EE4140: Simulation Assignment - I

Ayush Mukund Jamdar EE20B018

November 2, 2023

The assignment aims to

- demonstrate the action of pulse shaping filters,
- analyze the theoretical probability of error and the bounds studied by verifying against simulation, and
- simulate a distorting (ISI) channel modelled as a an L-tap channel to study the performance of the Viterbi Algorithm, a Maximum Likelihood Sequence Estimation (MLSE) approach.

1 Question

We work with a raised cosine pulse shaped transmission.

$$g(t) = sinc\left(\frac{\pi t}{T}\right) \cdot \left(\frac{cos\left(\frac{\pi \beta t}{T}\right)}{1 - \left(\frac{2\beta t}{T}\right)^2}\right)$$

Here β is the excess bandwidth and T is the signal period. We sample it at $T_s = T/J$ and truncate it to LT on each side. The $I(k)$ signal is zero interleaved with $J-1$ zeros (an important step) to ensure the time stamp match of the two T and T/J sampling intervals.

In this question, the signal is BPSK. We use 16 bits. Since there is no direct algorithm involved in this question unlike the Viterbi in Q3, I am omitting code from the report. However, a well commented Jupyter Notebook should be available with the report.

1.1 $J = 4$, $L = 2$, $\beta = 0$

The plot for this part is Figure 1. Comments -

- High L results in a lower bandwidth.
- It is clear that for small L, we use a shorter sinc pulse which means lesser distortion. This increases bandwidth in the frequency domain.

1.2 Different Excess Bandwidth Factors

Here the plots are for $J = 8$, $L = 4$ and $\beta = [0, 0.5, 1]$. Comments -

- Higher the excess bandwidth, faster the decay in the time domain.
- It can be observed in the plot that the green curve with the highest β shows less ISI effect by staying within -1 and 1.

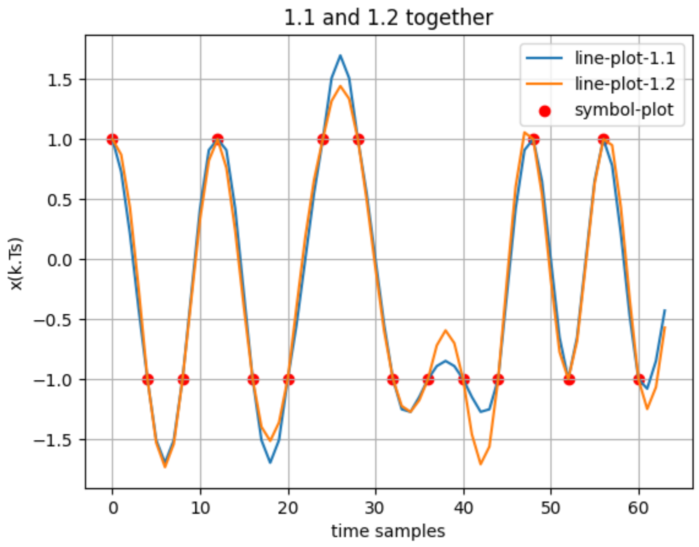

Figure 1: Q 1.1 and 1.2; The red dots indicate the symbols transmitted and the continuous curve is after pulse shaping with the RC. The blue curve is for $L = 2$ and the orange for $L = 4$

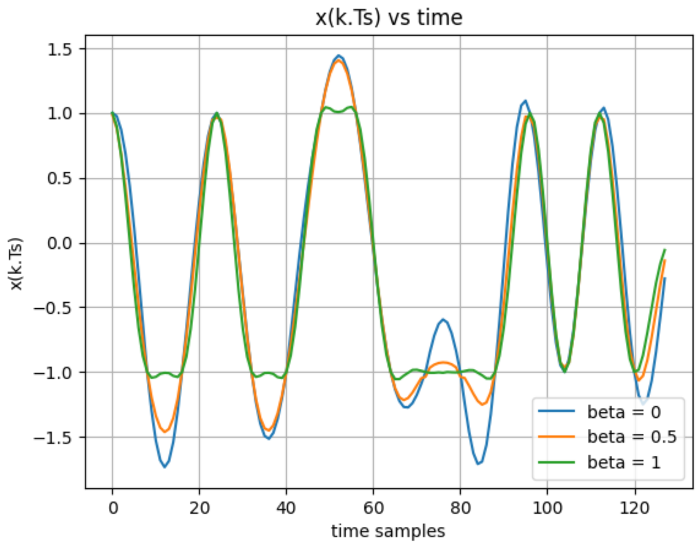

Figure 2: For Q1.3, varying excess bandwidth.

