crosses are randomly generated samples from one class, and the orange crosses are randomly generated samples from the other class. The dark blue and green regions are where the model predicts that points will fall into those respective classes. All of the points are correctly classified. The model does not look too overfitted, and there is a reasonable margin between the two sample sets. The loss vs. iteration count is smooth and monotonically decreasing, which shows a nice convergence.

ECE472 – Project 3

Jonathan Lam

September 22, 2020

Notes

- The keras library and its builtin utilities were used for this project. kerastuner was used for automatic tuning. (It has a very appealing logging output.) As expected, the final output takes quite a while to iterate over the search space, but consistently produces the desired results (classification accuracy > 95.5%).
- The MNIST dataset files were downloaded from <http://yann.lecun.com/exdb/mnist/> and parsed with ordinary file operations.
- The train dataset (60,000 images) was randomly split into 50,000 train/10,000 validation.
- The neural network comprises some hidden layers and a final softmax (logit) layer. The `tf.keras.losses.SparseCategoricalCrossEntropy` loss was used to evaluate the function and calculate the gradient. The `tf.keras.optimizers.Adam` optimizer was used. Each hidden layer included a dense linear layer with L^2 regularization, batch normalization, a leaky ReLU activation function, and dropout (following the activation function).
- Tunable parameters include:
 - Number of hidden layers
 - Width (of each hidden layer)
 - L^2 penalization coefficient (for each hidden layer)
 - Adam optimizer learning rate

Source Code

The final tuned trained model generated by this code produces 97.60% accuracy on the test dataset. (This can be seen at the bottom of the code output in the following section.)

```
### Jonathan Lam
### Prof. Curro
### ECE 472 Deep Learning
### 9 / 20 / 20
### Project 3

### MNIST data from http://yann.lecun.com/exdb/mnist/
```

```

import numpy as np
import tensorflow as tf
import kerastuner as kt

# 32-bit big-endian byte buffer to int
def be32_to_int(buf):
    return np.ndarray(shape=(1,), dtype='>i4', buffer=buf)[0]

# see URL of MNIST data source for data layout
def read_mnist_file(filename, is_labels):
    with open(filename, 'rb') as mnist_file:
        # read header info
        be32_to_int(mnist_file.read(4))      # read and discard magic number
        num_imgs = be32_to_int(mnist_file.read(4))
        num_rows = None if is_labels else be32_to_int(mnist_file.read(4))
        num_cols = None if is_labels else be32_to_int(mnist_file.read(4))

        # read raw image data
        data = np.frombuffer(mnist_file.read(), dtype=np.uint8)
    return num_imgs, num_rows, num_cols, data

# read train/validation dataset files
num_train_file_imgs, num_rows, num_cols, train_file_imgs = \
    read_mnist_file('train-images-idx3-ubyte', False)
_, _, train_file_lbls = read_mnist_file('train-labels-idx1-ubyte', True)

# read test dataset files
num_test, _, _, test_imgs = read_mnist_file('t10k-images-idx3-ubyte', False)
_, _, test_lbls = read_mnist_file('t10k-labels-idx1-ubyte', True)

# P := # features (pixels) = num_rows x num_cols
# N := num_imgs
# reshape features to NxP, scale to [0, 1)
train_file_imgs = train_file_imgs.reshape(-1, num_rows * num_cols) / 255.
test_imgs = test_imgs.reshape(-1, num_rows * num_cols) / 255.
# reshape labels to Nx1
train_file_lbls = train_file_lbls.reshape(-1, 1)
test_lbls = test_lbls.reshape(-1, 1)

# split train/validation dataset into train and validation datasets
indices = np.arange(num_train_file_imgs)
np.random.shuffle(indices)
cutoff = 50000
train_imgs, val_imgs, train_lbls, val_lbls = \
    train_file_imgs[indices[:cutoff],:], \
    train_file_imgs[indices[cutoff:],:], \
    train_file_lbls[indices[:cutoff],:], \
    train_file_lbls[indices[cutoff:],:]

# tunable model builder
def build_model(hp):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.Input(shape=(num_rows * num_cols,)))

    # tunable hidden layers
    for i in range(hp.Int('layers', 3, 5)):
        model.add(tf.keras.layers.Dense(

```

```

        units=hp.Choice('units' + str(i), [64, 128, 256]),
        kernel_regularizer=tf.keras.regularizers.L1L2(
            l2=hp.Float('l2_' + str(i), 1e-8, 1e-4, sampling='log'))))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.LeakyReLU())
model.add(tf.keras.layers.Dropout(
    rate=hp.Float('dropout_' + str(i), 0.1, 0.5, step=0.2)))

# final layer: calculate logits
model.add(tf.keras.layers.Dense(
    units=10,
    kernel_regularizer=tf.keras.regularizers.L1L2(
        l2=hp.Float('l2_final', 1e-8, 1e-4, sampling='log'))))

# set up model loss and optimizer
model.compile(
    optimizer=tf.keras.optimizers.Adam(
        learning_rate=hp.Float('learning_rate', 1e-4, 1e-1, sampling='log')),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])

return model

# create kerastuner hyperband tuner
tuner = kt.Hyperband(build_model, objective='val_acc', max_epochs=10)

# print search space summary
tuner.search_space_summary()

# search for best hyperparameters, evaluate on validation set
tuner.search(train_imgs, train_lbls, validation_data=(val_imgs, val_lbls))

# print best result
tuner.results_summary()

# train model with best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
model = tuner.hypermodel.build(best_hps)
model.fit(train_imgs, train_lbls, epochs=10)

# evaluate model on test dataset
print('Evaluating on test dataset:')
model.evaluate(test_imgs, test_lbls, verbose=2)

```

Sample Output

All but the final kerastuner trial is omitted for brevity. This is a “sample” because the validation and training sets were randomly partitioned from the official MNIST training data file.

```

/usr/bin/python3.7 /home/jon/Documents/ece472/proj3/proj3.py
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/
    ↪ init_ops.py:1251: calling VarianceScaling.__init__ (from tensorflow.python.ops.
    ↪ init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
    ↪ constructor

```

```

[Search space summary]
|-Default search space size: 12
> layers (Int)
|-default: None
|-max_value: 5
|-min_value: 3
|-sampling: None
|-step: 1
> units0 (Choice)
|-default: 64
|-ordered: True
|-values: [64, 128, 256]
> l2_0 (Float)
|-default: 1e-08
|-max_value: 0.0001
|-min_value: 1e-08
|-sampling: log
|-step: None
> dropout_0 (Float)
|-default: 0.5
|-max_value: 0.5
|-min_value: 0.1
|-sampling: None
|-step: 0.2
> units1 (Choice)
|-default: 64
|-ordered: True
|-values: [64, 128, 256]
> l2_1 (Float)
|-default: 1e-08
|-max_value: 0.0001
|-min_value: 1e-08
|-sampling: log
|-step: None
> dropout_1 (Float)
|-default: 0.5
|-max_value: 0.5
|-min_value: 0.1
|-sampling: None
|-step: 0.2
> units2 (Choice)
|-default: 64
|-ordered: True
|-values: [64, 128, 256]
> l2_2 (Float)
|-default: 1e-08
|-max_value: 0.0001
|-min_value: 1e-08
|-sampling: log
|-step: None
> dropout_2 (Float)
|-default: 0.5
|-max_value: 0.5
|-min_value: 0.1
|-sampling: None
|-step: 0.2
> l2_final (Float)
|-default: 1e-08

```

```

|-max_value: 0.0001
|-min_value: 1e-08
|-sampling: log
|-step: None
> learning_rate (Float)
|-default: 0.0001
|-max_value: 0.1
|-min_value: 0.0001
|-sampling: log
|-step: None

... keras tuning output truncated ...

[Trial summary]
|-Trial ID: 901d97905dbd0f468748596d357a3183
|-Score: 0.9519000053405762
|-Best step: 0
> Hyperparameters:
|-dropout_0: 0.30000000000000004
|-dropout_1: 0.30000000000000004
|-dropout_2: 0.30000000000000004
|-dropout_3: 0.30000000000000004
|-dropout_4: 0.30000000000000004
|-l2_0: 8.353674406062614e-06
|-l2_1: 1.1390330286140142e-05
|-l2_2: 9.532844285293838e-08
|-l2_3: 1.2916292421935892e-08
|-l2_4: 1.6675587436184615e-05
|-l2_final: 8.072807530434067e-07
|-layers: 5
|-learning_rate: 0.00044957516806020043
|-tuner/bracket: 1
|-tuner/epochs: 4
|-tuner/initial_epoch: 0
|-tuner/round: 0
|-units0: 128
|-units1: 128
|-units2: 256
|-units3: 128
|-units4: 128
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [=====] - 10s 202us/sample - loss: 0.6770 - acc:
    ↪ 0.8044 - val_loss: 0.2488 - val_acc: 0.9309
Epoch 2/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.3694 - acc:
    ↪ 0.8967 - val_loss: 0.1911 - val_acc: 0.9472
Epoch 3/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.2932 - acc:
    ↪ 0.9178 - val_loss: 0.1636 - val_acc: 0.9555
Epoch 4/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.2540 - acc:
    ↪ 0.9301 - val_loss: 0.1448 - val_acc: 0.9600
Epoch 5/10
50000/50000 [=====] - 10s 193us/sample - loss: 0.2217 - acc:
    ↪ 0.9382 - val_loss: 0.1321 - val_acc: 0.9637
Epoch 6/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.1974 - acc:

```

```

    ↪ 0.9454 - val_loss: 0.1189 - val_acc: 0.9667
Epoch 7/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.1862 - acc:
    ↪ 0.9482 - val_loss: 0.1188 - val_acc: 0.9658
Epoch 8/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.1718 - acc:
    ↪ 0.9520 - val_loss: 0.1143 - val_acc: 0.9673
Epoch 9/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.1621 - acc:
    ↪ 0.9550 - val_loss: 0.1053 - val_acc: 0.9711
Epoch 10/10
50000/50000 [=====] - 10s 194us/sample - loss: 0.1507 - acc:
    ↪ 0.9578 - val_loss: 0.1051 - val_acc: 0.9705
Evaluating on test dataset:
10000/10000 - 1s - loss: 0.0904 - acc: 0.9760

```

80% classification accuracy with fewer parameters

Attempt 1

A single softmax layer without any regularization (the rest of the program and the model are unchanged) produced 92.65% accuracy. The number of weights is

$$\text{num_weights} = \text{size } W_{dense} + \text{size } b_{dense} = (28 \times 28) \times 10 + 10 = 7850$$

```

model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(num_rows * num_cols,)))
model.add(tf.keras.layers.Dense(units=10))

```

I was able to check the number of weights using

```
np.array([w.size for layer in model.layers for w in layer.get_weights()]).sum()
```

Attempt 2

By performing an average pooling (5x5 pools with stride 5), I was able to maintain a 82.6% accuracy and greatly reduce the number of weights.

$$\text{num_weights} = \text{size } W_{dense} + \text{size } b_{dense} = (5 \times 5) \times 10 + 10 = 260$$

```

# reshape inputs to allow for pooling correctly
train_imgs = train_imgs.reshape(-1, num_rows, num_cols, 1) / 255.
test_imgs = test_imgs.reshape(-1, num_rows, num_cols, 1) / 255.

# ...

model.add(tf.keras.layers.AveragePooling2D(pool_size=(5, 5)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=10))

```

Attempt 3

By adding a convolutional layer, I was able to increase the pool sizes and further decrease the number of weights:

$$\text{num_weights} = \text{filters} \times (\text{filter_size}+1) + \text{size } W_{dense} + \text{size } b_{dense} = 2 \times (5 \times 5 + 1) + 8 \times 10 + 10 = 142$$

This achieved an accuracy of 85.25%.

```
# reshape inputs to allow for pooling correctly
train_imgs = train_imgs.reshape(-1, num_rows, num_cols, 1) / 255.
test_imgs = test_imgs.reshape(-1, num_rows, num_cols, 1) / 255.

# ...

model.add(tf.keras.layers.Conv2D(filters=2, kernel_size=(5,5)))
model.add(tf.keras.layers.MaxPool2D(pool_size=(9,9)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=10,
                               kernel_regularizer=tf.keras.regularizers.L1L2(l2=0.0001)))
```

Other attempts

To try to reduce the number of features, I tried downsizing and center-cropping the images. I also tried multiple small convolutional layers. Neither of these methods really got the number of weights to decrease much more than in Attempt 3.

Unattempted attempts

If I had more time to spare on this homework, I think the goal would be to use some complex hardcoded filters that would make each of the numbers linearly spaced after a convolution and filtering layer, so that a regression can be performed to classify the data (since this requires much fewer parameters in the final step than a softmax). At this point however, the point might be moot because the filters would not be learned (although we could still learn the weights for the final dense layer).

ECE472 – Project 4

Jonathan Lam

October 1, 2020

Project description: Training a (ResNet) CNN on CIFAR-10, CIFAR-100. CIFAR-10 accuracy should be state-of-the-art, and CIFAR-100 top-5 accuracy should be at least 80%.

Contents

1 Model	2
1.1 Dataset	2
1.2 Image preprocessing	2
1.3 Structure	2
1.4 ResNet blocks	2
1.5 Regularization	2
1.6 Hyperparameter selection/tuning	3
1.7 Differences between CIFAR-10 and CIFAR-100 models	3
2 Notes on implementation and training	3
3 Results	4
4 Acknowledgments	5
5 Source code	5
5.1 Setup	5
5.2 Data entry (CIFAR-10)	5
5.3 Data entry (CIFAR-100)	6
5.4 Model	6
5.5 Image preprocessing/augmentation and training	8
5.6 Model evaluation	8
6 Code output	8
6.1 CIFAR-10	8
6.2 CIFAR-100	11

1 Model

1.1 Dataset

CIFAR datasets were the Python datasets downloaded from [1]. Each dataset was already split into 50000/10000 train/test. Images are 32x32 color images, and the (categorical) labels are 0-9 for CIFAR-10 and 0-99 (“fine labels”) for CIFAR-100. The samples are equally split between the different categories.

1.2 Image preprocessing

The pixel values were manually standardized to a $N(0, 1)$ distribution, and then augmented using `tf.keras.preprocessing.image.ImageDataGenerator`. This involves slight shifting, vertical and horizontal flipping, and some angle rotation. See the source code for more details.

1.3 Structure

I used the structure of ResNet-34, pictured in Figure 1, as rough guidance for what an overall network structure should look like. In the end, the number of filters per layer and the number of layers was varied to try to decrease training time and increase accuracy.

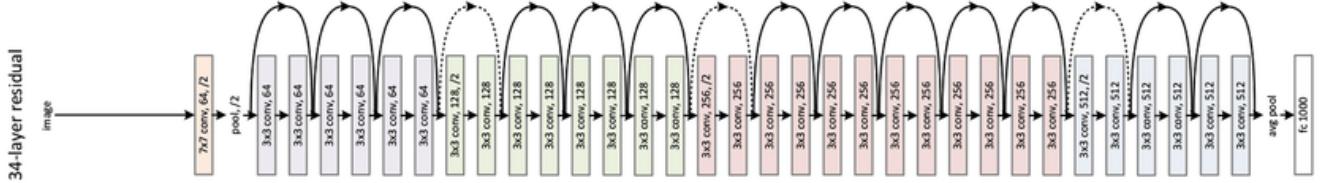


Figure 1: ResNet-34 overall structure. Source: [4]

1.4 ResNet blocks

The individual blocks were the improved ResNet units described in [3]. Namely, these were the “pre-activation” ResNet blocks proposed in that paper, pictured in Figure 2.

In my model, ELUs were used in the place of ReLUs, similarly to the last project. There was also a dropout layer at the end of every ResNet unit for regularization. The he-normal initialization method was used for convolutional layer weights.

1.5 Regularization

A small dropout regularization was performed in each ResNet block. This doesn’t show up in the ResNet papers [2, 3], but I wanted to try using it since we covered it in class.

Similar to the MNIST classification project, L2 regularization was performed on the weights (this time for the filters on the convolutional layers).

Since the accuracy (CIFAR-10) and top-5 accuracy (CIFAR-100) were similar between the training and test datasets, I believe that this level of regularization is sufficient. In this particular training

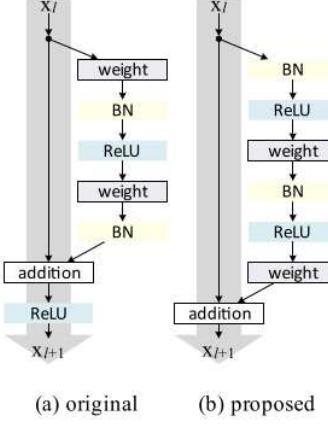


Figure 2: ResNet block structure. (a) The original structure proposed in the original ResNet formulation [2]; (b) The improved structure proposed in [3]. Source: [3]

case for CIFAR-10, the test dataset accuracy is slightly higher than the training dataset accuracy (93.42% on the test dataset as opposed to 90.22% on the training dataset), which is just due to chance.

1.6 Hyperparameter selection/tuning

Hyperparameters were selected manually (see Figure 3) and was not tuned systematically by a builtin tuner like `kerastuner` on a validation set. The reasoning for this is given in the following section (i.e., time constraints). With more time performance could probably be improved further with hyperparameter tuning.

1.7 Differences between CIFAR-10 and CIFAR-100 models

The model used was between the two models was the same, except that the final dense layer had different widths due to the nature of softmax (10 for CIFAR-10, and 100 for CIFAR-100). The only differences were in the data entry (i.e., different filenames, and for CIFAR-100 we were looking at the “fine labels” field rather than the “labels” field of the input) and in the model evaluation: for CIFAR-10, the performance metric was classification accuracy; for CIFAR-100, the performance metric was top-5 classification accuracy.

2 Notes on implementation and training

- Most of the training was performed on Google Colab. Initially (and for all of the previous projects), I had been running the Python code on my desktop computer (i7-2600, no TensorFlow-compatible GPU) and laptop (i7-7500U, no TensorFlow-compatible GPU), both

Hyperparameter	Selected value	Justification
L2 coefficient	0.0001	The default value for <code>tf.keras.regularizers.L2</code> is 0.01 but that seemed to make the convergence much slower. Choosing a much smaller value did the trick.
Learning rate (Adam)	$0.001(0.99)^{\text{epoch}}$	I had originally used the Adam optimizer with its default learning rate, but manually changing the maximum learning rate seemed to help with convergence with higher epochs.
ResNet blocks	12	ResNet-34 includes 17 ResNet blocks, but I reduced the number to try to reduce epoch time. It still meets the desired results after 100 epochs.
# Filters	32, 64, 128	Similar to ResNet-34, as you go deeper in the network you have a higher number of filters for convolutional layers. I chose smaller values so that it would train faster.
Epochs	100	I guess this could be “set” using early stopping, but using the fixed value of 100 epochs was able to get both models to approximately 90% accuracy, which was good enough.

Figure 3: Hyperparameter selection justification

of which were greatly outperformed by running on Colab with a GPU. For example, an epoch that ran in roughly three minutes on my desktop ran in roughly 30 seconds on Colab.

- I did not implement cross-validation on a holdout set for hyperparameter tuning. Thus all of the hyperparameters were manually set as I tried to improve the model. This was due to time and hardware constraints, namely:
 - When training locally, the training time was very slow (a few hours).
 - When running on Google Colab, there is a timeout period, which means that I have to be constantly checking on the notebook (or have a script periodically ping the page). This was somewhat unreliable and required a lot of manual attention for long-running training sessions.
 - Tuning with `kerastuner.Hyperband` (as I did for the previous project) would require many more times the training time than a single train. Because of the short time span of this assignment and the little time that I had to work on it due to other classes, I was more focused on making larger improvements to the network (in order to meet the assignment goal on the test dataset) rather than making fine adjustments that would take a very long time to figure out by validation.

3 Results

Both the CIFAR-10 and CIFAR-100 were trained over 100 epochs. The classification accuracy on CIFAR-10 was 93.42%, and the top-5 classification accuracy on CIFAR-100 was 88.40%. This

achieves the goal of 80% top-5 classification accuracy on CIFAR-100. According to benchmarks.ai [5], the top state-of-the-art models train at 99% test accuracy, which is far higher than what is achieved here. This might have been truly state-of-the-art around 2013 through 2017, in which the top accuracy was below 98%, but more recent models have achieved between 98% to 99% test accuracy.

This accuracy is comparable to that reported in [2] on CIFAR-10, which reported a 6.43% error (93.57% accuracy) with a 110-layer ResNet. The improved ResNet units (that my model is more closely based on) achieved a 5.46% error (94.54% accuracy) with a 164-layer ResNet architecture.

The time it takes to train the model on Google Colab with GPU enabled is roughly 43 seconds per epoch, so each model takes roughly 4300 seconds, or 71 minutes, to train.

4 Acknowledgments

- Yuval Ofek – Showed me the power of running on Colab GPUs rather than running it locally.
- Mark Kozykowski – Shared insights on some model improvements, e.g., image preprocessing with `keras.preprocessing.image.ImageDataGenerator` and using `tf.keras.callbacks.LearningRateScheduler` to adjust the maximum learning rate.

5 Source code

5.1 Setup

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pickle

train_imgs = np.zeros((0, 3072))
train_lbls = []
```

5.2 Data entry (CIFAR-10)

```
# train datasets; split up into 6 training batches
for i in range(1, 6):
    with open('../data_batch_' + str(i), 'rb') as file:
        file_data = pickle.load(file, encoding='bytes')
        train_imgs = np.vstack((train_imgs, file_data[b'data']))
        train_lbls += file_data[b'labels']

train_lbls = tf.keras.utils.to_categorical(np.array(train_lbls))
train_imgs = train_imgs.reshape(-1, 3, 32, 32)
train_imgs = np.moveaxis(train_imgs, 1, -1)

# standardize data
train_imgs = (train_imgs - np.mean(train_imgs)) / np.std(train_imgs)

# test dataset
```

```

with open('./test_batch', 'rb') as file:
    file_data = pickle.load(file, encoding='bytes')
    test_imgs = file_data[b'data'].reshape(-1, 3, 32, 32)
    test_lbls = tf.keras.utils.to_categorical(np.array(file_data[b'labels']))
    test_imgs = np.moveaxis(test_imgs, 1, -1)
    test_imgs = (test_imgs - np.mean(test_imgs)) / np.std(test_imgs)

```

5.3 Data entry (CIFAR-100)

```

# train datasets
with open('./train', 'rb') as file:
    file_data = pickle.load(file, encoding='bytes')
    train_imgs = np.vstack((train_imgs, file_data[b'data']))
    train_lbls += file_data[b'fine_labels']

train_lbls = tf.keras.utils.to_categorical(np.array(train_lbls))
train_imgs = train_imgs.reshape(-1, 3, 32, 32)
train_imgs = np.moveaxis(train_imgs, 1, -1)

# standardize data
train_imgs = (train_imgs - np.mean(train_imgs)) / np.std(train_imgs)

# test dataset
with open('./test', 'rb') as file:
    file_data = pickle.load(file, encoding='bytes')
    test_imgs = file_data[b'data'].reshape(-1, 3, 32, 32)
    test_lbls = tf.keras.utils.to_categorical(np.array(file_data[b'fine_labels']))
    test_imgs = np.moveaxis(test_imgs, 1, -1)
    test_imgs = (test_imgs - np.mean(test_imgs)) / np.std(test_imgs)

```

5.4 Model

This is the code for the CIFAR-100 model. Two changes are made for the CIFAR-10 case:

- The last dense layer should have a width of 10.
- The model metric should be changed from top-5 accuracy to accuracy.

```

# initial number of filters for "first stage"; will be doubled twice
# as you progress deeper into the network
num_filters = 32
# for now, layers should be a multiple of 3
layers = 12

# input: 32x32x3 (3 = # color channels)
input = tf.keras.Input(shape=(32, 32, 3))

# do an initial convolution layer to increase dimensionality
x = tf.keras.layers.Conv2D(filters=num_filters,
                           kernel_size=7,
                           strides=1,
                           padding='same',

```

```

    kernel_regularizer=tf.keras.regularizers.l2(1e-6),
    kernel_initializer='he_normal')(input)

for i in range(layers):

    # increase number of filters twice as you go deeper in the network
    # 1x1 convolutional layer to change dimensionality
    if i > 0 and i % (layers / 3) == 0:
        num_filters *= 2
        x = tf.keras.layers.Conv2D(filters=num_filters,
                                  kernel_size=1,
                                  padding='same',
                                  kernel_regularizer=tf.keras.regularizers.l2(1e-6),
                                  kernel_initializer='he_normal')(x)

    # first batchnorm, activation, conv2d
    unit = tf.keras.layers.BatchNormalization()(x)
    unit = tf.keras.layers.ReLU()(unit)

    # in first layer of a "block," no skip connection and use 2x2 strides to
    # decrease image dimensions, see ResNet-34 diagram; for other units, add a
    # skip connection
    if i > 0 and i % (layers / 3) == 0:
        unit = tf.keras.layers.Conv2D(filters=num_filters,
                                      kernel_size=3,
                                      padding='same',
                                      strides=2,
                                      kernel_regularizer=tf.keras.regularizers.l2(1e-6)
                                      ,
                                      kernel_initializer='he_normal')(unit)
        x = unit
    else:
        unit = tf.keras.layers.Conv2D(filters=num_filters,
                                      kernel_size=3,
                                      padding='same',
                                      kernel_regularizer=tf.keras.regularizers.l2(1e-6)
                                      ,
                                      kernel_initializer='he_normal')(unit)
        x = tf.keras.layers.Add()([x, unit])

    # second batchnorm, activation, conv2d
    unit = tf.keras.layers.BatchNormalization()(x)
    unit = tf.keras.layers.ReLU()(unit)
    unit = tf.keras.layers.Conv2D(filters=num_filters,
                                kernel_size=3,
                                padding='same',
                                kernel_initializer='he_normal',
                                kernel_regularizer=tf.keras.regularizers.l2(1e-6))(unit)
    unit = tf.keras.layers.Dropout(rate=0.1)(unit)
    x = tf.keras.layers.Add()([x, unit])

# final part: batchnorm, pooling, dense layer (logits for softmax)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.AveragePooling2D(pool_size=8)(x)
x = tf.keras.layers.Flatten()(x)

# for CIFAR-100, units=100; for CIFAR-10, units=10

```

```

x = tf.keras.layers.Dense(units=100,
                          kernel_initializer='he_normal',
                          kernel_regularizer=tf.keras.regularizers.L1L2(l2=1e-6))(x)

model = tf.keras.models.Model(inputs=[input], outputs=x)

# set up model loss and optimizer
# for CIFAR-100, tf.keras.metrics.TopKCategoricalAccuracy(k=5);
# for CIFAR-10, use 'accuracy'
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.TopKCategoricalAccuracy(k=5)])

```

5.5 Image preprocessing/augmentation and training

```

# feature preprocessing
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    fill_mode='nearest',
    horizontal_flip=True,
    vertical_flip=True,
)

# training
datagen.fit(train_imgs)

def learning_rate_scheduler(epoch):
    return 1e-3 * 0.99**epoch

model.fit_generator(datagen.flow(train_imgs, train_lbls),
                    callbacks=[tf.keras.callbacks.LearningRateScheduler(
                        learning_rate_scheduler)],
                    epochs=100, verbose=1)

```

5.6 Model evaluation

```

print('Evaluating on test dataset')
model.evaluate(test_imgs, test_lbls)

```

6 Code output

6.1 CIFAR-10

```

Epoch 1/100
1563/1563 [=====] - 44s 28ms/step - loss: 1.7498 - accuracy: 0.3513
Epoch 2/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4927 - accuracy: 0.4610
Epoch 3/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3685 - accuracy: 0.5118
Epoch 4/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2806 - accuracy: 0.5473
Epoch 5/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2195 - accuracy: 0.5708
Epoch 6/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1637 - accuracy: 0.5912
Epoch 7/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1174 - accuracy: 0.6063
Epoch 8/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.0714 - accuracy: 0.6274
Epoch 9/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.0310 - accuracy: 0.6421
Epoch 10/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.9921 - accuracy: 0.6576
Epoch 11/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.9654 - accuracy: 0.6647
Epoch 12/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.9349 - accuracy: 0.6783
Epoch 13/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.9045 - accuracy: 0.6888
Epoch 14/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.8809 - accuracy: 0.6986
Epoch 15/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.8495 - accuracy: 0.7096
Epoch 16/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.8235 - accuracy: 0.7209
Epoch 17/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.8076 - accuracy: 0.7244
Epoch 18/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7935 - accuracy: 0.7325
Epoch 19/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7762 - accuracy: 0.7381
Epoch 20/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7591 - accuracy: 0.7458
Epoch 21/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7415 - accuracy: 0.7502
Epoch 22/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7253 - accuracy: 0.7580
Epoch 23/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7176 - accuracy: 0.7641
Epoch 24/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.7008 - accuracy: 0.7674
Epoch 25/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6915 - accuracy: 0.7706
Epoch 26/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6771 - accuracy: 0.7773
Epoch 27/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6628 - accuracy: 0.7834
Epoch 28/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6594 - accuracy: 0.7832
Epoch 29/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.6415 - accuracy: 0.7920
Epoch 30/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6367 - accuracy: 0.7928
Epoch 31/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6256 - accuracy: 0.7966
Epoch 32/100
1563/1563 [=====] - 44s 28ms/step - loss: 0.6177 - accuracy: 0.7991
Epoch 33/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.6069 - accuracy: 0.8015
Epoch 34/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.6019 - accuracy: 0.8047
Epoch 35/100
1563/1563 [=====] - 43s 28ms/step - loss: 0.5895 - accuracy: 0.8105
Epoch 36/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5819 - accuracy: 0.8123
Epoch 37/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5758 - accuracy: 0.8130
Epoch 38/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5690 - accuracy: 0.8167
Epoch 39/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5584 - accuracy: 0.8218
Epoch 40/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5511 - accuracy: 0.8243
Epoch 41/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5488 - accuracy: 0.8244
Epoch 42/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5448 - accuracy: 0.8250
Epoch 43/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5385 - accuracy: 0.8288
Epoch 44/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5316 - accuracy: 0.8294
Epoch 45/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5247 - accuracy: 0.8320
Epoch 46/100

```

```

1563/1563 [=====] - 43s 27ms/step - loss: 0.5177 - accuracy: 0.8363
Epoch 47/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5098 - accuracy: 0.8400
Epoch 48/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5073 - accuracy: 0.8389
Epoch 49/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.5002 - accuracy: 0.8432
Epoch 50/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4958 - accuracy: 0.8437
Epoch 51/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4911 - accuracy: 0.8472
Epoch 52/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4885 - accuracy: 0.8469
Epoch 53/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4814 - accuracy: 0.8491
Epoch 54/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4754 - accuracy: 0.8515
Epoch 55/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4726 - accuracy: 0.8533
Epoch 56/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4667 - accuracy: 0.8543
Epoch 57/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4622 - accuracy: 0.8585
Epoch 58/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4579 - accuracy: 0.8576
Epoch 59/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4582 - accuracy: 0.8573
Epoch 60/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4521 - accuracy: 0.8603
Epoch 61/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.4472 - accuracy: 0.8610
Epoch 62/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4441 - accuracy: 0.8631
Epoch 63/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4410 - accuracy: 0.8638
Epoch 64/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4389 - accuracy: 0.8651
Epoch 65/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.4364 - accuracy: 0.8673
Epoch 66/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4315 - accuracy: 0.8671
Epoch 67/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4293 - accuracy: 0.8677
Epoch 68/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4242 - accuracy: 0.8700
Epoch 69/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.4187 - accuracy: 0.8733
Epoch 70/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4166 - accuracy: 0.8725
Epoch 71/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4103 - accuracy: 0.8754
Epoch 72/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.4078 - accuracy: 0.8766
Epoch 73/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.4074 - accuracy: 0.8768
Epoch 74/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.4024 - accuracy: 0.8783
Epoch 75/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3995 - accuracy: 0.8801
Epoch 76/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3937 - accuracy: 0.8827
Epoch 77/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3942 - accuracy: 0.8805
Epoch 78/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3892 - accuracy: 0.8846
Epoch 79/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3878 - accuracy: 0.8825
Epoch 80/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3851 - accuracy: 0.8838
Epoch 81/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3828 - accuracy: 0.8841
Epoch 82/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3798 - accuracy: 0.8863
Epoch 83/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3781 - accuracy: 0.8871
Epoch 84/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3761 - accuracy: 0.8868
Epoch 85/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3669 - accuracy: 0.8894
Epoch 86/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3693 - accuracy: 0.8895
Epoch 87/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3637 - accuracy: 0.8911
Epoch 88/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3634 - accuracy: 0.8912
Epoch 89/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3607 - accuracy: 0.8918
Epoch 90/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3568 - accuracy: 0.8942
Epoch 91/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3510 - accuracy: 0.8974
Epoch 92/100

```

```

1563/1563 [=====] - 43s 27ms/step - loss: 0.3525 - accuracy: 0.8946
Epoch 93/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3511 - accuracy: 0.8954
Epoch 94/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3482 - accuracy: 0.8975
Epoch 95/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3459 - accuracy: 0.8988
Epoch 96/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3457 - accuracy: 0.8989
Epoch 97/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3423 - accuracy: 0.8996
Epoch 98/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3361 - accuracy: 0.9016
Epoch 99/100
1563/1563 [=====] - 42s 27ms/step - loss: 0.3396 - accuracy: 0.9005
Epoch 100/100
1563/1563 [=====] - 43s 27ms/step - loss: 0.3362 - accuracy: 0.9022
Evaluating on test dataset
313/313 [=====] - 3s 9ms/step - loss: 0.2387 - accuracy: 0.9342
[0.23870298266410828, 0.9341999888420105]

```

6.2 CIFAR-100

```

Epoch 1/100
1563/1563 [=====] - 42s 27ms/step - loss: 4.0681 - top_k_categorical_accuracy: 0.2525
Epoch 2/100
1563/1563 [=====] - 42s 27ms/step - loss: 3.6597 - top_k_categorical_accuracy: 0.3778
Epoch 3/100
1563/1563 [=====] - 44s 28ms/step - loss: 3.3683 - top_k_categorical_accuracy: 0.4632
Epoch 4/100
1563/1563 [=====] - 44s 28ms/step - loss: 3.1648 - top_k_categorical_accuracy: 0.5156
Epoch 5/100
1563/1563 [=====] - 44s 28ms/step - loss: 2.9978 - top_k_categorical_accuracy: 0.5584
Epoch 6/100
1563/1563 [=====] - 44s 28ms/step - loss: 2.8675 - top_k_categorical_accuracy: 0.5908
Epoch 7/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.7648 - top_k_categorical_accuracy: 0.6177
Epoch 8/100
1563/1563 [=====] - 43s 27ms/step - loss: 2.6732 - top_k_categorical_accuracy: 0.6388
Epoch 9/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.5991 - top_k_categorical_accuracy: 0.6559
Epoch 10/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.5330 - top_k_categorical_accuracy: 0.6707
Epoch 11/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.4589 - top_k_categorical_accuracy: 0.6873
Epoch 12/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.3976 - top_k_categorical_accuracy: 0.6997
Epoch 13/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.3379 - top_k_categorical_accuracy: 0.7119
Epoch 14/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.2853 - top_k_categorical_accuracy: 0.7228
Epoch 15/100
1563/1563 [=====] - 43s 27ms/step - loss: 2.2403 - top_k_categorical_accuracy: 0.7344
Epoch 16/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.1943 - top_k_categorical_accuracy: 0.7417
Epoch 17/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.1394 - top_k_categorical_accuracy: 0.7536
Epoch 18/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.0985 - top_k_categorical_accuracy: 0.7650
Epoch 19/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.0604 - top_k_categorical_accuracy: 0.7693
Epoch 20/100
1563/1563 [=====] - 43s 28ms/step - loss: 2.0263 - top_k_categorical_accuracy: 0.7773
Epoch 21/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.9891 - top_k_categorical_accuracy: 0.7846
Epoch 22/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.9609 - top_k_categorical_accuracy: 0.7893
Epoch 23/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.9264 - top_k_categorical_accuracy: 0.7963
Epoch 24/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.8968 - top_k_categorical_accuracy: 0.8026
Epoch 25/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.8769 - top_k_categorical_accuracy: 0.8055
Epoch 26/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.8459 - top_k_categorical_accuracy: 0.8112
Epoch 27/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.8216 - top_k_categorical_accuracy: 0.8134
Epoch 28/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.7960 - top_k_categorical_accuracy: 0.8205
Epoch 29/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.7752 - top_k_categorical_accuracy: 0.8224
Epoch 30/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.7529 - top_k_categorical_accuracy: 0.8265
Epoch 31/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.7329 - top_k_categorical_accuracy: 0.8301
Epoch 32/100

```

```

1563/1563 [=====] - 43s 27ms/step - loss: 1.7136 - top_k_categorical_accuracy: 0.8319
Epoch 33/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.6925 - top_k_categorical_accuracy: 0.8377
Epoch 34/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.6768 - top_k_categorical_accuracy: 0.8409
Epoch 35/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.6568 - top_k_categorical_accuracy: 0.8439
Epoch 36/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.6362 - top_k_categorical_accuracy: 0.8465
Epoch 37/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.6228 - top_k_categorical_accuracy: 0.8509
Epoch 38/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.6114 - top_k_categorical_accuracy: 0.8527
Epoch 39/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.5943 - top_k_categorical_accuracy: 0.8555
Epoch 40/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.5731 - top_k_categorical_accuracy: 0.8586
Epoch 41/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.5627 - top_k_categorical_accuracy: 0.8592
Epoch 42/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.5467 - top_k_categorical_accuracy: 0.8631
Epoch 43/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.5284 - top_k_categorical_accuracy: 0.8651
Epoch 44/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.5174 - top_k_categorical_accuracy: 0.8640
Epoch 45/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.4951 - top_k_categorical_accuracy: 0.8711
Epoch 46/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4918 - top_k_categorical_accuracy: 0.8701
Epoch 47/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4754 - top_k_categorical_accuracy: 0.8723
Epoch 48/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4596 - top_k_categorical_accuracy: 0.8756
Epoch 49/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4506 - top_k_categorical_accuracy: 0.8773
Epoch 50/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4426 - top_k_categorical_accuracy: 0.8788
Epoch 51/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4234 - top_k_categorical_accuracy: 0.8806
Epoch 52/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.4232 - top_k_categorical_accuracy: 0.8805
Epoch 53/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4075 - top_k_categorical_accuracy: 0.8847
Epoch 54/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.4013 - top_k_categorical_accuracy: 0.8854
Epoch 55/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3845 - top_k_categorical_accuracy: 0.8884
Epoch 56/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.3836 - top_k_categorical_accuracy: 0.8878
Epoch 57/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3690 - top_k_categorical_accuracy: 0.8905
Epoch 58/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.3541 - top_k_categorical_accuracy: 0.8906
Epoch 59/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3458 - top_k_categorical_accuracy: 0.8937
Epoch 60/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3425 - top_k_categorical_accuracy: 0.8939
Epoch 61/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3249 - top_k_categorical_accuracy: 0.8953
Epoch 62/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3209 - top_k_categorical_accuracy: 0.8963
Epoch 63/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.3073 - top_k_categorical_accuracy: 0.8980
Epoch 64/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2956 - top_k_categorical_accuracy: 0.9012
Epoch 65/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2907 - top_k_categorical_accuracy: 0.9019
Epoch 66/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2829 - top_k_categorical_accuracy: 0.9024
Epoch 67/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2730 - top_k_categorical_accuracy: 0.9028
Epoch 68/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2616 - top_k_categorical_accuracy: 0.9049
Epoch 69/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2598 - top_k_categorical_accuracy: 0.9054
Epoch 70/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2408 - top_k_categorical_accuracy: 0.9087
Epoch 71/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2406 - top_k_categorical_accuracy: 0.9088
Epoch 72/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2295 - top_k_categorical_accuracy: 0.9095
Epoch 73/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2252 - top_k_categorical_accuracy: 0.9109
Epoch 74/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.2168 - top_k_categorical_accuracy: 0.9125
Epoch 75/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.2135 - top_k_categorical_accuracy: 0.9114
Epoch 76/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1976 - top_k_categorical_accuracy: 0.9138
Epoch 77/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1951 - top_k_categorical_accuracy: 0.9131
Epoch 78/100

```

