Wireless Communications Lab 2

Ariana Olson

February 2019

1 Multiple Antennas for Space-Division Multiple Access

Please see the Appendix for the code used to implement the receivers in this section. In addition, the code can be found on Github. `space_division.py` is the script in which the receivers were implemented.

1.1 System Overview

The signals transmitted in this section of the lab were transmitted from two separate device with single and antennas and received by a device with two antennas. The devices were wired together for timing synchronization, so the received signals could be assumed to have no frequency or phase offset to correct for. Additionally, the channels are assumed to be flat fading.

The transmitted signal were composed as follows:

1. Transmitter 1 and transmitter 2 send 5000 zero samples at the same time.
2. Transmitter 1 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. Transmitter 2 sends the equivalent number of zero samples during this time.
3. Transmitter 1 and transmitter 2 send 5000 zero samples at the same time.
4. Transmitter 2 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. Transmitter 1 sends the equivalent number of zero samples during this time.
5. Transmitter 1 and transmitter 2 send 5000 zero samples at the same time.
6. Transmitter 1 and transmitter 2 send 1024 data bits encoded with BPSK and 40 sample long rectangular pulses at the same time.

Figure 1 shows the transmitted samples, and the signals detected at the receiver are shown in Figure 2.

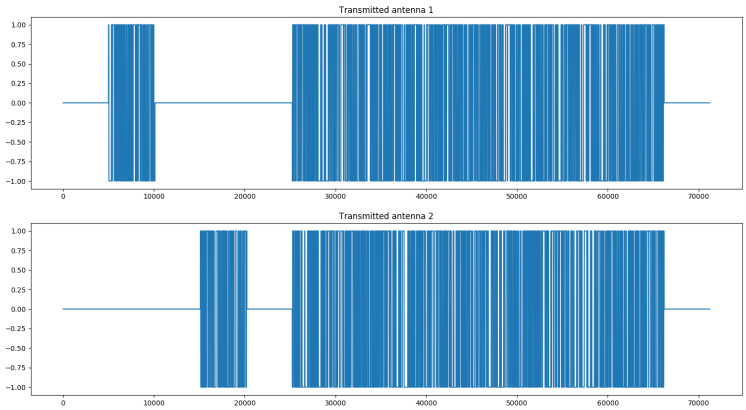

Figure 1: The signals transmitted by antennas 1 and 2.

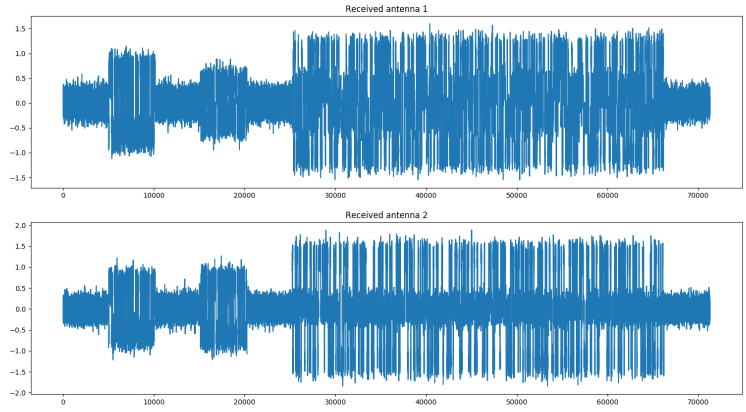

Figure 2: The signals received at antennas 1 and 2. Both signals have larger amplitude sections corresponding to when the headers were transmitted, and significant interference between the signals is evident.

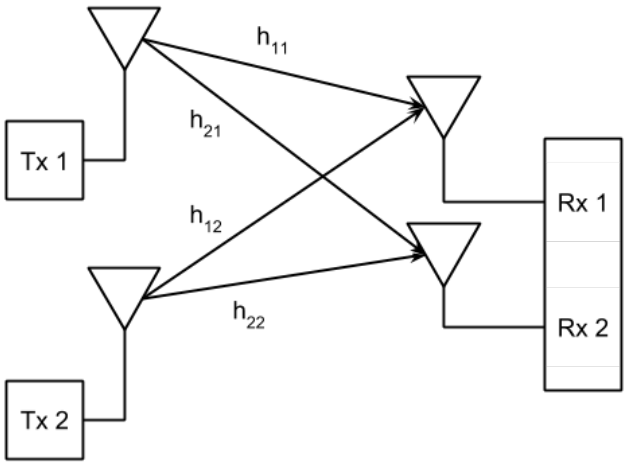

Figure 3: There are 4 channels that must be estimated in order to recover the signals.

1.2 Channel Estimation

Channel estimation was done using the known transmitted headers. There are four channels in this system to estimate. The diagram in Figure 3 shows these channels. Channel h_{ij} is the path to antenna i from transmitted j.

The channels can be estimated with the assumption that they are flat fading. For a given sample,

$$y[k] = hx[k]$$

Because the headers are known, we can take the average of a section of the received signals where a header was received divided by the corresponding header. These channel estimations are placed in the matrix H according to their indices. For the four-channel system, the matrix is:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

1.3 Zero-Forcing Receiver

The zero forcing receiver is implemented by applying the inverse of the channel matrix H to the received signals. The signals can be represented using matrix

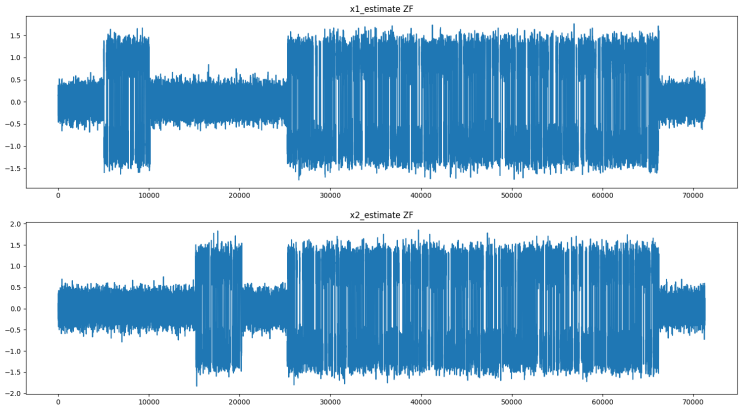

Figure 4: The signals separated using a zero-forcing receiver. It appears by inspection that the interference between signals has been significantly reduced. There is only one section corresponding to the headers in each signal.

operations:

$$Y = HX + N$$

Y is a 2 by n matrix in which the first row contains the received signal at antenna 1 and the second row contains the received signal at antenna 2. X is a 2 by n matrix with rows corresponding to the signals transmitted at antenna 1 and antenna 2. H is the channel estimation matrix. N is a 2 by n matrix of the added noise.

To recover the signals, we can apply the weight matrix W to the received signals, where $W = H^{-1}$.

$$\hat{X} = WY$$

If the signal to noise ratio is high, $\hat{X}$ will be nearly equal to X. The result of applying the zero forcing receiver can be seen in Figure 4.