1.3 Periodogram

We need to plot the Power Spectral Density of the pulse shape, not by a direct analysis, but rather by a Monte-Carlo simulation. We use $R = 100$ runs of an algorithm that does the following -

1. Take a bit sequence of 1024, after pulse shaping, there will be 8192 samples.
2. Take an 8192 point FFT and square the magnitude response.
3. Average it over R.
4. The same procedure is repeated for a rectangular pulse shape too.

Both estimated PSDs, for the RC pulse and the rectangular pulse are plotted in the following figure. As expected, the rectangular pulse transforms to a *sinc* function with long tails on either side. The RC pulse however, has a limited bandwidth with no endless tails.

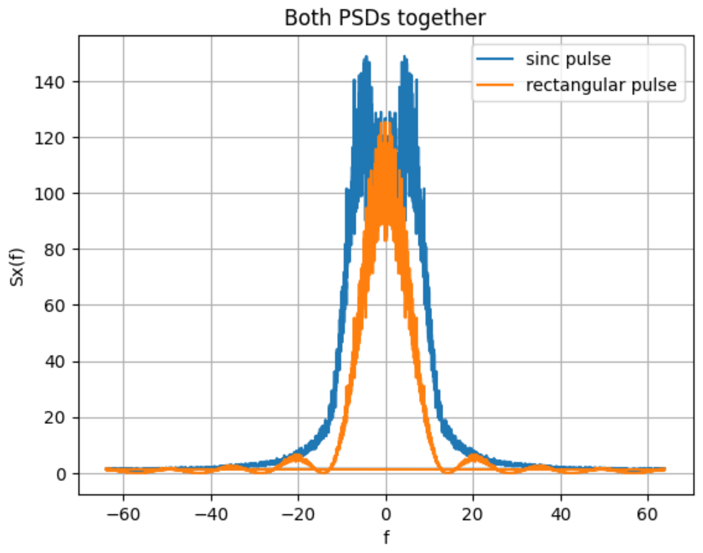

Figure 3: Periodogram for 1.4; Note that the signal magnitude was scaled with an intent to make the difference clear when plotted together.

2 Question

Here, the goal is the compare the theoretical average probability of symbol error P_e and the computer simuated Symbol Error Rate (SER) for BPSK, QPSK and 16-QAM. It is given that all constellations have unit energy per bit. In the analysis that follows, $2d$ is the minimum distance between two constellation points and not d.

2.1 P_e

This is the theoretical error probability. Following are the equations -

$$P_e^{2PAM} = q = \frac{1}{2} \cdot erfc\left(\frac{d}{\sigma\sqrt{2}}\right)$$

$$P_e^{4QAM} = 1 - \left(1 - \frac{1}{2} \cdot erfc\left(\frac{d}{\sigma\sqrt{(2)}}\right)\right)^2$$

$$P_e^{16QAM} = 1 - \frac{1}{16}\left(4(1-q)^2 + 8(1-q)(1-2q) + 4(1-2q)\right)^2$$

Plotted against SNR that is varied in steps of 2dB from 0dB to 14 dB in Figure 4.

2.2 Theoretical Probability with Bounds

Here we plot the bounds along with the actual theoretical P_e for 4-QAM. In the plot legend, (2.2) indicates the section index and the last integer indicates the following -

1. Union bound using all pairwise symbol errors.
2. Union bound using only nearest neighbours.
3. Replacing $erfc()$ by the Chernoff bound.
4. Accurate P_e from the previous section.
5. SER (See next section.)