```

1563/1563 [=====] - 43s 28ms/step - loss: 1.1925 - top_k_categorical_accuracy: 0.9139
Epoch 79/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1755 - top_k_categorical_accuracy: 0.9180
Epoch 80/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1669 - top_k_categorical_accuracy: 0.9181
Epoch 81/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1592 - top_k_categorical_accuracy: 0.9186
Epoch 82/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1631 - top_k_categorical_accuracy: 0.9184
Epoch 83/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1433 - top_k_categorical_accuracy: 0.9211
Epoch 84/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1510 - top_k_categorical_accuracy: 0.9204
Epoch 85/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1361 - top_k_categorical_accuracy: 0.9218
Epoch 86/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1320 - top_k_categorical_accuracy: 0.9218
Epoch 87/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1281 - top_k_categorical_accuracy: 0.9226
Epoch 88/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1182 - top_k_categorical_accuracy: 0.9245
Epoch 89/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1142 - top_k_categorical_accuracy: 0.9263
Epoch 90/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.1017 - top_k_categorical_accuracy: 0.9282
Epoch 91/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.1026 - top_k_categorical_accuracy: 0.9276
Epoch 92/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.0994 - top_k_categorical_accuracy: 0.9276
Epoch 93/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0958 - top_k_categorical_accuracy: 0.9259
Epoch 94/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0832 - top_k_categorical_accuracy: 0.9289
Epoch 95/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0741 - top_k_categorical_accuracy: 0.9303
Epoch 96/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0705 - top_k_categorical_accuracy: 0.9303
Epoch 97/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0670 - top_k_categorical_accuracy: 0.9307
Epoch 98/100
1563/1563 [=====] - 43s 28ms/step - loss: 1.0651 - top_k_categorical_accuracy: 0.9313
Epoch 99/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0631 - top_k_categorical_accuracy: 0.9323
Epoch 100/100
1563/1563 [=====] - 43s 27ms/step - loss: 1.0502 - top_k_categorical_accuracy: 0.9328
Evaluating on test dataset
313/313 [=====] - 3s 8ms/step - loss: 1.5470 - top_k_categorical_accuracy: 0.8840
[1.5469876527786255, 0.8840000033378601]

```

References

- [1] <https://www.cs.toronto.edu/~kriz/cifar.html>
- [2] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
- [3] He, Kaiming, et al. "Identity mappings in deep residual networks." *European conference on computer vision*. Springer, Cham, 2016.
- [4] Recombination of Artificial Neural Networks - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/ResNet-neural-network-architecture-ResNet-34-pictured-image-from-11_fig6_330400293 [accessed 1 Oct, 2020]
- [5] <https://benchmarks.ai/cifar-10>

ECE472 – Quiz 10

Jonathan Lam

December 2, 2020

1. *What do you think is preventing main stream usage of these types of architectures?* (arXiv:1410.5401, arXiv:1808.00508, *Nature*, 538(7626):471–476)

I believe the reason for this is a fairly straightforward one: there is no practical domain for NTMs (DNCs) or NALUs at the current moment where they perform better than any other method (deep architecture or otherwise). These models' relevance is mostly one of theoretical interest; in particular, the NTM's ability to turn discrete memory operations into differentiable operations (by some fuzzy attention mechanism), the DNC's more advanced memory operations, and the generalizing ability for NTMs (by hardcoding in some common numerical functions).

Then, what could be some practical use-cases for these systems? The authors of the NTM and DNC models use their models to evaluate common simple problems, such as copying memory, sorting, and using graph structures. Many of the mainstream models we look at achieve SOTA performance in some domain, as we've seen with ResNets (training deep networks), transformers (speech/text models like GPT-3), and MobileNets (small computational and size resources). In the case of the NTM, this would either mean SOTA performance in existing algorithms, or some efficient algorithm for previously unsolved problems.

The problem I see with this is by attempting to recreate a more general problem-solving machine, we lose the ability to perform very well in any given task. (The opposite is the case for NALUs, which I discuss below.) When many of these simple algorithms require speed, accuracy, and have memory or hardware constraints, a neural network is hardly the best tool for the job. If we have errors in even simple tasks like copying and repeating a piece of data a given number of times, when the corresponding algorithm is trivially simple and optimally fast, then it is not suitable for a task. One potential solution would be to encode some information about the problem in the network, but this clearly does not generalize to other tasks. Another potential “solution” is to use NTMs on higher-level problems, e.g., distributed computing or a logic constraint checker; however, given the accuracy of the NTM on low-level problems, we can hardly expect anything out of accuracy, speed (computational performance), or memory efficiency. At which point the best we might be able to do is

equip the NTM “controller” with higher-level differentiable operations in addition to read/write heads: differentiable data structures, differentiable API calls, etc. This would require a very tiresome method to produce what would probably be at best heuristic results, by coding some human knowledge into the program (efficient algorithms); but this is the price of attempting to make a more generalized problem-solving machine.

(Of course, this is assuming that the end goal is to create some sort of generalized problem-solving machine. There may be some interesting and practical use-case for differentiable memory operations that I can’t fathom.)

On the other hand, NALUs are the opposite – they encode human knowledge of a problem into a structure dedicated to the problem at hand. While this allows for generalization in the domain of arithmetic operations, which is the authors’ stated goal, these NALUs are likely unsuitable for any other task. Moreover, there is no task that the NALU can perform that a regular ALU cannot perform much more efficiently. Thus it is difficult to find a suitable use case for NALUs (it is unfortunate that the authors also do not provide any sort of motivation for NALUs other than generalization for learning arithmetic operations).

tl;dr: These can’t approach SOTA (e.g., hardware and handcoded low-level algorithms) in their domains, even if they can beat other deep neural architectures, and it doesn’t seem that any other practical applications of these architectures have been discovered yet.

ECE472 – Quiz 11

Jonathan Lam

December 8, 2020

NeurIPS 2020: “Towards Neural Programming Interfaces”

I glanced over three or four other randomly-chosen papers before settling on this one. This one definitely caught my eye because of its relationship to APIs, and it far surpassed my expectation in its attempt to truly create a neural-network based interface (“neural programming interface,” or NPI). Not only that, but it’s a wonderful agglomeration of some of the later topical methods we covered in class, e.g., text-generation models, adversarial models, and style-transfer (the latter two are used in a physically similar but functionally very different way). It was also interesting that this paper sprung up concurrently with and independently of a different method to control the output of neural networks (PPLM), and the authors describe how their NPI performs favorably and doesn’t require pre-labeled training data.

The general idea of this paper is that, given some pretrained neural network (e.g., some large generative model based on a transformer architecture), we can train an auxiliary, separate neural network to bias the outputs of the pre-trained one. It’s called an “NPI” due to its similarity to an API: this NPI can pass high-level commands to the mostly-black-box-of-the-larger-neural-network to retrieve an output that was modified by those commands. Example commands for a text-generative network like GPT-2 are to avoid using a certain word/phrase (e.g., offensive language filtering) or topic control. A very general use of this is to undo bias: if bias is found in the network’s generated text, then we can apply a command that reverses that bias.

At a very high-level, the NPI is trained to generate a certain “command” by recording some of the activations of the main network when inputs are fed to the main network. The corresponding outputs are analyzed with respect to their similarity to the desired command (by a loss related to the style-transfer loss), and labeled accordingly. This auto-labeling allows us to train for the best set of activation values to produce the desired command. The NPI is used by adding those recorded activation values onto the equivalent activations during inference time to bias the main network to produce similar outputs. Lastly, the NPI is trained adversarially against a discriminator and a content-classifier.

This is comparable to fine-tuning on a specific domain, but the authors note that NPIs overcome two shortcomings of fine-tuning: “catastrophic forgetting” of the language-model and more fine-tuned “content control” (e.g., filtering at

the granularity of words and phrases rather than subject matter).

A question I might ask someone else about this paper is: *What does this model relate to the trend of model biggening (e.g., GPT-2 vs. GPT-3)? What are some implications?*

I ask this because the intent of the paper, as well as the methodologies involved, deal with specific types of architectures, e.g., the super-large architectures such as GPT-{1,2,3}, MEENA, GeDi, and similar text- or speech-generating neural networks, all of which are very large networks. In particular, the authors focus most of their tests on GPT-2, and attempt to manipulate it to produce goal-oriented tasks, and their results show a fair degree of success.

This begs the question: why not just train a larger network? With few changes to GPT-2 other than increasing the number of parameters, GPT-3 has shown wonders in zero-, one-, or few-shot learning (i.e., no fine-tuning) in different tasks. The authors address this briefly by saying that GPT-3 is a predictive model rather than a goal-oriented text generative model. I'm not entirely convinced that this is a strong distinction.

I believe that this discourse between larger and smaller networks, and the trade-offs between fine-tuning, few-shot learning, and NPIs is interesting. On a (relatively) smaller generative network, we can fine-tune to specific tasks to improve domain-specific knowledge. On a larger network, we can miss out on a lot of the domain-specific training and still get high accuracy in the same tasks. But NPIs encourage some flexibility on both ends of the spectrum: we can use this, rather than fine-tuning, on a smaller network to achieve comparable performance on domain-specific tasks (without the same pitfalls of fine-tuning). On a larger network, perhaps this can achieve better granularity in domain-specific tasks than few-shot learning.

In both cases, the analogy to an API is very apt: we can use these NPI “controllers” to give us the output we need by providing specific commands. Just like an API, this may be a useful step in making large models more available to researchers in their particular applications (with specific commands). If this readily outperforms fine-tuning, perhaps this will encourage more medium-sized models that can be programmed to perform a variety of tasks more effectively than before, which may be useful to researchers who do not have the ability to train the larger models.

ECE472 – Quiz 1

Jonathan Lam

September 9, 2020

1. *Compare and contrast symbolic differentiation, numeric differentiation, and automatic differentiation.*

Symbolic differentiation This provides a (usually closed-form) expression for the derivative of an expression w.r.t. (some or all of) its input variables. This has the benefits of being mathematically exact and easy to interpret; having the closed form expression for a derivative may be useful, e.g., for finding extrema (zeros of the derivative expression) analytically. However, this suffers from expression swell (since repeated chain rule on complex expressions can quickly make the expression grow very long), which can take a lot of memory and lead to many repeated calculations. This also suffers from manipulating code to generate the derivative expression, which may be unwieldy.

Numeric differentiation This approximates the derivative using a difference quotient. While this is faster than symbolic differentiation for complex expressions, it is very prone to error. The error not only grows with the length of the calculation, but round-off and truncation errors occur due to the inherent nature of the derivative being defined as a limit where the denominator approaches zero, and of the limited size of numeric types on computers.

Automatic differentiation This is the preferred way of doing differentiation of an expression with respect to its inputs and outputs, since it provides exact values like symbolic differentiation (within the limits of numeric types) and provides only constant-time overhead for each input variable (like numeric differentiation). This works by performing the same calculations as symbolic differentiation (and therefore producing an identical result), but not storing or calculating the entire derivative expression. Rather, it only stores the values of subexpressions and their derivatives (at a small memory overhead), which can be used by the chain rule to calculate the derivatives of larger subexpressions (in forward accumulation mode; the chain rule is applied in a slightly different way in reverse accumulation mode).

2. Compare and contrast forward-mode automatic differentiation with reverse-mode automatic differentiation.

Any particular expression s has a series of constants and variables (the most primitive subexpressions) and a hierarchical tree of subexpressions. Except for the entire expression s , each subexpression s_1 is combined with another subexpression s_2 to form another subexpression s_3 (in the case of a binary expression, but it is not hard to see how this would work with operations with different arities). To calculate the value of the subexpression s_3 , we have to know (i.e., calculate and store) the values of s_1 and s_2 before hand; so we always evaluate expressions from the inside-out, calculating more primitive subexpressions before being able to calculate the value of the expressions that depend on them.

Forward-mode autodiff works the same way. Assume there is a variable x , and we want to find $\frac{\partial s_3}{\partial x}$. Instead of only requiring the *values* of s_1 and s_2 before calculating this partial derivative, we also have to know $\frac{\partial s_1}{\partial x}$, $\frac{\partial s_2}{\partial x}$, $\frac{\partial s_3}{\partial s_1}$, and $\frac{\partial s_3}{\partial s_2}$, since the chain rule states:

$$\frac{\partial s_3}{\partial x} = \frac{\partial s_3}{\partial s_1} \frac{\partial s_1}{\partial x} + \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial x}$$

The latter two depends on the operation being performed on s_1 and s_2 to form s_3 , and the former two have already been calculated by performing this kind of calculation recursively from the lower expressions upwards.

This is straightforward because it is a direct application of the chain rule, but it involves calculating and recording the partial derivatives of every intermediate value w.r.t. each input variable. Reverse-mode differentiation is different in that we can aggregate the values in the opposite direction: the same chain rule equation above can be accumulated in reverse. For example, imagine that there is a subexpression s_4 , which is used directly in subexpressions s_5 , s_6 , and s_7 . This gives the following equation (also by chain rule), which looks similar to the above equation. (It's still chain rule, but from the output variable's perspective.)

$$\frac{\partial y}{\partial s_4} = \frac{\partial y}{\partial s_5} \frac{\partial s_5}{\partial s_4} + \frac{\partial y}{\partial s_6} \frac{\partial s_6}{\partial s_4} + \frac{\partial y}{\partial s_7} \frac{\partial s_7}{\partial s_4}$$

Algorithmically, we start from calculating the partial derivatives of the subexpressions that y is directly dependent on through this relation. Then, we can move “backwards”: in this case, we calculate $\frac{\partial y}{\partial s_4}$ based on the partial derivatives of its dependencies, i.e., $\frac{\partial y}{\partial s_5}$, $\frac{\partial y}{\partial s_6}$, and $\frac{\partial y}{\partial s_7}$.

Why do this? While forward accumulation builds up partial derivatives for each subexpression w.r.t. each input variable, reverse accumulation builds up partial derivatives for each subexpression w.r.t. each output variable. Since we usually have many more features than output variables, the former is much more expensive and calculates a lot of values we don't need, since we don't care about the partial derivatives of each of the subexpressions w.r.t. input variables.

ECE472 – Quiz 1

Jonathan Lam

September 16, 2020

1. *Explain the key differences between Adam and the basic gradient descent algorithm.*

The basic stochastic gradient descent algorithm takes the gradient of the loss function based on a batch of samples, and it changes all of the variables in the direction of the negative gradient. The basic idea of the gradient descent doesn't mention much about step size or convergence, which could easily become problematic when the gradient is not smooth or somewhat irregular ("pathological curvature"). The easy thing to do (and this basic approach works for the first two homework assignments) is to assign a small value to the learning rate α and assume that the gradient of the objective (loss) function is smooth enough that there are no problems with the descent. When there are problems with convergence, you can manually make the step size smaller (this was my naive approach to nonconvergence for homework 2, in which I decreased α by a factor of 2 every 1000 epochs).

The main difference for Adam descent is that it stores a weighted gradient average (very similar to momentum) and a weighted gradient-squared average (which can be loosely interpreted as an (uncentered) "variance"). These averages are calculated much like momentum is, with the current weight calculated from the previous weight (multiplied by a multiplicative decay factor) combined with the current gradient (or gradient-squared) – this in itself offers the benefit of momentum, similar to RMSProp. (It's important to note that this operations is not very expensive, as it is linear in time and space w.r.t. the number of coefficients, rather than the size of the input, and only involves the first-order gradient like regular stochastic gradient descent.) (These weighted averages are scaled by a factor to correct their initialization bias.)

Using these calculated values, Adam descent has a different update rule: instead of subtracting the gradients (multiplied by some learning factor) from the weights, the update rule is:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where θ_t is the vector of weights at the current iteration, θ_{t-1} is the vector of weights at the previous iteration, α is the (roughly) upper limit on the learning rate, \hat{m} is the vector of initialization-bias-corrected weighted gradient averages, \hat{v} is the vector of initialization-bias-corrected weighted gradient "variances", and ϵ is a small nonzero constant to avoid division by zero.

The reason we have this more complex update rule is that it is "automatically annealing." If we treat \hat{m} as roughly the gradient (with some momentum), and treat \hat{v} as some sort of variance/noise/uncertainty factor, then \hat{m} denotes the (negative of) the direction we should

be moving towards to minimize the objective, and \hat{v} indicates how sure we are of moving in the correct direction. The greater the gradient \hat{m} , the further we should move; the larger the uncertainty \hat{v} , the less we should move. This “intelligently” sets the step size, and it does so for each weight coefficient by considering its gradient and gradient-squared.

Other benefits of Adam is that it converges well and works well on sparse matrices (similar to AdaGrad). The cost is that each iteration is slightly more complex than that of a basic stochastic gradient descent scheme, but this cost is likely greatly outweighed by Adam’s benefits.

2. Why does momentum really work?

We have the same kind of convergence problem that was mentioned in the above answer when choosing the learning rate α . We want the largest learning rate possible without causing issues with convergence.

The article “Why Momentum Really Works?” illustrates the point mathematically by making an eigenvalue decomposition of two problems. These problems are simple enough that we can essentially solve the problem in closed form by considering each parameter separately, and the article shows how this can be used to find the general update rule for the model variables. This turns out to limit the possible range of α by the minimum and maximum eigenvalues of the feature matrix. When the analysis is repeated with the momentum update rule rather than the plain update rule:

$$\begin{aligned} \theta_t &\leftarrow \theta_{t-1} - \alpha \nabla f(\theta_{t-1}) && \text{(Ordinary GD update rule)} \\ \begin{cases} z_t &\leftarrow \beta z_{t-1} + \nabla f(\theta_{t-1}), \\ \theta_t &\leftarrow \theta_{t-1} - \alpha z_t \end{cases} && \text{(Momentum GD update rules)} \end{aligned}$$

where z_t is the weighted average after iteration t , then α can be increased further without divergence, leading to faster divergence. When the learning rate α and the momentum coefficient β have optimal values, then the condition number κ (which indicates convergence rate, lower is better) is roughly the square root of the κ without momentum (which means a much faster convergence).

That approach is purely mathematical, but intuitive understanding is given by the layman’s analogy in physics: a damped spring. α multiplied by the eigenvalues λ_i of the problem matrix can be interpreted as an driving force, and β is analogous to the damping coefficient. As with damped harmonic oscillator problems, there is an optimal β , which is the critically-damped scenario. Without damping (and without manually decreasing α as t increases), it’s likely that there will be large oscillations around complicated gradients such as “ravines” (leading to slow convergence) and may settle in local minimas (leading to a nonoptimal solutions). With momentum, however, we have some information about gradients from past iterations, so we have a less-local view of the overall gradient topology and is less likely to have these problems. These intuitive robustness of momentum translates to the mathematical advantage of being able to increase the learning rate without having convergence issues.

ECE472 – Quiz 3

Jonathan Lam

September 22, 2020

1. *Describe the main method and key takeaways from “probes.” What was the main hang-up that prevented the authors from completing more significant experimentation?* (arXiv:1610.01644)

The authors attempt to make the effects of layers of a neural network more interpretable. Their method estimates how well any particular layer estimates the result; i.e., if a softmax function (in the case of a classification problem) with optimized parameters were applied on the inputs of that layer, and evaluated with the typical benchmarks (e.g., loss or classification error rate). In other words, it is as if we treat this layer as the first layer in the neural network, and only have a single linear classification layer to estimate the output. This involves its own optimization (for each layer) that must be independent of the main training optimization (which can be achieved by calculating gradients separately or manually stopping gradient backpropagation) – hence the name “probe.”

Using this method, the authors discover that the error with these linear classifier probes is strictly decreasing with increasing depth (from the input) in an optimized neural network; in other words, each layer can be thought of as acting as a better linear classifier than the previous; or in other other words, each layer makes the features more linearly-separable. This shows the (perhaps not obvious) effect that no layer in an optimized network can increase the error, which the authors term the “greedy” aspect of neural networks. By probing different layers, one can view how much a particular layer decreases the linear classification error, which can be used in diagnosing or removing unnecessary layers from a neural network.

While the authors seemed to have great success with their methods, they had problems when layers had a very large width. Normally, very wide layers are manageable because all of the data in that layer is part of a function “pipeline” and can immediately be processed by the next layer and immediately discarded. However, in this method we are interpreting (and thus storing data about) the inputs of every layer we want to study. The authors state that when there a layer is millions of inputs wide, this may be on the magnitude of gigabytes (for that layer alone). They state a few ways to mitigate this error (mostly involving reducing dimensionality).

2. *Describe the main method and key takeaways from “confidence penalty.”* (arXiv:1701.06548)

As a motivation for their work, the authors state that a functional view of machine learning is about how good the distribution of output probabilities is, given some input. This can be thought of as how well a model generalizes (hence relating to regularization) but does not care about the internal parameterization of a neural network like the other forms of regularization do. Thus, the authors study the effect of penalizing models based on the (entropy of the)

distribution of their outputs; in other words, they look for models that has a “smoother” output distribution. Similar methods of “output regularization” include smoothing or adding noise to labels, label dropout, and distillation.

This “confidence penalty” is performed by adding an entropy term to the loss function (negative log-likelihood of the softmax (multiclass-classification) function) to penalize models with a lower entropy. (Additionally, an annealed confidence penalty and a hinge loss function help aid convergence rates.) In general, they find that this model outperforms the aforementioned other methods of “output regularization” in almost all of their test cases.

3. *Is there a difference at inference time between batch-norm and batch-renorm?* (arXiv:1702.03275)

No. Batch-renorm attempts to rectify (minimize) the difference between the normalization between training and inference that occurs in batch-norm (i.e., training uses minibatch mean and standard deviation, while inference uses population/running average mean and standard deviation). In particular, this is done by altering the *training* stage by applying an affine transformation to the normalization step to make it more closely approximate the average mean and standard deviations (as used by the inference), and thus batch-renorm does not change how inference is carried out.

ECE472 – Quiz 4

Jonathan Lam

October 1, 2020

1. *Explain how the main technique from “Delving Deep into Rectifiers” may alleviate a core issue in the training scheme for “Going Deeper with Convolutions.” (Think about what they did to “get-it-to-work”) (arXiv:1502.01852, arXiv:1409.4842)*

The authors of the Inception network (“Going Deeper with Convolutions”) cite that “Given the relatively large depth of the network, the ability to propagate gradients back through all the layers in an effective manner was a concern.” This is not only a problem with the Inception, but a general concern with deeper networks in general. To combat this, the authors of the Inception network “add[ed] auxiliary classifiers connected to these intermediate layers, we would expect to encourage discrimination in the lower stages in the classifier, increase the gradient signal that gets propagated back, and provide additional regularization.” (Another way to mitigate this is by using skip connections like a ResNet (as discussed in the next question), as this allows for less-obstructed passthrough of gradients between layers.)

While their reasoning for this is sound, it is arguably a workaround and likely does not generalize to all neural network architectures. “Delving Deep into Rectifiers” attacked a deeper problem in the networks: that the rectifier activation functions used throughout the Inception network may not be optimally passing gradients through. The authors of this paper discovered that the initialization of filter weights when using rectifiers is important to how well the rectifier passes through the signal (i.e., how much a rectifier affects the standard deviation of the weights). By choosing the standard deviations of the randomly-initialized weights (rather than choosing a constant standard deviation), the authors are able to have a better translation of gradients through the network, which helps with training deeper networks (such as Inception).

2. Explain the core effect of pre-activation from “Identity Mappings in Deep Residual Networks” compared to the original residual formulation. (arXiv:1603.05027)

The original ResNet formulation (from “Deep Residual Learning for Image Recognition” (arXiv:1512.03385)) used the idea that perhaps networks could learn better if they only had to learn the “residual” (which is somewhat like the error from the output) rather than try to learn the entire output. How this was implemented was by allowing “skip connections” between layers, which (roughly) passed through the gradient of the previous layer as well as that of the current layer – with this modification over a regular neural network, the difference caused by the current layer (the residual) can be learned. This helps train deeper networks faster, and also helps with learning identity layers (which may be useful in very deep networks).

The paper “Identity Mappings in Deep Residual Networks” examines different variations of the original ResNet formulation. The most significant change they find is that if, rather than using an activation function after each ResNet block (i.e., after combining the output of the current layer and the previous layer), they created an asymmetric form (reordering) of the components of the ResNet such that the activation function acts as the first function in the ResNet block (see the following figure, taken from this paper).

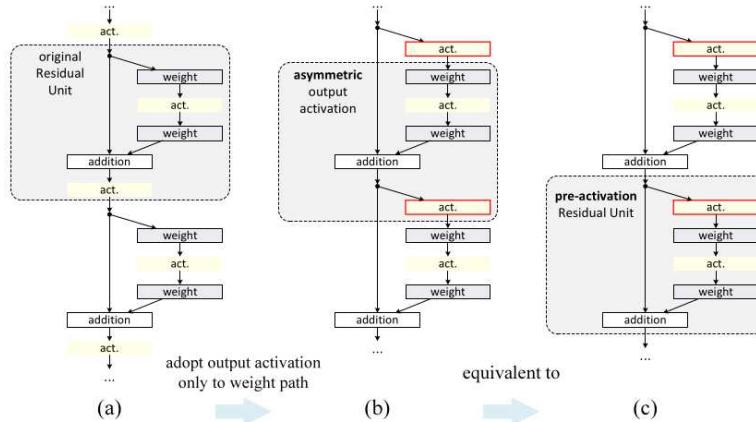


Figure 5. Using asymmetric after-addition activation is equivalent to constructing a pre-activation Residual Unit.

By doing this, they create “paths” for each (modified) ResNet block that “branch off” and “rejoin” the “main path.” Most of this is very similar to the original, except that now no activation functions were on the “main path.” This means that the main path is (very simply) the sum of the original signal and all of the residuals (and the gradient is also a simple sum of the residual gradients). The authors reasoned that this allows gradients of earlier layers to much more easily flow between layers (as opposed to being “obstructed” by the activation functions on the main path), which allows for faster optimization (and also reduced overfitting according to the author’s tests).

ECE472 – Quiz 5

Jonathan Lam

October 7, 2020

1. *Describe one of the experiments in “Rethinking Generalization” and what the implications were.* (arXiv:1611.03530)

The first experiment the authors describe is using standard neural network architectures to fit random labels. What they find is somewhat surprising: these networks, which work so well to classify images, can also classify both randomly-generated labels or randomly-generated images (in both cases, the labels are independent of the features) with high accuracy almost as if they were classifying the original image data (i.e., convergence is only slower by a linear factor). This is not surprising given the promising capability these networks have shown recently, but it is a problem when it comes to thinking about generalization: if it is capable enough to pick up patterns so well out of nonsensical data, but the same network also works well on true data, how do we truly know how well our model generalizes? In one case, the model completely overfits (because there is no relationship when the features or labels are randomized), but in another case, the same exact model works well.

In other words, it is hard to tell when a model overfits and when it does not, because the same model is capable of both. The authors go on to describe that while the common regularization techniques used (e.g., L_2 , L_1 , dropout, batch-norm) often aid generalization, this likely makes up a small part of how well a model generalizes, and there is likely a lot still unknown about the exact nature of generalization.

2. *Compare and contrast SqueezeNet with MobileNets.* (arXiv:1602.07360, arXiv:1704.04861)

The motivation for both is roughly the same: smaller neural networks (in terms of memory) would likely be less computationally expensive to train and use, be easier to distill to other systems, and be smaller in size to transmit (e.g., in the case of software updates). SqueezeNet focused more on the memory footprint, whereas MobileNets (which came a little later) was worried both about small network memory size and about minimizing the computational cost of the neural network during inference time. Both are also focused around the typical convolutional neural network architecture (e.g., AlexNet).

SqueezeNet took into account a few basic principles to try to reduce model parameters, namely by using smaller convolutional filters (mostly 1×1 filters and some 3×3 filters) and a small number of channels (i.e., size of the output of a convolutional layer). They messed around with the percentage of 3×3 filters and additional compression to achieve AlexNet-like accuracy with 50x fewer parameters. They were not, however, aiming to reduce computational cost like the MobileNet authors were, as this is usually a less strict constraint (memory is strictly limited by memory size, but computational cost is measured in time and is more flexible).

MobileNet, on the other hand, changed the nature of the convolutional layers themselves. Usually, the number of parameters for a convolutional layers is the product of the size of the filter, the number of input channels, and the number of output parameters, which can quickly add up. For a given convolutional layer with M input channels and N output channels and a filter with size D , this means MND parameters. A typical convolutional layer also involves convolving M different filters (one for each input channel) for each of the N output channels and then taking a linear combination, which can be very expensive. MobileNets reduces the computations necessary by “factoring” the convolutional layer into two steps: generate only one set of M different filters (one for each of the input channels), and then use a regular 1×1 convolutional layer to take the linear combination of the filters. This has two effects in reducing computation (decreasing the number of filters by N and decreasing filter size by a factor of D), and the authors only empirically experienced a trivial decrease in accuracy. The authors also provided two hyperparameters as a mechanism for a user to customize this network to meet more specific resource constraints. This caused MobileNet to outperform (in accuracy and in speed) SqueezeNet when it had a similar number of parameters.

3. *What do you think of DenseNets?* (arXiv:1608.06993)

I’m pretty surprised at their results – they exceed my expectations of what feels to me like a ResNet implementation with more skip connections. They more or less took the idea of ResNets and put it on steroids, along with some measures to make sure that the number of parameters doesn’t explode (i.e., adding bottlenecking layers and constraining the dense connections to three smaller blocks to prevent too many skip connections), and it seems to outperform pretty much all of the other methods without needing to be too deep or have too many parameters (i.e., a 100-layer DenseNet outperforms and has fewer parameters than a 1000-layer ResNet).

In one sense, this can be thought of as a continuation of “Identity Mappings in Deep Neural Networks” (arXiv:1603.05027), which explored the role of identity activations so that the gradient wouldn’t be obfuscated as it traveled to lower layers (i.e., this architecture had the nice property that the overall gradient would be the sum of the gradients of each ResNet block). Here, they take it a step further by actually passing through earlier feature maps, not obfuscated by the earlier layers at all, which reinforces the aforementioned paper’s claim that direct passthrough of information from earlier layers as skip connections work very well. I think it might be very interesting to look into other possible ways to similar self-reinforcing methods.

ECE472 – Quiz 6

Jonathan Lam

October 14, 2020

1. Explain MAML carefully... (think gradients) (arXiv:1703.03400)

Model-Agnostic Meta-Learning (MAML) is aimed towards making a network more sensitive to the tasks for which it is intended for use (e.g., regression, classification). This means that the network will be able to more quickly train with fewer examples (e.g., in the case of one- or few-shot training). As opposed to prior attempts at this, which usually involve adding more parameters or using a specialized architecture aimed at optimizing learning patterns (“meta-learning”), the authors of this paper surmise that a network can be pre-trained so that the initialized weights throughout the network generally tend to be more sensitive to the changes in the inputs (i.e., they will have a larger change in gradient for the kinds of tasks the network is intended for). They state, “the intuition behind this approach is that some internal representations are more transferrable than others.” Therefore, MAML is essentially an initialization method, in that it does not depend on or affect the structure of the network (and can be applied to many different learning tasks and network structures, as the authors demonstrate).

In particular MAML works by having a pre-training session that involves optimizing weights such that *the potential for gradient optimization upon training is maximized*. This is just another optimization (gradient descent) problem that embeds the original optimization (gradient descent) problem, and is all encoded in this formula:

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i}) = \min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta})}\right)$$

The expression on the left indicates the intent: we want to find the weights that would, when trained on a few tasks (producing θ' , the weights after the stochastic gradient descent rule) picked from the tasks training dataset with a representative sample distribution (i.e., $\mathcal{T}_i \sim p(\mathcal{T})$), minimize the loss. In other words, we aim to choose the set of weights that cause the greatest decrease in loss after it an update rule (on a representative distribution of tasks), which can be thought of as increasing the sensitivity of the loss to updates. Note that this requires some knowledge of the distribution of tasks during pre-training, and involves calculating a second-order gradient (gradient of a gradient) (or at least approximating this with a first-order approximation).

2. *What are the main benefits and weaknesses of Neural ODEs?* (arXiv:1806.07366)

Benefits :

Memory efficiency From an intuitive perspective, there are no more layers like in the more traditional ResNet, but rather a differentiable state function. While we had to store much intermediate state for weights in order to propagate gradients in the more traditional structure (making memory usage linear w.r.t. number of layers), using the adjoint sensitivity model for reverse-mode autodiff is much more memory efficient (roughly constant, according to Table 1 in the report).

ODE solvers can be treated as a black box. Not explicitly mentioned as a benefit, but this allows for existing ODE solvers to be somewhat plug-and-play, because the described implementation doesn't require any knowledge about the internal state of the ODE solver.

Powerful existing ODE solvers The authors cite a long and successful history of ODE solvers. Some advanced ODE solvers use adaptive methods that scale really well to large networks. Next point is related.

Adjustable precision/computing power Some of the ODE solvers allow for customizable levels of accuracy.

“Scalable and invertible normalizing flows” I'm not too sure about what this means, because we did not cover the normalizing flow task, but neural ODEs are supposed to be beneficial for this type of problem.

Arbitrary evaluation points This can be thought of as nicely accommodating for interpolated or extrapolated data points. This is nice, for example, if most of the data comes in discretized measurements (e.g., at specific points in time) but now you want to provide an inference on a data point that came at some in-between value. My (very layman's) understanding of this is that having the ODE solution be a differentiable (and thus smooth) function rather than being discretely parameterized like in a traditional ResNet allows for this smooth interpolation and extrapolation. The extrapolation proficiency of neural ODEs is demonstrated in Figure 8.

Drawbacks :

Non-uniqueness There is not always a unique solution to an ODE, so this is a potential problem. However, the authors note that finite weights (which can be encouraged by regularization) and traditional nonlinearities (e.g., activations) like tanh and ReLU don't pose this problem.

Minibatching is inefficient Minibatching, which is a common theme in SGD, may be more inefficient. My intuition for this is that an ODE only allows a fixed-size input (and a single gradient update) at a time; the authors mention that a mini-batch can be made by concatenating the batch samples' states, but this may require more computation than if doing each sample separately.

3. *Does magnet loss require any extra label information per example compared to softmax cross entropy?* (arXiv:1511.05939)

No.

The authors mention at the beginning that traditional classification methods (i.e., not distance metric learning (DML)) tend to discard all but the label information by the end of the network (and thus these are not well-suited to classification on different classes). DML attempts to make the classification retain more information up until the last step (i.e., the inputs get encoded into some vector space, and then similarity is measured by some function of distance in that vector space), which offers some benefits, e.g., zero-shot learning and easier visualization of similarity.

However, the only supervision given is still the labels (just like in softmax) – the algorithm provided generates a vector representation for DML, but this is all unsupervised. In other words, the labels provide guidance for the results to be clustered into “classes,” but there is an unsupervised subclustering within those classes into smaller “clusters” using a K-means method and a loss function that penalizes cluster overlap. This allows for magnet loss to generate multi-dimensional vectors with some notion of similarity without supervised guidance.

ECE472 – Quiz 7

Jonathan Lam

October 21, 2020

1. *Which component method do you find most compelling from all of these papers? Justify what's interesting to you.*

After reading a few papers on ResNets and its variants, the MSG-GAN proposal was one that I thought was most intuitive and compelling. The original ResNet paper (arXiv:1512.03385) proposed some skip connections, and the authors suggested that this helped the gradients “flow” easier through the network, allowing for faster training and deeper networks. Then we read about variants such as restructuring the network so that the “ResNet blocks” were only connected via identity mappings (arXiv:1603.05027), which further un-obstructed the flow of the gradient through the network. Then we read a paper on DenseNets (arXiv:1608.06993), which create many more skip connections in a groups of ResNet blocks, which further improved the performance of (and decreased the depth necessary for) ResNets.

The overall theme in these (as opposed to “regular” neural networks without skip connections) seems to be that the more skip connections there are, the easier the gradient can flow through the network. Thus, the aim of this paper (arXiv:1903.06048) seemed pretty intuitive to me: add skip connections between the generator and discriminator to promote gradient flow. Like before, the generator and discriminator are all connected as one network, with the generator’s output fed into the discriminator. However, previously the output of the generator was the only input to the discriminator; MSG-GAN proposes that all of the intermediate resolutions used by the generator while generating the final image should be fed to the discriminator. This seems very much like skip connections to me.

The authors of MSG-GAN aim to tackle the instability problem, which they state is due to “insubstantial overlap between the supports of the real and fake distributions,” i.e., if the fake image distribution is too far from the real distribution and the generative model does not get informative gradients back. These connections between the generator and discriminator were aimed towards increasing the gradient flow at different resolutions (without adding extra generators and/or discriminators, and without adding extra hyperparameters like in progressive growing). I found it interesting that this intuition indeed helped in this case as well; the authors state that it improved stability, robustness to learning rate, and led to higher-quality images (using the FID metric).

2. Are there methods that you think are unreasonable?

When reading these, I didn't find any unreasonable per se, i.e., such that I don't believe that their methods are sound. I didn't have the time to spend on understanding the math-theoretical argument proposed in "Smoothness and Stability in GANs" (arXiv:2002.04185), so that might be the most unreasonable in a loose definition of "unreasonable." From the abstract, I understand that the authors are attempting to better understand the inherent instability in GANs. It's very difficult to understand the specifics of their research without any diagrams or simulations, or at least without the math background (e.g., understanding "Jensen-Shannon divergence ... f -divergences ... Wasserstein distance ..., and maximum-mean discrepancy ... Kullback-Leibler divergence"). Without any simple examples, diagrams, or source code, I believe that it is very difficult for anyone without a great knowledge of convergence theory or loss functions to be able to interpret and use these results. Again, this may not be unreasonable because I am not likely the intended audience of this paper (but rather researchers who are hoping to understand GAN stability better), but no matter the reader I think a concrete example would be very helpful.

(On the other hand, the other three papers all were more reasonable to read and understand, and all provided a suitably-intuitive motivation and concrete examples. The StyleGAN paper (arXiv:1912.04958) aimed to fix GAN image artifacts due to the model forcing itself to fit the normalization method, and the spectral normalization paper (arXiv:1802.05957) attempts to improve stability by doing data-independent spectrum-based (eigenvalue-based) normalization on its weights. The MSG-GAN paper, discussed in the first question, made sense in the context of ResNets. I would say that all of these are reasonable.)

ECE472 – Quiz 8

Jonathan Lam

November 11, 2020

1. *Make an argument about super-convergence as it relates to epoch-wise double-descent.*
(arXiv:1708.07120, arXiv:1912.02292)

Double-descent is a phenomenon related to the “effective model complexity” (EMC) of a deep neural network, which can roughly be thought of as the number of samples the model can correctly classify (a measure of “capacity”), as well as the number of training samples. The authors show that EMC is a function both of model complexity (more parameters means a greater network capacity) and number of training epochs (training more epochs intuitively means being able to classify more samples correctly). The phenomenon is that for a low EMC, there is a “U”-shaped curve as the model is under-parameterized, then begins to fit, and then overfits; after a certain point the model begins a “second descent” as the model generalizes again with higher EMC. The increase in test error when overfitting follows the conventional wisdom that the model becomes overly-specialized to the input data, but this logic seems to fail with the second descent. This overfitting occurs when the EMC is close to the number of training samples.

(A possible intuition for this is that as the EMC gets close to the number of input samples, then it almost exactly models the data, and tries to fit it very closely; the model is a perfect fit for the data. As the EMC increases, then the model becomes “over-parameterized”; i.e., many different models can fit the training data, and so the model can generalize better. (E.g., given 1001 points (none of which lie on the same vertical line), you can fit a 1000th-degree polynomial perfectly in exactly one way; with a higher-degree polynomial you can still fit these points but in infinitely many ways, and possibly with a smoother interpolation between the test points.))

Superconvergence is related to a training method that uses cyclical (or “1cycle”) learning rates. This is in contrast to the conventional global or monotonically-decreasing piecewise-constant learning rate. This provides a regularizing effect, and may be well-suited to the specific type of overfitting that occurs with double-descent. Cyclical learning rates (and the “1cycle” method, in which the learning rate is kept high until near the end of training) discourage the premature convergence that happens when the EMC approaches the number of training samples. Other forms of regularization also slow convergence, but they slow the increasing of the EMC as well; superconvergence makes the EMC increase even faster with its large learning rates, potentially increasing the EMC quickly past the double-descent hump. Similarly to regularization, learning rate optimizers like Adam may try to prematurely converge when the EMC approaches the number of training samples by slowing learning, and this will also make it harder to overcome the double-descent hump.

ECE472 – Quiz 9

Jonathan Lam

November 18, 2020

1. *Identify something you find “unsavory” about these kinds of models; explain and argue.*
(arXiv:1810.04805, arXiv:2005.14165, arXiv:2007.14062)

I chose to focus on some of the experiments in the GPT-3 paper, as this seems to be the one generating the most excitement recently. (I also watched this interview with GPT-3 that is fascinating). While the empirical evidence (e.g., that video interview) is very convincing in being able to demonstrate the kinds of questions that GPT-3 is able to provide rational (and sometimes humorous) responses, it's still very hard to see when things are canned and when they are not.

On the other hand, the BERT and Big Bird papers do not have this problem: they focus on domain-specific tasks in NLP (the Big Bird paper also tackles genomics, but it provides a convincing argument how this is similar to NLP tasks), e.g., GLUE and SQuAD.

With the use of a state-of-the-art NLP language model (transformers, the BERT model), I believe GPT-3's ability to do comprehension and language generation tasks are impressive and well-justified. However, when it comes to some other tasks mentioned – in particular the tasks involving some sort of factual lookup (information retrieval) or reasoning (logical reasoning and arithmetic) – I don't think that their claims are very satisfactory (“unsavory”?). Perhaps my bias as an engineer makes me overly skeptical that this LM can actually perform more generalized (non-language-related) reasoning tasks. Fleshying out these tasks, as the GPT-3 paper describes them:

“Closed Book Question Answering” (Information Retrieval) The authors state that they “measure GPT-3’s ability to answer questions about broad factual knowledge” in a “closed-book” manner, i.e., they don’t allow training on external, content-specific resources, and (like the rest of the paper) they don’t perform fine-tuning. GPT-3 has SOTA performance here.

“Common Sense Reasoning” This task “consider[s] three datasets which attempt to capture physical or scientific reasoning, as distinct from sentence completion, reading comprehension, or broad knowledge question answering.” Examples include questions from pre-college science exams, and some questions “which simple statistical or information retrieval systems were unable to correctly answer.” GPT-3 has SOTA performance here.

Arithmetic This includes two- to five-digit addition and subtraction, two-digit multiplication, and one-digit composite (composition of operations). GPT-3 reaches high accuracy on the two- and three-digit addition and subtraction on the largest model (175B parameters), but does not reach high accuracy for the other tasks or in smaller models. With

these results, the authors conclude that “Overall, GPT-3 displays reasonable proficiency at moderately complex arithmetic in few-shot, one-shot, and even zero-shot settings.”

The problem I have with these three in particular is that they all have some claim to logical reasoning (unrelated to the LM), but it seems to me that all of these could be attributed to having examples in the training set. With such a large input set from the Internet, even after some filtering and deduplication I wouldn’t be surprised if many of their results are achieved by finding similar results in the training dataset rather than by doing what we might consider “logical reasoning.” Information retrieval is inherently dependent on the information content of the training dataset; even if it is “closed-source” as in their definition (i.e., not training on specialized data sources), the fact that the information may exist in the dataset makes it seem unremarkable that GPT-3 should perform well as an information retrieval system. On both the common sense reasoning and arithmetic tasks, I imagine that many common questions of a similar form may be encountered on the web (e.g., KhanAcademy and the wealth of educational websites). Even though the authors mention that they attempt to reduce data contamination (and they had a bug that allowed some unwanted contamination) and specifically mention that they attempted to search for the math problems in their training dataset to see if the arithmetic was memorized, I am still not too convinced that the overlap was not large. For “common sense problems,” there are so many variants of the same questions on the Internet that I wouldn’t be surprised if many of the answers to the logical reasoning questions were memorized in some slightly different form that survived the data contamination filtering. In particular for arithmetic, there are multiple forms in which an arithmetic operation may be shown; if it came in the form of a word problem, and GPT-3 correctly parses the word problem and “extracts” the arithmetic, then this goes to show the strong LM in GPT-3 but not necessarily its arithmetic/computational/logical reasoning skills. Also, the fact that it did well on two- and three-digit numbers could be another indication of memorization (this is not a high number of permutations out of the hundreds of billions of tokens in the training dataset); good performance on larger operands would be more convincing.

I should qualify my criticisms: I do not know too much about the nature of the Common Crawl dataset, and perhaps I am assuming that it is larger and more all-encompassing than it actually is. (I cannot imagine what a trillion words can contain. My mind is limited to understanding what a few terabytes of digital data can hold.) Of course, working at such a scale (of parameters, data (and data filtering to prevent contamination), etc.) is unprecedented, and the work at OpenAI is all very fascinating. I just would like to see more research into the claims about any reasoning abilities past the scope of conventional NLP metrics as being more than memorization, because I believe the performance they achieve, due to their training methods and assumptions, might be due largely to memorization of examples in the training set.

Part 0: The model

The following classes define the regression techniques. `BaseLinearRegressionModel` includes some common utility functions for all of the regression techniques, including a `validate()` function and a `subsetSplit()` function. It is subclassed by `BasicLinearRegressionModel`, `RidgeRegressionModel`, and `LassoRegressionModel`, which implement different training techniques (overriding the `fit()` method).

Part 0a: Base class

```
In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from sklearn import linear_model

# base class for multi-input single-output regression
class BaseLinearRegressionModel:

    # features should be a ndarray NxP ndarray, labels a Nx1 ndarray,
    # featureNames should be an array of length P
    def __init__(self, features=None, labels=None, featureNames=None,
                 copySubsetsFrom=None):

        # if copySubsetsFrom is set, use this as a copy constructor,
        # don't redo subset split so that the same train/validate/test
        # subsets are available
        if copySubsetsFrom is not None:
            self._features = copySubsetsFrom._features
            self._labels = copySubsetsFrom._labels
            self._featureNames = copySubsetsFrom._featureNames
            self._subsets = copySubsetsFrom._subsets
            # don't copy validation data nor betas, these should be recalculated
            # with a new model
            return

        if features is None or labels is None or featureNames is None:
            raise Exception('missing parameters')

        self._features = features
        self._labels = labels
        self._featureNames = featureNames

        self._beta = None
        self._validationData = None

        self.subsetSplit()

    # this should be overridden using the regression strategy in subclasses
    def fit(self):
        raise Exception('cannot call fit() on abstract base regression class')

    # this very basic validation function performs a sweep and returns
    # the MSEs calculated for each beta on the validation data; the results are
    # stored in self._validationData so that they can be plotted later
    def validate(self, lbdas, betas):
        # calculate mses for all betas over the training set
        mses = np.array([self.mse(beta, subset='validation') for beta in betas])

        # store results of validation for use in graphing
        self._validationData = lbdas, mses, betas

        # return beta with minimum mse
        return betas[mses == np.min(mses), :].flatten()

    def plotValidation(self):
        lbdas, mses, betas = self._validationData
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(15,15))

        # plot coefficients vs. lambda
        ax1.plot(np.log(lbdas), betas[:, 1:])
        ax1.axvline(np.log(lbdas[mses == np.min(mses)]), ls='--')
        ax1.legend(list(self._featureNames) + ['optimal lambda'],
                   bbox_to_anchor=(1.05, 1, loc='upper left'))
        ax1.set_title('coefficients vs. log(lambda)')
        ax1.set_ylabel('coefficients')
        ax1.set_xlabel('log(lambda)')
        ax1.grid('on')

        # plot mse vs. lambda
        ax2.plot(np.log(lbdas), mses)
        ax2.axvline(np.log(lbdas[mses == np.min(mses)]), ls='--')
        ax2.set_title('mse (on validation subset) vs. log(lambda)')
        ax2.set_ylabel('mse')
        ax2.set_xlabel('log(lambda)')
```

```

    ax2.grid('on')

    return fig

# create a 80/10/10 train/validation/test subset split; this can be called
# after fitting to create a new subset partitioning
def subsetSplit(self):
    # N := total number of samples
    N, _ = self._features.shape

    # invalidate beta and validation data, since these were run on a
    # different subset partitioning
    self._beta = None
    self._validationData = None

    # splits (80/10/10)
    s1, s2 = 0.8, 0.9

    # randomly assign samples to datasets with given splits
    shufIndices = np.arange(N)
    np.random.shuffle(shufIndices)
    self._subsets = {
        'train': {
            'features': self._features[shufIndices[:int(s1 * N)]],
            'labels': self._labels[shufIndices[:int(s1 * N)]],
        },
        'validation': {
            'features': self._features[shufIndices[int(s1 * N):int(s2 * N)]],
            'labels': self._labels[shufIndices[int(s1 * N):int(s2 * N)]],
        },
        'test': {
            'features': self._features[shufIndices[int(s2 * N):]],
            'labels': self._labels[shufIndices[int(s2 * N):]],
        },
    }

    # normalize each feature in feature matrix X; z-score makes this easy
    # this is equivalent to sklearn.preprocessing.StandardScaler (I checked it
    # manually -- the output of this function matches that of StandardScaler)
    @staticmethod
    def _normalizeFeatures(X):
        result = np.zeros(X.shape)
        for i in range(X.shape[1]):
            # prevent error if stdev == 0; normalized is all zeros
            if np.std(X[:, i]) == 0:
                result[:, i] = np.zeros(X.shape[0])
                continue

            # feature = (feature - featureMean) / featureStdev (i.e., z-score)
            result[:, i] = stats.zscore(X[:, i])
        return result

    # helper function to get features and labels for a given subset;
    # returns the normalized augmented feature matrix X (with column of 1's)
    # and labels y for the given subset as a tuple; also returns
    # N, P, where N is number of training samples and P is number of features
    # (number of cols - 1, because of the column of 1's); default subset
    # is training subset
    def _subsetXy(self, subset='train'):
        # X := training feature set
        # Y := training label set
        X = self._subsets[subset]['features']
        y = self._subsets[subset]['labels']

        # N := number of training samples
        # P := number of features
        N, P = X.shape

        # scale down feature set
        X = self._normalizeFeatures(X)

        # augment X into Nx(P+1) matrix by adding a column of 1's at front
        X = np.insert(X, 0, [1] * N, axis=1)
        return X, y, N, P

    # compute and return mean-squared error of beta (on test subset)

```

```

# uses the last computed beta by default, but can use a given beta
# (i.e., when doing validation)
def mse(self, beta, subset='test'):
    if beta is None and self._beta is None:
        return None

    # get test labels and normalized test subset features
    X, y, _, _ = self._subsetXy(subset)
    return np.mean((y - (X @ beta)) ** 2)

# compute baseline mse (just a bias at the average value)
def baselineMse(self):
    # get labels of training subset
    _, y, _, P = self._subsetXy()

    # run mse on test subset against the mean of the training subset labels
    baselineBeta = np.zeros(P + 1)
    baselineBeta[0] = np.mean(y)
    return self.mse(baselineBeta)

# return correlation coefficient matrix of normalized features
# this is performed on all subsets to get a sense of the population
def corrCoef(self):
    return np.corrcoef(self._normalizeFeatures(self._features).T)

# compute and return stderrs and z-scores for beta (on training subset)
def zScores(self):
    beta = self._beta
    if beta is None:
        return None

    X, y, N, P = self._subsetXy()

    # estimate variance (eq. 3.8), calculate stderr and z-scores (eq. 3.12)
    variance = np.sum((y - (X @ beta)) ** 2) / (N - P - 1)
    stdErrs = np.sqrt(variance * np.diagonal(np.linalg.pinv(X.T @ X)))
    zScores = beta / stdErrs
    return stdErrs, zScores

# beta getter
def beta(self):
    return self._beta

```

Part 0b: No regularization model

```

In [ ]: # no regularization, basic linear regression; not to be confused with
# BaseLinearRegressionModel, which is the base class for all of the
# linear regression models here
class BasicLinearRegressionModel(BaseLinearRegressionModel):

    def fit(self):
        # get normalized features matrix and labels for training subset
        X, y, _, _ = self._subsetXy()

        # compute beta (eq. 3.6)
        self._beta = np.linalg.pinv(X.T @ X) @ X.T @ y
        return self._beta

```

Part 0c: Ridge regression model

Regression class with ridge regularization and validation. Validation works by performing a sweep on selected values in $\lambda \in [1, 1000]$.