1.4 Minimum-Mean-Squared-Error (MMSE) Receiver

The MMSE receiver also used the channel estimation matrix H. However, it uses these estimations differently in order to perform better in the presence of noise. Let the row vectors of H be defined as h_1 and h_2. For the MMSE receiver, the weight matrix is defined as

$$Ph_1h_1^\dagger + Ph_2h_2^\dagger + \sigma I_2$$

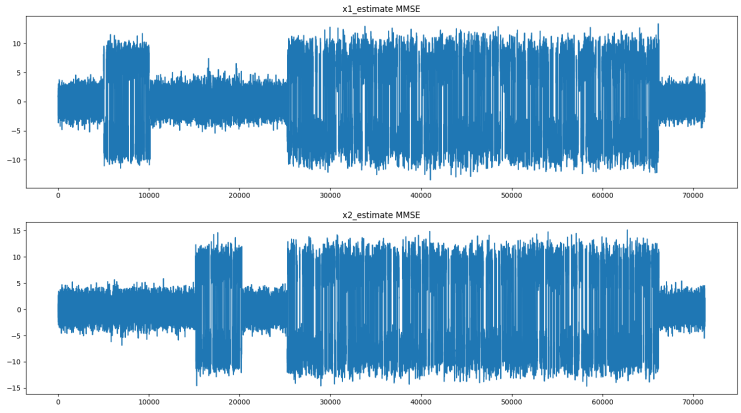

Figure 5: The signals separated using a minimum-mean-squared-error receiver. Like the signals separated using the zero forcing receiver, this plot demonstrates that significant separation of the signals occurred because there is only one header section in each signal.

Where P is the power of the transmitted signal, and σ is the variance of the noise. The power of the transmitted signal in this case was taken to be 1, and the variance of the noise was estimated by taking the variance of the first section of the signal at antenna corresponding to where zero samples were transmitted. The MMSE weight matrix is applied to the received signal in the same way as in the zero forcing receiver. The results of applying the MMSE receiver can be seen in Figure 5.

1.5 Analysis

The received and separated signals were decoded by taking the sign of the majority of the samples in each sampling period. The bit error rate was then calculated by comparing the data bit seqeunce of the transmitted signal with the corresponding sections of both of the decoded recovered signals. In both cases, the bit-error rate was 1. This is probably because the noise in the received signals was not high enough to effect the receivers significantly.

The signal to noise ratio of the signals was measured using the following process:

- Take the RMS of the zero-sample portion of the desired signal.
- Take the RMS of the data portion of the desired signal.

- Convert both RMS values to decibels. For this application, decibels are measured with respect to a power of 1.
- Subtract the RMS in dB of the noise from the RMS in dB of the data.

$$SNR = 20\log_{10}(RMS\{noise\}) - 20\log_{10}(RMS\{data\})$$

Please see the appendix for implementation details.

The average SNR across channels of the raw received signal is 12.62dB. The SNR across channels of the signal recovered with the zero-forcing receiver is 11.63dB, and the average SNR across channels of the signal recovered with the MMSE receiver is 10.55. So, both receivers decreased the SNR, but the MMSE receiver decreased the SNR more.

If there were more than two transmitters and/or receivers, the transmit signal would have to be constructed differently. In the case that the channel estimation matrix is still square, it is possible to simply append more sequences of headers for each transmitter. See section 2.2 for an example of this. As the number of antennas increases, the number of entries in the channel estimation matrix grow exponentially, and the length of the transmitted signals grows, which decreases the data rate. So, this approach is not optimal for a system with many antennas transmitting. A different approach to channel estimation would be needed that does not rely on one header sent at a time. If this reliance is gone, all of the headers could be sent at once, and the data rate would remain constant as the number of antennas increased.

2 MIMO Channels

Please see the Appendix for the code used to implement the receivers in this section. In addition, the code can be found on Github. `mimo.py` is the script in which the receivers in this section were implemented.

2.1 System Overview

IN this section of the lab, we simulate a communications system with 4 transmition antennas and 4 receiving antennas. Four signals were sent through a simulated 4x4 channel which output four "received" signals. The transmitted signals were constructeucted as follows:

1. All four transmitters send 5000 zero samples at the same time.
2. Transmitter 1 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. All other transmitters send an equivalent number of zero samples during this time.
3. All four transmitters send 5000 zero samples at the same time.

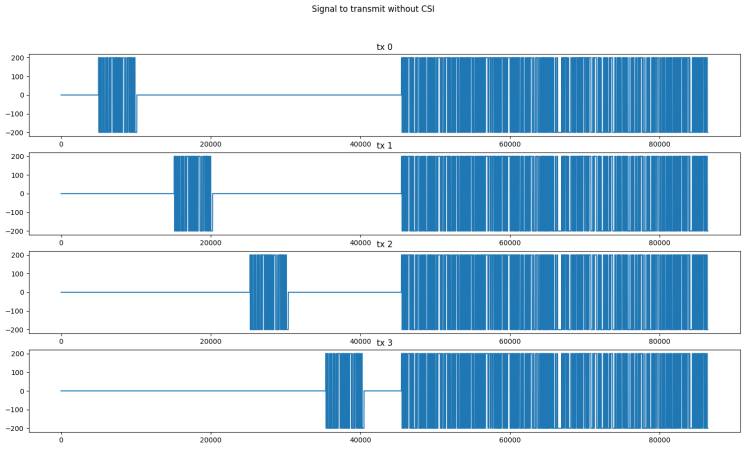

Figure 6: The signals transmitted through the simulated 4x4 channel.

4. Transmitter 2 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. All other transmitters send an equivalent number of zero samples during this time.
5. All four transmitters send 5000 zero samples at the same time.
6. Transmitter 3 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. All other transmitters send an equivalent number of zero samples during this time.
7. All four transmitters send 5000 zero samples at the same time.
8. Transmitter 4 sends 128 pseudo-random bits of data, encoded with BPSK and 40 sample long rectangular pulses. All other transmitters send an equivalent number of zero samples during this time.
9. All four transmitters send 5000 zero samples at the same time.
10. All four transmitters send 1024 data bits encoded with BPSK and 40 sample long rectangular pulses at the same time.

The signals are amplified by a factor of 200 before being transmitted through the 4x4 channel. Depending on the receiver used, a transformation was applied to the signal before sending it through the channel.

2.2 Channel Estimation

Channel estimation was performed using the same process as described in Section 1.2, using the four header sections. For the 4x4 channel system, the channel

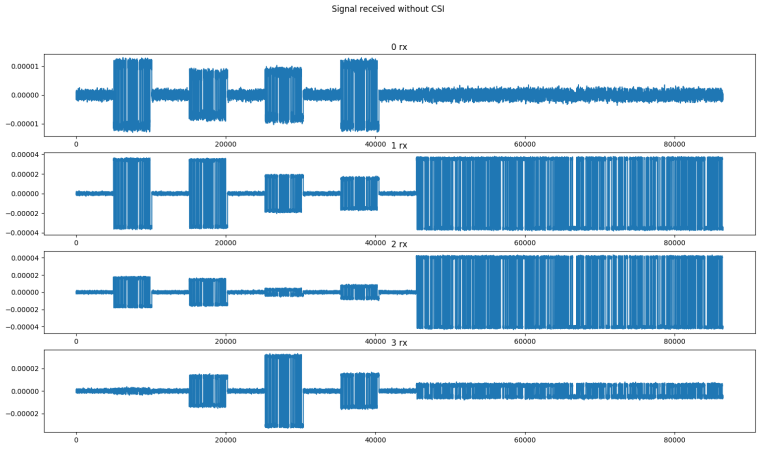

Figure 7: The signals received after sending the transmit signal (Figure 6) through the simulated 4x4 channel.

estimation matrix is:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & h_{34} \\ h_{41} & h_{42} & h_{43} & h_{44} \end{bmatrix}$$

2.3 Transmission without CSI

In this section of the lab, the channel state information (CSI) of the simulated channel was not known. The signal transmitted through the channel is shown in Figure 6. No transformation based on CSI waas performed on this signal. The signal received is shown in Figure 7. A zero forcing receiver was used to separate the received signals. The zero forcing receiver was implemented with the same process as described previously in section 1.3 using the 4x4 channel estimation matrix H. The separated signals are shown in Figure 8.