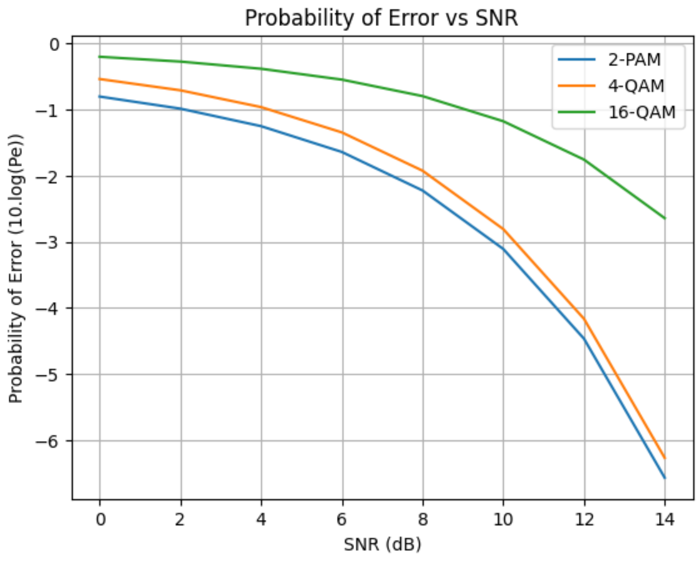

Figure 4: Probability of error: Theoretical

Although the latter two bounds merge with the accurate numbers for high SNR, the difference is clear at zero SNR where Chernoff bound is the highest, followed by the full union bound, then the nearest neighbours and finally the accurate ($3 > 1 > 2 > 4$).

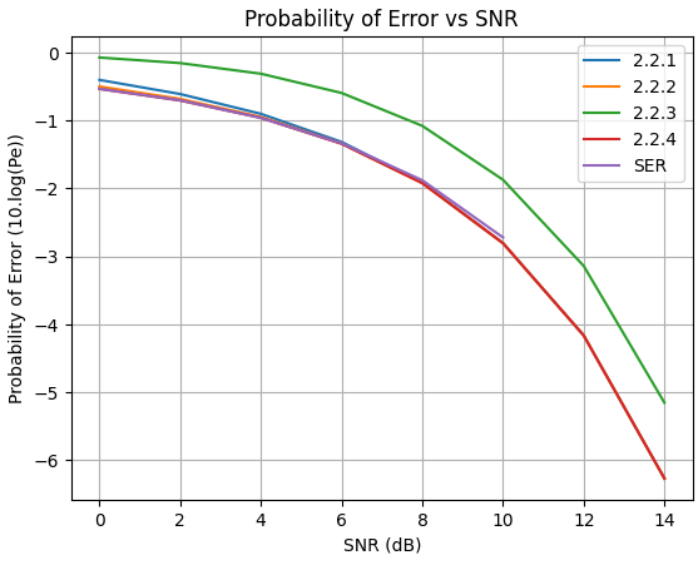

Figure 5: Error Probability with Bounds. Note that SER disappears after 10dB because the computed error goes to zero.

2.3 Simulated Error Probability

Explanation of approach - we use a large number of symbols of 4-QAM to transmit, add AWGN noise, perform the ML decoding at the receiving end and finally, calculate how many went wrong.

We begin by setting up the problem.

```python
        N_sim = 10000
        I_k_real = np.random.choice([-d_4qam, d_4qam], size=(N_sim))
        I_k_imag = np.random.choice([-d_4qam, d_4qam], size=(N_sim))
        I_k = I_k_real + 1j*I_k_imag
```

Next, we have the simulator function that operates for both 4-QAM and 16-QAM (for (2.5)). The algorithm takes the following steps -