```
In [ ]: # ridge regularization model; sweeps the parameter for lambda (rather than
# using DoF)
class RidgeLinearRegressionModel(BaseLinearRegressionModel):

    def fit(self):
        # get normalized features matrix and labels for training subset
        X, y, _, P = self._subsetXy()

        # column of 1's in X to avoid penalizing the bias
        X = X[:, 1:]

        # generate lambdas, beta candidates to sweep
        lbdas = np.hstack((np.linspace(0.0001, 0.01, 100),
                           np.linspace(0.01, 1, 100),
                           np.linspace(1, 100, 100),
                           np.linspace(100, 10000, 100)))
        betaCandidates = np.array([
            # eq. 3.44: ridge regression
            np.linalg.pinv(X.T @ X + lbda * np.eye(P)) @ X.T @ y
            for lbda in lbdas
        ])

        # reinsert column of biases, estimate bias with mean of y
        betaCandidates = np.insert(betaCandidates, 0,
                                    [np.mean(y)] * len(lbdas), axis=1)

        # choose best beta by validation
        self._beta = self.validate(lbdas, betaCandidates)
        return self._beta
```

Part 0d: Lasso regression model

Regression class with lasso regularization and validation. Uses the `sklearn.linear_model.Lasso` function on the training inputs, and sweeps $\alpha \in [0.001, 0.9]$. (Here, α is represented with the `lbda` variable to make it more consistent with the ridge regression parameters.)

```
In [ ]: class LassoLinearRegressionModel(BaseLinearRegressionModel):

    def fit(self):
        # get normalized features matrix and labels for training subset
        X, y, _, P = self._subsetXy()

        # remove leading 1's from feature matrix, sklearn.linear_model.Lasso
        # will take care of intercepts
        X = X[:,1:]

        # generate lambdas and betas to sweep over
        lbdas = np.hstack((np.linspace(0.0001, 0.01, 100),
                           np.linspace(0.01, 1, 100)))
        betaCandidates = np.zeros((len(lbdas), P+1))
        for i, lbda in enumerate(lbdas):
            model = linear_model.Lasso(alpha=lbda, fit_intercept=True)
            model.fit(X, y)
            betaCandidates[i,:] = np.hstack((model.intercept_, model.coef_))

        # choose best beta by validation
        self._beta = self.validate(lbdas, betaCandidates)
        return self._beta
```

Part 1: Prostate cancer dataset

Load prostate data file as a Pandas array and do some basic cleaning.

Features:

- lcaweight: log cancer volume
- lweight: log prostate weight
- age: patient age
- lbph: log of benign prostate hyperplasia
- svi: seminal vesicle invasion
- lcp: log of capsular penetration
- gleason: Gleason score
- pgg45: percent of Gleason scores 4 or 5

Target:

- lpsa: log of prostate-specific antigen

```
In [ ]: # read in pandas dataframe from csv, drop first column (indices)
df = pd.read_csv('https://web.stanford.edu/~hastie/ElemStatLearn/datasets/prostate.data', '\t')
df = df.drop(df.columns[0], axis=1)

# ignore training label (67/30 split), we'll do our own 80/10/10 split and
# approximate the results rather than try to match the values in the textbook
df = df.drop('train', axis=1)

# get features and labels from the dataset
features = df.drop('lpsa', axis=1)
labels = df.loc[:, 'lpsa']

print('Prostate cancer dataset preview')
df
```

Prostate cancer dataset preview

Out[]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	lpsa
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	-0.430783
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	-0.162519
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	-0.162519
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	-0.162519
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	0.371564
...
92	2.830268	3.876396	68	-1.386294	1	1.321756	7	60	4.385147
93	3.821004	3.896909	44	-1.386294	1	2.169054	7	40	4.684443
94	2.907447	3.396185	52	-1.386294	1	2.463853	7	10	5.143124
95	2.882564	3.773910	68	1.558145	1	1.558145	7	80	5.477509
96	3.471966	3.974998	68	0.438255	1	2.904165	7	20	5.582932

97 rows × 9 columns

Load data into model, print out feature correlation.

```
In [ ]: # create the basic linear regression model, run fit, get values
basicReg = BasicLinearRegressionModel(features.to_numpy(),
                                      labels.to_numpy(), features.columns)
print(f'baseline mse: {basicReg.baselineMse()}\n')
print('Feature correlation')
pd.DataFrame(data=np.around(basicReg.corrCoef(), 2),
              index=features.columns,
              columns=features.columns)
```

baseline mse: 0.8808019334148127

Feature correlation

Out[]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45
lcavol	1.00	0.28	0.22	0.03	0.54	0.68	0.43	0.43
lweight	0.28	1.00	0.35	0.44	0.16	0.16	0.06	0.11
age	0.22	0.35	1.00	0.35	0.12	0.13	0.27	0.28
lbph	0.03	0.44	0.35	1.00	-0.09	-0.01	0.08	0.08
svi	0.54	0.16	0.12	-0.09	1.00	0.67	0.32	0.46
lcp	0.68	0.16	0.13	-0.01	0.67	1.00	0.51	0.63
gleason	0.43	0.06	0.27	0.08	0.32	0.51	1.00	0.75
pgg45	0.43	0.11	0.28	0.08	0.46	0.63	0.75	1.00

Part 1a: No regularization

```
In [ ]: basicReg.fit()
print(f'mse: {basicReg.mse(basicReg.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((basicReg.beta(),
                                       basicReg.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']], features.columns)))
```

mse: 0.4957672668104931

Out[]:

	beta	stderr	zscore
bias	2.43	0.08	29.83
lcavol	0.62	0.11	5.40
lweight	0.31	0.10	3.00
age	-0.16	0.10	-1.60
lbph	0.13	0.10	1.31
svi	0.31	0.11	2.85
lcp	-0.14	0.14	-1.05
gleason	0.15	0.13	1.13
pgg45	0.00	0.14	0.01

Part 1b: Ridge regularization

Run the linear regression with ridge regularization. This uses the same subset partitioning as the basic linear regression without regularization.

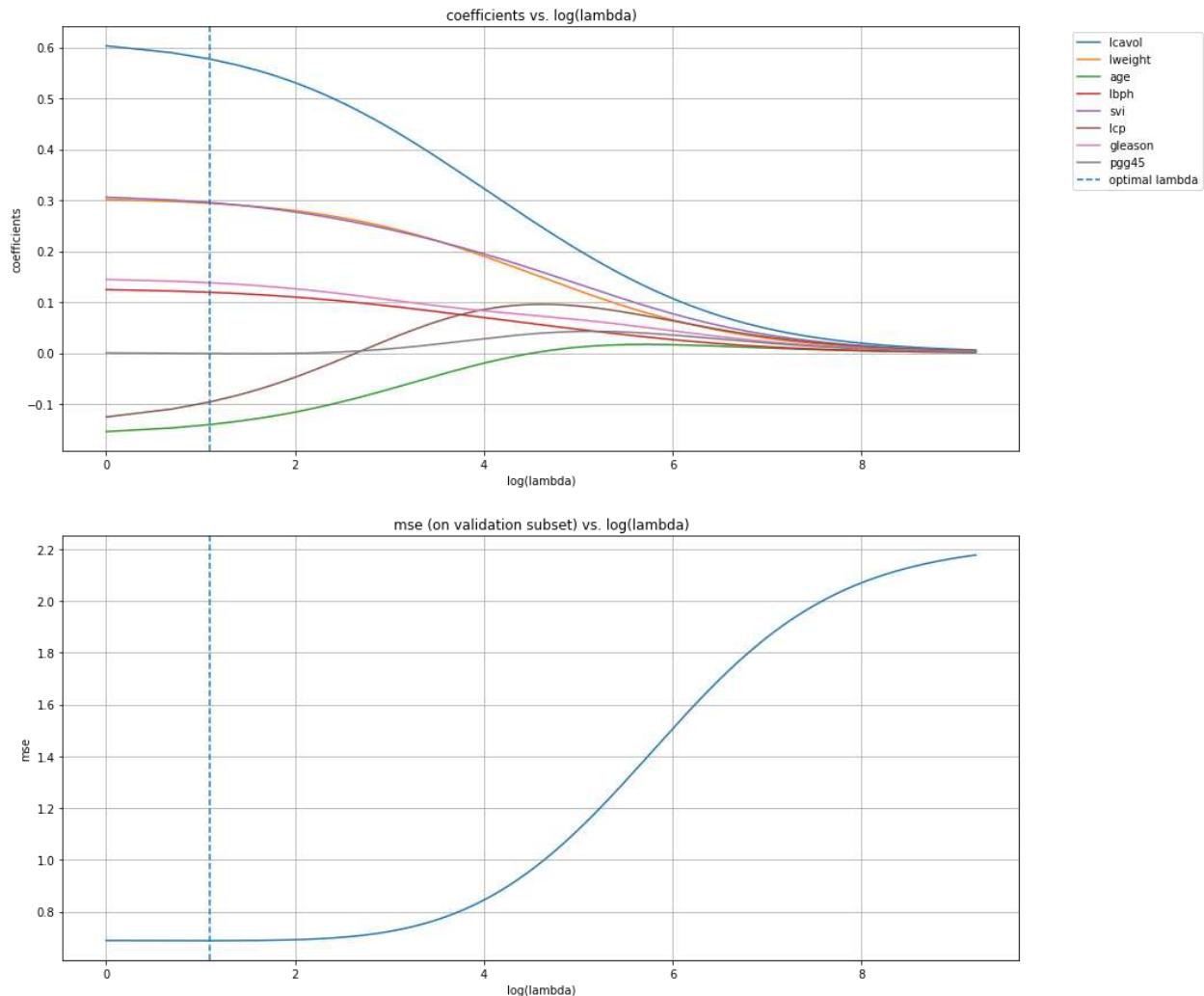
```
In [ ]: # create ridge reg. with same training/validation/test data as first regression
ridgeReg = RidgeLinearRegressionModel(copySubsetsFrom=basicReg)
ridgeReg.fit()
print(f'mse: {ridgeReg.mse(ridgeReg.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((ridgeReg.beta(),
                                         ridgeReg.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']], features.columns)))
```

mse: 0.4820368716805007

Out[]:

	beta	stderr	zscore
bias	2.43	0.08	29.77
lcavol	0.58	0.11	5.04
lweight	0.29	0.10	2.88
age	-0.14	0.10	-1.39
lbph	0.12	0.10	1.22
svi	0.30	0.11	2.70
lcp	-0.10	0.14	-0.70
gleason	0.14	0.13	1.05
pgg45	-0.00	0.14	-0.01

In []: ridgeReg.plotValidation().show()



Part 1c: Lasso regularization

Run the linear regression with lasso regularization. This uses the same subset partitioning as the basic linear regression without regularization.

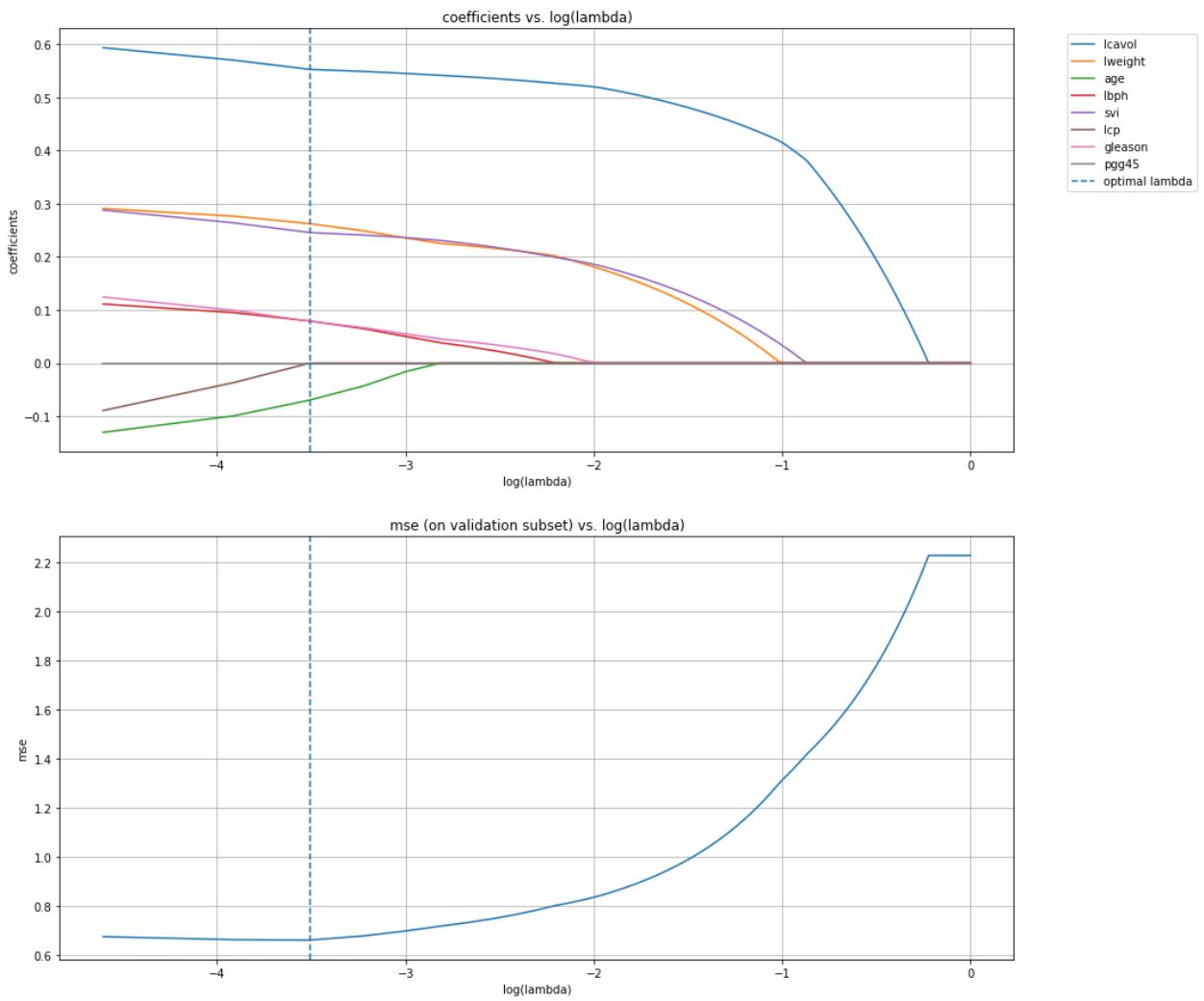
```
In [ ]: # create lasso reg. with same training/validation/test data as first regression
lassoReg = LassoLinearRegressionModel(copySubsetsFrom=basicReg)
lassoReg.fit()
print(f'mse: {lassoReg.mse(lassoReg.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((lassoReg.beta(),
                                         lassoReg.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']],features.columns)))
```

mse: 0.4488124977580511

Out[]:

	beta	stderr	zscore
bias	2.43	0.08	29.35
lcavol	0.55	0.12	4.76
lweight	0.26	0.10	2.54
age	-0.07	0.10	-0.68
lbph	0.08	0.10	0.80
svi	0.25	0.11	2.21
lcu	-0.00	0.14	-0.00
gleason	0.08	0.13	0.59
pgg45	0.00	0.14	0.00

```
In [ ]: lassoReg.plotValidation().show()
```



Part 2: Red Wine dataset

Attribute information:

For more information, read [Cortez et al., 2009].

Input variables (based on physicochemical tests):

- 1 - fixed acidity
- 2 - volatile acidity
- 3 - citric acid
- 4 - residual sugar
- 5 - chlorides
- 6 - free sulfur dioxide
- 7 - total sulfur dioxide
- 8 - density
- 9 - pH
- 10 - sulphates
- 11 - alcohol

Output variable (based on sensory data):

- 12 - quality (score between 0 and 10)

Dataset source: [UCI "Wine Quality" dataset \(<https://archive.ics.uci.edu/ml/datasets/Wine+Quality>\).](https://archive.ics.uci.edu/ml/datasets/Wine+Quality)

The features are all quantitative and there is no missing data, so no data cleaning needs to be performed.

```
In [ ]: df2 = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv', ';')
features2 = df2.drop(['quality'], axis=1)
labels2 = df2.loc[:, 'quality']

print('Red wine dataset preview')
df2
```

Red wine dataset preview

Out[]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

1599 rows × 12 columns

```
In [ ]: basicReg2 = BasicLinearRegressionModel(features2.to_numpy(),
                                             labels2.to_numpy(), features2.columns)
print(f'baseline mse: {basicReg2.baselineMse()}\n')
print('Feature correlation')
pd.DataFrame(data=np.around(basicReg2.corrCoef(), 2),
              index=features2.columns,
              columns=features2.columns)
```

baseline mse: 0.5645694630468364

Feature correlation

Out[]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
fixed acidity	1.00	-0.26	0.67	0.11	0.09	-0.15	-0.11	0.67	-0.68	0.18	-0.06
volatile acidity	-0.26	1.00	-0.55	0.00	0.06	-0.01	0.08	0.02	0.23	-0.26	-0.20
citric acid	0.67	-0.55	1.00	0.14	0.20	-0.06	0.04	0.36	-0.54	0.31	0.11
residual sugar	0.11	0.00	0.14	1.00	0.06	0.19	0.20	0.36	-0.09	0.01	0.04
chlorides	0.09	0.06	0.20	0.06	1.00	0.01	0.05	0.20	-0.27	0.37	-0.22
free sulfur dioxide	-0.15	-0.01	-0.06	0.19	0.01	1.00	0.67	-0.02	0.07	0.05	-0.07
total sulfur dioxide	-0.11	0.08	0.04	0.20	0.05	0.67	1.00	0.07	-0.07	0.04	-0.21
density	0.67	0.02	0.36	0.36	0.20	-0.02	0.07	1.00	-0.34	0.15	-0.50
pH	-0.68	0.23	-0.54	-0.09	-0.27	0.07	-0.07	-0.34	1.00	-0.20	0.21
sulphates	0.18	-0.26	0.31	0.01	0.37	0.05	0.04	0.15	-0.20	1.00	0.09
alcohol	-0.06	-0.20	0.11	0.04	-0.22	-0.07	-0.21	-0.50	0.21	0.09	1.00

Part 2a: No regularization

```
In [ ]: basicReg2.fit()
print(f'mse: {basicReg2.mse(basicReg2.beta(), "test")}\n')
pd.DataFrame(data=np.around(np.vstack((basicReg2.beta(),
                                         basicReg2.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']],features2.columns)))
```

mse: 0.4360136340429069

Out[]:

	beta	stderr	zscore
bias	5.62	0.02	314.70
fixed acidity	0.07	0.05	1.32
volatile acidity	-0.21	0.02	-8.60
citric acid	-0.05	0.03	-1.68
residual sugar	0.03	0.02	1.11
chlorides	-0.09	0.02	-3.90
free sulfur dioxide	0.04	0.02	1.46
total sulfur dioxide	-0.09	0.03	-3.54
density	-0.04	0.04	-0.91
pH	-0.08	0.03	-2.43
sulphates	0.15	0.02	6.80
alcohol	0.32	0.03	10.17

Part 2b: Ridge regularization

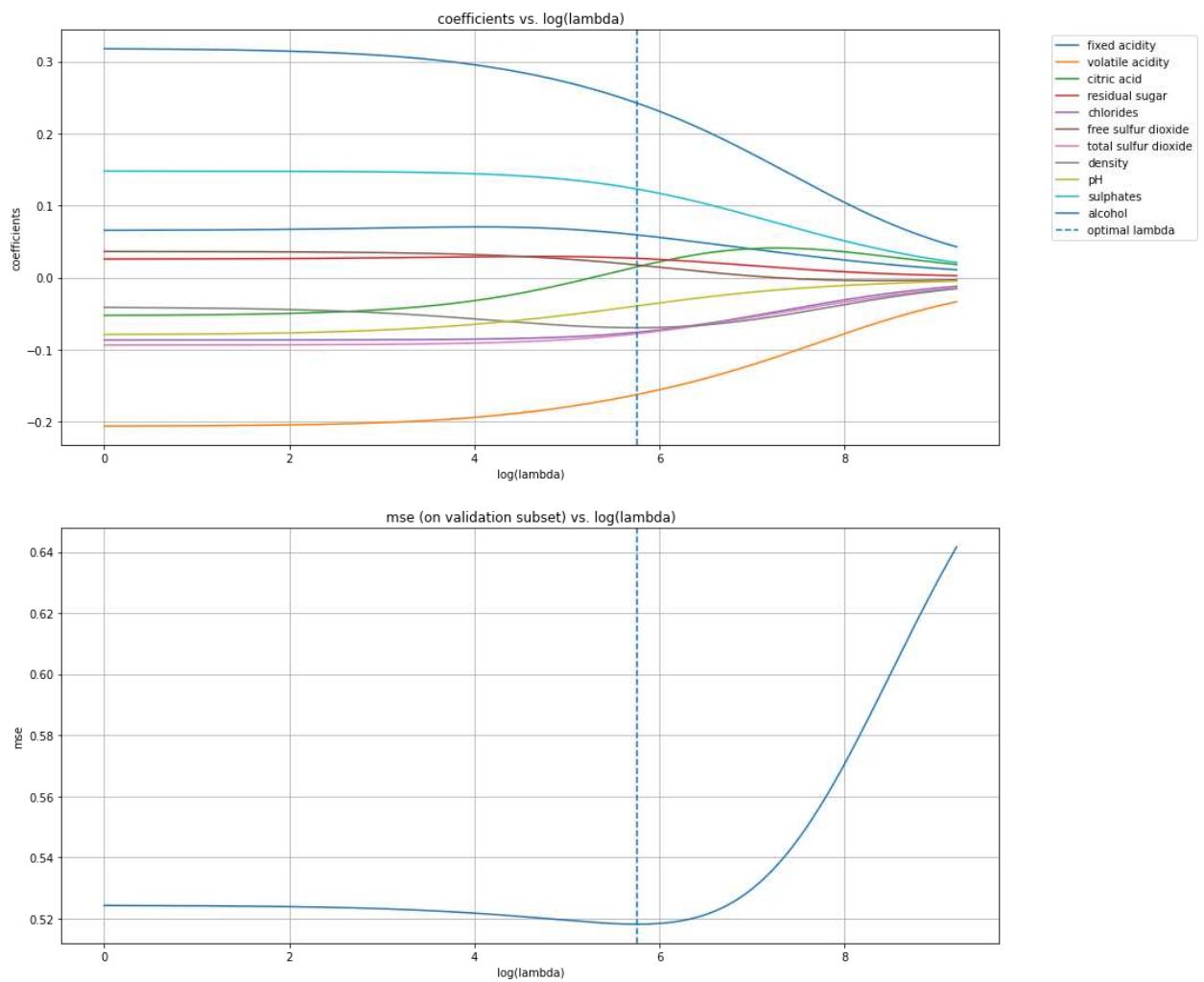
```
In [ ]: ridgeReg2 = RidgeLinearRegressionModel(copySubsetsFrom=basicReg2)
ridgeReg2.fit()
print(f'mse: {ridgeReg2.mse(ridgeReg2.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((ridgeReg2.beta(),
                                         ridgeReg2.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']],features2.columns)))
```

mse: 0.4164360004441624

Out[]:

	beta	stderr	zscore
bias	5.62	0.02	311.84
fixed acidity	0.06	0.05	1.18
volatile acidity	-0.16	0.02	-6.70
citric acid	0.02	0.03	0.47
residual sugar	0.03	0.02	1.15
chlorides	-0.08	0.02	-3.38
free sulfur dioxide	0.02	0.03	0.70
total sulfur dioxide	-0.08	0.03	-2.91
density	-0.07	0.05	-1.54
pH	-0.04	0.03	-1.19
sulphates	0.12	0.02	5.59
alcohol	0.24	0.03	7.67

```
In [ ]: ridgeReg2.plotValidation().show()
```



Part 2c: Lasso regularization

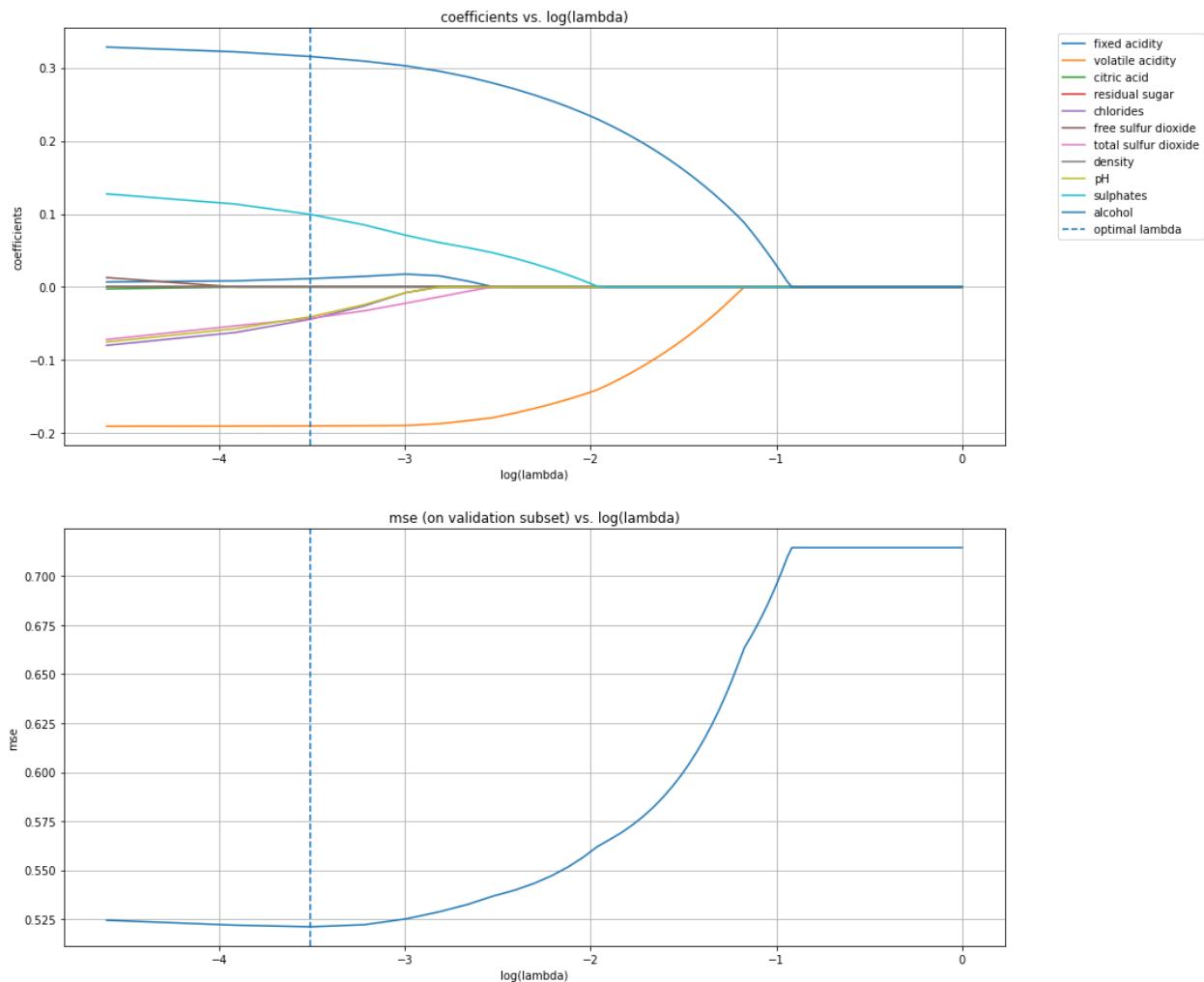
```
In [ ]: lassoReg2 = LassoLinearRegressionModel(copySubsetsFrom=basicReg2)
lassoReg2.fit()
print(f'mse: {lassoReg2.mse(lassoReg2.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((lassoReg2.beta(),
                                         lassoReg2.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']],features2.columns)))
```

mse: 0.4218980815804822

Out[]:

	beta	stderr	zscore
bias	5.62	0.02	311.62
fixed acidity	0.01	0.05	0.23
volatile acidity	-0.19	0.02	-7.84
citric acid	0.00	0.03	0.00
residual sugar	0.00	0.02	0.00
chlorides	-0.04	0.02	-1.96
free sulfur dioxide	0.00	0.03	0.00
total sulfur dioxide	-0.04	0.03	-1.60
density	-0.00	0.05	-0.00
pH	-0.04	0.03	-1.23
sulphates	0.10	0.02	4.52
alcohol	0.32	0.03	9.99

```
In [ ]: lassoReg2.plotValidation().show()
```



Part 3: Feature engineering

As a basic form of feature engineering, I'll try to add some powers (nonlinear terms) of the original features of the red wine dataset and see if this improves the results.

(I also messed around somewhat with interaction terms by summing/multiplying arbitrary features, but was unable to get any sort of consistent and significant results from this. I wasn't sure of a systematic way of displaying any interaction terms, so I left them out.)

```
In [ ]: features3 = features2.copy()
labels3 = labels2.copy()

features3Squared = features3 ** 2
features3Squared.columns = [f'{name}^2' for name in features3Squared.columns]
features3Cubed = features3 ** 3
features3Cubed.columns = [f'{name}^3' for name in features3Cubed.columns]

features3All = pd.concat([features3, features3Squared, features3Cubed], axis=1)
print('Preview of engineered features')
features3All
```

Preview of engineered features

Out[]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	(fixed acidity)^2	(vc acid)
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	54.76	0.4
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	60.84	0.7
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	60.84	0.5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	125.44	0.0
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	54.76	0.4
...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	38.44	0.3
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	34.81	0.3
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	39.69	0.2
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	34.81	0.4
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	36.00	0.0

1599 rows × 33 columns

```
In [ ]: basicReg3 = BasicLinearRegressionModel(features3All.to_numpy(),
                                             labels3.to_numpy(), features3All.columns)
print(f'baseline mse: {basicReg3.baselineMse()}\n')
print('Feature correlation')
pd.DataFrame(data=np.around(basicReg3.corrCoef(), 2),
              index=features3All.columns,
              columns=features3All.columns)
```

baseline mse: 0.6722580082660844

Feature correlation

Out[]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	(fixed acidity)^2
fixed acidity	1.00	-0.26	0.67	0.11	0.09	-0.15	-0.11	0.67	-0.68	0.18	-0.06	0.99
volatile acidity	-0.26	1.00	-0.55	0.00	0.06	-0.01	0.08	0.02	0.23	-0.26	-0.20	-0.26
citric acid	0.67	-0.55	1.00	0.14	0.20	-0.06	0.04	0.36	-0.54	0.31	0.11	0.67
residual sugar	0.11	0.00	0.14	1.00	0.06	0.19	0.20	0.36	-0.09	0.01	0.04	0.11
chlorides	0.09	0.06	0.20	0.06	1.00	0.01	0.05	0.20	-0.27	0.37	-0.22	0.09
free sulfur dioxide	-0.15	-0.01	-0.06	0.19	0.01	1.00	0.67	-0.02	0.07	0.05	-0.07	-0.15
total sulfur dioxide	-0.11	0.08	0.04	0.20	0.05	0.67	1.00	0.07	-0.07	0.04	-0.21	-0.11
density	0.67	0.02	0.36	0.36	0.20	-0.02	0.07	1.00	-0.34	0.15	-0.50	0.67
pH	-0.68	0.23	-0.54	-0.09	-0.27	0.07	-0.07	-0.34	1.00	-0.20	0.21	-0.68
sulphates	0.18	-0.26	0.31	0.01	0.37	0.05	0.04	0.15	-0.20	1.00	0.09	0.18
alcohol	-0.06	-0.20	0.11	0.04	-0.22	-0.07	-0.21	-0.50	0.21	0.09	1.00	-0.06
(fixed acidity)^2	0.99	-0.25	0.66	0.12	0.08	-0.15	-0.12	0.65	-0.65	0.18	-0.03	1.00
(volatile acidity)^2	-0.22	0.97	-0.49	-0.00	0.05	-0.02	0.05	-0.00	0.23	-0.25	-0.16	-0.22
(citric acid)^2	0.66	-0.47	0.95	0.15	0.25	-0.09	-0.02	0.37	-0.50	0.31	0.14	0.66
(residual sugar)^2	0.06	-0.02	0.10	0.93	0.07	0.20	0.18	0.28	-0.07	0.00	-0.01	0.06
(chlorides)^2	0.03	0.02	0.20	0.01	0.94	0.01	0.03	0.09	-0.23	0.40	-0.15	0.03
(free sulfur dioxide)^2	-0.11	-0.02	-0.04	0.26	0.03	0.94	0.57	0.01	0.04	0.04	-0.06	-0.11
(total sulfur dioxide)^2	-0.09	0.05	0.07	0.23	0.02	0.53	0.92	0.03	-0.11	0.02	-0.14	-0.09
(density)^2	0.67	0.02	0.37	0.36	0.20	-0.02	0.07	1.00	-0.34	0.15	-0.50	0.67
(pH)^2	-0.68	0.24	-0.54	-0.09	-0.26	0.07	-0.07	-0.34	1.00	-0.19	0.21	-0.68
(sulphates)^2	0.15	-0.20	0.27	-0.01	0.41	0.05	0.08	0.12	-0.23	0.96	0.04	0.15
(alcohol)^2	-0.07	-0.20	0.11	0.04	-0.22	-0.07	-0.20	-0.50	0.20	0.09	1.00	-0.07
(fixed acidity)^3	0.96	-0.24	0.63	0.12	0.06	-0.15	-0.12	0.63	-0.61	0.17	-0.00	0.96
(volatile acidity)^3	-0.18	0.88	-0.40	-0.01	0.04	-0.03	0.03	-0.02	0.20	-0.22	-0.11	-0.18
(citric acid)^3	0.60	-0.37	0.85	0.15	0.29	-0.09	-0.04	0.35	-0.46	0.30	0.13	0.60
(residual sugar)^3	0.03	-0.03	0.08	0.81	0.07	0.19	0.14	0.22	-0.06	0.00	-0.04	0.03
(chlorides)^3	0.01	0.00	0.19	-0.01	0.84	0.01	0.02	0.05	-0.19	0.38	-0.11	0.01
(free sulfur dioxide)^3	-0.09	-0.02	-0.03	0.31	0.04	0.80	0.45	0.03	0.02	0.02	-0.05	-0.09
(total sulfur dioxide)^3	-0.05	0.01	0.09	0.22	-0.01	0.34	0.70	-0.02	-0.11	-0.00	-0.05	-0.05
(density)^3	0.67	0.02	0.37	0.36	0.20	-0.02	0.07	1.00	-0.34	0.15	-0.50	0.67
(pH)^3	-0.68	0.24	-0.53	-0.09	-0.25	0.07	-0.06	-0.34	1.00	-0.18	0.21	-0.68
(sulphates)^3	0.10	-0.13	0.21	-0.01	0.40	0.05	0.11	0.09	-0.24	0.86	-0.01	0.10
(alcohol)^3	-0.07	-0.20	0.11	0.05	-0.21	-0.07	-0.19	-0.50	0.20	0.09	0.99	-0.07

Part 3a: No regularization