2.4 Transmission with known CSI

In this section of the lab, the CSI of the simulated channel was known. It is useful here to define some new terms:

- S is the matrix of the signals to transmit before a transformation has been applied (See Figure 6). The signals are contained in the rows of S.
- X is the matrix of the transformed signals to transmit (See Figure 9). The signals are contained in the rows of X

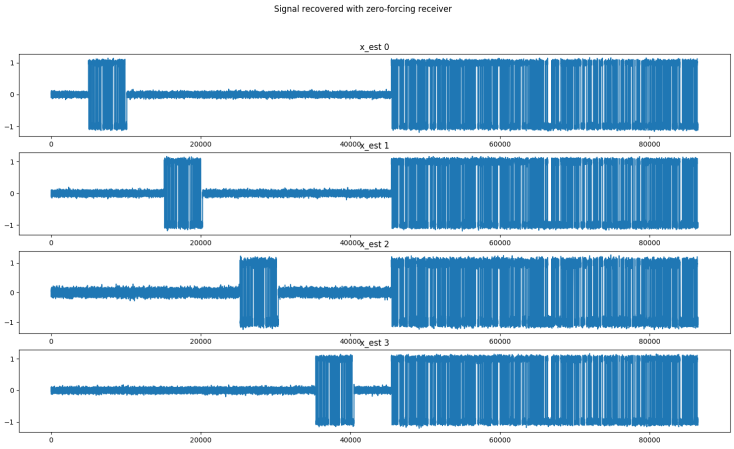

Figure 8: The received signals recovered using a zero-forcing receiver. No transformation was applied to the transmitted signal based on CSI.

- $\hat{S}$ is the matrix of estimates of S after the signal is recovered. If the recovery is perfect, $S = \hat{S}$.

Using the singular value decomposition of the channel estimations from the zero-forcing receiver implementation, the transmitted signal was transformed as follows:

$$H = U\Sigma V^{\dagger}$$
$$X = VS$$

So, after going through the simulated channel, the received signal is

$$Y = HX$$
$$Y = U\Sigma V^{\dagger}VS$$
$$Y = U\Sigma S$$

The signal received after transmitting X is shown in Figure 10. To recover the signal, the following transformation was applied:

$$\hat{S} = U^{\dagger}Y$$
$$\hat{S} = U^{dagger}U\Sigma S$$
$$\hat{S} = \Sigma S$$

Σ is a diagonal matrix, so it represents a set of scaling factors, which can be corrected for if desired by normalizing the amplitudes of the received signal. For this application, it was not necessary to perform normalization before decoding the data. The signal recovered with this approach is shown in Figure 11

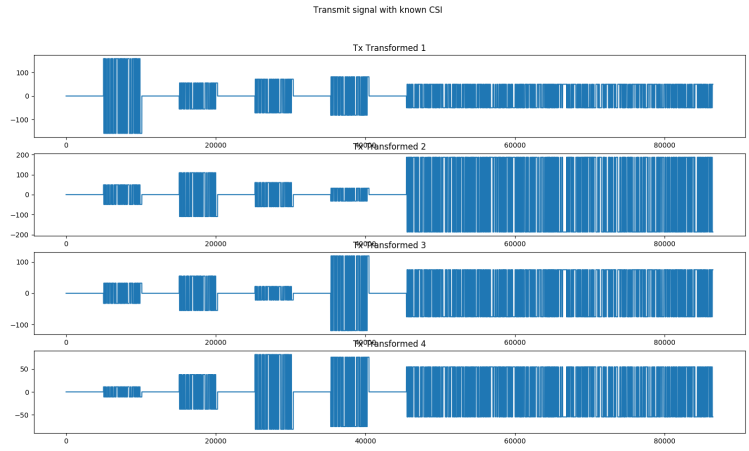

Figure 9: The signals to transmit transformed using the known channel state information. The signals were amplified by a factor of 200 before being transmitted through the simulated 4x4 channel.

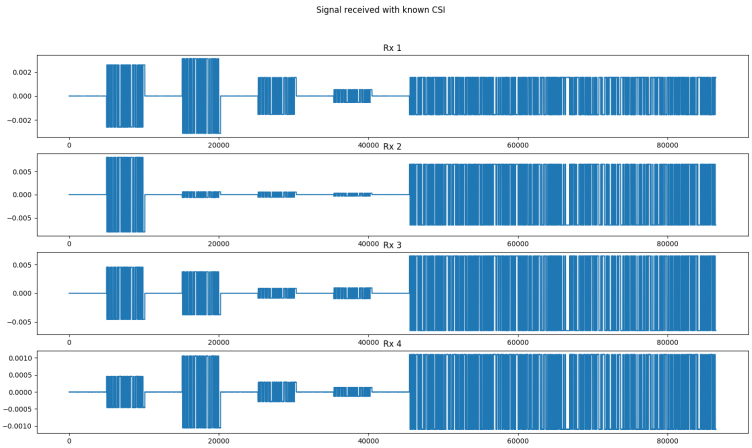

Figure 10: The signals received after sending the transformed transmit signal (Figure 9) through the simulated 4x4 channel with known channel state information.

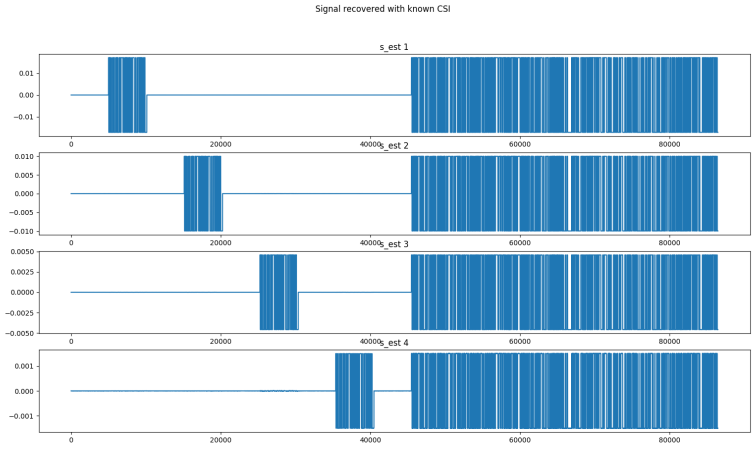

Figure 11: The signals recovered using known CSI.
Principles of Wireless Communcations

2.5 Analysis

Both the zero forcing receiver and the CSI receiver performed with a bit-error rate of 0. By visual inspection of the results, it appears that the signal to noise ratio of the signals recovered with the zero forcing receiver are lower than in the signals recovered using CSI. The average SNR of the non-transformed received signals is 30.94dB, and the average SNR of the signals recovered with the zero forcing receiver is 23.31. This is expected because the zero forcing receiver can amplify noise.