1. Generate and add noise.
2. Map received $r(k)$ to nearest constellation symbols (decoding). See the `if-else` block for how this is done.
3. Compare with the sent $I(k)$ and calculate symbol error rate.

```python
        # 2.3
        N_sim = 10000
        I_k_real = np.random.choice([-d_4qam, d_4qam], size=(N_sim))
        I_k_imag = np.random.choice([-d_4qam, d_4qam], size=(N_sim))
        I_k = I_k_real + 1j*I_k_imag

        def SER_simulator(I_k, sigma_range, N_simulations, type = '4QAM'):
            symbol_error_rates = []

            if type == '4QAM':
                for sigma in sigma_range:
                    noise_real = np.random.normal(0, sigma, size=(N_simulations))
                    noise_imag = np.random.normal(0, sigma, size=(N_simulations))
                    noise = noise_real + 1j*noise_imag
                    r_k = I_k + noise # received signal

                    # using the decision boundaries, get r_k to nearest symbol
                    r_k_nearest_syms = np.zeros(len(r_k))*(0.0 + 0.0j)
                    r_k_nearest_syms[np.where(r_k.real < 0)] = -d_4qam
                    r_k_nearest_syms[np.where(r_k.real >= 0)] = d_4qam
                    r_k_nearest_syms[np.where(r_k.imag < 0)] += -1j*d_4qam
                    r_k_nearest_syms[np.where(r_k.imag >= 0)] += 1j*d_4qam

                    # now find number of errored symbols
                    error_indices = (np.where(r_k_nearest_syms != I_k))[0]
                    error_count = len(error_indices)
                    symbol_error_rate = error_count/N_simulations
                    symbol_error_rates.append(symbol_error_rate)

            elif type == '16QAM':
                for sigma in sigma_range:
                    noise_real = np.random.normal(0, sigma, size=(N_simulations))
                    noise_imag = np.random.normal(0, sigma, size=(N_simulations))
                    noise = noise_real + 1j*noise_imag
                    r_k = I_k + noise

                    # using the decision boundaries, get r_k to nearest symbol
                    r_k_nearest_syms = np.zeros(len(r_k))*(0.0 + 0.0j)
                    r_k_nearest_syms[np.where(r_k.real < -2*d_16qam)] = -3*d_16qam
                    r_k_nearest_syms[np.where((r_k.real >= -2*d_16qam) & (r_k.real <
                    ↪  0))] = -d_16qam
                    r_k_nearest_syms[np.where((r_k.real >= 0) & (r_k.real < 2*d_16qam))]
                    ↪  = d_16qam
                    r_k_nearest_syms[np.where(r_k.real >= 2*d_16qam)] = 3*d_16qam
                    r_k_nearest_syms[np.where(r_k.imag < -2*d_16qam)] += -3j*d_16qam
                    r_k_nearest_syms[np.where((r_k.imag >= -2*d_16qam) & (r_k.imag <
                    ↪  0))] += -1j*d_16qam
                    r_k_nearest_syms[np.where((r_k.imag >= 0) & (r_k.imag < 2*d_16qam))]
                    ↪  += 1j*d_16qam
                    r_k_nearest_syms[np.where(r_k.imag >= 2*d_16qam)] += 3j*d_16qam

                    # now find number of errored symbols
                    error_indices = np.where(r_k_nearest_syms != I_k)[0]
                    error_count = len(error_indices)
                    symbol_error_rate = error_count/N_simulations
                    symbol_error_rates.append(symbol_error_rate)

            return symbol_error_rates
```

The approach is now complete. The SER plot is included in Figure 5. Note that SER disappears after 10dB because the computed error goes to zero.

2.4 SER for 16-QAM

The symbol mapping for this part is similar to the previous section, except a few more conditions, and has been explained there. Here, we plot the SER for this constellation along with nearest neighbours P_e and the accurate P_e.