```
In [ ]: basicReg3.fit()
print(f'mse: {basicReg3.mse(basicReg3.beta(), "test")}\n')
pd.DataFrame(data=np.around(np.vstack((basicReg3.beta(),
                                         basicReg3.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([['bias']],features3All.columns)))
```

mse: 0.3875555114427124

Out[]:

	beta	stderr	zscore
bias	5.61	0.02	315.19
fixed acidity	0.19	0.78	0.24
volatile acidity	-0.42	0.26	-1.60
citric acid	0.00	0.13	0.02
residual sugar	-0.07	0.16	-0.44
chlorides	-0.18	0.13	-1.40
free sulfur dioxide	0.28	0.15	1.85
total sulfur dioxide	0.09	0.14	0.67
density	-1701.64	5690.80	-0.30
pH	4.42	8.40	0.53
sulphates	1.54	0.30	5.08
alcohol	-7.29	3.63	-2.01
(fixed acidity)^2	-0.03	1.51	-0.02
(volatile acidity)^2	0.53	0.51	1.03
(citric acid)^2	-0.24	0.29	-0.84
(residual sugar)^2	0.20	0.32	0.63
(chlorides)^2	0.39	0.28	1.38
(free sulfur dioxide)^2	-0.48	0.26	-1.85
(total sulfur dioxide)^2	-0.36	0.20	-1.77
(density)^2	3392.76	11382.99	0.30
(pH)^2	-8.22	16.64	-0.49
(sulphates)^2	-2.33	0.58	-3.99
(alcohol)^2	14.77	7.14	2.07
(fixed acidity)^3	-0.10	0.75	-0.13
(volatile acidity)^3	-0.29	0.27	-1.05
(citric acid)^3	0.20	0.19	1.07
(residual sugar)^3	-0.12	0.19	-0.65
(chlorides)^3	-0.29	0.17	-1.65
(free sulfur dioxide)^3	0.26	0.14	1.82
(total sulfur dioxide)^3	0.21	0.10	2.08
(density)^3	-1691.18	5692.22	-0.30
(pH)^3	3.72	8.26	0.45
(sulphates)^3	0.99	0.30	3.26
(alcohol)^3	-7.22	3.53	-2.05

Part 3b: Ridge regularization

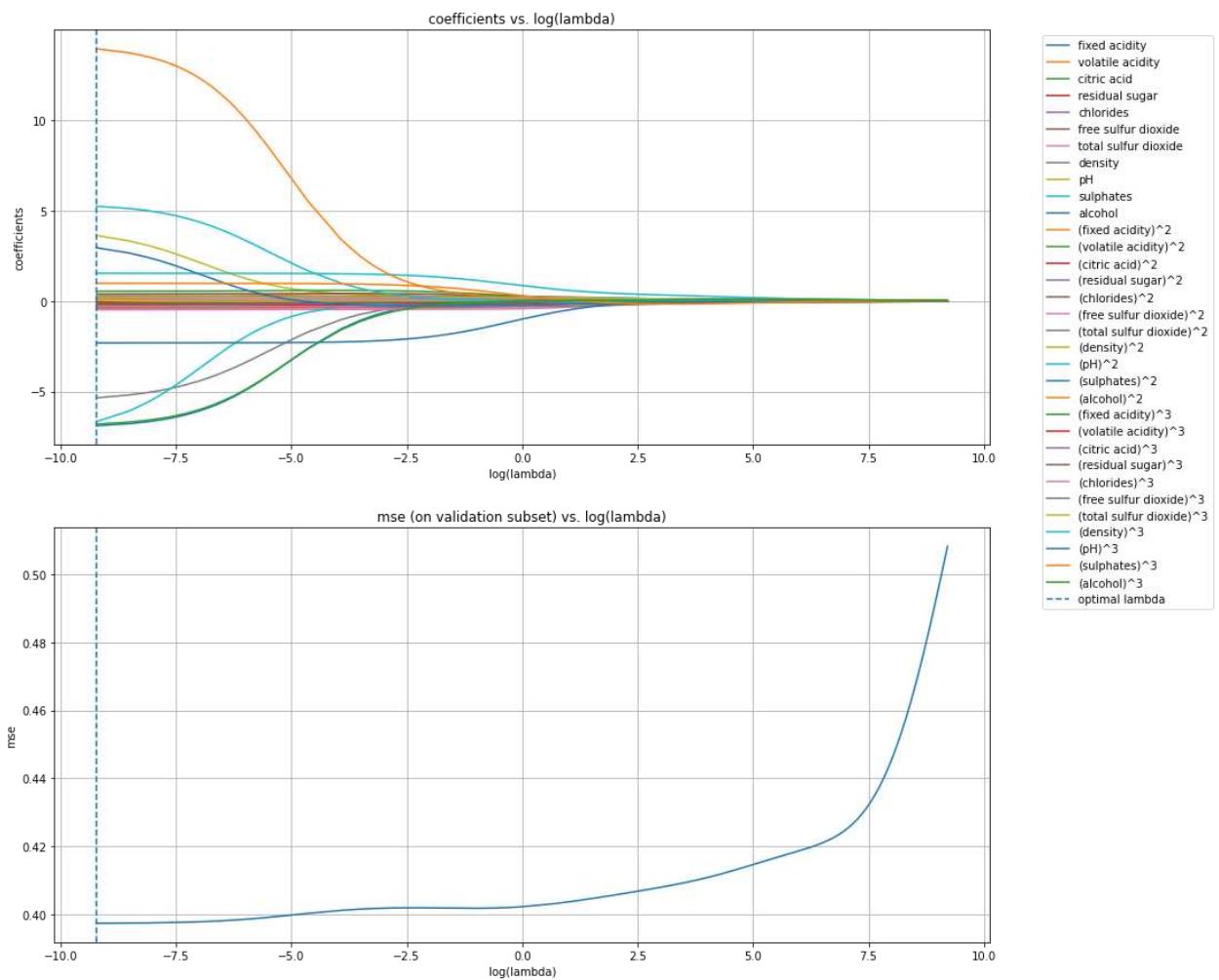
```
In [ ]: ridgeReg3 = RidgeLinearRegressionModel(copySubsetsFrom=basicReg3)
ridgeReg3.fit()
print(f'mse: {ridgeReg3.mse(ridgeReg3.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((ridgeReg3.beta(),
                                         ridgeReg3.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([ 'bias'],features3All.columns)))
```

mse: 0.3878974791375196

Out[]:

	beta	stderr	zscore
bias	5.61	0.02	315.18
fixed acidity	0.13	0.78	0.16
volatile acidity	-0.43	0.26	-1.63
citric acid	0.00	0.13	0.01
residual sugar	-0.07	0.16	-0.44
chlorides	-0.18	0.13	-1.42
free sulfur dioxide	0.28	0.15	1.86
total sulfur dioxide	0.09	0.14	0.66
density	-5.35	5691.01	-0.00
pH	3.62	8.40	0.43
sulphates	1.54	0.30	5.06
alcohol	-6.89	3.63	-1.90
(fixed acidity)^2	0.11	1.51	0.07
(volatile acidity)^2	0.54	0.51	1.06
(citric acid)^2	-0.24	0.29	-0.83
(residual sugar)^2	0.20	0.32	0.64
(chlorides)^2	0.39	0.28	1.40
(free sulfur dioxide)^2	-0.48	0.26	-1.87
(total sulfur dioxide)^2	-0.35	0.20	-1.75
(density)^2	0.05	11383.41	0.00
(pH)^2	-6.66	16.64	-0.40
(sulphates)^2	-2.32	0.58	-3.97
(alcohol)^2	13.95	7.15	1.95
(fixed acidity)^3	-0.18	0.75	-0.24
(volatile acidity)^3	-0.29	0.27	-1.08
(citric acid)^3	0.20	0.19	1.06
(residual sugar)^3	-0.13	0.19	-0.67
(chlorides)^3	-0.29	0.17	-1.66
(free sulfur dioxide)^3	0.26	0.14	1.84
(total sulfur dioxide)^3	0.20	0.10	2.06
(density)^3	5.24	5692.43	0.00
(pH)^3	2.95	8.26	0.36
(sulphates)^3	0.99	0.30	3.24
(alcohol)^3	-6.80	3.53	-1.93

```
In [ ]: ridgeReg3.plotValidation().show()
```



Part 3c: Lasso regularization

```
In [ ]: lassoReg3 = LassoLinearRegressionModel(copySubsetsFrom=basicReg3)
lassoReg3.fit()
print(f'mse: {lassoReg3.mse(lassoReg3.beta())}\n')
pd.DataFrame(data=np.around(np.vstack((lassoReg3.beta(),
                                         lassoReg3.zScores())),2).T,
              columns=['beta', 'stderr', 'zscore'],
              index=np.hstack(([bias'],features3All.columns)))
```

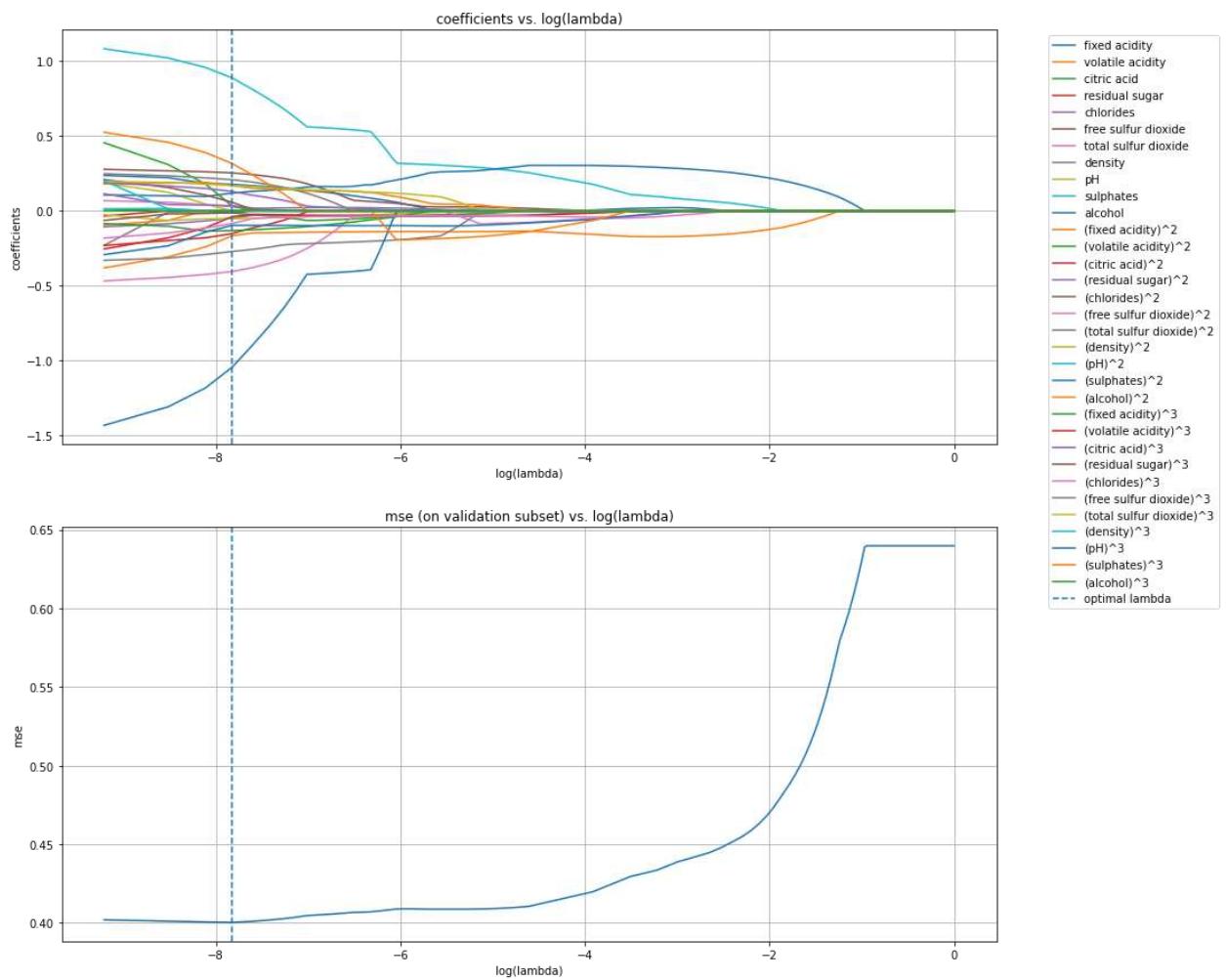
```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 167.93274436154616, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 113.00987914063819, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 61.944968997659885, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 28.808391083487777, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 15.466242922416711, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 7.277722951212638, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 2.674452762610656, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 1.9201090298432177, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 1.564819994521315, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 1.3161042136997594, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 0.5523356411632108, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 0.2765362897703767, tolerance: 0.08329695074276783
    positive)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_coordinate_descent.py:476: Convergence
Warning: Objective did not converge. You might want to increase the number of iterations. Duality g
ap: 0.14244091888997445, tolerance: 0.08329695074276783
    positive)

mse: 0.3932675740271798
```

Out[]:

	beta	stderr	zscore
bias	5.61	0.02	313.49
fixed acidity	0.18	0.79	0.23
volatile acidity	-0.17	0.27	-0.62
citric acid	-0.01	0.13	-0.10
residual sugar	0.00	0.16	0.00
chlorides	-0.05	0.13	-0.38
free sulfur dioxide	0.25	0.15	1.69
total sulfur dioxide	0.03	0.14	0.21
density	-0.00	5721.67	-0.00
pH	0.00	8.44	0.00
sulphates	0.89	0.31	2.91
alcohol	0.12	3.65	0.03
(fixed acidity)^2	0.00	1.51	0.00
(volatile acidity)^2	0.03	0.52	0.06
(citric acid)^2	-0.15	0.29	-0.52
(residual sugar)^2	0.04	0.32	0.11
(chlorides)^2	0.06	0.28	0.21
(free sulfur dioxide)^2	-0.40	0.26	-1.56
(total sulfur dioxide)^2	-0.27	0.20	-1.33
(density)^2	-0.05	11444.73	-0.00
(pH)^2	0.00	16.73	0.00
(sulphates)^2	-1.04	0.59	-1.78
(alcohol)^2	0.17	7.18	0.02
(fixed acidity)^3	-0.13	0.75	-0.17
(volatile acidity)^3	-0.04	0.27	-0.14
(citric acid)^3	0.13	0.19	0.69
(residual sugar)^3	-0.01	0.19	-0.07
(chlorides)^3	-0.08	0.18	-0.48
(free sulfur dioxide)^3	0.21	0.14	1.48
(total sulfur dioxide)^3	0.17	0.10	1.70
(density)^3	-0.00	5723.09	-0.00
(pH)^3	-0.10	8.31	-0.01
(sulphates)^3	0.32	0.31	1.03
(alcohol)^3	0.00	3.55	0.00

```
In [ ]: lassoReg3.plotValidation().show()
```



Answers to questions

Which features did the Lasso select for you to include in your model? Do these features make sense?

- In the prostate cancer dataset, mostly only the lcp and age features were suppressed with the Lasso. I don't know much about prostate cancer, so I can't interpret the results of this too much.
- In the red wine dataset, there were a few coefficients that were very low even in the no-regularization case, such as fixed acidity, density, and pH. The factors that were most severely suppressed were the total and fixed sulfur concentrations, as well as the sulfates feature. I don't know much about wine either, but it makes sense that people are not looking for the amount of sulfur in their wine when considering whether it's good or not.

Compute the MSE on the training dataset and the test dataset for all methods and comment on the results. Compare this MSE to a baseline MSE.

In both datasets, the baseline MSE was higher than the MSE calculated for any of the regression methods, as expected. See the chart following this section.

Stretch goal: Add nonlinear and interaction terms to your dataset and try to improve the performance. Are you able to do so?

I tried adding arbitrary interaction terms, but without knowing much about what makes quality wine I was unable to find a combination that consistently produced significant results (and I wasn't able to find a way to do this systematically). The above example shows adding nonlinear terms (squared and cubed values of the original sample dataset). When doing this:

- The baseline MSE stayed around the same value (≈ 0.5 to 0.7).
- The regression MSEs were a little lower than without these extra features (≈ 0.3 to 0.4 , rather than ≈ 0.4 to 0.5 without the extra features).
- The regularized models don't seem to outperform the no-regularization model by any significant amount.

Thus, even by this very basic feature engineering (adding powers of features), there seems to be a small but noticeable improvement in overall MSEs.

```
In [ ]: print('MSE summary')
mses = [[basicReg.baselineMse(), basicReg.mse(basicReg.beta()),
         ridgeReg.mse(ridgeReg.beta()), lassoReg.mse(lassoReg.beta())],
        [basicReg2.baselineMse(), basicReg2.mse(basicReg2.beta()),
         ridgeReg2.mse(ridgeReg2.beta()), lassoReg2.mse(lassoReg2.beta())],
        [basicReg3.baselineMse(), basicReg3.mse(basicReg3.beta()),
         ridgeReg3.mse(ridgeReg3.beta()), lassoReg3.mse(lassoReg3.beta())]]
pd.DataFrame(data=mses,
              columns=['baseline', 'no reg.', 'ridge', 'lasso'],
              index=['prostate cancer', 'red wine', 'red wine feature eng.'])
```

MSE summary

Out[]:

	baseline	no reg.	ridge	lasso
prostate cancer	0.880802	0.495767	0.482037	0.448812
red wine	0.564569	0.436014	0.416436	0.421898
red wine feature eng.	0.672258	0.387556	0.387897	0.393268

Other comments

- There was a lot of variation between runs, so there were often runs where ridge and/or lasso did worse than the no-regularization case. Given the small number of samples (small training set, validation set, and testing set all are not helpful) and a single validation (rather than k-fold cross validation), this is somewhat expected.
- There were even some times that the baseline performed better than the regressions, which made me very skeptical; however, I verified the numbers in some of these cases and conclude that this is due to the wide variation between randomly-selected train and test datasets (again due to the small size), not due to a problem in the model.
- On some runs, the validation gave a minimum MSE when $\lambda = 0$ (or $\alpha = 0$). Again, this is probably due to the small sample size.

ECE475 - Frequentist Machine Learning

Assignment 2 -- Logistic Regression

Jonathan Lam, Tiffany Yu, Harris Paspuleti

Implement logistic regression with stochastic gradient descent as the optimization algorithm, with and without the L2 regularization penalty.

Divide your data into roughly 80% train, 10% validation, 10% test as in the previous assignment and use the validation dataset to tune any parameters.

Defining the Models

```
In [1]: # Setting up
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from tabulate import tabulate
from sklearn import preprocessing
import seaborn as sb
from functools import partial

/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: Future
Warning: pandas.util.testing is deprecated. Use the functions in the public API
at pandas.testing instead.
    import pandas.util.testing as tm
```

Base Logistic Classifier

All logistic classifiers subclass from this. This mostly includes the subset initialization (80/10/10 split and optional standardization), since this is common for all of the classifiers.

Each classifier must implement its own `train()` or `validate()` methods.

```
In [2]: # base logistic classifier class; includes some common utilities;
# this class is abstract and doesn't contain a train() or validate() method,
# which must be implemented in its inheritors
class BaseLogisticClassifier:

    # X: NxP ndarray (features)
    # y: Nx1 ndarray (labels)
    # alpha: (maximum) learning rate
    # copySubsetsFrom: treat this as a copy constructor, copy
    # over the subsets from a different BaseLogisticClassifier
    def __init__(self, X, y, alpha=0.01,
                 copySubsetsFrom=None,
                 standardizeFeatures=False):
        self._alpha = alpha
        self._lambda = 0

        # copySubsetsFrom is provided; copy subsplits
        if copySubsetsFrom is not None:
            self._subsets = copySubsetsFrom._subsets.copy()
            P = self._subsets['train']['X'].shape[1] - 1

        # X, y are provided; manually split subsets
        else:
            N, P = X.shape

            # add column of 1's to X
            X = np.hstack((np.ones((N, 1)), X))

            # randomly split data into training, validation, test
            indices = np.arange(N)
            np.random.shuffle(indices)
            split1, split2 = int(N*0.8), int(N*0.9)
            self._subsets = {
                'train': {
                    'X': X[indices[:split1], :],
                    'y': y[indices[:split1], :]
                },
                'validate': {
                    'X': X[indices[split1:split2], :],
                    'y': y[indices[split1:split2], :]
                },
                'test': {
                    'X': X[indices[split2:], :],
                    'y': y[indices[split2:], :]
                }
            }

            # print the lengths of the dataset and each set
            print("Length of dataset:", N)
            print("Length of training:", split1)
            print("Length of validation:", split2-split1)
            print("Length of test:", N-split2)

        # initialize weight vector (includes the bias, hence the P+1)
        self._theta = np.zeros((P+1, 1))

        # initialize Adam coefficients
        # Adam is an optimization algorithm that can be
        # used instead of the classical stochastic gradient descent procedure
        # to update network weights iterative based in training data.
        # adam optimizer
        # beta1. The exponential decay rate for the first moment estimates (e.
        # A q)
```

Binary Logistic Classifier

This is the unregularized case for the binary logistic classifier. It uses an Adam optimizer in the `step()` function for faster, smoother convergence. ([Reference for Adam implementation \(https://arxiv.org/abs/1412.6980\)](https://arxiv.org/abs/1412.6980))

Notes on implementation:

- The Adam optimizer is used for better convergence than SGD.
- The functions are all vectorized and the batch size is arbitrary. Right now, `train()` uses the entire training dataset on each iteration (full batch).

In [3]:

```
class BinaryLogisticClassifier(BaseLogisticClassifier):
```

```
    # hypothesis function (returns yhat); uses trained theta
    # returns N x 1
    def h(self, X):
        return 1 / (1 + np.exp(-X @ self._theta))

    # update function
    # theta_j := theta_j + alpha(y_i - h_theta(x_i)) * x_i_j
    # returns (P+1) x 1
    def grad(self, X, y):
        return X.T @ (y - self.h(X))

    # log likelihood
    def l(self, subset):
        X, y = self._subsets[subset]['X'], self._subsets[subset]['y']
        return y.T @ np.log(self.h(X)) + (1 - y).T @ np.log(1 - self.h(X))

    # percent classified wrong on training subset
    def pctWrong(self, subset='test'):
        X, y = self._subsets[subset]['X'], self._subsets[subset]['y']
        N, _ = X.shape

        # epsilon to prevent prediction of exactly 0.5 to be classified as
        # correct for label being either 0 or 1
        ep = 0.00001

        return np.sum(np.round(np.abs(self.h(X) - y - ep))) / N

    # adam update step
    def step(self, iter, includeMask=None):
        # update adam moments
        thetaGrad = self.grad(self._subsets['train']['X'], self._subsets['train']['y'])
        # weighted average of the gradient
        self._ztheta = self._beta1 * self._ztheta + (1 - self._beta1) * thetaGrad
        ad
        # weighted average of the square gradient
        self._zthetaSquared = self._beta2 * self._zthetaSquared + (1 - self._beta2) * thetaGrad ** 2

        # adam bias-corrected moments
        bcZTheta = self._ztheta / (1 - self._beta1 ** (iter + 1))
        bcZThetaSquared = self._zthetaSquared / (1 - self._beta2 ** (iter + 1))

        # adam update rule
        self._theta += self._alpha * bcZTheta / (np.sqrt(bcZThetaSquared) + self._ep)

        # exclude certain features (for stepwise)
        if includeMask is not None:
            self._theta *= includeMask

    def train(self, iterations=2000, includeMask=None):
        # store loglikelihoods for graphing later
        self._loglikelihoods = np.zeros(iterations)
        self._theta = np.zeros((self._subsets['train']['X'].shape[1], 1))

        for i in range(iterations):
            self.step(i, includeMask)
            self._loglikelihoods[i] = self.l(subset='train')

    # baseline sets the bias to the average label and zeros elsewhere
```

Stepwise Logistic Classifier

This inherits from the binary logistic classifier, and adds a validate function that follows the algorithm:

- Create a list of the features called `exclude`
- Create an empty list called `include`
- Train the model with no features
- While `exclude` is not empty:
 - Loop through the features of `exclude` :
 - Add the current feature to the model, and calculate the % classified incorrectly
 - Choose the feature that, when added to the existing model, provides the lowest % classification error
 - Add this feature to `include` and to the existing model, and remove it from `exclude`
- We now have P different models, each of which has a different number of features included. Return the model that has the lowest classification error.

```
In [4]: class StepwiseLogisticClassifier(BinaryLogisticClassifier):

    def validate(self):
        _, P = self._subsets['train']['X'].shape
        P -= 1

        # list of features to exclude and include
        exclude = list(range(P))
        include = []
        # list of features to include
        includeMask = np.zeros((P+1, 1))
        includeMask[0] = 1

        pctWrongs = np.zeros((P+1, 1))
        #calculate the percent wrong relative to the validate set of data
        pctWrongs[0] = 1 - np.mean(self._subsets['validate']['y'])

        # loops over number of features in model
        for i in range(P):

            # find best next feature to include
            bestPctWrong, bestFeature = float('inf'), None
            for feature in exclude:
                # copy includeMask into currentIncludeMask, unmask feature
                currentIncludeMask = np.array(includeMask)
                currentIncludeMask[feature+1] = 1

                # train on currentIncludeMask
                self.train(includeMask=currentIncludeMask)

                # calculate percent wrong on validation set
                #Trying to find out when it gives you the least error
                pctWrong = self.pctWrong(subset='validate')
                if pctWrong < bestPctWrong:
                    bestPctWrong = pctWrong
                    bestFeature = feature

            # minimize percent wrong
            pctWrongs[i+1] = bestPctWrong

            # add feature to includeMask, remove from exclude
            exclude.remove(bestFeature)
            include.append(bestFeature)
            includeMask[bestFeature] = 1

        # find minimum of pctWrongs
        bestNumFeatures = np.argwhere(pctWrongs == np.min(pctWrongs))[0,0]
        bestIncludeMask = np.zeros((P+1, 1))
        bestIncludeMask[0] = 1
        for i in range(bestNumFeatures):
            bestIncludeMask[include[i]+1] = 1

        # retrain with best include mask, return theta
        self.train(includeMask=bestIncludeMask)
        return self._theta, include[:bestNumFeatures]
```

L2 Logistic Classifier

This inherits from the binary logistic classifier, and modifies the gradient to penalize the bias.

The `validate()` method sweeps `lambda` through a logspace.

This also standardizes the features.

```
In [5]: #Class to calculate L2 regularization
class L2LogisticClassifier(BinaryLogisticClassifier):

    # make sure to standardize features
    def __init__(self, X, Y, alpha=0.01, copySubsetsFrom=None):
        super().__init__(X, Y, alpha=alpha,
                         copySubsetsFrom=copySubsetsFrom,
                         standardizeFeatures=True)

    # update function with L2 penalty
    # theta_j := theta_j + alpha(y_i - h_theta(x_i)) * x_i_j
    # returns (P+1)
    # SGD = j+alpha(y(i)-hθ(x(i)))x(i)j
    def grad(self, X, y):
        # don't penalize the bias
        return X.T @ (y - self.h(X)) - 2 * self._lambda * np.vstack((np.zeros((1,1)), self._theta[1:,:]))

    def validate(self):
        #create a bunch of lambdas in order to iterate through them
        lams = np.logspace(-20, 5, 100)

        #Removing the ones because we don't want to regularize the bias term
        P = self._subsets['train']['X'].shape[1] - 1
        self._subsets['train']['X'][0,:] = np.ones((1, P+1))
        self._subsets['validate']['X'][0,:] = np.ones((1, P+1))
        self._subsets['test']['X'][0,:] = np.ones((1, P+1))

        bestPctWrong, bestLambda = float('inf'), None
        pctWrongs = np.zeros_like(lams)

        for i, lam in enumerate(lams):
            self._lambda = lam
            self.train()

            pctWrong = self.pctWrong(subset='validate')
            pctWrongs[i] = pctWrong
            # calculate percent wrong on validation set
            #Trying to find out when it gives you the least error
            if pctWrong < bestPctWrong:
                bestPctWrong = pctWrong
                bestLambda = lam

            self._lambda = bestLambda
            self.train()
        return self._theta
```

L1 Logistic Classifier

Stretch goal #1 (3 points): Implement the L1 penalty as well, and produce a Lasso plot like figure 4.13. Include your results in the % correct table. Use the validation dataset to select the optimal lambda and determine the most important features. Do those features agree with the stepwise feature selection?

There are lots of ways to implement the L1 penalty, one possible way is the naive one detailed in this paper:
<https://www.aclweb.org/anthology/P09-1054.pdf> (<https://www.aclweb.org/anthology/P09-1054.pdf>)

This inherits from the binary logistic classifier. It standardizes the features, and applies a L1 penalty using an approximation for the gradient of the L1 penalty loss term as described in the above paper.

```
In [6]: # taken mostly literally from (Tsuruoka et al., 2009); involves an
# estimate of the gradient of the L1 norm (abs function) that involves
# some "memory" for improved performance
class L1LogisticClassifier(BinaryLogisticClassifier):

    # make sure to standardize features
    def __init__(self, X, Y, alpha=0.01, copySubsetsFrom=None):
        super().__init__(X, Y, alpha=alpha,
                         copySubsetsFrom=copySubsetsFrom,
                         standardizeFeatures=True)

    # apply this after Adam update rule (would be difficult to incorporate with
    # Adam)
    def applyL1Penalty(self):
        for i, theta_i in enumerate(self._theta.reshape(-1)):
            # start from 1 to not penalize the bias
            if i == 0:
                continue

            z = theta_i
            if theta_i > 0:
                self._theta[i, 0] = max(0., theta_i - (self._u + self._q[i]))
            elif theta_i < 0:
                self._theta[i, 0] = min(0., theta_i + (self._u + self._q[i]))
            self._q[i] += theta_i - z

    # log likelihood
    def l(self, subset):
        X, y = self._subsets[subset]['X'], self._subsets[subset]['y']
        return y.T @ np.log(self.h(X)) + (1 - y).T @ np.log(1 - self.h(X))

    def train(self, iterations):
        self._theta *= 0.
        self._u = 0.
        self._q = self._theta.copy().reshape(-1)
        self._N = self._subsets['train']['X'].shape[0]

        # loglikelihoods are for graphing later
        self._loglikelihoods = np.zeros(iterations)

        for i in range(iterations):
            self._u += self._alpha * self._C / self._N
            self.step(i)
            self.applyL1Penalty()
            self._loglikelihoods[i] = self.l(subset='train')

    def validate(self, iterations=2000):
        # just to be sure; undo l2 regularization
        self._lambda = 0.

        # l1 regularization parameter; C is the letter used in the text
        cIteration = np.logspace(-8, 0, 30)
        bestPctWrong, bestC = float('inf'), None
        pctWrongs = np.zeros_like(cIteration)

        # note: coefficients includes bias
        coefficients = np.zeros((cIteration.size, self._theta.size))

        for j, c in enumerate(cIteration):
            self._C = c
            self.train(iterations)

            coefficients[j, :] = self._theta.reshape(-1)

            pctWrongs[j] = np.mean(np.abs(coefficients[j, :] - self._theta))

        bestPctWrong = np.min(pctWrongs)
        bestC = cIteration[pctWrongs.argmin()]

```

Multinomial (Trinary) Logistic Regression

Stretch goal #2 (3 points): Extend your unregularized logistic regression to multinomial regression(i.e. more than binary classification). It is a pretty straightforward extension, but its not covered in elements of stats. You can google for derivations if you want, but mainly all you really need is to find the gradient of the loss function in the multinomial case. This is covered in section 4.3.4 of another classic ML text by Bishop(bootleg pdf here: <http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop%20-%20Pattern%20Recognition%20And%20Machine%20Learning%20-%20Springer%20%202006.pdf> (<http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop%20-%20Pattern%20Recognition%20And%20Machine%20Learning%20-%20Springer%20%202006.pdf>). The equation for the gradient is eqn 4.109. Test your approach on a simple M-ary classification dataset like the Iris dataset from UCI.

For lack of time, this implementation is hardcoded for a $K = 3$ problem (trinary), but it still incorporates the same concepts as any multinomial problem.

The dataset feature matrix shape should be the same, but the dataset labels should be a one-hot encoded matrix (i.e., $N \times K$).

Most of the functions are rewritten, since the binary classification equivalents are no longer sufficient. The Adam optimizer is replaced with a simpler gradient descent update rule.

```
In [7]: # multinomial case, hardcoded for K=3; uses simple SGA rather than Adam
class TrinaryLogisticClassifier(BaseLogisticClassifier):

    # returns NxK matrix, where each row is the predicted probabilities
    # of each of the K classes
    def h(self, X):
        a1 = np.exp(X @ self._theta1)
        a2 = np.exp(X @ self._theta2)
        return np.hstack((a1/(1+a1+a2), a2/(1+a1+a2), 1/(1+a1+a2)))

    # returns (gradTheta1, gradTheta2)
    def grad(self):
        X, y = self._subsets['train']['X'], self._subsets['train']['y']
        P = X.shape[1] - 1

        # eq. 4.109 (p. 209) of "Pattern Recognition and Machine Learning"
        # but a little vectorized
        grads = np.zeros((P+1, 2))
        for j in range(2):
            grads[:,j] = X.T @ (y[:,j] - self.h(X)[:,j])
        return grads

    # calculate percent wrong: compares argmax of estimate and label
    def pctWrong(self, subset='test'):
        X, y = self._subsets[subset]['X'], self._subsets[subset]['y']
        N = X.shape[0]
        return np.sum(np.abs(np.argmax(self.h(X), axis=1) - \
            np.argmax(y, axis=1))) / N

    # hardcoded 3-class classifier (e.g., for UCI Iris dataset)
    def trinaryClassificationTrain(self, iterations=2000):
        N, P = self._subsets['train']['X'].shape
        P -= 1

        # do the binary classification problem K-1 times
        self._theta1 = np.zeros((P+1, 1))
        self._theta2 = np.zeros((P+1, 1))

        for i in range(iterations):
            # use basic sgd (not adam)
            grads = self.grad()
            self._theta1 += self._alpha * grads[:,0][:,np.newaxis]
            self._theta2 += self._alpha * grads[:,1][:,np.newaxis]

        return self._theta1, self._theta2
```

Running the Models

Binary Classification of the SAHD Dataset

Replicate the analysis of the South African heart disease dataset from the Elements of Statistical Learning textbook and plot figure 4.12. Additionally, report the % correct for all 3 models (unregularized, stepwise, and L2 regularized) in a table. Instead of plotting the tables and dropping terms based on Z score, select the optimal model using forward stepwise via cross-validation and report which features are the most important.

```
In [8]: # Import the South African heart disease dataset (in Google Colab)
# Read through data and create dataset
sahdDataset = pd.read_csv('https://web.stanford.edu/~hastie/ElemStatLearn/datasets/SAheart.data', index_col=0)

# Textbook drops adiposity and typea
sahdDataset = sahdDataset.drop(['adiposity', 'typea'], axis=1)

# Turn famhist into a quantitative variable
sahdDataset['famhist'] = (sahdDataset['famhist'] == 'Present')*1

# Creates a graph like figure 4.12
sb.pairplot(sahdDataset, hue = 'chd', palette="hls", height = 3)

# list the features
term = list(sahdDataset.columns.values[:-1])

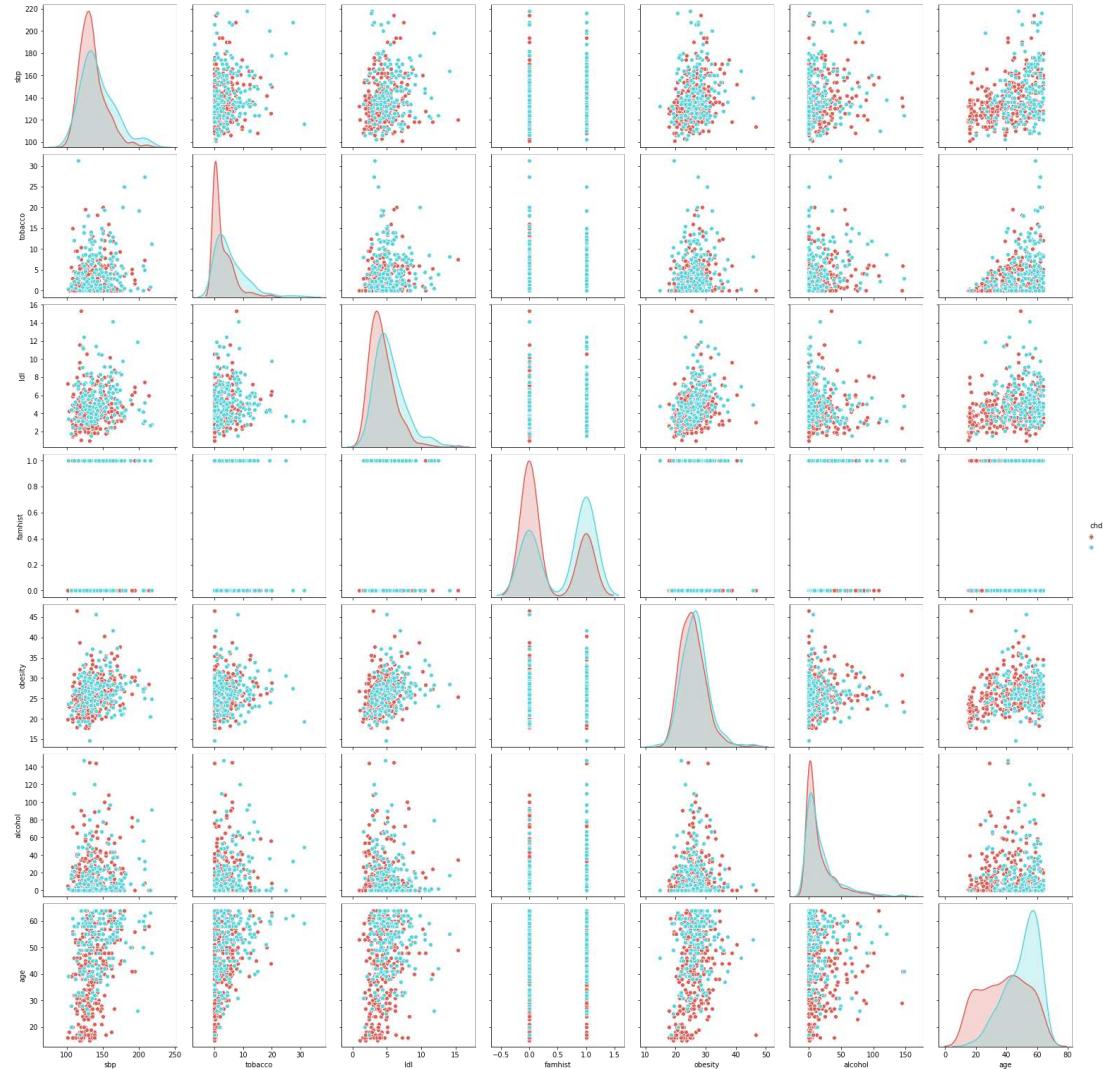
# Generate Features matrix : NxP
sahdDatasetX = sahdDataset.drop(['chd'], axis=1).to_numpy()
# Generate Label matrix : Nx1
sahdDatasety = sahdDataset.loc[:, 'chd'].to_numpy().reshape(-1, 1)

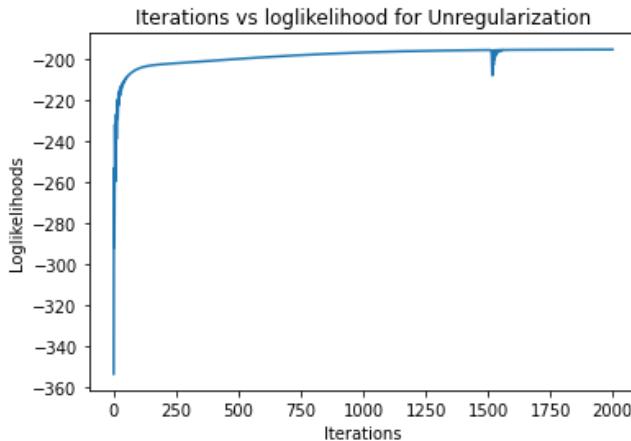
# PART 1: RECREATE TABLE 4.
binaryClassifier = BinaryLogisticClassifier(sahdDatasetX, sahdDatasety)
binaryClassifier.train()
binaryClassifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for Unregularization')
correct_unregularized = np.around((1 - binaryClassifier.pctWrong()) * 100)
print(f'theta: {binaryClassifier.theta()}\n% classified correct for unregularized: {np.around((1 - binaryClassifier.pctWrong()) * 100)}%')
correct_baseline = np.around((1. - binaryClassifier.baselinePctWrong()) * 100)
```

```

Length of dataset: 462
Length of training: 369
Length of validation: 46
Length of test: 47
theta: [[-3.97227696e+00]
        [ 2.62565496e-03]
        [ 6.69992005e-02]
        [ 1.82021868e-01]
        [ 8.19843244e-01]
        [-3.19375348e-02]
        [ 2.45009621e-03]
        [ 4.91780240e-02]]
% classified correct for unregularized: 72.0%

```



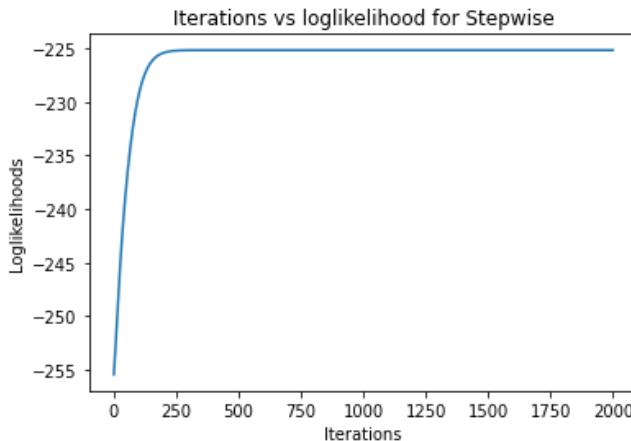


```
In [9]: # PART 2: STEPWISE
stepwiseClassifier = StepwiseLogisticClassifier(None, None, copySubsetsFrom=bin
aryClassifier)
_, optimalFeatures = stepwiseClassifier.validate()
stepwiseClassifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for Stepwise')

correct_stepwise = np.around((1 - stepwiseClassifier.pctWrong()) * 100)
print(f'theta: {stepwiseClassifier.theta()}\n% classified correct for stepwise:
{np.around((1 - stepwiseClassifier.pctWrong()) * 100)}%')

#report which features are the most important
print('The most important features(s) in order: ', [term[optimalFeature] for op
timalFeature in optimalFeatures])

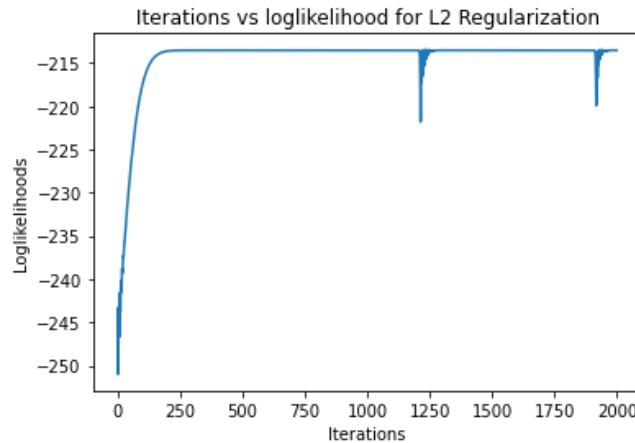
theta: [[-1.11244286]
 [ 0.          ]
 [ 0.13245139]
 [ 0.          ]
 [ 0.          ]
 [ 0.          ]
 [ 0.          ]
 [ 0.          ]]
% classified correct for stepwise: 70.0%
The most important features(s) in order: ['tobacco']
```



```
In [10]: #PART 3: L2 REGULARIZATION
l2Classifier = L2LogisticClassifier(None, None, copySubsetsFrom=binaryClassifier)
l2Classifier.validate()
l2Classifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for L2 Regularization')

correct_L2regularized = np.around((1 - l2Classifier.pctWrong()) * 100)
print(f'theta: {l2Classifier.theta()}\n% classified correct for L2 regularized: {np.around((1 - l2Classifier.pctWrong()) * 100)}%')

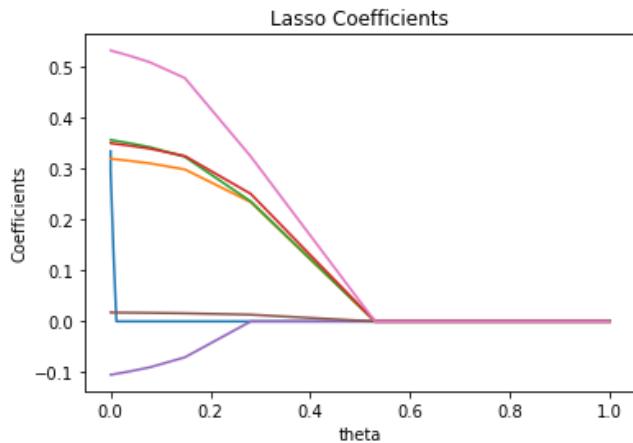
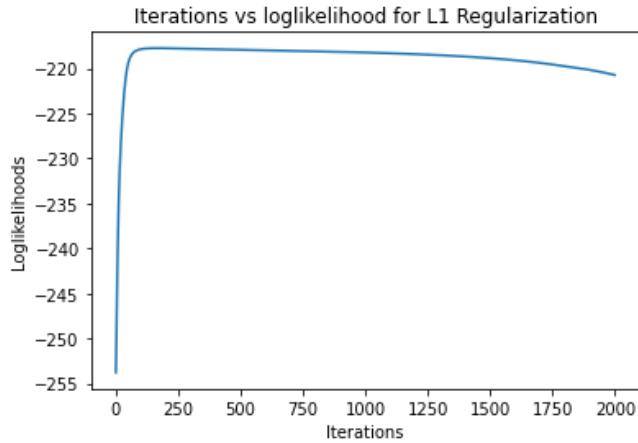
theta: [[-0.0042613 ]
 [ 0.10590091]
 [ 0.13889517]
 [ 0.13383285]
 [ 0.13248296]
 [ 0.03336894]
 [ 0.02454988]
 [ 0.20732039]]
% classified correct for L2 regularized: 77.0%
```



```
In [11]: # STRETCH GOAL 1: L1 REGULARIZATION
l1Classifier = L1LogisticClassifier(None, None, copySubsetsFrom=binaryClassifier)
bestC, coefficients = l1Classifier.validate()
l1Classifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for L1 Regularization')

cIterations = np.logspace(-8, 0, 30)
plt.figure()
plt.plot(cIterations, coefficients[:,1:])
plt.xlabel('theta')
plt.ylabel('Coefficients')
plt.title('Lasso Coefficients')
correct_L1regularized = np.around((1 - l1Classifier.pctWrong()) * 100)
print(f'theta: {l1Classifier.theta()}\n% classified correct for L1 regularized: {np.around((1 - l1Classifier.pctWrong()) * 100)}%')

theta: [[0.16911823]
 [0.]
 [0.23362606]
 [0.23488502]
 [0.24982351]
 [0.]
 [0.01322457]
 [0.32304502]]
% classified correct for L1 regularized: 79.0%
```



Additionally, report the % correct for all 3 models (unregularized, stepwise, and L2 regularized) in a table.