Please see Section 1.5 for an explanation of how the following signal to noise ratios were calculated. The average SNR of the transformed received signals is 31.0, and the average SNR of the signals recovered with CSI is 75.39, which is much higher than the zero forcing receiver signals. An SNR value this high indicates an excellent signal according to this article. So, using CSI to estimate the transmitted signal results in a much stronger and easier to decode signal than a zero forcing receiver. In addition, using CSI could eliminate the need to send the headers sequentially instead of all at once, because the channel would only need to be estimated once, and then subsequent data packets could send all headers at once, which increases the data rate. This would work well assuming that the channel is not rapidly changing. If the channel is changing rapidly, a receiver the the zero-forcing receiver or an MMSE receiver will perform better because they do not rely on the channel being know ahead of time.

3 Appendix

3.1 space_division.py

```
"""Use and compare a zero forcing receiver and
    a MMSE receiver on data transmitted through a
    4x4 channel.

This script is for the Principles of Wireless
    Communications Lab Part a.
"""
from __future__ import print_function, division

import matplotlib.pyplot as plt
import numpy as np

from util import signal_util, receivers

# Read transmitted signals.
tx1 = signal_util.load_complex_float32('data/
    tx1.dat')
tx2 = signal_util.load_complex_float32('data/
    tx2.dat')

signals = [(tx1, 'Transmitted antenna 1'), (tx2
    , 'Transmitted antenna 2')]
signal_util.make_subplots(signals)

# Read received signals.
rx1 = signal_util.load_complex_float32('data/
    rx1.dat')
rx2 = signal_util.load_complex_float32('data/
    rx2.dat')

signals = [(rx1, 'Received antenna 1'), (rx2, '
    Received antenna 2')]
signal_util.make_subplots(signals)

# Grab slices of the signal used in the
    receiver implementations.
tx_header1, tx_header2 = signal_util.
    get_headers_tx(tx1, tx2)
tx_data1, tx_data2 = signal_util.get_data_tx(
    tx1, tx2)
```

```
rx_header1, rx_header2, zeros1, zeros2, data1,
    data2 = signal_util.get_slices_rx(rx1, rx2)

# Calculate the SNR of the received signal.
noise_rx1 = rx1[:3 * signal_util.ZERO_SAMPLES
    // 4]
noise_rx2 = rx2[:3 * signal_util.ZERO_SAMPLES
    // 4]

snr = signal_util.calculate_snr(np.vstack((
    noise_rx1, noise_rx2)), np.vstack((data1,
    data2)))
print(np.average(snr), 'Average before recovery
    ')

# Decode the transmitted data into a bit
    sequence for calculating bit-error
# rate.
symbols1_tx = signal_util.decode_bpsk(tx_data1)
symbols2_tx = signal_util.decode_bpsk(tx_data2)

# Estimate the channels between antennas.
H = receivers.estimate_channel(rx_header1,
    rx_header2, zeros1, zeros2,
          tx_header1, tx_header2)

# Recover signals with zero-forcing receiver.
W_zf = receivers.calculate_weights_zero_forcing
    (H)

x1_est, x2_est = receivers.recover_signals(rx1,
     rx2, W_zf)

# Plot the recovered signals.
estimates = [x1_est, x2_est]
estimate_titles = ['x1_estimate ZF', '
    x2_estimate ZF']
signals = [(x, y) for x, y in zip(estimates,
    estimate_titles)]
signal_util.make_subplots(signals)

# Calculate the bit-error rate of recovered
    signals.
_, _, noise_1, noise_2, data1_zf, data2_zf =
    signal_util.get_slices_rx(x1_est, x2_est)
```

```
symbols1_zf = signal_util.decode_bpsk(data1_zf)
symbols2_zf = signal_util.decode_bpsk(data2_zf)

error1_zf = signal_util.calculate_error_rate(
    symbols1_tx, symbols1_zf)
error2_zf = signal_util.calculate_error_rate(
    symbols2_tx, symbols2_zf)

# Calculate the SNR of the recovered signals.
snr = signal_util.calculate_snr(np.vstack((
    noise_1, noise_2)), np.vstack((data1_zf,
    data2_zf)))
print(np.average(snr))

print("ZF error in 1: {}, in 2: {}".format(
    error1_zf, error2_zf))

# Recover signals with MMSE receiver.
tx_power = 1
sigma = signal_util.estimate_noise_var(rx1)
W_mmse = receivers.calculate_weights_mmse(
    tx_power, sigma, H)

x1_est, x2_est = receivers.recover_signals(rx1,
     rx2, W_mmse)

# Plot the recovered signals.
estimates = [x1_est, x2_est]
estimate_titles = ['x1_estimate MMSE', '
    x2_estimate MMSE']
signals = [(x, y) for x, y in zip(estimates,
    estimate_titles)]
signal_util.make_subplots(signals)

# Calculate the bit-error rate of the recovered
     signals.
_, _, noise_1, noise_2, data1_mmse, data2_mmse
    = signal_util.get_slices_rx(x1_est, x2_est)
symbols1_mmse = signal_util.decode_bpsk(
    data1_mmse)
symbols2_mmse = signal_util.decode_bpsk(
    data2_mmse)

error1_mmse = signal_util.calculate_error_rate(
    symbols1_tx, symbols1_mmse)
```

```
error2_mmse = signal_util.calculate_error_rate(
    symbols2_tx, symbols2_mmse)

# Calculate the SNR of the recovered signals.
snr = signal_util.calculate_snr(np.vstack((
    noise_1, noise_2)), np.vstack((data1_mmse,
    data2_mmse)))
print(np.average(snr))

print("MMSE error in 1: {}, in 2: {}".format(
    error1_mmse, error2_mmse))
```