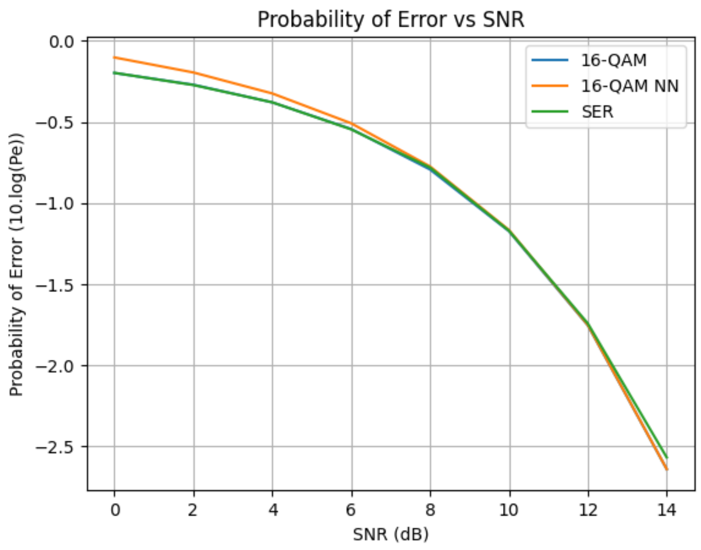

Figure 6: Simulated Error Rate (SER) for 16-QAM. The blue line is for the accurate P_e, NN is Nearest Neighbours and SER for this section.

3 Question: The Viterbi Algorithm

Here we implement from scratch the Viterbi Algorithm to recover a sequence of information that was transmitted through an ISI channel and was corrupted by AWGN noise. We also perform a few experiments with the algorithm parameters and analyze the results.

3.1 The Problem

We have a 4-ary PAM alphabet. The channel is modeled by an L-tap function which is in this case -

$$F(z) = \left(0.8 - z^{-1} + 0.6z^{-2}\right)/\sqrt{2}$$

The received sequence $r(k)$ is defined by the measurement model as:

$$r(k) = \sum_{l=0}^{L-1} f_l I(k-l) + v(k)$$

where, $v(k)$ is noise and the scalars f_l are obtained from the filter coefficients. Note that average symbol energy is given to be **1**. This implies that d (which is half the d_{min}) is equal to $1/\sqrt{5}$. Energy per bit ϵ_{bit} is 0.5 because there are four symbols in 4-PAM.

The goal is to construct the Viterbi Algorithm that will estimate this sequence. The algorithm has been implemented and explained in the next section.

3.2 The Algorithm

This section describes the algorithm, which is not required directly for the assignment. However, this section demonstrates my understanding of the process and is hence, included.

We begin by setting up the problem constants. SNR is varied from $0dB$ to $16dB$. We send $N = 100,002$ random symbols of 4-PAM. $L = 3$ and $M = 4$.

```
    e_bit = 0.5
    d = 1/np.sqrt(5)
    N_sym = 100002
    channel_f = [0.8, -1, 0.6] / np.sqrt(2)

    I_k = np.random.choice([-3*d, -d, d, 3*d], size=N_sym-2)
    I_k = np.append(I_k, np.array((3*d, 3*d))) # these will be used for decoding
    snr_range = np.arange(0, 18, 2)
    sigma_range = np.sqrt(e_bit*(10**(-snr_range/10)))
```

Next, the following function calculates $r(k)$ given the channel constants, $I(k)$ and noise $V(k)$.

```
    def get_r(I_k, V_k, channel_f):
        '''
        A general L-tap channel is described here

        input: symbol sequence, noise to be added and channel coefficients
        output: r_k received (modelled) sequence
        '''
        L = len(channel_f)
        I_k_padded = np.concatenate((np.zeros(len(channel_f)-1), I_k))
        r_k = np.convolve(I_k_padded, channel_f, 'same')[np.ceil((L-1)/2).astype(int) :
        ↪  -np.floor((L-1)/2).astype(int)]

        r_k += V_k
        return r_k
```