```
In [12]: correct = list([[correct_baseline, correct_unregularized, correct_stepwise, correct_L2regularized, correct_L1regularized]])
models = list(['Baseline', 'Unregularized', 'Stepwise', 'L2 regularization', 'L1 regularization'])

table = tabulate(correct, headers=models, tablefmt='pretty')
print('Table 1: % Correct for all the Models')
print(table)
```

Table 1: % Correct for all the Models

Baseline	Unregularized	Stepwise	L2 regularization	L1 regularization
60.0	72.0	70.0	77.0	79.0

Notes on most important features chosen:

- The most important features to include fluctuates each time the logistic regression classifier is run
- Although it fluctuates, there are a few terms that always seem to be the top such as tobacco, famhist, sbp (systolic blood pressure), and age
- The features that were indicated makes sense because tobacco is known to cause heart failure, especially those who had smoked in the past
- Family history also is an indicator if a person might have health issues in the future because some health problems are due to genetics
- The older you get, the more likely you might have heart failure

Other notes:

- The unregularized and stepwise model gave the highest accuracy for almost all of the times the data was classified
- Unregularized usually gave the highest accuracy
- L2 regularization is usually used to prevent overfitting; however, since this dataset was not that large, overfitting wasn't an issue which meant L2 regularization wasn't necessary
- L1 regularization gives an extremely high accuracy, which would make it the most optimal model
- Looking at the lasso plot, the features that are the most important would be tobacco, famhist, age, ldl
- The features from the lasso plot agree with the features from the stepwise

Binary Classification of the Breast Cancer Dataset

Repeat this analysis for a binary classification dataset of your choice from UCI or another repository.

From the dataset description:

Attribute.....Domain

1. Sample code number id number
2. Clump Thickness 1 - 10
3. Uniformity of Cell Size 1 - 10
4. Uniformity of Cell Shape 1 - 10
5. Marginal Adhesion 1 - 10
6. Single Epithelial Cell Size 1 - 10
7. Bare Nuclei 1 - 10
8. Bland Chromatin 1 - 10
9. Normal Nucleoli 1 - 10
10. Mitoses 1 - 10
11. Class: (2 for benign, 4 for malignant)

```
In [25]: # Dataset Description: http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.names
# Read through data and create dataset
bcDataset = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data', names=["ID", "ClumpThickness", "Uniformity_CellSize", "Uniformity_CellShape", "MarginalAdhesion", "SingleEpithelialCellSize", "BareNuclei", "BlandChromatin", "NormalNucleoli", "Mitosis", "Class"])
bcDataset.pop('ID')

# There are ? for missing data, so we drop the rows that have them
bcDataset = bcDataset.apply(partial(pd.to_numeric, errors='coerce'))
bcDataset = bcDataset.dropna(axis = 0)

# Change labels to have 0 for benign and 1 for malignant
bcDataset['Class'] = (bcDataset['Class'] == 4)*1

# Creates a graph like figure 4.12
sb.pairplot(bcDataset, hue = 'Class', palette="hls", height = 3)

# list the features
term = list(bcDataset.columns.values[:-1])

# Generate Features matrix : NxP
bcDatasetX = bcDataset.drop(['Class'], axis=1).to_numpy()
# Generate Label matrix : Nx1
bcDatasety = bcDataset.loc[:, 'Class'].to_numpy().reshape(-1, 1)

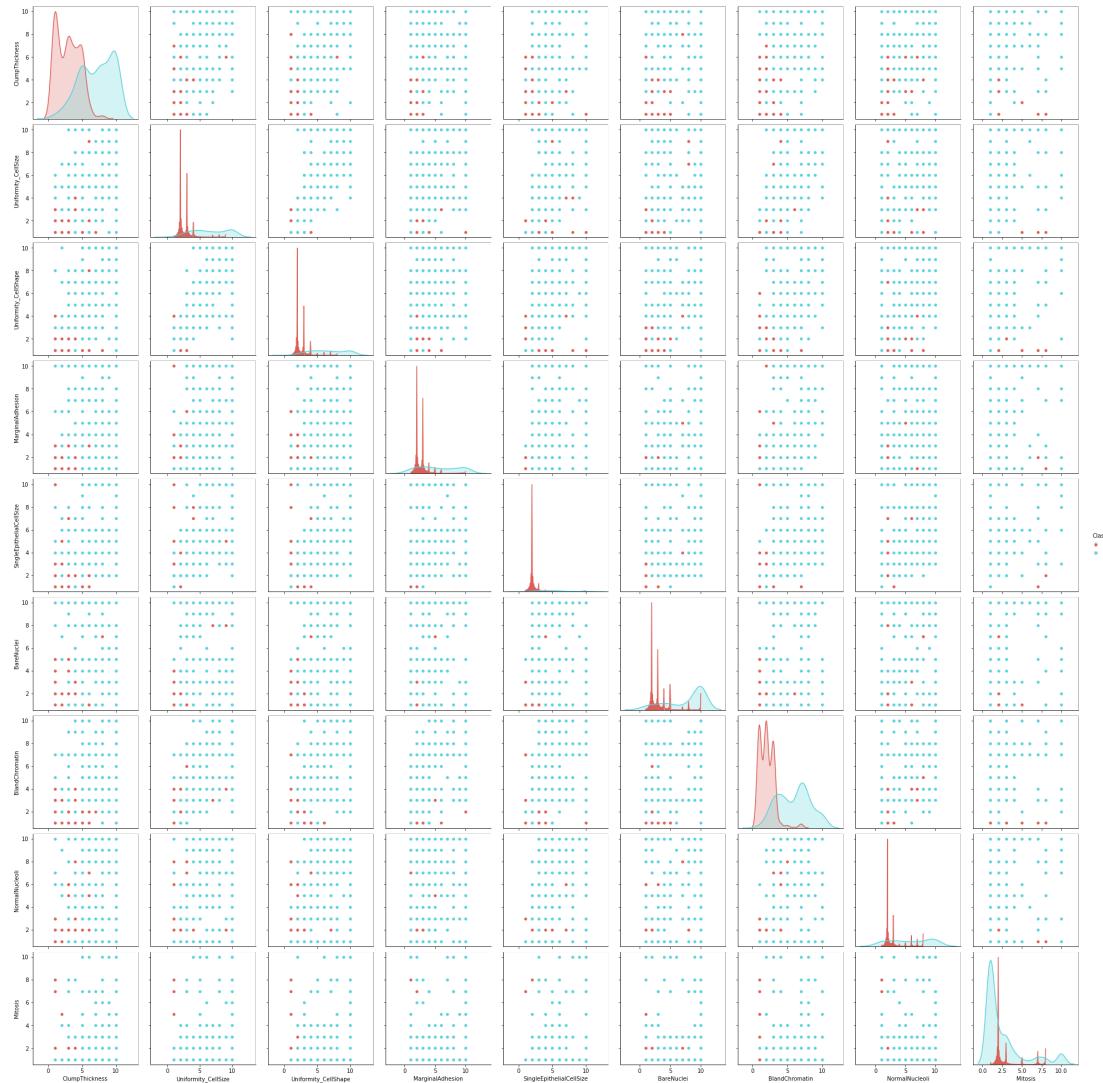
# PART 1: RECREATE TABLE 4.
binaryClassifier = BinaryLogisticClassifier(bcDatasetX, bcDatasety)
binaryClassifier.train()
binaryClassifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for Unregularization')
correct_unregularized = np.around((1 - binaryClassifier.pctWrong()) * 100)
print(f'theta: {binaryClassifier.theta()}\n% classified correct for unregularized: {np.around((1 - binaryClassifier.pctWrong()) * 100)}%')
correct_baseline = np.around((1 - binaryClassifier.baselinePctWrong()) * 100)
```

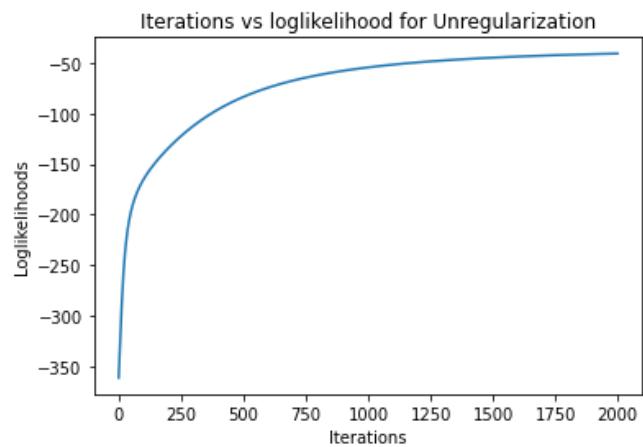
```

Length of dataset: 683
Length of training: 546
Length of validation: 68
Length of test: 69
theta: [[-7.38720214]
[ 0.26532443]
[ 0.13456965]
[ 0.40163983]
[ 0.18669206]
[ 0.02447402]
[ 0.28032114]
[ 0.36474628]
[ 0.1738657 ]
[ 0.1841025 ]]

% classified correct for unregularized: 93.0%

```



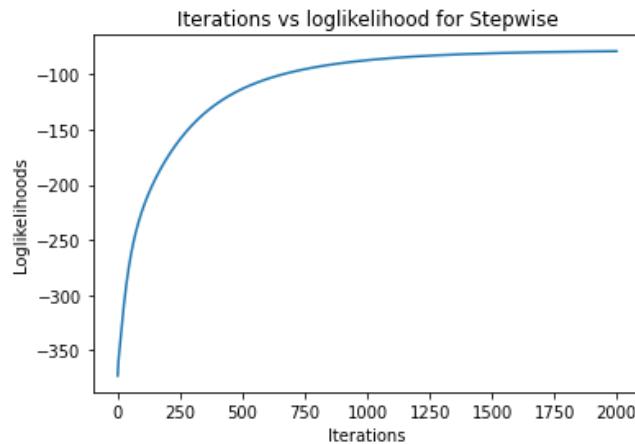


```
In [26]: # PART 2: STEPWISE
stepwiseClassifier = StepwiseLogisticClassifier(None, None, copySubsetsFrom=bin
aryClassifier)
_, optimalFeatures = stepwiseClassifier.validate()
stepwiseClassifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for Stepwise')

correct_stepwise = np.around((1 - stepwiseClassifier.pctWrong()) * 100)
print(f'theta: {stepwiseClassifier.theta()}\n% classified correct for stepwise:
{np.around((1 - stepwiseClassifier.pctWrong()) * 100)}%')

#report which features are the most important
print('The most important features(s) in order: ', [term[optimalFeature] for op
timalFeature in optimalFeatures])

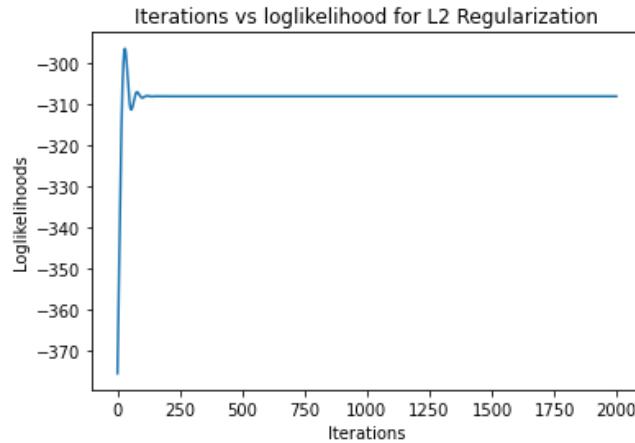
theta: [[-5.92093473]
 [ 0.          ]
 [ 1.19591208]
 [ 0.          ]
 [ 0.          ]
 [ 0.28331059]
 [ 0.          ]
 [ 0.          ]
 [ 0.37411502]
 [ 0.          ]]
% classified correct for stepwise: 91.0%
The most important features(s) in order: ['SingleEpithelialCellSize', 'Uniform
ity_CellSize', 'NormalNucleoli']
```



```
In [27]: #PART 3: L2 REGULARIZATION
l2Classifier = L2LogisticClassifier(None, None, copySubsetsFrom=binaryClassifier)
l2Classifier.validate()
l2Classifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for L2 Regularization')

correct_L2regularized = np.around((1 - l2Classifier.pctWrong()) * 100)
print(f'theta: {l2Classifier.theta()}\n% classified correct for L2 regularized:
{np.around((1 - l2Classifier.pctWrong()) * 100)}%')

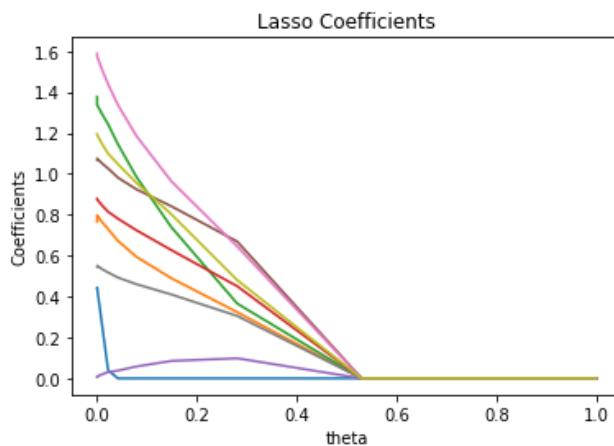
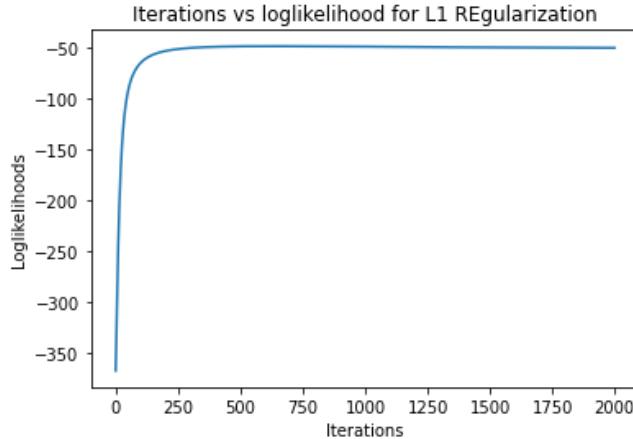
theta: [[0.01741748]
[0.04328471]
[0.04999025]
[0.05081505]
[0.0437788 ]
[0.04164145]
[0.05106646]
[0.04696949]
[0.04385446]
[0.0249921 ]]
% classified correct for L2 regularized: 93.0%
```



```
In [28]: # STRETCH GOAL 1: L1 REGULARIZATION
l1Classifier = L1LogisticClassifier(None, None, copySubsetsFrom=binaryClassifier)
bestC, coefficients = l1Classifier.validate()
l1Classifier.plotLoglikelihood()
plt.title('Iterations vs loglikelihood for L1 Regularization')

cIterations = np.logspace(-8, 0, 30)
plt.figure()
plt.plot(cIterations, coefficients[:,1:])
plt.xlabel('theta')
plt.ylabel('Coefficients')
plt.title('Lasso Coefficients')
correct_L1regularized = np.around((1 - l1Classifier.pctWrong()) * 100)
print(f'theta: {l1Classifier.theta()}\n% classified correct for L1 regularized: {np.around((1 - l1Classifier.pctWrong()) * 100)}%')

theta: [[1.21049955]
 [0.        ]
 [0.48962964]
 [0.73918101]
 [0.62710808]
 [0.08504367]
 [0.84160619]
 [0.96372163]
 [0.41017053]
 [0.80232168]]
% classified correct for L1 regularized: 93.0%
```



Additionally, report the % correct for all 3 models (unregularized, stepwise, and L2 regularized) in a table.

```
In [29]: correct = list([[correct_baseline, correct_unregularized, correct_stepwise, correct_L2regularized, correct_L1regularized]])
models = list(['Baseline', 'Unregularized', 'Stepwise', 'L2 regularization', 'L1 regularization'])

table = tabulate(correct, headers=models, tablefmt='pretty')
print('Table 1: % Correct for all the Models')
print(table)
```

Table 1: % Correct for all the Models

Baseline	Unregularized	Stepwise	L2 regularization	L1 regularization
65.0	93.0	91.0	93.0	93.0

Notes on selected features:

- The feature that was chosen most was Uniformity Cell size. Almost all of the times the data was processed, this feature was chose
- For breast cancer, the growth of cells is usually the biggest indicator if the person has cancer or not (spreading of the diseased cells within their body and could be seen in a form of a tumor)
- Marginal Adhesion also was chosen as one of the most important features; however, this occurred very rarely

Other notes:

- Both unregularized and L2 regularization produced about the same amount of accuracy
- These two models gave the highest accuracy, but just from looking at the accuracy, we cannot chose which model is the most optimal
- Stepwise model is the least optimal
- L1 regularization gives us the highest accuracy, which would make it the most optimal model
- Looking at the lasso plot, the most important features are Mitosis, Bland Chromatin, Bare Nuclei
- The features from the lasso plot don't match with the stepwise

Multinomial Classification of the Iris dataset

Features include sepal length and width, petal length and width, and the target is one of three types of iris flowers.

The classification works very well, typically classifying around 98% correct on the training subset and ~100% correct on the test subset, which indicates that the dataset is highly linearly-separable.

```
In [22]: # iris dataset for multiclass (3-class regression)
irisDataset = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-datasets/iris/iris.data')

# one-hot encode labels
irisDatasety = np.vstack((
    (irisDataset.iloc[:,4] == 'Iris-setosa').to_numpy(dtype=np.float32),
    (irisDataset.iloc[:,4] == 'Iris-versicolor').to_numpy(dtype=np.float32),
    (irisDataset.iloc[:,4] == 'Iris-virginica').to_numpy(dtype=np.float32))).T

# feature matrix
irisDatasetX = irisDataset.iloc[:, :4].to_numpy()

irisClassifier = TrinaryLogisticClassifier(irisDatasetX, irisDatasety)
irisClassifier.trinaryClassificationTrain()
print(f'% correct on iris dataset: {(1 - irisClassifier.pctWrong())*100}')
pd.DataFrame(data=np.hstack((irisClassifier._theta1, irisClassifier._theta2)),
              columns=['theta_1', 'theta_2'],
              index=['bias', 'sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'])
```

Length of dataset: 149
Length of training: 119
Length of validation: 15
Length of test: 15
% correct on iris dataset: 100.0

Out[22]:

	theta_1	theta_2
bias	2.726092	6.315390
sepal length (cm)	6.251550	6.358187
sepal width (cm)	11.632527	5.278686
petal length (cm)	-15.197718	-9.101620
petal width (cm)	-7.129879	-9.910898

```
In [23]: print('Comparison of prediction to actual labels')
pd.DataFrame(data=np.hstack((np.around(irisClassifier.h(irisClassifier._subsets['test']['X'])), 1), irisClassifier._subsets['test']['y'])),
              columns=['PC1', 'PC2', 'PC3', 'AC1', 'AC2', 'AC3'])
```

Comparison of prediction to actual labels

Out[23]:

	PC1	PC2	PC3	AC1	AC2	AC3
0	0.0	0.0	1.0	0.0	0.0	1.0
1	0.0	0.0	1.0	0.0	0.0	1.0
2	0.0	1.0	0.0	0.0	1.0	0.0
3	0.0	1.0	0.0	0.0	1.0	0.0
4	1.0	0.0	0.0	1.0	0.0	0.0
5	0.0	1.0	0.0	0.0	1.0	0.0
6	0.0	1.0	0.0	0.0	1.0	0.0
7	0.0	1.0	0.0	0.0	1.0	0.0
8	0.0	0.0	1.0	0.0	0.0	1.0
9	1.0	0.0	0.0	1.0	0.0	0.0
10	0.0	0.0	1.0	0.0	0.0	1.0
11	0.0	0.7	0.3	0.0	1.0	0.0
12	1.0	0.0	0.0	1.0	0.0	0.0
13	0.0	1.0	0.0	0.0	1.0	0.0
14	1.0	0.0	0.0	1.0	0.0	0.0

Here, "PC1" means "predicted class 1", "AC1" means "actual class 1". The predictions match the actual labels very well.

Frequentist Machine Learning

Assignment 3

Jonathan Lam, Tiffany Yu, Harris Pasqualeti

Re-implement the example in section 7.10.2 using any simple, out of the box classifier (like K nearest neighbors from sci-kit). Reproduce the results for the incorrect and correct way of doing cross-validation.

In []:

```
#setting up

import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import RepeatedKFold

# Consider a scenario with N = 50 samples in two equal-sized classes, and
# p = 5000 quantitative predictors (standard Gaussian) that are independent
# of the class labels.

# generate 50 random samples of N(0,1) data w/ 5000 features
X = np.random.normal(0,1,[50, 5000])
# The true (test) error rate of any classifier is 50%
Y = np.concatenate([np.zeros(25), np.ones(25)])
np.random.shuffle(Y)

#INCORRECT
CV_correct = []

# Screen the predictors: find a subset of "good" predictors that show fairly
# strong (univariate) correlation with the class labels
# preprocessing.MinMaxScaler needed because non-negative values needed for Sele
ctKBest
X_new = preprocessing.MinMaxScaler().fit_transform(X)
X_new = SelectKBest(chi2, k=100).fit_transform(X_new, Y)

# Using just this subset of predictors, build a multivariate classifier.
neigh = KNeighborsClassifier(n_neighbors=1)

# Use cross-validation to estimate the unknown tuning parameters and to estimat
e
# the prediction error of the final model.
# source: https://scikit-learn.org/stable/modules/generated/sklearn.model_selec
tion.RepeatedKFold.html
rkf = RepeatedKFold(n_splits=5, n_repeats=50)
for train_index, test_index in rkf.split(X_new):
    X_train, X_test = X_new[train_index], X_new[test_index]
    Y_train, Y_test = Y[train_index], Y[test_index]
    neigh.fit(X_train, Y_train)
    CV_correct.append(1-neigh.score(X_test, Y_test))

print("Average CV Error Rate:", np.around(np.array(CV_correct).mean(), 3))

"""

What has happened? The problem is that the predictors have an unfair
advantage, as they were chosen in step (1) on the basis of all of the samples.
Leaving samples out after the variables have been selected does not cor-
rectly mimic the application of the classifier to a completely independent
test set, since these predictors "have already seen" the left out samples.

"""

# CORRECT
CV_correct = []
kbest = SelectKBest(chi2, k=100)

# divide the samples into K CV folds at random
rkf = RepeatedKFold(n_splits=5, n_repeats=50)
for train_index, test_index in rkf.split(X):
```

```
Average CV Error Rate: 0.038
Average CV Error Rate: 0.547
```

The average CV error rate from the incorrect way using cross-validation is much lower than the average CV error rate using the correct way using cross-validation. The correct way has a higher average error because it has not seen the test samples, so it cannot use them as predictors.

In []:

```
#Setting up
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import xgboost as xgb

# sklearn utility functions for training
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import RandomizedSearchCV
from sklearn import metrics

class XGBoostClassifier:

    def __init__(self, X, y, feature_names):

        self.feature_names = feature_names

        # split into test and training datasets
        self.X_train, self.X_test, self.y_train, self.y_test = \
            train_test_split(X, y, \
                            test_size=0.2, \
                            random_state=np.random.randint(0, 100), \
                            shuffle=True)

        # standardize features
        self.X_train = StandardScaler().fit_transform(self.X_train)
        self.X_test = StandardScaler().fit_transform(self.X_test)

        # turn into xgboost dmatrices
        self.train_dm = xgb.DMatrix(pd.DataFrame(data=self.X_train, \
                                                   columns=feature_names), \
                                   label=self.y_train)
        self.test_dm = xgb.DMatrix(pd.DataFrame(data=self.X_test, \
                                               columns=feature_names), \
                                  label=self.y_test)

    # train the model, and show evaluation statistics on the test dataset
    def train_eval(self, num_rounds, max_depth):
        eval_list = [(self.train_dm, 'train'), (self.test_dm, 'eval')]
        xgb_classifier = xgb.XGBRegressor()

        parms = {
            'nthread': [4],
            'objective': ['reg:squarederror'], # textbook uses Huber loss;
                                            # colab doesn't have newest xgboost
                                            # so we just use reg:squarederror
            'learning_rate': [0.05],
            'max_depth': [max_depth],
            'n_estimators': [num_rounds],
            'eval_metric': ['mae'], # mean absolute error
        }
        # grid search to find the optimal parameters
        xgb_grid = GridSearchCV(xgb_classifier,
                               parms,
                               cv=2,
                               n_jobs=5,
                               verbose=True)

        xgb_grid.fit(self.X_train, self.y_train)
```

```

print(f'Best score from grid search: {xgb_grid.best_score_}')
print(f'Best parameters from grid search: {xgb_grid.best_params_}')
self.progress = {}
self.xgb_classifier = xgb.train(xgb_grid.best_params_,
                                 self.train_dm,
                                 num_boost_round=100,
                                 evals=eval_list,
                                 evals_result=self.progress,
                                 early_stopping_rounds=100,
                                 verbose_eval=False)

# plot average absolute error vs. iterations; assumes model has
# already been trained
def gb_mae(self):
    return (self.progress['eval']['mae'])

def random_forest(self, n_trees, m):
    self.trees = list(range(1, n_trees))
    self._mae = np.zeros(len(self.trees))
    for i in range(n_trees - 1):
        tree = self.trees[i]
        regressor = RandomForestRegressor(n_estimators=tree, max_features=m)
        regressor.fit(self.X_train, self.y_train)
        y_pred = regressor.predict(self.X_test)
        self._mae[i] = metrics.mean_absolute_error(self.y_test, y_pred)
        print(i)
def rf_mae(self):
    return self._mae
def num_trees(self):
    return self.trees

# california housing dataset from sklearn
from sklearn.datasets import fetch_california_housing
cal_housing = fetch_california_housing()
X = pd.DataFrame(cal_housing.data, columns=cal_housing.feature_names)
y = cal_housing.target

n_trees = 275

# create classifier
classifier = XGBoostClassifier(X, y, cal_housing.feature_names)

```

In []:

```

# train classifier
classifier.train_eval(n_trees, 4)
GBM_depth_4 = classifier.gb_mae()

plt.figure()
plt.plot(GBM_depth_4)

```

Fitting 2 folds for each of 1 candidates, totalling 2 fits

```

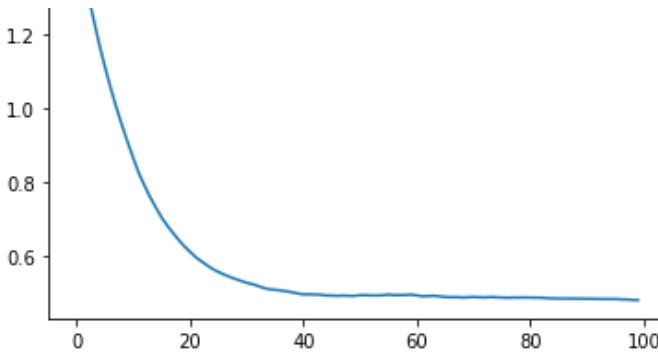
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done    2 out of    2 | elapsed:      3.4s remaining:      0.0s
[Parallel(n_jobs=5)]: Done    2 out of    2 | elapsed:      3.4s finished

```

Best score from grid search: 0.8049024718829725
 Best parameters from grid search: 0.8049024718829725

Out[]:

[<matplotlib.lines.Line2D at 0x7f66b1b2a080>]



In []:

```
classifier.train_eval(n_trees, 6)
GBM_depth_6 = classifier.gbm_mae()

plt.figure()
plt.plot(GBM_depth_6)
```

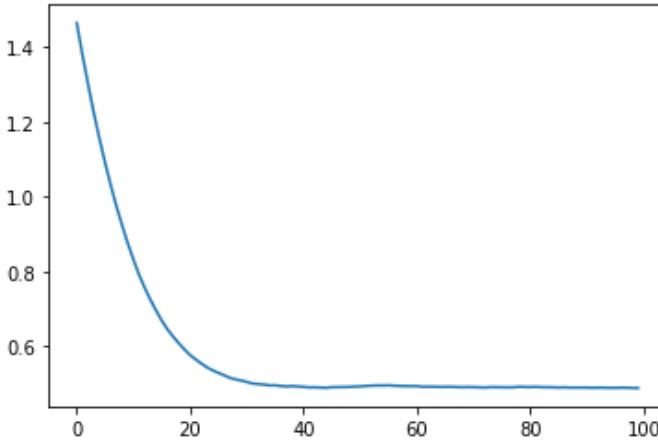
Fitting 2 folds for each of 1 candidates, totalling 2 fits

```
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done    2 out of  2 | elapsed:      5.4s remaining:      0.0s
[Parallel(n_jobs=5)]: Done    2 out of  2 | elapsed:      5.4s finished
```

Best score from grid search: 0.8192566595418167
Best parameters from grid search: 0.8192566595418167

Out[]:

```
[<matplotlib.lines.Line2D at 0x7f66b1aab1d0>]
```



In []:

```
classifier.random_forest(n_trees, 2)
RF_m_2 = classifier.rf_mae()
```

In []:

```
classifier.random_forest(n_trees, 6)
RF_m_6 = classifier.rf_mae()
```

In []:

```
plt.figure()
plt.plot(GBM_depth_4)
plt.plot(GBM_depth_6)
plt.plot(RF_m_2)
plt.plot(RF_m_6)
plt.legend(['GBM depth=4', 'GBM depth=6', 'RF m=2', 'RF m=6'])
plt.title('California Housing Data')
```

```
plt.ylim([0, 1])
plt.xlim([0, n_trees])
plt.xlabel('Number of Trees')
plt.show()
```

Frequentist Machine Learning

Assignment 5

Jonathan Lam, Tiffany Yu, Harris Pasqualeti

Defining the Models

We use the `xgboost` package for gradient-boosted trees and the `sklearn` `RandomForestRegressor` for random forests.

In [1]:

```
!pip install xgboost==1.2.0

# setting up
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import xgboost as xgb

# sklearn utility functions for training
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
from sklearn import metrics

plt.rcParams['figure.figsize'] = [12, 12]

# define regressor models for both XGB and RF ensemble models
class Regressor:

    def __init__(self, X_train, X_test, y_train, y_test, feature_names, n_trees):
        self.feature_names = feature_names
        self.trees = list(range(1, n_trees))
        self._mae = []
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test

    def train_eval(self, depth, type=None):
        self._mae = []
        for tree in self.trees:
            if type == 'xgb':
                self.regressor = xgb.XGBRegressor(objective="reg:pseudohubererror",
                                                   eta=0.05,
                                                   max_depth=depth,
                                                   n_estimators=tree,
                                                   n_jobs=2)
            elif type == 'rf':
                self.regressor = RandomForestRegressor(n_estimators=tree,
                                                       max_features=depth)
            self.regressor.fit(self.X_train, self.y_train)
            y_pred = self.regressor.predict(self.X_test)
            self._mae.append(metrics.mean_absolute_error(self.y_test, y_pred))

    def mae(self):
```

```

    return self._mae

# plot relative importance plot; assumes model has already been trained
def plot_importance(self, type=None):
    features = self.feature_names
    importances = self.regressor.feature_importances_
    indices = np.argsort(importances)
    if type == 'xgb':
        plt.title('Feature Importances for Gradient Trees')
    elif type == 'rf':
        plt.title('Feature Importances for Random Forest')
    plt.barh(range(len(indices)), importances[indices], color='b', align='center')
    plt.yticks(range(len(indices)), [features[i] for i in indices])
    plt.xlabel('Relative Importance')
    plt.show()

```

Collecting xgboost==1.2.0
 Downloading https://files.pythonhosted.org/packages/f6/5c/1133b5b8f4f2fa740ff27abdd35b8e79ce6e1f8d6480a07e9bc1cdafea2/xgboost-1.2.0-py3-none-manylinux2010_x86_64.whl (148.9MB)
 |██████████| 148.9MB 77kB/s
 Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from xgboost==1.2.0) (1.18.5)
 Requirement already satisfied: scipy in /usr/local/lib/python3.6/dist-packages (from xgboost==1.2.0) (1.4.1)
 Installing collected packages: xgboost
 Found existing installation: xgboost 0.90
 Uninstalling xgboost-0.90:
 Successfully uninstalled xgboost-0.90
 Successfully installed xgboost-1.2.0

Running the Models

California housing Dataset

**Replicate figure 15.3 comparing random forests and gradient boosted trees. You can use whatever package you wish, you don't have to use xgboost if you'd rather keep everything in sci-kit learn.
 Compare the feature importance found by random forests and gradient boosted trees.**

In [20]:

```

# california housing dataset from sklearn
from sklearn.datasets import fetch_california_housing
cal_housing = fetch_california_housing()
X = pd.DataFrame(cal_housing.data, columns=cal_housing.feature_names)
y = cal_housing.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# create regressor
n_trees = 275
cal_reg = Regressor(X_train, X_test, y_train, y_test, cal_housing.feature_names, n_trees)

# train classifier
cal_reg.train_eval(4, 'xgb')
GBM_depth_4 = cal_reg.mae()

cal_reg.train_eval(6, 'xgb')
GBM_depth_6 = cal_reg.mae()

cal_reg.plot_importance('xgb')

```

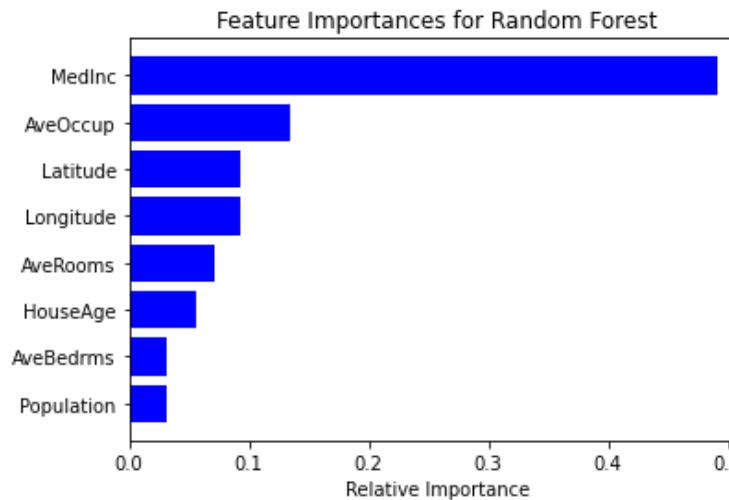
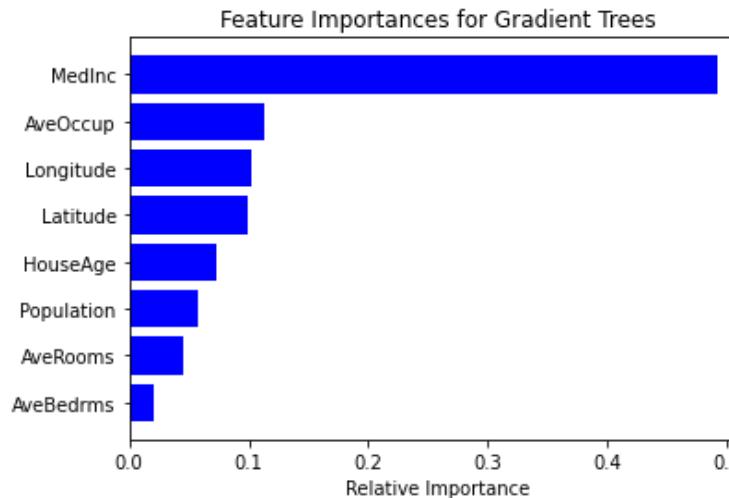
```

cal_reg.train_eval(2, 'rf')
RF_m_2 = cal_reg.mae()

cal_reg.train_eval(6, 'rf')
RF_m_6 = cal_reg.mae()

cal_reg.plot_importance('rf')

```



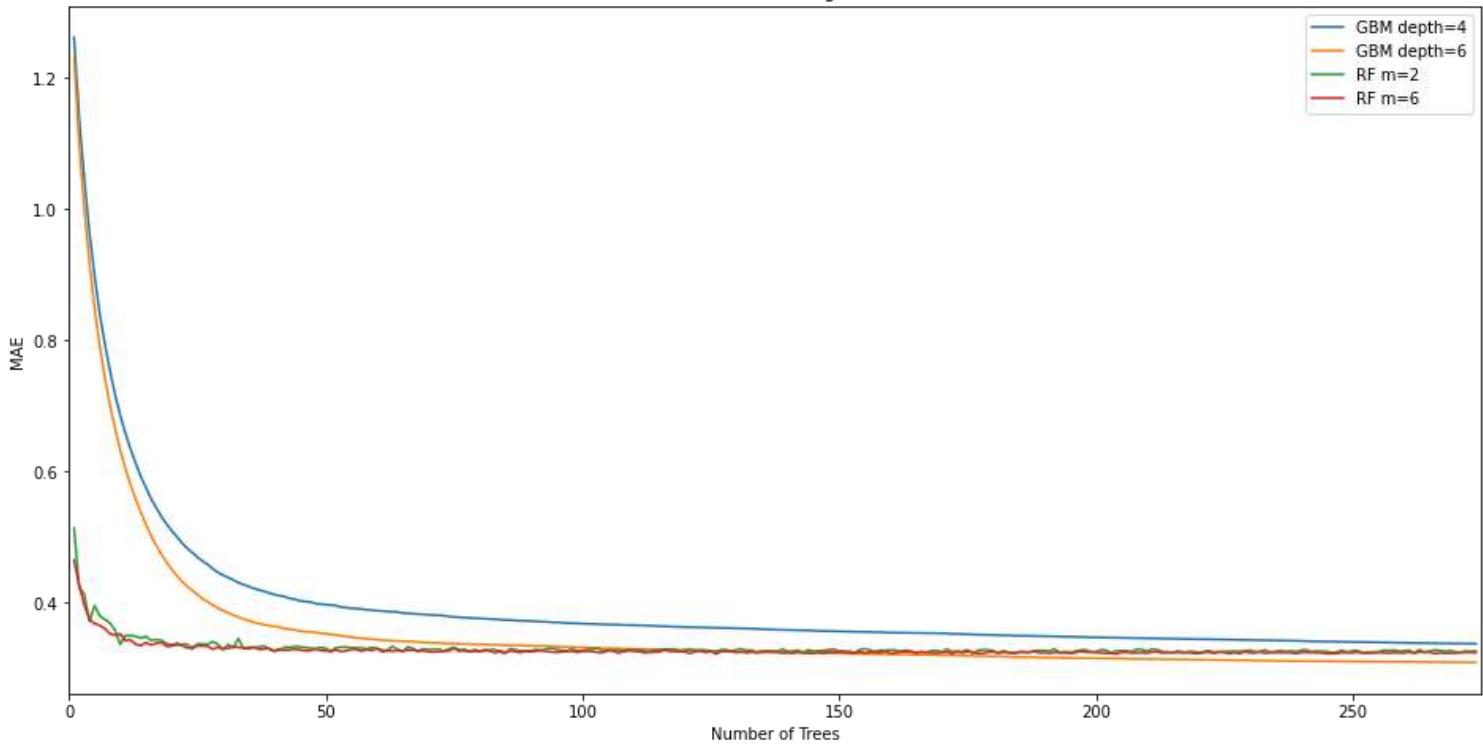
The graphs above show the relative variable importance for each of the eight predictor variables. The feature with the largest bar is the feature that is the most relevant predictor. For both of the feature importance graphs for the California dataset, median income has the greatest importance, which would make the most sense because the amount of income a person has dictates how much they can afford and is one of the main deciding factors of buying a house. The next three features with the greatest importance are average household members, latitude, and longitude, which is the same for both gradient boost and random forest. The rest of the features are not in the same order and have the least relevance, which means they are less influential. Each feature is plotted w.r.t. its F-score, which is a metric that sums up how many times each feature is split on.

In [21]:

```

plt.figure(figsize=(16, 8))
plt.plot(cal_reg.trees, GBM_depth_4)
plt.plot(cal_reg.trees, GBM_depth_6)
plt.plot(cal_reg.trees, RF_m_2)
plt.plot(cal_reg.trees, RF_m_6)
plt.legend(['GBM depth=4', 'GBM depth=6', 'RF m=2', 'RF m=6'])
plt.title('California Housing Data')
plt.xlim([0, n_trees])
plt.xlabel('Number of Trees')
plt.ylabel('MAE')
plt.show()

```



The above graph shows the average absolute error given the number of trees or n estimators for the California housing dataset. For both gradient boosting trees, we can see that as the number of trees increase the average error decreases. We see that the random forest models converge very quickly, and then remain roughly constant after that; their MAE is also a little noisy even after convergence. For the gradient boosted models, they converge slower but the depth=6 model eventually beats the MAE of the random forests, and it looks as though the depth=4 model will also beat the MAE of the random forests given a few more trees (n_estimators). The distinction between the two gradient boosted models' performances is clear, but the random forests are hardly distinguishable for this dataset.

Boston Housing Dataset

Select another dataset and repeat the analysis. Pick a dataset we have not yet studied in class.

In [18]:

```
# boston housing dataset from sklearn
from sklearn.datasets import load_boston
bos_housing = load_boston()
X = pd.DataFrame(bos_housing.data, columns=bos_housing.feature_names)
y = bos_housing.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# create classifier
n_trees = 275
bos_reg = Regressor(X_train, X_test, y_train, y_test, bos_housing.feature_names, n_trees)

# train classifier
bos_reg.train_eval(4, 'xgb')
GBM_depth_4 = bos_reg.mae()

bos_reg.train_eval(6, 'xgb')
GBM_depth_6 = bos_reg.mae()

bos_reg.plot_importance('xgb')

bos_reg.train_eval(2, 'rf')
RF_m_2 = bos_reg.mae()

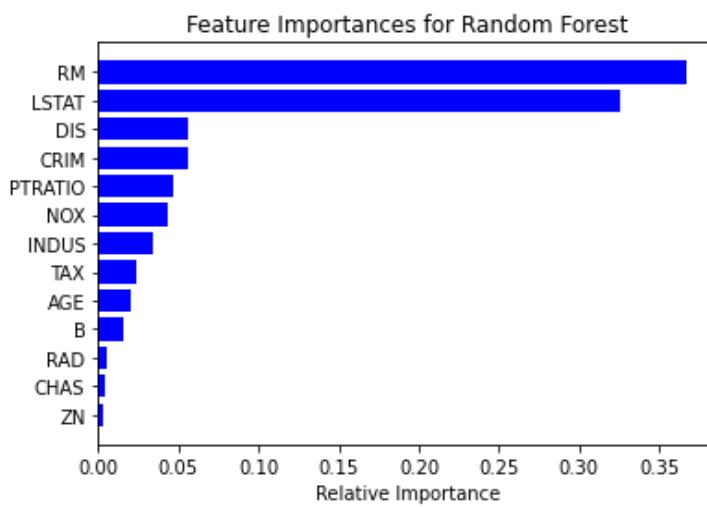
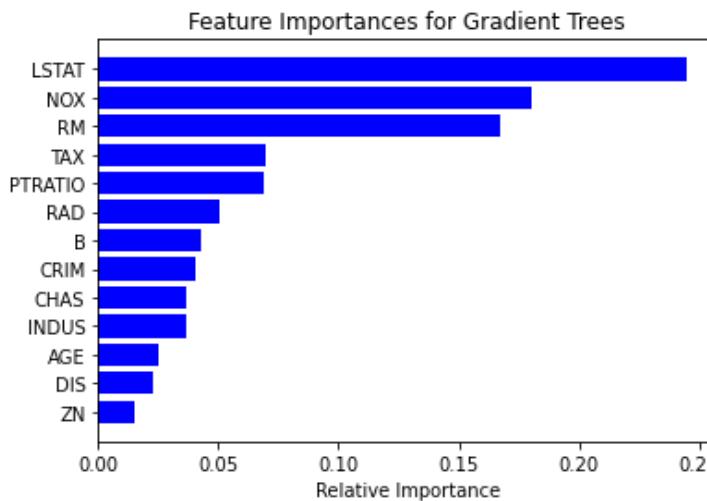
bos_reg.train_eval(6, 'rf')
```