3.2 mimo.py

```
"""Use and compare a zero forcing receiver and
    a CSI receiver for a 4x4 MIMO Channel.

This script is for the Principles of Wireless
    Communications Lab Part b.
"""
from __future__ import print_function, division

import matplotlib.pyplot as plt
import numpy as np

from MIMOChannel import MIMOChannel4x4
from util import signal_util, receivers

# Generate matrices containing data and header
    bits.
data = signal_util.generate_symbols_mimo(
    signal_util.DATA_BITS, 10)
headers = signal_util.generate_symbols_mimo(
    signal_util.HEADER_BITS, 5)

# Generate the samples to transmit.
bpsk_data = signal_util.generate_data_mimo(data
    , signal_util.PULSE_SIZE)
bpsk_headers = signal_util.generate_data_mimo(
    headers, signal_util.PULSE_SIZE)

# Put it all together into a matrix of 4
    complete signals. See doc comments on
# signal_util.create_tx_mimo for information on
    how these were generated.
```

```
tx_mimo = signal_util.create_tx_mimo(
    bpsk_headers, bpsk_data, signal_util.
    ZERO_SAMPLES)

# Need to apply a gain before transmitting
    through the channel.
gain_amplitude = 200

# Plot the non-transformed transmitted signal
    with amplification.
signals = [(gain_amplitude * tx_mimo[i, :], 'tx
    {}'.format(i)) for i in range(4)]
plt.suptitle('Signal to transmit without CSI')
signal_util.make_subplots(signals)

# Use a Zero Forcing Receiver to recover a
    signal without CSI.

# Send the signal through the channel.
# NOTE: MIMOChannel4x4 returns a numpy matrix,
    and the functions written in the
# util/ directory are written for ndarrays. The
     returned matrix is converted
# back to an ndarray for compatibility.
rx_mimo = np.asarray(MIMOChannel4x4(
    gain_amplitude * np.mat(tx_mimo)))

# Plot the result of sending the non-
    transformed signals through the channel.
signals = [(rx_mimo[i, :], '{} rx'.format(i))
    for i in range(4)]
plt.suptitle('Signal received without CSI')
signal_util.make_subplots(signals)

# Find the indices of the portions of the
    signals used to estimate the
# channels and the data portions of the signals
    .
# TODO: Consider using xcorr to find sections
    of data instead of doing it
# manually.
header_samples = signal_util.PULSE_SIZE *
    signal_util.HEADER_BITS
data_samples = signal_util.PULSE_SIZE *
    signal_util.DATA_BITS
```

```
header1_start = signal_util.ZERO_SAMPLES
header1_end = signal_util.ZERO_SAMPLES +
   header_samples

header2_start = signal_util.ZERO_SAMPLES +
   header1_end
header2_end = header2_start + header_samples

header3_start = signal_util.ZERO_SAMPLES +
   header2_end
header3_end = header3_start + header_samples

header4_start = signal_util.ZERO_SAMPLES +
   header3_end
header4_end = header4_start + header_samples

data_start = header4_end + signal_util.
   ZERO_SAMPLES
data_end = data_start + data_samples

# Initialize matrix to hold signal slices. A
    signal at sections[i, j, :]
# corresponds to the section of rx at antenna i
    + 1 where a header was sent
# from tx antenna j + 1.
sections = np.zeros((4, 4, header1_end -
   header1_start), dtype=np.complex128)

# The headers at the antenna they were intended
    for.
header11 = np.copy(rx_mimo[0, header1_start:
   header1_end])
header22 = np.copy(rx_mimo[1, header2_start:
   header2_end])
header33 = np.copy(rx_mimo[2, header3_start:
   header3_end])
header44 = np.copy(rx_mimo[3, header4_start:
   header4_end])

sections[0, 0, :] = header11
sections[1, 1, :] = header22
sections[2, 2, :] = header33
sections[3, 3, :] = header44
```

```
# The interference of header 1 in non-rx
    antenna 1 signals.
zeros21 = np.copy(rx_mimo[1, header1_start:
    header1_end])
zeros31 = np.copy(rx_mimo[2, header1_start:
    header1_end])
zeros41 = np.copy(rx_mimo[3, header1_start:
    header1_end])

sections[1, 0, :] = zeros21
sections[2, 0, :] = zeros31
sections[3, 0, :] = zeros41

# The interference of header 2 in non-rx
    antenna 2 signals.
zeros12 = np.copy(rx_mimo[0, header2_start:
    header2_end])
zeros32 = np.copy(rx_mimo[2, header2_start:
    header2_end])
zeros42 = np.copy(rx_mimo[3, header2_start:
    header2_end])

sections[0, 1, :] = zeros12
sections[2, 1, :] = zeros32
sections[3, 1, :] = zeros42

# The interference of header 3 in non-rx
    antenna 3 signals.
zeros13 = np.copy(rx_mimo[0, header3_start:
    header3_end])
zeros23 = np.copy(rx_mimo[1, header3_start:
    header3_end])
zeros43 = np.copy(rx_mimo[3, header3_start:
    header3_end])

sections[0, 2, :] = zeros13
sections[1, 2, :] = zeros23
sections[3, 2, :] = zeros43

# The interference of header 4 in non-rx
    antenna 4 signals.
zeros14 = np.copy(rx_mimo[0, header4_start:
    header4_end])
zeros24 = np.copy(rx_mimo[1, header4_start:
    header4_end])
```

```
zeros34 = np.copy(rx_mimo[2, header4_start:
    header4_end])

sections[0, 3, :] = zeros14
sections[1, 3, :] = zeros24
sections[2, 3, :] = zeros34

# Estimate the channel from these signal slices
    .
H = receivers.estimate_channel_mimo(sections,
    bpsk_headers)

# Calculate the ZF weight matrix
W_zf = receivers.calculate_weights_zero_forcing
    (H)

# Use ZF to estimate the signals.
x_est = receivers.recover_signals_mimo(rx_mimo,
     W_zf)

# Plot the recovered signals.
signals = [(x_est[i, :], 'x_est {}'.format(i))
    for i in range(4)]
plt.suptitle('Signal recovered with zero-
    forcing receiver')
signal_util.make_subplots(signals)

# Transmit and recover MIMO signals with known
    CSI.

# Transform the signal using CSI.
U, S, tx_transform = receivers.preprocess_tx(
    tx_mimo * gain_amplitude, H)

# Send the transformed signal through the
    channel.
# NOTE: MIMOChannel4x4 takes in a numpy matrix
    and returns a numpy matrix. The
# conversions are to maintain compatibility
    with the functions in the util/
# directory that use ndarrys.
rx_mimo_csi = np.asarray(MIMOChannel4x4(np.mat(
    tx_transform) * gain_amplitude))

# Recover the signal using CSI.
```

```
s_est = receivers.recover_signals_csi(
    rx_mimo_csi, U)

# Plot the transmitted, received and recovered
    signals.
signals_tx = [(tx_transform[i, :], 'Tx
    Transformed {}'.format(i + 1)) for i in range
    (4)]
signals_rx = [(rx_mimo_csi[i, :], 'Rx {}'.
    format(i + 1)) for i in range(4)]
signals_s_est = [(s_est[i, :], 's_est {}'.
    format(i + 1)) for i in range(4)]

plt.suptitle('Transmit signal with known CSI')
signal_util.make_subplots(signals_tx)

plt.suptitle('Signal received with known CSI')
signal_util.make_subplots(signals_rx)

plt.suptitle('Signal recovered with known CSI')
signal_util.make_subplots(signals_s_est)

# These are the indices of the section of noise
    right before data is
# transmitted.
noise_start = data_start - signal_util.
    ZERO_SAMPLES
noise_end = data_start

# Calculate SNRs.
rx_noise = rx_mimo[:, noise_start:noise_end]
rx_csi_noise = rx_mimo_csi[:, noise_start:
    noise_end]
x_est_noise = x_est[:, noise_start:noise_end]
s_est_noise = s_est[:, noise_start:noise_end]

rx_data = rx_mimo[:, data_start:data_end]
rx_csi_data = rx_mimo_csi[:, data_start:
    data_end]
x_est_data = x_est[:, data_start:data_end]
s_est_data = s_est[:, data_start:data_end]

snr_rx = signal_util.calculate_snr(rx_noise,
    rx_data)
```

```
snr_rx_csi = signal_util.calculate_snr(
   rx_csi_noise, rx_data)
snr_x_est = signal_util.calculate_snr(
   x_est_noise, x_est_data)
snr_s_est = signal_util.calculate_snr(
   s_est_noise, s_est_data)

print('Average SNR rx: {}, Average SNR ZF: {}'.
   format(
    np.average(snr_rx),
    np.average(snr_x_est)))

print('Average SNR rx with CSI: {}, Average SNR
    CSI: {}'.format(
    np.average(snr_rx_csi),
    np.average(snr_s_est)))

# Calculate the bit-error rates of the two
    different receivers.
data_bits = np.zeros(data.shape)

for i in range(data_bits.shape[0]):
    data_bits[i, :] = signal_util.decode_bpsk(
       bpsk_data[i, :])

# Error in the zero-forcing receiver
    implementation.
bits_x_est_1 = signal_util.decode_bpsk(x_est[0,
    data_start:data_end])
error_x_est_1 = signal_util.
   calculate_error_rate(data_bits[0, :],
   bits_x_est_1.real)

bits_x_est_2 = signal_util.decode_bpsk(x_est[1,
    data_start:data_end])
error_x_est_2 = signal_util.
   calculate_error_rate(data_bits[1, :],
   bits_x_est_2.real)

bits_x_est_3 = signal_util.decode_bpsk(x_est[2,
    data_start:data_end])
error_x_est_3 = signal_util.
   calculate_error_rate(data_bits[2, :],
   bits_x_est_3.real)
```

```
bits_x_est_4 = signal_util.decode_bpsk(x_est[3,
    data_start:data_end])
error_x_est_4 = signal_util.
   calculate_error_rate(data_bits[3, :],
   bits_x_est_4.real)

print('''error rate from ZF receiver.
         Signal 1: {},
         Signal 2: {},
         signal 3: {},
         Signal 4: {} '''.format(error_x_est_1,
            error_x_est_2, error_x_est_3,
             error_x_est_4))

# Error in the CSI receiver implementation.
bits_s_est_1 = signal_util.decode_bpsk(s_est[0,
    data_start:data_end])
error_s_est_1 = signal_util.
   calculate_error_rate(data_bits[0, :],
   bits_s_est_1.real)

bits_s_est_2 = signal_util.decode_bpsk(s_est[1,
    data_start:data_end])
error_s_est_2 = signal_util.
   calculate_error_rate(data_bits[1, :],
   bits_s_est_2.real)

bits_s_est_3 = signal_util.decode_bpsk(s_est[2,
    data_start:data_end])
error_s_est_3 = signal_util.
   calculate_error_rate(data_bits[2, :],
   bits_s_est_3.real)

bits_s_est_4 = signal_util.decode_bpsk(s_est[3,
    data_start:data_end])
error_s_est_4 = signal_util.
   calculate_error_rate(data_bits[3, :],
   bits_s_est_4.real)

print('''error rate from CSI receiver.
         Signal 1: {},
         Signal 2: {},
         Signal 3: {},
         Signal 4: {} '''.format(error_s_est_1,
            error_s_est_2, error_s_est_3,
```

```
            error_s_est_4))
```