A few helper functions here and there. The use of these will be clear in the main algorithm function.

```
ef tm_helper(nodes_sliced, channel_f):
        r_hat_k = np.zeros(nodes_sliced.shape[0])
        for i in range(nodes_sliced.shape[1]):
            r_hat_k += nodes_sliced[:, i] * channel_f[i]

        r_hat_k = r_hat_k.reshape(-1, 1)
        return r_hat_k

    def convolve_with_channel(sequence, channel_f):
        sequence = sequence[::-1]
        convolved_seq = np.convolve(sequence, channel_f, "valid")[0]
        return convolved_seq

    def get_best_symbol_node_indices(transition_metric, L):
        '''
        Written for 4-PAM and 3-tap channel
        returns best node indices (0 to M^(L-1))
        dim = M^(L-1) x 1 where each element signifies the best previous node index for
        ↪  that node
        '''
        n_nodes = transition_metric.shape[0]
        M = transition_metric.shape[1]
        best_symbol_indices = []
        for i in range(M ** (L-2)):
            paths_at_nodes = np.array([transition_metric[i + (M**(L-2))*j] for j in
            ↪  range(M)])
            best_path_to_nodes = (np.argmin(paths_at_nodes, axis=0).reshape((M, 1))) *
            ↪  (M**(L-2)) + i
            best_symbol_indices.append(best_path_to_nodes)


        best_symbol_indices = np.array(best_symbol_indices).reshape((n_nodes, 1))
        return best_symbol_indices
```

Finally, the main algorithm, note that the algorithm is **generalized** to work for any legitimate M and L. We start by defining the symbols and nodes. The `nodes` are defined in reverse such that the first symbol to arrive will be the last in that node's state symbols. Thanks to the algorithm we only need to maintain M^{L-1} nodes as a Trellis (not not an ever-diverging Tree). We compute `r_hat` that is the noiseless estimate of r at each step for each node.

Next, we compute and initialize the `cumulative_metric` for each of the nodes. The `survivor sequences` corresponding to each node are initialized too. Once these steps are completed, we have arrived at the trellis, finishing the past $L-1$ stages that went into expanding nodes.

Running a loop for each symbol from here, we perform the following steps in order. Note that `delta` is the trace-back length.

- Compute the transition metric for each possible transition.
- Use the sum of the transition metric and cumulative metric to find which path was the best to get to a node.
- Once the best path is known, store the survivor sequence and metrics to the next node.
- If it is past `delta` iterations, start decoding symbols using the best cumulative metric so far.
- The last `delta` symbols are decoded using the fact that the last two symbols were known, thus converging the trellis into a known state.

Here is the complete algorithm -

```
    def viterbi(r_k, d, M, channel_f, sigma, N_sym, delta=30):
        '''
        Here the MLSE is performed using the Viterbi Algorithm
        The function is written for M-PAM with an L-tap channel

        input: d is half of d_min, r_k is the received sequence
        output: the MLSE sequence for r_k

        delta must be > 2
        '''

        symbols_half = np.array([d+2*d*j for j in range(M//2)])
        symbols = np.concatenate((symbols_half[::-1] * (-1), symbols_half))

```

```python
        L = len(channel_f)
        n_nodes = M ** (L-1) # number of nodes in Trellis

        nodes = np.fliplr(np.array(list(product(symbols, repeat=L-1))))

        r_hat = np.array([[convolve_with_channel(np.concatenate((np.array([s]), node)),
        ↪  channel_f) for s in symbols] for node in nodes])

        # Now, start the algorithm
        # we need to maintain a survivor seq and cumulative metric
        cumulative_metric = np.zeros((n_nodes, 1))

        for j in range(L-1):
            # loop until the trellis is ready
            # here we compute transition metrics and update the cumulative metric vector
            # we need to do this
            # f0 * nodes[-1] for j = 1
            # f1 * nodes[-1] + f0 * nodes[-2] for j = 2
            # f2 * nodes[-1] + f1 * nodes[-2] + f0 * nodes[-3] for j = 3
            # and so on
            transition_metric = (tm_helper(nodes[:, -j-1:], channel_f) - r_k[j]) ** 2
            cumulative_metric += transition_metric
        # now initialize the survivor seqs matrix0
        survivor_seqs = np.fliplr(nodes)

        mlse_seq = np.zeros((N_sym, 1))

        for i in range(L-1, N_sym):
            if i % 10000 == 0:
                print("Iter: ", i)

            transition_metric = (r_hat - r_k[i]) ** 2
            tm_plus_cm = transition_metric + cumulative_metric
            # 16x4 best_symbol_node_indices = get_best_symbol_node_indices(tm_plus_cm,
            ↪  L) # this should be 16x1

            # now we need to select the best sequence
            survivor_seqs_new = []
            cumulative_metric_new = np.zeros((n_nodes, 1))

            # now add the best transition metric to the cumulative metric
            for j in range(n_nodes):
                new_seq = np.append(survivor_seqs[best_symbol_node_indices[j][0], :],
                ↪  symbols[j % M])
                survivor_seqs_new.append(new_seq)
                cumulative_metric_new[j] = (
                                        (cumulative_metric + transition_metric)
                                        [best_symbol_node_indices[j], j % M]
                                    )

            cumulative_metric = cumulative_metric_new

            survivor_seqs = np.array(survivor_seqs_new)

            if i >= delta:
                mlse_seq[i-delta] = survivor_seqs[np.argmin(cumulative_metric), 0]
                survivor_seqs = survivor_seqs[:, 1:]

            if i == N_sym-1:
                mlse_seq[i-delta+1:] = survivor_seqs[-1, :].reshape((delta, 1))
                # here we use the information that the last two symbols are 3d and 3d

        return mlse_seq.reshape((N_sym, ))
```