```

RF_m_6 = bos_reg.mae()

bos_reg.plot_importance('rf')

```



The 13 features of the boston dataset include:

1. CRIM - per capita crime rate by town
2. ZN - proportion of residential land zoned for lots over 25,000 sq. ft
3. INDUS - porportion of non-retail business acres per town
4. CHAS - Charles River dummy variable
5. NOX - nitric oxides concentration (parts per 10 million)
6. RM - average number of rooms per dwelling
7. AGE -proportion of owner-occupied units built prior to 1940
8. DIS - weighted distances to five Boston employment centres
9. RAD - index of accessibility to radial highways
10. TAX - full-value property-tax rate per \$10,000
11. PTRATIO - pupil-teacher ratio by town
12. B - $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT - % lower status of the population

The features from the gradient boosting trees and random forest feature importance graphs differ slightly. From the gradient boosting tree graph, LSTAT has the greatest relevancy which means the lower percentage of the status of the population has the greatest relavancy of who buys houses in Boston. In addition NOX and RM have the next greatest relavancy, which means has some relavancy. In the random forest feature importance graph, RM and LSTAT have the greatest relavancy, which is similar to the other feature importance graph. The rest of the features match in how much relavancy they have to the boston dataset; however, the random forest feature importance graph has more features with close to 0 relative importance, which implies that those features don't have any impact for the datset.

In [19]:

```
plt.figure(figsize=(16, 8))
plt.plot(bos_reg.trees, GBM_depth_4)
plt.plot(bos_reg.trees, GBM_depth_6)
plt.plot(bos_reg.trees, RF_m_2)
plt.plot(bos_reg.trees, RF_m_6)
plt.legend(['GBM depth=4', 'GBM depth=6', 'RF m=2', 'RF m=6'])
plt.title('Boston Housing Data')
plt.xlim([0, n_trees])
plt.xlabel('Number of Trees')
plt.ylabel('MAE')
plt.show()
```



The above graph shows the average absolute error given the number of trees or n estimators for the Boston housing dataset. As before, the gradient-boosted trees do slightly better than random forests, but takes longer to reach a minimum. For random forest, the average error, although a little bit noisy, leveled off to a constant number fairly quickly, which shows that we would not need to train it for a large number of trees. Here the distinction between the two random forest models is more clear than in the California dataset example: the higher m (maximum number of features per tree) has a noticeably lower MAE. On the other hand, the two gradient-boosted tree models converge to nearly the same value, although we see that the higher depth converges to that value faster.

ECE 475 Project 6: Market Basket Analysis

Tiffany Yu, Jonathan Lam, Harris Paspuleti

The goal of this project is to use market basket analysis to draw interesting inferences (association rules) about some dataset.

In []:

```
# mlxtend has a priori algorithm implementation
!pip install mlxtend

import pandas as pd
import numpy as np
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt

from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
```

Dataset

FIFA 18 Player Statistics

This dataset contains the attributes for every player that is registered in the latest edition of FIFA 18 database; it can be used for soccer/videogame analysis as it contains attributes such as skill moves, overall, potential, position, etc. While none of us know much about soccer, we wanted to see if there were any interesting associations we could learn from the data even without knowing much about the sport.

Content

- Every player featuring in FIFA 18
- 70+ attributes
- Player and Flag Images
- Playing Position Data
- Attributes based on actual data of the latest EA's FIFA 18 game
- Attributes include on all player style statistics like Dribbling, Aggression, - GK Skills etc.
- Player personal data like Nationality, Photo, Club, Age, Wage, Salary etc.

Data Source

The data is scraped from the website <https://sofifa.com> by extracting the Player personal data and Player Ids and then the playing and style statistics.

In []:

```
# grab the data
raw_data = 'https://raw.githubusercontent.com/4m4n5/fifa18-all-player-statistics/master/2019/data.csv'
dataframe = pd.read_csv(raw_data, sep=',', header='infer', error_bad_lines=False)
```

Preprocessing

Preprocessing Step 1: Drop unusable columns and rows

Some columns were extraneous, included data that we didn't know how to interpret, or included data that was unique to a player. For example, there was a column for the player's index, the player photo, and some fields like "Real Face" that we weren't familiar with. Additionally, there were many rows with acronyms (e.g., "LS", "ST", etc.) with values that we weren't sure how to interpret.

Some rows also had missing data; we dropped this using `pd::dropna()`.

The dataframe after this initial preprocessing step is shown below.

In []:

```
# preprocessing the data by removing unnecessary columns
dataframe = dataframe.drop(columns=[
    'Unnamed: 0', 'ID', 'Photo', 'Flag', 'Club Logo', 'Loaned From', 'Real Face', 'LS',
    'ST', 'RS', 'LW', 'LF', 'CF', 'RF', 'RW', 'LAM', 'CAM', 'RAM', 'LM', 'LCM', 'CM', 'RCM',
    'RM', 'LWB', 'LDM', 'LB', 'LCB', 'CB', 'RCB', 'RB', 'CDM', 'RDM', 'RWB'
]).dropna()
dataframe
```

Out []:

	Name	Age	Nationality	Overall	Potential	Club	Value	Wage	Special	Preferred Foot	International Reputation	Weighted
0	L. Messi	31	Argentina	94	94	FC Barcelona	€110.5M	€565K	2202	Left	5.0	
1	Cristiano Ronaldo	33	Portugal	94	94	Juventus	€77M	€405K	2228	Right	5.0	
2	Neymar Jr	26	Brazil	92	93	Paris Saint-Germain	€118.5M	€290K	2143	Right	5.0	
3	De Gea	27	Spain	91	93	Manchester United	€72M	€260K	1471	Right	4.0	
4	K. De Bruyne	27	Belgium	91	92	Manchester City	€102M	€355K	2281	Right	4.0	
...
18202	J. Lundstram	19	England	47	65	Crewe Alexandra	€60K	€1K	1307	Right	1.0	
18203	N. Christoffersson	19	Sweden	47	63	Trelleborgs FF	€60K	€1K	1098	Right	1.0	
18204	B. Worman	16	England	47	67	Cambridge United	€60K	€1K	1189	Right	1.0	
18205	D. Walker-Rice	17	England	47	66	Tranmere Rovers	€60K	€1K	1228	Right	1.0	
18206	G. Nugent	16	England	46	66	Tranmere Rovers	€60K	€1K	1321	Right	1.0	

16643 rows × 56 columns

Preprocessing Step 2: Determining bins

Before binning features, we plotted histograms of some of the quantitative fields so that we could see what the distributions look like.

Since this is a videogame, much of the numerical ratings were presented as numbers out of 100, so we decided that binning most fields into quartiles was good enough for our purposes. (Using larger bins was also problematic because of the amount of time it took to run the algorithm with more items.)

```

In [ ]:

# list of numerical fields that can be binned
to_bin = ['Age', 'Overall', 'Potential', 'Value', 'Wage', 'Release Clause',
          'Jersey Number', 'FKAccuracy', 'LongPassing', 'BallControl', 'Acceleration',
          'SprintSpeed', 'Agility', 'Reactions', 'Balance', 'ShotPower', 'Jumping',
          'Stamina', 'Strength', 'LongShots', 'Aggression', 'Interceptions',
          'Positioning', 'Vision', 'Penalties', 'Composure', 'Marking',
          'StandingTackle', 'SlidingTackle', 'GKDiving', 'GKHandling', 'GKKicking',
          'GKPositioning', 'GKReflexes']
]

# histograms; see: https://realpython.com/python-histograms/
for col in to_bin:
    # An "interface" to matplotlib.axes.Axes.hist() method
    n, bins, patches = plt.hist(x=dataframe.loc[:, col], bins='auto', color='#0504aa',
                                 alpha=0.7, rwidth=0.85)

    plt.title(col)
    plt.grid(axis='y', alpha=0.75)
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    plt.text(23, 45, r'$\mu=15, b=3$')
    maxfreq = n.max()
    # Set a clean upper y-axis limit.
    plt.ylim(ymax=np.ceil(maxfreq / 10) * 10 if maxfreq % 10 else maxfreq + 10)
    plt.figure()

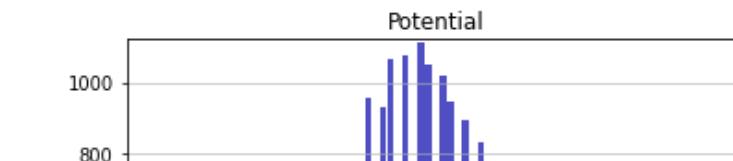
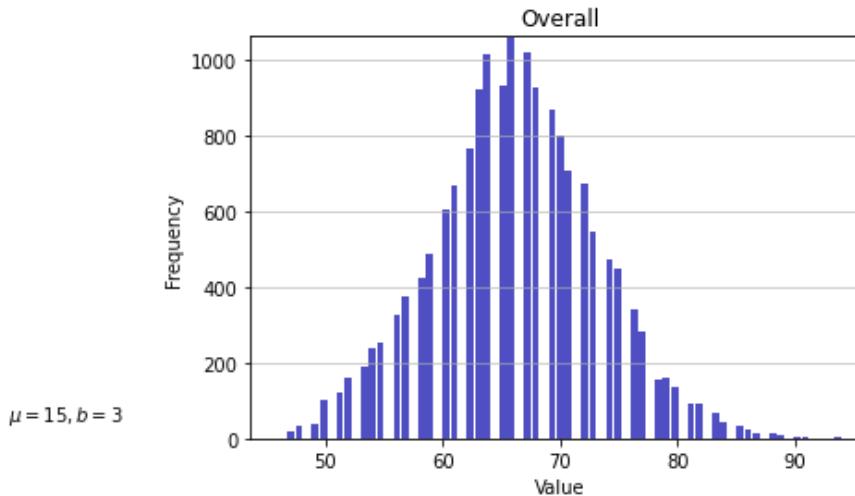
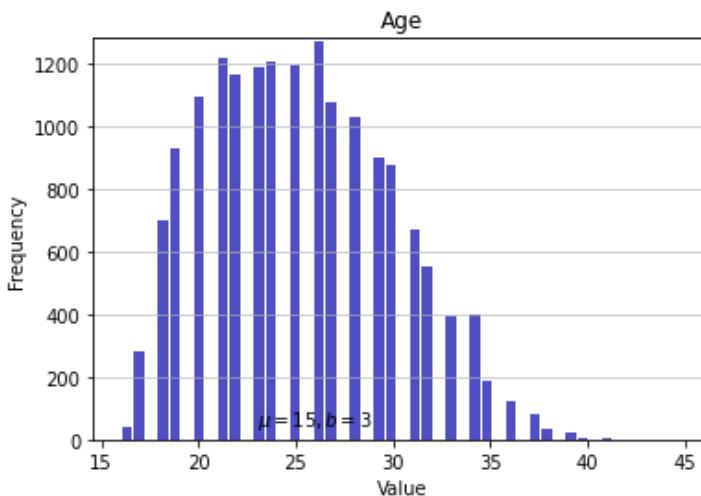
```

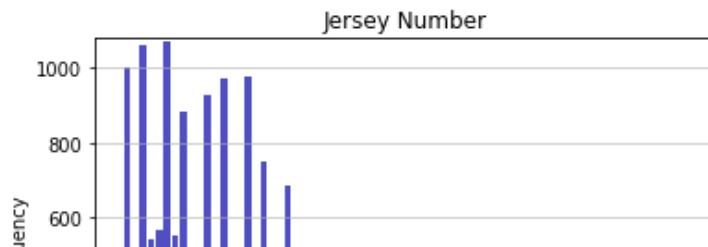
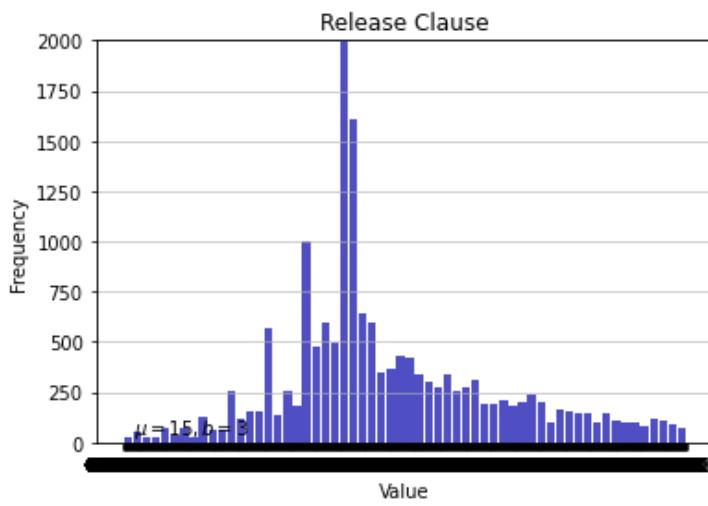
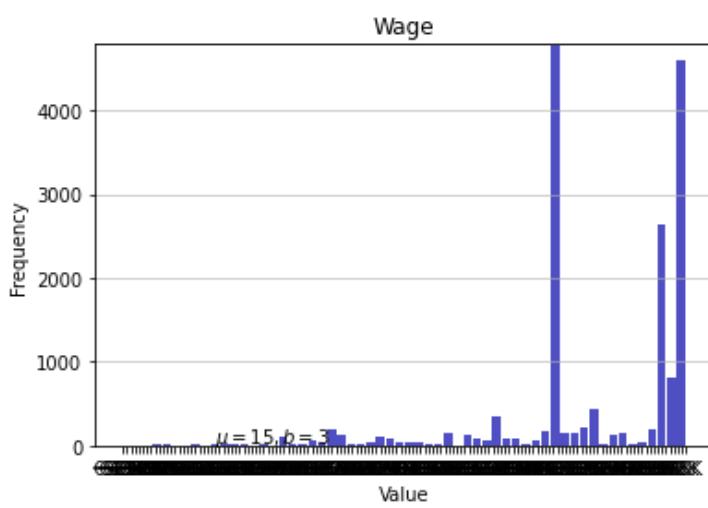
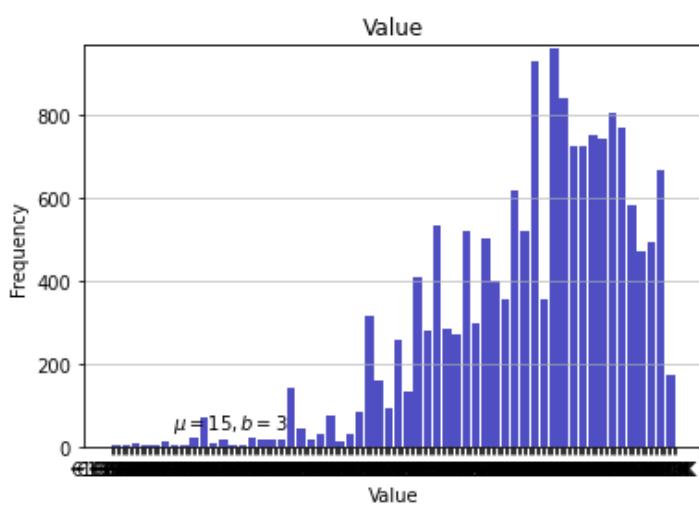
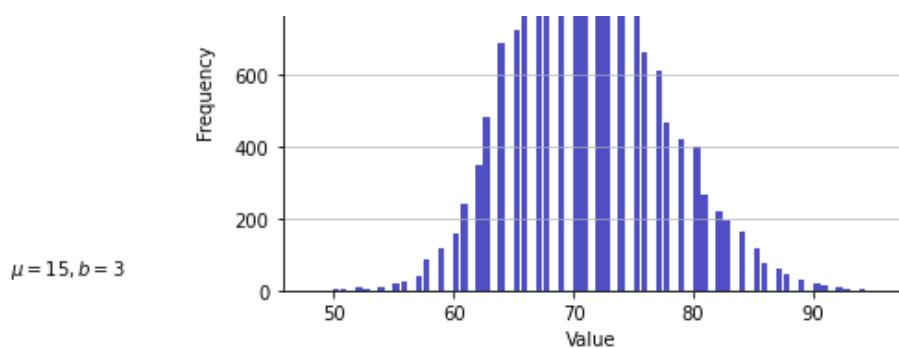
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:15: RuntimeWarning: More than 20 figures have been opened. Figures created through the pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly closed and may consume too much memory. (To control this warning, see the rcParam `figure.max_open_warning`).

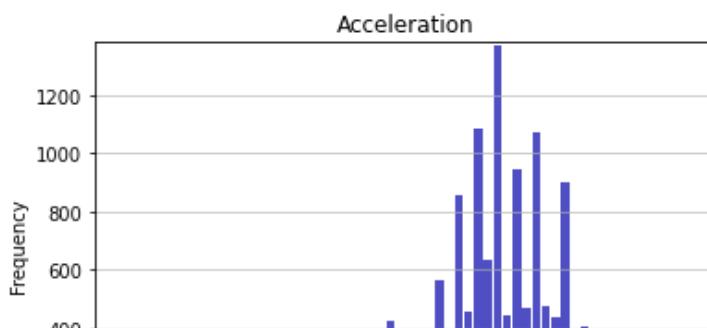
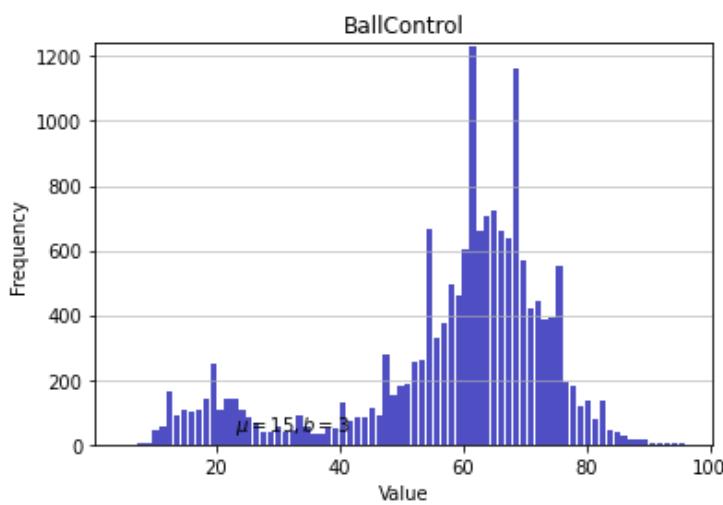
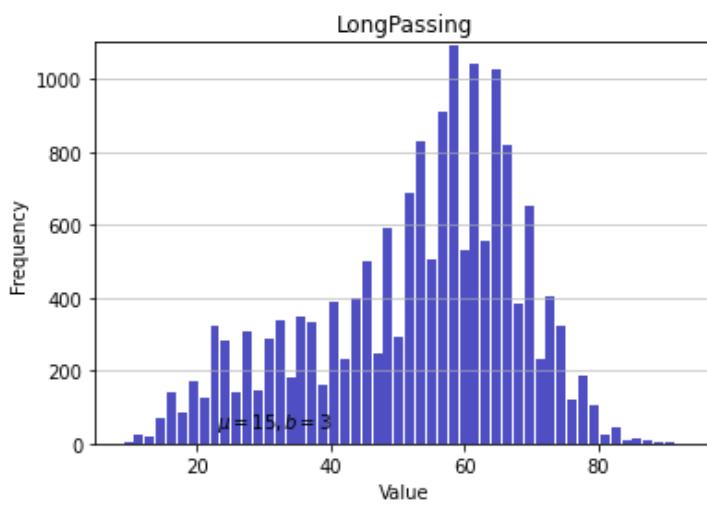
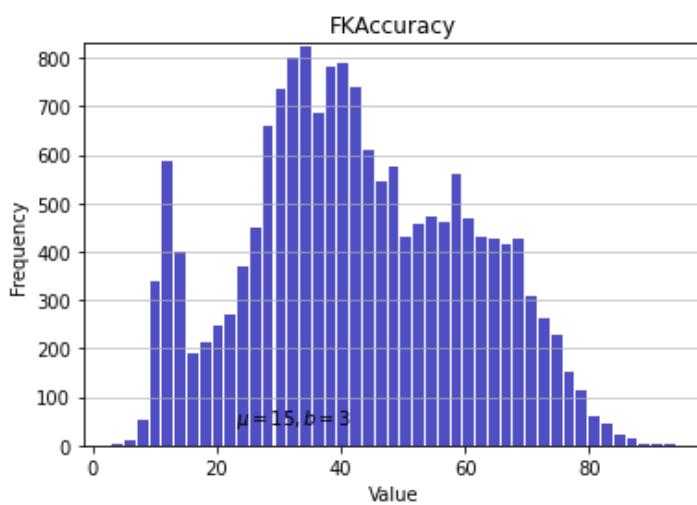
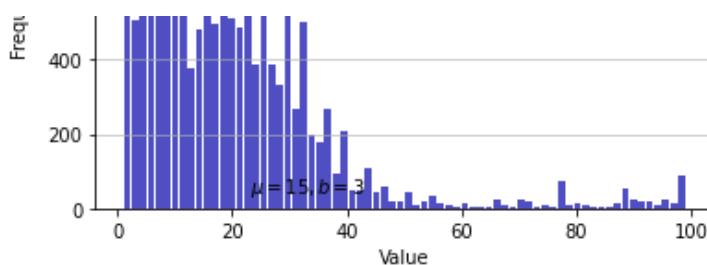
```

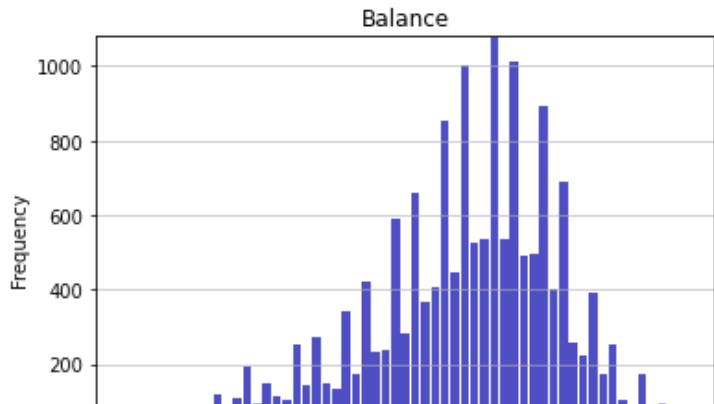
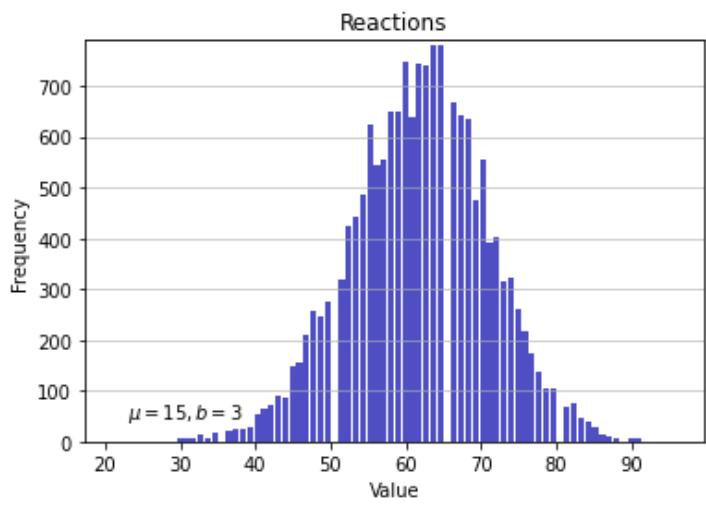
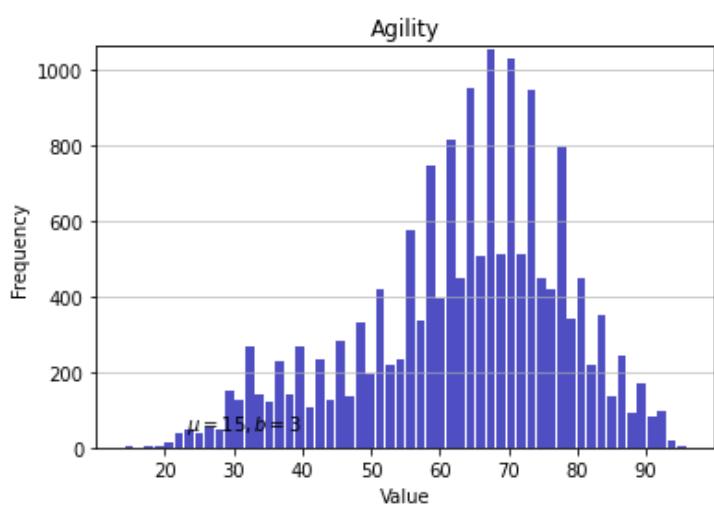
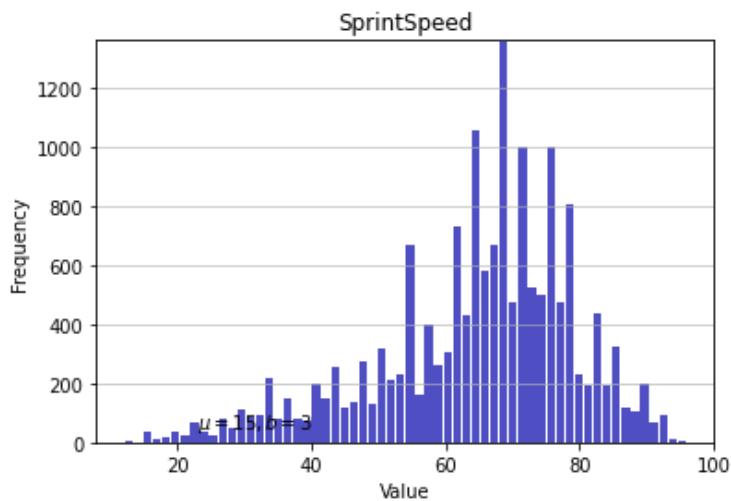
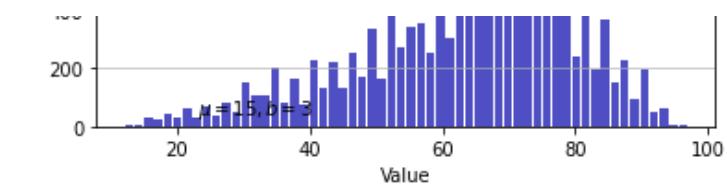
from ipykernel import kernelapp as app

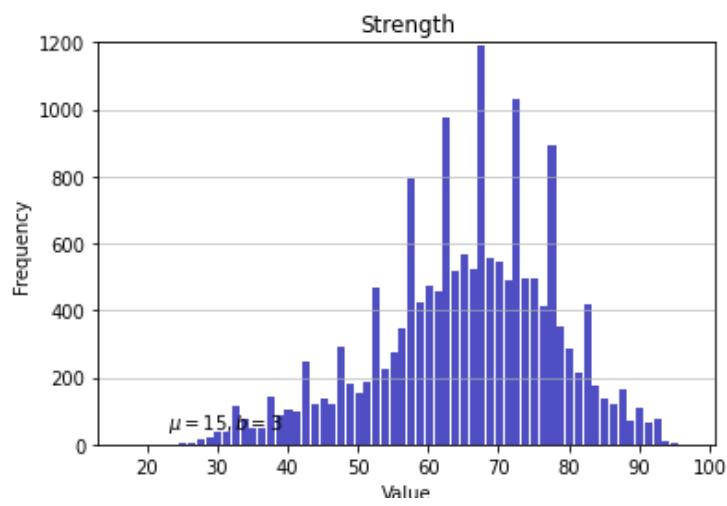
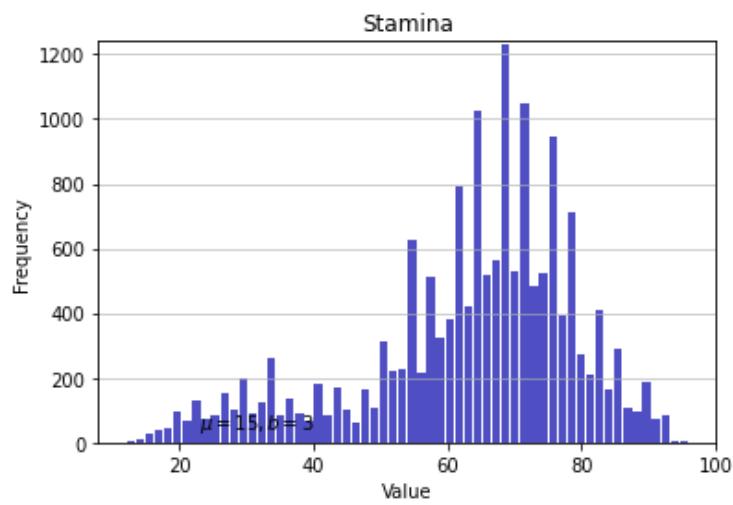
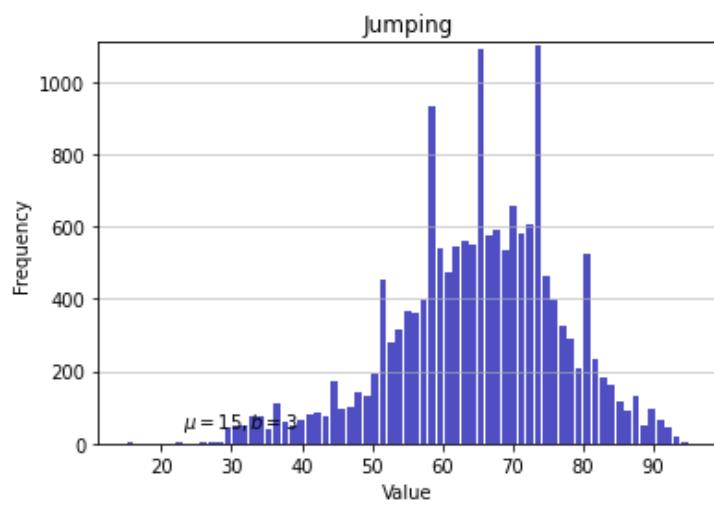
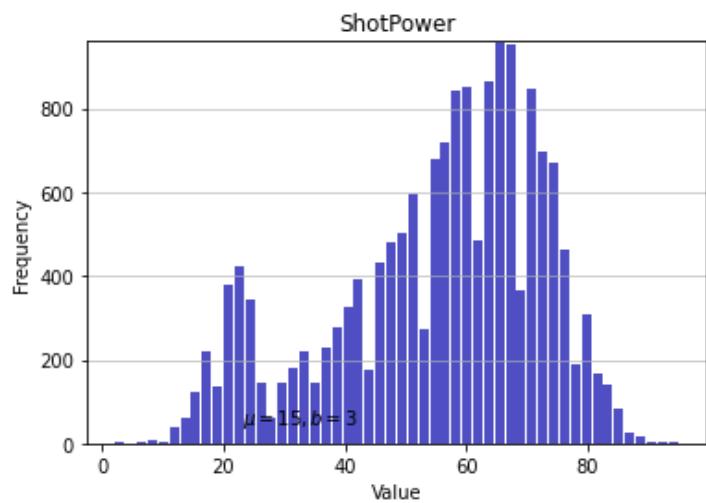
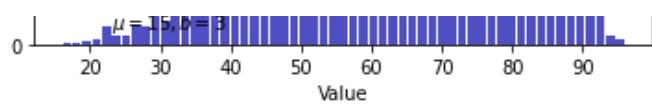
```



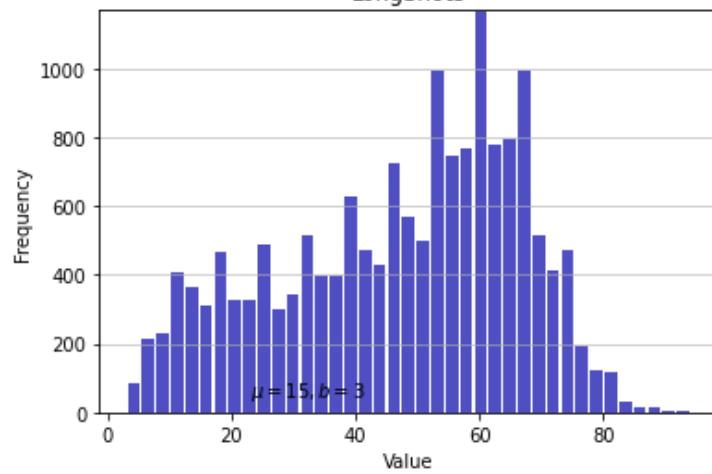




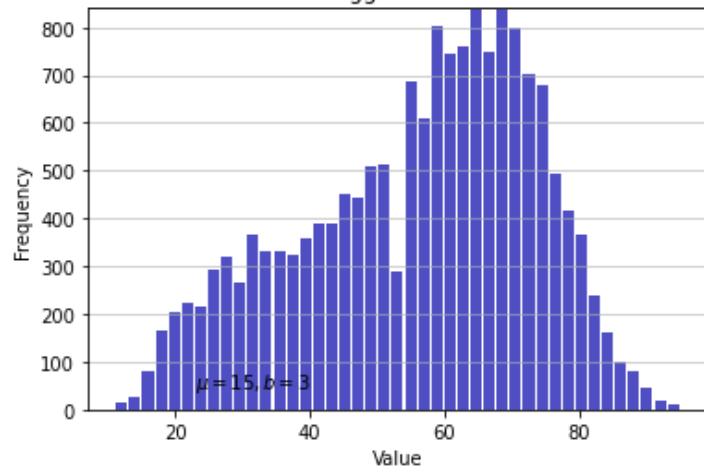




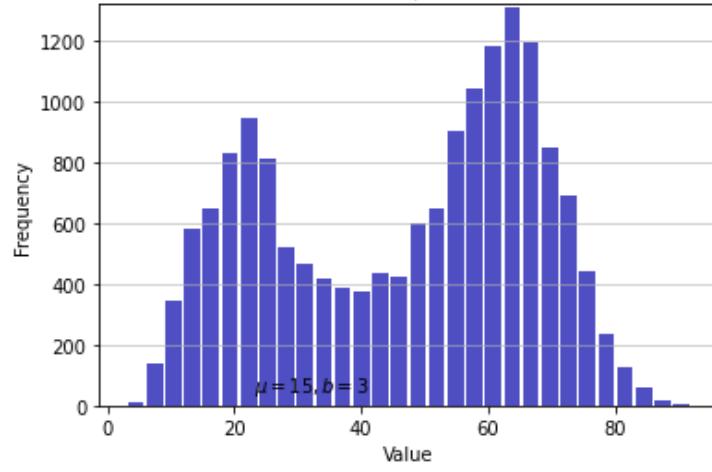
LongShots



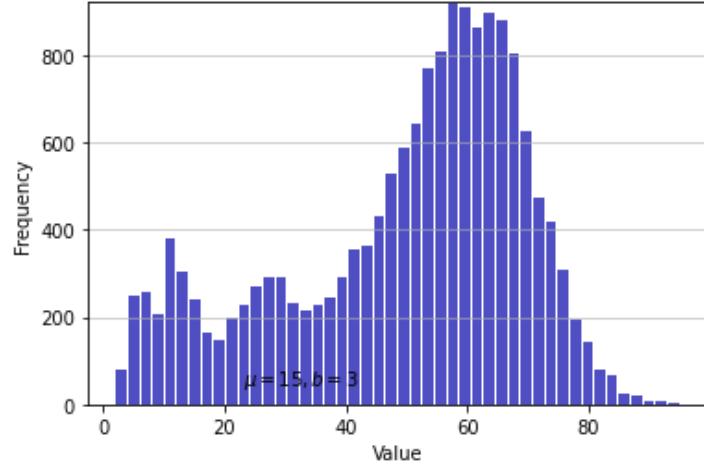
Aggression



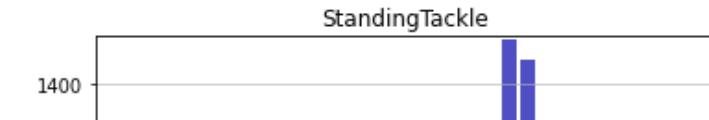
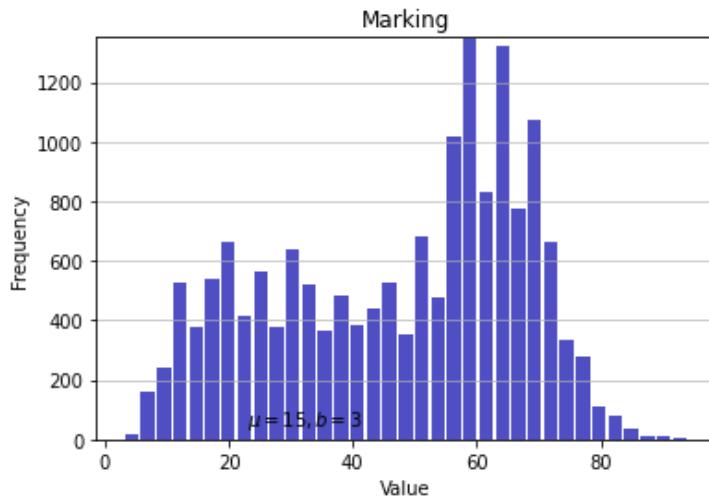
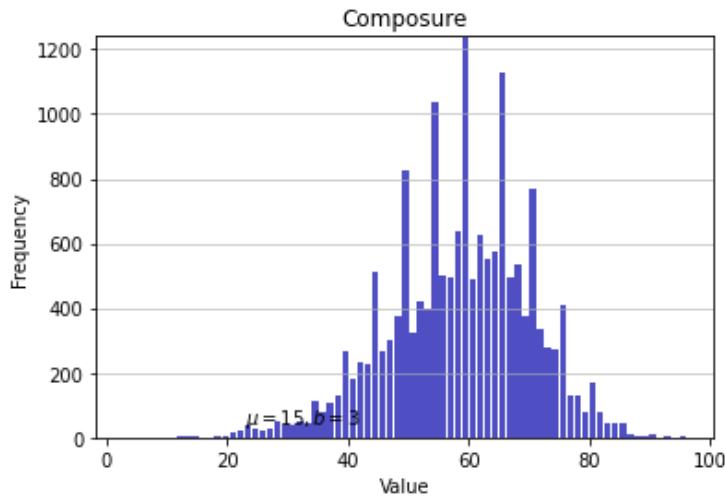
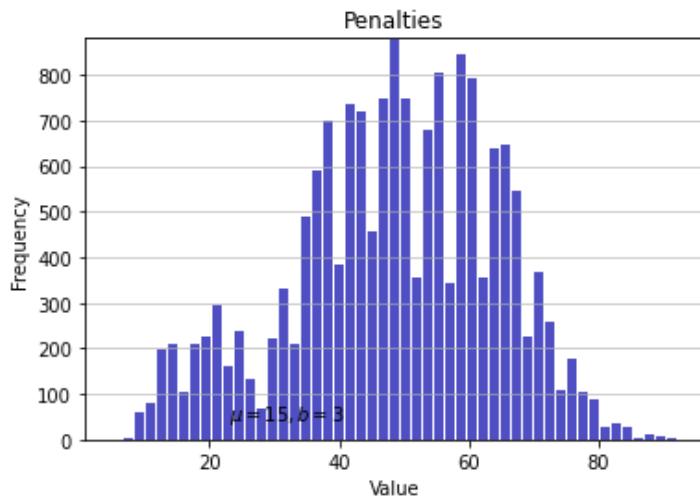
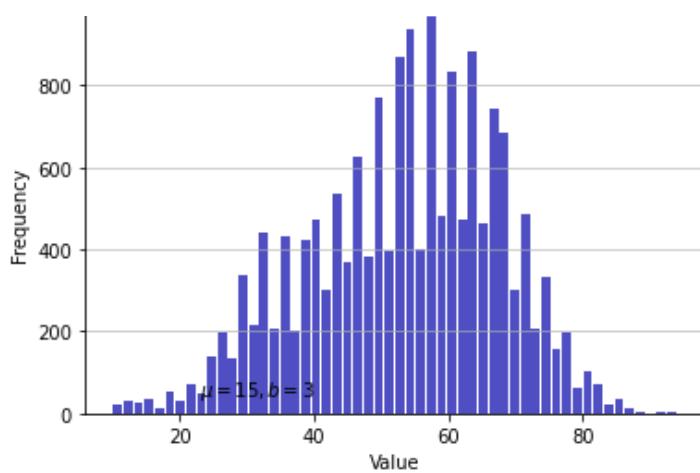
Interceptions

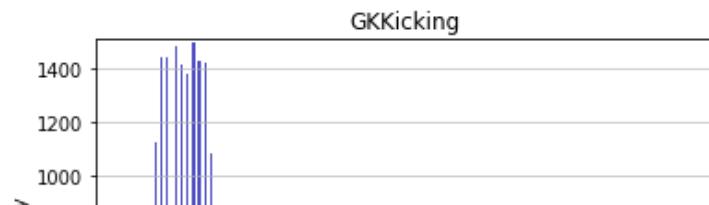
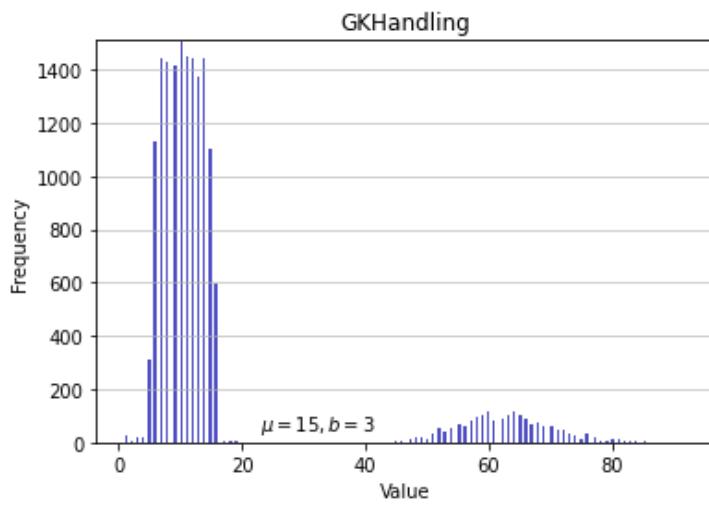
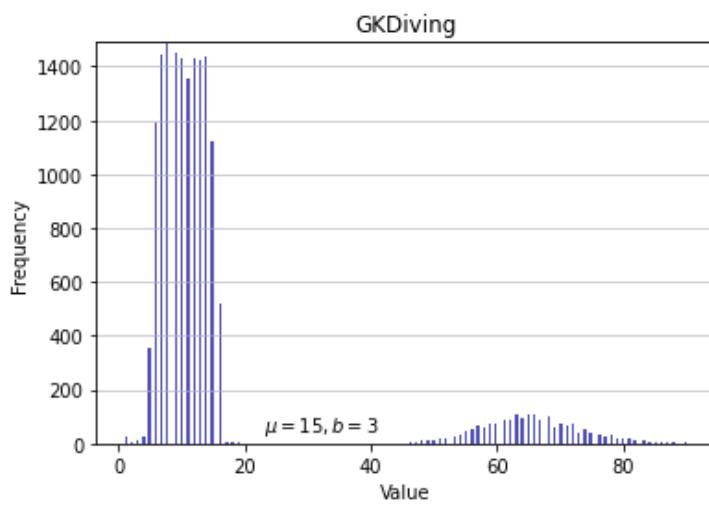
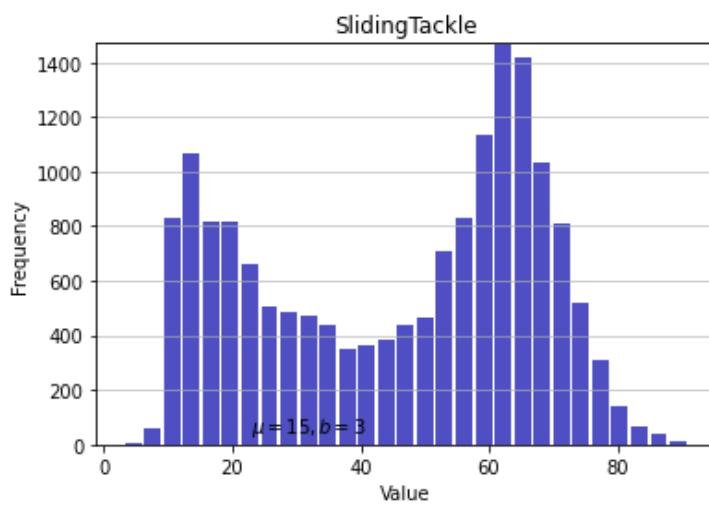
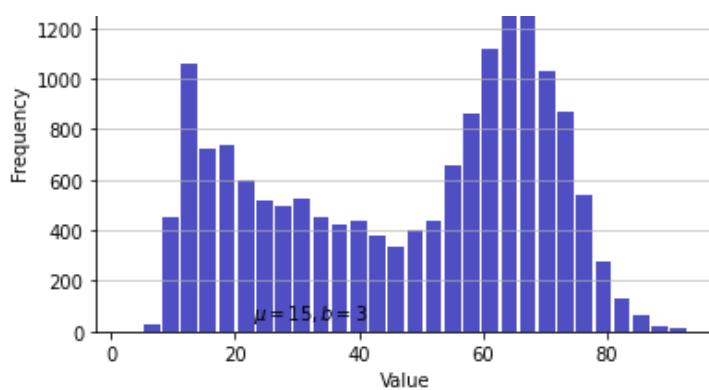


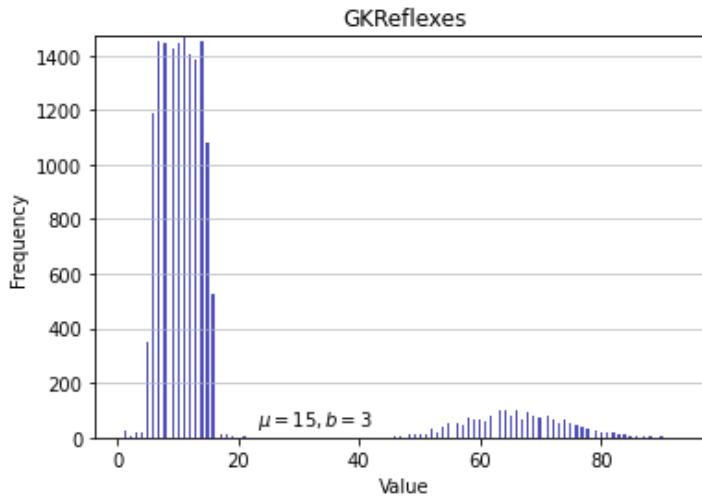
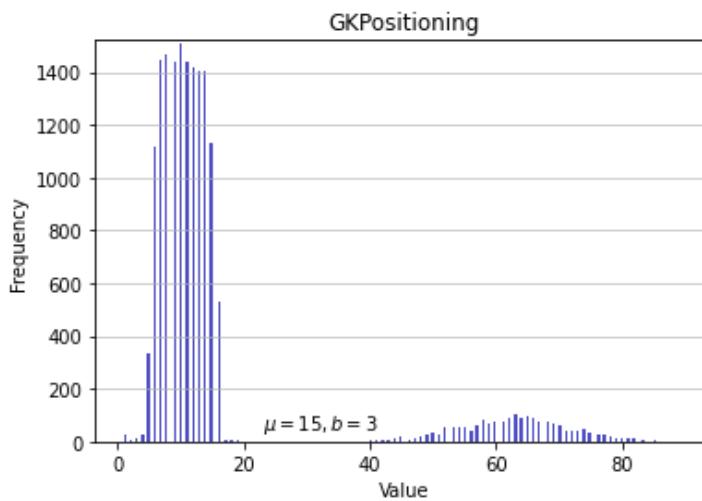
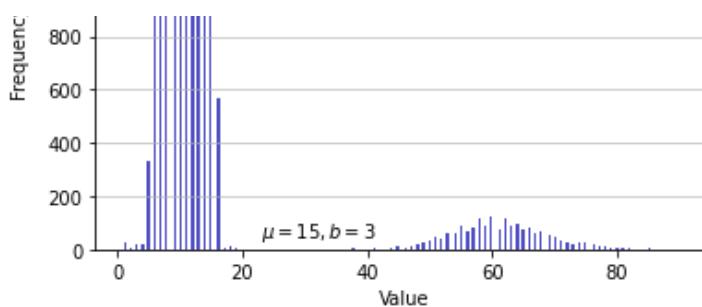
Positioning



Vision







<Figure size 432x288 with 0 Axes>

Preprocessing Step 3: Extracting numbers and binning

Some of the numerical fields were formatted, and the numerical value had to be extracted first before binning. For example, money values were written with a euro sign prefix and the magnitude was indicated with a "M" or "K" postfix (indicating millions or thousands).

We decided to bin the players' ages into bins of age 10, and monetary values (e.g., "Wage," "Release Clause," "Value") into logarithmic bins. Most of the player stats are values between 0 and 100, and we decided to bin these into quartiles (larger bins would be noninformative, and smaller bins would make the algorithm take too long).

The dataframe of the binned values is shown below the code.

In []:

```
# binning quantitative values
def bin(col, bins, col_name):
    new_col = col.copy()

    for i, bin_min in enumerate(bins):
        bin_max = float('Inf') if i==len(bins)-1 else bins[i+1]
        new_col[(col >= bin_min) & (col < bin_max)] = f'{bin_min}<={col_name}<{bin_max}'
```

```

return new_col

# make a money column a number
def money_to_number(df_col):
    col = df_col.copy()

    for i, val in enumerate(col):
        if val[-1] == 'M':
            val = int(float(val[1:-1]) * 1000000)
        elif val[-1] == 'K':
            val = int(float(val[1:-1]) * 1000)
        else:
            val = int(float(val[1:]))
    col.iloc[i] = val

    return col.astype('int32')

# create a copy so we don't modify the original dataframe in place
dataframe_binned = dataframe.copy()

dataframe_binned.loc[:, 'Age'] = bin(dataframe.loc[:, 'Age'],
                                      [0, 10, 20, 30, 40, 50], 'Age')
dataframe_binned.loc[:, 'Overall'] = bin(dataframe.loc[:, 'Overall'],
                                         [0, 25, 50, 75, 100], 'Overall')
dataframe_binned.loc[:, 'Potential'] = bin(dataframe.loc[:, 'Potential'],
                                           [25, 50, 75, 100], 'Potential')
dataframe_binned.loc[:, 'Value'] = bin(money_to_number(dataframe.loc[:, 'Value']),
                                       [0, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9], 'Value')
dataframe_binned.loc[:, 'Wage'] = bin(money_to_number(dataframe.loc[:, 'Wage']),
                                       [0, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9], 'Wage')
dataframe_binned.loc[:, 'Release Clause'] = bin(money_to_number(dataframe.loc[:, 'Release Clause']),
                                                [0, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9],
                                                'Release Clause')

to_bin = [
    'Jersey Number', 'FKAccuracy', 'LongPassing', 'BallControl',
    'Acceleration', 'SprintSpeed', 'Agility', 'Reactions', 'Balance',
    'ShotPower', 'Jumping', 'Stamina', 'Strength', 'LongShots', 'Aggression',
    'Interceptions', 'Positioning', 'Vision', 'Penalties', 'Composure',
    'Marking', 'StandingTackle', 'SlidingTackle', 'GKDiving', 'GKHandling',
    'GKKicking', 'GKPositioning', 'GKReflexes']

for col in to_bin:
    dataframe_binned.loc[:, col] = bin(dataframe.loc[:, col], [0, 25, 50, 75, 100], col)
dataframe_binned

```

Out[]:

	Name	Age	Nationality	Overall	Potential	Club	
0	L. Messi	30<=Age<40	Argentina	75<=Overall<100	75<=Potential<100	FC Barcelona	100000000.0<=Value<100000
1	Cristiano Ronaldo	30<=Age<40	Portugal	75<=Overall<100	75<=Potential<100	Juventus	10000000.0<=Value<10000
2	Neymar Jr	20<=Age<30	Brazil	75<=Overall<100	75<=Potential<100	Paris Saint-Germain	100000000.0<=Value<100000
3	De Gea	20<=Age<30	Spain	75<=Overall<100	75<=Potential<100	Manchester United	10000000.0<=Value<10000
4	K. De Bruyne	20<=Age<30	Belgium	75<=Overall<100	75<=Potential<100	Manchester City	100000000.0<=Value<100000
...
18202	J. Lundstram	10<=Age<20	England	25<=Overall<50	50<=Potential<75	Crewe Alexandra	10000.0<=Value<10

18203	N. Christoffersson	10<=Age<20 Age	Sweden Nationality	25<=Overall<50 Overall	50<=Potential<75 Potential	Trelleborgs Club	10000.0<=Value<10
18204	B. Worman	10<=Age<20	England	25<=Overall<50	50<=Potential<75	Cambridge United	10000.0<=Value<10
18205	D. Walker-Rice	10<=Age<20	England	25<=Overall<50	50<=Potential<75	Tranmere Rovers	10000.0<=Value<10
18206	G. Nugent	10<=Age<20	England	25<=Overall<50	50<=Potential<75	Tranmere Rovers	10000.0<=Value<10

16643 rows × 56 columns



Preprocessing Step 4: Choosing features

It is clear that the table becomes very wide at this point. If we included all of the features from the previous section (categorical and binned quantitative features), the a priori algorithm took way too long to run. We experimented with a few different combinations of which features to include. The following is one possible combination of features to perform an analysis on (an explanation of this choice will follow in a later section).

In []:

```
# didn't include most of the rows in the analysis, because the a priori
# algorithm takes too long
cols = [
    'Preferred Foot', 'Age', 'Aggression', 'Nationality',
    'Body Type', 'Reactions', 'Position', 'Balance',
    'Contract Valid Until', 'Jersey Number', 'Penalties', 'Vision',
    # 'SprintSpeed', 'Agility', 'Reactions', 'ShotPower', 'Jumping', 'Stamina',
    # 'Strength', 'LongShots', 'Aggression', 'Composure', 'Agility',
    # 'Interceptions', 'Positioning', 'Composure', 'Marking', 'StandingTackle',
    # 'SlidingTackle', 'GKDiving', 'International Reputation', 'Body Type',
    # 'GKHandling', 'GKKicking', 'GKPositioning', 'GKReflexes', 'Overall',
    # 'Value', 'Wage', 'FKAccuracy', 'LongPassing', 'BallControl', 'Acceleration'
]
```

Preprocessing Step 5: One-hot encoding items

Now that all of the data is binned, it is one-hot encoded. This is the necessary data input format for the a priori algorithm.

In []:

```
# format data for the apriori algorithm
def one_hot_encode_column(df, col):
    items = np.unique(np.array(df.loc[:,col]))
    items_onehot = df.loc[:,col][:, np.newaxis] == items[np.newaxis, :]
    return pd.DataFrame(columns=[col+' '+str(item) for item in items],
                         data=items_onehot,
                         dtype=np.int32)

basket_sets = pd.concat([
    one_hot_encode_column(dataframe_binned, col) for col in cols
], axis=1)
basket_sets

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:4: FutureWarning: Support for multi-dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be removed in a future version. Convert to a numpy array before indexing instead.
  after removing the cwd from sys.path.
```

Out []:

Preferred	Preferred	Age	Age	Age	Age	Aggression	Aggression
-----------	-----------	-----	-----	-----	-----	------------	------------

	Foot Left Preferred	Foot Right Preferred	10<=Age<20 Age	20<=Age<30 Age	30<=Age<40 Age	40<=Age<50 Age	0<=Aggression<25 Aggression	25<=Aggression<50 Aggression	50<
0	Foot Left	1 Right	10<=Age<20 0	20<=Age<30 0	30<=Age<40 1	40<=Age<50 0	0<=Aggression<25 0	25<=Aggression<50 0	50< 1
1	0	1	0	0	1	0	0	0	0
2	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	0	0	0	1
4	0	1	0	1	0	0	0	0	0
...
16638	0	1	1	0	0	0	0	0	1
16639	0	1	1	0	0	0	0	0	1
16640	0	1	1	0	0	0	0	0	1
16641	0	1	1	0	0	0	0	0	1
16642	0	1	1	0	0	0	0	0	0

16643 rows × 237 columns

Performing Market Basket Analysis

Finding itemsets

Now that the data is correctly formatted, we can run the a priori market basket analysis. We use mlxtend's a priori implementation to find itemsets with a minimum support of 0.05.

A preview of some of the itemsets with the highest support are shown below. The supports of the visible itemsets make sense. E.g., most of the players are right-footed, most of them are in their twenties, and the itemsets with small support are more specific.

(Note that, with the current feature set, this implementation takes a few minutes to complete this step. Using all of the features, this algorithm did not even complete overnight.)

In []:

```
# for usage of apriori() see: https://pbpython.com/market-basket-analysis.html
frequent_itemsets = apriori(basket_sets, min_support=0.05, use_colnames=True)
frequent_itemsets
```

Out[]:

	support	itemsets
0	0.229526	(Preferred Foot Left)
1	0.770474	(Preferred Foot Right)
2	0.117287	(Age 10<=Age<20)
3	0.681307	(Age 20<=Age<30)
4	0.200745	(Age 30<=Age<40)
...
2829	0.064472	(Aggression 50<=Aggression<75, Jersey Number 0...)
2830	0.054197	(Aggression 50<=Aggression<75, Penalties 50<=P...)
2831	0.063330	(Aggression 50<=Aggression<75, Penalties 50<=P...)
2832	0.059725	(Penalties 50<=Penalties<75, Jersey Number 0<=...)
2833	0.054858	(Aggression 50<=Aggression<75, Penalties 50<=P...)

2834 rows × 2 columns

Finding association rules

We grab a list of the association rules from the itemsets where the confidence is greater than 0.8, and the lift is greater than 7. A preview of these is shown below.

In []:

```
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
pd.set_option('max_colwidth', 100)

rules[(rules['lift'] >= 7) & (rules['confidence'] >= 0.8)]
```

Out []:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
95	(Aggression 0<=Aggression<25)	(Position GK)	0.056180	0.114162	0.053957	0.960428	8.412842	0.047543	22.385363
199	(Position GK)	(Penalties 0<=Penalties<25)	0.114162	0.100703	0.093553	0.819474	8.137530	0.082056	4.981529
200	(Penalties 0<=Penalties<25)	(Position GK)	0.100703	0.114162	0.093553	0.928998	8.137530	0.082056	12.476171
665	(Preferred Foot Right, Position GK)	(Penalties 0<=Penalties<25)	0.102566	0.100703	0.084720	0.826011	8.202442	0.074392	5.168687
666	(Preferred Foot Right, Penalties 0<=Penalties<25)	(Position GK)	0.090308	0.114162	0.084720	0.938124	8.217470	0.074411	14.316283
669	(Penalties 0<=Penalties<25)	(Preferred Foot Right, Position GK)	0.100703	0.102566	0.084720	0.841289	8.202442	0.074392	5.654511
963	(Age 20<=Age<30, Position GK)	(Penalties 0<=Penalties<25)	0.069819	0.100703	0.056360	0.807229	8.015937	0.049329	4.665103
964	(Age 20<=Age<30, Penalties 0<=Penalties<25)	(Position GK)	0.060987	0.114162	0.056360	0.924138	8.094962	0.049398	11.676954
1448	(Body Type Normal, Position GK)	(Penalties 0<=Penalties<25)	0.081536	0.100703	0.067055	0.822402	8.166612	0.058844	5.063676
1449	(Body Type Normal, Penalties 0<=Penalties<25)	(Position GK)	0.071622	0.114162	0.067055	0.936242	8.200984	0.058879	13.893668
1552	(Reactions 50<=Reactions<75, Penalties 0<=Penalties<25)	(Position GK)	0.074926	0.114162	0.068978	0.920609	8.064054	0.060424	11.157978
1687	(Balance 25<=Balance<50, Position GK)	(Penalties 0<=Penalties<25)	0.074746	0.100703	0.063330	0.847267	8.413522	0.055803	5.888029
1688	(Balance 25<=Balance<50, Penalties 0<=Penalties<25)	(Position GK)	0.065673	0.114162	0.063330	0.964318	8.446922	0.055833	24.826175
1696	(Jersey Number 0<=Jersey Number<25, Penalties 0<=Penalties<25)	(Position GK)	0.059304	0.114162	0.054197	0.913880	8.005112	0.047427	10.286141
1700	(Position GK, Vision 25<=Vision<50)	(Penalties 0<=Penalties<25)	0.075708	0.100703	0.063811	0.842857	8.369732	0.056187	5.722799
1701	(Penalties 0<=Penalties<25, Vision)	(Position GK)	0.068497	0.114162	0.063811	0.931579	8.160141	0.055991	12.946861

	25<=Vision<50) antecedents (Age 20<=Age<30, Position GK)	consequents (Penalties 0<=Penalties<25)	antecedent support 0.062789	consequent support 0.100703	support 0.050952	confidence 0.811483	lift 8.058184	leverage 0.044629	conviction 4.770383
2278	Preferred Foot Right, Position GK	(Position GK)	0.054618	0.114162	0.050952	0.932893	8.171654	0.044717	13.200437
2279	(Age 20<=Age<30, Preferred Foot Right, Penalties 0<=Penalties<25)	(Position GK)	0.060987	0.102566	0.050952	0.835468	8.145690	0.044697	5.454466
3018	(Age 20<=Age<30, Penalties 0<=Penalties<25)	(Preferred Foot Right, Position GK)	0.072883	0.100703	0.060626	0.831822	8.260151	0.053287	5.347291
3019	(Preferred Foot Right, Body Type Normal, Penalties 0<=Penalties<25)	(Position GK)	0.064351	0.114162	0.060626	0.942110	8.252389	0.053280	15.302135
3025	(Body Type Normal, Penalties 0<=Penalties<25)	(Preferred Foot Right, Position GK)	0.071622	0.102566	0.060626	0.846477	8.253022	0.053280	5.845583
3164	(Preferred Foot Right, Reactions 50<=Reactions<75, Penalties 0<=Penalties<25)	(Position GK)	0.067055	0.114162	0.062489	0.931900	8.162950	0.054834	13.007830
3170	(Reactions 50<=Reactions<75, Penalties 0<=Penalties<25)	(Preferred Foot Right, Position GK)	0.074926	0.102566	0.062489	0.834002	8.131393	0.054804	5.406283
3403	(Preferred Foot Right, Balance 25<=Balance<50, Position GK)	(Penalties 0<=Penalties<25)	0.066755	0.100703	0.057081	0.855086	8.491162	0.050359	6.205708
3404	(Preferred Foot Right, Balance 25<=Balance<50, Penalties 0<=Penalties<25)	(Position GK)	0.059064	0.114162	0.057081	0.966429	8.465412	0.050338	26.387232
3410	(Balance 25<=Balance<50, Penalties 0<=Penalties<25)	(Preferred Foot Right, Position GK)	0.065673	0.102566	0.057081	0.869167	8.474255	0.050345	6.859411
3415	(Preferred Foot Right, Position GK, Vision 25<=Vision<50)	(Penalties 0<=Penalties<25)	0.068197	0.100703	0.057922	0.849339	8.434100	0.051055	5.969018
3416	(Preferred Foot Right, Penalties 0<=Penalties<25, Vision 25<=Vision<50)	(Position GK)	0.061407	0.114162	0.057922	0.943249	8.262361	0.050912	15.609075
3422	(Penalties 0<=Penalties<25, Vision 25<=Vision<50)	(Preferred Foot Right, Position GK)	0.068497	0.102566	0.057922	0.845614	8.244613	0.050897	5.812927
5132	(Body Type Normal, Reactions 50<=Reactions<75, Penalties 0<=Penalties<25)	(Position GK)	0.054798	0.114162	0.050652	0.924342	8.096750	0.044396	11.708466

Discussion

The table shown above is for associations with high confidence (> 0.8) and lift (> 7).

The first thing to notice is that many of the association rules shown seem to be centered around goalkeepers. It seems that goalkeepers have very distinctive attributes, e.g.:

- They are not very aggressive ($0 \leq \text{aggression} < 25$)
- They are not prone to causing penalties ($0 \leq \text{penalties} < 25$)
- They have a "normal" body type, have medium reactions, and have few penalties.

There are a lot of repeated associations in this list, but far and large they are mostly about goalkeepers. This is likely related to the bimodal distributions in the earlier histograms, where goalkeepers clearly stood out from the rest; most likely the goalkeepers are very specialized while the other players are more diverse, and therefore all of these selected associations involve goalkeepers.

These results are specific to this choice of features. When we tried different set of features (e.g., including values such as "International Reputation," "Value," "Wage," "Overall," etc.), there were many less-interesting correlations. Many of the associations we saw were that the highest-paid players were those with the highest international reputation, and who had the highest overall and potential scores.

We also did not include many of the player statistics because the algorithm takes too long to run otherwise. If we knew more about soccer or FIFA 18, we could probably draw much more interesting conclusions by using the most relevant or interesting statistics by choosing different sets of features.

ECE 475 Project 7: Recommender Systems (& NMF)

Tiffany Yu, Jonathan Lam, Harris Paspuleti

To implement a basic recommendation system. Many of those datasets are already loaded into the Surprise package to make this easy. You should tune and cross validate your system to select the best values for the # of latent dimensions, the regularization parameter, and any other hyperparameters.

In []:

```
# get surprise package for NMF
!pip install scikit-surprise

Collecting scikit-surprise
  Downloading https://files.pythonhosted.org/packages/97/37/5d334adaf5ddd65da99fc65f6507e
0e4599d092ba048f4302fe8775619e8/scikit-surprise-1.1.1.tar.gz (11.8MB)
    |██████████| 11.8MB 8.3MB/s
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (fr
om scikit-surprise) (0.17.0)
Requirement already satisfied: numpy>=1.11.2 in /usr/local/lib/python3.6/dist-packages (f
rom scikit-surprise) (1.18.5)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.6/dist-packages (fr
om scikit-surprise) (1.4.1)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages (fro
m scikit-surprise) (1.15.0)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... done
  Created wheel for scikit-surprise: filename=scikit_surprise-1.1.1-cp36-cp36m-linux_x86_
64.whl size=1670943 sha256=804d6c6fafdc481a633606b7f462c7a284a9177d87f39128828be1f9e86e1a
11
  Stored in directory: /root/.cache/pip/wheels/78/9c/3d/41b419c9d2aff5b6e2b4c0fc8d25c5382
02834058f9ed110d0
Successfully built scikit-surprise
Installing collected packages: scikit-surprise
Successfully installed scikit-surprise-1.1.1
```

In []:

```
import csv
import pandas as pd
import numpy as np
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt

from surprise import NMF
from surprise import Dataset
from surprise.model_selection import cross_validate
from surprise.model_selection.search import GridSearchCV
from surprise.model_selection.split import train_test_split
from surprise import accuracy
```

Dataset: MovieLens 100K

MovieLens 100K movie ratings. Stable benchmark dataset. 100,000 ratings from 1000 users on 1700 movies. Released 4/1998.

This dataset includes a set of user details, movie details, and user ratings for movies. Each user rated at least 20 movies.

For our current analysis, we do not use the user nor the movie details (only use the ratings information).

In []:

```
# Load movielens-100k dataset (https://surprise.readthedocs.io/en/stable/dataset.html)
data = Dataset.load_builtin('ml-100k')

# Split into training and testing sets
train, test = train_test_split(data, test_size=0.2)

# show the first few ratings: tuples of (user, movie, rating)
some_ratings = []
for i, rating in enumerate(train.all_ratings()):
    if i > 20:
        break;
    some_ratings.append(rating)

print("Here is a sample of the ratings to show their format:")
some_ratings
```

Here is a sample of the ratings to show their format:

Out[]:

```
[ (0, 0, 3.0),
(0, 425, 5.0),
(0, 421, 5.0),
(0, 565, 3.0),
(0, 396, 5.0),
(0, 698, 4.0),
(0, 867, 4.0),
(0, 27, 2.0),
(0, 888, 3.0),
(0, 371, 2.0),
(0, 253, 3.0),
(0, 753, 5.0),
(0, 904, 4.0),
(0, 1009, 3.0),
(0, 62, 2.0),
(0, 1013, 2.0),
(0, 49, 3.0),
(0, 166, 4.0),
(0, 820, 3.0),
(0, 342, 3.0),
(0, 707, 2.0)]
```

Hyperparameter tuning

We use the Surprise library's builtin `GridSearchCV` for automatic hyperparameter tuning and cross-validation.

All models learn a bias for the user and movie matrices, and use the default number of epochs (50).

Explanation of hyperparameters:

- `n_factors`: Dimensionality of the latent space
- `reg_pu`: Regularization coefficient for the p_u (user) matrix
- `reg_qi`: Regularization coefficient for the q_i (item/movie) matrix
- `reg_bu`: Regularization coefficient for the user matrix bias
- `reg_bi`: Regularization coefficient for the movie matrix bias

For sake of time, we did not try very many hyperparameter combinations.

In []:

```
# grid search
params = {
    'n_factors': [10, 20, 30, 40],
    'reg_pu': [0.02, 0.002],
    'reg_qi': [0.02, 0.002],
```

```

    'biased': [True],
    'reg_bu': [0.02, 0.002],
    'reg_bi': [0.02, 0.002]
}
grid_search = GridSearchCV(NMF, params, refit=True,
                           return_train_measures=True, n_jobs=-1)
grid_search.fit(data)

algo = grid_search.best_estimator['rmse']

```

Just to get an idea of what our model looks like (and make sure it is learning), print out some of the learned parameters and the cross validation results:

In []:

```
print(grid_search.best_score, grid_search.best_params, grid_search.cv_results)
```