3.3 signal_util.py

```
"""This file contains functions for a creating
    and analyzing signals for both
Part a and Part b of the Principles of Wireless
    Communications Lab.
"""
import numpy as np
import matplotlib.pyplot as plt

HEADER_BITS = 128
DATA_BITS = 1024
PULSE_SIZE = 40
ZERO_SAMPLES = 5000
TOTAL_SAMPLES = PULSE_SIZE * (HEADER_BITS * 2 +
    DATA_BITS) + ZERO_SAMPLES * 3

def load_complex_float32(path):
    """Loads a .dat file of 32 bit floating
        point values as a single complex
    signal.

    It is assumed that the .dat file being
        loaded encodes complex numbers as
    interleaved 32 bit floating point real and
        imaginary values. This function
    formats that data into a single complex
        array.

    Args:
        path (string): The relative path to the
            .dat file.

    Returns:
        y (complex 1D ndarray): The complex
            signal.
    """
    tmp = np.fromfile(path, dtype=np.float32)

    y = tmp[::2] + 1j * tmp[1::2]

    # Make these vectors read-only to avoid
        inadvertently changing them later.
```

```
    y.flags.writeable = False
    return y

def get_headers_tx(tx1, tx2):
    """Grab the 128 psuedo-random bits from the
        transmitted signals.

    The transmitted signals are each made up of
        the following parts:

    1. 5000 zero samples.
    2. 128 psuedo-random bits from tx1, encoded
        using BPSK and 40 sample long
        rectangular pulses. 128 zeros from tx2.
    3. 5000 zero samples.
    4. 128 psuedo-random bits from tx2, encoded
        using BPSK and 40 sample long
        rectangular pulses. 128 zeros from
            pulse 2.
    5. 5000 zero samples from both tx1 and tx2.
    6. 1024 data bits, encoded using BPSK and
       40 sample long rectangular pulses from
       both tx1 and
        tx2.

    Args:
        tx1 (complex numpy array): The signal
            transmitted at antenna 1.
        tx2 (complex numpy array): The signal
            transmitted at antenna 2.

    Returns:
        header1, header2 (complex numpy arrays)
            : The headers of tx1 and tx2,
            respectively.
    """
    header1_start = ZERO_SAMPLES
    header1_end = header1_start + (HEADER_BITS
       * PULSE_SIZE)
    header1 = np.copy(tx1[header1_start:
       header1_end])

    header2_start = header1_end + ZERO_SAMPLES
    header2_end = header2_start + (HEADER_BITS
       * PULSE_SIZE)
```

```
    header2 = np.copy(tx2[header2_start:
        header2_end])

    return header1, header2

def get_data_tx(tx1, tx2):
    """Extract the data portion of the received
        signals.

    Args:
        tx1 (complex 1D ndarray): The signal
            transmitted at antenna 1.
        tx2 (complex 1D ndarray): The signal
            transmitted at antenna 2.

    Returns:
        data1, data2

        data1 (complex 1D ndarray): The data
            portion of tx1.
        data2 (complex 1D ndarray): The data
            portion of tx2.
    """
    data_start = 3 * ZERO_SAMPLES + 2 * (
        HEADER_BITS * PULSE_SIZE)
    data_end = data_start + (DATA_BITS *
        PULSE_SIZE)

    data1 = np.copy(tx1[data_start:data_end])
    data2 = np.copy(tx2[data_start:data_end])

    return data1, data2

def get_slices_rx(rx1, rx2):
    """Grab the segments of the received
        signals corresponding to when the two
        headers were transmitted.

    A 128 bit pseudo-random header is
        transmitted from antenna 1, followed by a
    series of 5000 zero samples, then a 128 bit
        header is sent from antenna 2.
    This function grabs the segments of both
        received signals corresponding to
```

```
    the times at which the transmitted headers
        were received.

    Args:
        rx1 (complex numpy array): The signal
            transmitted at antenna 1.
        rx2 (complex numpy array): The signal
            transmitted at antenna 2.

    Returns:
        header1, header2, zeros1, zeros2, data1
            , data2

        header1, header2 (complex numpy arrays)
            : The segments of rx1 and rx2
              corresponding to where there should
                  be a header ideally.
        zero1, zero2 (complex numpy arrays):
            The segments of rx1 and rx2
              corresponding to where a header was
                  transmitted and we should
                 ideally
              expect zeros.
        data1, data2 (complex numpy arrays):
            The data portions of rx1 and rx2.
    """
    rx1_thresh = np.sqrt(np.mean(np.square(np.
        abs(rx1))))

    # These values were found through
        inspection of the data. This is very
    # specific to the data and should not be
        used as a generalized function.
    header1_start = np.argmax(np.abs(rx1) >
        rx1_thresh) - 2
    header1_end = header1_start + (HEADER_BITS
        * PULSE_SIZE)

    header1 = np.copy(rx1[header1_start:
        header1_end])
    zeros2 = np.copy(rx2[header1_start:
        header1_end])

    header2_start = header1_end + ZERO_SAMPLES
```

```
    header2_end = header2_start + (HEADER_BITS
        * PULSE_SIZE)

    header2 = np.copy(rx2[header2_start:
        header2_end])
    zeros1 = np.copy(rx1[header2_start:
        header2_end])

    data1_start = header2_end + ZERO_SAMPLES
    data2_start = header2_end + ZERO_SAMPLES

    data1_end = data1_start + (DATA_BITS *
        PULSE_SIZE)
    data2_end = data2_start + (DATA_BITS *
        PULSE_SIZE)

    data1 = np.copy(rx1[data1_start:data1_end])
    data2 = np.copy(rx2[data2_start:data2_end])

    return header1, header2, zeros1, zeros2,
        data1, data2

def estimate_noise_var(rx):
    """Estimate the variance in the noise from
        a portion of the received signal that
        contains no data.

    Args:
        rx (complex 1D ndarray): A received
            signal.

    Returns:
        sigma (float): The estimated variance
            of the noise.
    """
    # Take a portion of the beginning of the
        signal as the noise.
    noise = rx[:3 * ZERO_SAMPLES // 4]
    return noise.var()

def make_subplots(signals):
    """Makes and displays subplots of given
        signals.

    Args:
```

```
        signals (list of tuples): A list of
            tuples where the first element is
              the signal data to plot, and the
                second element is the title of
                the
              subplot.
    """
    num_plots = len(signals)
    for i, signal in enumerate(signals):
        plt.subplot(num_plots, 1, i + 1)
        plt.plot(signal[0])
        plt.title(signal[1])
    plt.show()

def decode_bpsk(data):
    """Decode BPSK data into symbols.

    Args:
        data (complex 1D ndarray): A signal