Now we run the algorithm for the deltas asked for. Note that the known bits are exclued from the error measurement.

```python
    deltas = [3, 6, 10, 20, 40]
    SER = {delta: [] for delta in deltas}

    for delta in deltas:
        for sigma in sigma_range:
            print("Delta = ", delta, "| Sigma = ", sigma)
            noise = np.random.normal(0, sigma, size=N_sym)
            r_k = get_r(I_k, noise, channel_f)
            mlse_sequence = viterbi(r_k, d, channel_f, sigma, 3, N_sym)
            error_indices = np.where(mlse_sequence[:-2] != I_k[:-2])[0]
            error_count = len(error_indices)
            error_rate = error_count / (N_sym-2)
            SER[delta].append(error_rate)
            print("Error Rate = ", error_rate)
```

3.3 Results

The entire result is summarized in the figure here.

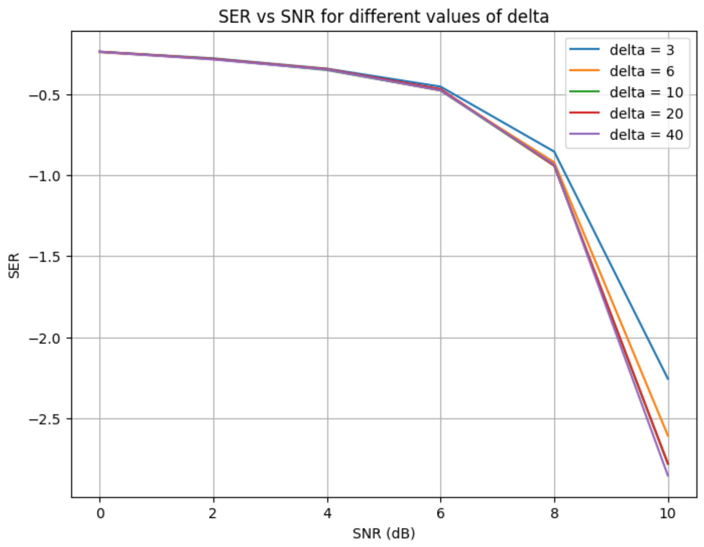

Figure 7: Simulated Error Rate (SER) against SNR for different trace-back lengths. Please note that the SER is plotted only upto 10 dB because the SER falls to zero for lower noise.

We can observe how the SER reduces with increase in SNR. It can also be noted that higher `delta` has lower SER towards the end.

4 Question 4

Here it is time to leverage the generalization of the algorithm and run it for 2-PAM and a 6-tap channel ($M = 2$ and $L = 4$).

$$f = [1, -0.95, 0.5, 0.15, 0.2, 0.1]/C$$

where $C = 1.49$ is the normalization factor.

Now we run the VA for this problem at $\delta = 30$ for 100,004 symbols.

Next, we truncate the channel model by only taking the first three taps (the weaker taps are ignored). This makes $L = 3$, but this is only in the VA construction. Here the measurements have contribution from all the 6-taps.