```

{'rmse': 0.953552256033319, 'mae': 0.7469554124946353} {'rmse': {'n_factors': 10, 'reg_pu': 0.02, 'reg_qi': 0.02, 'biased': True, 'reg_bu': 0.002, 'reg_bi': 0.002}, 'mae': {'n_factors': 10, 'reg_pu': 0.02, 'reg_qi': 0.02, 'biased': True, 'reg_bu': 0.002, 'reg_bi': 0.002}} {'split0_test_rmse': array([1.12671965, 0.95368045, 0.95090741, 0.95168667, 0.96127206,
   0.96771622, 0.96343015, 0.97002468, 1.10883185, 0.96771741,
   0.9631692 , 0.96989675, 0.97158942, 0.9766609 , 0.97787778,
   0.98198263, 1.31584452, 1.18207468, 1.2660726 , 1.23342571,
   1.27359044, 1.31343888, 1.45571804, 1.47059832, 1.47713136,
   1.90276703, 1.30630309, 1.25483712, 1.80809646, 1.76598118,
   1.52439289, 1.60655147, 1.38715191, 1.2228146 , 1.28089097,
   1.46255891, 1.93239573, 1.68333947, 1.44744428, 2.32940068,
   1.31393296, 1.39052051, 1.6002692 , 1.29269805, 1.61572357,
   1.36836678, 1.71614943, 1.7443026 , 1.51647067, 1.48985898,
   1.23764595, 1.25609113, 1.32320693, 1.8089286 , 1.51132864,
   1.35366777, 1.38264658, 1.93861343, 1.56012656, 1.33076641,
   1.66453789, 1.43874401, 1.42124056, 1.42699327]), 'split0_train_rmse': array([1.09494471, 0.82894334, 0.81912312, 0.82866759, 0.81609649,
   0.81928227, 0.81197344, 0.8109066 , 1.05844246, 0.81570303,
   0.81565164, 0.81170432, 0.81342611, 0.81052242, 0.80694748,
   0.8082808 , 1.27987854, 1.17072659, 1.24279676, 1.20336299,
   1.26260439, 1.28340369, 1.43907055, 1.43765754, 1.44416105,
   1.87082839, 1.276055 , 1.23322462, 1.79514949, 1.74761332,
   1.49740724, 1.56995021, 1.36279707, 1.20396436, 1.25769037,
   1.42251341, 1.92393414, 1.67406066, 1.41587232, 2.33549028,
   1.28809376, 1.36891215, 1.56319027, 1.26819225, 1.59076439,
   1.34689121, 1.67426619, 1.72748165, 1.47324816, 1.47586992,
   1.19972935, 1.21738893, 1.2793612 , 1.77473322, 1.49019601,
   1.3236147 , 1.35176904, 1.93283963, 1.54228655, 1.29451702,
   1.64866795, 1.41419831, 1.39653201, 1.40498487]), 'split1_test_rmse': array([0.9578362 , 0.95660042, 0.96030242, 0.95824099, 0.97006745,
   0.96924462, 0.97296251, 0.96784403, 0.967115 , 0.96812397,
   0.97132114, 0.96953766, 0.97416366, 0.97853791, 0.98172119,
   0.97390263, 1.24864739, 1.56488316, 1.31758603, 1.28044844,
   1.38535283, 2.10757309, 2.0311708 , 1.25891636, 1.44532336,
   1.23976034, 1.47038099, 1.5209942 , 1.55909743, 1.95103046,
   1.43080111, 1.45021189, 1.36112209, 1.28249282, 1.3153543 ,
   1.42651181, 1.50118939, 1.48415302, 1.43893092, 1.51442187,
   1.96476608, 1.47338056, 1.6501675 , 1.4957885 , 2.02389459,
   2.11349589, 1.94558135, 2.28154057, 1.47319227, 1.34365673,
   1.24881571, 1.49416376, 1.55417545, 1.77999499, 1.45323653,
   1.40291997, 1.48223372, 1.38457983, 2.25498252, 1.49256109,
   1.45813015, 1.60603217, 1.94118652, 1.70531816]), 'split1_train_rmse': array([0.82723242, 0.82623144, 0.8207223 , 0.81705689, 0.81663112,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6247481 , 1.45564082, 1.99278958,
   2.1098266 , 1.92392821, 2.28467115, 1.44665497, 1.31277695,
   1.2235223 , 1.45679334, 1.53262283, 1.76703582, 1.42255336,
   0.81672125, 0.81449109, 0.81293855, 0.81328882, 0.81372156,
   0.80844257, 0.81020483, 0.80806669, 0.80749009, 0.80640939,
   0.80220142, 1.22846284, 1.54054907, 1.27499007, 1.25725866,
   1.36573022, 2.09444374, 2.01256114, 1.22961918, 1.43085508,
   1.19893464, 1.42678317, 1.49092354, 1.50725412, 1.94103673,
   1.40323207, 1.40886717, 1.32141833, 1.2471837 , 1.28090066,
   1.39600264, 1.46884122, 1.46657492, 1.4124337 , 1.48744224,
   1.95140595, 1.4318439 , 1.6
```

1.38780033, 1.45209258, 1.34665376, 2.25890768, 1.46560841,
1.41322179, 1.54947572, 1.91232746, 1.69501755]), 'split2_test_rmse': array([0.950
64383, 0.95096055, 0.94962223, 0.9500036, 0.95833966,
0.96932538, 0.96192836, 0.96546021, 0.96027629, 0.95987607,
0.96944137, 0.96597663, 0.97640751, 0.96669372, 0.96985083,
0.97452261, 1.45997998, 2.0345994, 1.25876783, 1.18867224,
1.52553929, 1.21856352, 1.47322512, 1.65151549, 2.16393136,
1.30047121, 1.28784365, 1.92560739, 2.0533516, 1.6107345,
1.7291813, 1.5310527, 1.39841073, 1.52021681, 1.29448061,
1.18712045, 1.26363778, 1.59315696, 2.07191768, 1.34840731,
1.61098985, 1.60547988, 1.52687754, 2.23617758, 1.43618578,
1.75841425, 1.90521781, 1.4045004, 1.44667007, 1.51656437,
1.44219474, 1.16801012, 1.45782801, 1.3845492, 1.24080563,
1.75477752, 1.34014404, 1.4182336, 1.40958935, 1.53226556,
1.50047696, 2.44259323, 2.07185971, 2.16331437]), 'split2_train_rmse': array([0.82
610007, 0.82479289, 0.8214557, 0.82355385, 0.8193141,
0.81688956, 0.81209574, 0.81141458, 0.81461618, 0.81413053,
0.81414706, 0.81440253, 0.81102274, 0.80842588, 0.80715858,
0.80747788, 1.4395832, 2.02227308, 1.23986693, 1.16417239,
1.50389908, 1.18743723, 1.46503316, 1.64904204, 2.16333697,
1.28028017, 1.26903003, 1.90904926, 2.05212203, 1.58229121,
1.7090181, 1.51351441, 1.35980268, 1.4906797, 1.27593564,
1.15394371, 1.23443726, 1.57145944, 2.05750373, 1.31675708,
1.58759014, 1.57568775, 1.50328568, 2.22321941, 1.40945237,
1.74068722, 1.87951577, 1.38989903, 1.43465469, 1.51237645,
1.41861206, 1.13912279, 1.42162109, 1.35252327, 1.22136241,
1.7348762, 1.30403402, 1.40232066, 1.38401217, 1.51616178,
1.4827609, 2.44734427, 2.07448418, 2.14988323]), 'split3_test_rmse': array([0.952
90732, 0.9539179, 0.95346237, 0.95628911, 0.95829373,
0.96316094, 0.95875407, 0.96616403, 0.96683996, 0.96734258,
0.96869506, 0.96120746, 0.97199538, 0.97521209, 0.9753017,
0.97449908, 1.74349587, 1.26266514, 1.17039699, 1.29248545,
1.80056384, 1.31750135, 1.57674457, 1.67041406, 2.40256243,
1.32455094, 2.00408043, 1.2548823, 1.34944747, 1.91233224,
1.62165537, 1.40502404, 1.3180064, 1.26470931, 1.20349349,
1.91575091, 1.34169695, 1.98289808, 1.90086892, 1.39723572,
1.50589115, 1.40362242, 1.37605806, 1.47937256, 1.83075118,
1.54064472, 1.64456455, 1.47976693, 1.2921631, 1.65599155,
1.84058164, 1.55775434, 1.54793306, 1.40477873, 1.36530886,
1.83278872, 1.3705851, 1.38927018, 1.42809924, 1.3762888,
1.45880944, 1.53530377, 1.33686885, 1.41598549]), 'split3_train_rmse': array([0.82
561274, 0.82352473, 0.82321092, 0.82109328, 0.81971579,
0.81456774, 0.81210992, 0.81485983, 0.81562557, 0.81827036,
0.80983852, 0.80813111, 0.81137424, 0.81064366, 0.80640802,
0.80783452, 1.70621493, 1.22329865, 1.13326096, 1.26243811,
1.78228415, 1.28643869, 1.55492636, 1.63573052, 2.39810449,
1.29249424, 1.96609066, 1.22040543, 1.31083064, 1.88741828,
1.59679243, 1.37422276, 1.28184245, 1.24068041, 1.17927727,
1.90566713, 1.31792605, 1.95629873, 1.89276892, 1.34840026,
1.49439974, 1.37583372, 1.3501611, 1.46539466, 1.80957817,
1.52069854, 1.58580281, 1.44052631, 1.26827912, 1.62549052,
1.83721497, 1.52765175, 1.5084837, 1.34993367, 1.32716033,
1.79857527, 1.3354473, 1.37292723, 1.39724822, 1.3325284,
1.42370782, 1.48866219, 1.30857875, 1.39271521]), 'split4_test_rmse': array([0.954
5566, 0.95664939, 0.95736858, 0.95154091, 0.96634157,
0.97287203, 0.96974386, 0.97143182, 0.97046323, 0.96626402,
0.97379703, 0.96801789, 0.97231604, 0.97276194, 0.98264282,
0.98185131, 1.44304839, 1.35040008, 1.23130474, 1.62117892,
1.58190445, 1.20571812, 1.44651165, 1.62357041, 1.22304471,
1.43134355, 1.2085166, 1.54332326, 2.05982862, 1.77570712,
1.88489322, 1.73307092, 1.46126418, 1.29597786, 1.39784167,
2.23484977, 1.3741303, 1.46073103, 1.40637114, 1.40011268,
1.52735561, 1.99271722, 1.70643891, 1.61428064, 1.83208555,
1.68000218, 1.52894139, 1.33302387, 1.30719576, 1.39288763,
1.43192183, 1.32617638, 1.63152027, 1.34263301, 1.74770132,
1.27102044, 1.49669123, 1.53980187, 1.23449761, 1.30051032,
1.25080905, 1.28600028, 1.23798562, 1.43952827]), 'split4_train_rmse': array([0.82
884392, 0.82712005, 0.82263637, 0.82001217, 0.81812531,
0.81609555, 0.81143645, 0.80907516, 0.81268806, 0.80932167,
0.81184813, 0.80771344, 0.81200895, 0.80803485, 0.80926807,
0.80248564, 1.4135166, 1.31359614, 1.20375775, 1.57755923,
1.55911889, 1.16944291, 1.4132578, 1.60065981, 1.18953953,

	1.39580971, 1.17952447, 1.51796188, 2.07796179, 1.76244785,
	1.87539935, 1.70620512, 1.44520011, 1.25506248, 1.36690151,
	2.22649187, 1.3472133 , 1.44205405, 1.37417552, 1.38053615,
	1.49022944, 1.97301275, 1.69139741, 1.58066763, 1.81432778,
	1.6590389 , 1.49621728, 1.296639 , 1.2689719 , 1.37562751,
	1.40343832, 1.2987945 , 1.58847957, 1.31496403, 1.7190942 ,
	1.23763996, 1.48206292, 1.51584327, 1.18850302, 1.25631959,
	1.19873072, 1.24293872, 1.19881469, 1.41905309]), 'mean_test_rmse': array([0.98853
272,	0.95436174, 0.9543326 , 0.95355226, 0.96286289,
	0.96846384, 0.96536379, 0.96818496, 0.99470527, 0.96586481,
	0.96928476, 0.96692728, 0.9732944 , 0.97397331, 0.97747886,
	0.97735165, 1.44220323, 1.47892449, 1.24882564, 1.32324215,
	1.51339017, 1.43255899, 1.59667403, 1.53500293, 1.74239864,
	1.43977862, 1.45542495, 1.49992885, 1.76596431, 1.8031571 ,
	1.63818478, 1.5451822 , 1.38519106, 1.31724228, 1.29841221,
	1.64535837, 1.48261003, 1.64085571, 1.65310659, 1.59791565,
	1.58458713, 1.57314412, 1.57196224, 1.62366346, 1.74772813,
	1.69218476, 1.74809091, 1.64862687, 1.40713837, 1.47979185,
	1.44023198, 1.36043915, 1.50293274, 1.54417691, 1.4636762 ,
	1.52303488, 1.41446013, 1.53409978, 1.57745906, 1.40647843,
	1.4665527 , 1.66173469, 1.60182825, 1.63022791]), 'std_test_rmse': array([0.069133
23,	0.00212011, 0.0039883 , 0.0031494 , 0.00464488,
	0.00314527, 0.00521833, 0.00226059, 0.05715874, 0.00305758,
	0.0035293 , 0.00317621, 0.00178878, 0.00410008, 0.0046422 ,
	0.00373441, 0.17001433, 0.30580057, 0.04812968, 0.15344031,
	0.17955946, 0.34068247, 0.22218983, 0.15500148, 0.45647673,
	0.23963335, 0.28723895, 0.24643328, 0.27820963, 0.12078721,
	0.15835102, 0.11656043, 0.04702939, 0.10443626, 0.06245841,
	0.37733372, 0.2375818 , 0.18871866, 0.27778967, 0.36978221,
	0.21352064, 0.22321691, 0.11437867, 0.32310108, 0.20236638,
	0.24894188, 0.15713422, 0.34567333, 0.09065016, 0.1082451 ,
	0.21818985, 0.14553827, 0.10537685, 0.20554145, 0.16875867,
	0.22639619, 0.06295271, 0.20996219, 0.35424557, 0.09067038,
	0.13192817, 0.40490216, 0.33802085, 0.28753789]), 'mean_train_rmse': array([0.8805
4677,	0.82612249, 0.82142968, 0.82207676, 0.81797656,
	0.81671128, 0.81242133, 0.81183895, 0.86293222, 0.81422943,
	0.81198558, 0.81043125, 0.81117975, 0.80902338, 0.80723831,
	0.80565605, 1.41353122, 1.45408871, 1.2189345 , 1.29295828,
	1.49472735, 1.40423325, 1.5769698 , 1.51054182, 1.72519942,
	1.40766943, 1.42349666, 1.47431295, 1.74866361, 1.78416148,
	1.61636984, 1.51455194, 1.35421213, 1.28751413, 1.27214109,
	1.62092375, 1.4584704 , 1.62208956, 1.63055084, 1.5737252 ,
	1.56234381, 1.54505806, 1.54655651, 1.59862296, 1.72338246,
	1.67542849, 1.71194605, 1.62784343, 1.37836177, 1.46042827,
	1.4165034 , 1.32795026, 1.46611368, 1.511838 , 1.43607326,
	1.49650129, 1.38508117, 1.51411691, 1.55419153, 1.37302704,
	1.43341784, 1.62852384, 1.57814742, 1.61233079]), 'std_train_rmse': array([0.10720
473,	0.00186978, 0.0014457 , 0.00389999, 0.00142695,
	0.00152419, 0.00106372, 0.00195053, 0.09776048, 0.00292761,
	0.00265683, 0.00245439, 0.00175996, 0.00130825, 0.00105711,
	0.00271809, 0.16640661, 0.31100006, 0.04841506, 0.14684466,
	0.17744894, 0.34842298, 0.22296787, 0.15950261, 0.46846951,
	0.23988349, 0.28268019, 0.25050725, 0.30088586, 0.12476431,
	0.16470766, 0.11888022, 0.05421545, 0.10308236, 0.05983841,
	0.38879765, 0.24457545, 0.18631037, 0.28650911, 0.38519002,
	0.21773226, 0.22652377, 0.11645859, 0.32796989, 0.20219822,
	0.254993 , 0.16546108, 0.35853927, 0.09046614, 0.1087885 ,
	0.22865085, 0.14505923, 0.10774802, 0.21194027, 0.16803314,
	0.22662134, 0.06933664, 0.21716746, 0.36988833, 0.10048229,
	0.14446284, 0.42211091, 0.34858894, 0.29133463]), 'rank_test_rmse': array([15, 3,
2,	1, 4, 9, 5, 8, 16, 6, 10, 7, 11, 12, 14, 13, 29,
	33, 17, 20, 38, 26, 48, 41, 60, 27, 30, 36, 63, 64, 53, 43, 22, 19,
	18, 55, 35, 54, 57, 49, 47, 45, 44, 51, 61, 59, 62, 56, 24, 34, 28,
	21, 37, 42, 31, 39, 25, 40, 46, 23, 32, 58, 50, 52]), 'split0_test_mae': array([0.
85340428,	0.74764644, 0.74512498, 0.74475262, 0.75142782,
	0.75750802, 0.75333852, 0.7575778 , 0.85958439, 0.75544959,
	0.75394885, 0.7564572 , 0.75917736, 0.76289087, 0.76148228,
	0.76506923, 1.00083899, 0.89767931, 0.95503269, 0.94437036,
	0.95439935, 1.00608649, 1.11658584, 1.13982912, 1.13534963,
	1.54748801, 0.988979 , 0.94802946, 1.44428642, 1.39484323,
	1.17323967, 1.24693971, 1.08231383, 0.92751846, 0.97322231,
	1.13266444, 1.56138649, 1.30478834, 1.08959251, 1.97479931,

0.99978137, 1.06015886, 1.25437042, 0.98586245, 1.2639133 ,
1.05948447, 1.360073 , 1.36984881, 1.17227034, 1.14870524,
0.93822918, 0.9590122 , 0.99932788, 1.46166697, 1.16444477,
1.02667634, 1.0555321 , 1.56241185, 1.19652328, 1.01420747,
1.34519066, 1.11354883, 1.07578111, 1.10100424]), 'split0_train_mae': array([0.828
90759, 0.65041538, 0.64233104, 0.64868593, 0.63827099,
0.64247297, 0.63567967, 0.6351583 , 0.81863561, 0.63839991,
0.63791967, 0.63477498, 0.6362122 , 0.6339952 , 0.62875512,
0.63122395, 0.96967062, 0.88355219, 0.93517084, 0.91753747,
0.9397114 , 0.97895076, 1.09771219, 1.1131212 , 1.10962286,
1.51330514, 0.96470012, 0.92809919, 1.43106428, 1.37528469,
1.15008686, 1.21130201, 1.05582411, 0.90818614, 0.95168104,
1.09782126, 1.54871295, 1.29155306, 1.05803984, 1.97956884,
0.971164 , 1.0393474 , 1.21887503, 0.96039146, 1.24047674,
1.03849396, 1.31709732, 1.34973511, 1.13241042, 1.13498575,
0.90775298, 0.92729628, 0.96162597, 1.42370778, 1.14196786,
1.00363193, 1.02917875, 1.55494121, 1.17958016, 0.98317246,
1.32841904, 1.08775514, 1.04954214, 1.07673668]), 'split1_test_mae': array([0.7491
6607, 0.74874853, 0.75126797, 0.74916753, 0.75567557,
0.75586036, 0.75962952, 0.75578638, 0.75212779, 0.75352385,
0.75514196, 0.75604545, 0.75904808, 0.7610005 , 0.76313096,
0.7586177 , 0.95214262, 1.21355348, 1.00375792, 0.97103956,
1.05303688, 1.73351357, 1.65314416, 0.96529177, 1.122417 ,
0.93622833, 1.15381359, 1.17333288, 1.2017342 , 1.57007267,
1.08889174, 1.10541948, 1.04512319, 0.97313002, 1.00130008,
1.100923 , 1.14536317, 1.14309902, 1.09294164, 1.1468375 ,
1.5390504 , 1.14470085, 1.28885877, 1.16240198, 1.67436066,
1.73480152, 1.57192593, 1.92709296, 1.14582779, 1.02851579,
0.95471929, 1.14410375, 1.20920948, 1.40947893, 1.09714065,
1.05621847, 1.15307043, 1.05683378, 1.89076355, 1.15708667,
1.11801339, 1.26024047, 1.57915634, 1.35051446]), 'split1_train_mae': array([0.649
67565, 0.64953351, 0.64429604, 0.64177335, 0.63929033,
0.64046955, 0.6372398 , 0.63635262, 0.6372387 , 0.63742852,
0.63285634, 0.63387812, 0.63167247, 0.63227363, 0.62989856,
0.62709495, 0.92870597, 1.18995495, 0.96971564, 0.95207731,
1.03140204, 1.7208433 , 1.63394175, 0.93815955, 1.10314461,
0.90699021, 1.11571133, 1.14991279, 1.15676986, 1.56350452,
1.06591326, 1.0713976 , 1.01050357, 0.9393959 , 0.9720729 ,
1.07145534, 1.11672543, 1.12736464, 1.07054593, 1.12442545,
1.52597613, 1.11023008, 1.26576297, 1.1232566 , 1.64019267,
1.72976527, 1.5521985 , 1.9310018 , 1.12023157, 1.00222834,
0.93476597, 1.11284312, 1.19380748, 1.39735225, 1.06817351,
1.04020355, 1.12874076, 1.02192181, 1.89563071, 1.13339007,
1.07693181, 1.20985362, 1.54667171, 1.34123927]), 'split2_test_mae': array([0.7473
9421, 0.74784975, 0.74542023, 0.74588931, 0.75087521,
0.75978645, 0.75338316, 0.75555328, 0.75107524, 0.75195175,
0.75996583, 0.75463858, 0.76202813, 0.75649869, 0.75861401,
0.76304636, 1.1227001 , 1.65806025, 0.95876043, 0.91113034,
1.1915175 , 0.92380773, 1.12025252, 1.28606564, 1.78443254,
0.98232284, 0.97590514, 1.57197835, 1.66994225, 1.24464968,
1.36455234, 1.16210802, 1.06624604, 1.18549072, 0.98908815,
0.90521737, 0.95956649, 1.20536042, 1.70762061, 1.02907832,
1.2494465 , 1.24836905, 1.18176198, 1.88244991, 1.0989785 ,
1.38732907, 1.53921007, 1.07068354, 1.12924429, 1.20019893,
1.11055322, 0.88899929, 1.11997792, 1.03949982, 0.94963339,
1.38836246, 1.02851806, 1.10032943, 1.0697745 , 1.18592792,
1.15340111, 2.10453203, 1.69386965, 1.79388521]), 'split2_train_mae': array([0.648
23572, 0.64682518, 0.64380138, 0.6453061 , 0.64151861,
0.63983354, 0.63455134, 0.63441994, 0.63655227, 0.6366446 ,
0.63551094, 0.63654562, 0.63362252, 0.63111007, 0.62968951,
0.63056381, 1.10481705, 1.64682919, 0.94026819, 0.88715002,
1.17050114, 0.89247149, 1.10838849, 1.27877027, 1.78407061,
0.96352686, 0.95499137, 1.55111431, 1.6681957 , 1.21553067,
1.34502602, 1.14512431, 1.03365396, 1.15797972, 0.97216023,
0.87805667, 0.93137391, 1.18316778, 1.6929619 , 0.99797369,
1.22743756, 1.22482557, 1.15824059, 1.8669874 , 1.07242881,
1.37252412, 1.51409554, 1.05731487, 1.11591404, 1.19157528,
1.08619997, 0.86227315, 1.08423221, 1.0097523 , 0.93207226,
1.36708001, 0.99861368, 1.08661129, 1.04684715, 1.1701736 ,
1.12959176, 2.11052821, 1.69789271, 1.77896176]), 'split3_test_mae': array([0.7474
9069, 0.74772897, 0.74842601, 0.74973379, 0.74992447,
0.75500426, 0.75039508, 0.75669345, 0.75390989, 0.75756581,

0.75594678, 0.75127645, 0.75973009, 0.76176562, 0.76221314,
 0.76037692, 1.36482907, 0.96299614, 0.89085098, 0.97350275,
 1.4020625, 0.9998794, 1.21796715, 1.30430496, 2.05183291,
 1.01446486, 1.6315985, 0.95759352, 1.02444812, 1.54519864,
 1.25344414, 1.0881809, 0.99581831, 0.9575839, 0.92219967,
 1.52623805, 1.01085005, 1.62395343, 1.51963456, 1.05968371,
 1.16498008, 1.06443271, 1.03811574, 1.13679343, 1.4573391,
 1.17972873, 1.28603272, 1.13389341, 0.99547246, 1.28743871,
 1.44341054, 1.20924187, 1.19243611, 1.07157798, 1.03206357,
 1.46621396, 1.04370349, 1.05677748, 1.09484245, 1.04942898,
 1.09373936, 1.17579316, 1.01561364, 1.08680988]), 'split3_train_mae': array([0.647
36594, 0.64669332, 0.6460459, 0.64422488, 0.64137359,
0.63830323, 0.63607431, 0.63785185, 0.63900733, 0.64078712,
0.63298913, 0.63185018, 0.63388374, 0.63358647, 0.62923382,
0.63057699, 1.33081825, 0.93315456, 0.86089207, 0.95218984,
1.38554513, 0.97422358, 1.20333879, 1.27570633, 2.0531165,
0.98206473, 1.59192798, 0.92684039, 0.99289931, 1.52253944,
1.22944818, 1.06010314, 0.96460702, 0.93404872, 0.90481256,
1.51841853, 0.99583819, 1.60371934, 1.5123406, 1.0199443,
1.15457047, 1.0413878, 1.01594088, 1.12478922, 1.44085435,
1.15979338, 1.24082174, 1.100003, 0.97691289, 1.25948865,
1.44369927, 1.18053393, 1.15937413, 1.02753335, 1.00230245,
1.43514757, 1.01406161, 1.04105655, 1.07000269, 1.01597601,
1.06270524, 1.13725152, 0.98795827, 1.07071716]), 'split4_test_mae': array([0.7484
2693, 0.74886187, 0.75075723, 0.74523381, 0.75596175,
0.76133544, 0.75620984, 0.75872802, 0.75892279, 0.75561609,
0.7603749, 0.75611142, 0.75847172, 0.76031588, 0.76556849,
0.7655397, 1.09821746, 1.03602763, 0.93556613, 1.25712806,
1.22013492, 0.92445191, 1.09276157, 1.26335729, 0.92867365,
1.088312, 0.92132554, 1.17847636, 1.68435981, 1.41519813,
1.49612374, 1.36957798, 1.11661073, 0.99158688, 1.08333837,
1.86499256, 1.05704086, 1.12583068, 1.08397898, 1.06601687,
1.16760473, 1.61741264, 1.35397446, 1.25925362, 1.49141511,
1.32886695, 1.18807266, 1.01120228, 0.99586101, 1.06969341,
1.10297583, 1.01081157, 1.26573819, 1.02763901, 1.40114016,
0.97673214, 1.16881805, 1.16904615, 0.94251873, 0.98985755,
0.94935797, 0.97234509, 0.94652241, 1.10732041]), 'split4_train_mae': array([0.650
78512, 0.64968451, 0.64514702, 0.64286915, 0.64026699,
0.63908145, 0.63407197, 0.63288722, 0.63544128, 0.632937,
0.63445122, 0.631715, 0.63387055, 0.63185681, 0.63087556,
0.62696634, 1.0715412, 1.00155851, 0.91138682, 1.21508089,
1.19639503, 0.89344901, 1.05794394, 1.23498077, 0.89706969,
1.05683302, 0.89316113, 1.15178559, 1.69952709, 1.39815444,
1.48551117, 1.33911606, 1.10050454, 0.95812669, 1.05313117,
1.85164146, 1.03136095, 1.10081416, 1.04929204, 1.04435219,
1.13085953, 1.59353441, 1.33742017, 1.22796944, 1.47436128,
1.30714958, 1.15322502, 0.97868606, 0.96166201, 1.05161797,
1.08033286, 0.98576318, 1.22573045, 1.00027346, 1.37213222,
0.94417633, 1.15407207, 1.14700509, 0.90442049, 0.95263269,
0.90544037, 0.93377559, 0.91011386, 1.08354964]), 'mean_test_mae': array([0.769176
43, 0.74816711, 0.74819929, 0.74695541, 0.75277296,
0.7578989, 0.75459122, 0.75686779, 0.77512402, 0.75482142,
0.75707566, 0.75490582, 0.75969108, 0.76049431, 0.76220178,
0.76252998, 1.10774565, 1.15366336, 0.94879363, 1.01143422,
1.16423023, 1.11754782, 1.24014225, 1.19176976, 1.40454115,
1.11376321, 1.13432435, 1.16588211, 1.40495416, 1.43399247,
1.27525032, 1.19444522, 1.06122242, 1.007062, 0.99382972,
1.30600708, 1.14684141, 1.28060638, 1.29875366, 1.25528314,
1.22417262, 1.22701482, 1.22341627, 1.28535228, 1.39720134,
1.33804215, 1.38906287, 1.3025442, 1.08773518, 1.14691042,
1.10997761, 1.04243374, 1.15733792, 1.20197254, 1.12888451,
1.18284067, 1.08992843, 1.18907974, 1.2388845, 1.07930172,
1.1319405, 1.32529192, 1.26218863, 1.28790684]), 'std_test_mae': array([0.0421189
3, 0.00052622, 0.00257627, 0.00207681, 0.00253449,
0.00236897, 0.00311899, 0.00117408, 0.04231606, 0.00192231,
0.00260878, 0.00191848, 0.00123511, 0.00217314, 0.00226221,
0.00267262, 0.14286192, 0.2733946, 0.0365832, 0.12489549,
0.1528575, 0.31000146, 0.21091378, 0.12702029, 0.43400739,
0.22245015, 0.26048102, 0.22626188, 0.25919739, 0.11715052,
0.14308817, 0.10362484, 0.04018665, 0.0916547, 0.05224101,
0.34458845, 0.21609234, 0.18275541, 0.26389228, 0.36186004,
0.17711121, 0.20684095, 0.10803489, 0.31115507, 0.19799628,

0.22916758, 0.1468576, 0.33516373, 0.07641626, 0.09227225,
 0.18156516, 0.11803642, 0.0916909, 0.19198563, 0.1535582,
 0.20270216, 0.05882443, 0.19112703, 0.33583322, 0.07816591,
 0.12722096, 0.40080003, 0.31048645, 0.27126292]), 'mean_train_mae': array([0.68499
 4, 0.64863038, 0.64432428, 0.64457188, 0.6401441,
 0.64003215, 0.63552342, 0.63533399, 0.67337504, 0.63723943,
 0.63474546, 0.63375278, 0.6338523, 0.63256444, 0.62969051,
 0.62928521, 1.08111062, 1.13100988, 0.92348671, 0.9848071,
 1.14471095, 1.09198763, 1.22026503, 1.16814762, 1.38940485,
 1.08454399, 1.10409839, 1.14155045, 1.38969125, 1.41500275,
 1.2551971, 1.16540862, 1.03301864, 0.97954743, 0.97077158,
 1.28347865, 1.12480229, 1.26132379, 1.27663606, 1.23325289,
 1.20200154, 1.20186505, 1.19924793, 1.26067883, 1.37366277,
 1.32154526, 1.35548762, 1.28334817, 1.06142619, 1.1279792,
 1.09055021, 1.01374193, 1.12495405, 1.17172383, 1.10332966,
 1.15804788, 1.06493338, 1.17030719, 1.21929624, 1.05106897,
 1.10061764, 1.29583282, 1.23843574, 1.2702409]), 'std_train_mae': array([0.071966
 38, 0.00155718, 0.00125603, 0.00237975, 0.00123721,
 0.00141996, 0.00112477, 0.00168702, 0.07263951, 0.00256255,
 0.00186556, 0.0018239, 0.00144048, 0.0010763, 0.0007115,
 0.0018567, 0.14046603, 0.27809487, 0.03638612, 0.11766329,
 0.1524475, 0.31664789, 0.21227276, 0.12976543, 0.44696555,
 0.21967089, 0.25466522, 0.22785043, 0.2781981, 0.12269264,
 0.14741326, 0.10258581, 0.04530324, 0.09063293, 0.047962,
 0.35264579, 0.2202467, 0.18328409, 0.2723331, 0.37559852,
 0.18232026, 0.20712024, 0.10879717, 0.31502487, 0.19706522,
 0.23483244, 0.15452112, 0.3468106, 0.07557929, 0.09277133,
 0.19106079, 0.11733633, 0.09425046, 0.19535864, 0.15134019,
 0.20196598, 0.06368943, 0.19708641, 0.34933642, 0.08543281,
 0.13625382, 0.41728906, 0.32010159, 0.27420609]), 'rank_test_mae': array([15, 2,
 3, 1, 4, 10, 5, 8, 16, 6, 9, 7, 11, 12, 13, 14, 26,
 35, 17, 20, 37, 29, 48, 41, 62, 28, 32, 38, 63, 64, 51, 42, 22, 19,
 18, 57, 33, 52, 55, 49, 45, 46, 44, 53, 61, 59, 60, 56, 24, 34, 27,
 21, 36, 43, 30, 39, 25, 40, 47, 23, 31, 58, 50, 54]), 'mean_fit_time': array([8.7
 0077324, 9.18824935, 9.2647296, 9.27673426, 9.30260401,
 9.31332755, 9.28104296, 9.35330048, 9.13306317, 9.16470718,
 8.83388491, 9.00846195, 8.79199295, 8.92526679, 8.67318897,
 9.12619419, 11.79453897, 11.813445, 11.80383377, 11.73262701,
 13.3070085, 11.77521586, 11.52053161, 11.56952367, 11.17670321,
 11.54245973, 11.27851124, 11.58191743, 11.39072914, 11.40067682,
 11.12915335, 11.29315486, 14.03336987, 14.04079928, 13.91670623,
 13.86813145, 13.3442184, 15.78374019, 13.61588078, 14.00177021,
 13.44561887, 13.94466748, 13.537925, 13.76969047, 13.88315678,
 14.00760875, 13.87433329, 13.93899808, 16.38297491, 16.38706818,
 16.26456356, 16.20416617, 15.73288245, 15.88243442, 15.67933855,
 15.99228129, 15.7930501, 15.86172714, 15.80224471, 15.89892192,
 15.67439847, 16.02205677, 15.83664865, 14.4030746]), 'std_fit_time': array([0.461
 78218, 0.17651622, 0.081564, 0.20088733, 0.19487085,
 0.11412923, 0.09452429, 0.21995843, 0.32264241, 0.27410367,
 0.44974326, 0.45292282, 0.48771432, 0.5613795, 0.55415364,
 0.47016903, 0.03663351, 0.10916021, 0.1424056, 0.11406221,
 2.01106688, 0.06933074, 0.26586771, 0.61785455, 0.43533319,
 0.45689052, 0.40496399, 0.43774283, 0.51564935, 0.51069213,
 0.50163369, 0.7405142, 0.05950839, 0.20678139, 0.29246022,
 0.50034924, 0.571175, 2.27587699, 0.3716294, 0.44402895,
 0.64588315, 0.31571237, 0.50555493, 0.40439654, 0.24508718,
 0.28166377, 0.2696396, 0.35476953, 0.15721747, 0.16021795,
 0.15038527, 0.52488413, 0.54788713, 0.56125457, 0.49306142,
 0.40815473, 0.59979609, 0.38808099, 0.45052894, 0.57914859,
 0.43591216, 0.45492958, 0.71956365, 2.83924265]), 'mean_test_time': array([0.32855
 392, 0.33987112, 0.28374987, 0.29850364, 0.29417343,
 0.2571403, 0.25794449, 0.30069022, 0.33102102, 0.34844813,
 0.33161035, 0.33510728, 0.36483727, 0.29677863, 0.34120293,
 0.33787866, 0.26053619, 0.25765357, 0.25623589, 0.25146823,
 0.33518562, 0.25575504, 0.32877011, 0.29723454, 0.32811751,
 0.3177094, 0.38025622, 0.29158783, 0.37456713, 0.3353961,
 0.32473726, 0.2703567, 0.25639935, 0.27944288, 0.34474425,
 0.27782001, 0.36074743, 0.38743358, 0.32333484, 0.31697865,
 0.35323849, 0.31659946, 0.35647373, 0.31300406, 0.33982897,
 0.31289678, 0.35006819, 0.33592286, 0.26198096, 0.26265121,
 0.31552949, 0.34579372, 0.37260404, 0.32660551, 0.36445231,
 0.32939882, 0.36773071, 0.34792595, 0.38429432, 0.32091365,

Evaluating the model

In this section, we evaluate the model on all of the test dataset ratings, and manually calculate the RMSE and accuracy.

In [1]:

```
# Fit and test the model
predictions = algo.test(test)

prediction_array = []
for prediction in predictions:
    uid = int(prediction.uid)
    iid = int(prediction.iid)
    r_ui = int(prediction.r_ui)
    est = float(prediction.est)
    prediction_array.append([uid, iid, r_ui, est])

pred = np.array(prediction_array)

# calculating RMSE
rmse = np.sqrt(np.mean((pred[:,2] - pred[:,3]) ** 2))

# calculating accuracy
est_rounded = pred[:,3].astype(int)
accuracy = np.sum(pred[:,2] == est_rounded) / len(pred)
```

```
# print out test metrics
print(f'RMSE: {rmse}; Accuracy: {accuracy}')

# print out some predictions
print('Test dataset ratings vs. model predictions:')
print(np.vstack((pred[:,2], pred[:,3])).T)
```

RMSE: 0.8450194768270142; Accuracy: 0.6539
Test dataset ratings vs. model predictions:
[[5. 3.79930586]
 [5. 3.58003842]
 [3. 3.28499177]
 ...
 [3. 3.28001867]
 [4. 3.87734224]
 [4. 2.30047792]]

Making recommendations for a user

In []:

```
# do predictions!
num_users = len(train.all_users())
num_items = len(train.all_items())

# read in user and movie data
# (this was not used when making the recommender systems, but it's here so we can get some context)
user_data = pd.read_csv('https://raw.githubusercontent.com/tiyu0203/fml/master/u.user', '|', header=None)
item_data = pd.read_csv('https://raw.githubusercontent.com/tiyu0203/fml/master/u.item', '|', header=None)

# choose some arbitrary user
uid = 152

# get all of the first user's rated movies
all_ratings = []
for rating in train.all_ratings():
    all_ratings.append([*rating])
all_ratings = np.array(all_ratings)

# get all ids of movies that the user rates
user_rated_movies = all_ratings[all_ratings[:,0] == uid,:]
user_rated_movie_titles = item_data.loc[user_rated_movies[:,1], 1]
user_rated_movie_ratings = user_rated_movies[:,2]
print('Rated movies: ', np.vstack((user_rated_movie_titles, user_rated_movie_ratings)).T)

# predict highest recommendations (estimated >4.9 for this particular user)
recommendations = []
for i in range(num_items):
    if algo.predict(str(uid), str(i)).est > 4.9:
        recommendations.append(i)
recommendations = np.array(recommendations)
print('Recommended movies: \n', item_data.loc[recommendations, 1])

# show user information as well
print('User: ', user_data.loc[uid,:])
```

Rated movies: [['Ace Ventura: Pet Detective (1994)' 3.0]
['Tales From the Crypt Presents: Demon Knight (1995)' 5.0]
['Patton (1970)' 3.0]
['Jurassic Park (1993)' 3.0]
['Juror, The (1996)' 4.0]
['Dragonheart (1996)' 3.0]
["Preacher's Wife, The (1996)" 3.0]
['Madness of King George, The (1994)' 4.0]
['Nadja (1994)' 5.0]
['Little Women (1994)' 5.0]
['To Wong Foo Thanks for Everything! Julie Newmar (1995)' 4.0]

['Trainspotting (1996)' 5.0]
['Pulp Fiction (1994)' 4.0]
['Tales from the Hood (1995)' 4.0]
['Four Days in September (1997)' 3.0]
['Thin Man, The (1934)' 3.0]
['Jungle2Jungle (1997)' 2.0]
['Independence Day (ID4) (1996)' 1.0]
['Crow, The (1994)' 4.0]
['From Dusk Till Dawn (1996)' 4.0]
["Jackie Chan's First Strike (1996)" 5.0]
['Sting, The (1973)' 3.0]
['Sleepless in Seattle (1993)' 2.0]
['Last Dance (1996)' 2.0]
['Conan the Barbarian (1981)' 3.0]
['My Life as a Dog (Mitt liv som hund) (1985)' 3.0]
['Henry V (1989)' 3.0]
['Sudden Death (1995)' 4.0]
['Basic Instinct (1992)' 5.0]
['So I Married an Axe Murderer (1993)' 4.0]
['Contempt (M\AE9pris, Le) (1963)' 1.0]
['Wedding Singer, The (1998)' 2.0]
['Private Benjamin (1980)' 3.0]
['Kansas City (1996)' 1.0]
['Quick and the Dead, The (1995)' 3.0]
['Graduate, The (1967)' 3.0]
['Body Snatcher, The (1945)' 3.0]
['Meet Me in St. Louis (1944)' 1.0]
['To Kill a Mockingbird (1962)' 1.0]
['Bob Roberts (1992)' 3.0]
['Baby-Sitters Club, The (1995)' 3.0]
['Basquiat (1996)' 3.0]
['As Good As It Gets (1997)' 3.0]
['Naked Gun 33 1/3: The Final Insult (1994)' 3.0]
['Swingers (1996)' 5.0]
['Sense and Sensibility (1995)' 3.0]
['Mouse Hunt (1997)' 4.0]
['Copycat (1995)' 4.0]
['Legends of the Fall (1994)' 4.0]
['Hoodlum (1997)' 3.0]
['Commandments (1997)' 1.0]
['U Turn (1997)' 3.0]
["Schindler's List (1993)" 4.0]
['GoodFellas (1990)' 4.0]
['Leaving Las Vegas (1995)' 3.0]
['Hugo Pool (1997)' 1.0]
['Jude (1996)' 4.0]
['Shall We Dance? (1996)' 2.0]
['Wishmaster (1997)' 4.0]
['Alien: Resurrection (1997)' 5.0]
['Great Escape, The (1963)' 4.0]
['Wonderland (1997)' 4.0]
['Local Hero (1983)' 3.0]
['Haunted World of Edward D. Wood Jr., The (1995)' 2.0]
['Castle Freak (1995)' 3.0]
['Bride of Frankenstein (1935)' 1.0]
['Desperado (1995)' 4.0]
['Blues Brothers 2000 (1998)' 4.0]
['To Catch a Thief (1955)' 4.0]
['Ice Storm, The (1997)' 2.0]
['Alien (1979)' 2.0]
['Kiss the Girls (1997)' 1.0]
['Promesse, La (1996)' 3.0]
['Delicatessen (1991)' 4.0]
['Johnny Mnemonic (1995)' 3.0]
['Birdcage, The (1996)' 2.0]
['Clerks (1994)' 4.0]
['Threesome (1994)' 1.0]
['Star Wars (1977)' 3.0]
['Frighteners, The (1996)' 3.0]
["Mr. Holland's Opus (1995)" 4.0]
['In the Line of Duty 2 (1987)' 2.0]
['Fly Away Home (1996)' 3.0]

["Antonia's Line (1995)" 5.0]
['Blood & Wine (1997)' 2.0]
['Cinema Paradiso (1988)' 3.0]
['Desperate Measures (1998)' 2.0]
["April Fool's Day (1986)" 4.0]
['Young Frankenstein (1974)' 4.0]
['Twilight (1998)' 4.0]
['Cool Runnings (1993)' 2.0]
['Citizen Kane (1941)' 3.0]
['Maya Lin: A Strong Clear Vision (1994)' 2.0]
['Four Weddings and a Funeral (1994)' 5.0]
["Ulee's Gold (1997)" 4.0]
['Batman Returns (1992)' 3.0]
['Frisk (1995)' 3.0]
['Unforgiven (1992)' 4.0]
['Manon of the Spring (Manon des sources) (1986)' 2.0]
['Tom & Viv (1994)' 1.0]
['Ridicule (1996)' 3.0]
['Remains of the Day, The (1993)' 3.0]
['Lawnmower Man, The (1992)' 3.0]
['Dial M for Murder (1954)' 5.0]
['Seventh Seal, The (Sjunde inseglet, Det) (1957)' 4.0]
['Right Stuff, The (1983)' 5.0]
['Net, The (1995)' 3.0]
['Blade Runner (1982)' 2.0]
['Angels and Insects (1995)' 4.0]
['Cape Fear (1991)' 4.0]
['Liar Liar (1997)' 4.0]
['Bananas (1971)' 5.0]
['Bean (1997)' 4.0]
['Shadowlands (1993)' 4.0]
['Showgirls (1995)' 3.0]
['Simple Wish, A (1997)' 2.0]
["Someone Else's America (1995)" 4.0]
['Ninotchka (1939)' 3.0]
['Steel (1997)' 3.0]
['Crossing Guard, The (1995)' 4.0]
['Blue Chips (1994)' 2.0]
['Hudsucker Proxy, The (1994)' 2.0]
['Courage Under Fire (1996)' 4.0]
['Three Wishes (1995)' 2.0]
['When Harry Met Sally... (1989)' 2.0]
['Phenomenon (1996)' 5.0]
['Murder at 1600 (1997)' 3.0]
['Big Night (1996)' 3.0]
['Ghost and the Darkness, The (1996)' 2.0]
['Flintstones, The (1994)' 4.0]
['Homeward Bound: The Incredible Journey (1993)' 5.0]
["Romy and Michele's High School Reunion (1997)" 2.0]
['Supercop (1992)' 2.0]
['Jaws 3-D (1983)' 3.0]
['Mystery Science Theater 3000: The Movie (1996)' 4.0]
['Bogus (1996)' 4.0]
['Welcome to the Dollhouse (1995)' 4.0]
['Bad Boys (1995)' 3.0]
['Day the Earth Stood Still, The (1951)' 3.0]
['Like Water For Chocolate (Como agua para chocolate) (1992)' 4.0]
['Jaws (1975)' 4.0]
['Treasure of the Sierra Madre, The (1948)' 3.0]
['Davy Crockett, King of the Wild Frontier (1955)' 4.0]
['Shaggy Dog, The (1959)' 3.0]
['Bad Moon (1996)' 4.0]
['Flipper (1996)' 4.0]
['Star Trek III: The Search for Spock (1984)' 5.0]
['Aliens (1986)' 2.0]
['187 (1997)' 4.0]
['Theodore Rex (1995)' 5.0]
['In the Mouth of Madness (1995)' 3.0]
['Candidate, The (1972)' 4.0]
['Big Blue, The (Grand bleu, Le) (1988)' 2.0]
['Wolf (1994)' 4.0]
['Mad Love (1995)' 3.0]

```

['Apple Dumpling Gang, The (1975)' 4.0]
['Bonnie and Clyde (1967)' 2.0]
['Evil Dead II (1987)' 4.0]
['Air Bud (1997)' 2.0]
['Braveheart (1995)' 3.0]
['Pink Floyd - The Wall (1982)' 4.0]
[ "Carlito's Way (1993)" 2.0]
['Willy Wonka and the Chocolate Factory (1971)' 5.0]
['Wizard of Oz, The (1939)' 5.0]
['Sabrina (1995)' 3.0]
['Kiss Me, Guido (1997)' 3.0]
['3 Ninjas: High Noon At Mega Mountain (1998)' 4.0]
['Apocalypse Now (1979)' 3.0]
['Jumanji (1995)' 4.0]
['Hunt for Red October, The (1990)' 4.0]
['Brazil (1985)' 4.0]
['Sword in the Stone, The (1963)' 5.0]
['First Kid (1996)' 4.0]
['House of Yes, The (1997)' 4.0]
['Game, The (1997)' 1.0]]

```

Recommended movies:

12	Mighty Aphrodite (1995)
19	Angels and Insects (1995)
22	Taxi Driver (1976)
50	Legends of the Fall (1994)
59	Three Colors: Blue (1993)
...	
1639	Eighth Day, The (1996)
1642	Angel Baby (1995)
1645	Men With Guns (1997)
1650	Spanish Prisoner, The (1997)
1651	Temptress Moon (Feng Yue) (1996)

Name: 1, Length: 145, dtype: object
User: 0 153
1 25
2 M
3 student
4 60641
Name: 152, dtype: object

Stretch goal #1

Implement non-negative matrix factorization (NMF) using alternating least squares (ALS)

I.e., solve:

$$\min_{q,p} \sum_{(u,i) \in \kappa} (r_{ui} - q_i^T p_u)^2 + \lambda (\|q_i\|^2 + \|p_u\|^2)$$

(assuming q_i and p_u are column vectors)

Do this by minimizing q_i^T while keeping p_u fixed and vice versa and repeating until convergence.

See: <https://blog.insightdatascience.com/explicit-matrix-factorization-als-sgd-and-all-that-jazz-b00e4d9b21ea>

As a matrix problem (ridge regression):

$$\min_{P_u, Q_i} (R_{ui} - P_u Q_i^T)^T (R_{ui} - P_u Q_i^T) + \lambda_u \|P_u\|^2 + \lambda_i \|Q_i\|^2$$

Each row of Q_i and P_u is a item/user. (Let f denote latent space dimension, i denote the number of items, and u denote the number of users.)

$$P_u \in M_{u \times f}$$

$$Q_i \in M_{i \times f}$$

$$R_{ui} \in M_{u \times i}$$

Update rules:

$$\begin{aligned} p_u^T &\leftarrow r_u^T Q_i (Q_i^T Q_i + \lambda_u I_f)^{-1} \\ q_i^T &\leftarrow r_i^T P_u (P_u^T P_u + \lambda_u I_f)^{-1} \end{aligned}$$

(Here r_u and r_i are also column vectors, i.e., r_u is the transpose of the u th row of R_{ui} , and r_i is the i th row of R_{ui} .)



Model

This performs ALS as an algorithm for NMF. We did not implement a learned bias (this is similar to using `biased=False` in the Surprise library).

`__init__`

Initializes the model. Generates the (sparse) R_{ui} matrix, and initializes P_u and Q_i using random normal distributions.

Parameters:

- `dataset` : training dataset; tuples of `(user, item, rating)` in the format of the Surprise training dataset
- `n_factors` : latent dimension (default 15)
- `lambda_u` : L2 regularization coefficient for `P_u` (default 0.02)
- `lambda_i` : L2 regularization coefficient for `Q_i` (default 0.02)

`loss`

Calculates the loss on the training dataset. Used during training.

`update`

Performs the update rule on P_u (once for every user) and Q_i (once for every item). (This is considered one epoch).

By fixing Q_i , we can solve optimally for P_u (since this is a (quadratic) ridge regression problem), and vice versa. However, R_{ui} is sparse and we only want to train on the rated items (otherwise we would be training users to give most movies a zero rating). To accomplish this, we can only train on a single user at a time, only using the vector of movies that the user rates and "filtering" the items matrix Q_i to those same items. The same applies when solving for Q_i .

`train`

Performs the update rule `epoch` times, and reports the training loss.

Parameters:

- `epochs` : number of epochs to train

`predict`

Predict the rating for a given user and item.

Parameters:

- `user`: (integral) id of user to predict
- `item`: (integral) id of item to predict

`predict_all`

Predict the ratings for every user for every item (i.e., estimate the full, dense R_{ui}).

`test`

Predict the ratings given user, item pairs.

Parameters:

- `dataset`: Test dataset, tuples of (user, item, rating)

In []:

```
class ALS_NMF():

    # assumes dataset in the same format as the one given by the surprise library
    def __init__(self, dataset, n_factors=15, lambda_u=0.02, lambda_i=0.02):
        self.user_count, self.item_count = 0, 0
        for _ in dataset.all_users(): self.user_count += 1
        for _ in dataset.all_items(): self.item_count += 1

        self.R_ui = np.zeros((self.user_count, self.item_count))

        self.ratings = []
        for rating in dataset.all_ratings():
            self.ratings.append(rating)
            self.R_ui[rating[0], rating[1]] = rating[2]

        self.ratings = np.array(self.ratings, dtype=np.int)
        self.N = self.ratings.shape[0]

        # initializes P_u and Q_i
        self.P_u = np.random.normal(size=(self.user_count, n_factors))
        self.Q_i = np.random.normal(size=(self.item_count, n_factors))

        self.n_factors = n_factors

        self.lambda_u = lambda_u
        self.lambda_i = lambda_i

    # calculate loss (to check if update rule works)
    def loss(self):
        losses = np.zeros(self.N)
        for j, rating in enumerate(self.ratings):
            u, i, r = rating
            u, i = int(u), int(i)
            losses[j] = (r - self.P_u[u, :] @ self.Q_i[i, :].T) ** 2

        return np.mean(losses) + self.lambda_u * np.linalg.norm(self.P_u) + self.lambda_i * np.linalg.norm(self.Q_i)

    # update rule
    def update(self):
        # assume Q_i is fixed, update P_u
        # loop through each user
        for u in range(self.user_count):
            # find the items that this user has rated
            unfiltered_user_ratings = self.R_ui[u, :]
            user_rated_items = unfiltered_user_ratings > 0

            # "filtered" things
            r_u = unfiltered_user_ratings[user_rated_items]
```

```

Q_i = self.Q_i[user_rated_items, :]

    # convex quadratic optimal soln
    #  $P_u^T = r_u @ Q_i @ (Q_i^T @ Q_i + \lambda_u * I_f)^{-1}$ 
    self.P_u[u, :] = r_u @ Q_i @ np.linalg.pinv(Q_i.T @ Q_i + self.lambda_u * np.eye(self.n_factors))

    # assume  $P_u$  is fixed, update  $Q_i$ 
    # loop through each item
    for i in range(self.item_count):
        # find the users that rated this item
        unfiltered_item_ratings = self.R_ui[:, i].T
        users_rating_this_item = unfiltered_item_ratings > 0

        # "filtered" things
        r_i = unfiltered_item_ratings[users_rating_this_item]
        P_u = self.P_u[users_rating_this_item, :]

        # convex quadratic optimal soln
        self.Q_i[i, :] = r_i @ P_u @ np.linalg.inv(P_u.T @ P_u + self.lambda_i * np.eye(self.n_factors))

    # train method
    def train(self, epochs):
        print(f'Initial loss: {self.loss()}')
        for i in range(epochs):
            self.update()
            # for j, rating in enumerate(self.ratings):
            #     self.update(rating)
            print(f'Epoch: {i}; Loss: {self.loss()}')

    # evaluate
    #  $r_{ui} = p_u^T @ q_i$ 
    def predict(self, user, item):
        if user.item >= self.user_count or i >= self.item_count:
            est = np.random.random() * 5
        else:
            est = self.P_u[user, :] @ self.Q_i[item, :].T
        return est

    # predict cross product of all ratings for training set
    def predict_all(self):
        return self.P_u @ self.Q_i.T

    # this assumes that the test dataset is a list of tuples (unlike the train dataset)
    def test(self, dataset):
        ratings = []
        for u, i, r in (dataset if isinstance(dataset, list) else dataset.all_ratings()):
            ratings.append([self.predict(int(u), int(i)), r])
        return np.array(ratings)
:
```

Choose the parameters and train the model

When choosing the lambdas, if we chose a higher number the loss would converge to a much larger number; if we chose a small number, the loss would converge to a really small number as well as decrease in a steeper exponential decay. We left `n_factors` the same as the default from the NMF function in the surprise library. After testing a variety of numbers for epochs, we realized that the loss converged relatively quickly so we wouldn't need many epochs.

In []:

```

# choose regularization coefficient
lambda_u = 0.02
lambda_i = 0.02

# choose f
n_factors = 15

```

```

# choose epochs
epochs = 50

# run
model = ALS_NMF(train, n_factors, lambda_u, lambda_i)
model.train(epochs)

#Regularization coefficient for the p_u (user) matrix
P_u = model.P_u()
#Regularization coefficient for the q_i (item/movie) matrix
Q_i = model.Q_i()

print(f'P_u: {P_u}, Q_i: {Q_i}')

```

```

Initial loss: 34.80673227663466
Epoch: 0; Loss: 11.626744968095991
Epoch: 1; Loss: 7.1488177789313525
Epoch: 2; Loss: 6.855198026518326
Epoch: 3; Loss: 6.710935612941744
Epoch: 4; Loss: 6.60877224014469
Epoch: 5; Loss: 6.537936217060015
Epoch: 6; Loss: 6.484704692789716
Epoch: 7; Loss: 6.441820974992813
Epoch: 8; Loss: 6.40476382115052
Epoch: 9; Loss: 6.371439849812138
Epoch: 10; Loss: 6.343120364076325
Epoch: 11; Loss: 6.317881711400218
Epoch: 12; Loss: 6.293993108763518
Epoch: 13; Loss: 6.270658686126371
Epoch: 14; Loss: 6.249897929032098
Epoch: 15; Loss: 6.230931581722405
Epoch: 16; Loss: 6.215299061388394
Epoch: 17; Loss: 6.20149670735049
Epoch: 18; Loss: 6.189626131529177
Epoch: 19; Loss: 6.1799425362648055
Epoch: 20; Loss: 6.172100212112769
Epoch: 21; Loss: 6.165451455079193
Epoch: 22; Loss: 6.159029255519029
Epoch: 23; Loss: 6.153080408069977
Epoch: 24; Loss: 6.147599787424625
Epoch: 25; Loss: 6.142599610421463
Epoch: 26; Loss: 6.137694837852222
Epoch: 27; Loss: 6.132855586045467
Epoch: 28; Loss: 6.127942720259837
Epoch: 29; Loss: 6.123030971053174
Epoch: 30; Loss: 6.118364178585848
Epoch: 31; Loss: 6.114153134429683
Epoch: 32; Loss: 6.110514686016048
Epoch: 33; Loss: 6.107411928208701
Epoch: 34; Loss: 6.104623013399651
Epoch: 35; Loss: 6.101905470349132
Epoch: 36; Loss: 6.0991549062158725
Epoch: 37; Loss: 6.096329892911716
Epoch: 38; Loss: 6.093217255559823
Epoch: 39; Loss: 6.089669596566827
Epoch: 40; Loss: 6.08577225956736
Epoch: 41; Loss: 6.081678662186059
Epoch: 42; Loss: 6.077578280345212
Epoch: 43; Loss: 6.073496232967704
Epoch: 44; Loss: 6.069378567838467
Epoch: 45; Loss: 6.065248497242837
Epoch: 46; Loss: 6.0610428387872854
Epoch: 47; Loss: 6.057205415951149
Epoch: 48; Loss: 6.053581193456878
Epoch: 49; Loss: 6.05020695722102

```

Calculating the RMSE and Accuracy for Test dataset

Our RMSE on the test dataset is low, but the accuracy is no better than random guessing. We were not able to determine why this is the case. Increasing the regularization parameters to try to prevent overfitting did not help.

diagnose why this is the case. Increasing the regularization parameters to try to prevent overfitting did not help. In the next section we also test on the training dataset and show that does train well, so we are not sure why it does not generalize past the training dataset.

In [2]:

```
test_res = model.test(test)
rmse = np.sqrt(np.mean((test_res[:, 0] - test_res[:, 1]) ** 2))
accuracy = np.mean(np.round(test_res[:, 0]) == np.round(test_res[:, 1]))
print(f'RMSE: {rmse}, Accuracy: {accuracy}')
```

RMSE: 1.9356349226785399, Accuracy: 0.21745

Calculating the RMSE and Accuracy for the Train dataset

As a sanity check, we ran the train dataset in our model to see if there was an issue on how we implemented the algorithm. This indicates that our model does indeed train correctly, so we are not sure why it doesn't generalize.

In []:

```
test_res = model.test(train)
rmse = np.sqrt(np.mean((test_res[:, 0] - test_res[:, 1]) ** 2))
accuracy = np.mean(np.round(test_res[:, 0]) == np.round(test_res[:, 1]))
print(f'RMSE: {rmse}, Accuracy: {accuracy}')
```

RMSE: 0.5474335410733868, Accuracy: 0.6905625