            containing BPSK data.

    Returns:
        symbols (1D ndarray): An array of the
            decoded symbols.
    """
    symbols = np.zeros(data.shape[-1] //
       PULSE_SIZE)
    for i in range(0, data.shape[-1],
       PULSE_SIZE):
        samples = data[i:i + PULSE_SIZE]
        is_positive = (samples.real > 0).sum()
           > (PULSE_SIZE // 2)
        if is_positive:
            symbols[i // PULSE_SIZE] = 1
    return symbols

def calculate_error_rate(symbols_tx, symbols_rx
   ):
    """Calculate the percent error rate of a
       received and decoded data signal.

    Args:
        symbols_tx (1D ndarray): An array of
            transmitted bits.
```

```
        symbols_rx (1D ndarray): An array of
            received bits.

    Returns:
        percent_error (float): The percent
            error of the decoded received signal.
    """
    assert symbols_tx.shape == symbols_rx.shape
    return 100 - (100 * (symbols_rx ==
        symbols_tx).sum() / symbols_tx.shape)

def generate_symbols_mimo(num_symbols, seed):
    """Generate 4 random data sequences to
        transmit over the MIMO channel.

    Args:
        num_symbols (int): The number of
            symbols of data to generate.
        seed (int): The seed for the random
            number generator.

    Returns:
        symbols (ndarray of shape (4,
            num_symbols)): An array consisting
        of +1 and -1 values.
    """
    np.random.seed(seed)
    symbols = np.sign(np.random.randn(4,
        num_symbols))

    return symbols

def generate_data_mimo(symbols, symbol_period):
    """Modulate a series of symbols into BPSK
        data.

    Args:
        symbols (numpy array of shape (4,
            num_symbols)): An array of 1s and -1s
              representing bits to transmit.
        symbol_period (int): The number of
            samples in a single pulse.

    Returns:
```

```
            bpsk_data (ndarray of shape (4,
                num_symbols * symbol_period)): An
                array
                of 4 bpsk data signals to transmit.
        """
        pulse = np.ones(symbol_period)
        bpsk_list = []

        for i in range(4):
            x = np.zeros(symbol_period * symbols.
                shape[-1]-symbol_period+1)
            x[::symbol_period] = symbols[0, :]
            tmp = np.convolve(x, pulse)
            bpsk_list.append(tmp)

        bpsk_data = np.vstack(bpsk_list)
        return bpsk_data

    def create_tx_mimo(headers, data, num_zeros):
        """Creates 4 MIMO signals for transmitting
            through the channel.

        Each of the signals is constructed in the
            following way:
        1. A section of zero samples.
        2. A section of psuedo-random bits from
            tx1, encoded using BPSK
        3. A section of zero samples.
        2. A section of psuedo-random bits from
            tx2, encoded using BPSK
        3. A section of zero samples.
        2. A section of psuedo-random bits from
            tx3, encoded using BPSK
        3. A section of zero samples.
        2. A section of psuedo-random bits from
            tx4, encoded using BPSK
        5. A section of zero samples from all
            signals.
        6. A section of data bits, sent from all
            antennas.

        Args:
            headers (ndarray of shape (4,
                num_header_samples)): BPSK modulated
                headers to prepend to the data.
```

```
        data (ndarray of shape (4,
            num_data_samples)): BPSK modulated
            data to transmit.
        num_zeros(int): The number of zeros to
            use for padding.

    Returns:
        tx_mimo (ndarray of shape (4, total
            samples): The BPSK signals to
            transmit with MIMO over the channel.
    """
    header_samples = headers.shape[-1]
    data_samples = data.shape[-1]
    samples_per_signal = header_samples * 4 +
        data_samples + num_zeros * 5
    tx_mimo = np.zeros((4, samples_per_signal))

    header1_start = num_zeros
    header1_end = num_zeros + header_samples

    header2_start = num_zeros + header1_end
    header2_end = header2_start +
        header_samples

    header3_start = num_zeros + header2_end
    header3_end = header3_start +
        header_samples

    header4_start = num_zeros + header3_end
    header4_end = header4_start +
        header_samples

    data_start = header4_end + num_zeros
    data_end = data_start + data_samples

    tx_mimo[0, header1_start:header1_end] =
        headers[0, :]
    tx_mimo[1, header2_start:header2_end] =
        headers[1, :]
    tx_mimo[2, header3_start:header3_end] =
        headers[2, :]
    tx_mimo[3, header4_start:header4_end] =
        headers[3, :]

    tx_mimo[:, data_start:data_end] = data
```

```
    return tx_mimo

def rms(signals):
    """Calculate the RMS values of signals.

    Args:
        signals (complex 2D ndarray): A matrix
            with rows containing signals.

    Returns:
        rms (real 1D ndarray): A vector with
            entries corresponding to the rms
              of each row of signals.
    """
    rms = np.zeros(signals.shape[0])
    for i in range(signals.shape[0]):
        rms[i] = np.sqrt(np.mean(np.square(np.
            abs(signals[i, :]))))

    return rms

def linear_to_db(linear_signal):
    """Convert a signal from linear to dB.

    In this case, dB is in reference to 1.0.

    Args:
        linear_signal (real 2D ndarray): A
            matrix with rows containing the
            linear signals.

    Returns:
        db_signal (real 2D ndarray): A matrix
            with rows containing the signals
              with samples mesaured in dB.
    """
    reference = 1.0
    return 20 * np.log10(linear_signal)

def calculate_snr(noise_signal, data_signal):
    """Calculate the signal to noise ratio from
        the given signals.

    Args:
```

```
        noise_signal (complex 2D ndarray): A
            matrix with rows containing
              portions of a signal with just
                 noise. The samples of the signal
                 are
              linear amplitudes.
        data_signal (complex 2D ndarray): A
            matrix with rows containing
              portions of a signal with data and
                 additive noise. The samples of
              the signal are linear amplitudes.

    Returns:
        snr (float): The signal to noise ratio
            between noise_signal and data_signal,
              in dB.
    """
    rms_noise = rms(noise_signal)
    rms_data = rms(data_signal)

    noise_db = linear_to_db(rms_noise)
    data_db = linear_to_db(rms_data)

    snr = data_db - noise_db
    return snr
```