4.1 Code

First set the problem up. Note that symbol energy equals bit energy in 2-PAM. d is 1. `sigma` indicates standard deviation.

```python
e_bit = 1
d = 1
N_sym = 100004
channel_f = [1, -0.95, 0.5, 0.15, -0.2, -0.1]
L = len(channel_f)
channel_f /= np.linalg.norm(channel_f) # C

I_k = np.random.choice([-d, d], size=N_sym-(L-1))
I_k = np.append(I_k, np.array([d]*(L-1))) # these will be used for decoding
snr_range = np.arange(0, 20, 2)
sigma_range = np.sqrt(e_bit*(10**(-snr_range/10)))
```

Now the part for 6-taps.

```python
# 6-taps
delta = 30
M = 2
SER = []

# for delta in deltas:
for sigma in sigma_range:
    print("Delta = ", delta, "| Sigma = ", sigma)
    noise = np.random.normal(0, sigma, size=N_sym)
    r_k = get_r(I_k, noise, channel_f)
    mlse_sequence = viterbi(r_k, d, M, channel_f, sigma, N_sym, delta)
    error_indices = np.where(mlse_sequence[:-(L-1)] != I_k[:-(L-1)])[0]
    error_count = len(error_indices)
    error_rate = error_count / (N_sym-L)
    SER.append(error_rate)
    print("Error Rate = ", SER[-1])
```

Next, 3-taps in the VA mode. However, the received signal goes through all taps.

```python
channel_f_43 = channel_f[0:3]
L_43 = len(channel_f_43)

delta = 30
M = 2
SER_43 = []

# for delta in deltas[-1:]:
for sigma in sigma_range:
    print("Delta = ", delta, "| Sigma = ", sigma)
    noise = np.random.normal(0, sigma, size=N_sym)
    r_k = get_r(I_k, noise, channel_f) # measurements have all taps
    mlse_sequence = viterbi(r_k, d, M, channel_f_43, sigma, N_sym, delta)
    error_indices = np.where(mlse_sequence[:-(L_43-1)] != I_k[:-(L_43-1)])[0]
    error_count = len(error_indices)
    error_rate = error_count / (N_sym-L_43)
    SER_43.append(error_rate)
```

```
        print("Error Rate = ", error_rate)
```

4.2 Result

The same is plotted in Figure 8. We can observe that the error dramatically falls in both the cases but faster, when all 6 taps are used.

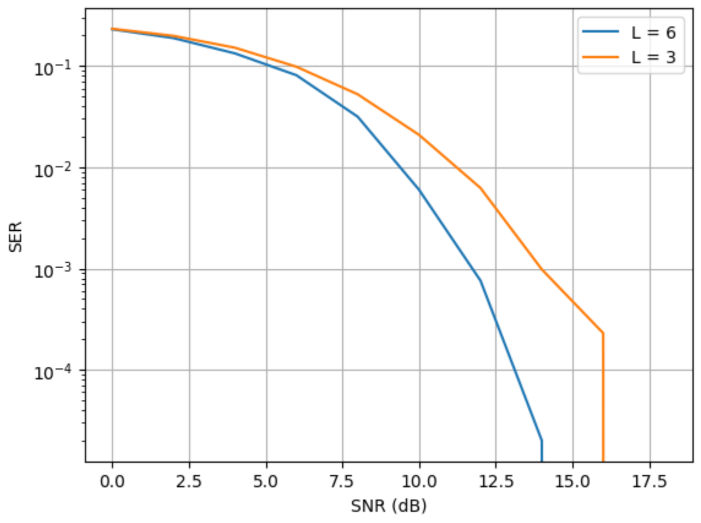

Figure 8: Simulated Error Rate (SER) against SNR for 6-tap and 3-tap algorithm.

5 Conclusion

The aims targeted at the beginning of this assignment are now seen to have been accomplished. We modelled pulse shaping filters, ran the nearest neighbours decoding for non-ISI signals and finally performed an MLSE in a complex ISI channel using the Viterbi Algorithm.