3.4 receivers.py

```
"""This file contains functions to implement
   the receivers in both Part a and
Part b of the Principles of Wireless
   Communications Lab 2
"""

from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt

def estimate_channel(rx1_header, rx2_header,
   zeros1, zeros2, tx1_header, tx2_header):
    """Estimate the channel using the headers
       sent from each antenna.

    Channel estimation is the same for both the
        zero-forcing and MMSE
```

```
    receivers. This function uses copies of the
        arrays given to it because any
    0s are replaced by small numbers ot avoid
        division errors.

    Args:
        rx1_header (complex 1D ndarray): The
            portion of the signal received at
                rx antenna 1 corresponding to the
                    header transmitted at tx antenna
                1.
        rx2_header (complex 1D ndarray): The
            portion of the signal received at
                rx antenna 2 corresponding to the
                    header transmitted at tx antenna
                2.
        zeros1 (complex 1D ndarray): The
            portion of the signal received at rx
                antenna 1 corresponding the the
                    header transmitted at tx antenna
                2.
        zeros2 (complex 1D ndarray): The
            portion of the signal received at rx
                antenna 2 corresponding the the
                    header transmitted at tx antenna
                1.
        tx1_header (complex 1D ndarray): The
            header transmitted from tx antenna
                1.
        tx2_header (complex 1D ndarray): The
            header transmitted from tx antenna
                2.

    Returns:
        H (complex (2, 2) ndarray): A matrix of
            channel estimations.
    """
    header11 = np.copy(rx1_header)
    header21 = np.copy(zeros2)
    header12 = np.copy(zeros1)
    header22 = np.copy(rx2_header)

    # Replace 0s in denominators to avoid
        division errors.
    tx1_header[tx1_header == 0] = 1e-12
    tx2_header[tx2_header == 0] = 1e-12
```

```
    H = np.zeros((2, 2), dtype=np.complex64)
    H[0][0] = np.mean(header11 / tx1_header)
    H[0][1] = np.mean(header12 / tx2_header)
    H[1][0] = np.mean(header21 / tx1_header)
    H[1][1] = np.mean(header22 / tx2_header)

    return H

def estimate_channel_mimo(rx_sections,
    tx_headers):
    """Estimate the channel using the headers
       sent from each antenna.

    Channel estimation is the same for both the
        zero-forcing and MMSE
    receivers. This function uses copies of the
        arrays given to it because any
    0s are replaced by small numbers ot avoid
       division errors.

    Args:
        rx_sections (complex (4, 4, header_bits
           ) ndarray): A matrix of portions
             of the signal with the first two
                indices corresponding to the
             antenna the signal was received at
                and the antenna the header sent
             suring this time was transmitted
                from.
        tx_headers (ndarray of shape (4,
           header_bits)): The known transmitted
             headers.

    Returns:
        H (complex (4, 4) ndarray): A matrix of
            channel estimations.
    """
    # Replace 0s in denominators to avoid
       division errors.
    tx_headers[tx_headers == 0] = 1e-12

    H = np.zeros((4, 4), dtype=np.complex64)
    for i in range(4):
        for j in range(4):
```

```
                H[i, j] = np.mean(rx_sections[i, j,
                    :] / tx_headers[i, :rx_sections.
                    shape[-1]])

    return H

def calculate_weights_zero_forcing(H):
    """Calculates the weight matrix for the
       zero-forcing receiver.

    Args:
        H (complex (2, 2) ndarray): A matrix of
            channel estimations.

    Returns:
        W (complex (2, 2) ndarray): A matrix of
            weights.
    """
    W = np.linalg.inv(H)
    return W

def calculate_weights_mmse(tx_power, sigma, H):
    """Calculates the weight vectors for the
       MMSE receiver.

    Args:
        tx_power (float): The power of the
            transmitted signal.
        H (complex (2, 2) ndarray): A matrix of
            channel estimations.
        rx1 (complex 1D ndarray): The signal
            received at rx antenna 1.
        rx2 (complex 1D ndarray): The signal
            received at rx antenna 2.

    Returns:
        w1 (complex 1D ndarray): The weight
            vector for recovering rx1.
        w2 (complex 1D ndarray): The weight
            vector for recovering rx2.
    """
    # Calculate R.
    h1 = H[0, :]
    h2 = H[1, :]
```

```
    R = tx_power * h1 * np.transpose(np.
        conjugate(h1)) + tx_power * h2 \
          * np.transpose(np.conjugate(h2)) +
             sigma * np.eye(2)

    # Calculate the weight matrix
    W = tx_power * np.linalg.inv(R) * H

    return W

def recover_signals(rx1, rx2, W):
    """Estimates the sent signals using the
       weight matrix.

    Args:
         rx1 (complex 1D ndarray): The signal
            received at rx antenna 1.
         rx2 (complex 1D ndarray): The signal
            received at rx antenna 2.
         W (complex (2, 2) ndarray): A matrix of
             weights.

    Returns:
         x1_est (complex 1D ndarray): The
            estimated signal transmitted from
             tx antenna 1.
         x2_est (complex 1D ndarray): The
            estimated signal transmitted from
             tx antenna 2.
    """
    y1 = np.copy(rx1)
    y2 = np.copy(rx2)

    ys = np.vstack((y1, y2))
    x_est = np.matmul(W, ys)

    x1_est = np.squeeze(x_est[0, :])
    x2_est = np.squeeze(x_est[1, :])

    return x1_est, x2_est

def recover_signals_mimo(rx, W):
    """Use a weight matrix to recover MIMO
       signals.
    Args:
```

```
        rx (complex (4, num_samples) ndarray):
            The received MIMO signals with
              each row as the received signal at
                 one antenna.
        W (complex (4, 4) ndarray): A matrix of
              weights to apply to the signal.

    Returns:
        x_est (complex (4, num_samples) ndarray
            ): The recovered MIMO signals in
              the same format as rx.
    """
    return np.matmul(W, rx)

def preprocess_tx(tx, H):
    """Transform the transmitted data with
       known channel information.

    Args:
        tx (complex (4, n) ndarray): The n
            sample long signals to transmit.
        H (complex (4, 4) ndarray): A matrix of
              channel estimations.

    Returns:
        U (complex (4, 4) ndarray): The unitary
              matrix U resulting from
              performing SVD on the channel
                 estimations.
        S (complex (4, 4) ndarray): The
              diagonal matrix S representing the
              singular values of the SVD.
        tx_transform: The signal to transmit
            that's been transformed using CSI.
    """
    U, S, Vh = np.linalg.svd(H)
    tx_transform = np.matmul(np.transpose(np.
       conjugate(Vh)), tx)
    return U, S, tx_transform

def recover_signals_csi(rx, U):
    """Recover signals transformed with known
       CSI.

    Args:
```

```
        rx (complex (4, n) ndarray: The n
            sample long received signals.
        U (complex (4, 4) ndarray: The unitary
            matrix U from the SVD of the channel
            estimates.

    Returns:
        s_est: The recovered transmitted
            symbols.
    """
    s_est = np.matmul(np.transpose(np.conjugate
        (U)), rx)
    return s_est
```